BUILD YOUR OWN ANGULARJS

Build Your Own AngularJS

Tero Parviainen

ISBN 978-952-93-3544-2

Contents

Introduction

This book is written for the working programmer, who either wants to learn AngularJS, or already knows AngularJS and wants to know what makes it tick.

AngularJS is not a small framework. It has a large surface area with many new concepts to grasp. Its codebase is also substantial, with 35K lines of JavaScript in it. While all of those new concepts and all of those lines of code give you powerful tools to build the apps you need, they also come with a learning curve.

I hate working with technologies I don't quite understand. Too often, it leads to code that just happens to work, not because you truly understand what it does, but because you went through a lot of trial and error to make it work. Code like that is difficult to change and debug. You can't reason your way through problems. You just poke at the code until it all seems to align.

Frameworks like AngularJS, powerful as they are, are prone to this kind of code. Do you understand how Angular does dependency injection? Do you know the mechanics of scope inheritance? What exactly happens during directive transclusion? When you don't know how these things work, as I didn't when I started working with Angular, you just have to go by what the documentation says. When that isn't enough, you try different things until you have what you need.

The thing is, while there's a lot of code in AngularJS, it's all just code. It's no different from the code in your applications. Most of it is well-factored, readable code. You can study it to learn how Angular does what it does. When you've done that, you're much better equipped to deal with the issues you face in your daily application development work. You'll know not only what features Angular provides to solve a particular problem, but also how those features work, how to get the most out of them, and where they fall short.

The purpose of this book is to help you demystify the inner workings of AngularJS. To take it apart and put it back together again, in order to truly understand how it works.

A true craftsman knows their tools well. So well that they could in fact make their own tools if needed. This book will help you get there with AngularJS.

How To Read This Book

During the course of the book we will be building an implementation of AngularJS. We'll start from the very beginning, and in each chapter extend the implementation with new capabilities.

While there are certain areas of functionality in Angular that are largely independent, most of the code you'll be writing builds on things implemented in previous chapters. That is why a sequential reading will help you get the most out of this book.

The format of the book is simple: Each feature is introduced by discussing what it is and why it's needed. We will then proceed to implement the feature following test-driven development practices: By writing failing tests and then writing the code to make them pass. As a result, we will produce not only the framework code, but also a test suite for it.

It is highly encouraged that you not only read the code, but also actually type it in and build your own Angular while reading the book. To really make sure you've grasped a concept, poke it from different directions: Write additional test cases. Try to intentionally break it in a few ways. Refactor the code to match your own style while keeping the tests passing.

If you're only interested in certain parts of the framework, feel free to skip to the chapters that interest you. While you may need to reference back occasionally, you should be able to poach the best bits of Angular to your own application or framework with little difficulty.

Source Code

The source code and test suite implemented in this book can be found on GitHub, at https://github.com/teropa/build-your-own-angularjs/.

To make following along easier, commits in the repository are ordered to match the order of events in the book. Each commit message references the corresponding section title in the book. Note that this means that during the production of the book, the history of the code repository may change as revisions are made.

There is also a Git tag for each chapter, pointing to the state of the codebase at the end of that chapter. You can download archives of the code corresponding to these tags from https://github.com/teropa/build-your-own-angularjs/releases.

Contributors

I would like to thank the following people or their valuable feedback and help during the writing of this book:

- Iftach Bar
- Xi Chen
- Wil Pannell
- Jesus Rodriguez

Early Access: Errata & Contributing

This book is under active development, and you are reading an early access version. The content is still incomplete, and is likely to contain bugs, typos, and other errors.

As an early access subscriber, you will receive updated versions of the book throughout its production process.

Your feedback is more than welcome. If you come across errors, or just feel like something could be improved, please file an issue to the book's Errata on GitHub: https://github.com/teropa/build-your-own-angularjs/issues. To make this process easier, there are links in the footer of each page that take you directly to the corresponding chapter's errata, and to the form with which you can file an issue.

Contact

Feel free to get in touch with me by sending an email to tero@teropa.info or tweeting at @teropa.

Version History

2014-05-18: Chapter 9: Modules And The Injector

Added Chapter 9 and fixed a number of errata.

2014-04-13: Chapter 7: Operator Expressions

Added Chapter 7. Also added coverage of literal collections in expressions to Chapters 5 and 6, and fixed some errata.

2014-03-29: Maintenance Release

Fixed a number of errata and introduced some features new to AngularJS.

2014-03-28: Chapter 6: Lookup And Function Call Expressions

Added Chapter 6.

2014-03-11: Part II Introduction and Chapter 5: Scalar Literal Expressions

Added Chapter 5 and fixed some minor errata.

2014-02-25: Maintenance Release

Fixed a number of errata.

2014-02-01: Chapter 4: Scope Events

Added Chapter 4 and fixed a number of errata.

2014-01-18: Chapter 3: Watching Collections

Added Chapter 3 and fixed a number of errata.

2014-01-07: Digest Optimization in Chapter 1

Described the new short-circuiting optimization for digest.

2014-01-06: Initial Early Access Release

First public release including the Introduction, the Setup chapter, and Chapters 1-2.

Project Setup

We are going to build a full-blown JavaScript framework. To make things much easier down the line, it's a good idea to spend some time setting up a project with a solid build process and automated testing. Fortunately there are excellent tools available for this purpose. We just need to pull them in and we'll be good to go.

In this warm-up chapter we'll set up a JavaScript library project using NPM and Grunt. We'll also enable static analysis with JSHint and unit testing with Jasmine.

Install The Required Tools

There are a couple of tools you need to install before we can get going. These tools are widely supported and available for Linux, Mac OS X, and Windows.

Instead of going through the installation processes of the tools here, I'll just point you to the relevant resources online:

- Node.js, the popular server-side JavaScript platform, is the underlying JavaScript runtime used for building and testing the project. Node.js also bundles NPM, the package manager we're going to use. Install Node and NPM following the official installation instructions.
- Grunt is the JavaScript build tool we'll be using. To obtain Grunt, install the `grunt-cli` package using NPM, as described on the Grunt website. This will make the `grunt` command available. You'll be using it a lot.

Before proceeding to create the project, make sure you have the `node`, `npm`, and `grunt` commands working in your terminal. Here's what they look like on my machine:

```
node -v
v0.10.21

npm -v
1.3.11

grunt -V
grunt-cli v0.1.9
grunt v0.4.1
```

Your output may be different depending on the versions installed. What's important is that the commands are there and that they work.

Create The Project Directories

Let's set up the basic directory structure for our library. We'll need directories are needed at this point: The project root directory, an `src` directory for sources, and a `test` directory for unit tests tests:

```
mkdir myangular
cd myangular
mkdir src
mkdir test
```

As the project grows we will extend this directory structure, but this is enough to get going.

Create package.json for NPM

In order to use NPM, we're going to need a file called `package.json`. This file is used to let NPM know some basic things about our project and, crucially, the external NPM packages it depends on.

Let's create a basic `package.json` in the project root directory with some basic metadata – the project name and version:

package.json

```
{
  "name": "my-own-angularjs",
  "version": "0.1.0"
}
```


Create Gruntfile.js for Grunt

The next thing we'll need is our Grunt build file. It is a plain JavaScript file containing the tasks and configurations needed for testing and packaging the project. You can think of it as the JavaScript equivalent of a `Makefile`, `Rakefile`, or `build.xml`.

Create `Gruntfile.js` with the following basic structure in the project root directory:

Gruntfile.js

```
module.exports = function(grunt) {

  grunt.initConfig({
  });

};
```

Everything in `Gruntfile.js` is wrapped in a function, which is assigned to the `module.exports` attribute. This is related to the Node.js module system, and has little practical significance to us. The interesting part is the call to `grunt.initConfig`. This is where we will pass our project configuration to Grunt.

We now have the directories and files necessary for a minimal JavaScript project in place.

"Hello, World!"

Before delving into static analysis and testing, let's add a bit of JavaScript code so that we have something to play with. The canonical "Hello, world!" will fit our purposes perfectly. Add the following function in a file called `hello.js` in the `src` directory:

src/hello.js

```
function sayHello() {
  return "Hello, world!";
}
```

Enable Static Analysis With JSHint

JSHint is a tool that reads in your JavaScript code and gives a report of any syntactic or structural problems within it. This process, called *linting*, is very useful to us since as library authors we don't want to be shipping code that may cause problems for other people.

JSHint integrates with Grunt using an NPM module called grunt-contrib-jshint. Let's install that module with the `npm` command:


```
npm install grunt-contrib-jshint --save-dev
```

When you run this command, a couple of things will happen: A directory called `node_modules` is created inside our project and the `grunt-contrib-jshint` module is downloaded into that directory.

Also, if you take a look at our `package.json` file, you'll see it has changed a bit: It now includes a `devDependencies` key and a nested `grunt-contrib-jshint` key. What this says is that `grunt-contrib-jshint` is a development-time dependency of our project.

We can now refer to `grunt-contrib-jshint` from our build file. Let's do that:

Gruntfile.js

```
module.exports = function(grunt) {

  grunt.initConfig({
    jshint: {
      all: ['src/**/*.js'],
      options: {
        globals: {
          _: false,
          $: false
        },
        browser: true,
        devel: true
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');

};
```

In `initConfig` we're configuring jshint to be run against all JavaScript files under the `src` directory. We're letting jshint know that we will be referencing two global variables defined outside of our own code, and that it should not raise errors when we do so. Those variables are `_` from Lo-Dash and `$` from jQuery. We're enabling the `browser` and `devel` JSHint environments, which will cause it to not raise errors when we refer to global variables commonly available in browsers, such as `setTimeout` and `console`.

We also load the `jshint` task itself into our build by invoking the `grunt.loadNpmTasks` function.

We're now all set to run JSHint:

```
grunt jshint
Running "jshint:all" (jshint) task
>> 1 file lint free.

Done, without errors.
```


Looks like our simple "Hello, world!" program is lint free!

Enable Unit Testing With Jasmine And Testem

Unit testing will be absolutely central to our development process. That means we also need a good test framework. We're going to use one called Jasmine, because it has a nice and simple API and it integrates well with Grunt.

For actually running the tests, we'll use a test runner called Testem. It will provide us with a nice terminal UI using which we can see the status of our tests.

Let's will install the Grunt integration library for Testem. That will bring in all the dependencies we need, including the Jasmine framework:

```
npm install grunt-contrib-testem --save-dev
```

Next, load and configure Testem and Jasmine in `Gruntfile.js`:

Gruntfile.js

```
module.exports = function(grunt) {

  grunt.initConfig({
    jshint: {
      all: ['src/**/*.js', 'test/**/*.js'],
      options: {
        globals: {
          _: false,
          $: false,
          jasmine: false,
          describe: false,
          it: false,
          expect: false,
          beforeEach: false
        },
        browser: true,
        devel: true
      }
    },
    testem: {
      unit: {
        options: {
          framework: 'jasmine2',
          launch_in_dev: ['PhantomJS'],
          before_tests: 'grunt jshint',
          serve_files: [
            'src/**/*.js',
            'test/**/*.js'
          ],
```



```
          watch_files: [
            'src/**/*.js',
            'test/**/*.js'
          ],
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-testem');

};
```

In `initConfig` we're declaring a Testem task called `unit`, which will run tests for all JavaScript files under `src`. The tests themselves will be located in JavaScript files under the `test` directory. While running, Testem will continuously watch these files for changes and rerun the test suite automatically when there are changes.

With the `launch_in_dev` option we are telling Testem that we want it to launch a PhantomJS browser for running the tests. PhantomJS is a headless Webkit browser that lets us run tests without necessarily having to manage external browsers. Install it now so that it'll be available to Testem:

```
npm install -g phantomjs
```

We also define a `before_tests` hook which will cause JSHint to be run before each test suite execution.

In test code we will be referring to a bunch of global variables defined by Jasmine. We add these variables to JSHint's `globals` object so that it will let us do that.

Note that we're also configuring JSHint to apply linting to our test code as well as our production code.

We're now ready to run tests, so let's create one. Add a file called `hello_spec.js` in the `test` directory, with the following contents:

test/hello_spec.js

```
describe("Hello", function() {

  it("says hello", function() {
    expect(sayHello()).toBe("Hello, world!");
  });

});
```

If you've ever used something like Ruby's rSpec, the format should look familiar: There's a top-level `describe` block, which acts as a grouping for a number of tests. The test cases themselves are defined in `it` blocks, each one defining a name and a test function.

Let's run the test by invoking the `testem` task with Grunt:

```
grunt testem:run:unit
```

This brings up the Testem UI and launch the tests in an invisible PhantomJS browser. It will also show a URL which you can open in any number of additional web browsers you have installed. All connected web browsers will server as Testem clients, and the test suite will be automatically run in each open browser.

Try editing `src/hello.js` and `test/hello_spec.js` while the Testem runner is open to see that the changes are picked up and the test suite is re-run.

As you work your way through this book, it's recommended to keep the Testem UI permanently running in a terminal window. This way you won't have to keep running the tests manually all the time, and you'll be sure to notice when the code breaks.

Our single test is passing and we can move forward.

Include Lo-Dash And jQuery

Angular itself does not require any third-party libraries (though it does use jQuery if it's available). However, in our case it makes sense to delegate some low-level details to existing libraries so that we can concentrate on what makes Angular Angular.

There are two categories of low-level operations that we can delegate to existing libraries:

- Array and object manipulation, such as equality checking and cloning, will be delegated to the Lo-Dash library.
- DOM querying and manipulation will be delegated to jQuery.

Both of these libraries are available as NPM packages. Let's go ahead and install them:

```
npm install lodash --save
```

```
npm install jquery --save
```

We're installing these libraries in the same way we previously installed the Grunt plugins for JSHint and Jasmine. The only difference is that in this case we specify the `--save` flag instead of `--save-dev`. That means these packages are *runtime* dependencies and not just development dependencies.

Let's check that we're actually able to use these libraries by tweaking our "Hello, world!" script a bit:

src/hello.js

```
function sayHello(to) {
  return _.template("Hello, <%= name %>!")({name: to});
}
```

The function now takes the receiver of the greeting as an argument, and constructs the resulting string using the Lo-Dash template function.

Let's also make the corresponding change to the unit test:

test/hello_spec.js

```
describe("Hello", function() {

  it("says hello to receiver", function() {
    expect(sayHello('Jane')).toBe("Hello, Jane!");
  });

});
```

If we now try to run the unit test (in the Testem runner), it doesn't quite seem to work. It cannot find a reference to Lo-Dash.

The problem is that although we've got Lo-Dash installed under `node_packages`, we're not loading it into our tests, so the `_` variable isn't defined. We can fix this by adding Lo-Dash as to the files served for Testem in `Gruntfile.js`. While we're at it, let's also include jQuery because we'll need it later:

Gruntfile.js

```
    testem: {
      unit: {
        options: {
          framework: 'jasmine2',
          launch_in_dev: ['PhantomJS'],
          before_tests: 'grunt jshint',
          serve_files: [
            'node_modules/lodash/lodash.js',
            'node_modules/jquery/dist/jquery.js',
            'src/**/*.js',
            'test/**/*.js'
          ],
          watch_files: [
            'src/**/*.js',
            'test/**/*.js'
          ]
        }
      }
    }
```

If you try launching the testem runner again, the test suite should now pass.

Define The Default Grunt Task

We will be running Testem all the time, so it makes sense to define it as the default task of our Grunt build. We can do that by adding the following at the end of `Gruntfile.js`, just before the closing curly brace:

Gruntfile.js

```
grunt.registerTask('default', ['testem:run:unit']);
```

This defines a `default` task that runs `testem:run:unit`. You can invoke it just by running `grunt` without any arguments:

```
grunt
```

Summary

In this chapter you've set up the project that will contain your very own implementation of Angular. You have:

- Installed Node.js, NPM, and Grunt.
- Configured a Grunt project that includes the JSHint linter and Jasmine unit testing support.

You are now all set to begin making your own AngularJS. You can remove `src/hello.js` and `test/hello_spec.js` so that you'll have a clean slate for the next section, which is all about Angular Scopes.

Part I

Scopes

We will begin our implementation of AngularJS with one of its central building blocks: Scopes. Scopes are used for many different purposes:

- Sharing data between controllers and views
- Sharing data between different parts of the application
- Broadcasting and listening for events
- Watching for changes in data

Of these several use cases, the last one is arguably the most interesting one. Angular scopes implement a *dirty-checking* mechanism, using which you can get notified when a piece of data on a scope changes. It can be used as-is, but it is also the secret sauce of *data binding*, one of Angular's primary selling points.

In this first part of the book you will implement Angular scopes. We will cover four main areas of functionality:

1. The digest cycle and dirty-checking itself, including `$watch`, `$digest`, and `$apply`.
2. Scope inheritance – the mechanism that makes it possible to create scope hierarchies for sharing data and events.
3. Efficient dirty-checking for collections – arrays and objects.
4. The event system – `$on`, `$emit`, and `$broadcast`.

Chapter 1

Scopes And Digest

Angular scopes are plain old JavaScript objects, on which you can attach properties just like you would on any other object. However, they also have some added capabilities for observing changes in data structures. These observation capabilities are implemented using *dirty-checking* and executed in a *digest cycle*. That is what we will implement in this chapter.

Scope Objects

Scopes are created by using the `new` operator on a `Scope` constructor. The result is a plain old JavaScript object. Let's make our very first test case for this basic behavior.

Create a test file for scopes in `test/scope_spec.js` and add the following test case to it:

test/scope_spec.js

```
/* jshint globalstrict: true */
/* global Scope: false */
'use strict';

describe("Scope", function() {

  it("can be constructed and used as an object", function() {
    var scope = new Scope();
    scope.aProperty = 1;

    expect(scope.aProperty).toBe(1);
  });

});
```

On the top of the file we enable ES5 strict mode, and also let JSHint know it's OK to refer to a global variable called `Scope` in the file.

This test just creates a `Scope`, assigns an arbitrary property on it and checks that it was indeed assigned.

> It may concern you that we are using `Scope` as a global function. That's definitely not good JavaScript style! We will fix this issue once we implement dependency injection later in the book.

> If you have Testem running in a terminal, you will see it fail after you've added this test case, because we haven't implemented `Scope` yet. This is exactly what we want, since an important step in test-driven development is seeing the test fail first. Throughout the book I'll assume the test suite is being continuously executed, and will not explicitly mention when tests should be run.

We can make this test pass easily enough: Create `src/scope.js` and set the contents as:

src/scope.js

```
/* jshint globalstrict: true */
'use strict';

function Scope() {
}
```

In the test case we're assigning a property (`aProperty`) on the scope. This is exactly how properties on the Scope work. They are plain JavaScript properties and there's nothing special about them. There are no special setters you need to call, nor restrictions on what values you assign. Where the magic happens instead is in two very special functions: `$watch` and `$digest`. Let's turn our attention to them.

Watching Object Properties: $watch And $digest

`$watch` and `$digest` are two sides of the same coin. Together they form the core of what the digest cycle is all about: Reacting to changes in data.

With `$watch` you can attach a *watcher* to a scope. A watcher is something that is notified when a change occurs in the scope. You create a watcher by providing two functions to `$watch`:

- A *watch function*, which specifies the piece of data you're interested in.
- A *listener function* which will be called whenever that data changes.

> As an Angular user, you actually usually specify a watch *expression* instead of a watch function. A watch expression is a string, like `"user.firstName"`, that you specify in a data binding, a directive attribute, or in JavaScript code. It is parsed and compiled into a watch function by Angular internally. We will implement this in Part 2 of the book. Until then we'll use the slightly lower-level approach of providing watch functions directly.

The other side of the coin is the `$digest` function. It iterates over all the watchers that have been attached on the scope, and runs their watch and listener functions accordingly.

To flesh out these building blocks, let's define a test case which asserts that you can register a watcher using `$watch`, and that the watcher's listener function is invoked when someone calls `$digest`.

To make things a bit easier to manage, add the test to a nested `describe` block in `scope_spec.js`. Also create a `beforeEach` function that initializes the scope, so that we won't have to repeat it for each test:

test/scope_spec.js

```
describe("Scope", function() {

  it("can be constructed and used as an object", function() {
    var scope = new Scope();
    scope.aProperty = 1;

    expect(scope.aProperty).toBe(1);
  });

  describe("digest", function() {

    var scope;

    beforeEach(function() {
      scope = new Scope();
    });

    it("calls the listener function of a watch on first $digest", function() {
      var watchFn    = function() { return 'wat'; };
      var listenerFn = jasmine.createSpy();
      scope.$watch(watchFn, listenerFn);

      scope.$digest();

      expect(listenerFn).toHaveBeenCalled();
    });

  });

});
```


In the test case we invoke `$watch` to register a watcher on the scope. We're not interested in the watch function just yet, so we just provide one that returns a constant value. As the listener function, we provide a Jasmine Spy. We then call `$digest` and check that the listener was indeed called.

> A spy is Jasmine terminology for a kind of mock function. It makes it convenient for us to answer questions like "Was this function called?" and "What arguments was it called with?"

There are a few things we need to do to make this test case pass. First of all, the Scope needs to have some place to store all the watchers that have been registered. Let's add an array for them in the `Scope` constructor:

src/scope.js

```
function Scope() {
  this.$$watchers = [];
}
```

> The double-dollar prefix `$$` signifies that this variable should be considered private to the Angular framework, and should not be called from application code.

Now we can define the `$watch` function. It'll take the two functions as arguments, and store them in the `$$watchers` array. We want every Scope object to have this function, so let's add it to the prototype of `Scope`:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn
  };
  this.$$watchers.push(watcher);
};
```

Finally there is the `$digest` function. For now, let's define a very simple version of it, which just iterates over all registered watchers and calls their listener functions:

src/scope.js

```
Scope.prototype.$digest = function() {
  _.forEach(this.$$watchers, function(watcher) {
    watcher.listenerFn();
  });
};
```

The test pass but this version of `$digest` isn't very useful yet. What we really want is to check if the values specified by the watch functions have actually changed, and *only then* call the respective listener functions. This is called *dirty-checking*.

Checking for Dirty Values

As described above, the watch function of a watcher should return the piece of data whose changes we are interested in. Usually that piece of data is something that exists on the scope. To make accessing the scope from the watch function more convenient, we want to call it with the current scope as an argument. A watch function that's interested in a `firstName` attribute on the scope may then do something like this:

```
function(scope) {
  return scope.firstName;
}
```

This is the general form that watch functions usually take: Pluck some value from the scope and return it.

Let's add a test case for checking that the scope is indeed provided as an argument to the watch function:

test/scope_spec.js

```
it("calls the watch function with the scope as the argument", function() {
  var watchFn    = jasmine.createSpy();
  var listenerFn = function() { };
  scope.$watch(watchFn, listenerFn);

  scope.$digest();

  expect(watchFn).toHaveBeenCalledWith(scope);
});
```

This time we create a Spy for the watch function and use it to check the watch invocation.

The simplest way to make this test pass is to modify `$digest` to do something like this:

src/scope.js

```
Scope.prototype.$digest = function() {
  var self = this;
  _.forEach(this.$$watchers, function(watcher) {
    watcher.watchFn(self);
    watcher.listenerFn();
  });
};
```

Of course, this is not quite what we're after. The `$digest` function's job is really to call the watch function and compare its return value to whatever the same function returned last time. If the values differ, the watcher is *dirty* and its listener function should be called. Let's go ahead and add a test case for that:

test/scope_spec.js


```
it("calls the listener function when the watched value changes", function() {
  scope.someValue = 'a';
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.someValue; },
    function(newValue, oldValue, scope) { scope.counter++; }
  );

  expect(scope.counter).toBe(0);

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.someValue = 'b';
  expect(scope.counter).toBe(1);

  scope.$digest();
  expect(scope.counter).toBe(2);

});
```

We first plop two attributes on the scope: A string and a number. We then attach a watcher that watches the string and increments the number when the string changes. The expectation is that the counter is incremented once during the first `$digest`, and then once every subsequent `$digest` *if* the value has changed.

Notice that we also specify the contract of the listener function: Just like the watch function, it takes the scope as an argument. It's also given the new and old values of the watcher. This makes it easier for application developers to check what exactly has changed.

To make this work, `$digest` has to remember what the last value of each watch function was. Since we already have an object for each watcher, we can conveniently store the last value there. Here's a new definition of `$digest` that checks for value changes for each watch function:

src/scope.js

```
Scope.prototype.$digest = function() {
  var self = this;
  var newValue, oldValue;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      watcher.last = newValue;
```



```
      watcher.listenerFn(newValue, oldValue, self);
    }
  });
};
```

For each watcher, we compare the return value of the watch function to what we've previously stored in the `last` attribute. If the values differ, we call the listener function, passing it both the new and old values, as well as the scope object itself. Finally, we set the `last` attribute of the watcher to the new return value, so we'll be able to compare to that next time.

We've now implemented the essence of Angular scopes: Attaching watches and running them in a digest.

We can also already see a couple of important performance characteristics that Angular scopes have:

- Attaching data to a scope does not by itself have an impact on performance. If no watcher is watching a property, it doesn't matter if it's on the scope or not. Angular does not iterate over the properties of a scope. It iterates over the watches.
- Every watch function is called during every `$digest`. For this reason, it's a good idea to pay attention to the number of watches you have, as well as the performance of each individual watch function or expression.

Initializing Watch Values

Comparing a watch function's return value to the previous one stored in `last` works fine most of the time, but what does it do on the first time a watch is executed? Since we haven't set `last` at that point, it's going to be `undefined`. That doesn't quite work when the first *legitimate* value of the watch is also `undefined`:

test/scope_spec.js

```
it("calls listener when watch value is first undefined", function() {
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.someValue; },
    function(newValue, oldValue, scope) { scope.counter++; }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```


We should be calling the listener function here too. What we need is to initialize the `last` attribute to something we can guarantee to be unique, so that it's different from anything a watch function might return.

A *function* fits this purpose well, since JavaScript functions are so-called *reference values* - they are not considered equal to anything but themselves. Let's introduce a function value on the top level of `scope.js`:

src/scope.js

```
function initWatchVal() { }
```

Now we can stick this function into the `last` attribute of new watches:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn,
    last: initWatchVal
  };
  this.$$watchers.push(watcher);
};
```

This way new watches will *always* have their listener functions invoked, whatever their watch functions might return.

What also happens though is the `initWatchVal` gets handed to listeners as the old value of the watch. We'd rather not leak that function outside of `scope.js`. For new watches, we should instead provide the new value as the old value:

test/scope_spec.js

```
it("calls listener with new value as old value the first time", function() {
  scope.someValue = 123;
  var oldValueGiven;

  scope.$watch(
    function(scope) { return scope.someValue; },
    function(newValue, oldValue, scope) { oldValueGiven = oldValue; }
  );

  scope.$digest();
  expect(oldValueGiven).toBe(123);
});
```

In `$digest`, as we call the listener, we just check if the old value is the initial value and replace it if so:

src/scope.js

```
Scope.prototype.$digest = function() {
  var self = this;
  var newValue, oldValue;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      watcher.last = newValue;
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
    }
  });
};
```

Getting Notified Of Digests

If you would like to be notified whenever an Angular scope is digested, you can make use of the fact that each watch is executed during each digest: Just register a watch without a listener function. Let's add a test case for this.

test/scope_spec.js

```
it("may have watchers that omit the listener function", function() {
  var watchFn = jasmine.createSpy().and.returnValue('something');
  scope.$watch(watchFn);

  scope.$digest();

  expect(watchFn).toHaveBeenCalled();
});
```

The watch doesn't necessarily have to return anything in a case like this, but it can, and in this case it does. When the scope is digested our current implementation throws an exception. That's because it's trying to invoke a non-existing listener function. To add support for this use case, we need to check if the listener is omitted in `$watch`, and if so, put an empty no-op function in its place:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    last: initWatchVal
  };
  this.$$watchers.push(watcher);
};
```

If you use this pattern, do keep in mind that Angular will look at the return value of `watchFn` even when there is no `listenerFn`. If you return a value, that value is subject to dirty-checking. To make sure your usage of this pattern doesn't cause extra work, just don't return anything. In that case the value of the watch will be constantly `undefined`.

Keep Digesting While Dirty

The core of the implementation is now there, but we're still far from done. For instance, there's a fairly typical scenario we're not supporting yet: The listener functions themselves may also change properties on the scope. If this happens, and there's *another* watcher looking at the property that just changed, it might not notice the change during the same digest pass:

test/scope_spec.js

```
it("triggers chained watchers in the same digest", function() {
  scope.name = 'Jane';

  scope.$watch(
    function(scope) { return scope.nameUpper; },
    function(newValue, oldValue, scope) {
      if (newValue) {
        scope.initial = newValue.substring(0, 1) + '.';
      }
    }
  );

  scope.$watch(
    function(scope) { return scope.name; },
    function(newValue, oldValue, scope) {
      if (newValue) {
        scope.nameUpper = newValue.toUpperCase();
      }
    }
  );

  scope.$digest();
  expect(scope.initial).toBe('J.');
```

```
  scope.name = 'Bob';
  scope.$digest();
  expect(scope.initial).toBe('B.');

});
```

We have two watchers on this scope: One that watches the `nameUpper` property, and assigns `initial` based on that, and another that watches the `name` property and assigns `nameUpper` based on that. What we expect to happen is that when the `name` on the scope changes, the `nameUpper` and `initial` attributes are updated accordingly during the digest. This, however, is not the case.

> We're deliberately ordering the watches so that the dependent one is registered first. If the order was reversed, the test would pass right away because the watches would happen to be in just the right order. However, dependencies between watches do not rely on their registration order, as we're about to see.

What we need to do is to modify the digest so that it keeps iterating over all watches *until the watched values stop changing*. Doing multiple passes is the only way we can get changes applied for watchers that rely on other watchers.

First, let's rename our current `$digest` function to `$$digestOnce`, and adjust it so that it runs all the watchers once, and returns a boolean value that determines whether there were any changes or not:

src/scope.js

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      watcher.last = newValue;
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
      dirty = true;
    }
  });
  return dirty;
};
```

Then, let's redefine `$digest` so that it runs the "outer loop", calling `$$digestOnce` as long as changes keep occurring:

src/scope.js

```
Scope.prototype.$digest = function() {
  var dirty;
  do {
    dirty = this.$$digestOnce();
  } while (dirty);
};
```

`$digest` now runs all watchers *at least* once. If, on the first pass, any of the watched values has changed, the pass is marked dirty, and all watchers are run for a second time. This goes on until there's a full pass where none of the watched values has changed and the situation is deemed stable.

> Angular scopes don't actually have a function called `$$digestOnce`. Instead, the digest loops are all nested within `$digest`. Our goal is clarity over performance, so for our purposes it makes sense to extract the inner loop to a function.

We can now make another important observation about Angular watch functions: They may be run many times per each digest pass. This is why people often say watches should be *idempotent*: A watch function should have no side effects, or only side effects that can happen any number of times. If, for example, a watch function fires an Ajax request, there are no guarantees about how many requests your app is making.

Giving Up On An Unstable Digest

In our current implementation there's one glaring omission: What happens if there are two watches looking at changes made by each other? That is, what if the state *never* stabilizes? Such a situation is shown by the test below:

test/scope_spec.js

```
it("gives up on the watches after 10 iterations", function() {
  scope.counterA = 0;
  scope.counterB = 0;

  scope.$watch(
    function(scope) { return scope.counterA; },
    function(newValue, oldValue, scope) {
      scope.counterB++;
    }
  );

  scope.$watch(
    function(scope) { return scope.counterB; },
    function(newValue, oldValue, scope) {
      scope.counterA++;
    }
  );
```

```
  expect((function() { scope.$digest(); })).toThrow();

});
```

We expect `scope.$digest` to throw an exception, but it never does. In fact, the test never finishes. That's because the two counters are dependent on each other, so on each iteration of `$digestOnce` one of them is going to be dirty.

> Notice that we're not calling the `scope.$digest` function directly. Instead we're passing a function to Jasmine's `expect` function. It will call that function for us, so that it can check that it throws an exception like we expect.

> Since this test will never finish running you'll need to kill the Testem process and start it again once we've fixed the issue.

What we need to do is keep running the digest for some acceptable number of iterations. If the scope is still changing after those iterations we have to throw our hands up and declare it's probably never going to stabilize. At that point we might as well throw an exception, since whatever the state of the scope is it's unlikely to be what the user intended.

This maximum amount of iterations is called the TTL (short for "Time To Live"). By default it is set to 10. The number may seem small, but bear in mind this is a performance sensitive area since digests happen often and each digest runs all watch functions. It's also unlikely that a user will have more than 10 watches chained back-to-back.

> It is actually possible to adjust the TTL in Angular. We will return to this later when we discuss providers and dependency injection.

Let's go ahead and add a loop counter to the outer digest loop. If it reaches the TTL, we'll throw an exception:

src/scope.js

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  do {
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty);
};
```

This updated version causes our interdependent watch example to throw an exception, as our test expected. This should keep the digest from running off on us.

Short-Circuiting The Digest When The Last Watch Is Clean

In the current implementation, we keep iterating over the watch collection until we have witnessed one full round where every watch was clean (or where the TTL was reached).

Since there can be a large amount of watches in a digest loop, it is important to execute them as few times as possible. That is why we're going to apply one specific optimization to the digest loop.

Consider a situation with 100 watches on a scope. When we digest the scope, only the first of those 100 watches happens to be dirty. That single watch "dirties up" the whole digest round, and we have to do another round. On the second round, none of the watches are dirty and the digest ends. But we had to do 200 watch executions before we were done!

What we can do to cut the number of executions in half is to keep track of the last watch we have seen that was dirty. Then, whenever we encounter a clean watch, we check whether it's also the last watch we have seen that was dirty. If so, it means a full round has passed where no watch has been dirty. In that case there is no need to proceed to the end of the current round. We can exit immediately instead. Here's a test case for just that:

test/scope_spec.js

```
it("ends the digest when the last watch is clean", function() {

  scope.array = _.range(100);
  var watchExecutions = 0;

  _.times(100, function(i) {
    scope.$watch(
      function(scope) {
        watchExecutions++;
        return scope.array[i];
      },
      function(newValue, oldValue, scope) {
      }
    );
  });

  scope.$digest();
  expect(watchExecutions).toBe(200);

  scope.array[0] = 420;
  scope.$digest();
  expect(watchExecutions).toBe(301);

});
```

We first put an array of 100 items on the scope. We then attach a 100 watches, each watching a single item in the array. We also add a local variable that's incremented whenever a watch is run, so that we can keep track of the total number of watch executions.

We then run the digest once, just to initialize the watches. During that digest each watch is run twice.

Then we make a change to the very first item in the array. If the short-circuiting optimization were in effect, that would mean the digest would short-circuit on the first watch during second iteration and end immediately, making the number of total watch executions just 301 and not 400.

As mentioned, this optimization can be implemented by keeping track of the last dirty watch. Let's add a field for it to the `Scope` constructor:

src/scope.js

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
}
```

Now, whenever a digest begins, let's set this field to `null`:

src/scope.js

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  do {
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty);
};
```

In `$$digestOnce`, whenever we encounter a dirty watch, let's assign it to this field:

src/scope.js

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      self.$$lastDirtyWatch = watcher;
      watcher.last = newValue;
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
      dirty = true;
    }
  });
  return dirty;
};
```


Also in `$$digestOnce`, whenever we encounter a *clean* watch that also happens to have been the last dirty watch we saw, let's break out of the loop right away and return a falsy value to let the outer `$digest` loop know it should also stop iterating:

src/scope.js

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      self.$$lastDirtyWatch = watcher;
      watcher.last = newValue;
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
      dirty = true;
    } else if (self.$$lastDirtyWatch === watcher) {
      return false;
    }
  });
  return dirty;
};
```

> Returning `false` causes LoDash to exit the for-each loop immmediately. Since we won't have seen any dirty watches this time, `dirty` will be undefined, and that'll be the return value of the function.

The optimization is now in effect. There's one corner case we need to cover though, which we can tease out by adding a watch from the listener of another watch:

test/scope_spec.js

```
it("does not end digest so that new watches are not run", function() {

  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.$watch(
        function(scope) { return scope.aValue; },
        function(newValue, oldValue, scope) {
```

```
          scope.counter++;
        }
      );
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

The second watch is not being executed. The reason is that on the second digest iteration, just before the new watch would run, we're ending the digest because we're detecting the *first* watch as the last dirty watch that's now clean. Let's fix this by re-setting `$$lastDirtyWatch` when a watch is added, effectively disabling the optimization:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    last: initWatchVal
  };
  this.$$watchers.push(watcher);
  this.$$lastDirtyWatch = null;
};
```

Now our digest cycle is potentially a lot faster than before. In a typical application, this optimization may not always eliminate iterations as effectively as in our example, but it does well enough on average that the Angular team has decided to include it.

Now, let's turn our attention to *how* we're actually detecting that something has changed.

Value-Based Dirty-Checking

For now we've been comparing the old value to the new with the strict equality operator `===`. This is fine in most cases, as it detects changes to all primitives (numbers, strings, etc.) and also detects when an object or an array changes to a new one. But there is also another way Angular can detect changes, and that's detecting when something *inside* an object or an array changes. That is, you can watch for changes in *value*, not just in reference.

This kind of dirty-checking is activated by providing a third, optional boolean flag to the `$watch` function. When the flag is `true`, value-based checking is used. Let's add a test that expects this to be the case:

test/scope_spec.js


```
it("compares based on value if enabled", function() {
  scope.aValue = [1, 2, 3];
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    },
    true
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.aValue.push(4);
  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

The test increments a counter whenever the `scope.aValue` array changes. When we push an item to the array, we're expecting it to be noticed as a change, but it isn't. `scope.aValue` is still the same array, it just has different contents now.

Let's first redefine `$watch` to take the boolean flag and store it in the watcher:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    valueEq: !!valueEq,
    last: initWatchVal
  };
  this.$$watchers.push(watcher);
  this.$$lastDirtyWatch = null;
};
```

All we do is add the flag to the watcher, coercing it to a real boolean by negating it twice. When a user calls `$watch` without a third argument, `valueEq` will be `undefined`, which becomes `false` in the watcher object.

Value-based dirty-checking implies that if the old or new values are objects or arrays we have to iterate through everything contained in them. If there's any difference in the two values, the watcher is dirty. If the value has other objects or arrays nested within, those will also be recursively compared by value.

Angular ships with its own equal checking function, but we're going to use the one provided by Lo-Dash instead because it does everything we need at this point. Let's define a new function that takes two values and the boolean flag, and compares the values accordingly:

src/scope.js


```
Scope.prototype.$$areEqual = function(newValue, oldValue, valueEq) {
  if (valueEq) {
    return _.isEqual(newValue, oldValue);
  } else {
    return newValue === oldValue;
  }
};
```

In order to notice changes in value, we also need to change the way we store the old value for each watcher. It isn't enough to just store a reference to the current value, because any changes made within that value will also be applied to the reference we're holding. We would never notice any changes since essentially `$$areEqual` would always get two references to the same value. For this reason we need to make a deep copy of the value and store that instead.

Just like with the equality check, Angular ships with its own deep copying function, but for now we'll be using the one that comes with Lo-Dash.

Let's update `$digestOnce` so that it uses the new `$$areEqual` function, and also copies the `last` reference if needed:

src/scope.js

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (!self.$$areEqual(newValue, oldValue, watcher.valueEq)) {
      self.$$lastDirtyWatch = watcher;
      watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
      dirty = true;
    } else if (self.$$lastDirtyWatch === watcher) {
      return false;
    }
  });
  return dirty;
};
```

Now our code supports both kinds of equality-checking, and our test passes.

Checking by value is obviously a more involved operation than just checking a reference. Sometimes a lot more involved. Walking a nested data structure takes time, and holding a

deep copy of it also takes up memory. That's why Angular does not do value-based dirty checking by default. You need to explicitly set the flag to enable it.

> There's also a third dirty-checking mechanism Angular provides: Collection watching. We will implement it in Chapter 3.

Before we're done with value comparison, there's one more JavaScript quirk we need to handle.

NaNs

In JavaScript, `NaN` (Not-a-Number) is not equal to itself. This may sound strange, and that's because it is. If we don't explicitly handle `NaN` in our dirty-checking function, a watch that has `NaN` as a value will always be dirty.

For value-based dirty-checking this case is already handled for us by the Lo-Dash `isEqual` function. For reference-based checking we need to handle it ourselves. This can be illustrated using a test:

test/scope_spec.js

```
it("correctly handles NaNs", function() {
  scope.number = 0/0; // NaN
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.number; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

We're watching a value that happens to be `NaN` and incrementing a counter when it changes. We expect the counter to increment once on the first `$digest` and then stay the same. Instead, as we run the test we're greeted by the "TTL reached" exception. The scope isn't stabilizing because `NaN` is always considered to be different from the last value.

Let's fix that by tweaking the `$$areEqual` function:

src/scope.js

```
Scope.prototype.$$areEqual = function(newValue, oldValue, valueEq) {
  if (valueEq) {
    return _.isEqual(newValue, oldValue);
  } else {
    return newValue === oldValue ||
      (typeof newValue === 'number' && typeof oldValue === 'number' &&
       isNaN(newValue) && isNaN(oldValue));
  }
};
```

Now the watch behaves as expected with `NaN`s as well.

With the value-checking implementation in place, let's now switch our focus to the ways in which you can interact with a scope from application code.

$eval - Evaluating Code In The Context of A Scope

There are a few ways in which Angular lets you execute some code in the context of a scope. The simplest of these is `$eval`. It takes a function as an argument and immediately executes that function giving it the scope itself as an argument. It then returns whatever the function returned. `$eval` also takes an optional second argument, which it just passes as-is to the function.

Here are a couple of unit tests showing how one can use `$eval`:

test/scope_spec.js

```
it("executes $eval'ed function and returns result", function() {
  scope.aValue = 42;

  var result = scope.$eval(function(scope) {
    return scope.aValue;
  });

  expect(result).toBe(42);
});

it("passes the second $eval argument straight through", function() {
  scope.aValue = 42;

  var result = scope.$eval(function(scope, arg) {
    return scope.aValue + arg;
  }, 2);

  expect(result).toBe(44);
});
```

Implementing `$eval` is straightforward:

src/scope.js

```
Scope.prototype.$eval = function(expr, locals) {
  return expr(this, locals);
};
```

So what is the purpose of such a roundabout way to invoke a function? One could argue that `$eval` makes it just slightly more explicit that some piece of code is dealing specifically with the contents of a scope. `$eval` is also a building block for `$apply`, which is what we'll be looking at next.

However, probably the most interesting use of `$eval` only comes when we start discussing *expressions* instead of raw functions. Just like with `$watch`, you can give `$eval` a string expression. It will compile that expression and execute it within the context of the scope. We will implement this in the second part of the book.

$apply - Integrating External Code With The Digest Cycle

Perhaps the best known of all functions on `Scope` is `$apply`. It is considered the standard way to integrate external libraries to Angular. There's a good reason for this.

`$apply` takes a function as an argument. It executes that function using `$eval`, and then kick-starts the digest cycle by invoking `$digest`. Here's a test case for this:

test/scope_spec.js

```
it("executes $apply'ed function and starts the digest", function() {
  scope.aValue = 'someValue';
  scope.counter = 0;

  scope.$watch(
    function(scope) {
      return scope.aValue;
    },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$apply(function(scope) {
    scope.aValue = 'someOtherValue';
  });
  expect(scope.counter).toBe(2);

});
```


We have a watch that's watching `scope.aValue` and incrementing a counter. We test that the watch is executed immediately when `$apply` is invoked.

Here is a simple implementation that makes the test pass:

src/scope.js

```
Scope.prototype.$apply = function(expr) {
  try {
    return this.$eval(expr);
  } finally {
    this.$digest();
  }
};
```

The `$digest` call is done in a `finally` block to make sure the digest will happen even if the supplied function throws an exception.

The big idea of `$apply` is that we can execute some code that isn't aware of Angular. That code may still change things on the scope, and as long as we wrap the code in `$apply` we can be sure that any watches on the scope will pick up on those changes. When people talk about integrating code to the "Angular lifecycle" using `$apply`, this is essentially what they mean. There really isn't much more to it than that.

$evalAsync - Deferred Execution

In JavaScript it's very common to execute a piece of code "later" – to defer its execution to some point in the future when the current execution context has finished. The usual way to do this is by calling `setTimeout()` with a zero (or very small) delay parameter.

This pattern applies to Angular applications as well, though the preferred way to do it is by using the `$timeout` service that, among other things, integrates the delayed function to the digest cycle with `$apply`.

But there is also another way to defer code in Angular, and that's the `$evalAsync` function on `Scope`. `$evalAsync` takes a function and schedules it to run later *but* still during the ongoing digest *or* before the next digest. You can, for example, defer some code from within a watch listener function, knowing that while that code is deferred, it'll still be invoked within the current digest iteration.

The reason why `$evalAsync` is often preferrable to a `$timeout` with zero delay has to do with the browser event loop. When you use `$timeout` to schedule some work, you relinquish control to the browser, and let it decide when to run the scheduled work. The browser may then choose to execute other work before it gets to your timeout. It may, for example, render the UI, run click handlers, or process Ajax responses. `$evalAsync`, on the other hand, is much more strict about when the scheduled work is executed. Since it'll happen during the ongoing digest, it's guaranteed to run before the browser decides to do anything else. This difference between `$timeout` and `$evalAsync` is especially significant when you

want to prevent unnecessary rendering: Why let the browser render DOM changes that are going to be immediately overridden anyway?

Here's the contract of `$evalAsync` expressed as a unit test:

test/scope_spec.js

```
it("executes $evalAsynced function later in the same cycle", function() {
  scope.aValue = [1, 2, 3];
  scope.asyncEvaluated = false;
  scope.asyncEvaluatedImmediately = false;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.$evalAsync(function(scope) {
        scope.asyncEvaluated = true;
      });
      scope.asyncEvaluatedImmediately = scope.asyncEvaluated;
    }
  );

  scope.$digest();
  expect(scope.asyncEvaluated).toBe(true);
  expect(scope.asyncEvaluatedImmediately).toBe(false);

});
```

We call `$evalAsync` in the watcher's listener function, and then check that the function was executed during the same digest, but *after* the listener function had finished executing.

The first thing we need is a way to store the `$evalAsync` jobs that have been scheduled. We can do this with an array, which we initialize in the `Scope` constructor:

src/scope.js

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
}
```

Let's next define `$evalAsync`, so that it adds the function to execute on this queue:

src/scope.js

```
Scope.prototype.$evalAsync = function(expr) {
  this.$$asyncQueue.push({scope: this, expression: expr});
};
```


> The reason we explicitly store the current scope in the queued object is related to scope inheritance, which we'll discuss in the next chapter.

We've added bookkeeping for the functions that are to be executed, but we still need to actually execute them. That will happen in `$digest`: The first thing we do in `$digest` is consume everything from this queue and invoke all the deferred functions using `$eval` on the scope that was attached to the async task:

src/scope.js

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty);
};
```

The implementation guarantees that if you defer a function while the scope is still dirty, the function will be invoked later but still during the same digest. That satisfies our unit test.

Scheduling $evalAsync from Watch Functions

In the previous section we saw how a function scheduled from a listener function using `$evalAync` will be executed in the same digest loop. But what happens if you schedule an `$evalAsync` from a *watch function*? Granted, this is something one should not do, since watch function are supposed to be side-effect free. But it is still *possible* to do it, so we should make sure it doesn't wreak havoc on the digest.

If we consider a situation where a watch function schedules an `$evalAsync` *once*, everything seems to be in order. The following test case passes with our current implementation:

test/scope_spec.js


```
it("executes $evalAsynced functions added by watch functions", function() {
  scope.aValue = [1, 2, 3];
  scope.asyncEvaluated = false;

  scope.$watch(
    function(scope) {
      if (!scope.asyncEvaluated) {
        scope.$evalAsync(function(scope) {
          scope.asyncEvaluated = true;
        });
      }
      return scope.aValue;
    },
    function(newValue, oldValue, scope) { }
  );

  scope.$digest();

  expect(scope.asyncEvaluated).toBe(true);
});
```

So what's the problem? As we have seen, we keep running the digest loop as long as there is at least one watch that is dirty. In the test case above, this is the case in the first iteration, when we first return `scope.aValue` from the watch function. That causes the digest to go into the next iteration, during which it also runs the function we scheduled using `$evalAsync`. But what if we schedule an `$evalAsync` when no watch is actually dirty?

test/scope_spec.js

```
it("executes $evalAsynced functions even when not dirty", function() {
  scope.aValue = [1, 2, 3];
  scope.asyncEvaluatedTimes = 0;

  scope.$watch(
    function(scope) {
      if (scope.asyncEvaluatedTimes < 2) {
        scope.$evalAsync(function(scope) {
          scope.asyncEvaluatedTimes++;
        });
      }
      return scope.aValue;
    },
    function(newValue, oldValue, scope) { }
  );

  scope.$digest();

  expect(scope.asyncEvaluatedTimes).toBe(2);
});
```


This version does the `$evalAsync` twice. On the second time, the watch function won't be dirty since `scope.aValue` hasn't changed. That means the `$evalAsync` also doesn't run since the `$digest` has terminated. While it would be run on the *next* digest, we really want it to run during this one. That means we need to tweak the termination condition in `$digest` to also see whether there's something in the async queue:

src/scope.js

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
};
```

The test passes, but now we've introduced another problem. What if a watch function *always* schedules something using `$evalAsync`? We might expect it to cause the iteration limit to be reached, but it does not:

test/scope_spec.js

```
it("eventually halts $evalAsyncs added by watches", function() {
  scope.aValue = [1, 2, 3];

  scope.$watch(
    function(scope) {
      scope.$evalAsync(function(scope) { });
      return scope.aValue;
    },
    function(newValue, oldValue, scope) { }
  );

  expect(function() { scope.$digest(); }).toThrow();
});
```

This test case will run forever, since the `while` loop in `$digest` never terminates. What we need to do is also check the status of the async queue in our TTL check:

src/scope.js


```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
};
```

Now we can be sure the digest will terminate, regardless of whether it's running because it's dirty or because there's something in its async queue.

Scope Phases

Another thing `$evalAsync` does is to schedule a `$digest` if one isn't already ongoing. That means, whenever you call `$evalAsync` you can be sure the function you're deferring will be invoked "very soon" instead of waiting for something else to trigger a digest.

For this to work there needs to be some way for `$evalAsync` to check whether a `$digest` is already ongoing, because in that case it won't bother scheduling one. For this purpose, Angular scopes implement something called a *phase*, which is simply a string attribute in the scope that stores information about what's currently going on.

As a unit test, let's make an expectation about a field called `$$phase`, which should be "$digest" during a digest, "$apply" during an apply function invocation, and `null` otherwise:

test/scope_spec.js

```
it("has a $$phase field whose value is the current digest phase", function() {
  scope.aValue = [1, 2, 3];
  scope.phaseInWatchFunction = undefined;
  scope.phaseInListenerFunction = undefined;
  scope.phaseInApplyFunction = undefined;

  scope.$watch(
    function(scope) {
      scope.phaseInWatchFunction = scope.$$phase;
      return scope.aValue;
    },
    function(newValue, oldValue, scope) {
```

```
      scope.phaseInListenerFunction = scope.$$phase;
    }
  );

  scope.$apply(function(scope) {
    scope.phaseInApplyFunction = scope.$$phase;
  });

  expect(scope.phaseInWatchFunction).toBe('$digest');
  expect(scope.phaseInListenerFunction).toBe('$digest');
  expect(scope.phaseInApplyFunction).toBe('$apply');

});
```

> We don't need to explicitly call `$digest` here to digest the scope, because invoking `$apply` does it for us.

In the `Scope` constructor, let's introduce the `$$phase` field, setting it initially as `null`:

src/scope.js

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$phase = null;
}
```

Next, let's define a couple of functions for controlling the phase: One for setting it and one for clearing it. Let's also add an additional check to make sure we're not trying to set a phase when one is already active:

src/scope.js

```
Scope.prototype.$beginPhase = function(phase) {
  if (this.$$phase) {
    throw this.$$phase + ' already in progress.';
  }
  this.$$phase = phase;
};

Scope.prototype.$clearPhase = function() {
  this.$$phase = null;
};
```

In `$digest`, let's now set the phase as `"$digest"` for the duration of the outer digest loop:

src/scope.js


```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  this.$beginPhase("$digest");
  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      this.$clearPhase();
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
  this.$clearPhase();
};
```

Let's also tweak `$apply` so that it also sets the phase for itself:

src/scope.js

```
Scope.prototype.$apply = function(expr) {
  try {
    this.$beginPhase("$apply");
    return this.$eval(expr);
  } finally {
    this.$clearPhase();
    this.$digest();
  }
};
```

And finally we can add the scheduling of the `$digest` into `$evalAsync`. Let's first define the requirement as a unit test:

test/scope_spec.js

```
it("schedules a digest in $evalAsync", function(done) {
  scope.aValue = "abc";
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
```



```
  );

  scope.$evalAsync(function(scope) {
  });

  expect(scope.counter).toBe(0);
  setTimeout(function() {
    expect(scope.counter).toBe(1);
    done();
  }, 50);
});
```

We check that the digest is indeed run, not immediately during the `$evalAsync` call but just slightly after it. Our definition of "slightly after" here is after a 50 millisecond timeout. To make `setTimeout` work with Jasmine, we use its asynchronous test support: The test case function takes an optional `done` callback argument, and will only finish once we have called it, which we do after the timeout.

The `$evalAsync` function can now check the current phase of the scope, and if there isn't one (and no async tasks have been scheduled yet), schedule the digest.

src/scope.js

```
Scope.prototype.$evalAsync = function(expr) {
  var self = this;
  if (!self.$$phase && !self.$$asyncQueue.length) {
    setTimeout(function() {
      if (self.$$asyncQueue.length) {
        self.$digest();
      }
    }, 0);
  }
  self.$$asyncQueue.push({scope: self, expression: expr});
};
```

With this implementation, when you invoke `$evalAsync` you can be sure a digest will happen in the near future, regardless of when or where you invoke it.

If you call `$evalAsync` when a digest is already running, your function will be evaluated during that digest. If there is no digest running, one is started. We use `setTimeout` to defer the beginning of the digest slightly. This way callers of `$evalAsync` can be ensured the function will always return immediately instead of evaluating the expression synchronously, regardless of the current status of the digest cycle.

Running Code After A Digest - $$postDigest

There's one more way you can attach some code to run in relation to the digest cycle, and that's by scheduling a `$$postDigest` function.

The double dollar sign in the name of the function hints that this is really an internal facility for Angular, rather than something application developers should use. But it is there, so we'll also implement it.

Just like `$evalAsync`, `$$postDigest` schedules a function to run "later". Specifically, the function will be run *after* the next digest has finished. Similarly to `$evalAsync`, a function scheduled with `$$postDigest` is executed just once. Unlike `$evalSync`, scheduling a `$$postDigest` function does *not* cause a digest to be scheduled, so the function execution is delayed until the digest happens for some other reason. Here's a unit test the specifies these requirements:

test/scope_spec.js

```
it("runs a $$postDigest function after each digest", function() {
  scope.counter = 0;

  scope.$$postDigest(function() {
    scope.counter++;
  });

  expect(scope.counter).toBe(0);

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$digest();
  expect(scope.counter).toBe(1);

});
```

As the name implies, `$$postDigest` functions run *after* the digest, so if you make changes to the scope from within `$$postDigest` they won't be immediately picked up by the dirty-checking mechanism. If that's what you want, you should call `$digest` or `$apply` manually:

test/scope_spec.js

```
it("does not include $$postDigest in the digest", function() {
  scope.aValue = 'original value';

  scope.$$postDigest(function() {
    scope.aValue = 'changed value';
  });
  scope.$watch(
    function(scope) {
```



```
      return scope.aValue;
    },
    function(newValue, oldValue, scope) {
      scope.watchedValue = newValue;
    }
  );

  scope.$digest();
  expect(scope.watchedValue).toBe('original value');

  scope.$digest();
  expect(scope.watchedValue).toBe('changed value');

});
```

To implement `$$postDigest` let's first initialize one more array in the `Scope` constructor:

src/scope.js

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$postDigestQueue = [];
  this.$$phase = null;
}
```

Next, let's implement `$$postDigest` itself. All it does is add the given function to the queue:

src/scope.js

```
Scope.prototype.$$postDigest = function(fn) {
  this.$$postDigestQueue.push(fn);
};
```

Finally, in `$digest`, let's drain the queue and invoke all those functions once the digest has finished:

src/scope.js

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  this.$beginPhase("$digest");
  do {
```



```
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      this.$clearPhase();
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
  this.$clearPhase();

  while (this.$$postDigestQueue.length) {
    this.$$postDigestQueue.shift()();
  }
};
```

We consume the queue by removing functions from the beginning of the array using `Array.shift()` until the array is empty, and by immediately executing those functions. `$$postDigest` functions are not given any arguments.

Handling Exceptions

Our `Scope` implementation is becoming something that resembles the one in Angular. It is, however, quite brittle. That's mainly because we haven't put much thought into exception handling.

If an exception occurs in a watch function, an `$evalAsync` function, or a `$$postDigest` function, our current implementation will just give up and stop whatever it's doing. Angular's implementation, however, is actually much more robust than that. Exceptions thrown before, during, or after a digest are caught and logged, and the operation then resumed where it left off.

> Angular actually forwards exceptions to a special `$exceptionHandler` service. Since we don't have such a service yet, we'll simply log the exceptions to the console for now.

In watches there are two points when exceptions can happen: In the watch functions and in the listener functions. In either case, we want to log the exception and continue with the next watch as if nothing had happened. Here are two test cases for the two functions:

test/scope_spec.js

```
it("catches exceptions in watch functions and continues", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
```

```
    function(scope) { throw "error"; },
    function(newValue, oldValue, scope) { }
  );
  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});

it("catches exceptions in listener functions and continues", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      throw "Error";
    }
  );
  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

In both cases we define two watches, the first of which throws an exception. We check that the second watch is still executed.

To make these tests pass we need to modify the `$$digestOnce` function and wrap the execution of each watch in a `try...catch` clause:

src/scope.js

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    try {
      newValue = watcher.watchFn(self);
      oldValue = watcher.last;
```

```
      if (!self.$$areEqual(newValue, oldValue, watcher.valueEq)) {
        self.$$lastDirtyWatch = watcher;
        watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
        watcher.listenerFn(newValue,
          (oldValue === initWatchVal ? newValue : oldValue),
          self);
        dirty = true;
      } else if (self.$$lastDirtyWatch === watcher) {
        return false;
      }
    } catch (e) {
      console.error(e);
    }
  });
  return dirty;
};
```

What remains is exception handling in `$evalAsync` and `$postDigest`. Both are used to execute arbitrary functions in relation to the digest loop – one before a digest and one after it. In neither case do we want an exception to cause the loop to end prematurely.

For `$evalAsync` we can define a test case that checks that a watch is run even when an exception is thrown from one of the functions scheduled for `$evalAsync`:

test/scope_spec.js

```
it("catches exceptions in $evalAsync", function(done) {
  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$evalAsync(function(scope) {
    throw "Error";
  });

  setTimeout(function() {
    expect(scope.counter).toBe(1);
    done();
  }, 50);

});
```

For `$postDigest` the digest will already be over, so it doesn't make sense to test it with a watch. We can test it with a second `$postDigest` function instead, making sure it also executes:

test/scope_spec.js

```
it("catches exceptions in $$postDigest", function() {
  var didRun = false;

  scope.$$postDigest(function() {
    throw "Error";
  });
  scope.$$postDigest(function() {
    didRun = true;
  });

  scope.$digest();
  expect(didRun).toBe(true);
});
```

Fixing both `$evalAsync` and `$postDigest` involves changing the `$digest` function. In both cases we wrap the function execution in `try...catch`:

test/scope_spec.js

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  this.$beginPhase('$digest');
  do {
    while (this.$$asyncQueue.length) {
      try {
        var asyncTask = this.$$asyncQueue.shift();
        asyncTask.scope.$eval(asyncTask.expression);
      } catch (e) {
        console.error(e);
      }
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
  this.$clearPhase();

  while (this.$$postDigestQueue.length) {
    try {
      this.$$postDigestQueue.shift()();
    } catch (e) {
```

```
      console.error(e);
    }
  }
};
```

Our digest cycle is now a lot more robust when it comes to exceptions.

Destroying A Watch

When you register a watch, most often you want it to stay active as long as the scope itself does, so you don't ever really explicitly remove it. There are cases, however, where you want to destroy a particular watch while still keeping the scope operational. That means we need a removal operation for watches.

The way Angular implements this is actually quite clever: The `$watch` function in Angular has a return value. It is a function that, when invoked, destroys the watch that was registered. If a user wants to be able to remove a watch later, they just need to keep hold of the function returned when they registered the watch, and then call it once the watch is no longer needed:

test/scope_spec.js

```
it("allows destroying a $watch with a removal function", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  var destroyWatch = scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.aValue = 'def';
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.aValue = 'ghi';
  destroyWatch();
  scope.$digest();
  expect(scope.counter).toBe(2);
});
```


To implement this, we need to return a function that removes the watch from the `$$watchers` array:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn,
    valueEq: !!valueEq,
    last: initWatchVal
  };
  self.$$watchers.push(watcher);
  this.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
    }
  };
};
```

While that takes care of the watch removal itself, there are a few corner cases we need to deal with before we have a robust implementation. They all have to do with the not too uncommon use case of removing a watch *during a digest*.

First of all, a watch might remove *itself* in its own watch or listener function. This should not affect other watches:

test/scope_spec.js

```
it("allows destroying a $watch during digest", function() {
  scope.aValue = 'abc';

  var watchCalls = [];

  scope.$watch(
    function(scope) {
      watchCalls.push('first');
      return scope.aValue;
    }
  );

  var destroyWatch = scope.$watch(
    function(scope) {
      watchCalls.push('second');
      destroyWatch();
    }
  );
```



```
  scope.$watch(
    function(scope) {
      watchCalls.push('third');
      return scope.aValue;
    }
  );

  scope.$digest();
  expect(watchCalls).toEqual(['first', 'second', 'third', 'first', 'third']);
});
```

In the test we have three watches, the second of which removes itself when it's first called. We verify that the watches are iterated in the correct order: During the first turn of the loop each watch is executed once. Then, since the digest was dirty, each watch is executed again, but this time the second watch is no longer there.

What's happening instead is that when the second watch removes itself, the watch collection gets shifted to the left, causing `$$digestOnce` to skip the third watch during that round.

The trick is to reverse the `$$watches` array, so that new watches are added to the beginning of it and iteration is done from the end to the beginning. When a watches is then removed, the part of the watch array that gets shifted has already been handled during that digest iteration and it won't affect the rest of it.

When adding a watch, we should use `Array.unshift` instead of `Array.push`:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    last: initWatchVal,
    valueEq: !!valueEq
  };
  this.$$watchers.unshift(watcher);
  this.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
      self.$$lastDirtyWatch = null;
    }
  };
};
```

Then, when iterating, we should use `_.forEachRight` instead of `_.forEach` to reverse the iteration order:

src/scope.js


```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEachRight(this.$$watchers, function(watcher) {
    try {
      newValue = watcher.watchFn(self);
      oldValue = watcher.last;
      if (!self.$$areEqual(newValue, oldValue, watcher.valueEq)) {
        self.$$lastDirtyWatch = watcher;
        watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
        watcher.listenerFn(newValue,
          (oldValue === initWatchVal ? newValue : oldValue),
          self);
        dirty = true;
      } else if (self.$$lastDirtyWatch === watcher) {
        return false;
      }
    } catch (e) {
      console.error(e);
    }
  });
  return dirty;
};
```

The next case is a watch removing *another watch*. Observe the following test case:

src/scope.js

```
it("allows a $watch to destroy another during digest", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) {
      return scope.aValue;
    },
    function(newValue, oldValue, scope) {
      destroyWatch();
    }
  );

  var destroyWatch = scope.$watch(
    function(scope) { },
    function(newValue, oldValue, scope) { }
  );

  scope.$watch(
    function(scope) { return scope.aValue; },
```



```
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

This test case fails. The culprit is our short-circuiting optimization. Recall that in `$$digestOnce` we see whether the current watch was the last dirty one seen and is now clean. If so, we end the digest. What happens in this test case is:

1. The first watch is executed. It is dirty, so it is stored in `$$lastDirtyWatch` and its listener is executed. The listener destroys the second watch.
2. The first watch is executed *again*, because it has moved one position down in the watcher array. This time it is clean, and since it is also in `$$lastDirtyWatch`, the digest ends. We never get to the third watch.

We should eliminate the short-circuiting optimization on watch removal so that this does not happen:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn,
    valueEq: !!valueEq,
    last: initWatchVal
  };
  self.$$watchers.unshift(watcher);
  this.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
      self.$$lastDirtyWatch = null;
    }
  };
};
```

The final case to consider is when a watch removes *several watches* when executed:

test/scope_spec.js

```
it("allows destroying several $watches during digest", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  var destroyWatch1 = scope.$watch(
    function(scope) {
      destroyWatch1();
      destroyWatch2();
    }
  );

  var destroyWatch2 = scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(0);
});
```

The first watch destroys not only itself, but also a second watch that would have been executed next. While we don't expect that second watch to execute, we don't expect an exception to be thrown either, which is what actually happens.

What we need to do is check that the current watch actually exists while we're iterating:

src/scope.js

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEachRight(this.$$watchers, function(watcher) {
    try {
      if (watcher) {
        newValue = watcher.watchFn(self);
        oldValue = watcher.last;
        if (!self.$$areEqual(newValue, oldValue, watcher.valueEq)) {
          self.$$lastDirtyWatch = watcher;
          watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
          watcher.listenerFn(newValue,
            (oldValue === initWatchVal ? newValue : oldValue),
            self);
          dirty = true;
        } else if (self.$$lastDirtyWatch === watcher) {
          return false;
        }
      }
    } catch (e) {
```



```
      console.error(e);
    }
  });
  return dirty;
};
```

And finally we can rest assured our digest will keep on running regardless of watches being removed.

Summary

We've already come a long way, and have a perfectly usable implementation of an Angular-style dirty-checking scope system. In the process you have learned about:

- The two-sided process underlying Angular's dirty-checking: `$watch` and `$digest`.
- The dirty-checking loop and the TTL mechanism for short-circuiting it.
- The difference between reference-based and value-based comparison.
- Executing functions on the digest loop in different ways: Immediately with `$eval` and `$apply` and time-shifted with `$evalAsync` and `$$postDigest`.
- Exception handling in the Angular digest.

There is, of course, a lot more to Angular scopes than this. In the next chapter we'll start looking at how scopes can *inherit from other scopes*, and how watches can watch things not only on the scope they are attached to, but also on that scope's parents.

Chapter 2

Scope Inheritance

In this chapter we'll look into the way scopes connect to each other using *inheritance*. This is the mechanism that allows an Angular scope to access properties on its parents, all the way up to the root scope.

Just like a subclass in Java, C#, or Ruby shares the fields and methods declared in its parent class, a scope in Angular shares the contents of its parents. And just like class inheritance, scope inheritance can sometimes make things difficult to understand. There are things on a scope that you can't see when you look at the code, because they've been attached somewhere else. You can also easily cause unintended changes to happen, by manipulating something on the parent scope, that's also being watched by some of its *other* children. There's great power in scope inheritance but you have to use it carefully.

Angular's scope inheritance mechanism actually builds fairly directly on JavaScript's prototypal object inheritance, adding just a few things on top of it. That means you'll understand Angular's scope inheritance best when you understand JavaScript's prototype chains. It also means that to implement scope inheritance there isn't a huge amount of code for us to write.

The Root Scope

So far we have been working with a single scope object, which we created using the `Scope` constructor:

```
var scope = new Scope();
```

A scope created like this is a *root scope*. It's called that because it has no parent, and it is typically the root of a whole tree of child scopes.

In reality you will never create a scope in this way. In an Angular application, there is exactly one root scope (available by injecting `$rootScope`). All other scopes are its descendants, created for controllers and directives.

Making A Child Scope

Though you can make as many root scopes as you want, what usually happens instead is you create a child scope for an existing scope (or let Angular do it for you). This can be done by invoking a function called `$new` on an existing scope.

Let's test-drive the implementation of `$new`. Before we start, first add a new nested `describe` block in `test/scope_spec.js` for all our tests related to inheritance. The test file should have a structure like the following:

test/scope_spec.js

```
describe("Scope", function() {

  describe("digest", function() {

    // Tests from the previous chapter...

  });

  describe("inheritance", function() {

    // Tests for this chapter

  });

});
```

The first thing about a child scope is that it shares the properties of its parent scope:

test/scope_spec.js

```
it("inherits the parent's properties", function() {
  var parent = new Scope();
  parent.aValue = [1, 2, 3];

  var child = parent.$new();

  expect(child.aValue).toEqual([1, 2, 3]);
});
```

The same is not true the other way around. A property defined on the child doesn't exist on the parent:

test/scope_spec.js

```
it("does not cause a parent to inherit its properties", function() {
  var parent = new Scope();

  var child = parent.$new();
  child.aValue = [1, 2, 3];

  expect(parent.aValue).toBeUndefined();
});
```

The sharing of the properties has nothing to do with *when* the properties are defined. When a property is defined on a parent scope, all of the scope's *existing* child scopes also get the property:

test/scope_spec.js

```
it("inherits the parent's properties whenever they are defined", function() {
  var parent = new Scope();
  var child = parent.$new();

  parent.aValue = [1, 2, 3];

  expect(child.aValue).toEqual([1, 2, 3]);
});
```

You can also *manipulate* a parent scope's properties from the child scope, since both scopes actually point to the same value:

test/scope_spec.js

```
it("can manipulate a parent scope's property", function() {
  var parent = new Scope();
  var child = parent.$new();
  parent.aValue = [1, 2, 3];

  child.aValue.push(4);

  expect(child.aValue).toEqual([1, 2, 3, 4]);
  expect(parent.aValue).toEqual([1, 2, 3, 4]);
});
```

You can also *watch* a parent scope's properties from a child scope:

test/scope_spec.js

```
it("can watch a property in the parent", function() {
  var parent = new Scope();
  var child = parent.$new();
  parent.aValue = [1, 2, 3];
  child.counter = 0;

  child.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    },
    true
  );

  child.$digest();
  expect(child.counter).toBe(1);

  parent.aValue.push(4);
  child.$digest();
  expect(child.counter).toBe(2);

});
```

> You may have noticed the child scope also has the `$watch` function, which we defined for `Scope.prototype`. This happens through exactly the same inheritance mechanism as what we use for the user-defined attributes: Since the parent scope inherits `Scope.prototype`, and the child scope inherits the parent scope, everything defined in `Scope.prototype` is available in every scope!

Finally, everything discusses above applies to scope hierarchies of arbitrary depths:

test/scope_spec.js

```
it("can be nested at any depth", function() {
  var a   = new Scope();
  var aa  = a.$new();
  var aaa = aa.$new();
  var aab = aa.$new();
  var ab  = a.$new();
  var abb = ab.$new();

  a.value = 1;

  expect(aa.value).toBe(1);
  expect(aaa.value).toBe(1);
  expect(aab.value).toBe(1);
  expect(ab.value).toBe(1);
  expect(abb.value).toBe(1);
```

```
  ab.anotherValue = 2;

  expect(abb.anotherValue).toBe(2);
  expect(aa.anotherValue).toBeUndefined();
  expect(aaa.anotherValue).toBeUndefined();
});
```

For everything we've specified so far, the implementation is actually very straightforward. We just need to tap into JavaScript's object inheritance, since Angular scopes are deliberately modeled closely on how JavaScript itself works. Essentially, when you create a child scope, its parent will be made its *prototype*.

> We won't spend much time discussing what prototypes in JavaScript mean. If you feel like taking a refresher on them, DailyJS has very good articles on prototypes and inheritance.

Let's create the `$new` function on our `Scope` prototype. It creates a child scope for the current scope, and returns it:

src/scope.js

```
Scope.prototype.$new = function() {
  var ChildScope = function() { };
  ChildScope.prototype = this;
  var child = new ChildScope();
  return child;
};
```

In the function we first create a *constructor function* for the child and put it in a local variable. The constructor doesn't really need to do anything, so we just make it an empty function. We then set `Scope` as the prototype of `ChildScope`. Finally we create a new object using the `ChildScope` constructor and return it.

This short function is enough to make all our test cases pass!

Attribute Shadowing

One aspect of scope inheritance that commonly trips up Angular newcomers is the *shadowing* of attributes. While this is a direct consequence of using JavaScript prototype chains, it is worth discussing.

It is clear from our existing test cases that when you *read* an attribute from a scope, it will look it up on the prototype chain, finding it from a parent scope if it doesn't exist on the current one. Then again, when you *assign* an attribute on a scope, it is only available on that scope and its children, *not* its parents.

The key realization is that this rule also applies when we reuse an attribute name on a child scope:

test/scope_spec.js


```
it("shadows a parent's property with the same name", function() {

  var parent = new Scope();
  var child = parent.$new();

  parent.name = 'Joe';
  child.name = 'Jill';

  expect(child.name).toBe('Jill');
  expect(parent.name).toBe('Joe');

});
```

When we assign an attribute on a child that already exists on a parent, this does not change the parent. In fact, we now have two *different* attributes on the scope chain, both called `name`. This is commonly referred to as *shadowing*: From the child's perspective, the `name` attribute of the parent is *shadowed* by the `name` attribute of the child.

This is a common source of confusion, and there are of course genuine use cases for mutating state on a parent scope. To get around this, a common pattern is to *wrap* the attribute in an object. The contents of that object can be mutated (just like in the array manipulation example from the last section):

test/scope_spec.js

```
it("does not shadow members of parent scope's attributes", function() {

  var parent = new Scope();
  var child = parent.$new();

  parent.user = {name: 'Joe'};
  child.user.name = 'Jill';

  expect(child.user.name).toBe('Jill');
  expect(parent.user.name).toBe('Jill');

});
```

The reason this works is that we don't assign anything on the child scope. We merely *read* the `user` attribute from the scope and assign something within that object. Both scopes have a reference to the same `user` object, which is a plain JavaScript object that has nothing to do with scope inheritance.

> This pattern can be rephrased as the *Dot Rule*, referring to the amount of property access dots you have in an expression that makes changes to the scope. As phrased by Miško Hevery, "Whenever you use ngModel, there's got to be a dot in there somewhere. If you don't have a dot, you're doing it wrong."

Separated Watches

We have already seen that we can attach watches on child scopes, since a child scope inherits all the parent's methods, including `$watch` and `$digest`. But where are the watches actually stored and on which scope are they executed?

In our current implementation, all the watches are in fact stored on the root scope. That's because we define the `$$watchers` array in `Scope`, the root scope constructor. When any child scope accesses the `$$watchers` array (or any other property initialized in the constructor), they get the root scope's copy of it through the prototype chain.

This has one significant implication: *Regardless of what scope we call `$digest` on, we will execute all the watches in the scope hierarchy.* That's because there's just one watch array: The one in the root scope. This isn't exactly what we want.

What we really want to happen when we call `$digest` is to digest the watches attached to the scope we called, and its children. *Not* the watches attached to its parents or any other children they might have, which is what currently happens:

test/scope_spec.js

```
it("does not digest its parent(s)", function() {
  var parent = new Scope();
  var child = parent.$new();

  parent.aValue = 'abc';
  parent.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.aValueWas = newValue;
    }
  );

  child.$digest();
  expect(child.aValueWas).toBeUndefined();
});
```

This test fails because when we call `child.$digest()`, we are in fact executing the watch attached to `parent`. Let's fix this.

The trick is to assign each child scope its own `$$watchers` array:

src/scope.js

```
Scope.prototype.$new = function() {
  var ChildScope = function() { };
  ChildScope.prototype = this;
  var child = new ChildScope();
  child.$$watchers = [];
  return child;
};
```


You may have noticed that we're doing attribute shadowing here, as discussed in the previous section. The `$$watchers` array of each scope shadows the one in its parent. Every scope in the hierarchy has its own watchers. When we call `$digest` on a scope, it is the watchers from that exact scope that get executed.

Recursive Digestion

In the previous section we discussed how calling `$digest` should not run watches up the hierarchy. It should, however, run watches *down* the hierarchy, on the children of the scope we're calling. This makes sense because some of those watches down the hierarchy could be watching our properties through the prototype chain.

Since we now have a separate watcher array for each scope, as it stands child scopes are *not* being digested when we call `$digest` on the parent. We need to fix that by changing `$digest` to work not only on the scope itself, but also its children.

The first problem we have is that a scope doesn't currently have any idea if it has children or not, or who those children might be. We need each scope to keep a record of its child scopes. This needs to happen both for root scopes and child scopes. Let's keep the scopes in an array attribute called `$$children`:

test/scope_spec.js

```
it("keeps a record of its children", function() {
  var parent = new Scope();
  var child1 = parent.$new();
  var child2 = parent.$new();
  var child2_1 = child2.$new();

  expect(parent.$$children.length).toBe(2);
  expect(parent.$$children[0]).toBe(child1);
  expect(parent.$$children[1]).toBe(child2);

  expect(child1.$$children.length).toBe(0);

  expect(child2.$$children.length).toBe(1);
  expect(child2.$$children[0]).toBe(child2_1);

});
```

We need to initialize the `$$children` array in the root scope constructor:

src/scope.js


```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$postDigestQueue = [];
  this.$$children = [];
  this.$$phase = null;
}
```

Then we need to add new children to this array as they are created. We also need to assign those children their own `$$children` array (which shadows the one in the parent), so that we don't run into the same problem we had with `$$watchers`. Both of these changes go in `$new`:

src/scope.js

```
Scope.prototype.$new = function() {
  var ChildScope = function() { };
  ChildScope.prototype = this;
  var child = new ChildScope();
  this.$$children.push(child);
  child.$$watchers = [];
  child.$$children = [];
  return child;
};
```

Now that we have bookkeeping for the children, we can discuss digesting them. We want a `$digest` call on a parent to execute watches in a child:

test/scope_spec.js

```
it("digests its children", function() {
  var parent = new Scope();
  var child = parent.$new();

  parent.aValue = 'abc';
  child.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.aValueWas = newValue;
    }
  );

  parent.$digest();
  expect(child.aValueWas).toBe('abc');
});
```

Notice how this test is basically a mirror image of the test in the last section, where we asserted that calling `$digest` on a child should *not* run watches on the parent.

To make this work, we need to change `$$digestOnce` to run watches throughout the hierarchy. To make that easier, let's first add a helper function `$$everyScope` (named after JavaScript's `Array.every`) that executes an arbitrary function once for each scope in the hierarchy until the function returns a falsy value:

src/scope.js

```
Scope.prototype.$$everyScope = function(fn) {
  if (fn(this)) {
    return this.$$children.every(function(child) {
      return child.$$everyScope(fn);
    });
  } else {
    return false;
  }
};
```

The function invokes `fn` once for the current scope, and then recursively calls itself on each child.

We can now use this function in `$$digestOnce` to form an outer loop for the whole operation:

src/scope.js

```
Scope.prototype.$$digestOnce = function() {
  var dirty;
  var self = this;
  this.$$everyScope(function(scope) {
    var newValue, oldValue;
    _.forEachRight(scope.$$watchers, function(watcher) {
      try {
        if (watcher) {
          newValue = watcher.watchFn(scope);
          oldValue = watcher.last;
          if (!scope.$$areEqual(newValue, oldValue, watcher.valueEq)) {
            self.$$lastDirtyWatch = watcher;
            watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
            watcher.listenerFn(newValue,
              (oldValue === initWatchVal ? newValue : oldValue),
              scope);
            dirty = true;
          } else if (self.$$lastDirtyWatch === watcher) {
            dirty = false;
            return false;
          }
        }
      } catch (e) {
        console.error(e);
```



```
      }
    });
    return dirty !== false;
  });
  return dirty;
};
```

The `$$digestOnce` function now runs through the whole hierarchy and returns a boolean indicating whether any watch anywhere in the hierarchy was dirty.

When the inner loop enables the last dirty watch optimization, it explicitly sets `dirty` to `false` and exits, causing the outer loop to also end.

Notice that we've replaced the references to `this` in the inner loop to the particular `scope` variable being worked on. *The watch functions must be passed the scope object they were originally attached to, not the scope object we happen to call `$digest` on.*

Notice also that with the `$$lastDirtyWatch` attribute we are always referring to the topmost scope. The short-circuiting optimization needs to account for all watches in the scope hierarchy. If we would set `$$lastDirtyWatch` on the current scope it would shadow the parent's attribute.

> Angular.js does not actually have an array called `$$children` on the scope. Instead, if you look at the source, you'll see that it maintains the children in a bespoke linked list style group of variables: `$$nextSibling`, `$$prevSibling`, `$$childHead`, and `$$childTail`. This is an optimization for making it cheaper to add and remove scopes by not having to manipulate an array. Functionally it does the same as our array of `$$children` does.

Digesting The Whole Tree from $apply and $evalAsync

As we saw in the previous sections, `$digest` works only from the current scope down. This is not the case with `$apply`. When you call `$apply` in Angular, that goes directly to the root and digests *the whole scope hierarchy.* Our implementation does not do that yet, as the following test illustrates:

test/scope_spec.js

```
it("digests from root on $apply", function() {
  var parent = new Scope();
  var child = parent.$new();
  var child2 = child.$new();

  parent.aValue = 'abc';
  parent.counter = 0;
  parent.$watch(
```

```
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  child2.$apply(function(scope) { });

  expect(parent.counter).toBe(1);
});
```

When we call `$apply` on a child, it doesn't currently trigger a watch in its grandparent.

To make this work, we first of all need scopes to have a reference to their root so that they can trigger the digestion on it. We could find the root by walking up the prototype chain, but it's much more straightforward to just have an explicit `$$root` attribute available. We can set one up in the root scope constructor:

src/scope.js

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$postDigestQueue = [];
  this.$$root = this;
  this.$$children = [];
  this.$$phase = null;
}
```

This alone makes `$$root` available to every scope in the hierarchy, thanks to the prototypal inheritance chain.

The change we still need to make in `$apply` is straightforward. Instead of calling `$digest` on the current scope, we do so on the root scope:

src/scope.js

```
Scope.prototype.$apply = function(expr) {
  try {
    this.$beginPhase('$apply');
    return this.$eval(expr);
  } finally {
    this.$clearPhase();
    this.$$root.$digest();
  }
};
```

Note that we still evaluate the given function on the current scope, not the root scope, by virtue of calling `$eval` on `this`. It's just the digest that we want to run from the root down.

The fact that `$apply` digests all the way from the root is one of the reasons it is the preferred method for integrating external code to Angular in favor of plain `$digest`: If you don't know exactly what scopes are relevant to the change you're making, it's a safe bet to just involve all of them.

It is notable that since most Angular applications have just one root scope, `$apply` does cause every watch on every scope in the whole application to be executed. Armed with the knowledge about this difference between `$digest` and `$apply`, you may sometimes call `$digest` instead of `$apply` when you need that extra bit of performance.

Having covered `$digest` and `$apply` we have one more digest-triggering function to discuss, and that's `$evalAsync`. As it happens, it works just like `$apply` in that it schedules a digest on the root scope, not the scope being called. Expressing this as a unit test:

test/scope_spec.js

```
it("schedules a digest from root on $evalAsync", function(done) {
  var parent = new Scope();
  var child = parent.$new();
  var child2 = child.$new();

  parent.aValue = 'abc';
  parent.counter = 0;
  parent.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  child2.$evalAsync(function(scope) { });

  setTimeout(function() {
    expect(parent.counter).toBe(1);
    done();
  }, 50);
});
```

This test is very similar to the previous one: We check that calling `$evalAsync` on a scope causes a watch on its grandparent to execute.

Since we already have the root scope readily available, the change to `$evalAsync` is very simple. We just call `$$digest` on the root scope instead of `this`:

src/scope.js


```
Scope.prototype.$evalAsync = function(expr) {
  var self = this;
  if (!self.$$phase && !self.$$asyncQueue.length) {
    setTimeout(function() {
      if (self.$$asyncQueue.length) {
        self.$$root.$digest();
      }
    }, 0);
  }
  this.$$asyncQueue.push({scope: this, expression: expr});
};
```

Armed with the `$$root` attribute we can also now revisit `$$digestOnce` to really make sure we are always referring to the correct `$$lastDirtyWatch` for checking the state of the short-circuiting optimization. We should always refer to the `$$lastDirtyWatch` of root, no matter which scope `$digest` was called on:

src/scope.js

```
Scope.prototype.$$digestOnce = function() {
  var dirty;
  this.$$everyScope(function(scope) {
    var newValue, oldValue;
    _.forEachRight(scope.$$watchers, function(watcher) {
      try {
        if (watcher) {
          newValue = watcher.watchFn(scope);
          oldValue = watcher.last;
          if (!scope.$$areEqual(newValue, oldValue, watcher.valueEq)) {
            scope.$$root.$$lastDirtyWatch = watcher;
            watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
            watcher.listenerFn(newValue,
              (oldValue === initWatchVal ? newValue : oldValue),
              scope);
            dirty = true;
          } else if (scope.$$root.$$lastDirtyWatch === watcher) {
            dirty = false;
            return false;
          }
        }
      } catch (e) {
        console.error(e);
      }
    });
    return dirty !== false;
  });
  return dirty;
};
```

Isolated Scopes

We've seen how the relationship between a parent scope and a child scope is very intimate when prototypal inheritance is involved. Whatever attributes the parent has, the child can access. If they happen to be object or array attributes the child can also change their contents.

Sometimes we don't want quite this much intimacy. At times it would be convenient to have a scope be a part of the scope hierarchy, but not give it access to everything its parents contain. This is what *isolated scopes* are for.

The idea behind scope isolation is simple: We make a scope that's part of the scope hierarchy just like we've seen before, but we do *not* make it prototypally inherit from its parent. It is cut off – or isolated – from its parent's prototype chain.

An isolated scope can be created by passing a boolean value to the `$new` function. When it is `true` the scope will be isolated. When it is `false` (or omitted/`undefined`), prototypal inheritance will be used. When a scope is isolated, it doesn't have access to the attributes of its parent:

test/scope_spec.js

```
it("does not have access to parent attributes when isolated", function() {
  var parent = new Scope();
  var child = parent.$new(true);

  parent.aValue = 'abc';

  expect(child.aValue).toBeUndefined();
});
```

And since there is no access to the parent's attributes, there is of course no way to watch them either:

test/scope_spec.js

```
it("cannot watch parent attributes when isolated", function() {
  var parent = new Scope();
  var child = parent.$new(true);

  parent.aValue = 'abc';
  child.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.aValueWas = newValue;
    }
  );

  child.$digest();
  expect(child.aValueWas).toBeUndefined();
});
```


Scope isolation is set up in `$new`. Based on the boolean argument given, we either make a child scope as we've been doing so far, or create an independent scope using the `Scope` constructor. In both cases the new scope is added to the current scope's children:

src/scope.js

```
Scope.prototype.$new = function(isolated) {
  var child;
  if (isolated) {
    child = new Scope();
  } else {
    var ChildScope = function() { };
    ChildScope.prototype = this;
    child = new ChildScope();
  }
  this.$$children.push(child);
  child.$$watchers = [];
  child.$$children = [];
  return child;
};
```

> If you've used isolated scopes with Angular directives, you'll know that an isolated scope is usually not *completely* cut off from its parent. Instead you can explicitly define a mapping of attributes the scope will get from its parent.
> However, this mechanism is not built into scopes. It is part of the implementation of directives. We will return to this discussion when we implement isolation in directives.

Since we've now broken the prototypal inheritance chain, we need to revisit the discussions about `$digest`, `$apply`, and `$evalAsync` from earlier in this chapter.

Firstly, we want `$digest` to walk down the inheritance hierarchy. This one we're already handling, since we include isolated scopes in their parent's `$$children`. That means the following test also passes already:

test/scope_spec.js

```
it("digests its isolated children", function() {
  var parent = new Scope();
  var child = parent.$new(true);

  child.aValue = 'abc';
  child.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.aValueWas = newValue;
    }
```



```
  );

  parent.$digest();
  expect(child.aValueWas).toBe('abc');
});
```

In the case of `$apply` and `$evalAsync` we're not quite there yet. We wanted those operations to begin digestion from the root, but isolated scopes in the middle of the hierarchy break this assumption, as the following two failing test cases illustrate:

test/scope_spec.js

```
it("digests from root on $apply when isolated", function() {
  var parent = new Scope();
  var child = parent.$new(true);
  var child2 = child.$new();

  parent.aValue = 'abc';
  parent.counter = 0;
  parent.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  child2.$apply(function(scope) { });

  expect(parent.counter).toBe(1);
});

it("schedules a digest from root on $evalAsync when isolated", function(done) {
  var parent = new Scope();
  var child = parent.$new(true);
  var child2 = child.$new();

  parent.aValue = 'abc';
  parent.counter = 0;
  parent.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  child2.$evalAsync(function(scope) { });

  setTimeout(function() {
    expect(parent.counter).toBe(1);
```



```
    done();
  }, 50);
});
```

Notice that these are basically the same test cases we wrote earlier when discussing `$apply`, and `$evalAsync`, only in this case we make one of the scopes an isolated one.

The reason the tests fail is that we're relying on the `$$root` attribute to point to the root of the hierarchy. Non-isolated scopes have that attribute inherited from the actual root. Isolated scopes do not. In fact, since we use the `Scope` constructor to create isolated scopes, and that constructor assigns `$$root`, each isolated scope has a `$$root` attribute that points to *itself*. This is not what we want.

The fix is simple enough. All we need to do is to modify `$new` to reassign `$$root` to the actual root scope. While we're at it, let's also reassign the `$$lastDirtyWatch` attribute that has to do with the digest short-circuiting optimization. We want to apply the optimization to the whole hierarchy:

src/scope.js

```
Scope.prototype.$new = function(isolated) {
  var child;
  if (isolated) {
    child = new Scope();
    child.$$root = this.$$root;
    child.$$lastDirtyWatch = this.$$lastDirtyWatch;
  } else {
    var ChildScope = function() { };
    ChildScope.prototype = this;
    child = new ChildScope();
  }
  this.$$children.push(child);
  child.$$watchers = [];
  child.$$children = [];
  return child;
};
```

Before we've got everything about inheritance covered, there's one more thing we need to fix in the context of isolated scopes, and that is the queues in which we store `$evalAsync` and `$$postDigest` functions. Recall that we drain the `$$asyncQueue` and the `$$postDigestQueue` in `$digest`, and we do it without taking any extra measures related to child or parent scopes. We simply assume there is one instance of each queue that represents all the queued tasks in the whole hierarchy.

For non-isolated scopes this is exactly the case: Whenever we access one of the queues from any scope, we're accessing the same queue because it's inherited by every scope. Not so for isolated scopes. Just like `$$root` earlier, `$$asyncQueue` and `$$postDigestQueue` are *shadowed* in isolated scopes by local versions created in the `Scope` constructor. This has the unfortunate effect that a function scheduled on an isolated scope with `$evalAsync` or `$$postDigest` is never executed:

test/scope_spec.js


```
it("executes $evalAsync functions on isolated scopes", function(done) {
  var parent = new Scope();
  var child = parent.$new(true);

  child.$evalAsync(function(scope) {
    scope.didEvalAsync = true;
  });

  setTimeout(function() {
    expect(child.didEvalAsync).toBe(true);
    done();
  }, 50);
});

it("executes $$postDigest functions on isolated scopes", function() {
  var parent = new Scope();
  var child = parent.$new(true);

  child.$$postDigest(function() {
    child.didPostDigest = true;
  });
  parent.$digest();

  expect(child.didPostDigest).toBe(true);
});
```

Just like with `$$root` (and `$$lastDirtyWatch`), what we want is each and every scope in the hierarchy to share the same copy of `$$asyncQueue` and `$$postDigestQueue`, regardless of whether they're isolated or not. When a scope is not isolated, they get a copy automatically. When it is isolated, we need to explicitly assign it:

src/scope.js

```
Scope.prototype.$new = function(isolated) {
  var child;
  if (isolated) {
    child = new Scope();
    child.$$root = this.$$root;
    child.$$lastDirtyWatch = this.$$lastDirtyWatch;
    child.$$asyncQueue = this.$$asyncQueue;
    child.$$postDigestQueue = this.$$postDigestQueue;
  } else {
    var ChildScope = function() { };
    ChildScope.prototype = this;
    child = new ChildScope();
  }
  this.$$children.push(child);
  child.$$watchers = [];
  child.$$children = [];
  return child;
};
```


And finally we have everything set up properly!

Destroying Scopes

In the lifetime of a typical Angular application, page elements come and go as the user is presented with different views and data. This also means that the scope hierarchy grows and shrinks during the lifetime of the application, with controller and directive scopes being added and removed.

In our implementation we can create child scopes, but we don't have a mechanism for removing them yet. An ever-growing scope hierarchy is not very convenient when it comes to performance – not least because of all the watches that come with it! So we obviously need a way to destroy scopes.

Destroying a scope basically just means removing it from the `$$children` collection of its parent. It will thus no longer be digested recursively, making its watches effectively dead. And since the scope is no longer referenced, it will at some point just cease to exist as the garbage collector of the JavaScript environment reclaims it. (This of course only works as long as you don't have external references to the scope or its watches from within application code.)

Since destroying a scope just means removing it from its parent, destroying a root scope is a no-op – there's no parent to remove it from.

The destroy operation is implemented in a scope function called `$destroy`. When called, it destroys the scope:

test/scope_spec.js

```
it("is no longer digested when $destroy has been called", function() {
  var parent = new Scope();
  var child = parent.$new();

  child.aValue = [1, 2, 3];
  child.counter = 0;
  child.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    },
    true
  );

  parent.$digest();
  expect(child.counter).toBe(1);

  child.aValue.push(4);
  parent.$digest();
  expect(child.counter).toBe(2);
```

```
  child.$destroy();
  child.aValue.push(5);
  parent.$digest();
  expect(child.counter).toBe(2);
});
```

In `$destroy` we will need a reference to the scope's parent. We don't have one yet, so let's just add one to `$new`. Whenever a child scope is created, its direct parent will be assigned to the `$parent` attribute:

src/scope.js

```
Scope.prototype.$new = function(isolated) {
  var child;
  if (isolated) {
    child = new Scope();
    child.$$root = this.$$root;
    child.$$lastDirtyWatch = this.$$lastDirtyWatch;
    child.$$asyncQueue = this.$$asyncQueue;
    child.$$postDigestQueue = this.$$postDigestQueue;
  } else {
    var ChildScope = function() { };
    ChildScope.prototype = this;
    child = new ChildScope();
  }
  this.$$children.push(child);
  child.$$watchers = [];
  child.$$children = [];
  child.$parent = this;
  return child;
};
```

> Notice that `$parent` is prefixed with just one dollar sign instead of two. That means it's deemed by the makers of Angular to be something that's OK to reference from application code. However, using it is usually considered an anti-pattern because of the tight coupling between scopes that it introduces.

Now we're ready to implement `$destroy`. It will, as long as it's not called on the root scope, find the current scope from its parent's `$$children` array and then remove it:

src/scope.js

```
Scope.prototype.$destroy = function() {
  if (this === this.$$root) {
    return;
  }
```

```
  var siblings = this.$parent.$$children;
  var indexOfThis = siblings.indexOf(this);
  if (indexOfThis >= 0) {
    siblings.splice(indexOfThis, 1);
  }
};
```

Summary

In this chapter we've taken our Scope implementation from mere individual scope objects to whole hierarchies of interconnected scopes that inherit attributes from their parents. With this implementation we can support the kind of scope hierarchy that's built into every AngularJS application.

You have learned about:

- How child scopes are created.
- The relationship between scope inheritance and JavaScript's native prototypal inheritance.
- Attribute shadowing and its implications.
- Recursive digestion from a parent scope to its child scopes.
- The difference between `$digest` and `$apply` when it comes to the starting point of digestion.
- Isolated scopes and how they differ from normal child scopes.
- How child scopes are destroyed.

In the next chapter we'll cover one more thing related to watchers: Angular's built-in `$watchCollection` mechanism for efficiently watching for changes in objects and arrays.

Chapter 3

Watching Collections

In Chapter 1 we implemented two different strategies for identifying changes in watches: By reference and by value. The way you choose between the two is by passing a boolean flag to the `$watch` function.

In this chapter we are going implement the third and final strategy for identifying changes. This one you enable by registering your watch using a separate function, called `$watchCollection`.

The use case for `$watchCollection` is that you want to know when something in an array or an object has changed: When items or attributes have been added, removed, or reordered.

As we saw in Chapter 1, doing this is already possible by registering a value-based watch, by passing `true` as the third argument to `$watch`. That strategy, however, does way more work than is actually needed in our use case. It deep-watches the *whole object graph* that is reachable from the return value of the watch function. Not only does it notice when items are added or removed, but it also notices when anything *within* those items, at any depth, changes. This means it needs to also keep full deep copies of old values and inspect them to arbitrary depths during the digest.

`$watchCollection` is basically an optimization of this value-based version of `$watch` we already have. Because it only watches collections on the shallow level, it can get away with an implementation that's faster and uses less memory than full-blown deep watches do.

This chapter is all about `$watchCollection`. While conceptually simple, the function packs a punch. By knowing how it works you'll be able to use it to full effect. Implementing `$watchCollection` also serves as a case study for writing watches that specialize in certain kinds of data structures, which is something you may need to do when building Angular applications.

Setting Up The Infrastructure

Let's create a stub for the `$watchCollection` function on `Scope`.

The function's signature will be very similar to that of `$watch`: It takes a watch function that should return the value we want to watch, and a listener function that will be called

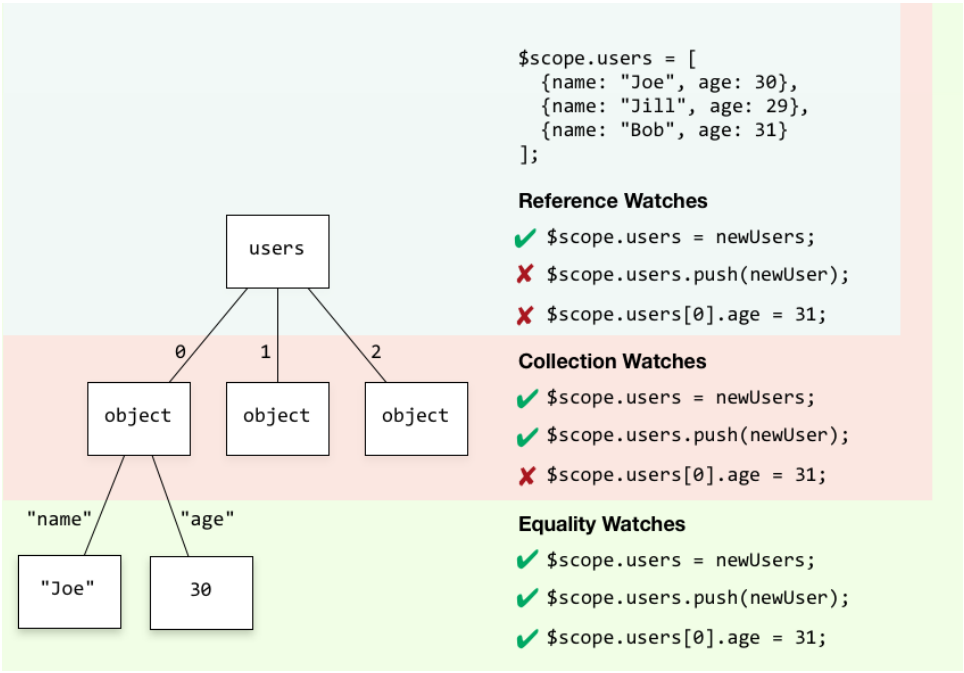

Figure 3.1: The three watch depths in Angular.js. Reference and equality/value-based watches were implemented in Chapter 1. Collection watches are the topic of this chapter.

when the watched value changes. Internally, the function *delegates* to the `$watch` function by supplying with its own, locally created versions of a watch function and a listener function:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {

  var internalWatchFn = function(scope) {
  };

  var internalListenerFn = function() {
  };

  return this.$watch(internalWatchFn, internalListenerFn);
};
```

As you may recall, the `$watch` function returns a function with which the watch can be removed. By returning that function directly to the original caller, we also enable this possibility for `$watchCollection`.

Let's also set up a `describe` block for our tests, in a similar fashion as we did in the previous chapter. It should be a nested `describe` block within the top-level `describe` block for `Scope`:

test/scope_spec.js

```
describe("$watchCollection", function() {

  var scope;

  beforeEach(function() {
    scope = new Scope();
  });

});
```

Detecting Non-Collection Changes

The purpose of `$watchCollection` is to watch arrays and objects. However, it does also work when the value returned by the watch function is a non-collection. In that case it falls back to working as if you'd just called `$watch` instead. While this is possibly the least useful aspect of `$watchCollection`, implementing it first will let us conveniently flesh out the function's structure.

Here's a test for verifying this basic behavior:

test/scope_spec.js


```
it("works like a normal watch for non-collections", function() {
  var valueProvided;

  scope.aValue = 42;
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      valueProvided = newValue;
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
  expect(valueProvided).toBe(scope.aValue);

  scope.aValue = 43;
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

We're using `$watchCollection` to watch a number on the scope. In the listener function we increment a counter, and also capture a local variable new value. We then assert that the watch cals the listener with the new value as a normal, non-collection watch would.

> We're ignoring the `oldValue` argument for now. It needs some special care in the context of `$watchCollection` and we will return to it later in this chapter.

During the watch function invocation, `$watchCollection` first invokes the *original* watch function to obtain the value we want to watch. It then checks for changes in that value compared to what it was previously and stores the value for the next digest cycle:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  var newValue;
  var oldValue;

  var internalWatchFn = function(scope) {
    newValue = watchFn(scope);

    // Check for changes

    oldValue = newValue;
```



```
  };

  var internalListenerFn = function() {
  };

  return this.$watch(internalWatchFn, internalListenerFn);
};
```

By keeping `newValue` and `oldValue` declarations outside of the internal watch function body, we can share them between the internal watch and listener functions. They will also persist between digest cycles in the closure formed by the `$watchCollection` function. This is particularly important for the old value, since we need to compare to it across digest cycles.

The listener function just delegates to the original listener function, passing it the new and old values, as well as the scope:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  var self = this;
  var newValue;
  var oldValue;

  var internalWatchFn = function(scope) {
    newValue = watchFn(scope);

    // Check for changes

    oldValue = newValue;
  };

  var internalListenerFn = function() {
    listenerFn(newValue, oldValue, self);
  };

  return this.$watch(internalWatchFn, internalListenerFn);
};
```

If you recall, the way `$digest` determines whether the listener should be called or not is by comparing successive return values of the watch function. Our internal watch function, however, is not currently returning anything, so the listener function will never be called.

What should the internal watch function return? Since nothing outside of `$watchCollection` will ever see it, it doesn't make that much difference. The only important thing is that it is *different between successive invocations* if there have been changes.

The way Angular implements this is by introducing an integer counter and incrementing it whenever a change is detected. Each watch registered with `$watchCollection` gets its own

counter that keeps incrementing for the lifetime of that watch. By then having the internal watch function return this counter, we ensure that the contract of the watch function is fulfilled.

In the non-collection case, we can just compare the new and old values by reference:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  var self = this;
  var newValue;
  var oldValue;
  var changeCount = 0;

  var internalWatchFn = function(scope) {
    newValue = watchFn(scope);

    if (newValue !== oldValue) {
      changeCount++;
    }
    oldValue = newValue;

    return changeCount;
  };

  var internalListenerFn = function() {
    listenerFn(newValue, oldValue, self);
  };

  return this.$watch(internalWatchFn, internalListenerFn);
};
```

With that, the non-collection test case passes. But what about if the non-collection value happens to be `NaN`?

test/scope_scope.js

```
it("works like a normal watch for NaNs", function() {
  scope.aValue = 0/0;
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
```



```
  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

The reason this test fails is the same as we had with `NaN` in Chapter 1: `NaN`s are not equal to each other. Instead of using `!==` for (in)equality comparison, let's just use the helper function `$$areEqual` since we already know how to handle `NaN`s there:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  var self = this;
  var newValue;
  var oldValue;
  var changeCount = 0;

  var internalWatchFn = function(scope) {
    newValue = watchFn(scope);

    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;

    return changeCount;
  };

  var internalListenerFn = function() {
    listenerFn(newValue, oldValue, self);
  };

  return this.$watch(internalWatchFn, internalListenerFn);
};
```

The final argument, `false` explicitly tells `$$areEqual` not to use value comparison. In this case we just want to compare references.

Now we have the basic structure for `$watchCollection` in place. We can turn our attention to collection change detection.

Detecting New Arrays

The internal watch function will have two top-level conditional branches: One that deals with objects and one that deals with things other than objects. Since JavaScript arrays are objects, they will also be handled in the first branch. Within that branch we'll have two nested branches: One for arrays and one for other objects.

For now we can just determine the object-ness or array-ness of the value by using Lo-Dash's `_.isObject` and `_.isArray` functions:

src/scope.js

```
var internalWatchFn = function(scope) {
  newValue = watchFn(scope);

  if (_.isObject(newValue)) {
    if (_.isArray(newValue)) {

    } else {

    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

The first thing we can check if we have an array value is to see if the value was also an array previously. If it wasn't, it has obviously changed:

src/scope.js

```
it("notices when the value becomes an array", function() {
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.arr; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.arr = [1, 2, 3];
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```


Since our array-branch in the internal watch function is currently empty, we don't notice this change. Let's change that by simply checking the type of the old value:

src/scope.js

```
var internalWatchFn = function(scope) {
  newValue = watchFn(scope);

  if (_.isObject(newValue)) {
    if (_.isArray(newValue)) {
      if (!_.isArray(oldValue)) {
        changeCount++;
        oldValue = [];
      }
    } else {

    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

If the old value isn't an array, we record a change. We also initialize the old value as an empty array. In a moment we will start to mirror the contents of the array we're watching in this internal array, but for now this passes our test.

Detecting New Or Removed Items in Arrays

The next thing that may happen with an array is that its length may change when items are added or removed. Let's add a couple of tests for that, one for each operation:

test/scope_spec.js

```
it("notices an item added to an array", function() {
  scope.arr = [1, 2, 3];
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.arr; },
    function(newValue, oldValue, scope) {
      scope.counter++;
```

```
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.arr.push(4);
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});

it("notices an item removed from an array", function() {
  scope.arr = [1, 2, 3];
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.arr; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.arr.shift();
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

In both cases we have an array that we're watching on the scope and we manipulate the contents of that array. We then check that the changes are picked up during the next digest.

These kinds of changes can be detected by simply looking at the length of the array and comparing it to the length of the old array. We must also sync the new length to our internal `oldValue` array, which we do by assigning its length:

src/scope.js

```
var internalWatchFn = function(scope) {
  newValue = watchFn(scope);

  if (_.isObject(newValue)) {
```

```
    if (_.isArray(newValue)) {
      if (!_.isArray(oldValue)) {
        changeCount++;
        oldValue = [];
      }
      if (newValue.length !== oldValue.length) {
        changeCount++;
        oldValue.length = newValue.length;
      }
    } else {

    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

Detecting Replaced or Reordered Items in Arrays

There's one more kind of change that we must detect with arrays, and that's when items are replaced or reordered without the array's length changing:

test/scope_spec.js

```
it("notices an item replaced in an array", function() {
  scope.arr = [1, 2, 3];
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.arr; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.arr[1] = 42;
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
```



```
  expect(scope.counter).toBe(2);
});

it("notices items reordered in an array", function() {
  scope.arr = [2, 1, 3];
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.arr; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.arr.sort();
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

To detect changes like this, we actually need to iterate over the array, at each index comparing the items in the old and new arrays. Doing that will pick up both replaced values and reordered values. While we're iterating, we also sync up the internal `oldValue` array with the new array's contents:

src/scope.js

```
var internalWatchFn = function(scope) {
  newValue = watchFn(scope);

  if (_.isObject(newValue)) {
    if (_.isArray(newValue)) {
      if (!_.isArray(oldValue)) {
        changeCount++;
        oldValue = [];
      }
      if (newValue.length !== oldValue.length) {
        changeCount++;
        oldValue.length = newValue.length;
      }
      _.forEach(newValue, function(newItem, i) {
        if (newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
```



```
        }
      });
    } else {

    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

We iterate the new array with the `_.forEach` function from Lo-Dash. It provides us with both the item and the index for each iteration. We use the index to get the corresponding item from the old array.

With this implementation we can detect any changes that might happen to an array, without actually having to copy or even traverse any nested data structures *within* that array.

Array-Like Objects

We've got arrays covered but there's one more special case we need to think about.

In addition to proper arrays – things that inherit the `Array` prototype – JavaScript environments have a few objects that behave like arrays without actually being arrays. Angular's `$watchCollection` treats these kinds of objects as arrays, so we will also want to do so.

One array-like object is the `arguments` local variable that every function has, and that contains the arguments given to that function invocation. Let's check whether we currently support that by adding a test case.

test/scope_spec.js

```
it("notices an item replaced in an arguments object", function() {
  (function() {
    scope.arrayLike = arguments;
  })(1, 2, 3);
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.arrayLike; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
```

```
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.arrayLike[1] = 42;
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

We construct an anonymous function that we immediately call with a few arguments, and store those arguments on the scope. That gives us the array-like `arguments` object. We then check whether changes in that object are picked up by our `$watchCollection` implementation.

Another array-like object is the DOM `NodeList`, which you get from certain operations on the DOM, such as `querySelectorAll` and `getElementsByTagName`. Let's test that too.

test/scope_spec.js

```
it("notices an item replaced in a NodeList object", function() {
  document.documentElement.appendChild(document.createElement('div'));
  scope.arrayLike = document.getElementsByTagName('div');

  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.arrayLike; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  document.documentElement.appendChild(document.createElement('div'));
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

We first add a `<div>` element to the DOM, and then obtain a `NodeList` object by calling `getElementsByTagName` on `document`. We place the list on the scope and attach a watch for it. When we want to cause a change to the list, we just append another `<div>` to the DOM. Being a so-called live collection, the list will immediately be augmented with the new element. We check that our `$watchCollection` also detects this.

As it turns out, both of these test cases fail. The problem is with the Lo-Dash `_.isArray` function, which only checks for proper arrays and not other kinds of array-like objects. We need to create our own predicate function that is more appropriate for our use case.

Let's put this predicate in a new source file, which we'll use for these kinds of generic helper functions. Create a file called `Angular.js` in the `src` directory. Then add the following contents to it:

src/Angular.js

```
/* jshint globalstrict: true */
'use strict';

_.mixin({
  isArrayLike: function(obj) {
    if (_.isNull(obj) || _.isUndefined(obj)) {
      return false;
    }
    var length = obj.length;
    return _.isNumber(length);
  }
});
```

We use the Lo-Dash `_.mixin` function to extend the Lo-Dash library with an object containing our own functions. For now we just add one, called `isArrayLike`, that takes an object and returns a boolean value indicating whether that object is array-like.

We check for array-likeness by just checking that the object exists and that it has a `length` attribute with a numeric value. This isn't perfect, but it'll do for now.

> AngularJS also has a `src/Angular.js` file for generic internal utilities. The main difference to our version is that it does not use Lo-Dash because Angular doesn't include Lo-Dash.

Now all we need to do is call this new predicate instead of `_isArray` in `$watchCollection`:

src/scope.js

```
var internalWatchFn = function(scope) {
  newValue = watchFn(scope);

  if (_.isObject(newValue)) {
    if (_.isArrayLike(newValue)) {
      if (!_.isArray(oldValue)) {
        changeCount++;
```



```
        oldValue = [];
      }
      if (newValue.length !== oldValue.length) {
        changeCount++;
        oldValue.length = newValue.length;
      }
      _.forEach(newValue, function(newItem, i) {
        if (newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
        }
      });
    } else {

    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

Note that while we deal with any kind of array-like objects, the internal `oldValue` array is always a proper array, not any other array-like object.

> Strings also match our definition of array-likeness since they have a `length` attribute and provide indexed attributes for individual characters. However, a JavaScript `String` is not a JavaScript `Object`, so our outer `_.isObject` guard prevents it from being treated as a collection. That means `$watchCollection` treats Strings as non-collections.
> Since JavaScript Strings are immutable and you cannot change their contents, watching them as collections would not be very useful anyway.

Detecting New Objects

Let's turn our attention to objects, or more precisely, objects *other than* arrays and array-like objects. This basically means dictionaries such as:

```
{
  aKey: 'aValue',
  anotherKey: 42
}
```

The way we detect changes in objects will be similar to what we just did with arrays. The implementation for objects will be simplified a bit by the fact that there are no "object-like objects" to complicate things. On the other hand, we will need to do a bit more work in the change detection since objects don't have the handy `length` property that we used with arrays.

To begin with, just like with arrays, let's make sure we're covering the case where a value becomes an object when it previously wasn't one:

test/scope_spec.js

```
it("notices when the value becomes an object", function() {
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.obj; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.obj = {a: 1};
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

We can handle this case in the same manner as we did with arrays. If the old value wasn't an object, make it one and record a change:

src/scope.js

```
var internalWatchFn = function(scope) {
  newValue = watchFn(scope);

  if (_.isObject(newValue)) {
    if (_.isArrayLike(newValue)) {
      if (!_.isArray(oldValue)) {
        changeCount++;
        oldValue = [];
      }
      if (newValue.length !== oldValue.length) {
        changeCount++;
        oldValue.length = newValue.length;
      }
```



```
      _.forEach(newValue, function(newItem, i) {
        if (newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
        }
      });
    } else {
      if (!_.isObject(oldValue) || _.isArrayLike(oldValue)) {
        changeCount++;
        oldValue = {};
      }
    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

Note that since arrays are also objects, we can't just check the old value with `_.isObject`. We also need to exclude arrays and array-like objects with `_.isArrayLike`.

Detecting New Or Replaced Attributes in Objects

We want a new attribute added to an object to trigger a change:

test/scope_spec.js

```
it("notices when an attribute is added to an object", function() {
  scope.counter = 0;
  scope.obj = {a: 1};

  scope.$watchCollection(
    function(scope) { return scope.obj; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.obj.b = 2;
  scope.$digest();
  expect(scope.counter).toBe(2);
```



```
  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

We also want to trigger a change when the value of an existing attribute changes:

test/scope_spec.js

```
it("notices when an attribute is changed in an object", function() {
  scope.counter = 0;
  scope.obj = {a: 1};

  scope.$watchCollection(
    function(scope) { return scope.obj; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.obj.a = 2;
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

Both of these cases can be dealt with in the same way. We will iterate over all the attributes in the new object, and check whether they have the same values in the old object:

src/scope.js

```
var internalWatchFn = function(scope) {
  var key;
  newValue = watchFn(scope);

  if (_.isObject(newValue)) {
    if (_.isArrayLike(newValue)) {
      if (!_.isArray(oldValue)) {
        changeCount++;
        oldValue = [];
      }
      if (newValue.length !== oldValue.length) {
        changeCount++;
```

```
        oldValue.length = newValue.length;
      }
      _.forEach(newValue, function(newItem, i) {
        if (newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
        }
      });
    } else {
      if (!_.isObject(oldValue) || _.isArrayLike(oldValue)) {
        changeCount++;
        oldValue = {};
      }
      for (key in newValue) {
        if (newValue.hasOwnProperty(key)) {
          if (oldValue[key] !== newValue[key]) {
            changeCount++;
            oldValue[key] = newValue[key];
          }
        }
      }
    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

While we iterate, we also sync the old object with the attributes of the new object, so that we have them for the next digest.

> The `hasOwnProperty` clause in the `for` loop is a common JavaScript idiom for checking that a property is attached to the object itself instead of being inherited through the prototype chain. `$watchCollection` does not watch inherited properties in objects.

Detecting Removed Attributes in Objects

The remaining operation to discuss in the context of objects is the removal of attributes:

test/scope_spec.js

```
it("notices when an attribute is removed from an object", function() {
  scope.counter = 0;
  scope.obj = {a: 1};

  scope.$watchCollection(
    function(scope) { return scope.obj; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  delete scope.obj.a;
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

With arrays we were able to handle removed items by just truncating our internal array (by assigning its `length`) and then iterating over both arrays simultaneously, checking that all items are the same. With objects we cannot do this. To check whether attributes have been removed from an object, we need a second loop. This time we'll loop over the attributes of the *old* object and see if they're still present in the new one. If they're not, they no longer exist and we also remove them from our internal object:

src/scope.js

```
var internalWatchFn = function(scope) {
  var key;
  newValue = watchFn(scope);

  if (_.isObject(newValue)) {
    if (_.isArrayLike(newValue)) {
      if (!_.isArray(oldValue)) {
        changeCount++;
        oldValue = [];
      }
      if (newValue.length !== oldValue.length) {
        changeCount++;
        oldValue.length = newValue.length;
      }
      _.forEach(newValue, function(newItem, i) {
        if (newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
```



```
        }
      });
    } else {
      if (!_.isObject(oldValue) || _.isArrayLike(oldValue)) {
        changeCount++;
        oldValue = {};
      }
      for (key in newValue) {
        if (newValue.hasOwnProperty(key)) {
          if (oldValue[key] !== newValue[key]) {
            changeCount++;
            oldValue[key] = newValue[key];
          }
        }
      }
      for (key in oldValue) {
        if (oldValue.hasOwnProperty(key) && !newValue.hasOwnProperty(key)) {
          changeCount++;
          delete oldValue[key];
        }
      }
    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

Preventing Unnecessary Object Iteration

We are now iterating over object keys twice. For very large objects this can be expensive. Since we're working within a watch function that gets executed in every single digest, we need to take care not to do too much work.

For this reason we will apply one important optimization to the object change detection.

Firstly, we are going to keep track of the sizes of the old and new objects:

- For the old object, we keep a variable around that we increment whenever an attribute is added and decrement whenever an attribute is removed.
- For the new object, we calculate its size during the first loop in the internal watch function.

By the time we're done with the first loop we know the current sizes of the two objects. Then, we only launch into the second loop if the size of the old collection is larger than the

size of the new one. If the sizes are equal, there cannot have been any removals and we can skip the second loop entirely. Here's the final `$wathCollection` implementation after we've applied this optimization:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  var self = this;
  var newValue;
  var oldValue;
  var oldLength;
  var changeCount = 0;

  var internalWatchFn = function(scope) {
    var newLength, key;
    newValue = watchFn(scope);

    if (_.isObject(newValue)) {
      if (_.isArrayLike(newValue)) {
        if (!_.isArray(oldValue)) {
          changeCount++;
          oldValue = [];
        }
        if (newValue.length !== oldValue.length) {
          changeCount++;
          oldValue.length = newValue.length;
        }
        _.forEach(newValue, function(newItem, i) {
          if (newItem !== oldValue[i]) {
            changeCount++;
            oldValue[i] = newItem;
          }
        });
      } else {
        if (!_.isObject(oldValue) || _.isArrayLike(oldValue)) {
          changeCount++;
          oldValue = {};
          oldLength = 0;
        }
        newLength = 0;
        for (key in newValue) {
          if (newValue.hasOwnProperty(key)) {
            newLength++;
            if (oldValue.hasOwnProperty(key)) {
              if (oldValue[key] !== newValue[key]) {
                changeCount++;
                oldValue[key] = newValue[key];
              }
            } else {
              changeCount++;
              oldLength++;
```



```
              oldValue[key] = newValue[key];
            }
          }
        }
        if (oldLength > newLength) {
          changeCount++;
          for (key in oldValue) {
            if (oldValue.hasOwnProperty(key) && !newValue.hasOwnProperty(key)) {
              oldLength--;
              delete oldValue[key];
            }
          }
        }
      }
    } else {
      if (!self.$$areEqual(newValue, oldValue, false)) {
        changeCount++;
      }
      oldValue = newValue;
    }

    return changeCount;
  };

  var internalListenerFn = function() {
    listenerFn(newValue, oldValue, self);
  };

  return this.$watch(internalWatchFn, internalListenerFn);
};
```

Note that we now have to handle new and changed attributes differently, since with new attributes we need to increment the `oldLength` variable.

Dealing with Objects that Have A length

We're almost done covering different kinds collections, but there's still one special kind of object that we'd better consider.

Recall that we determine the array-likeness of an object by checking whether it has a numeric `length` attribute. How, then, do we handle the following object?

```
{
  length: 42,
  otherKey: 'abc'
}
```


This is not an array-like object. It just happens to have an attribute called `length`. Since it is not difficult to think of a situation where such an object might exist in an application, we need to deal with this.

Let's add a test where we check that changes in an object that happens to have a `length` property are actually detected:

test/scope_spec.js

```
it("does not consider any object with a length property an array", function() {
  scope.obj = {length: 42, otherKey: 'abc'};
  scope.counter = 0;

  scope.$watchCollection(
    function(scope) { return scope.obj; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();

  scope.obj.newKey = 'def';
  scope.$digest();

  expect(scope.counter).toBe(2);
});
```

When you run this test, you'll see that the listener is not invoked after there's been a change in the object. That's because we've decided it's an array because it has a `length`, and the array change detection doesn't notice the new object key.

The fix for this is simple enough. Instead of considering *all* objects with a numeric `length` property as array-like, let's narrow it down to objects with a numeric `length` property *and* with a numeric property for the key one smaller than the length. So for example, if the object has a `length` with 42, there must also be the attribute `41` in it. Arrays and array-like objects pass that requirement.

This only works for non-zero lengths though, so we need to relax the condition when the length is zero:

src/Angular.js

```
_.mixin({
  isArrayLike: function(obj) {
    if (_.isNull(obj) || _.isUndefined(obj)) {
      return false;
    }
    var length = obj.length;
    return length === 0 ||
      (_.isNumber(length) && length > 0 && (length - 1) in obj);
  }
});
```


This makes our test pass, and does indeed work for *most* objects. The check isn't foolproof, but it is the best we can practically do.

Handing The Old Collection Value To Listeners

The contract of the watch listener function is that it gets three arguments: The new value of the watch function, the previous value of the watch function, and the scope. In this chapter we have respected that contract by providing those values, but the way we have done it is problematic, especially when it comes to the previous value.

The problem is that since we are maintaining the old value in `internalWatchFn`, it will already have been updated to the new value by the time we call the listener function. The values given to the listener function are always identical. This is the case for non-collections:

test/scope_spec.js

```
it("gives the old non-collection value to listeners", function() {
  scope.aValue = 42;
  var oldValueGiven;

  scope.$watchCollection(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      oldValueGiven = oldValue;
    }
  );

  scope.$digest();

  scope.aValue = 43;
  scope.$digest();

  expect(oldValueGiven).toBe(42);
});
```

It is the case for arrays:

test/scope_spec.js

```
it("gives the old array value to listeners", function() {
  scope.aValue = [1, 2, 3];
  var oldValueGiven;

  scope.$watchCollection(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
```

```
      oldValueGiven = oldValue;
    }
  );

  scope.$digest();

  scope.aValue.push(4);
  scope.$digest();

  expect(oldValueGiven).toEqual([1, 2, 3]);
});
```

And it is the case for objects:

test/scope_spec.js

```
it("gives the old object value to listeners", function() {
  scope.aValue = {a: 1, b: 2};
  var oldValueGiven;

  scope.$watchCollection(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      oldValueGiven = oldValue;
    }
  );

  scope.$digest();

  scope.aValue.c = 3;
  scope.$digest();

  expect(oldValueGiven).toEqual({a: 1, b: 2});
});
```

The implementation for the value comparison and copying works well and efficiently for the change detection itself, so we don't really want to change it. Instead, we'll introduce *another* variable that we'll keep around between digest iterations. We'll call it `veryOldValue`, and it will hold a copy of the old collection value that we will *not* change in `internalWatchFn`.

Maintaining `veryOldValue` requires copying arrays or objects, which is expensive. We've gone through great lengths in order to *not* copy full collections each time in collection watches. So we really only want to maintain `veryOldValue` if we actually have to. We can check that by seeing if the listener function given by the user actually takes at least two arguments:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  var self = this;
  var newValue;
  var oldValue;
  var oldLength;
  var veryOldValue;
  var trackVeryOldValue = (listenerFn.length > 1);
  var changeCount = 0;

  // ...

};
```

The `length` property of `Function` contains the number of declared arguments in the function. If there's more than one, i.e. (`newValue`, `oldValue`), or (`newValue`, `oldValue`, `scope`), only then do we enable the tracking of the very old value.

Note that this means you won't incur the cost of copying the very old value in `$watchCollection` unless you declare `oldvalue` in your listener function arguments. It also means that you can't just reflectively look up the old value from your listener's `arguments` object. You'll need to actually declare it.

The rest of the work happens in `internalListenerFn`. Instead of handing `oldValue` to the listener, it should hand `veryOldValue`. It then needs to copy the *next* `veryOldValue` from the current value, so that it can be given to the listener next time. We can use `_.clone` to get a shallow copy of the collection, and it'll also work with primitives:

src/scope.js

```
var internalListenerFn = function() {
  listenerFn(newValue, veryOldValue, self);

  if (trackVeryOldValue) {
    veryOldValue = _.clone(newValue);
  }
};
```

In Chapter 1 we discussed the role of `oldValue` on the first invocation to a listener function. For that invocation it should be identical to the new value. That should hold true for `$watchCollection` listeners too:

test/scope_spec.js

```
it("uses the new value as the old value on first digest", function() {
  scope.aValue = {a: 1, b: 2};
  var oldValueGiven;

  scope.$watchCollection(
    function(scope) { return scope.aValue; },
```

```
    function(newValue, oldValue, scope) {
      oldValueGiven = oldValue;
    }
  );

  scope.$digest();

  expect(oldValueGiven).toEqual({a: 1, b: 2});
});
```

The test does not pass since our old value is actually `undefined`, since we've never assigned anything to `veryOldValue` before the first invocation of the listener.

We need to set a boolean flag denoting whether we're on the first invocation, and call the listener differently based on that:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  var self = this;
  var newValue;
  var oldValue;
  var oldLength;
  var veryOldValue;
  var trackVeryOldValue = (listenerFn.length > 1);
  var changeCount = 0;
  var firstRun = true;

  // ...

  var internalListenerFn = function() {
    if (firstRun) {
      listenerFn(newValue, newValue, self);
      firstRun = false;
    } else {
      listenerFn(newValue, veryOldValue, self);
    }

    if (trackVeryOldValue) {
      veryOldValue = _.clone(newValue);
    }
  };

  return this.$watch(internalWatchFn, internalListenerFn);
};
```

Summary

In this chapter we've added the third and final dirty-checking mechanism to our implementation of `Scope`: Shallow collection-watching.

The `$watchCollection` function is not simple, but that's mostly because it provides an important, non-trivial facility: We can watch for changes in large arrays and objects much more efficiently than we could with just deep-watching.

You have learned about:

- How `$watchCollection` can be used with arrays, objects, and other values.
- What `$watchCollection` does with arrays.
- What `$watchCollection` does with objects.
- Array-like objects and their role in `$watchCollection`.

The next chapter concludes our implementation of scopes. We will add the other main functional area that scopes provide in addition to dirty-checking: Events.

Chapter 4

Scope Events

In the preceding chapters we have implemented almost everything that Angular scopes do, including watches and the digest cycle. In this final chapter on scopes we are going to implement the scope event system, which completes the picture.

As you'll see, the event system has actually very little to do with the digest system, so you could say that scope objects provide two unrelated pieces of functionality. The reason it is useful to have the event system on scopes is the structure that the scope hierarchy forms. The scope tree that stems from the root scope forms a structural hierarchy baked into each Angular application. Propagating events through this hierarchy provides natural channels of communication.

Publish-Subscribe Messaging

The scope event system is basically an implementation of the widely used *publish-subscribe* messaging pattern: When something significant happens you can *publish* that information on the scope as an *event*. Other parts of the application may have *subscribed* to receive that event, in which case they will get notified. As a publisher you don't know how many, if any, subscribers are receiving the event. As a subscriber, you don't really know where an event comes from. The scope acts as a mediator, decoupling publishers from subscribers.

This pattern has a long history, and has also been widely employed in JavaScript applications. For example, jQuery provides a custom event system, as does Backbone.js. Both of them can be used for pub/sub messaging.

Angular's implementation of pub/sub is in many ways similar to other implementations, but has one key difference: The Angular event system is baked into the scope hierarchy. Rather than having a single point through which all events flow, we have the scope tree where events may propagate up and down.

When you publish an event on a scope, you choose between two propagation modes: Up the scope hierarchy or down the scope hierarchy. When you go up, subscribers on the current and its ancestor scopes get notified. This is called *emitting* an event. When go

down, subscribers on the current scope and its descendant scopes get notified. This is called *broadcasting* an event.

In this chapter we'll implement both of these propagation models. The two are actually so similar that we'll implement them in tandem. This will also highlight the differences they do have.

> The scope event system deals with application-level events - events that you publish as an application developer. It does not propagate native DOM events that the browser originates, such as clicks or resizes. For dealing with native events there is `angular.element`, which we will encounter later in the book.

Setup

We will still be working on the scope objects, so all the code will go into `src/scope.js` and `test/scope_spec.js`. Let's begin by putting in place a new nested `describe` block for our tests inside the outermost `describe` block.

test/scope_spec.js

```
describe("Events", function() {

  var parent;
  var scope;
  var child;
  var isolatedChild;

  beforeEach(function() {
    parent = new Scope();
    scope = parent.$new();
    child = scope.$new();
    isolatedChild = scope.$new(true);
  });

});
```

What we're about implement has a lot to do with scope inheritance hierarchies, so for convenience we're setting one up in a `beforeEach` function. It has a scope with a parent and two children, one of them isolated. This should cover everything we need to test about inheritance.

Registering Event Listeners: $on

To get notified about an event you need to register to get notified. In AngularJS the registration is done by calling the function `$on` on a Scope object. The function takes two

arguments: The name of the event of interest, and the listener function that will get called when that event occurs.

> The AngularJS term for the suscribers in pub/sub is *listeners*, so that is the term we will also be using from now on.

Listeners registered through `$on` will receive both emitted and broadcasted events. There is, in fact, no way to limit the events received by anything else than the event name.

What should the `$on` function actually do? Well, it should store the listener somewhere so that it can find it later when events are fired. For storage we'll put an object in the attribute `$$listeners` (the double-dollar denoting that it should be considered private). The object's keys will be event names, and the values will be arrays holding the listener functions registered for a particular event. So, we can test that listeners registered with `$on` are stored accordingly:

test/scope_spec.js

```
it("allows registering listeners", function() {
  var listener1 = function() { };
  var listener2 = function() { };
  var listener3 = function() { };

  scope.$on('someEvent', listener1);
  scope.$on('someEvent', listener2);
  scope.$on('someOtherEvent', listener3);

  expect(scope.$$listeners).toEqual({
    someEvent: [listener1, listener2],
    someOtherEvent: [listener3]
  });

});
```

We will need the `$$listeners` object to exist on the scope. Let's set one up in the constructor:

src/scope.js

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$postDigestQueue = [];
  this.$$root = this;
  this.$$children = [];
  this.$$listeners = {};
  this.$$phase = null;
}
```


The `$on` function should check whether we already have a listener collection for the event given, and initialize one if not. It can then just push the new listener function to the collection:

src/scope.js

```
Scope.prototype.$on = function(eventName, listener) {
  var listeners = this.$$listeners[eventName];
  if (!listeners) {
    this.$$listeners[eventName] = listeners = [];
  }
  listeners.push(listener);
};
```

Since the location of each listener in the scope hierarchy is significant, we do have a slight problem with the current implementation of `$$listeners`: All listeners in the whole scope hierarchy will go into the same `$$listeners` collection. Instead, what we need is a *separate* `$$listeners` collection for each scope:

test/scope_spec.js

```
it("registers different listeners for every scope", function() {
  var listener1 = function() { };
  var listener2 = function() { };
  var listener3 = function() { };

  scope.$on('someEvent', listener1);
  child.$on('someEvent', listener2);
  isolatedChild.$on('someEvent', listener3);

  expect(scope.$$listeners).toEqual({someEvent: [listener1]});
  expect(child.$$listeners).toEqual({someEvent: [listener2]});
  expect(isolatedChild.$$listeners).toEqual({someEvent: [listener3]});
});
```

This test fails because both `scope` and `child` actually have a reference to *the same* `$$listeners` collection and `isolatedChild` doesn't have one at all. We need to tweak the child scope constructor to explicitly give each new child scope its own `$$listeners` collection. For a non-isolated scope it will shadow the one in its parent. This is exactly the same solution as we used for `$$watchers` in Chapter 2:

test/scope_spec.js


```
Scope.prototype.$new = function(isolated) {
  var child;
  if (isolated) {
    child = new Scope();
    child.$$root = this.$$root;
    child.$$lastDirtyWatch = this.$$lastDirtyWatch;
    child.$$asyncQueue = this.$$asyncQueue;
    child.$$postDigestQueue = this.$$postDigestQueue;
  } else {
    var ChildScope = function() { };
    ChildScope.prototype = this;
    child = new ChildScope();
  }
  this.$$children.push(child);
  child.$$watchers = [];
  child.$$listeners = {};
  child.$$children = [];
  child.$parent = this;
  return child;
};
```

The basics of $emit and $broadcast

Now that we have listeners registered, we can put them to use and fire events. As we've discussed, there are two functions for doing that: `$emit` and `$broadcast`.

The basic functionality of both functions is that when you call them with an event name as an argument, they will call all the listeners that have been registered for that event name. Correspondingly, of course, they do *not* call listeners for other event names:

test/scope_spec.js

```
it("calls the listeners of the matching event on $emit", function() {
  var listener1 = jasmine.createSpy();
  var listener2 = jasmine.createSpy();
  scope.$on('someEvent', listener1);
  scope.$on('someOtherEvent', listener2);

  scope.$emit('someEvent');

  expect(listener1).toHaveBeenCalled();
  expect(listener2).not.toHaveBeenCalled();
});

it("calls the listeners of the matching event on $broadcast", function() {
  var listener1 = jasmine.createSpy();
  var listener2 = jasmine.createSpy();
  scope.$on('someEvent', listener1);
```



```
  scope.$on('someOtherEvent', listener2);

  scope.$broadcast('someEvent');

  expect(listener1).toHaveBeenCalled();
  expect(listener2).not.toHaveBeenCalled();
});
```

> We're using Jasmine's *spy functions* to represent our listener functions. They are special stub functions that do nothing but record whether they have been called or not, and what arguments they've been called with. With spies we can very conveniently check what the scope is doing with our listeners.
> If you've used mock objects or other kinds of test doubles before, spies should look familiar. They might just as well be called mock functions.

These tests can be made pass by introducing the `$emit` and `$broadcast` functions that, for now, behave identically. They find the listeners for the event name, and call each of them in succession:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var listeners = this.$$listeners[eventName] || [];
  _.forEach(listeners, function(listener) {
    listener();
  });
};

Scope.prototype.$broadcast = function(eventName) {
  var listeners = this.$$listeners[eventName] || [];
  _.forEach(listeners, function(listener) {
    listener();
  });
};
```

Dealing with Duplication

We've defined two almost identical test cases and two identical functions. It's apparent there's going to be a lot of similarity between the two event propagation mechanisms. Let's meet this duplication head-on before we go any further, so we won't end up having to write everything twice.

For the event functions themselves we can extract the common behavior, which is the delivery of the event, to a function that both `$emit` and `$broadcast` use. Let's call it `$$fireEventOnScope`:

src/scope.js


```
Scope.prototype.$emit = function(eventName) {
  this.$$fireEventOnScope(eventName);
};

Scope.prototype.$broadcast = function(eventName) {
  this.$$fireEventOnScope(eventName);
};

Scope.prototype.$$fireEventOnScope = function(eventName) {
  var listeners = this.$$listeners[eventName] || [];
  _.forEach(listeners, function(listener) {
    listener();
  });
};
```

> The original AngularJS does not have a `$$fireEventOnScope` function. Instead it just duplicates the code of the common behavior between `$emit` and `$broadcast`.

That's much better. But we can go one further and also eliminate duplication in the test suite. We can wrap the test cases that describe common functionality to a loop which runs once for `$emit` and once for `$broadcast`. Within the loop body we can dynamically look the correct function up. Replace the two test cases we added earlier with this:

test/scope_spec.js

```
_.forEach(['$emit', '$broadcast'], function(method) {

  it("calls listeners registered for matching events on "+method, function() {
    var listener1 = jasmine.createSpy();
    var listener2 = jasmine.createSpy();
    scope.$on('someEvent', listener1);
    scope.$on('someOtherEvent', listener2);

    scope[method]('someEvent');

    expect(listener1).toHaveBeenCalled();
    expect(listener2).not.toHaveBeenCalled();
  });

});
```

Since Jasmine's `describe` blocks are just functions, we can run arbitrary code in them. Our loop effectively defines two test cases for each `it` block within it.

Event Objects

We're currently calling the listeners without any arguments, but that isn't quite how Angular works. What we should do instead is pass the listeners an *event object*.

The event objects used by scopes are regular JavaScript objects that carry information and behavior related to the event. We'll attach several attributes to the event, but to begin with, each event has a `name` attribute with the name of the event. Here's a test case for it (or rather, two test cases):

test/scope_spec.js

```
it("passes an event object with a name to listeners on "+method, function() {
  var listener = jasmine.createSpy();
  scope.$on('someEvent', listener);

  scope[method]('someEvent');

  expect(listener).toHaveBeenCalled();
  expect(listener.calls.mostRecent().args[0].name).toEqual('someEvent');
});
```

> The `calls.mostRecent()` function of a Jasmine spy contains information about the last time that spy was called. It has an `args` attribute containing an array of the arguments that were passed to the function.

An important aspect of event objects is that *the same exact event object is passed to each listener*. Application developers attach additional attributes on it for communicating extra information between listeners. This is significant enough to warrant its own unit test(s):

test/scope_spec.js

```
it("passes the same event object to each listener on "+method, function() {
  var listener1 = jasmine.createSpy();
  var listener2 = jasmine.createSpy();
  scope.$on('someEvent', listener1);
  scope.$on('someEvent', listener2);

  scope[method]('someEvent');

  var event1 = listener1.calls.mostRecent().args[0];
  var event2 = listener2.calls.mostRecent().args[0];

  expect(event1).toBe(event2);
});
```

We can construct this event object in the `$$fireEventOnScope` function and pass it to the listeners:

src/scope.js


```
Scope.prototype.$$fireEventOnScope = function(eventName) {
  var event = {name: eventName};
  var listeners = this.$$listeners[eventName] || [];
  _.forEach(listeners, function(listener) {
    listener(event);
  });
};
```

Additional Listener Arguments

When you emit or broadcast an event, an event name by itself isn't always enough to communicate everything about what's happening. It's very common to associate *additional arguments* with the event. You can do that by just adding any number of arguments after the event name:

```
aScope.$emit('eventName', 'and', 'additional', 'arguments');
```

We need to pass these arguments on to the listener functions. They should receive them, correspondingly, as additional arguments after the event object. This is true for both `$emit` and `$broadcast`:

test/scope_spec.js

```
it("passes additional arguments to listeners on "+method, function() {
  var listener = jasmine.createSpy();
  scope.$on('someEvent', listener);

  scope[method]('someEvent', 'and', ['additional', 'arguments'], '...');

  expect(listener.calls.mostRecent().args[1]).toEqual('and');
  expect(listener.calls.mostRecent().args[2]).toEqual(['additional', 'arguments']);
  expect(listener.calls.mostRecent().args[3]).toEqual('...');
});
```

In both `$emit` and `$broadcast`, we'll grab whatever additional arguments were given to the function and pass them along to `$$fireEventOnScope`. We can get the additional arguments by calling the Lo-Dash `_.rest` function with the `arguments` object, which gives us an array of all the function's arguments except the first one:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var additionalArgs = _.rest(arguments);
  this.$$fireEventOnScope(eventName, additionalArgs);
};

Scope.prototype.$broadcast = function(eventName) {
  var additionalArgs = _.rest(arguments);
  this.$$fireEventOnScope(eventName, additionalArgs);
};
```


In `$$fireEventOnScope` we cannot simply pass the additional arguments on to the listeners. That's because the listeners are not expecting the additional arguments as a single array. They are expecting them as regular arguments to the function. Thus, we need to `apply` the listener functions with an array that contains both the event object and the additional arguments:

src/scope.js

```
Scope.prototype.$$fireEventOnScope = function(eventName, additionalArgs) {
  var event = {name: eventName};
  var listenerArgs = [event].concat(additionalArgs);
  var listeners = this.$$listeners[eventName] || [];
  _.forEach(listeners, function(listener) {
    listener.apply(null, listenerArgs);
  });
};
```

That gives us the desired behaviour.

Returning The Event Object

An additional characteristic that both `$emit` and `$broadcast` have is that they return the event object that they construct, so that the originator of the event can inspect its state after it has finished propagating:

test/scope_spec.js

```
it("returns the event object on "+method, function() {
  var returnedEvent = scope[method]('someEvent');

  expect(returnedEvent).toBeDefined();
  expect(returnedEvent.name).toEqual('someEvent');
});
```

The implementation is trivial - we just return the event object:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var additionalArgs = _.rest(arguments);
  return this.$$fireEventOnScope(eventName, additionalArgs);
};

Scope.prototype.$broadcast = function(eventName) {
```

```
  var additionalArgs = _.rest(arguments);
  return this.$$fireEventOnScope(eventName, additionalArgs);
};

Scope.prototype.$$fireEventOnScope = function(eventName, additionalArgs) {
  var event = {name: eventName};
  var listenerArgs = [event].concat(additionalArgs);
  var listeners = this.$$listeners[eventName] || [];
  _.forEach(listeners, function(listener) {
    listener.apply(null, listenerArgs);
  });
  return event;
};
```

Deregistering Event Listeners

Before we get into the differences between `$emit` and `$broadcast`, let's get one more important common requirement out of the way: You should be able to not only register event listeners, but also deregister them.

The mechanism for deregistering an event listener is the same as deregistering a watch, which we implemented back in Chapter 1: The registration function returns a deregistration function. Once that deregistration function has been called, the listener no longer receives any events:

test/scope_spec.js

```
it("can be deregistered "+method, function() {
  var listener = jasmine.createSpy();
  var deregister = scope.$on('someEvent', listener);

  deregister();

  scope[method]('someEvent');

  expect(listener).not.toHaveBeenCalled();
});
```

A simple implementation of removal does exactly what we did with watches - just splices the listener away from the collection:

src/scope.js

```
Scope.prototype.$on = function(eventName, listener) {
  var listeners = this.$$listeners[eventName];
  if (!listeners) {
    this.$$listeners[eventName] = listeners = [];
```



```
  }
  listeners.push(listener);
  return function() {
    var index = listeners.indexOf(listener);
    if (index >= 0) {
      listeners.splice(index, 1);
    }
  };
};
```

There is one special case we must be careful with, however: It is very common that a listener removes *itself* when it gets fired, for example when the purpose is to invoke a listener just once. This kind of removal happens *while we are iterating the listeners array*. The result is that the iteration jumps over one listener - the one immediately after the removed listener:

test/scope_spec.js

```
it("does not skip the next listener when removed on "+method, function() {
  var deregister;

  var listener = function() {
    deregister();
  };
  var nextListener = jasmine.createSpy();

  deregister = scope.$on('someEvent', listener);
  scope.$on('someEvent', nextListener);

  scope[method]('someEvent');

  expect(nextListener).toHaveBeenCalled();
});
```

What this means is that we can't just go and remove the listener directly. What we can do instead is to *replace* the listener with something indicating it has been removed. `null` will do just fine for this purpose:

src/scope.js

```
Scope.prototype.$on = function(eventName, listener) {
  var listeners = this.$$listeners[eventName];
  if (!listeners) {
    this.$$listeners[eventName] = listeners = [];
  }
  listeners.push(listener);
  return function() {
    var index = listeners.indexOf(listener);
```

```
    if (index >= 0) {
      listeners[index] = null;
    }
  };
};
```

Then, in the listener iteration, we can check for listeners that are `null` and take corrective action. We do need to switch from using `_.forEach` to a manual `while` loop to make this work:

src/scope.js

```
Scope.prototype.$$fireEventOnScope = function(eventName, additionalArgs) {
  var event = {name: eventName};
  var listenerArgs = [event].concat(additionalArgs);
  var listeners = this.$$listeners[eventName] || [];
  var i = 0;
  while (i < listeners.length) {
    if (listeners[i] === null) {
      listeners.splice(i, 1);
    } else {
      listeners[i].apply(null, listenerArgs);
      i++;
    }
  }
  return event;
};
```

Emitting Up The Scope Hierarchy

Now we finally get to the part that's *different* between `$emit` and `$broadcast`: The direction in which events travel in the scope hierarchy.

When you *emit* an event, that event gets passed to its listeners on the current scope, and then up the scope hierarchy, to its listeners on each scope up to and including the root.

The test for this should go *outside* the `forEach` loop we created earlier, because it concerns `$emit` only:

test/scope_spec.js

```
it("propagates up the scope hierarchy on $emit", function() {
  var parentListener = jasmine.createSpy();
  var scopeListener = jasmine.createSpy();

  parent.$on('someEvent', parentListener);
  scope.$on('someEvent', scopeListener);
```



```
  scope.$emit('someEvent');

  expect(scopeListener).toHaveBeenCalled();
  expect(parentListener).toHaveBeenCalled();
});
```

Let's try to implement this as simply as possible, by looping up the scopes in `$emit`. We can get to each scope's parent using the `$parent` attribute introduced in Chapter 2:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var additionalArgs = _.rest(arguments);
  var scope = this;
  do {
    scope.$$fireEventOnScope(eventName, additionalArgs);
    scope = scope.$parent;
  } while (scope);
};
```

This works, almost. We've now broken the contract about returning the event object to the caller of `$emit`, but also recall that we discussed the importance of passing the *same* event object to every listener. This requirement holds across scopes as well, but we're failing the following test for it:

test/scope_spec.js

```
it("propagates the same event up on $emit", function() {
  var parentListener = jasmine.createSpy();
  var scopeListener = jasmine.createSpy();

  parent.$on('someEvent', parentListener);
  scope.$on('someEvent', scopeListener);

  scope.$emit('someEvent');

  var scopeEvent = scopeListener.calls.mostRecent().args[0];
  var parentEvent = parentListener.calls.mostRecent().args[0];
  expect(scopeEvent).toBe(parentEvent);
});
```

This means we'll need to undo some of that duplication-removal we did earlier, and actually construct the event object in `$emit` and `$broadcast` both, and then pass it to `$$fireEventOnScope`. While we're at it, let's pull the whole `listenerArgs` construction out of `$$fireEventOnScope`. That way we won't have to construct it again for each scope:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var event = {name: eventName};
  var listenerArgs = [event].concat(_.rest(arguments));
  var scope = this;
  do {
    scope.$$fireEventOnScope(eventName, listenerArgs);
    scope = scope.$parent;
  } while (scope);
  return event;
};

Scope.prototype.$broadcast = function(eventName) {
  var event = {name: eventName};
  var listenerArgs = [event].concat(_.rest(arguments));
  this.$$fireEventOnScope(eventName, listenerArgs);
  return event;
};

Scope.prototype.$$fireEventOnScope = function(eventName, listenerArgs) {
  var listeners = this.$$listeners[eventName] || [];
  var i = 0;
  while (i < listeners.length) {
    if (listeners[i] === null) {
      listeners.splice(i, 1);
    } else {
      listeners[i].apply(null, listenerArgs);
      i++;
    }
  }
};
```

We've introduced a bit of duplication here, but nothing too bad. It'll actually come in handy as `$emit` and `$broadcast` start to diverge more.

Broadcasting Down The Scope Hierarchy

`$broadcast` is basically the opposite of `$emit`: It invokes the listeners on the current scope and all of its direct and indirect descendant scopes - isolated or not:

test/scope_spec.js

```
it("propagates down the scope hierarchy on $broadcast", function() {
  var scopeListener = jasmine.createSpy();
  var childListener = jasmine.createSpy();
  var isolatedChildListener = jasmine.createSpy();

  scope.$on('someEvent', scopeListener);
  child.$on('someEvent', childListener);
```

```
  isolatedChild.$on('someEvent', isolatedChildListener);

  scope.$broadcast('someEvent');

  expect(scopeListener).toHaveBeenCalled();
  expect(childListener).toHaveBeenCalled();
  expect(isolatedChildListener).toHaveBeenCalled();
});
```

Just for completeness, let's also make sure that `$broadcast` propagates one and the same event object to all listeners:

test/scope_spec.js

```
it("propagates the same event down on $broadcast", function() {
  var scopeListener = jasmine.createSpy();
  var childListener = jasmine.createSpy();

  scope.$on('someEvent', scopeListener);
  child.$on('someEvent', childListener);

  scope.$broadcast('someEvent');

  var scopeEvent = scopeListener.calls.mostRecent().args[0];
  var childEvent = childListener.calls.mostRecent().args[0];
  expect(scopeEvent).toBe(childEvent);
});
```

Iterating scopes on `$broadcast` isn't quite as straightforward as it was on `$emit`, since there isn't a direct path down. Instead the scopes diverge in a tree structure. What we need to do is traverse the tree. More precisely, we need the same kind of depth-first traversal of the tree we already have in `$$digestOnce`. Indeed, we can just reuse the `$$everyScope` function introduced in Chapter 2:

src/scope.js

```
Scope.prototype.$broadcast = function(eventName) {
  var event = {name: eventName};
  var listenerArgs = [event].concat(_.rest(arguments));
  this.$$everyScope(function(scope) {
    scope.$$fireEventOnScope(eventName, listenerArgs);
    return true;
  });
  return event;
};
```

It's apparent why broadcasting an event is potentially much more expensive than emitting one: Emitting goes up the hierarchy in a straight path, and scope hierarchies don't usually get that deep. Broadcasting, on the other hand, traverses across the tree as well. Broadcasting from the root scope will visit *each and every scope in your application.*

Including The Current And Target Scopes in The Event Object

At the moment our event object contains just one attribute: The event name. Next, we're going to bundle some more information on it.

If you're familiar with DOM events in the browser, you'll know that they come with a couple of useful attributes: `target`, which identifies the DOM element on which the event occurred, and `currentTarget`, which identifies the DOM element on which the event handler was attached. Since DOM events propagate up and down the DOM tree, the two may be different.

Angular scope events have an analogous pair of attributes: `targetScope` identifies the scope on which the event occurred, and `currentScope` identifies the scope on which the listener was attached. And, since scope events propagate up and down the scope tree, these two may also be different.

	Event originated in	Listener attached in
DOM Events	`target`	`currentTarget`
Scope Events	`targetScope`	`currentScope`

Beginning with `targetScope`, the idea is that it points to the same scope, no matter which listener is currently handling the event. For `$emit`:

test/scope_spec.js

```
it("attaches targetScope on $emit", function() {
  var scopeListener = jasmine.createSpy();
  var parentListener = jasmine.createSpy();

  scope.$on('someEvent', scopeListener);
  parent.$on('someEvent', parentListener);

  scope.$emit('someEvent');

  expect(scopeListener.calls.mostRecent().args[0].targetScope).toBe(scope);
  expect(parentListener.calls.mostRecent().args[0].targetScope).toBe(scope);
});
```

And for `$broadcast`:

test/scope_spec.js

```
it("attaches targetScope on $broadcast", function() {
  var scopeListener = jasmine.createSpy();
  var childListener = jasmine.createSpy();
```



```
  scope.$on('someEvent', scopeListener);
  child.$on('someEvent', childListener);

  scope.$broadcast('someEvent');

  expect(scopeListener.calls.mostRecent().args[0].targetScope).toBe(scope);
  expect(childListener.calls.mostRecent().args[0].targetScope).toBe(scope);
});
```

To make these tests pass, all we need to do is, in both `$emit` and `$broadcast`, attach `this` to the event object as the target scope:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var event = {name: eventName, targetScope: this};
  var listenerArgs = [event].concat(_.rest(arguments));
  var scope = this;
  do {
    scope.$$fireEventOnScope(eventName, listenerArgs);
    scope = scope.$parent;
  } while (scope);
  return event;
};

Scope.prototype.$broadcast = function(eventName) {
  var event = {name: eventName, targetScope: this};
  var listenerArgs = [event].concat(_.rest(arguments));
  this.$$everyScope(function(scope) {
    scope.$$fireEventOnScope(eventName, listenerArgs);
    return true;
  });
  return event;
};
```

Conversely, `currentScope` should *differ* based on what scope the listener was attached to. It should point to exactly that scope.

In this case we can't use Jasmine spies for testing because with spies we can only verify invocations after the fact. The `currentScope` is mutating during the scope traversal, so we have to record its *momentary value* when a listener is called. We can do that with our own listener functions and local variables:

For `$emit`:

test/scope_spec.js

```
it("attaches currentScope on $emit", function() {
  var currentScopeOnScope, currentScopeOnParent;
  var scopeListener = function(event) {
    currentScopeOnScope = event.currentScope;
  };
  var parentListener = function(event) {
    currentScopeOnParent = event.currentScope;
  };

  scope.$on('someEvent', scopeListener);
  parent.$on('someEvent', parentListener);

  scope.$emit('someEvent');

  expect(currentScopeOnScope).toBe(scope);
  expect(currentScopeOnParent).toBe(parent);
});
```

And for `$broadcast`:

test/scope_spec.js

```
it("attaches currentScope on $broadcast", function() {
  var currentScopeOnScope, currentScopeOnChild;
  var scopeListener = function(event) {
    currentScopeOnScope = event.currentScope;
  };
  var childListener = function(event) {
    currentScopeOnChild = event.currentScope;
  };

  scope.$on('someEvent', scopeListener);
  child.$on('someEvent', childListener);

  scope.$broadcast('someEvent');

  expect(currentScopeOnScope).toBe(scope);
  expect(currentScopeOnChild).toBe(child);
});
```

Luckily the actual implementation is much more straightforward than the tests. All we need to do is attach the scope we're currently iterating on:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var event = {name: eventName, targetScope: this};
  var listenerArgs = [event].concat(_.rest(arguments));
  var scope = this;
  do {
    event.currentScope = scope;
    scope.$$fireEventOnScope(eventName, listenerArgs);
    scope = scope.$parent;
  } while (scope);
  return event;
};

Scope.prototype.$broadcast = function(eventName) {
  var event = {name: eventName, targetScope: this};
  var listenerArgs = [event].concat(_.rest(arguments));
  this.$$everyScope(function(scope) {
    event.currentScope = scope;
    scope.$$fireEventOnScope(eventName, listenerArgs);
    return true;
  });
  return event;
};
```

Now event listeners can make decisions based on where in the scope hierarchy events are coming from and where they're being listened to.

Stopping Event Propagation

Another feature DOM events have, and one very commonly used, is *stopping* them from propagating further. DOM event objects have a function called `stopPropagation` for this purpose. It can be used in situations where you have, say, click handlers on multiple levels of the DOM, and don't want to trigger all of them for a particular event.

Scope events also have a `stopPropagation` method, but *only when they've been emitted*. Broadcasted events cannot be stopped. (This further emphasizes the fact that broadcasting is expensive).

What this means is that when you emit an event, and one of its listeners stops its propagation, listeners on parent scopes will never see that event:

test/scope_spec.js

```
it("does not propagate to parents when stopped", function() {
  var scopeListener = function(event) {
    event.stopPropagation();
  };
  var parentListener = jasmine.createSpy();
```

```
  scope.$on('someEvent', scopeListener);
  parent.$on('someEvent', parentListener);

  scope.$emit('someEvent');

  expect(parentListener).not.toHaveBeenCalled();
});
```

So the event does not go to parents but, crucially, it does still get passed to all remaining listeners *on the current scope*. It is only the propagation to *parent scopes* that is stopped:

test/scope_spec.js

```
it("is received by listeners on current scope after being stopped", function() {
  var listener1 = function(event) {
    event.stopPropagation();
  };
  var listener2 = jasmine.createSpy();

  scope.$on('someEvent', listener1);
  scope.$on('someEvent', listener2);

  scope.$emit('someEvent');

  expect(listener2).toHaveBeenCalled();
});
```

The first thing we need is a boolean flag that signals whether someone has called `stopPropagation` or not. We can introduce that in the closure formed by `$emit`. Then we need the actual `stopPropagation` function itself, which gets attached to the event object. Finally, the `do..while` loop we have in `$emit` should check for the status of the flag before going up a level:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var propagationStopped = false;
  var event = {
    name: eventName,
    targetScope: this,
    stopPropagation: function() {
      propagationStopped = true;
    }
  };
  var listenerArgs = [event].concat(_.rest(arguments));
  var scope = this;
  do {
    event.currentScope = scope;
```



```
    scope.$$fireEventOnScope(eventName, listenerArgs);
    scope = scope.$parent;
  } while (scope && !propagationStopped);
  return event;
};
```

Preventing Default Event Behavior

In addition to `stopPropagation`, DOM events can also be cancelled in another way, and that is by preventing their "default behavior". DOM events have a function called `preventDefault` that does this. Its purpose is to prevent the effect that the event would have natively in the browser, but still letting all its listeners know about it. For example, when `preventDefault` is called on a click event fired on a hyperlink, the browser does not follow the hyperlink, but all click handlers are still invoked.

Scope events also have a `preventDefault` function. This is true for both emitted and broadcasted events. However, since scope events do not have any built-in "default behavior", calling the function has very little effect. It does one thing, which is to set a boolean flag called `defaultPrevented` on the event object. The flag does not alter the scope event system's behavior, but may be used by, say, custom directives to make decisions about whether or not they should trigger some default behavior once the event has finished propagating. The Angular `$locationService` does this when it broadcasts location events, as we'll see later in the book.

So, all we need to do is test that when a listener calls `preventDefault()` on the event object, its `defaultPrevented` flag gets set. This behavior is identical for both `$emit` and `$broadcast`, so add the following test to the loop in which we've added the common behaviors:

test/scope_spec.js

```
it("is sets defaultPrevented when preventDefault called on "+method, function() {
  var listener = function(event) {
    event.preventDefault();
  };
  scope.$on('someEvent', listener);

  var event = scope[method]('someEvent');

  expect(event.defaultPrevented).toBe(true);
});
```

The implementation here is similar to what we just did in `stopPropagation`: There's a function that sets a boolean flag attached to the event object. The difference is that this time the boolean flag is also attached to the event object, and that this time we don't make any decisions based on the value of the boolean flag. For `$emit`:

src/scope.js

```
Scope.prototype.$emit = function(eventName) {
  var propagationStopped = false;
  var event = {
    name: eventName,
    targetScope: this,
    stopPropagation: function() {
      propagationStopped = true;
    },
    preventDefault: function() {
      event.defaultPrevented = true;
    }
  };
  var listenerArgs = [event].concat(_.rest(arguments));
  var scope = this;
  do {
    event.currentScope = scope;
    scope.$$fireEventOnScope(eventName, listenerArgs);
    scope = scope.$parent;
  } while (scope && !propagationStopped);
  return event;
};
```

And for `$broadcast`:

src/scope.js

```
Scope.prototype.$broadcast = function(eventName) {
  var event = {
    name: eventName,
    targetScope: this,
    preventDefault: function() {
      event.defaultPrevented = true;
    }
  };
  var listenerArgs = [event].concat(_.rest(arguments));
  this.$$everyScope(function(scope) {
    event.currentScope = scope;
    scope.$$fireEventOnScope(eventName, listenerArgs);
    return true;
  });
  return event;
};
```

Broadcasting Scope Removal

Sometimes it is useful to know when a scope is removed. A typical use case for this is in directives, where you might set up DOM listeners and other references which should be

cleaned up when the directive's element gets destroyed. The solution to this is listening to an event named `$destroy` on the directive's scope. (Notice the dollar sign on the event name. It indicates it is an event coming from the Angular framework rather than application code.)

Where does this `$destroy` event come from? Well, we should fire it when a scope gets removed, which is when someone calls the `$destroy` function on it:

test/scope_spec.js

```
it("fires $destroy when destroyed", function() {
  var listener = jasmine.createSpy();
  scope.$on('$destroy', listener);

  scope.$destroy();

  expect(listener).toHaveBeenCalled();
});
```

When a scope gets removed, all of its child scopes get removed too. Their listeners should also receive the `$destroy` event:

test/scope_spec.js

```
it("fires $destroy on children destroyed", function() {
  var listener = jasmine.createSpy();
  child.$on('$destroy', listener);

  scope.$destroy();

  expect(listener).toHaveBeenCalled();
});
```

How do we make this work? Well, we have exactly the function needed to fire an event on a scope and its children: `$broadcast`. We should use it to broadcast the `$destroy` event from the `$destroy` function:

src/scope.js

```
Scope.prototype.$destroy = function() {
  if (this === this.$$root) {
    return;
  }
  var siblings = this.$parent.$$children;
  var indexOfThis = siblings.indexOf(this);
  if (indexOfThis >= 0) {
    this.$broadcast('$destroy');
    siblings.splice(indexOfThis, 1);
  }
};
```


Handling Exceptions

There's just one thing remaining that we need to do, and that is to deal with the unfortunate fact that exceptions occur. When a listener function does something that causes it to throw an exception, that should not mean that the event stops propagating. Our current implementation does not only that, but also actually causes the exception to be thrown all the way out to the caller of `$emit` or `$broadcast`. That means these test cases (defined inside the `forEach` loop for `$emit` and `$broadcast`) currently throw exceptions:

test/scope_spec.js

```
it("does not stop on exceptions on "+method, function() {
  var listener1 = function(event) {
    throw 'listener1 throwing an exception';
  };
  var listener2 = jasmine.createSpy();
  scope.$on('someEvent', listener1);
  scope.$on('someEvent', listener2);

  scope[method]('someEvent');

  expect(listener2).toHaveBeenCalled();
});
```

Just like with watch functions, `$evalAsync` functions, and `$$postDigest` functions, we need to wrap each listener invocation in a `try..catch` clause and handle the exception. For now the handling means merely logging it to the console, but at a later point we'll actually forward it to a special exception handler service:

src/scope.js

```
Scope.prototype.$$fireEventOnScope = function(eventName, listenerArgs) {
  var listeners = this.$$listeners[eventName] || [];
  var i = 0;
  while (i < listeners.length) {
    if (listeners[i] === null) {
      listeners.splice(i, 1);
    } else {
      try {
        listeners[i].apply(null, listenerArgs);
      } catch (e) {
        console.error(e);
      }
      i++;
    }
  }
};
```


Summary

We've now implemented the Angular scope event system in full, and with that we have a fully featured Scope implementation! Everything that Angular's scopes do, our scopes now do too.

In this chapter you've learned:

- How Angular's event system builds on the classic pub/sub pattern.
- How event listeners are registered on scopes
- How events are fired on scopes
- What the differences between `$emit` and `$broadcast` are
- What the contents of scope event objects are
- How some of the scope attributes are modeled after the DOM event model
- When and how scope events can be stopped

In future chapters we will integrate the scope implementation to other parts of Angular as we build them. In the next chapter we will begin this work by starting to look at expressions and the Angular expression language. Expressions are intimately tied to scope watches and enable a much more succint way to express the thing you want to watch. With expressions, instead of:

```
function(scope) { return scope.aValue; }
```

you can just say:

```
'aValue'
```


Part II

Expressions And Filters

If scopes are the core of Angular's change detection system, then expressions are a way to gain frictionless access to that core: They allow us to concisely access and manipulate data on scopes and run computation on it.

We can use expressions in JavaScript application code, often to great effect, but that's not the main use case for them. The real value of expressions is in allowing us to bind data and behavior to HTML markup. We use expressions when providing attributes to directives like `ngClass` or `ngClick`, and we use them when binding data to the contents and attributes of DOM elements with the "mustache" operator `{{ }}`.

This approach of attaching behavior to HTML isn't free of controversy, and to many people it is uncomfortably reminiscent of the `onClick="javascript:doStuff()"` style of JavaScript we used in the early days of the dynamic web. Fortunately, as we will see, there are some key differences between Angular expressions and arbitrary JavaScript code that greatly diminishes this problem.

In this part of the book we will implement Angular expressions. In the process, you will learn in great detail what you can and cannot do with them and how they work their magic.

Closely connected to expressions are Angular *filters* - those functions we run by adding the Unix-style pipe character `'|'` to expressions to modify their return values. We will see how filters are constructed, how they can be invoked from JavaScript, and how they integrate with Angular expressions.

A Whole New Language

So what exactly are Angular expressions? Aren't they just pieces of JavaScript you sprinkle in your HTML markup? That's close, but not quite true.

Angular expressions are custom-designed to access and manipulate data on scope objects, and to *not* do much else. Expressions are definitely modeled very closely on JavaScript, and as Miško Hevery has pointed out, you could implement much of the expression system with just a few lines of JavaScript:

```
function parse(expr) {
  return function(scope) {
    with (scope) {
      return eval(expr);
    }
  }
}
```

This function takes an expression string and returns a function. That function executes the expression by evaluating it as JavaScript code. It also sets the context of the code to be a scope object using the JavaScript `with` statement.

This implementation of expressions is problematic because it uses some iffy JavaScript facilities: The use of both `eval` and `with` is frowned upon, and `with` is actually forbidden in ECMAScript 5 strict mode.

But the main problem is that this implementation doesn't get us all the way there. While Angular expressions are *almost* pure JavaScript, they also extend it: The filter expressions

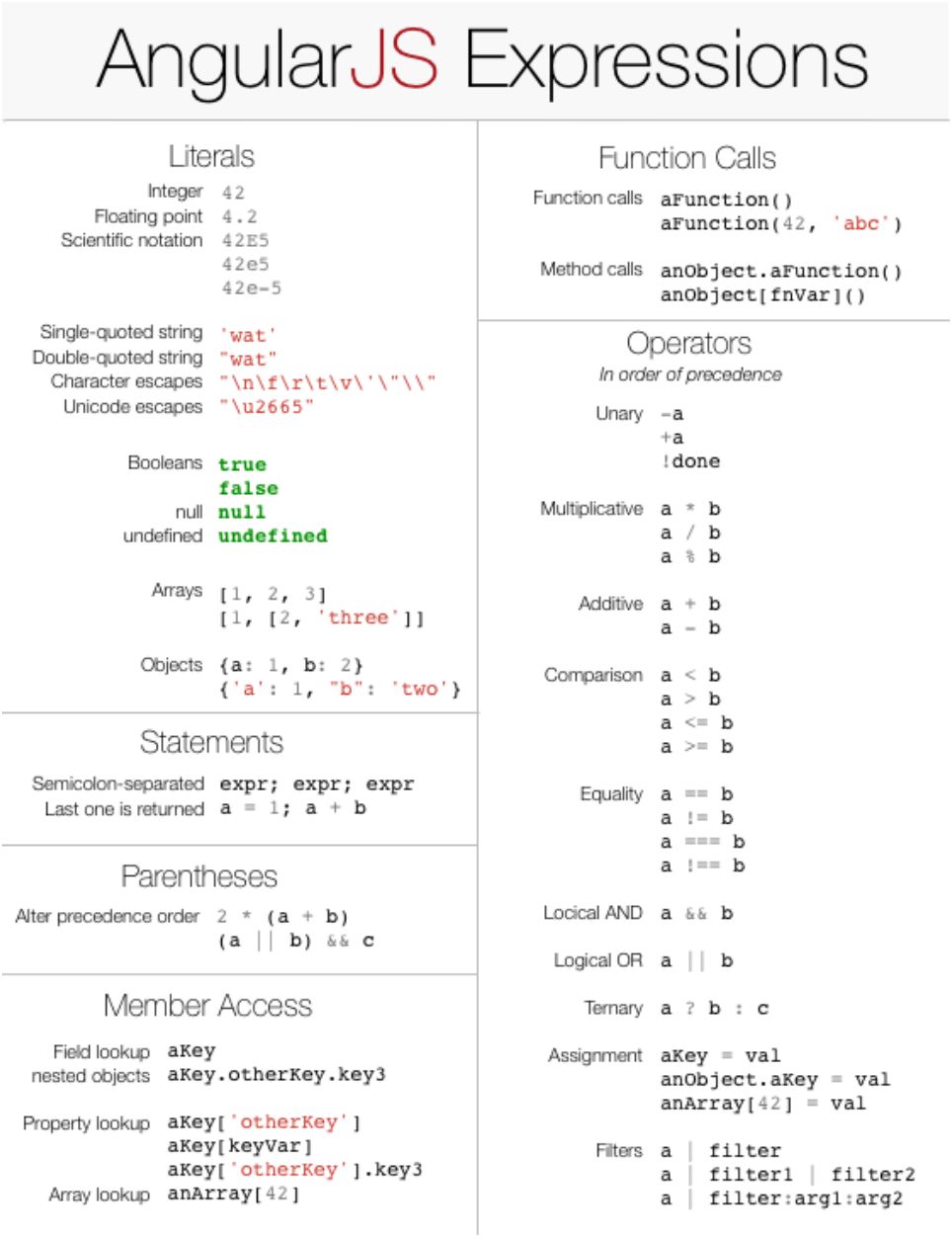

Figure 4.1: A cheatsheet of the full capabilities of the Angular expression language. PDF at http://teropa.info/blog/2014/03/23/angularjs-expressions-cheatsheet.html

that use the pipe character `|` are not valid JavaScript, and as such cannot be processed by `eval`.

There are also some serious security considerations with parsing and executing arbitrary JavaScript code using `eval`. This is particularly true when the code originates in HTML, where you typically include user-generated content. This opens up a whopping new vector for cross-site scripting attacks. That is why Angular expressions are constrained to execute in the context of a scope, and not a global object like `window`. Angular also does its best to prevent you from doing anything dangerous with expressions, as we will see.

So, if expressions aren't JavaScript, what are they? You could say they are a whole new programming language. A JavaScript derivative that's optimized for short expressions, removes most control flow statements, and adds filtering support.

Starting in the following chapter we are going to implement this programming language. It involves taking strings that represent expressions and returning JavaScript functions that execute the computation in those expressions. This will bring us to the world of parsers and lexers - the tools of language designers.

What We Will Skip

There are a few things the Angular expression implementation includes that we will omit in the interest of staying closer to the essence of expressions:

- Angular expressions support *promise unwrapping*, where you can refer to `$q` promises in expressions and Angular will resolve those promises asynchronously behind the scenes. This feature is deprecated in AngularJS 1.2, and will be removed at some point. We will just ignore it.
- The Angular expression parser does a lot of work so that application programmers can get clear error messages when parsing goes wrong. This involves a lot of bookkeeping related to the locations of characters and tokens in the input strings. We will skip most of that bookkeeping, making our implementation simpler at the cost of having error messages that aren't quite so user friendly.
- The expression parser supports the HTML Content Security Policy, and modifies its behavior slightly when there is one present. We will skip those modifications at this point and defer CSP discussion to an appendix.

Chapter 5

Literal Expressions

To begin our implementation of the Angular expression parser, let's make a version that can parse *literal* expressions - expressions that represent themselves, like `"42"`, and unlike `fourtyTwo`.

Literal expressions are the simplest kinds of expressions to parse. By adding support for them first we can flesh out the structure and dynamics of all the pieces that make up expression parsing.

Setup

The code that parses Angular expressions will go into a new file called `src/parse.js`, named after the `$parse` service that it will provide.

In that file, there will be one external-facing function, called `parse`. It takes an Angular expression string and returns a function that executes that expression in a certain context:

src/parse.js

```
/* jshint globalstrict: true */
'use strict';

function parse(expr) {
  // return ...
}
```

Internally the file will use two objects that do all the work: A *Lexer* and a *Parser*. They have distinct responsibilities in different phases of the job:

- The *lexer* takes the original expression string and returns a collection of *tokens* parsed from that string. For example, the string `"a + b"` would result in tokens for `a`, `+`, and `b`.

- The *parser* takes the token collection produced by the lexer and returns a function that evaluates the expression in a given context. For example, the tokens for `a`, `+`, and `b` would result in a function that looks up `a` and `b` from a context and returns their sum.

Internally, `Lexer` is a constructor function. Objects constructed with it have a method called `lex` that executes the tokenization:

src/parse.js

```
function Lexer() {
}

Lexer.prototype.lex = function(text) {

};
```

`Parser` is another constructor function. It takes a lexer object as an argument. Objects constructed with it have a method called `parse` that invokes the lexer and then does work with the resulting tokens:

src/parse.js

```
function Parser(lexer) {
  this.lexer = lexer;
}

Parser.prototype.parse = function(text) {
  this.tokens = this.lexer.lex(text);
};
```

The main `parse` function just wires these two together and runs them:

src/parse.js

```
function parse(expr) {
  var lexer = new Lexer();
  var parser = new Parser(lexer);
  return parser.parse(expr);
}
```

This is the high-level structure of `parse.js`. In the rest of this and the following few chapters we'll populate the `Lexer` and the `Parser` with the functionality that makes the magic happen. In the case of simple literals, which is what we'll do first, most of the work will be done by the lexer. The parser's role will grow more important as we build more elaborate expressions formed out of multiple tokens.

Parsing Integers

The most simple literal value one can parse is a plain integer, such as `42`. Its simplicity makes it a good starting point for our parser implementation.

Let's add our first parser test case to express what we want. Create the file `test/parse_spec.js` and set the contents as follows:

test/parse_spec.js

```
/* jshint globalstrict: true */
/* global parse: false */
'use strict';

describe("parse", function() {

  it("can parse an integer", function() {
    var fn = parse('42');
    expect(fn).toBeDefined();
    expect(fn()).toBe(42);
  });

});
```

In the preamble we set the file to strict mode and let JSHint know it's OK to refer to a global called `parse` (which we'll no longer have to do when we've implemented dependency injection).

In the test case itself we define the contract for `parse`: It takes a string and returns a function. That function evaluates to the parsed value of the original string.

To implement this, let's first consider the output of the lexer. We've discussed that it outputs a collection of tokens, but what exactly is a token? For our purposes, a token is an object that gives the parser all the information it needs. At this point, we'll consider two pieces of such information:

- `text` - The text that the token was parsed from.
- `fn` - A function that evaluates to the parsed value of the text

For the number 42, our token would then be something like:

```
{
  text: '42',
  fn: function() { return 42; }
}
```

Why have a function wrapping `42` instead of just using the literal `42` directly? When we get to non-literal values we'll see we don't always know the concrete value of the token at parse time, so for some tokens we need to do some actual work in `fn`. When we also wrap literals in functions we get a uniform interface for all kinds of tokens.

So, let's implement number parsing in the lexer, so we get the data structure similar to the one above from the expression string `'42'`.

The `lex` function of the lexer forms basically one big loop, which iterates over all characters in the given string. During the iteration, it forms the collection of tokens the string included:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
  }

  return this.tokens;
};
```

This function outline does nothing yet (except loop infinitely), but it sets up the fields we'll need during the iteration:

- `text` - The original string
- `index` - Our current character index in the string
- `ch` - The current character
- `tokens` - The resulting collection of tokens.

The `while` loop is where we'll add the behavior for dealing with different kinds of characters. Let's do so now for numbers:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch)) {
      this.readNumber();
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```


If the current character is a number, we delegate to a helper method called `readNumber` to read it in. If the character isn't a number, it's something we can't currently deal with so we just throw an exception.

The `isNumber` check is simple:

src/parse.js

```
Lexer.prototype.isNumber = function(ch) {
  return '0' <= ch && ch <= '9';
};
```

We use the numeric `<=` operator to check that the value of the character is between the values of `'0'` and `'9'`. JavaScript uses lexicographical comparison here, as opposed to numeric comparison, but in the single-digit case they amount to the same thing.

The `readNumber` method, on a high level, has a structure similar to `lex`: It loops over the text character by character, building up the number as it goes:

src/parse.js

```
Lexer.prototype.readNumber = function() {
  var number = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (this.isNumber(ch)) {
      number += ch;
    } else {
      break;
    }
    this.index++;
  }
};
```

The `while` loop reads the current character. Then, if the character is a number, it's concatenated to the local `number` variable and the character index is advanced. If the character is not a number, the loop is terminated.

This gives us the `number` string, but we don't do anything with it yet. We need to coerce it to an actual number, and emit a token:

src/parse.js


```
Lexer.prototype.readNumber = function() {
  var number = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (this.isNumber(ch)) {
      number += ch;
    } else {
      break;
    }
    this.index++;
  }
  number = 1 * number;
  this.tokens.push({
    text: number,
    fn: _.constant(number)
  });
};
```

We first coerce the string to a number by multiplying `1` by it. We then add a new token to the `this.tokens` collection. The token's `text` attribute is the number itself, and its `fn` attribute is a function that always returns the number.

The lexer is now doing its work on integer parsing. Next, we'll focus our attention on the parser. Turns out that in simple literal expressions, like `42`, the parser doesn't really need to do much. The final value of the expression is directly emitted by the lexer. So, at this point, we can get away with having the parser merely take the first token emitted by the lexer and return its `fn` attribute:

src/parse.js

```
Parser.prototype.parse = function(text) {
  this.tokens = this.lexer.lex(text);
  return _.first(this.tokens).fn;
};
```

In any case, that was quite a lot of work for turning `'42'` to `42`! However, now we have a lot of the machinery related to expression parsing in place, and a solid platform to stand on.

> You may have noticed we're only considering *positive* integers here. That is because we will handle negative numbers differently, by considering the minus sign as an *operator* instead of being part of the number itself.

Marking Literals And Constants

We've seen how the parser returns a function that can be used to evaluate the original expression. The returned function should not be just a plain function, though. It should have a couple of extra attributes attached to it:

- `literal` - a boolean value denoting whether the expression was a literal value, such as the integers we've seen so far.
- `constant` - a boolean value denoting whether the expression was a constant, i.e. a literal primitive, or a collection of literal values.

For example, `42` is both literal and constant, as is `[42, 'abc']`. On the other hand, something like `[42, 'abc', aVariable]` is a literal but not a constant.

Users of the `$parse` service occasionally use these flags to make decisions, and we will also do so later in the book, so we need to implement them.

> There's also a third attribute, called `assign`, in the returned functions. We will get to know it once we implement assignment expression parsing.

Integers are both literal and constant, so we can add a test case for checking that the parser sets those flags:

test/parse_spec.js

```
it("makes integers both constant and literal", function() {
  var fn = parse('42');
  expect(fn.constant).toBe(true);
  expect(fn.literal).toBe(true);
});
```

Before the parser can set `literal` and `constant`, it needs some information from the lexer about the nature of each token. The lexer will provide some of this information by attaching a boolean attribute called `json` to each token. When it's `true`, that means the token is a valid JSON expression. We can certainly set that for the numbers we're currently parsing:

src/parse.js

```
Lexer.prototype.readNumber = function() {
  var number = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (this.isNumber(ch)) {
      number += ch;
    } else {
      break;
    }
    this.index++;
  }
```



```
  number = 1 * number;
  this.tokens.push({
    text: number,
    fn: _.constant(number),
    json: true
  });
};
```

In the parser we can look at the value of this flag and set `literal` and `constant` accordingly. Let's extract the "getting the first token" behavior to a new `Parser` function called `primary` (named after the fact that it parses a so-called "primary" or top-level expression). This new function also sets the flags, which are both `true` for JSON tokens:

src/parse.js

```
Parser.prototype.primary = function() {
  var token = this.tokens[0];
  var primary = token.fn;
  if (token.json) {
    primary.constant = true;
    primary.literal = true;
  }
  return primary;
};
```

Later on we will see parse results where the distinction of constant and literal is not quite this straightforward.

The `parse` method can now just invoke `primary`:

src/parse.js

```
Parser.prototype.parse = function(text) {
  this.tokens = this.lexer.lex(text);
  return this.primary();
};
```

With that piece of the parser contract out of the way, let's extend our parsing capabilities with new kinds of literals.

Parsing Floating Point Numbers

Our lexer can currently only deal with integers, and not floating point numbers such as `4.2`:

test/parse_spec.js

```
it("can parse a floating point number", function() {
  var fn = parse('4.2');
  expect(fn()).toBe(4.2);
});
```

Fixing this is straightforward. All we need to do is allow a character in `readNumber` to be the dot `'.'` in addition to a number:

src/parse.js

```
Lexer.prototype.readNumber = function() {
  var number = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === '.' || this.isNumber(ch)) {
      number += ch;
    } else {
      break;
    }
    this.index++;
  }
  number = 1 * number;
  this.tokens.push({
    text: number,
    json: true,
    fn: _.constant(number)
  });
};
```

We don't need to do anything special to parse the dot, since JavaScript's built-in number coercion can handle it.

When a floating point number's integer part is zero, Angular expressions let you omit the integer part completely, just like JavaScript does. Our implementation doesn't yet work with that though, causing this test to fail:

test/parse_spec.js

```
it("can parse a floating point number without an integer part", function() {
  var fn = parse('.42');
  expect(fn()).toBe(0.42);
});
```


The reason is that in the `lex` function we're making the decision about going into `readNumber` by seeing whether the current character is a number. We should also do that when it is a dot, and the *next* character will be a number.

Firstly, for looking at the next character, let's add a function to the lexer called `peek`. It returns the next character in the text, without moving the current character index forward. If there is no next character, `peek` will return `false`:

src/parse.js

```
Lexer.prototype.peek = function() {
  return this.index < this.text.length - 1 ?
    this.text.charAt(this.index + 1) :
    false;
};
```

The `lex` function will now use it to make the decision about going to `readNumber`:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
        (this.ch === '.' && this.isNumber(this.peek()))) {
      this.readNumber();
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

And that covers floating point numbers!

Parsing Scientific Notation

The third and final way to express a number in Angular expressions is scientific notation, which actually consists of two numbers: The coefficient and the exponent, separated by the character `e`. For example, the number 42,000, or 42 * 10^3, can be expressed as `42e3`. As a unit test:

test/parse_spec.js

```
it("can parse a number in scientific notation", function() {
  var fn = parse('42e3');
  expect(fn()).toBe(42000);
});
```

Also, the coefficient in scientific notation does not have to be an integer:

test/parse_spec.js

```
it("can parse scientific notation with a float coefficient", function() {
  var fn = parse('.42e2');
  expect(fn()).toBe(42);
});
```

The exponent in scientific notation may also be negative, causing the coefficient to be multiplied by negative powers of ten:

test/parse_spec.js

```
it("can parse scientific notation with negative exponents", function() {
  var fn = parse('4200e-2');
  expect(fn()).toBe(42);
});
```

The exponent may also be explicitly expressed as positive, by having the `+` sign precede it:

test/parse_spec.js

```
it("can parse scientific notation with the + sign", function() {
  var fn = parse('.42e+2');
  expect(fn()).toBe(42);
});
```

Finally, the `e` character in between the coefficient and the exponent may also be an uppercase `E`:

test/parse_spec.js


```
it("can parse upcase scientific notation", function() {
  var fn = parse('.42E2');
  expect(fn()).toBe(42);
});
```

Now that we have the spec for scientific notation, how should we go about implementing it? The most straightforward approach might be to just downcase each character, pass it right through to the number if it is `e`, `-`, or `+`, and rely on JavaScript's number coercion to do the rest. And that really does make our tests pass:

src/parse.js

```
Lexer.prototype.readNumber = function() {
  var number = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index).toLowerCase();
    if (ch === '.' || ch === 'e' || ch === '-' ||
        ch === '+' || this.isNumber(ch)) {
      number += ch;
    } else {
      break;
    }
    this.index++;
  }
  number = 1 * number;
  this.tokens.push({
    text: number,
    json: true,
    fn: _.constant(number)
  });
};
```

As you might have guessed, we can't get away with it quite that easily. While this implementation parses scientific notation correctly, it is also too lenient about invalid notation, letting through broken number literals such as the following:

test/parse_spec.js

```
it("will not parse invalid scientific notation", function() {
  expect(function() { parse('42e-'); }).toThrow();
  expect(function() { parse('42e-a'); }).toThrow();
});
```

Let's tighten things up. Firstly, we'll need to introduce the concept of an *exponent operator*. That is, a character that is allowed to come after the `e` character in scientific notation. That may be a number, the plus sign, or the minus sign:

src/parse.js


```
Lexer.prototype.isExpOperator = function(ch) {
  return ch === '-' || ch === '+' || this.isNumber(ch);
};
```

Next, we need to use this check in `readNumber`. First of all, let's undo the damage we did in our naïve implementation and introduce an empty `else` branch where we will handle scientific notation:

src/parse.js

```
Lexer.prototype.readNumber = function() {
  var number = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index).toLowerCase();
    if (ch === '.' || this.isNumber(ch)) {
      number += ch;
    } else {

    }
    this.index++;
  }
  number = 1 * number;
  this.tokens.push({
    text: number,
    json: true,
    fn: _.constant(number)
  });
};
```

There are three situations we need to consider:

- If the current character is `e`, and the next character is a valid exponent operator, we should add the current character to the result and proceed.
- If the current character is `+` or `-`, and the previous character was `e`, and the next character is a number, we should add the current character to the result and proceed.
- If the current character is `+` or `-`, and the previous character was `e`, and there is no numeric next character, we should throw an exception.
- Otherwise we should terminate the number parsing and emit the result token.

Here's the same expressed in code:

src/parse.js


```
Lexer.prototype.readNumber = function() {
  var number = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index).toLowerCase();
    if (ch === '.' || this.isNumber(ch)) {
      number += ch;
    } else {
      var nextCh = this.peek();
      var prevCh = number.charAt(number.length - 1);
      if (ch === 'e' && this.isExpOperator(nextCh)) {
        number += ch;
      } else if (this.isExpOperator(ch) && prevCh === 'e' &&
                 nextCh && this.isNumber(nextCh)) {
        number += ch;
      } else if (this.isExpOperator(ch) && prevCh === 'e' &&
                 (!nextCh || !this.isNumber(nextCh))) {
        throw "Invalid exponent";
      } else {
        break;
      }
    }
    this.index++;
  }
  number = 1 * number;
  this.tokens.push({
    text: number,
    json: true,
    fn: _.constant(number)
  });
};
```

Notice that in the second and third branches we are doing the check for `+` or `-` by reusing the `isExpOperator` function. While `isExpOperator` also accepts a number, that can't be the case here since if it was a number it would have activated the very first `if` clause in the `while` loop.

That function is quite a mouthful, but it now gives us the full capabilities of number parsing in Angular expressions - apart from negative numbers, which we will handle with the `-` operator later.

Parsing Strings

With numbers out of the way, let's go ahead and extend the capabilities of the parser to strings. It is almost as straightforward as parsing numbers, but there are a couple of special cases we need to take care of.

At its simplest, a string in an expression is just a sequence of characters wrapped in single or double quotes:

test/parse_spec.js


```
it("can parse a string in single quotes", function() {
  var fn = parse("'abc'");
  expect(fn()).toEqual('abc');
});

it("can parse a string in double quotes", function() {
  var fn = parse('"abc"');
  expect(fn()).toEqual('abc');
});
```

In `lex` we can detect whether the current character is one of those quotes, and step into a function for reading strings, which we'll implement in a moment:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
        (this.ch === '.' && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.ch === '\'' || this.ch === '"') {
      this.readString();
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

On a high level, `readString` is very similar to `readNumber`, it consumes the expression text using a `while` loop and builds up a string into a local variable. The only difference is that before entering the while loop, we'll increment the character index once to get past the opening quote character:

src/parse.js


```
Lexer.prototype.readString = function() {
  this.index++;
  var string = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);

    this.index++;
  }
};
```

So what should we do inside the loop? There are two things: If the current character is something other than a quote, we should just append it to the string. If it *is* a quote, we should emit a token and terminate, since the quote ends the string. After the loop, we'll throw an exception if we're still reading a string because it means the string was not terminated before the expression ended:

src/parse.js

```
Lexer.prototype.readString = function() {
  this.index++;
  var string = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === '\'' || ch === '"') {
      this.index++;
      this.tokens.push({
        fn: _.constant(string)
      });
      return;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

That's a good start for string lexing, but we're not done yet. For instance, we're being a bit too lenient with the parsing again, by allowing a string to be terminated by a different kind of quote than what it was opened with:

test/parse_spec.js

```
it("will not parse a string with mismatching quotes", function() {
  expect(function() { parse('"abc\''); }).toThrow();
});
```

We need to make sure a string ends with the same quote as it started with. Firstly, let's pass the opening quote character into `readString` from the `lex` function:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) || (this.ch === '.' && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.ch === '\'' || this.ch === '"') {
      this.readString(this.ch);
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

In `readString`, we can now check the string termination with the passed-in quote character, rather than a the literal ' or ":

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var string = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === quote) {
      this.index++;
      this.tokens.push({
        fn: _.constant(string)
      });
      return;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```


Before going further with the string parsing behavior, let's consider the token we're emitting for a moment. It currently only has the `fn` key. It should also have the `json` attribute that causes string tokens to be marked literal and constant, which is always the case for strings:

test/parse_spec.js

```
it("marks strings as literal and constant", function() {
  var fn = parse('"abc"');
  expect(fn.literal).toBe(true);
  expect(fn.constant).toBe(true);
});
```

Making this test pass is very simple. We just need to set the `json` flag on the token:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var string = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === quote) {
      this.index++;
      this.tokens.push({
        json: true,
        fn: _.constant(string)
      });
      return;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

The other attribute we should add to string tokens is the `text` attribute, which in the case of strings holds the string itself *with* the surrounding quotes. We won't add a unit test for it since it doesn't have any external effects at this point, but we'll add the attribute anyway. What we need to do is build up an additional string into which we append all the characters we come across:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var rawString = quote;
  var string = '';
```

```
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    rawString += ch;
    if (ch === quote) {
      this.index++;
      this.tokens.push({
        text: rawString,
        json: true,
        fn: _.constant(string)
      });
      return;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

We initialize the variable with the opening quote, and then just append all characters to it as long as we're parsing the string.

Just like JavaScript strings, Angular expression strings may also have escape characters in them. There are two kinds of escapes we'll need to support:

- Single character escapes: Newline `\n`, form feed `\f`, carriage return `\r`, horizontal tab `\t`, vertical tab `\v`, the single quote character `\'`, the double quote character `\"`, and the backslash `\\`.
- Unicode escape sequences, which begin with `\u` and contain a four-digit hexadecimal character code value. For example, `\u00A0` denotes a non-breaking space character.

We need to handle these escapes explicitly in `readString`, since if we don't do anything, the resulting (JavaScript) string will not include them. For example, when there's a newline `\n` in the input, it'll come out as two characters in the output, `'\'` and `'n'`. This is not what we want. What we want instead is the actual character represented by the escape. Let's consider single character escapes first:

test/parse_spec.js

```
it("will parse a string with character escapes", function() {
  var fn = parse('"\\n\\r\\\\"');
  expect(fn()).toEqual('\n\r\\');
});
```

What we'll do during the parse is look out for the backslash `\` and move into an "escape mode", in which we'll handle the succeeding characters differently:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var rawString = quote;
  var string = '';
  var escape = false;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    rawString += ch;
    if (escape) {

    } else if (ch === quote) {
      this.index++;
      this.tokens.push({
        text: rawString,
        json: true,
        fn: _.constant(string)
      });
      return;
    } else if (ch === '\\') {
      escape = true;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

In escape mode, if we're looking at a single-character escape, we should see what character it is and replace it with the corresponding escape character. Let's add a "constant" top-level object in `parse.js` for storing the escape characters we support:

src/parse.js

```
var ESCAPES = {'n':'\n', 'f':'\f', 'r':'\r', 't':'\t',
               'v':'\v', '\'':'\'', '"':'"'};
```

Then, in `readString`, let's look the escape character up from this object. If it's there, we'll append the replacement character. If it isn't, we'll just append the original character as-is, effectively ignoring the escape backslash:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var rawString = quote;
  var string = '';
  var escape = false;
```

```
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    rawString += ch;
    if (escape) {
      var replacement = ESCAPES[ch];
      if (replacement) {
        string += replacement;
      } else {
        string += ch;
      }
      escape = false;
    } else if (ch === quote) {
      this.index++;
      this.tokens.push({
        text: rawString,
        json: true,
        fn: _.constant(string)
      });
      return;
    } else if (ch === '\\') {
      escape = true;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

Note that we don't actually need a replacement character for the backslash \\, because this implementation does the right thing without it.

Finally, let's consider unicode escapes:

test/parse_spec.js

```
it("will parse a string with unicode escapes", function() {
  var fn = parse('"\\u00A0"');
  expect(fn()).toEqual('\u00A0');
});
```

What we need to do is see if the character following the backslash is `u`, and if it is, grab the next four characters, parse them as a hexadecimal (base 16) number, and look up the character corresponding to that number code. For the lookup, we can use the built-in JavaScript `String.fromCharCode` function:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var rawString = quote;
  var string = '';
  var escape = false;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    rawString += ch;
    if (escape) {
      if (ch === 'u') {
        var hex = this.text.substring(this.index + 1, this.index + 5);
        this.index += 4;
        string = String.fromCharCode(parseInt(hex, 16));
      } else {
        var replacement = ESCAPES[ch];
        if (replacement) {
          string += replacement;
        } else {
          string += ch;
        }
      }
      escape = false;
    } else if (ch === quote) {
      this.index++;
      this.tokens.push({
        text: rawString,
        json: true,
        fn: _.constant(string)
      });
      return;
    } else if (ch === '\\') {
      escape = true;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

The final issue to consider is what happens when the character code following `\u` is not actually valid. We should throw an exception in those situations:

test/parse_spec.js

```
it("will not parse a string with invalid unicode escapes", function() {
  expect(function() { parse('"\\u00T0"'); }).toThrow();
});
```

We'll use a regular expression to check that what follows `\u` is exactly four characters that are either numbers or letters between a-f, i.e. valid hexadecimal digits. We will accept either upper or lower case:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var rawString = quote;
  var string = '';
  var escape = false;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    rawString += ch;
    if (escape) {
      if (ch === 'u') {
        var hex = this.text.substring(this.index + 1, this.index + 5);
        if (!hex.match(/[\da-f]{4}/i)) {
          throw 'Invalid unicode escape';
        }
        this.index += 4;
        string = String.fromCharCode(parseInt(hex, 16));
      } else {
        var replacement = ESCAPES[ch];
        if (replacement) {
          string += replacement;
        } else {
          string += ch;
        }
      }
      escape = false;
    } else if (ch === quote) {
      this.index++;
      this.tokens.push({
        text: rawString,
        json: true,
        fn: _.constant(string)
      });
      return;
    } else if (ch === '\\') {
      escape = true;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

And now we're able to parse strings!

Parsing `true`, `false`, and `null`

The third type of literals we'll add support for are boolean literals `true` and `false`, and the literal `null`. They are all so-called *identifier* tokens because they are bare alphanumeric character sequences in the input. We are going to see lots of identifiers as we go along. Often they're used to look up scope attributes by name, but they can also be reserved words such as `true`, `false`, or `null`. Those should be represented by the corresponding JavaScript values in the parser output:

test/parse_spec.js

```
it("will parse null", function() {
  var fn = parse('null');
  expect(fn()).toBe(null);
});

it("will parse true", function() {
  var fn = parse('true');
  expect(fn()).toBe(true);
});

it("will parse false", function() {
  var fn = parse('false');
  expect(fn()).toBe(false);
});
```

An identifier is a character sequence that begins with a lower or upper case letter, the underscore character, or the dollar character:

src/parse.js

```
Lexer.prototype.isIdent = function(ch) {
  return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') ||
    ch === '_' || ch === '$';
};
```

When we encounter such a character, we'll parse the identifier using a new function called `readIdent`:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
```



```
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
        (this.ch === '.' && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.ch === '\'' || this.ch === '"') {
      this.readString(this.ch);
    } else if (this.isIdent(this.ch)) {
      this.readIdent();
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

Once in `readIdent`, we'll read in the identifier token very similarly as we did with strings:

src/parse.js

```
Lexer.prototype.readIdent = function() {
  var text = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (this.isIdent(ch) || this.isNumber(ch)) {
      text += ch;
    } else {
      break;
    }
    this.index++;
  }

  var token = {text: text};

  this.tokens.push(token);
};
```

Note that an identifier may also contain numbers, but it may not begin with one.

Now we have the raw text of the identifier, but we don't yet have a primary expression token, since there is no `fn` attribute attached to it.

Every identifier token will have some kind of function attached to its `fn` attribute. These functions will vary greatly based on what the identifier is, since identifier tokens can represent so many things. Many of these functions will be looked up from a top-level object called `OPERATORS`. For `true`, `false`, and `null`, the functions are very simple - they just return the corresponding JavaScript value:

src/parse.js


```
var OPERATORS = {
  'null': _.constant(null),
  'true': _.constant(true),
  'false': _.constant(false)
};
```

The attachment of the operator is now straightforward. We just check whether there is a matching operator, and if there is, plop it right onto the token:

src/parse.js

```
Lexer.prototype.readIdent = function() {
  var text = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (this.isIdent(ch) || this.isNumber(ch)) {
      text += ch;
    } else {
      break;
    }
    this.index++;
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
  }

  this.tokens.push(token);
};
```

The literals `true`, `false`, and `null` are also all literal and constant tokens, and should be marked as such:

test/parse_spec.js

```
it("marks booleans as literal and constant", function() {
  var fn = parse('true');
  expect(fn.literal).toBe(true);
  expect(fn.constant).toBe(true);
});

it("marks null as literal and constant", function() {
  var fn = parse('null');
  expect(fn.literal).toBe(true);
  expect(fn.constant).toBe(true);
});
```

As we set the `fn` attribute in `readIdent`, we can also just set the `json` attribute that will take care of this:

src/parse.js

```
Lexer.prototype.readIdent = function() {
  var text = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (this.isIdent(ch) || this.isNumber(ch)) {
      text += ch;
    } else {
      break;
    }
    this.index++;
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
    token.json = true;
  }

  this.tokens.push(token);
};
```

Later we will see operator functions that are much more involved than just returning a constant value.

Parsing Whitespace

Before we start discussing multi-token expressions, let's consider the question of whitespace. Expressions like `'[1, 2, 3]'`, `'a = 42'`, and `'aFunction (42)'` all contain whitespace characters. What's common to all of them is that the whitespace is completely optional and will be ignored by the parser. This is true for (almost) all whitespace in Angular expressions.

test/parse_spec.js

```
it('ignores whitespace', function() {
  var fn = parse(' \n42 ');
  expect(fn()).toEqual(42);
});
```

The characters we consider to be whitespace will be the space, the carriage return, the horizontal and vertical tabs, the newline, and the non-breaking space:

test/parse_spec.js


```
Lexer.prototype.isWhitespace = function(ch) {
  return (ch === ' ' || ch === '\r' || ch === '\t' ||
    ch === '\n' || ch === '\v' || ch === '\u00A0');
};
```

In `lex` we will just move the current character pointer forward when we encounter one of these characters:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
        (this.ch === '.' && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.ch === '\'' || this.ch === '"') {
      this.readString(this.ch);
    } else if (this.isIdent(this.ch)) {
      this.readIdent();
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

Parsing Arrays

Numbers, strings, booleans, and `null` are all so-called *scalar* literal expressions. They are simple, singular values that consist of just one token each. Now we'll turn our attention to multiple token expressions. The first one of those is arrays.

The most simple array you can have is an empty one. It consists of just an opening square bracket and a closing square bracket:

test/parse_spec.js

```
it("will parse an empty array", function() {
  var fn = parse('[]');
  expect(fn()).toEqual([]);
});
```

Simple though this may be, it is the first expression we've seen that isn't just a single token. The Lexer is going to emit two tokens for this expression, one for each square bracket. We'll emit these tokens right from the `lex` function:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.ch === '.' && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.ch === '\'' || this.ch === '"') {
      this.readString(this.ch);
    } else if (this.ch === '[' || this.ch === ']') {
      this.tokens.push({
        text: this.ch,
        json: true
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent();
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

The Lexer does nothing interesting with these tokens. It just emits them. They also don't have an `fn` function attached to them, so you could not use a single square bracket as the whole value of an expression. It needs to be a part of something bigger, and here's where the responsibility shifts to the Parser.

The `primary` function in the Parser currently just takes the first token of the expression and treats that as the whole thing. Since arrays are also primary expressions, `primary` is

going to have to learn some new tricks. To begin with, the `primary` expression should be handled differently depending on whether it begins with an opening square bracket token or not:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }
  return primary;
};
```

The `expect` function used here is something we don't have yet. We're using it in two ways: Once with a string argument and once with no arguments.

When called without arguments, the `expect` function's contract is "consume and return the next token, or return `undefined` if there are no more tokens":

src/parse.js

```
Parser.prototype.expect = function() {
  if (this.tokens.length > 0) {
    return this.tokens.shift();
  }
};
```

When there is an argument, it means "consume and return the next token, but only if its text matches the one given":

src/parse.js

```
Parser.prototype.expect = function(e) {
  if (this.tokens.length > 0) {
    if (this.tokens[0].text === e || !e) {
      return this.tokens.shift();
    }
  }
};
```


The `arrayDeclaration` function is also new. This is where we will consume the tokens related to an array and construct the array expression. When we enter the function the opening square bracket will already have been consumed. Since we're only concerned with empty arrays for now, what remains is the closing square bracket:

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  this.consume(']');
};
```

The `consume` function used here is basically the same thing as `expect`, but with one major difference: It will actually throw an exception if a matching token is not found. The closing square bracket in arrays is definitely not optional, so we need to be strict about it:

src/parse.js

```
Parser.prototype.consume = function(e) {
  if (!this.expect(e)) {
    throw 'Unexpected. Expecting '+e;
  }
};
```

Like any expression, the result of an array declaration should be a function that produces the final value of the expression. For scalar literals we just used the `fn` attributes from tokens directly, but for arrays we'll need to construct the expression function in the parser. Let's make one that produces an empty array:

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  this.consume(']');
  return function() {
    return [];
  };
};
```

So that is how a basic, empty array is produced: The Lexer emits its opening and closing square brackets as tokens, `Parser.primary` notices the opening square bracket and delegates to `Parser.arrayDeclaration`, which does the rest.

When array literals have elements in them, they are separated by commas. The elements may be anything, even other arrays:

test/parse_spec.js


```
it("will parse a non-empty array", function() {
  var fn = parse('[1, "two", [3]]');
  expect(fn()).toEqual([1, 'two', [3]]);
});
```

That comma between the values needs to be emitted from the Lexer. Like the square brackets, it's emitted as-is, as a plain text token:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.ch === '.' && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.ch === '\'' || this.ch === '"') {
      this.readString(this.ch);
    } else if (this.ch === '[' || this.ch === ']' || this.ch === ',') {
      this.tokens.push({
        text: this.ch,
        json: true
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent();
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

The element parsing happens in `arrayDeclaration`. It should check whether the array is immediately closed as an empty array or not. If not, there's element parsing to be done. The element expressions are collected to a local array called `elementFns`:

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  var elementFns = [];
  if (!this.peek(']')) {

  }
  this.consume(']');
  return function(self, locals) {
    return [];
  };
};
```

We need to define the `peek` function used above before getting into parsing the elements. It is basically the same as `expect`, but does *not* consume the token it looks at:

src/parse.js

```
Parser.prototype.peek = function(e) {
  if (this.tokens.length > 0) {
    var text = this.tokens[0].text;
    if (text === e || !e) {
      return this.tokens[0];
    }
  }
};
```

Now we can actually redefine `expect` in terms of `peek` so we don't need to duplicate their common logic. If a matching token is found with `peek`, `expect` consumes it:

src/parse.js

```
Parser.prototype.expect = function(e) {
  var token = this.peek(e);
  if (token) {
    return this.tokens.shift();
  }
};
```

If there are elements in the array, we'll consume them in a loop. The loop terminates when we see no longer see a comma following the last element. Each element is recursively parsed as *another* primary expression. That's where the name `elementFns` come from - they are element expression *functions* at this point:

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  var elementFns = [];
  if (!this.peek(']')) {
    do {
      elementFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(']');
  return function(self, locals) {
    return [];
  };
};
```

Then, when the expression is evaluated, we will evaluate each of the element expression functions in turn, to obtain the final array of elements. The LoDash `_.map` function comes in handy here:

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  var elementFns = [];
  if (!this.peek(']')) {
    do {
      elementFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(']');
  return function() {
    return _.map(elementFns, function(elementFn) {
      return elementFn();
    });
  };
};
```

Arrays in Angular expressions also allow you to use a trailing comma, i.e. a comma after which there are no more elements in the array. This is consistent with standards-compliant JavaScript implementations:

test/parse_spec.js

```
it("will parse an array with trailing commas", function() {
  var fn = parse('[1, 2, 3, ]');
  expect(fn()).toEqual([1, 2, 3]);
});
```

To support a trailing comma, we need to tweak the `do..while` loop so that it's prepared for a situation where it doesn't have an element expression to parse. If it sees the closing square bracket, it should break out early:

src/parse.js


```
Parser.prototype.arrayDeclaration = function() {
  var elementFns = [];
  if (!this.peek(']')) {
    do {
      if (this.peek(']')) {
        break;
      }
      elementFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(']');
  return function() {
    var elements = _.map(elementFns, function(elementFn) {
      return elementFn();
    });
    return elements;
  };
};
```

As we earlier marked scalar literal expressions with the `literal` and `constant` flags, so should we do with arrays. An array literal is always a literal. It is a constant if it only contains other literal constants. For now that is always the case, though in the next chapter we'll see situations where it's not.

test/parse_spec.js

```
it("marks array literals as literal and constant", function() {
  var fn = parse('[1, 2, 3]');
  expect(fn.literal).toBe(true);
  expect(fn.constant).toBe(true);
});
```

At this point all we need to to is set both flags to `true` for the expression function returned from `arrayLiteral`:

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  var elementFns = [];
  if (!this.peek(']')) {
    do {
      if (this.peek(']')) {
        break;
      }
      elementFns.push(this.primary());
    } while (this.expect(','));
  }
```



```
  this.consume(']');
  var arrayFn = function() {
    var elements = _.map(elementFns, function(elementFn) {
      return elementFn();
    });
    return elements;
  };
  arrayFn.literal = true;
  arrayFn.constant = true;
  return arrayFn;
};
```

Parsing Objects

The final expression type we will add support for in this chapter is object literals. That is, key-value pairs such as `{a: 1, b: 2}`. In expressions, objects are often used not only as data literals, but also as configuration for directives such as ngClass and ngStyle.

Parsing objects is in many ways similar to parsing arrays, with a couple of key differences. Again, let's first take the case of an empty collection. An empty object should evaluate to an empty object:

test/parse_spec.js

```
it("will parse an empty object", function() {
  var fn = parse('{}');
  expect(fn()).toEqual({});
});
```

For objects we are going to need three more character tokens from the Lexer: The opening and closing curly braces, and the colon for denoting key-value pairs:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.ch === '.' && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.ch === '\'' || this.ch === '"') {
      this.readString(this.ch);
```



```
    } else if (this.ch === '[' || this.ch === ']' || this.ch === ',' ||
                this.ch === '{' || this.ch === '}' || this.ch === ':') {
      this.tokens.push({
        text: this.ch,
        json: true
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent();
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

This `else if` branch is getting a little bit unwieldy. Let's add a helper method to Lexer that'll make it easier to check if the current character matches a number of alternatives. The function takes a string, and checks whether the current character matches any character in that string:

src/parse.js

```
Lexer.prototype.is = function(chs) {
  return chs.indexOf(this.ch) >= 0;
};
```

Now we can make the code in `lex` more concise:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.is('.') && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.is('\'"')) {
      this.readString(this.ch);
    } else if (this.is('[],{}:')) {
```



```
      this.tokens.push({
        text: this.ch,
        json: true
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent();
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

Objects, like arrays, are a primary expression. `Parser.primary` looks out for opening curly braces and when one is seen, delegates to a new method called `object`:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }
  return primary;
};
```

The `object` method's structure is basically the same as that of `arrayDeclaration`. It consumes the object, including the closing curly brace, and returns a function that evaluates to the object. The expression is both literal and constant (for the time being):

src/parse.js

```
Parser.prototype.object = function() {
  this.consume('}');
  var objectFn = function() {
    return {};
  };
  objectFn.literal = true;
  objectFn.constant = true;
  return objectFn;
};
```

When an object is not empty, its keys are identifiers or strings, and its values may be any other expressions. Here's a test case with identifier keys:

test/parse_spec.js

```
it("will parse a non-empty object", function() {
  var fn = parse('{a: 1, b: [2, 3], c: {d: 4}}');
  expect(fn()).toEqual({a: 1, b: [2, 3], c: {d: 4}});
});
```

Just like in array parsing, in object parsing we have a `do..while` loop that consumes the keys and values separated by commas:

src/parse.js

```
Parser.prototype.object = function() {
  if (!this.peek('}')) {
    do {

    } while (this.expect(','));
  }
  this.consume('}');
  var objectFn = function() {
    return {};
  };
  objectFn.literal = true;
  objectFn.constant = true;
  return objectFn;
};
```

Inside the loop body, we will first read in the key by expecting the next token. Then we'll consume the colon character that should separate the key and a value. Finally we'll consume the value, which is not just a token, but possibly a whole other primary expression.

Once we have the key and a value, we store them in an array so that we can use them later. For the keys we just store the token's `text` attribute (populated back in `Lexer.readIdent`). For the values we store the whole expression function:

src/parse.js


```
Parser.prototype.object = function() {
  var keyValues = [];
  if (!this.peek('}')) {
    do {
      var keyToken = this.expect();
      this.consume(':');
      var valueExpression = this.primary();
      keyValues.push({key: keyToken.text, value: valueExpression});
    } while (this.expect(','));
  }
  this.consume('}');
  var objectFn = function() {
    return {};
  };
  objectFn.literal = true;
  objectFn.constant = true;
  return objectFn;
};
```

At evaluation time we'll then construct the actual object. We create an empty object and then iterate over the key-value pairs and stick them into the object. Keys we just put in as-is, but values need to be invoked since they are expression functions themselves:

src/parse.js

```
Parser.prototype.object = function() {
  var keyValues = [];
  if (!this.peek('}')) {
    do {
      var keyToken = this.expect();
      this.consume(':');
      var valueExpression = this.primary();
      keyValues.push({key: keyToken.text, value: valueExpression});
    } while (this.expect(','));
  }
  this.consume('}');
  var objectFn = function() {
    var object = {};
    _.forEach(keyValues, function(kv) {
      object[kv.key] = kv.value();
    });
    return object;
  };
  objectFn.literal = true;
  objectFn.constant = true;
  return objectFn;
};
```

An object's keys are not always identifiers. They may also be strings in single or double quotes. That's useful when you have characters in keys that aren't allowed in identifiers, such as whitespace or `'-'`:

test/parse_spec.js

```
it("will parse an object with string keys", function() {
  var fn = parse('{"a key": 1, \'another-key\': 2}');
  expect(fn()).toEqual({'a key': 1, 'another-key': 2});
});
```

The test fails because in Lexer, as we produce a string token, its text property becomes the *raw* string the token was parsed out of, with the quote characters included. We could invoke the `fn` function to get to the processed String, but what Angular actually does is it has another key in string tokens, called `string`, that just holds the string value of the token:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var rawString = quote;
  var string = '';
  var escape = false;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    rawString += ch;
    if (escape) {
      if (ch === 'u') {
        var hex = this.text.substring(this.index + 1, this.index + 5);
        if (!hex.match(/[\da-f]{4}/i)) {
          throw 'Invalid unicode escape';
        }
        this.index += 4;
        string = String.fromCharCode(parseInt(hex, 16));
      } else {
        var replacement = ESCAPES[ch];
        if (replacement) {
          string += replacement;
        } else {
          string += ch;
        }
      }
      escape = false;
    } else if (ch === quote) {
      this.index++;
      this.tokens.push({
        text: rawString,
        string: string,
        json: true,
```



```
        fn: _.constant(string)
      });
      return;
    } else if (ch === '\\') {
      escape = true;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

When obtaining the object key in `Parser.object`, we'll first try to get the key token's `string` attribute and then fall back to `text` attribute if needed:

src/parse.js

```
Parser.prototype.object = function() {
  var keyValues = [];
  if (!this.peek('}')) {
    do {
      var keyToken = this.expect();
      this.consume(':');
      var valueExpression = this.primary();
      keyValues.push({
        key: keyToken.string || keyToken.text,
        value: valueExpression
      });
    } while (this.expect(','));
  }
  this.consume('}');
  var objectFn = function() {
    var object = {};
    _.forEach(keyValues, function(kv) {
      object[kv.key] = kv.value();
    });
    return object;
  };
  objectFn.literal = true;
  objectFn.constant = true;
  return objectFn;
};
```

And now we're able to parse all the literals that the Angular expression language supports!

Integrating Expressions to Scopes

Since we now have a workable version of the to-be `$parse` service, we can already go ahead and integrate it to our scope implementation. Granted, the expressions we're able to parse aren't that interesting or useful yet, but after the next chapter they will be!

What we'll do is modify the external API of `Scope` so that it'll accept expression strings as well as raw functions in the following operations:

- `$watch` (both the watch function and the listener function)
- `$watchCollection`
- `$eval` (and, by association, `$apply` and `$evalAsync`)

Internally, `Scope` will use the `parse` function (and later, the `$parse` service) to parse those expressions into functions.

Since `Scope` will still support the use of raw functions, we will need to check whether the arguments given are strings, or if they are functions already. We will do this in `parse`:

test/parse_spec.js

```
it('returns the function itself when given one', function() {
  var fn = function() { };
  expect(parse(fn)).toBe(fn);
});
```

While we're at it, let's also make things more robust by dealing with situations where the argument given to `parse` is something unexpected (or there wasn't one at all). In those cases, `parse` should just return a function that does nothing:

test/parse_spec.js

```
it('still returns a function when given no argument', function() {
  expect(parse()).toEqual(jasmine.any(Function));
});
```

In `parse`, we'll make a decision about what to do based on the type of the given argument. If it is a string, we parse it as before. If it is a function, we just return it. In other cases, we return `_.noop`, which is a Lo-Dash-provided function that does nothing:

src/parse.js

```
function parse(expr) {
  switch (typeof expr) {
    case 'string':
      var lexer = new Lexer();
      var parser = new Parser(lexer);
      return parser.parse(expr);
    case 'function':
      return expr;
    default:
      return _.noop;
  }
}
```


Now, in `scope.js`, let's first add a (temporary) JSHint directive to the top that makes it OK for us to reference the global `parse` function:

src/scope.js

```
/* global parse: false */
```

The first instance where we accept expressions is the watch function of `$watch`. Let's make a test for that (in the `describe("digest")` test block:

test/scope_spec.js

```
it('accepts expressions for watch functions', function() {
  var theValue;

  scope.$watch('42', function(newValue, oldValue, scope) {
    theValue = newValue;
  });
  scope.$digest();

  expect(theValue).toBe(42);
});
```

All we need to do to make this work is invoke `parse` for the given watch function, and store its return value in the watch object instead of the original argument:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;
  var watcher = {
    watchFn: parse(watchFn),
    listenerFn: listenerFn || function() { },
    last: initWatchVal,
    valueEq: !!valueEq
  };
  this.$$watchers.unshift(watcher);
  this.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
      self.$$lastDirtyWatch = null;
    }
  };
};
```


Using expression strings in watches enables us to add a new optimization that will in some cases make the digest loop go faster. Earlier in the chapter we saw how constant expressions have their `constant` flag set to `true`. A constant expression will always return the same value. This means that after a watch with a constant expression has been triggered for the first time, it'll never become dirty again!

We can just as well remove a constant watch completely after its first invocation, so we'll have one less item to iterate during subsequent digests:

test/scope_spec.js

```
it('removes constant watches after first invocation', function() {
  scope.$watch('42', function() {});
  scope.$digest();

  expect(scope.$$watchers.length).toBe(0);
});
```

We can do this by wrapping the listener function in a new function that, in addition to invoking the original listener, removes the watch:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;

  watchFn = parse(watchFn);

  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    last: initWatchVal,
    valueEq: !!valueEq
  };

  if (watchFn.constant) {
    watcher.listenerFn = function(newValue, oldValue, scope) {
      listenerFn(newValue, oldValue, scope);
      var index = self.$$watchers.indexOf(watcher);
      if (index >= 0) {
        self.$$watchers.splice(index, 1);
      }
    };
  }

  this.$$watchers.unshift(watcher);
  this.$$lastDirtyWatch = null;
```



```
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
      self.$$lastDirtyWatch = null;
    }
  };
};
```

The second instance where we accept expressions is the *listener* function of watches. Since we don't yet support expressions with side-effects (such as assignments), we can't yet test this very well, but what we can do is to see that it actually does something without throwing exceptions:

test/scope_spec.js

```
it('accepts expressions for listener functions', function() {
  scope.$watch('42', '"fourty-two"');
  scope.$digest();
});
```

This does throw an exception, though it gets handled by the `try..catch` in `$$digestOnce`. In any case, let's add the parse invocation for the listener function:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;

  watchFn = parse(watchFn);
  listenerFn = parse(listenerFn);

  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn,
    last: initWatchVal,
    valueEq: !!valueEq
  };
  this.$$watchers.unshift(watcher);
  this.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
      self.$$lastDirtyWatch = null;
    }
  };
};
```


Note that we no longer need to substitute a missing listener with a no-op function here, since we are already doing that in `parse`.

Since `$watch` now accepts expressions, `$watchCollection` should do the same. Just like with `$watch`, we are a bit constrained in how we can currently test that in the case of listeners though. Add these in the `describe("$watchCollection")` test block:

test/scope_spec.js

```
it('accepts expressions for watch functions', function() {
  var theValue;

  scope.$watchCollection('[1, 2, 3]', function(newValue, oldValue, scope) {
    theValue = newValue;
  });
  scope.$digest();

  expect(theValue).toEqual([1, 2, 3]);
});

it('accepts expressions for listener functions', function() {
  var theValue;

  scope.$watchCollection('[1, 2, 3]', '"one-two-three"');
  scope.$digest();
});
```

To make this work, we also need to call `parse` in `$watchCollection`, for both the watch function and the listener function:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  var self = this;
  var newValue;
  var oldValue;
  var oldLength;
  var changeCount = 0;

  watchFn = parse(watchFn);
  listenerFn = parse(listenerFn);

  // The rest of the function unchanged
};
```

Next, we should support expressions in `$eval`:

test/scope_spec.js


```
it('accepts expressions in $eval', function() {
  expect(scope.$eval('42')).toBe(42);
});
```

Also, since `$apply` and `$evalAsync` are built on top of `$eval`, they will also support expressions. And again, since our expression parsing support is still so limited, we cannot yet test the `$evalAsync` case very thoroughly:

test/scope_spec.js

```
it('accepts expressions in $apply', function() {
  expect(scope.$apply('42')).toBe(42);
});

it('accepts expressions in $evalAsync', function(done) {
  scope.$evalAsync('42');
  scope.$$postDigest(done);
});
```

In `$eval` we can just parse the incoming expression and then invoke it:

src/scope.js

```
Scope.prototype.$eval = function(expr, locals) {
  return parse(expr)(this, locals);
};
```

Note that we're passing the `locals` argument on to the expression function. In upcoming chapters we will see the implications of doing this.

Summary

We now have a very limited but functional implementation of the Angular expression parser. While building it you have learned:

- That the expression parser runs internally in two phases: Lexing and parsing.
- How the parser deals with integers, floating point numbers, and scientific notation.
- That the functions returned by the parser have additional attributes called `literal` and `constant` that carry information about the nature of the expression.
- How the parser deals with strings.
- How the parser deals with literal booleans and `null`.
- How the parser deals with whitespace - by ignoring it.
- How the parser deals with arrays and objects, and how it recursively parses their contents.
- How expression parsing is integrated to scopes, and which scope operations support expressions.

In the next chapter we'll extend our parser's capabilities with much more interesting expressions: Those that actually access scope attributes.

Chapter 6

Lookup And Function Call Expressions

By now we have a simple expression language that can express literals, but that isn't very useful in any real world scenario. What the Angular expression language is designed for is accessing data on scopes, and occasionally also manipulating that data.

In this chapter we will add those capabilities. We'll learn to access and assign attributes in different ways and to call functions. We will also implement many of the security measures the Angular expression language takes to prevent dangerous expressions from getting through.

Simple Attribute Lookup

The simplest kind of scope attribute access you can do is to look up something by name: The expression `'aKey'` finds the `aKey` attribute from a scope object and returns it:

test/parse_spec.js

```
it('looks up an attribute from the scope', function() {
  var fn = parse('aKey');
  expect(fn({aKey: 42})).toBe(42);
  expect(fn({})).toBeUndefined();
  expect(fn()).toBeUndefined();
});
```

Notice how the functions returned by `parse` actually take a JavaScript object as an argument. That object is almost always an instance of `Scope`, which the expression will access or manipulate. It doesn't necessarily have to be a Scope though, and in unit tests we can just use plain object literals. Since literal expressions don't do anything with scopes we haven't used this argument before, but that will change in this chapter.

When parsed, the expression `'aKey'` amounts to a single token, which is an identifier. We have already implemented the parsing of identifiers for the purposes of `true`, `false`, and `null`. Now we will extend our identifier support to include attribute lookup.

Recall that identifier tokens have an `fn` attribute, which contains a function that evaluates the token. That function is used as the return value of the whole parse operation if it is the only token in the expression, as it has been so far. For `true`, `false`, and `null` the function just returns a constant value `true`, `false`, or `null`. For other identifiers, the function should look up something from the scope. These kinds of functions are called getters. We will create one in `Lexer.readIdent` if no operator is found for the identifier:

src/parse.js

```
Lexer.prototype.readIdent = function() {
  var text = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (this.isIdent(ch) || this.isNumber(ch)) {
      text += ch;
    } else {
      break;
    }
    this.index++;
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
    token.json = true;
  } else {
    token.fn = getterFn(text);
  }

  this.tokens.push(token);
};
```

`getterFn` is an internal function in `parse.js` that takes the text value of an identifier token, and returns a function that looks up the corresponding attribute from a scope:

src/parse.js

```
var getterFn = function(ident) {
  return function(scope) {
    return scope ? scope[ident] : undefined;
  };
};
```


The test now passes and we're able to look up things from scopes! That instantly makes the expression language much more useful, especially in watch expressions.

> If you've written AngularJS applications before, you've probably noticed that the Angular expression language is very forgiving when it comes to missing attributes. Unlike JavaScript, it basically never throws exceptions when you reference attributes from objects that don't exist. In the implementation of `getterFn` we're starting to see why: Before referencing the attribute we first check that the object exists. If it doesn't, we just return `undefined`.

Nested Attribute Lookup

In addition to just referencing an attribute, you can go deeper in the same expression and look up something in a nested data structure:

test/parse_spec.js

```
it('looks up a 2-part identifier path from the scope', function() {
  var fn = parse('aKey.anotherKey');
  expect(fn({aKey: {anotherKey: 42}})).toBe(42);
  expect(fn({aKey: {}})).toBeUndefined();
  expect(fn({})).toBeUndefined();
});
```

We expect the expression to reach down to `aKey.anotherKey`, or return `undefined` if one or both of the keys are missing.

We don't currently accept dots in identifier expressions, so before doing anything else we need to change that:

src/parse.js

```
Lexer.prototype.readIdent = function() {
  var text = '';
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === '.' || this.isIdent(ch) || this.isNumber(ch)) {
      text += ch;
    } else {
      break;
    }
    this.index++;
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
```



```
    token.json = true;
  } else {
    token.fn = getterFn(text);
  }

  this.tokens.push(token);
};
```

The `getterFn` function should now behave differently depending on whether there's a dot in the expression or not. What we'll do is split the given identifier at the dot if there is one, and delegate to one of two helper functions in each case:

src/parse.js

```
var getterFn = function(ident) {
  var pathKeys = ident.split('.');
  if (pathKeys.length === 1) {
    return simpleGetterFn1(pathKeys[0]);
  } else {
    return simpleGetterFn2(pathKeys[0], pathKeys[1]);
  }
};
```

The non-nested version, `simpleGetterFn1` does what `getterFn` did before. It returns a function that just looks up the key from the scope:

src/parse.js

```
var simpleGetterFn1 = function(key) {
  return function(scope) {
    return scope ? scope[key] : undefined;
  };
};
```

The nested version, `simpleGetterFn2` is slightly more involved. It first looks up the first key, and then the second key from the resulting value, which is presumably another object:

src/parse.js

```
var simpleGetterFn2 = function(key1, key2) {
  return function(scope) {
    if (!scope) {
      return undefined;
    }
    scope = scope[key1];
    return scope ? scope[key2] : undefined;
  };
};
```

Just like in the simple lookup case, we're being very lenient about missing objects. If either of the two lookups fails because there's nothing to look up from, we just return `undefined`. JavaScript would throw an error if the first key did not exist.

Arbitrarily Nested Attribute Lookup

It goes without saying that sometimes you need to go deeper than two nested attributes into a data structure. A 4-part identifier lookup should work just as well as a 2-part one does:

test/parse_spec.js

```
it('looks up a 4-part identifier path from the scope', function() {
  var fn = parse('aKey.secondKey.thirdKey.fourthKey');
  expect(fn({aKey: {secondKey: {thirdKey: {fourthKey: 42}}}})).toBe(42);
  expect(fn({aKey: {secondKey: {thirdKey: {}}}})).toBeUndefined();
  expect(fn({aKey: {}})).toBeUndefined();
  expect(fn()).toBeUndefined();
});
```

Now, we could just continue doing what we've been doing and introduce functions like `simpleGetterFn3`, `simpleGetterFn4`, etc., but that becomes impractical very quickly. AngularJS does have the simple getters for the single and double part getters, but for the rest it uses a generic one:

src/parse.js

```
var getterFn = function(ident) {
  var pathKeys = ident.split('.');
  if (pathKeys.length === 1) {
    return simpleGetterFn1(pathKeys[0]);
  } else if (pathKeys.length === 2) {
    return simpleGetterFn2(pathKeys[0], pathKeys[1]);
  } else {
    return generatedGetterFn(pathKeys);
  }
};
```

A straightforward way to implement `generatedGetterFn` would be to just iterate over the array of keys whenever the expression is evaluated. But this is not what we're going to do. Remember that expressions are often used in watches, and watches get executed very often. That means they have to be very fast. Even simply iterating over keys every time is considered too much, so what Angular does instead is to iterate over the keys at *parse time* and generate a function that doesn't need to iterate at evaluation time. It does this by using code generation.

What we'll do is build up a string of the JavaScript code that does the lookup and then evaluate it into a function:

src/parse.js


```
var generatedGetterFn = function(keys) {
  var code = '';

  /* jshint -W054 */
  return new Function('scope', code);
  /* jshint +W054 */
};
```

We're using the `Function` constructor to create the function. The `Function` constructor takes the names of a function's arguments and the source code string for the function. It then evaluates the code as JavaScript. This is basically a form of `eval`. JSHint doesn't like `eval`, so we need to tell it that we know we're doing something potentially dangerous and it should not raise warnings about it. (`W054` is one of the JSHint numeric warning codes, and stands for "The Function constructor is a form of eval.")

To build up the source code of the function, we iterate over the keys and generate code similar to what we have in the simple getters:

src/parse.js

```
var generatedGetterFn = function(keys) {
  var code = '';
  _.forEach(keys, function(key) {
    code += 'if (!scope) { return undefined; }\n';
    code += 'scope = scope["' + key + '"];\n';
  });
  code += 'return scope;\n';
  /* jshint -W054 */
  return new Function('scope', code);
  /* jshint +W054 */
};
```

When applied to an expression like `'aKey.secondKey.thirdKey.fourthKey'`, this function now generates a function that looks something like this:

```
function(scope) {
  if (!scope) { return undefined; }
  scope = scope["aKey"];
  if (!scope) { return undefined; }
  scope = scope["secondKey"];
  if (!scope) { return undefined; }
  scope = scope["thirdKey"];
  if (!scope) { return undefined; }
  scope = scope["fourthKey"];
  return scope;
}
```


No looping required once the parsing is done!

Technically we could also generate the one and two part versions with this function and remove `simpleGetterFn1` and `simpleGetterFn2` completely. However, Angular keeps the simple getters around, most likely because code generation is not free in terms of performance either, and you'd rather have even faster special cases for the most common, simple attribute lookups.

Getter Caching

Our getters are nicely performant, but we're still incurring the cost of instantiating a new function whenever an attribute lookup is parsed. If the expression `'aKey'` is parsed ten times, there will be ten identical getter functions created. We can do better than that, by simply caching the return values of `getterFn`.

Angular does this manually, but we have LoDash around and can simply wrap `getterFn` with the LoDash memoization function:

src/parse.js

```
var getterFn = _.memoize(function(ident) {
  var pathKeys = ident.split('.');
  if (pathKeys.length === 1) {
    return simpleGetterFn1(pathKeys[0]);
  } else if (pathKeys.length === 2) {
    return simpleGetterFn2(pathKeys[0], pathKeys[1]);
  } else {
    return generatedGetterFn(pathKeys);
  }
});
```

From now on, when `getterFn` is called twice with the same expression, on the second time it'll return the getter from its cache.

Locals

Until now all the functions returned by `parse` have taken one argument - the scope. The literal expression functions ignore even that.

There is a second argument that expressions should accept, called `locals`. You may recall that we've already seen this in `Scope.prototype.$eval`, which takes a `locals` argument and passes it on to `parse`.

The `locals` argument is basically just another object, like the `scope` argument is. The contract is that parsed expressions should access either the `scope` object or the `locals` object. They should try to use `locals` first and only if that fails fall back to `scope`. That means you can effectively override `scope` attributes with `locals`.

> The Angular `select` directive uses locals while tracking selected options, as we'll see later in the book.

We'll need to revisit our getters and make them accept the `locals` argument. For the one-part key version:

test/parse_spec.js

```
it('uses locals instead of scope when there is a matching key', function() {
  var fn = parse('aKey');
  expect(fn({aKey: 42}, {aKey: 43})).toBe(43);
});

it('does not use locals instead of scope when no matching key', function() {
  var fn = parse('aKey');
  expect(fn({aKey: 42}, {otherKey: 43})).toBe(42);
});
```

The function returned by `simpleGetterFn1` should now take two arguments: `scope` and `locals`. It should try to look up from `locals` first:

src/parse.js

```
var simpleGetterFn1 = function(key) {
  return function(scope, locals) {
    if (!scope) {
      return undefined;
    }
    return (locals && locals.hasOwnProperty(key)) ? locals[key] : scope[key];
  };
};
```

The story is similar for two-part keys:

test/parse_spec.js

```
it('uses locals when a 2-part key matches in locals', function() {
  var fn = parse('aKey.anotherKey');
  expect(fn(
    {aKey: {anotherKey: 42}},
    {aKey: {anotherKey: 43}}
  )).toBe(43);
});

it('does not use locals when a 2-part key does not match', function() {
  var fn = parse('aKey.anotherKey');
  expect(fn(
    {aKey: {anotherKey: 42}},
    {otherKey: {anotherKey: 43}}
  )).toBe(42);
});
```


The locals vs. scope rule only applies to the *first* part of the key, though. If the first part of the key matches in `locals`, that is where the lookup will be done, even if the second part does not match:

test/parse_spec.js

```
it('uses locals instead of scope when the first part matches', function() {
  var fn = parse('aKey.anotherKey');
  expect(fn({aKey: {anotherKey: 42}}, {aKey: {}})).toBeUndefined();
});
```

The implementation in `simpleGetterFn2` is similar to what we did in `simpleGetterFn1`:

src/parse.js

```
var simpleGetterFn2 = function(key1, key2) {
  return function(scope, locals) {
    if (!scope) {
      return undefined;
    }
    scope = (locals && locals.hasOwnProperty(key1)) ?
              locals[key1] :
              scope[key1];
    return scope ? scope[key2] : undefined;
  };
};
```

Naturally, the locals rule also applies to arbitrarily nested keys, but here also only on the first level:

test/parse_spec.js

```
it('uses locals when there is a matching local 4-part key', function() {
  var fn = parse('aKey.key2.key3.key4');
  expect(fn(
    {aKey: {key2: {key3: {key4: 42}}}},
    {aKey: {key2: {key3: {key4: 43}}}}
  )).toBe(43);
});

it('uses locals when there is the first part in the local key', function() {
  var fn = parse('aKey.key2.key3.key4');
  expect(fn(
    {aKey: {key2: {key3: {key4: 42}}}},
    {aKey: {}}
  )).toBeUndefined();
```



```
});

it('does not use locals when there is no matching 4-part key', function() {
  var fn = parse('aKey.key2.key3.key4');
  expect(fn(
    {aKey: {key2: {key3: {key4: 42}}}},
    {otherKey: {anotherKey: 43}}
  )).toBe(42);
});
```

In the generated code we need to do the locals check, but only on the *first* key part. Within the loop we can check whether we are on the first key by looking at the second argument `_.forEach` gives to the iterator function - the current array index:

src/parse.js

```
var generatedGetterFn = function(keys) {
  var code = '';
  _.forEach(keys, function(key, idx) {
    code += 'if (!scope) { return undefined; }\n';
    if (idx === 0) {
      code += 'scope = (locals && locals.hasOwnProperty("'+key+'")) ? '+
              'locals["' + key + '"] : '+
              'scope["' + key + '"];\n';
    } else {
      code += 'scope = scope["' + key + '"];\n';
    }
  });
  code += 'return scope;\n';
  /* jshint -W054 */
  return new Function('scope', 'locals', code);
  /* jshint +W054 */
};
```

Note that the generated function takes two arguments now, and we need to reflect that in the `Function` constructor call.

Square Bracket Property Access

We've seen how you can access attributes on scopes using the dot operator. The other way you can do that in Angular expressions (as you can in JavaScript) is by using square bracket notation:

test/parse_spec.js

```
it('parses a simple string property access', function() {
  var fn = parse('aKey["anotherKey"]');
  expect(fn({aKey: {anotherKey: 42}})).toBe(42);
});
```

The same notation also works with arrays. You just use numbers instead of strings as the key:

test/parse_spec.js

```
it('parses a numeric array access', function() {
  var fn = parse('anArray[1]');
  expect(fn({anArray: [1, 2, 3]})).toBe(2);
});
```

The square bracket notation is perhaps most useful when the key isn't known at parse time, but is itself looked up from the scope. You can't do that with the dot notation:

test/parse_spec.js

```
it('parses a property access with another key as property', function() {
  var fn = parse('lock[key]');
  expect(fn({key: 'theKey', lock: {theKey: 42}})).toBe(42);
});
```

Finally, the notation should be flexible enough to recursively allow more elaborate expressions - such as *other* property accesses - as the key:

test/parse_spec.js

```
it('parses property access with another access as property', function() {
  var fn = parse('lock[keys["aKey"]]');
  expect(fn({keys: {aKey: 'theKey'},  lock: {theKey: 42}})).toBe(42);
});
```

Like array and object literals, and unlike dot property access, a square bracket property access expression consists of multiple tokens. The expression `lock[key]` has four: The identifier token `lock`, a single character token `'['`, the identifier token `key`, and a single character token `']'`. That means we'll need to shift the work to the Parser, whereas with dot property access everything happened in the Lexer.

The expression `lock[key]` is a primary expression and will be parsed in `Parser.primary`. It should be able to handle an opening square bracket after the initial expression:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }

  if (this.expect('[')) {

  }
  return primary;
};
```

Note that a square bracket is now interpreted very differently in different positions. If it is the first character in a primary expression, it denotes an array declaration. If there's something preceding it, it's a property access.

The actual handling of the property access will happen in a new Parser function called `objectIndex`. The initial expression is given to it as an argument, and its return value is used as the final value of the primary expression.

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }

  if (this.expect('[')) {
    primary = this.objectIndex(primary);
  }
  return primary;
};
```


Once we are in this new function, we've already consumed the opening square bracket. What remains is whatever goes inside the square brackets, and then the closing square bracket. The part that goes inside the square brackets is actually another primary expression, so we'll consume that recursively, and then consume the closing square bracket:

src/parse.js

```
Parser.prototype.objectIndex = function(objFn) {
  var indexFn = this.primary();
  this.consume(']');
};
```

We now have the requisite parts of object lookup in place: We have the object expression to do the lookup from, given as an argument to `objectIndex`. We also have the expression that evaluates to the key, which is the one we got from between the square brackets. What remains is the return value of the whole expression which, as always, is a function that takes a scope and a locals object:

src/parse.js

```
Parser.prototype.objectIndex = function(objFn) {
  var indexFn = this.expression();
  this.consume(']');
  return function(scope, locals) {

  };
};
```

What we do in that function is invoke the subexpressions to get the concrete object and key, and then simply do the actual lookup with them:

src/parse.js

```
Parser.prototype.objectIndex = function(objFn) {
  var indexFn = this.primary();
  this.consume(']');
  return function(scope, locals) {
    var obj = objFn(scope, locals);
    var index = indexFn(scope, locals);
    return obj[index];
  };
};
```

The property access works now, but what if you consider several property accesses back to back?

test/parse_spec.js


```
it('parses several field accesses back to back', function() {
  var fn = parse('aKey["anotherKey"]["aThirdKey"]');
  expect(fn({aKey: {anotherKey: {aThirdKey: 42}}})).toBe(42);
});
```

That doesn't work since we're terminating the parse after the first closing square bracket, and never get to the `["aThirdKey"]` part. We can fix this by simply augmenting the `primary` function so that it keeps digging into the object as long as there are opening square brackets to consume. The return value of the previous property access always becomes the object of the next property access:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }

  while (this.expect('[')) {
    primary = this.objectIndex(primary);
  }
  return primary;
};
```

Field Access

Our language can now do field access with the dot operator and object indexing with square brackets, but what we're still missing is nested access using a combination of the two. We're not able to parse dot access after square brackets:

test/parse_spec.js

```
it('parses a field access after a property access', function() {
  var fn = parse('aKey["anotherKey"].aThirdKey');
  expect(fn({aKey: {anotherKey: {aThirdKey: 42}}})).toBe(42);
});
```


Not to mention nesting *another* square bracket access on that:

test/parse_spec.js

```
it('parses a chain of property and field accesses', function() {
  var fn = parse('aKey["anotherKey"].aThirdKey["aFourthKey"]');
  expect(fn({aKey: {anotherKey: {aThirdKey: {aFourthKey: 42}}}})).toBe(42);
});
```

This is because we've gotten away with not implementing dot characters as tokens at all, since we collapse multiple identifiers separated by dots into one token and generate a single getter for it. Now that we have a case where the dot operator is not immediately preceded by an identifier, we'll need to emit the dot as a token and then handle it in the parser.

Doing the token emission is simple - we just add it to the list of characters the Lexer emits as-is:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.is('.') && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.is('\'"')) {
      this.readString(this.ch);
    } else if (this.is('[],{}:.')) {
      this.tokens.push({
        text: this.ch,
        json: false
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent(this.ch);
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```


Now, in the `while` loop in which we're unrolling the square bracket accesses, we'll also unroll dotted field accesses. Basically, we allow a primary expression to be an arbitrary sequence of dot and square bracket accesses:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }

  var next;
  while ((next = this.expect('[', '.'))) {
    if (next.text === '[') {
      primary = this.objectIndex(primary);
    } else if (next.text === '.') {
      primary = this.fieldAccess(primary);
    }
  }

  return primary;
};
```

We now expect *either* an opening square bracket or a dot, and delegate to a different parser method based on which it is.

We also need to extend `expect` (and `peek`) so that it takes several alternatives for what token to expect next. Let's go ahead and bump the number of alternative tokens up to four:

src/parse.js

```
Parser.prototype.peek = function(e1, e2, e3, e4) {
  if (this.tokens.length > 0) {
    var text = this.tokens[0].text;
    if (text === e1 || text === e2 || text === e3 || text === e4 ||
        (!e1 && !e2 && !e3 && !e4)) {
      return this.tokens[0];
```



```
    }
  }
};

Parser.prototype.expect = function(e1, e2, e3, e4) {
  var token = this.peek(e1, e2, e3, e4);
  if (token) {
    return this.tokens.shift();
  }
};
```

The new `fieldAccess` method does something similar to `objectIndex`. It takes an expression that represents the source object of the lookup. It then consumes the next token (after the `.`), which we expect to be an identifier token. Earlier in the chapter we saw how identifier tokens have getter functions attached to their `fn` attributes. Those getters are now used to do the lookup:

src/parse.js

```
Parser.prototype.fieldAccess = function(objFn) {
  var getter = this.expect().fn;
  return function(scope, locals) {
    var obj = objFn(scope, locals);
    return getter(obj);
  };
};
```

> Note that here the getter is not given the original scope as an argument. Instead it gets the result of `objFn`, which is some object nested arbitrarily deep in a data structure. Many of the expression functions can be used this way, and that is why you'll sometimes see the first argument of an expression function being called `self` instead of `scope`.

Now we're able to dig into arbitrary depths in objects and arrays, using any combination of square bracket and dot notation.

Function Calls

In Angular expressions it is very common to not only look things up, but also to invoke functions:

test/parse_spec.js

```
it('parses a function call', function() {
  var fn = parse('aFunction()');
  expect(fn({aFunction: function() { return 42; }})).toBe(42);
});
```


One thing to understand about function calls is that there are really two things going on: First you look up the function to call, as with `'aFunction'` in the expression above, and then you call that function by using parentheses. The lookup part is no different from any other attribute lookup. After all, in JavaScript, functions are no different from other values.

This means that we can use the code we already have to look the function up, and what remains to be done is the invocation. Again, let's begin by adding the parentheses as character tokens that the Lexer will emit:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.is('.') && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.is('\'"')) {
      this.readString(this.ch);
    } else if (this.is('[],{}:.()')) {
      this.tokens.push({
        text: this.ch,
        json: false
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent(this.ch);
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

Function calls are handled by the primary expression function, just like property accesses. In the `while` loop in `Parser.primary` we'll consume not only square brackets and dots, but also opening parentheses. When we come across one, we delegate to a method called `functionCall`:

src/parse.js


```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }

  var next;
  while ((next = this.expect('[', '.', '('))) {
     if (next.text === '[') {
      primary = this.objectIndex(primary);
    } else if (next.text === '.') {
      primary = this.fieldAccess(primary);
    } else if (next.text === '(') {
      primary = this.functionCall(primary);
    }
  }

  return primary;
};
```

The very simple function call we have in our first test can be handled by consuming the closing parenthesis, obtaining the function to call using the preceding expression, and finally invoking it:

src/parse.js

```
Parser.prototype.functionCall = function(fnFn) {
  this.consume(')');
  return function(scope, locals) {
    var fn = fnFn(scope, locals);
    return fn();
  };
};
```

Of course, most function calls are not as simple as the one we saw earlier. What you often do with functions is pass arguments, and our naïve function implementation currently knows nothing about such things.

We should be able to handle simple parameters like integers:

test/parse_spec.js


```
it('parses a function call with a single number argument', function() {
  var fn = parse('aFunction(42)');
  expect(fn({aFunction: function(n) { return n; }})).toBe(42);
});
```

Parameters that look up something else from the scope:

test/parse_spec.js

```
it('parses a function call with a single identifier argument', function() {
  var fn = parse('aFunction(n)');
  expect(fn({n: 42, aFunction: function(arg) { return arg; }})).toBe(42);
});
```

Parameters that are function calls themselves:

test/parse_spec.js

```
it('parses a function call with a single function call argument', function() {
  var fn = parse('aFunction(argFn())');
  expect(fn({
    argFn: _.constant(42),
    aFunction: function(arg) { return arg; }
  })).toBe(42);
});
```

And combinations of the above as multiple arguments separated with commas:

test/parse_spec.js

```
it('parses a function call with multiple arguments', function() {
  var fn = parse('aFunction(37, n, argFn())');
  expect(fn({
    n: 3,
    argFn: _.constant(2),
    aFunction: function(a1, a2, a3) { return a1 + a2 + a3; }
  })).toBe(42);
});
```

In the `functionCall` method, if the next token is not the closing parenthesis - in other words, if there is something between the opening and closing parentheses - we should do something about it:

src/parse.js

```
Parser.prototype.functionCall = function(fnFn) {
  if (!this.peek(')')) {

  }
  this.consume(')');
  return function(scope, locals) {
    var fn = fnFn(scope, locals);
    return fn();
  };
};
```

What we do is read in *other* primary expressions as long as there are commas to be consumed. Each of those primary expressions is added to an array of argument expression functions. Just like what we did when parsing array elements:

src/parse.js

```
Parser.prototype.functionCall = function(fnFn) {
  var argFns = [];
  if (!this.peek(')')) {
    do {
      argFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(')');
  return function(scope, locals) {
    var fn = fnFn(scope, locals);
    return fn();
  };
};
```

When the expression is finally evaluated on some scope, we need to execute each of those argument expressions to obtain the concrete arguments for the function:

src/parse.js

```
Parser.prototype.functionCall = function(fnFn) {
  var argFns = [];
  if (!this.peek(')')) {
    do {
      argFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(')');
  return function(scope, locals) {
    var fn = fnFn(scope, locals);
    var args = _.map(argFns, function(argFn) {
      return argFn(scope, locals);
    });
    return fn.apply(null, args);
  };
};
```


We use the JavaScript `Function.apply` to convert our array of arguments to the sequence of actual function arguments. We're also binding the `this` keyword to `null`. As we'll soon see, that won't always be the case.

Ensuring Safety In Member Access

Since expressions are most often used within HTML and also often combined with user-generated content, it is important to do everything we can to prevent injection attacks, where users could execute arbitrary code by crafting particular kinds of expressions. The protection against this is mostly based on the fact that all expressions are strictly scoped on Scope objects by the Parser and Lexer: Apart from literals you can only work on things that have been attached to the scope. The objects with potentially dangerous operations, such as `window` simply aren't accessible.

There is one particular way around this though in our current implementation: In this chapter we have already seen how the JavaScript `Function` constructor takes a string and evaluates that string as the source code of a new function. We used it to generate getters. It turns out that if we don't take measures to prevent it, that same Function constructor can be used to evaluate arbitrary code in an expression.

The Function constructor is made available to an attacker by the fact that it is attached to the `constructor` attribute of every JavaScript function. If you have a function on the scope, as you often do, you can take its constructor in an expression, pass it a string of JavaScript code, and then execute the resulting function. At that point, all bets are off. What we would like to happen instead is that trying to use the function constructor throws an exception:

test/parse_spec.js

```
it('does not allow calling the function constructor', function() {
  expect(function() {
    var fn = parse('aFunction.constructor("return window;")()');
    fn({aFunction: function() { }});
  }).toThrow();
});
```

The security measure we'll take against these kinds of attacks is to simply disallow accessing any members on any objects that are called `constructor`. To that effect, we'll introduce a helper function that checks the name of a member and throws an exception if it's `constructor`:

src/parse.js

```
var ensureSafeMemberName = function(name) {
  if (name === 'constructor') {
    throw 'Referencing "constructor" field in expressions is disallowed!';
  }
};
```


Now we need to sprinkle calls to this function throughout the parsing code, in all locations where we are accessing attributes of scopes, locals, or nested objects. In simple getters of single attributes:

src/parse.js

```
var simpleGetterFn1 = function(key) {
  ensureSafeMemberName(key);
  return function(scope, locals) {
    if (!scope) {
      return undefined;
    }
    return (locals && locals.hasOwnProperty(key)) ? locals[key] : scope[key];
  };
};
```

In two-part attribute getters (this is what our test case actually hits):

src/parse.js

```
var simpleGetterFn2 = function(key1, key2) {
  ensureSafeMemberName(key1);
  ensureSafeMemberName(key2);
  return function(scope, locals) {
    if (!scope) {
      return undefined;
    }
    scope = (locals && locals.hasOwnProperty(key1)) ?
              locals[key1] :
              scope[key1];
    return scope ? scope[key2] : undefined;
  };
};
```

In arbitrarily nested attribute getters:

src/parse.js

```
var generatedGetterFn = function(keys) {
  var code = '';
  _.forEach(keys, function(key, idx) {
    ensureSafeMemberName(key);
    code += 'if (!scope) { return undefined; }\n';
    if (idx === 0) {
      code += 'scope = (locals && locals.hasOwnProperty("'+key+'")) ? '+
                'locals["' + key + '"] : '+
```



```
                'scope["' + key + '"];\n';
    } else {
      code += 'scope = scope["' + key + '"];\n';
    }
  });
  code += 'return scope;\n';
  /* jshint -W054 */
  return new Function('scope', 'locals', code);
  /* jshint +W054 */
};
```

We're still not covering square bracket property access here. Since it allows accessing variables by dynamic names, we often simply won't know at parse time whether they happen to access the Function constructor. For these cases there's another security measure, which we'll see a bit later in the chapter.

Method Calls

In JavaScript, a function call is not always just a function call. When a function is attached to an object as an attribute and invoked by first dereferencing it from the object using a dot or square brackets, the function's body will have the `this` keyword bound to the containing object. So, in this test case, `this` in `aFunction` should point to `aMember` because of the way we call it in the expression:

src/parse.js

```
it('calls functions accessed as properties with the correct this', function() {
  var scope = {
    anObject: {
      aMember: 42,
      aFunction: function() {
        return this.aMember;
      }
    }
  };
  var fn = parse('anObject["aFunction"]()');
  expect(fn(scope)).toBe(42);
});
```

Since we're explicitly using `null` in the call to `apply` in `functionCall`, this test fails. We should find a way to replace the `null` with the value of `anObject` instead.

Much of the magic happens in the `while` loop of `Parser.primary`. As we dig deeper and deeper to the data structure in that loop using dot and square bracket accessors, we're keeping track of the last thing seen by storing it in the local variable `primary`. For method calls, we should also introduce another local variable for storing the *second to last* thing

seen. If there's a method call in the expression, this second to last part of the expression will be the `this`, or the *context*, of that method call.

In the case of square bracket property access, we'll first make the current `primary` the `context` and then replace the `primary`. Then, in a later function call, we can pass in `context` to `functionCall`:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }

  var next;
  var context;
  while ((next = this.expect('[', '.', '('))) {
     if (next.text === '[') {
       context = primary;
       primary = this.objectIndex(primary);
    } else if (next.text === '.') {
       primary = this.fieldAccess(primary);
    } else if (next.text === '(') {
       primary = this.functionCall(primary, context);
    }
  }
  return primary;
};
```

In `functionCall` we can then use the context. We first evaluate it to a concrete value, and then use it as the first argument in `apply`ing the function:

src/parse.js

```
Parser.prototype.functionCall = function(fnFn, contextFn) {
  var argFns = [];
  if (!this.peek(')')) {
    do {
      argFns.push(this.primary());
    } while (this.expect(','));
```



```
  }
  this.consume(')');
  return function(scope, locals) {
    var context = contextFn ? contextFn(scope, locals) : scope;
    var fn = fnFn(scope, locals);
    var args = _.map(argFns, function(argFn) {
      return argFn(scope, locals);
    });
    return fn.apply(context, args);
  };
};
```

> Note that when a contextFn has not been given, that is, in the case where we're not dealing with a method call but a plain function call, we use `scope` as the value of `this`. This is generally true in functions invoked in Angular expressions: If they're not called as methods, `this` will point to the scope object.

We've got methods invoked through square bracket notation working, but of course the more common way to do it is to use dot notation:

test/parse_spec.js

```
it('calls functions accessed as fields with the correct this', function() {
  var scope = {
    anObject: {
      aMember: 42,
      aFunction: function() {
        return this.aMember;
      }
    }
  };
  var fn = parse('anObject.aFunction()');
  expect(fn(scope)).toBe(42);
});
```

This one is a little bit more tricky to implement, and that is because of how we are handling identifiers with dots in them. Recall that when we come across `anObject.aFunction`, we handle it all as one token, with a 2-part getter generated by Lexer. While this gives us a nicely performant implementation for attribute lookup, it breaks the context of method calls. The parser should use `anObject` as the context, but it never even sees it.

What we need to do is look out for method calls in `Lexer.readIdent` where we construct those multi-part tokens. If there's a method call (denoted by an opening parenthesis) in the expression, we should *not* emit the all-in-one token. We should instead emit all the parts of the expression as separate tokens: The identifier `anObject`, the text `'.'`, and the identifier `aFunction`.

Firstly, while we are constructing the text of the token, we need to keep track of two indexes: The index of the first character in the token, and the index of the last dot we have seen in it. This lets us later split the token if it turns out to be a method call:

src/parse.js


```
Lexer.prototype.readIdent = function() {
  var text = '';
  var start = this.index;
  var lastDotAt;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === '.' || this.isIdent(ch) || this.isNumber(ch)) {
      if (ch === '.') {
        lastDotAt = this.index;
      }
      text += ch;
    } else {
      break;
    }
    this.index++;
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
    token.json = true;
  } else {
    token.fn = getterFn(text);
  }

  this.tokens.push(token);
};
```

Once we've finished consuming the characters for the token, we'll see if it is actually part of a method call expression. Two criteria must be met for that to be the case: There must have been a dot in the token, and the first character *after* the token must be an opening parenthesis. If that is indeed the case, we extract the method name - the part of the token *after* the last dot, and set the text of the token itself to just include the part *before* the last token. For this we make use of the character indexes `start` and `lastDotAt`:

src/parse.js

```
Lexer.prototype.readIdent = function() {
  var text = '';
  var start = this.index;
  var lastDotAt;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === '.' || this.isIdent(ch) || this.isNumber(ch)) {
      if (ch === '.') {
        lastDotAt = this.index;
      }
      text += ch;
```



```
    } else {
      break;
    }
    this.index++;
  }

  var methodName;
  if (lastDotAt) {
    if (this.text.charAt(this.index) === '(') {
      methodName = text.substring(lastDotAt - start + 1);
      text = text.substring(0, lastDotAt - start);
    }
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
    token.json = true;
  } else {
    token.fn = getterFn(text);
  }

  this.tokens.push(token);
};
```

Note that we do not consume the opening parenthesis here, but just see if it is there. The actual consumption will be done in a later turn of the loop in `lex`.

Then, if this was indeed part of a method call, we should emit two additional tokens after the actual identifier token: A dot and an identifier token for the method name:

src/parse.js

```
Lexer.prototype.readIdent = function() {
  var text = '';
  var start = this.index;
  var lastDotAt;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === '.' || this.isIdent(ch) || this.isNumber(ch)) {
      if (ch === '.') {
        lastDotAt = this.index;
      }
      text += ch;
    } else {
      break;
    }
    this.index++;
  }
```

```
  var methodName;
  if (lastDotAt) {
    if (this.text.charAt(this.index) === '(') {
      methodName = text.substring(lastDotAt - start + 1);
      text = text.substring(0, lastDotAt - start);
    }
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
    token.json = true;
  } else {
    token.fn = getterFn(text);
  }

  this.tokens.push(token);

  if (methodName) {
    this.tokens.push({
      text: '.',
      json: false
    });
    this.tokens.push({
      text: methodName,
      fn: getterFn(methodName),
      json: false
    });
  }
};
```

These three tokens will be picked up by the `while` loop in the Parser's `primary` function and handled as a field access followed by a function call. All we need to do to make it work as a method call is assign the `context` variable on dotted field access, just like we did with square bracket field access:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
```



```
      primary.literal = true;
    }
  }

  var next;
  var context;
  while ((next = this.expect('[', '.', '('))) {
     if (next.text === '[') {
      context = primary;
      primary = this.objectIndex(primary);
    } else if (next.text === '.') {
      context = primary;
      primary = this.fieldAccess(primary);
    } else if (next.text === '(') {
      primary = this.functionCall(primary, context);
    }
  }
  return primary;
};
```

In Angular expressions, like in JavaScript, it is also legal to have some whitespace between the name of a function and the parentheses that call it. This already works in our implementation since we have the Lexer ignore all whitespace between tokens. What does not work, however, is having this kind of whitespace in a method call:

test/parse_spec.js

```
it('calls methods with whitespace before function call', function() {
  var scope = {
    anObject: {
      aMember: 42,
      aFunction: function() {
        return this.aMember;
      }
    }
  };
  var fn = parse('anObject.aFunction  ()');
  expect(fn(scope)).toBe(42);
});
```

This is due to the fact that in `readIdent`, when we check if the identifier is followed by a parenthesis, we're just looking at the next character, which might be whitespace. To fix this we just need to look over any whitespace following the identifier, and only consider the first character after that:

src/parse.js

```
Lexer.prototype.readIdent = function() {
  var text = '';
  var start = this.index;
  var lastDotAt;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === '.' || this.isIdent(ch) || this.isNumber(ch)) {
      if (ch === '.') {
        lastDotAt = this.index;
      }
      text += ch;
    } else {
      break;
    }
    this.index++;
  }

  var methodName;
  if (lastDotAt) {
    var peekIndex = this.index;
    while (this.isWhitespace(this.text.charAt(peekIndex))) {
      peekIndex++;
    }
    if (this.text.charAt(peekIndex) === '(') {
      methodName = text.substring(lastDotAt - start + 1);
      text = text.substring(0, lastDotAt - start);
    }
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
    token.json = true;
  } else {
    token.fn = getterFn(text);
  }

  this.tokens.push(token);

  if (methodName) {
    this.tokens.push({
      text: '.',
      json: false
    });
    this.tokens.push({
      text: methodName,
      fn: getterFn(methodName),
      json: false
    });
  }
};
```


We're almost done with method calls, but there's still one more corner case to be considered. We are tracking the `this` context in `Parser.primary`, but we're doing it just a bit too persistently. Consider the following case with two consecutive function calls. The first function returns another function, which is also immediately called:

test/parse_spec.js

```
it('clears the this context on function calls', function() {
  var scope = {
    anObject: {
      aMember: 42,
      aFunction: function() {
        return function() {
          return this.aMember;
        };
      }
    }
  };
  var fn = parse('anObject.aFunction()()');
  expect(fn(scope)).toBeUndefined();
});
```

In the second function, `this` is still bound to `anObject`. This would not be the case in JavaScript, and it should also not be the case in Angular expressions. The `this` context only applies to the function invoked immediately after attribute lookup.

The `this` context must be explicitly cleared when a function call happens in `Parser.primary`, so that any subsequent function calls aren't contaminated:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }

  var next;
  var context;
```



```
  while ((next = this.expect('[', '.', '('))) {
     if (next.text === '[') {
      context = primary;
      primary = this.objectIndex(primary);
    } else if (next.text === '.') {
      context = primary;
      primary = this.fieldAccess(primary);
    } else if (next.text === '(') {
      primary = this.functionCall(primary, context);
      context = undefined;
    }
  }
  return primary;
};
```

Ensuring Safe Objects

Earlier we discussed the dangers of script injection in Angular expressions and did some work to prevent anyone from invoking the Function constructor directly from an expression. Now we're ready to look at the second security measure Angular expressions do for us. That really has to do with protecting application developers from themselves, by not letting them attach dangerous things on scopes and then access them with expressions.

One of these dangerous objects is `window`. You could do great damage by calling some of the functions attached to `window`, so what Angular does is it completely prevents you from using it in expressions. Of course, you can't just call `window` members directly anyway since expressions only work on scopes, but you should also not be able to alias `window` as a scope attribute. If you were to try that, an exception should be thrown:

test/parse_spec.js

```
it('does not allow accessing window as property', function() {
  var fn = parse('anObject["wnd"]');
  expect(function() { fn({anObject: {wnd: window}}); }).toThrow();
});
```

The security measure against this is that when dealing with objects, we should check that they're not dangerous objects. We'll first introduce a helper function for this purpose:

src/parse.js

```
var ensureSafeObject = function(obj) {
  if (obj) {
    if (obj.document && obj.location && obj.alert && obj.setInterval) {
      throw 'Referencing window in Angular expressions is disallowed!';
    }
  }
  return obj;
};
```


Since there's no built-in reliable, cross-platform check for the "windowness" of an object (and just comparing to `window` with `===` would not work consistently when there are iframes around), we do the best we can by checking for a few attributes only `window` usually has: `document`, `location`, `alert`, and `setInterval`.

To make the test pass we should now call this function in `objectIndex` to check that the object referred to by the lookup is a safe one:

src/parse.js

```
Parser.prototype.objectIndex = function(objFn) {
  var indexFn = this.primary();
  this.consume(']');
  return function(scope, locals) {
    var obj = objFn(scope, locals);
    var index = indexFn(scope, locals);
    return ensureSafeObject(obj[index]);
  };
};
```

One should also not be able to *call a function* on `window`, if it happened to be attached on the scope:

test/parse_spec.js

```
it('does not allow calling functions of window', function() {
  var fn = parse('wnd.scroll(500, 0)');
  expect(function() { fn({wnd: window}); }).toThrow();
});
```

In this case what we should check is the context of the method call:

src/parse.js

```
Parser.prototype.functionCall = function(fnFn, contextFn) {
  var argFns = [];
  if (!this.peek(')')) {
    do {
      argFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(')');
  return function(scope, locals) {
    var context = ensureSafeObject(contextFn ? contextFn(scope, locals) : scope);
    var fn = fnFn(scope, locals);
    var args = _.map(argFns, function(argFn) {
      return argFn(scope, locals);
    });
    return fn.apply(context, args);
  };
};
```


So functions cannot be called *on* `window`, but it should also not be possible to call functions that *return* `window`:

test/parse_spec.js

```
it('does not allow functions to return window', function() {
  var fn = parse('getWnd()');
  expect(function() { fn({getWnd: _.constant(window)}); }).toThrow();
});
```

To prevent this there needs to be a check for the return value of the function call as well:

src/parse.js

```
Parser.prototype.functionCall = function(fnFn, contextFn) {
  var argFns = [];
  if (!this.peek(')')) {
    do {
      argFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(')');
  return function(scope, locals) {
    var context = ensureSafeObject(contextFn ? contextFn(scope, locals) : scope);
    var fn = fnFn(scope, locals);
    var args = _.map(argFns, function(argFn) {
      return argFn(scope, locals);
    });
    return ensureSafeObject(fn.apply(context, args));
  };
};
```

`window` is not the only dangerous object we should be looking out for. Another one is DOM elements. Having access to a DOM element would make it possible for an attacker to traverse and manipulate the contents of the web page, so they should also be forbidden:

test/parse_spec.js

```
it('does not allow calling functions on DOM elements', function() {
  var fn = parse('el.setAttribute("evil", "true")');
  expect(function() { fn({el: document.documentElement}); }).toThrow();
});
```

AngularJS implements the following check for the "DOM-elementness" of an object:

test/parse.js


```
var ensureSafeObject = function(obj) {
  if (obj) {
    if (obj.document && obj.location && obj.alert && obj.setInterval) {
      throw 'Referencing window in Angular expressions is disallowed!';
    } else if (obj.children &&
               (obj.nodeName || (obj.prop && obj.attr && obj.find))) {
      throw 'Referencing DOM nodes in Angular expressions is disallowed!';
    }
  }
  return obj;
};
```

The final dangerous object we'll consider is our old friend, the function constructor. While we're already making sure no one obtains the constructor by using the `constructor` property of functions, there's nothing to prevent someone from aliasing a function constructor on the scope with some other name:

test/parse_spec.js

```
it('does not allow calling the aliased function constructor', function() {
  var fn = parse('fnConstructor("return window;")');
  expect(function() {
    fn({fnConstructor: (function() { }).constructor});
  }).toThrow();
});
```

The check for this is quite a bit simpler than that of `window` and DOM element: The function `constructor` is also a function, so it'll also have a `constructor` property - one that points to itself.

src/parse.js

```
var ensureSafeObject = function(obj) {
  if (obj) {
    if (obj.document && obj.location && obj.alert && obj.setInterval) {
      throw 'Referencing window in Angular expressions is disallowed!';
    } else if (obj.children &&
               (obj.nodeName || (obj.prop && obj.attr && obj.find))) {
      throw 'Referencing DOM nodes in Angular expressions is disallowed!';
    } else if (obj.constructor === obj) {
      throw "Referencing Function in Angular expressions is disallowed!";
    }
  }
  return obj;
};
```


In this case the dangerous object isn't the function's context, nor is it the function's return value. It is the function itself. So we'll need one more call to `ensureSafeObject` for function calls:

src/parse.js

```
Parser.prototype.functionCall = function(fnFn, contextFn) {
  var argFns = [];
  if (!this.peek(')')) {
    do {
      argFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(')');
  return function(scope, locals) {
    var context = ensureSafeObject(contextFn ? contextFn(scope, locals) : scope);
    var fn = ensureSafeObject(fnFn(scope, locals));
    var args = _.map(argFns, function(argFn) { return argFn(scope, locals); });
    return ensureSafeObject(fn.apply(context, args));
  };
};
```

This is how Angular attempts to secure expressions from injection attacks. The security measures are by no means perfect, and there are almost certainly several ways to attach dangerous members to scopes even with these checks in place. However, the risk related to this is greatly diminished by the fact that before an attacker can use these dangerous members the application developer has to put them on the scope, which is something they should not be doing.

Assigning Values

Now we are going to take a look at how expressions can not only *access* data on scopes but also *put* data on scopes by using *assignments*. For example, it is perfectly legal for an expression to set a scope attribute to some value:

test/parse_spec.js

```
it('parses a simple attribute assignment', function() {
  var fn = parse('anAttribute = 42');
  var scope = {};
  fn(scope);
  expect(scope.anAttribute).toBe(42);
});
```

The value assigned does not have to be a simple literal either. It can be any primary expression, such as a function call:

test/parse_spec.js


```
it('can assign any primary expression', function() {
  var fn = parse('anAttribute = aFunction()');
  var scope = {aFunction: _.constant(42)};
  fn(scope);
  expect(scope.anAttribute).toBe(42);
});
```

Just as with most of the other new features in this chapter, we'll begin by having the Lexer emit a token for the Parser to use. This time we'll need one for the = sign that denotes assignments:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.is('.') && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.is('\'"')) {
      this.readString(this.ch);
    } else if (this.is('[],{}:.()=')) {
      this.tokens.push({
        text: this.ch,
        json: false
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent(this.ch);
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      throw 'Unexpected next character: '+this.ch;
    }
  }

  return this.tokens;
};
```

Assignment is not what we've been calling a "primary" expression, and it'll not be parsed by the existing `Parser.primary` function. It'll have a function of its own, which we'll call `assignment`.

We begin parsing an assignment by consuming the left hand side token, which is a primary expression. The left hand side is then followed by an equals sign token, and then the right hand side, which is another primary expression:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.primary();
  if (this.expect('=')) {
    var right = this.primary();

  }
  return left;
};
```

Given the left and right sides of the assignment, we return a function that will execute the assignment:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.primary();
  if (this.expect('=')) {
    var right = this.primary();
    return function(scope, locals) {

    };
  }
  return left;
};
```

What should this function actually do? We have the target expression of the assignment and we have the value expression to assign. But we have no means to actually do the assignment yet. The `left` expression just returns the value of something. It doesn't let us *replace* that value.

When an expression allows assignment, it will need to expose a special function that executes the assignment. This function will be stored on the expression object in a new attribute called `assign`, right next to the attributes for `literal` and `constant`.

The simplest kind of expression that allows assignment is a simple identifier expression. Recall that these are the expressions we construct right in the Lexer, in the `readIdent` function. They are the `fn` attributes we attach to identifier tokens. These `fn` attributes will need to have `assign` attributes so that we can set the value of the identifier:

src/parse.js


```
Lexer.prototype.readIdent = function() {
  var text = '';
  var start = this.index;
  var lastDotAt;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (ch === '.' || this.isIdent(ch) || this.isNumber(ch)) {
      if (ch === '.') {
        lastDotAt = this.index;
      }
      text += ch;
    } else {
      break;
    }
    this.index++;
  }

  var methodName;
  if (lastDotAt) {
    var peekIndex = this.index;
    while (this.isWhitespace(this.text.charAt(peekIndex))) {
      peekIndex++;
    }
    if (this.text.charAt(peekIndex) === '(') {
      methodName = text.substring(lastDotAt - start + 1);
      text = text.substring(0, lastDotAt - start);
    }
  }

  var token = {text: text};
  if (OPERATORS.hasOwnProperty(text)) {
    token.fn = OPERATORS[text];
    token.json = true;
  } else {
    token.fn = getterFn(text);
    token.fn.assign = function(self, value) {
      return setter(self, text, value);
    };
  }

  this.tokens.push(token);

  if (methodName) {
    this.tokens.push({
      text: '.',
      json: false
    });
    this.tokens.push({
      text: methodName,
      fn: getterFn(methodName),
      json: false
```

```
    });
  }
};
```

An assignment in an expression has three components:

- The object that'll hold the new attribute. This may be a Scope or some nested object attached to it.
- The name of the attribute to assign in that object.
- The new value of that attribute.

What the `assign` function takes as arguments are the first and the third components: `self` is the object that'll hold the attribute and `value` is the value. The name of the attribute is already known at parse time. It is the text of the token being parsed.

The `assign` function delegates to a helper function called `setter`, passing all the three components to it. `setter` then does the actual assignment work:

src/parse.js

```
var setter = function(object, key, value) {
  ensureSafeMemberName(key);
  object[key] = value;
  return value;
};
```

We make sure that we're not reassigning a dangerous member, and then we do the assignment. Finally we return the value that was assigned, since that is also what assignment expressions in JavaScript do.

Note that `setter` does not use locals. Assignments are never done on the locals object, but always on scope or its nested members.

We can now complete the circle by returning to the `assignment` function of `Parser`. Here we can look up the `assign` attribute of the left hand side of the assignment and invoke it with the scope and the value of the right hand side expression. If there is no `assign` member on the left token, that means we're trying to assign to something that isn't assignable and we should throw an exception:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.primary();
  if (this.expect('=')) {
    if (!left.assign) {
      throw 'Implies assignment but cannot be assigned to';
    }
    var right = this.primary();
    return function(scope, locals) {
      return left.assign(scope, right(scope, locals));
    };
  }
  return left;
};
```


The remaining issue is that `assignment` is currently an "orphan" function: It is not called by anything. The main function of `Parser` just consumes one primary expression and then exits.

Notice how we've constructed `assignment` so that it checks if there is an equals sign following the left hand side. If there is no equals sign it just returns the left hand side by itself. That means we can use the `assignment` function to parse either an assignment expression or just a plain primary expression. This pattern of trying to parse something and falling through to something else is something we'll see a lot more in the next chapter. Right now we can just replace the call to `primary` in `parse` with a call to `assignment`, and that'll take care of everything we've implemented so far:

src/parse.js

```
Parser.prototype.parse = function(text) {
  this.tokens = this.lexer.lex(text);
  return this.assignment();
};
```

We have a passing test suite, but let's walk through this once more because there's so much going on, and the code interleaves things that happen at parse time with things that happen at evaluation time. Here's what happens when you parse an assignment expression:

1. The parser's `parse` function is called with the expression `"anAttribute = 42"`
2. The parser calls the lexer and the lexer returns three tokens: An identifier token `anAttribute`, a plain text token `=`, and a literal token `42`.
   - `anAttribute` has an `fn` attribute that is a generated getter function. `fn` has a further `assign` attribute which is a generated setter function.
   - `=` is just a text token with no functions attached
   - `42` has an `fn` attribute that returns the constant number 42.
3. The parser moves to `assignment`
4. `assignment` consumes a primary expression. It will end up being the `fn` attribute of the `anAttribute` token. It is stored as the left hand side.
5. `assignment` tries to consume the `=` text token and succeeds.
6. `assignment` consumes another primary expression. It will end up being the `fn` attribute of the `42` token, which is a function that constantly returns 42. It is stored as the right hand side.
7. `assignment` returns a function that takes a scope and a locals object. `parse` then returns it to the caller and it becomes the value of the whole expression.

When you evaluate this expression in the context a scope, the function returned by `assignment` gets invoked with the scope as an argument:

1. It looks up the `assign` function from the left hand side expression. It is the setter created in step 2 above.
2. It evaluates the right hand side expression, and gets `42`.

3. It invokes the `assign` function with the scope and the value `42`.
4. The `assign` function calls `setter` with the scope, the string `"anAttribute"`, and `42`.
5. `setter` sets the attribute `'anAttribute'` on the scope.

The mechanics of assignment are now in place, but we're not quite done yet. For instance, our setter function can currently only handle a single-part key. Just like getters, it should be possible to assign to a nested path as well:

test/parse_spec.js

```
it('parses a nested attribute assignment', function() {
  var fn = parse('anObject.anAttribute = 42');
  var scope = {anObject: {}};
  fn(scope);
  expect(scope.anObject.anAttribute).toBe(42);
});
```

This one we can handle right in `setter`. The setter needs to split the given key into dot-separated parts and dig into the given object in a similar fashion as the getter does. At each step it must also ensure that the member being accessed is safe:

test/parse_spec.js

```
var setter = function(object, path, value) {
  var keys = path.split('.');
  while (keys.length > 1) {
    var key = keys.shift();
    ensureSafeMemberName(key);
    object = object[key];
  }
  object[keys.shift()] = value;
  return value;
};
```

The `while` loop consumes all the path parts except the last one, to obtain the containing object. The final path part is then consumed as the attribute name.

An interesting thing about nested setters like these in Angular expressions is that if some of the objects in the path don't exist, *they are created on the fly*:

test/parse_spec.js

```
it('creates the objects in the setter path that do not exist', function() {
  var fn = parse('some.nested.path = 42');
  var scope = {};
  fn(scope);
  expect(scope.some.nested.path).toBe(42);
});
```


This is in stark contrast to what JavaScript does. JavaScript would just raise an error on `some.nested` since `some` does not exist. The Angular expression engine happily assigns it.

The work for this is also done in `setter`. At each path part, it checks whether the key exists in the containing object. If it doesn't, a new object is created:

src/parse.js

```
var setter = function(object, path, value) {
  var keys = path.split('.');
  while (keys.length > 1) {
    var key = keys.shift();
    ensureSafeMemberName(key);
    if (!object.hasOwnProperty(key)) {
      object[key] = {};
    }
    object = object[key];
  }
  object[keys.shift()] = value;
  return value;
};
```

Angular expressions also allow setting attributes using the square bracket syntax:

test/parse_spec.js

```
it('parses an assignment through attribute access', function() {
  var fn = parse('anObject["anAttribute"] = 42');
  var scope = {anObject: {}};
  fn(scope);
  expect(scope.anObject.anAttribute).toBe(42);
});
```

The square bracket notation is handled by `Parser.objectIndex`. When it is part of an assignment expression, the return value of `Parser.objectIndex` becomes the left hand side. That means the return value of `objectIndex` must have the `assign` property the assignment operation needs:

test/parse_spec.js

```
Parser.prototype.objectIndex = function(objFn) {
  var indexFn = this.primary();
  this.consume(']');
  var objectIndexFn = function(scope, locals) {
    var obj = objFn(scope, locals);
    var index = indexFn(scope, locals);
    return ensureSafeObject(obj[index]);
```

```
  };
  objectIndexFn.assign = function(self, value, locals) {
    var obj = ensureSafeObject(objFn(self, locals));
    var index = indexFn(self, locals);
    return (obj[index] = value);
  };
  return objectIndexFn;
};
```

The `assign` function here first obtains the object to assign to, as well as the name of the attribute to assign. It then plainly assigns the attribute, and returns the return value of the assignment. This is essentially the same operations that `setter` does, but for the square bracket syntax.

Notice that `assign` takes a third argument here: A `locals` object. It needs it because the expression that looks up the object may need it. This means we'll need `assignment` to pass in the `locals` object as well, in case someone needs it:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.primary();
  if (this.expect('=')) {
    if (!left.assign) {
      throw 'Implies assignment but cannot be assigned to';
    }
    var right = this.primary();
    return function(scope, locals) {
      return left.assign(scope, right(scope, locals), locals);
    };
  }
  return left;
};
```

And finally, one can of course also assign into a dotted field access that comes after a square bracket field access:

test/parse_spec.js

```
it('parses assignment through field access after something else', function() {
  var fn = parse('anObject["otherObject"].nested = 42');
  var scope = {anObject: {otherObject: {}}};
  fn(scope);
  expect(scope.anObject.otherObject.nested).toBe(42);
});
```

This type of attribute access is handled by `fieldAccess`, so it is the return value of `fieldAccess` that becomes the left hand side of the assignment. Like we just did in `objectIndex`, we set an `assign` function for the assignment expression to use. This time we can reuse `setter` to do some of the work for us:

test/parse_spec.js

```
Parser.prototype.fieldAccess = function(objFn) {
  var token = this.expect();
  var getter = token.fn;
  var fieldAccessFn = function(scope, locals) {
    var obj = objFn(scope, locals);
    return getter(obj);
  };
  fieldAccessFn.assign = function(self, value, locals) {
    var obj = objFn(self, locals);
    return setter(obj, token.text, value);
  };
  return fieldAccessFn;
};
```

Arrays And Objects Revisited

Now that we can do lookups and assignments, we'll also want our collections - arrays and objects - to work with them. In the previous chapter we implemented *constant* literal collections, and now we're ready add support for *non-constant* literal collections. They are collections that contain non-literals, such as attribute lookups and function calls:

test/parse_spec.js

```
it('parses an array with non-literals', function() {
  var fn = parse('[a, b, c()]');
  expect(fn({a: 1, b: 2, c: _.constant(3)})).toEqual([1, 2, 3]);
});

it('parses an object with non-literals', function() {
  var fn = parse('{a: a, b: obj.c()}');
  expect(fn({
    a: 1,
    obj: {
      b: _.constant(2),
      c: function() {
        return this.b();
      }
    }
  })).toEqual({a: 1, b: 2});
});
```


We're actually remarkably close to having these kinds of expressions supported already. In the last chapter we already parsed array elements and object values as subexpressions. It's just that back then we were still ignoring the fact that expression functions have arguments. Array and object expressions need to take `scope` and `locals` as arguments and pass them along to any subexpressions:

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  var elementFns = [];
  if (!this.peek(']')) {
    do {
      if (this.peek(']')) {
        break;
      }
      elementFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(']');
  var arrayFn = function(scope, locals) {
    var elements = _.map(elementFns, function(elementFn) {
      return elementFn(scope, locals);
    });
    return elements;
  };
  arrayFn.literal = true;
  arrayFn.constant = true;
  return arrayFn;
};

Parser.prototype.object = function() {
  var keyValues = [];
  if (!this.peek('}')) {
    do {
      var keyToken = this.expect();
      this.consume(':');
      var valueExpression = this.primary();
      keyValues.push({
        key: keyToken.string || keyToken.text,
        value: valueExpression
      });
    } while (this.expect(','));
  }
  this.consume('}');
  var objectFn = function(scope, locals) {
    var object = {};
    _.forEach(keyValues, function(kv) {
      object[kv.key] = kv.value(scope, locals);
    });
    return object;
  };
```



```
  objectFn.literal = true;
  objectFn.constant = true;
  return objectFn;
};
```

As we discussed, arrays and objects are not necessarily constant anymore. An array is constant only if it only contains other constants:

test/parse_spec.js

```
it('makes arrays constant when they only contain constants', function() {
  var fn = parse('[1, 2, [3, 4]]');
  expect(fn.constant).toBe(true);
});
```

If an array contains even one non-constant, the whole array is not constant:

test/parse_spec.js

```
it('makes arrays non-constant when they contain non-constants', function() {
  expect(parse('[1, 2, a]').constant).toBe(false);
  expect(parse('[1, 2, [[[[[a]]]]]]').constant).toBe(false);
});
```

The same is true for objects. An object with constants only is also a constant:

test/parse_spec.js

```
it('makes objects constant when they only contain constants', function() {
  var fn = parse('{a: 1, b: {c: 3}}');
  expect(fn.constant).toBe(true);
});
```

Keys in object literals are always constant, but the values might not be. If there's at least one non-constant value in the object, the whole object is non-constant:

test/parse_spec.js

```
it('makes objects non-constant when they contain non-constants', function() {
  expect(parse('{a: 1, b: c}').constant).toBe(false);
  expect(parse('{a: 1, b: {c: d}}').constant).toBe(false);
});
```

For arrays we need to check if all of the element expressions is a constant. Only in that case is the array constant. We can use the LoDash `_.every` function to iterate over the element expressions:

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  var elementFns = [];
  if (!this.peek(']')) {
    do {
      if (this.peek(']')) {
        break;
      }
      elementFns.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(']');
  var arrayFn = function(scope, locals) {
    var elements = _.map(elementFns, function(elementFn) {
      return elementFn(scope, locals);
    });
    return elements;
  };
  arrayFn.literal = true;
  arrayFn.constant = _.every(elementFns, 'constant');
  return arrayFn;
};
```

For objects we do essentially the same check for the values. We use LoDash's `_.pluck` to collect the `value` attribute of each of the key-value pairs, and then invoke `_.every` on that:

src/parse.js

```
Parser.prototype.object = function() {
  var keyValues = [];
  if (!this.peek('}')) {
    do {
      var keyToken = this.expect();
      this.consume(':');
      var valueExpression = this.primary();
      keyValues.push({
        key: keyToken.string || keyToken.text,
        value: valueExpression
      });
    } while (this.expect(','));
  }
  this.consume('}');
  var objectFn = function(scope, locals) {
    var object = {};
    _.forEach(keyValues, function(kv) {
```

```
      object[kv.key] = kv.value(scope, locals);
    });
    return object;
  };
  objectFn.literal = true;
  objectFn.constant = _(keyValues).pluck('value').every('constant');
  return objectFn;
};
```

The final change we'll need to make to collection literals is related to what kind of expressions they may contain. At the moment they may only contain primary expressions, but they should also accept assignments as elements and values like JavaScript does:

test/parse_spec.js

```
it('allows an array element to be an assignment', function() {
  var fn = parse('[a = 1]');
  var scope = {};
  expect(fn(scope)).toEqual([1]);
  expect(scope.a).toBe(1);
});

it('allows an object value to be an assignment', function() {
  var fn = parse('{a: b = 1}');
  var scope = {};
  expect(fn(scope)).toEqual({a: 1});
  expect(scope.b).toBe(1);
});
```

What we need to do is parse the contents of arrays an objects as assignment expressions. That'll take care of both actual assignments and primary expressions (as `assignment` falls through to `primary`):

src/parse.js

```
Parser.prototype.arrayDeclaration = function() {
  var elementFns = [];
  if (!this.peek(']')) {
    do {
      if (this.peek(']')) {
        break;
      }
      elementFns.push(this.assignment());
    } while (this.expect(','));
  }
  this.consume(']');
  var arrayFn = function(scope, locals) {
    var elements = _.map(elementFns, function(elementFn) {
```



```
      return elementFn(scope, locals);
    });
    return elements;
  };
  arrayFn.literal = true;
  arrayFn.constant = _.every(elementFns, 'constant');
  return arrayFn;
};

Parser.prototype.object = function() {
  var keyValues = [];
  if (!this.peek('}')) {
    do {
      var keyToken = this.expect();
      this.consume(':');
      var valueExpression = this.assignment();
      keyValues.push({
        key: keyToken.string || keyToken.text,
        value: valueExpression
      });
    } while (this.expect(','));
  }
  this.consume('}');
  var objectFn = function(scope, locals) {
    var object = {};
    _.forEach(keyValues, function(kv) {
      object[kv.key] = kv.value(scope, locals);
    });
    return object;
  };
  objectFn.literal = true;
  objectFn.constant = _(keyValues).pluck('value').every('constant');
  return objectFn;
};
```

Summary

This chapter has been quite a ride. We have written lots of code, some of it pretty dense. While doing it we've taken our expression engine from something extremely simple to something that can be used to access and manipulate arbitrarily deep data structures on scopes. That is no small feat!

In this chapter you've learned:

- How Angular expressions do attribute lookup with dot and square bracket syntax.
- How code generation is used to achieve arbitrarily nested attribute lookups without having to loop at evaluation time.
- How getters are cached within the lexer

- How scope attributes can be overriden by locals
- How Angular expressions do function calls
- How Angular expressions obtain and preserve the `this` context in method calls
- Why and how access to the Function constructor is prevented for security reasons.
- How access to potentially dangerous objects is prevented for security reasons.
- How Angular expressions do attribute assignment for simple attributes and nested attributes.
- How array elements and object values may contain other expressions
- How the forgiving nature of Angular expressions is accomplished: When attributes are accessed, existence checks are done on every step. When attributes are assigned, intermediate objects are created automatically. Exceptions are never thrown because of missing attributes.

In the next chapter we'll extend the expression language with more *operators*. Once we're done with that, our language will have the vocabulary for doing arithmetic, comparisons, and logical expressions.

Chapter 7

Operator Expressions

The majority of expressions used in a typical Angular application are covered by the features we've implemented in the last two chapters. A simple field lookup or function call is often all you need. However, sometimes you do need to have some actual logic in your expressions, to make decisions or derive values from other values.

Angular expressions are mostly logic-free, in that you cannot use many things you could in a full-blown programming language. For example, `if` statements and looping constructs are not supported. Nevertheless, there are a few things you can do:

- Arithmetic: `+`, `-`, `*`, `/`, and `%`
- Numeric comparisons: `<`, `>`, `<=`, and `>=`
- Boolean algebra: `&&` and `||`
- Equality checks: `==`, `!=`, `===`, `!==`
- Conditionals using the ternary operator `a ? b : c`
- Filtering using the pipe operator `|`

This chapter covers all of the above except for filters, which is the subject of the next chapter.

When discussing operators, the question of *operator precedence* is central: Given an expression that combines multiple operators, which operators get applied first? Angular's operator precedence rules are, for all intents and purposes, the same as JavaScript's. We will see how the precedence order is enforced as we implement the operators. We will also see how the natural precedence order can be altered using parentheses.

Finally, we'll look at how you can actually execute several *statements* in a single Angular expression, by separating them with the semicolon character `;`.

Unary Operators

We'll begin from the operators with the highest precedence and work our way down in decreasing precedence order. On the very top are primary expressions, which we have

already implemented: Whenever there's a function call, a field access, or a square bracket property access, it gets evaluated before anything else. The first thing after primary expressions are *unary operator expressions*.

Unary operators are operators that have exactly one operand:

- The unary `-` numerically negates its operand, as in `-42` or `-a`.
- The unary `+` actually does nothing, but it can be used for clarity, as in `+42` or `+a`.
- The not operator `!` negates its operand's boolean value, as in `!true` or `!a`.

Let's start with the unary `+` operator since it is so blindingly simple. Basically, it just returns its operand as-is:

test/parse_spec.js

```
it('parses a unary +', function() {
  expect(parse('+42')()).toBe(42);
  expect(parse('+a')({a: 42})).toBe(42);
});
```

In `Parser`, we'll introduce a new method `unary` that deals with unary operators and falls back to `primary` for everything else:

src/parse.js

```
Parser.prototype.unary = function() {
  if (this.expect('+')) {

  } else {
    return this.primary();
  }
};
```

For the unary `+` in particular, what `unary` actually does is nothing but delegate to `primary`. There's nothing more that needs to be done for the operation:

src/parse.js

```
Parser.prototype.unary = function() {
  if (this.expect('+')) {
    return this.primary();
  } else {
    return this.primary();
  }
};
```


To have unary expressions actually parsed, we need to call `unary` from somewhere. Let's do so in `assignment`, where we have previously called `primary`. The left and right sides of an assignment expression aren't necessarily primary expressions, but might be unary expressions too:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.unary();
  if (this.expect('=')) {
    if (!left.assign) {
      throw 'Implies assignment but cannot be assigned to';
    }
    var right = this.unary();
    return function(scope, locals) {
      return left.assign(scope, right(scope, locals), locals);
    };
  }
  return left;
};
```

Because `unary` falls back to `primary`, `assignment` now supports either of them on both the left and right hand sides.

What remains now is the `Lexer` side of the equation. Our parser is prepared to handle a unary `+`, but the lexer is not emitting one yet.

In the previous chapter we handled several situations by just emitting plain text tokens from the Lexer, as we did with `[`, `]`, and `.`, for example. For operators we'll be doing something different. Recall that in Chapter 5, for `true`, `false`, and `null`, we introduced the `OPERATORS` object that maps strings to functions:

src/parse.js

```
var OPERATORS = {
  'null': _.constant(null),
  'true': _.constant(true),
  'false': _.constant(false)
};
```

This is where our operator implementations will go. For the unary `+` operator, we'll add an "implementation" that actually does nothing:

src/parse.js

```
var OPERATORS = {
  'null': _.constant(null),
  'true': _.constant(true),
  'false': _.constant(false),
  '+': function() { }
};
```


The function does nothing because it doesn't need to. The parser does not actually invoke it. It just needs to be there so the parser can see the `+` token.

> Because the unary `+` is not invoked, it could just as well have been implemented as a plain text token. The parser wouldn't have noticed any difference. However, as we get into using `+` in a non-unary context, this will change.

The test still does not pass. That is because the Lexer is still not emitting the `+`.

Back when we implemented `true`, `false`, and `null`, we did so in `readIdent`. That was possible, because `true`, `false`, and `null` are alphanumeric tokens that are also valid identifiers. The `+` character is not, so it won't be handled by `readIdent`. For these kinds of operators we need a final `else` branch in the `lex` method, that attempts to look up an operator for the current character from the `OPERATORS` object:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.is('.') && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.is('\'"')) {
      this.readString(this.ch);
    } else if (this.is('[],{}:.()=')) {
      this.tokens.push({
        text: this.ch,
        json: true
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent(this.ch);
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      var fn = OPERATORS[this.ch];
      if (fn) {
        this.tokens.push({
          text: this.ch,
          fn: fn
        });
        this.index++;
      } else {
```



```
        throw 'Unexpected next character: '+this.ch;
      }
    }
  }

  return this.tokens;
};
```

Basically, `lex` now tries to match the character against all the things it knows about, and if all else fails, sees if the `OPERATORS` object has something stored for it.

That takes care of the unary `+`, and lets us jump right into the next unary operator, which is a lot more interesting since it actually does something:

test/parse_spec.js

```
it('parses a unary !', function() {
  expect(parse('!true')()).toBe(false);
  expect(parse('!42')()).toBe(false);
  expect(parse('!a')({a: false})).toBe(true);
  expect(parse('!!a')({a: false})).toBe(false);
});
```

The not operator has the same semantics as JavaScript's. The final expectation in this test shows how it can also be applied multiple times in succession.

Let's also add this operator into the `OPERATORS` object. Its operator function takes three arguments: The `scope` (or "self") and `locals` objects needed by all expressions, and the operand expression. The function resolves the operand and then applies the operator to it:

src/parse.js

```
var OPERATORS = {
  'null': _.constant(null),
  'true': _.constant(true),
  'false': _.constant(false),
  '+': function() { },
  '!': function(self, locals, a) {
    return !a(self, locals);
  }
};
```

As is seen here, the final effect of using an operator like `!` in an Angular expression is that the same JavaScript operator gets applied. This is generally true for operators, and it is in the functions stored in `OPERATORS` where this bridging between Angular and JavaScript happens.

In the parser, the `unary` method can now call this operator function. It'll be in the operator token's `fn` attribute, so first of all we need to grab hold of the token:

src/parse.js


```
Parser.prototype.unary = function() {
  var operator;
  if (this.expect('+')) {
    return this.primary();
  } else if ((operator = this.expect('!'))) {

  } else {
    return this.primary();
  }
};
```

Like all other expressions, the return value of a unary expression has to be a function that takes the scope and the locals objects. Here we return one that invokes the not token's operator function. The operand is parsed as *another* unary (or primary) expression, making the chaining of nots possible:

src/parse.js

```
Parser.prototype.unary = function() {
  var parser = this;
  var operator;
  if (this.expect('+')) {
    return this.primary();
  } else if ((operator = this.expect('!'))) {
    return function(self, locals) {
      return operator.fn(self, locals, parser.unary());
    };
  } else {
    return this.primary();
  }
};
```

As we implement operators, we also need to pay attention to retaining the `literal` and `constant` flags of the final expressions returned by the parser. More precisely, we need to pay attention to the *`constant`* flag. Expressions with operators are by definition not literals, so we can ignore the `literal` flag.

When a constant value is negated, it remains constant. When a non-constant value is negated, it remains non-constant:

src/parse.js

```
it('parses negated value as constant if value is constant', function() {
  expect(parse('!true').constant).toBe(true);
  expect(parse('!!true').constant).toBe(true);
  expect(parse('!a').constant).toBeFalsy();
});
```


> We assert that the `constant` is "falsy" instead of comparing it to a concrete `false` value. That's because the flag is actually absent, or `undefined`, for non-constant values.

As we parse a unary operator we need to read the `constant` flag of the operand and assign it to the final expression. You could say the "constantness" of the expression is transitive:

src/parse.js

```
Parser.prototype.unary = function() {
  var parser = this;
  var operator;
  if (this.expect('+')) {
    return this.primary();
  } else if ((operator = this.expect('!'))) {
    var operand = parser.unary();
    var unaryFn = function(self, locals) {
      return operator.fn(self, locals, operand);
    };
    unaryFn.constant = operand.constant;
    return unaryFn;
  } else {
    return this.primary();
  }
};
```

The third and final unary operator we'll support is `-` for numeric negation:

test/parse_spec.js

```
it('parses a unary -', function() {
  expect(parse('-42')()).toBe(-42);
  expect(parse('-a')({a: -42})).toBe(42);
  expect(parse('--a')({a: -42})).toBe(-42);
});
```

Let's also make sure expressions with numeric negation are constant if their operands are:

test/parse_spec.js

```
it('parses numerically negated value as constant if needed', function() {
  expect(parse('-42').constant).toBe(true);
  expect(parse('-a').constant).toBeFalsy();
});
```


What's interesting about the unary - is that in the operator function it is not unary at all. It is actually our first *binary* operator. It is the subtraction of two numbers:

src/parse.js

```
var OPERATORS = {
  'null': _.constant(null),
  'true': _.constant(true),
  'false': _.constant(false),
  '+': function() { },
  '!': function(self, locals, a) {
    return !a(self, locals);
  },
  '-': function(self, locals, a, b) {
    a = a(self, locals);
    b = b(self, locals);
    return a - b;
  }
};
```

When used in a unary context, the parser then invokes the operator function of - with the constant 0 and the operand. This way we don't need two different implementations of the operator function:

src/parse.js

```
Parser.prototype.unary = function() {
  var parser = this;
  var operator;
  var operand;
  if (this.expect('+')) {
    return this.primary();
  } else if ((operator = this.expect('!'))) {
    operand = parser.unary();
    var unaryFn = function(self, locals) {
      return operator.fn(self, locals, operand);
    };
    unaryFn.constant = operand.constant;
    return unaryFn;
  } else if ((operator = this.expect('-'))) {
    operand = parser.unary();
    var binaryFn =  function(self, locals) {
      return operator.fn(self, locals, _.constant(0), operand);
    };
    binaryFn.constant = operand.constant;
    return binaryFn;
  } else {
    return this.primary();
  }
};
```


As we've seen, Angular expressions are much more lenient about operating on missing attributes than JavaScript is. When you use the unary - with an operand that does not actually exist, JavaScript would raise a reference error, but Angular will just give you zero:

test/parse_spec.js

```
it('fills missing value in unary - with zero', function() {
  expect(parse('-a')()).toBe(0);
});
```

This is due to the fact that the operator function for - substitutes any missing operands with zeros:

src/parse.js

```
var OPERATORS = {
  'null': _.constant(null),
  'true': _.constant(true),
  'false': _.constant(false),
  '+': function() { },
  '!': function(self, locals, a) {
    return !a(self, locals);
  },
  '-': function(self, locals, a, b) {
    a = a(self, locals);
    b = b(self, locals);
    return (_.isUndefined(a) ? 0 : a) - (_.isUndefined(b) ? 0 : b);
  }
};
```

Now we've covered all unary operators and taken a peek at binary operators too. Let's go right ahead and implement our first proper binary operators.

Multiplicative Operators

After unary operators, the operators with the highest precedence are numeric multiplicative operators: Multiplication, division, and remainder. Unsurprisingly, they all work just like they do in JavaScript:

test/parse_spec.js


```
it('parses a multiplication', function() {
  expect(parse('21 * 2')()).toBe(42);
});

it('parses a division', function() {
  expect(parse('84 / 2')()).toBe(42);
});

it('parses a remainder', function() {
  expect(parse('85 % 43')()).toBe(42);
});
```

In the `OPERATORS` object we'll store a binary operator function for each of these operators. They just resolve the operands and apply the corresponding JavaScript operator to them:

src/parse.js

```
var OPERATORS = {
  'null': _.constant(null),
  'true': _.constant(true),
  'false': _.constant(false),
  '+': function() { },
  '!': function(self, locals, a) {
    return !a(self, locals);
  },
  '-': function(self, locals, a, b) {
    a = a(self, locals);
    b = b(self, locals);
    return (_.isUndefined(a) ? 0 : a) - (_.isUndefined(b) ? 0 : b);
  },
  '*': function(self, locals, a, b) {
    return a(self, locals) * b(self, locals);
  },
  '/': function(self, locals, a, b) {
    return a(self, locals) / b(self, locals);
  },
  '%': function(self, locals, a, b) {
    return a(self, locals) % b(self, locals);
  }
};
```

The operators are executed by a new `Parser` method called `multiplicative`. Its shape is very similar to that of `unary` and `assignment`: It consumes the two operands as (unary) expressions, and it consumes the operator expression in between. It then returns a function that applies the operator to the operands:

src/parse.js

```
Parser.prototype.multiplicative = function() {
  var left = this.unary();
  var operator;
  if ((operator = this.expect('*', '/', '%'))) {
    var right = this.unary();
    var binaryFn = function(self, locals) {
      return operator.fn(self, locals, left, right);
    };
    binaryFn.constant = left.constant && right.constant;
    return binaryFn;
  } else {
    return left;
  }
};
```

The `multiplicative` function does not know which one of the three multiplicative operators it's looking at. It does not need to since the operator function in `operator.fn` is doing the actual work, and the operator functions all have the same signature.

Notice that the result of a multplicative operation is constant if both of its operands are constant.

Now, in `assignment` we'll replace the call to `unary` with a call to `multiplicative` so that multiplicative operators get applied:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.multiplicative();
  if (this.expect('=')) {
    if (!left.assign) {
      throw 'Implies assignment but cannot be assigned to';
    }
    var right = this.multiplicative();
    return function(scope, locals) {
      return left.assign(scope, right(scope, locals), locals);
    };
  }
  return left;
};
```

At the beginning of the chapter we discussed the importance of precedence rules. Now we're starting to see how they are actually defined. Instead of having a special "precedence order table" somewhere, the precedence order is implicit in the order in which the different parsing functions call each other.

Right now, our top-level parsing function is `assignment`. In turn, `assignment` invokes `multiplicative`, which invokes `unary`, which finally invokes `primary`. The very first thing each method does is to parse their "left hand side" operand using the next function in the chain. That means the *final* method in the chain is the one with the highest precedence. Our current precedence order then is:

1. Primary
2. Unary
3. Multiplicative
4. Assignment

As we continue adding more operators, it will be the function(s) we call them *from* that defines their precedence order.

Before moving forward, there's some refactoring we can do to reduce duplication. Both numeric negation and multiplicatives currently invoke a binary operator function, and do so very similarly. We can extract the invocation of a generic binary operator to a function that takes the two operands and the operator:

src/parse.js

```
Parser.prototype.binaryFn = function(left, op, right) {
  var fn = function(self, locals) {
    return op(self, locals, left, right);
  };
  fn.constant = left.constant && right.constant;
  return fn;
};
```

The `multiplicative` function now simplifies to:

src/parse.js

```
Parser.prototype.multiplicative = function() {
  var left = this.unary();
  var operator;
  if ((operator = this.expect('*', '/', '%'))) {
    return this.binaryFn(left, operator.fn, this.unary());
  } else {
    return left;
  }
};
```

The numeric negation branch of `unary` also simplifies to:

src/parse.js

```
Parser.prototype.unary = function() {
  var parser = this;
  var operator;
  if (this.expect('+')) {
    return this.primary();
  } else if ((operator = this.expect('!'))) {
    var operand = parser.unary();
    var unaryFn = function(self, locals) {
```

```
      return operator.fn(self, locals, operand);
    };
    unaryFn.constant = operand.constant;
    return unaryFn;
  } else if ((operator = this.expect('-'))) {
    return this.binaryFn(_.constant(0), operator.fn, parser.unary());
  } else {
    return this.primary();
  }
};
```

We've now broken the test that asserts the constantness of negated numbers. The culprit is the `_.constant(0)` function used in `unary`. The generic `binaryFn` function now checks the `constant` flag of both of its operands, and the function constructed by `_.constant(0)` has no such flag. We can remedy the situation by introducing that flag. Let's go right ahead and make the zero expression a field in `Parser` which we can reuse whenever it's needed:

src/parse.js

```
Parser.ZERO = _.extend(_.constant(0), {constant: true});
```

Then we'll use `Parser.ZERO` in `unary` instead of creating the constant zero function on the fly:

src/parse.js

```
Parser.prototype.unary = function() {
  var parser = this;
  var operator;
  if (this.expect('+')) {
    return this.primary();
  } else if ((operator = this.expect('!'))) {
    var operand = parser.unary();
    var unaryFn = function(self, locals) {
      return operator.fn(self, locals, operand);
    };
    unaryFn.constant = operand.constant;
    return unaryFn;
  } else if ((operator = this.expect('-'))) {
    return this.binaryFn(Parser.ZERO, operator.fn, parser.unary());
  } else {
    return this.primary();
  }
};
```

We're now successfully parsing multiplicative operations, but what happens if you have several of them back to back?

test/parse_spec.js


```
it('parses several multiplicatives', function() {
  expect(parse('36 * 2 % 5')()).toBe(2);
});
```

This doesn't work yet because `multiplicative` just parses at most one operation and then returns. If there's more to the expression than that, the rest just gets ignored.

The way to fix this is to keep consuming tokens in `multiplicative` as longs as there are more multiplicative operators to parse. The result of each step becomes the left hand side of the next step:

src/parse.js

```
Parser.prototype.multiplicative = function() {
  var left = this.unary();
  var operator;
  while ((operator = this.expect('*', '/', '%'))) {
    left = this.binaryFn(left, operator.fn, this.unary());
  }
  return left;
};
```

Since all the three multiplicative operators have the same precedence, they're applied from left to right, which is what our function now does.

Additive Operators

Right on the heels of multiplicative operators come additive operators: Addition and subtraction. We've already used both in a unary context, and now's the time to look at them as binary functions:

test/parse_spec.js

```
it('parses an addition', function() {
  expect(parse('20 + 22')()).toBe(42);
});

it('parses a subtraction', function() {
  expect(parse('42 - 22')()).toBe(20);
});
```

As discussed, additives come *after* multiplicatives in precedence:

test/parse_spec.js


```
it('parses multiplicatives on a higher precedence than additives', function() {
  expect(parse('2 + 3 * 5')()).toBe(17);
  expect(parse('2 + 3 * 2 + 3')()).toBe(11);
});
```

In the `OPERATORS` object we already have subtraction covered since the version we added for the unary context works perfectly well in a binary context. For addition we need to replace the no-op function with one that actually does the job:

src/parse.js

```
var OPERATORS = {
  'null': _.constant(null),
  'true': _.constant(true),
  'false': _.constant(false),
  '+': function(self, locals, a, b) {
    return a(self, locals) + b(self, locals);
  },
  '!': function(self, locals, a) {
    return !a(self, locals);
  },
  '-': function(self, locals, a, b) {
    a = a(self, locals);
    b = b(self, locals);
    return (_.isUndefined(a) ? 0 : a) - (_.isUndefined(b) ? 0 : b);
  },
  '*': function(self, locals, a, b) {
    return a(self, locals) * b(self, locals);
  },
  '/': function(self, locals, a, b) {
    return a(self, locals) / b(self, locals);
  },
  '%': function(self, locals, a, b) {
    return a(self, locals) % b(self, locals);
  }
};
```

On the parser side, we'll make a new function called `additive`, which looks just like `multiplicative` except for the operator characters it expects and the next operator functions it calls:

src/parse.js

```
Parser.prototype.additive = function() {
  var left = this.multiplicative();
  var operator;
  while ((operator = this.expect('+', '-'))) {
    left = this.binaryFn(left, operator.fn, this.multiplicative());
  }
  return left;
};
```


Additive operations are inserted between assignments and multiplicative operations in the precence order, which means `assignment` should now call `additive`, as `additive` calls `multiplicative`:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.additive();
  if (this.expect('=')) {
    if (!left.assign) {
      throw 'Implies assignment but cannot be assigned to';
    }
    var right = this.additive();
    return function(scope, locals) {
      return left.assign(scope, right(scope, locals), locals);
    };
  }
  return left;
};
```

As we saw with the unary `-`, a missing operand is treated as zero. This is also the case for subtraction. Either or both missing operands are replaced with zeros:

test/parse_spec.js

```
it('treats a missing subtraction operand as zero', function() {
  expect(parse('a - b')({a: 20})).toBe(20);
  expect(parse('a - b')({b: 20})).toBe(-20);
  expect(parse('a - b')({})).toBe(0);
});
```

Our implementation of subtraction in `OPERATORS` already does what we want here.

Addition should work similarly. If either of the operands is missing, a zero will be put in its place:

test/parse_spec.js

```
it('treats a missing addition operand as zero', function() {
  expect(parse('a + b')({a: 20})).toBe(20);
  expect(parse('a + b')({b: 20})).toBe(20);
});
```

However, when *both* operands are missing from an addition operation, it actually amounts to `undefined` instead of zero. One could argue this is an inconsistency in the Angular expression engine, but nevertheless this is how it works:

test/parse_spec.js


```
it('returns undefined from addition when both operands missing', function() {
  expect(parse('a + b')()).toBeUndefined();
});
```

For dealing with the missing operands, we need to have different branches in the implementation of +:

src/parse.js

```
var OPERATORS = {
  // ...
  '+': function(self, locals, a, b) {
    a = a(self, locals);
    b = b(self, locals);
    if (!_.isUndefined(a)) {
      if (!_.isUndefined(b)) {
        return a + b;
      } else {
        return a;
      }
    }
    return b;
  },
  // ...
};
```

Relational And Equality Operators

After arithmetic in the precedence order there are the different ways to compare things. For numbers, there are the four relational operators:

test/parse_spec.js

```
it('parses relational operators', function() {
  expect(parse('1 < 2')()).toBe(true);
  expect(parse('1 > 2')()).toBe(false);
  expect(parse('1 <= 2')()).toBe(true);
  expect(parse('2 <= 2')()).toBe(true);
  expect(parse('1 >= 2')()).toBe(false);
  expect(parse('2 >= 2')()).toBe(true);
});
```

For numbers as well as other kinds of values, there are the equality checks and their negations. Angular expressions support both the loose and strict equality operators of JavaScript:

test/parse_spec.js


```
it('parses equality operators', function() {
  expect(parse('42 == 42')()).toBe(true);
  expect(parse('42 == "42"')()).toBe(true);
  expect(parse('42 != 42')()).toBe(false);
  expect(parse('42 === 42')()).toBe(true);
  expect(parse('42 === "42"')()).toBe(false);
  expect(parse('42 !== 42')()).toBe(false);
});
```

Of these two families of operators, relationals take precedence:

test/parse_spec.js

```
it('parses relationals on a higher precedence than equality', function() {
  expect(parse('2 == "2" > 2 === "2"')()).toBe(false);
});
```

The test here checks that the order of operations is:

1. `2 == "2" > 2 === "2"`
2. `2 == false === 2`
3. `false === 2`
4. `false`

Instead of:

1. `2 == "2" > 2 === "2"`
2. `true > false`
3. `1 > 0`
4. `true`

Both relational and equality operators have lower precedence than additive operations:

test/parse_spec.js

```
it('parses additives on a higher precedence than relationals', function() {
  expect(parse('2 + 3 < 6 - 2')()).toBe(false);
});
```

This test checks that the order of application is:

1. `2 + 3 < 6 - 2`
2. `5 < 4`
3. `false`

And not:

1. `2 + 3 < 6 - 2`
2. `2 + true - 2`
3. `2 + 1 - 2`
4. `1`

All of these eight new operators are added to the `OPERATORS` object in a straightforward manner: They all take two operands, resolve them, and then apply the corresponding JavaScript operator:

src/parse.js

```
var OPERATORS = {
  // ...
  '<': function(self, locals, a, b) {
    return a(self, locals) < b(self, locals);
  },
  '>': function(self, locals, a, b) {
    return a(self, locals) > b(self, locals);
  },
  '<=': function(self, locals, a, b) {
    return a(self, locals) <= b(self, locals);
  },
  '>=': function(self, locals, a, b) {
    return a(self, locals) >= b(self, locals);
  },
  '==': function(self, locals, a, b) {
    return a(self, locals) == b(self, locals);
  },
  '!=': function(self, locals, a, b) {
    return a(self, locals) != b(self, locals);
  },
  '===': function(self, locals, a, b) {
    return a(self, locals) === b(self, locals);
  },
  '!==': function(self, locals, a, b) {
    return a(self, locals) !== b(self, locals);
  },
};
```

In the parser, two new functions are introduced - one for the equality operators and another one for the relational operators. We can't use one for both since that would break our precedence rules. Both of these functions take a familiar form:

src/parse.js


```
Parser.prototype.equality = function() {
  var left = this.relational();
  var operator;
  while ((operator = this.expect('==', '!=', '===', '!==', ))) {
    left = this.binaryFn(left, operator.fn, this.relational());
  }
  return left;
};

Parser.prototype.relational = function() {
  var left = this.additive();
  var operator;
  while ((operator = this.expect('<', '>', '<=', '>='))) {
    left = this.binaryFn(left, operator.fn, this.additive());
  }
  return left;
};
```

Equality is now our lowest precedence operator after assignment, so that is what assignment should delegate to:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.equality();
  if (this.expect('=')) {
    if (!left.assign) {
      throw 'Implies assignment but cannot be assigned to';
    }
    var right = this.equality();
    return function(scope, locals) {
      return left.assign(scope, right(scope, locals), locals);
    };
  }
  return left;
};
```

Finally, we'll need to make some changes to `Lexer.lex` to support these functions. Earlier in the chapter we introduced the conditional branch that looks operators up from the `OPERATORS` object. However, all the operators we had back then consisted of just a single character. Now we have operators that have two characters, such as `==`, or even three characters, such as `===`. These also need to be supported by that conditional branch. It should first see if the next three characters match an operator, then the next two characters, and finally just the next single character:

src/parse.js

```
Lexer.prototype.lex = function(text) {
  this.text = text;
  this.index = 0;
  this.ch = undefined;
  this.tokens = [];

  while (this.index < this.text.length) {
    this.ch = this.text.charAt(this.index);
    if (this.isNumber(this.ch) ||
          (this.is('.') && this.isNumber(this.peek()))) {
      this.readNumber();
    } else if (this.is('\'"')) {
      this.readString(this.ch);
    } else if (this.is('[],{}:.()=')) {
      this.tokens.push({
        text: this.ch,
        json: true
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent(this.ch);
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      var ch2 = this.ch + this.peek();
      var ch3 = this.ch + this.peek() + this.peek(2);
      var fn = OPERATORS[this.ch];
      var fn2 = OPERATORS[ch2];
      var fn3 = OPERATORS[ch3];
      if (fn3) {
        this.tokens.push({
          text: ch3,
          fn: fn3
        });
        this.index += 3;
      } else if (fn2) {
        this.tokens.push({
          text: ch2,
          fn: fn2
        });
        this.index += 2;
      } else if (fn) {
        this.tokens.push({
          text: this.ch,
          fn: fn
        });
        this.index++;
      } else {
        throw 'Unexpected next character: '+this.ch;
      }
    }
```



```
  }

  return this.tokens;
};
```

This code uses a modified version of `Lexer.peek` that can peek at not just the next character, but the nth character from the current index. It takes an optional argument for `n`, the default for which is `1`:

src/parse.js

```
Lexer.prototype.peek = function(n) {
  n = n || 1;
  return this.index + n < this.text.length ?
    this.text.charAt(this.index + n) :
    false;
};
```

The tests for equality operators still don't pass, even though we should have everything in place. The problem is that in the previous chapter we started consuming the equality character `=` as a text token, which we needed to implement assignments. What's happening now is that when the lexer sees the first `=` in `==`, it emits it right away without looking at the whole operator.

What we should do is firstly remove `=` from the collection of text tokens, by changing the line in `lex` from:

src/parse.js

```
    } else if (this.is('[],{}:.()=')) {
```

to

src/parse.js

```
    } else if (this.is('[],{}:.()')) {
```

Then, we should add the single equals sign to our collection of operators. The operator function itself does not need to do anything:

src/parse.js

```
var OPERATORS = {
  // ...
  '=': _.noop
};
```

The single equals sign is now emitted as an operator token instead of a text token. Since assignment doesn't use the operator function - it's just interested in the `text` attribute - this change does not cause it to behave any differently. But it lets the `OPERATORS` lookup code consider operators with three or two characters first, which is what we needed.

Logical Operators AND and OR

The two remaining binary operators we are going to implement are the logical operators `&&` and `||`. Their functionality in expressions is just what you would expect:

test/parse_spec.js

```
it('parses logical AND', function() {
  expect(parse('true && true')()).toBe(true);
  expect(parse('true && false')()).toBe(false);
});

it('parses logical OR', function() {
  expect(parse('true || true')()).toBe(true);
  expect(parse('true || false')()).toBe(true);
  expect(parse('fales || false')()).toBe(false);
});
```

Just like other binary operators, you can chain several logical operators back to back:

test/parse_spec.js

```
it('parses multiple ANDs', function() {
  expect(parse('true && true && true')()).toBe(true);
  expect(parse('true && true && false')()).toBe(false);
});

it('parses multiple ORs', function() {
  expect(parse('true || true || true')()).toBe(true);
  expect(parse('true || true || false')()).toBe(true);
  expect(parse('false || false || true')()).toBe(true);
  expect(parse('false || false || false')()).toBe(false);
});
```

An interesting detail about logical operators is that they are short-circuited. When the left hand side of an AND expression is falsy, the right hand side expression does not get evaluated at all, just like it would not in JavaScript:

test/parse_spec.js

```
it('short-circuits AND', function() {
  var invoked;
  var scope = {fn: function() { invoked = true; }};

  parse('false && fn()')(scope);

  expect(invoked).toBeUndefined();
});
```


Correspondingly, if the left hand side of an OR expression is truthy, the right hand side is not evaluated:

test/parse_spec.js

```
it('short-circuits OR', function() {
  var invoked;
  var scope = {fn: function() { invoked = true; }};

  parse('true || fn()')(scope);

  expect(invoked).toBeUndefined();
});
```

In precedence order, AND comes before OR:

test/parse_spec.js

```
it('parses AND with a higher precedence than OR', function() {
  expect(parse('false && true || true')()).toBe(true);
});
```

Here we test that the expression is evaluated as `(false && true) || true` rather than `false && (true || true)`.

Equality comes before both OR and AND in precedence:

test/parse_spec.js

```
it('parses OR with a lower precedence than equality', function() {
  expect(parse('1 === 2 || 2 === 2')()).toBeTruthy();
});
```

The way these operators are implemented follows a pattern that's familiar by now. In the `OPERATORS` object we have the operator functions that implement the operators in terms of JavaScript:

src/parse.js

```
var OPERATORS = {
  // ...
  '&&': function(self, locals, a, b) {
    return a(self, locals) && b(self, locals);
  },
  '||': function(self, locals, a, b) {
    return a(self, locals) || b(self, locals);
  }
};
```


In the parser, we have two new functions that implement the operators as binary functions - one for OR and one for AND:

src/parse.js

```
Parser.prototype.logicalOR = function() {
  var left = this.logicalAND();
  var operator;
  while ((operator = this.expect('||'))) {
    left = this.binaryFn(left, operator.fn, this.logicalOR());
  }
  return left;
};

Parser.prototype.logicalAND = function() {
  var left = this.equality();
  var operator;
  while ((operator = this.expect('&&'))) {
    left = this.binaryFn(left, operator.fn, this.equality());
  }
  return left;
};
```

Once again, since we're going from the operators with higher precedence downward, these operators are inserted to the parsing chain right after `assignment`:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.logicalOR();
  if (this.expect('=')) {
    if (!left.assign) {
      throw 'Implies assignment but cannot be assigned to';
    }
    var right = this.logicalOR();
    return function(scope, locals) {
      return left.assign(scope, right(scope, locals), locals);
    };
  }
  return left;
};
```


The Ternary Operator

The final operator we'll implement in this chapter (and the penultimate operator overall) is the C-style ternary operator, with which you can return one of two alternative values based on a test expression:

test/parse_spec.js

```
it('parses the ternary expression', function() {
  expect(parse('a === 42 ? true : false')({a: 42})).toBe(true);
  expect(parse('a === 42 ? true : false')({a: 43})).toBe(false);
});
```

The ternary operator is just below OR in the precedence chain, so ORs will get evaluated first:

test/parse_spec.js

```
it('parses OR with a higher precedence than ternary', function() {
  expect(parse('0 || 1 ? 0 || 2 : 0 || 3')()).toBe(2);
});
```

You can also nest ternary operators, though you could argue doing that doesn't result in very clear code:

test/parse_spec.js

```
it('parses nested ternaries', function() {
  expect(
    parse('a === 42 ? b === 42 ? "a and b" : "a" : c === 42 ? "c" : "none"')({
      a: 44,
      b: 43,
      c: 42
  })).toEqual('c');
});
```

Ternary expressions, like other expressions, are also constant if all their operands are constant:

test/parse_spec.js

```
it('makes ternaries constants if their operands are', function() {
  expect(parse('true ? 42 : 43').constant).toBeTruthy();
  expect(parse('true ? 42 : a').constant).toBeFalsy();
});
```


Unlike most of the operators we've seen in this chapter, the ternary operator is *not* implemented as an operator function in the `OPERATORS` object. Since there are two different parts to the operator, the `?` and the `:`, parsing it is a job for the parser, not the lexer.

What the lexer does need to do is emit the `?` character, which we haven't been doing so far. Change the line in `Lexer.lex` that considers text tokens to:

src/parse.js

```
    } else if (this.is('[],{}:.()?')) {
```

In `Parser` we'll introduce a new function `ternary` that implements this operator. First of all, it consumes the three operands as well as the two parts of the operator:

src/parse.js

```
Parser.prototype.ternary = function() {
  var left = this.logicalOR();
  if (this.expect('?')) {
    var middle = this.ternary();
    this.consume(':');
    var right = this.ternary();

  } else {
    return left;
  }
};
```

Given the three operand expressions, it returns an expression function that applies the JavaScript ternary operator. The expression's `constant` flag is set based on the `constant` flags of the operands:

src/parse.js

```
Parser.prototype.ternary = function() {
  var left = this.logicalOR();
  if (this.expect('?')) {
    var middle = this.ternary();
    this.consume(':');
    var right = this.ternary();
    var ternaryFn = function(self, locals) {
      return left(self, locals) ? middle(self, locals) : right(self, locals);
    };
    ternaryFn.constant = left.constant && middle.constant && right.constant;
    return ternaryFn;
  } else {
    return left;
  }
};
```


Again, we change the next operator looked at from `assignment`, now to `ternary`:

src/parse.js

```
Parser.prototype.assignment = function() {
  var left = this.ternary();
  if (this.expect('=')) {
    if (!left.assign) {
      throw 'Implies assignment but cannot be assigned to';
    }
    var right = this.ternary();
    return function(scope, locals) {
      return left.assign(scope, right(scope, locals), locals);
    };
  }
  return left;
};
```

The final precedence order of the operators can be read by looking at the order in which parsing functions are called *in reverse*:

1. Primary expressions: Lookups, function calls, method calls.
2. Unary expressions: `+a`, `-a`, `!a`.
3. Multiplicative arithmetic expressions: `a * b`, `a / b`, and `a % b`.
4. Additive arithmetic expressions: `a + b` and `a - b`.
5. Relational expressions: `a < b`, `a > b`, `a <= b`, and `a >= b`.
6. Equality testing expressions: `a == b`, `a != b`, `a === b`, and `a !== b`.
7. Logical AND expressions: `a && b`.
8. Logical OR expressions: `a || b`.
9. Ternary expressions: `a ? b : c`.
10. Assignments: `a = b`.

Altering The Precedence Order with Parentheses

Of course, the natural precedence order is not always what you want, and just like JavaScript and many other languages, Angular expressions give you the means to alter the precedence order by grouping operations using parentheses:

test/parse_spec.js

```
it('parses parentheses altering precedence order', function() {
  expect(parse('21 * (3 - 1)')()).toBe(42);
  expect(parse('false && (true || true)')()).toBe(false);
  expect(parse('-((a % 2) === 0 ? 1 : 2)')({a: 42})).toBe(-1);
});
```


The way this is implemented is actually remarkably simple. Since parentheses cut through the whole precedence table, they should be the very first thing we test for when parsing an expression or a subexpression. In concrete terms, that means they should be the first thing we test for in the `primary` function.

If an opening parenthesis is seen at the beginning of a primary expression, a whole new precedence chain is started for the expression that goes inside the parentheses. This effectively forces anything that's in parentheses to be evaluated before anything else around it:

src/parse.js

```
Parser.prototype.primary = function() {
  var primary;
  if (this.expect('(')) {
    primary = this.assignment();
    this.consume(')');
  } else if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else {
    var token = this.expect();
    primary = token.fn;
    if (token.json) {
      primary.constant = true;
      primary.literal = true;
    }
  }

  var next;
  var context;
  while ((next = this.expect('[', '.', '('))) {
     if (next.text === '[') {
      context = primary;
      primary = this.objectIndex(primary);
    } else if (next.text === '.') {
      context = primary;
      primary = this.fieldAccess(primary);
    } else if (next.text === '(') {
      primary = this.functionCall(primary, context);
      context = undefined;
    }
  }
  return primary;
};
```


Statements

Before we conclude the chapter, we'll look at a way you can execute multiple things in a single Angular expression.

Everything we've seen so far has been all about one expression that, in the end, results in one return value. However, this is not a hard limitation. You can actually have multiple, independent expressions in one expression string, if you just separate them with semicolons:

src/parse.js

```
it('parses several statements', function() {
  var fn = parse('a = 1; b = 2; c = 3');
  var scope = {};
  fn(scope);
  expect(scope).toEqual({a: 1, b: 2, c: 3});
});
```

When you do this, it is the value of the *last* expression that becomes the return value of the combined expression. The return values of any preceding expressions are effectively thrown away:

src/parse.js

```
it('returns the value of the last statement', function() {
  expect(parse('a = 1; b = 2; a + b')({})).toBe(3);
});
```

This means that if you have multiple expressions, every expression but the last one is probably going to be something that produces a side effect, such as an attribute assignment or a function call. Anything else would have no visible effect except for consuming CPU cycles. In imperative programming languages constructs like these are often called statements, as opposed to expressions, which is where the name we use comes from.

To implement statements, we first need the semicolon character to be emitted from the lexer so we can identify it in the parser. Let's add it to the growing collection of text token characters in `Lexer.lex`:

src/parse.js

```
    } else if (this.is('[],{}:.()?;')) {
```

In the parser, we're going to introduce a new function, that will actually become the top-level parsing function. It will, first of all, collect an array of statements as long as it can find some. Each statement can be any expression:

src/parse.js

```
Parser.prototype.statements = function() {
  var statements = [];
  do {
    statements.push(this.assignment());
  } while (this.expect(';'));
};
```

When an expression has no semicolons, which is the most common case, the `statements` array will hold exactly one item when the loop is done. In that case, we can just return that item, giving us the functionality of single expressions:

src/parse.js

```
Parser.prototype.statements = function() {
  var statements = [];
  do {
    statements.push(this.assignment());
  } while (this.expect(';'));
  if (statements.length === 1) {
    return statements[0];
  }
};
```

If there are more items in the array than one, we'll return a function that invokes each one and returns the return value of the last one:

src/parse.js

```
Parser.prototype.statements = function() {
  var statements = [];
  do {
    statements.push(this.assignment());
  } while (this.expect(';'));
  if (statements.length === 1) {
    return statements[0];
  } else {
    return function(self, locals) {
      var value;
      _.forEach(statements, function(statement) {
        value = statement(self, locals);
      });
      return value;
    };
  }
};
```

This new function is what we should now call right from the first function in `Parser`:

src/parse.js


```
Parser.prototype.parse = function(text) {
  this.tokens = this.lexer.lex(text);
  return this.statements();
};
```

Summary

We've grown our expression language to something that can derive values from other values using operators. It lets application developers have some actual logic in their watch expressions and data binding expressions. Compared to full-blown JavaScript, the logic we allow is very restricted and simple, but it is enough for most use cases. You could say that anything more complicated should not be put in expressions anyway, since they're really designed for data binding and watches rather than your application logic.

In this chapter you've learned:

- What operators the Angular expression language supports and how they're implemented.
- The precedence order of operators and the way the precedence order is baked into the parsing functions
- How the precedence order may be altered with parentheses
- How operator expressions may be constant, but may never be literal.
- How arithmetic expressions are more forgiving than JavaScript arithmetic when it comes to missing operands
- That expressions may consist of multiple statements, of which all but the last one are executed purely for side effects.

In the next chapter we'll finalize our implementation of the Angular expression language by implementing filters - the only major language feature Angular expressions have that JavaScript does *not* have.

Chapter 8

Filters

Part III

Modules And Dependency Injection

Dependency injection is one of the defining features of Angular, as well as one of its major selling points. To an Angular application developer, it is the glue that holds everything together. All the other Angular features are really just (semi-)independent tools that all happen to ship within the same codebase, but it is dependency injection that brings everything together into a coherent framework you can build your applications on.

Because the DI framework is so central to every Angular application, it is important to know how it works. Unfortunately, it also seems to be one of the more difficult parts of Angular to grasp, judging by the questions you see online almost on a daily basis. This is probably partly due to documentation, and partly due to the terminology used in the implementation, but it is also due to the fact that the DI framework solves a nontrivial problem: Wiring together the different parts of an application.

Dependency injection is also one of the more controversial features of Angular. When people criticise Angular, DI is often the reason they give for why they don't like it. Some people criticise the Angular injector implementation, some people the idea of DI in JavaScript in general. In both cases, the critics may have valid points but there also often seems to be an element of confusion present about what the Angular injector actually does, and why. Hopefully this part of the book will help you not only in application development, but also in alleviating some of the confusion people exhibit when discussing these things.

Chapter 9

Modules And The Injector

This chapter lays the groundwork for the dependency injection features of Angular. We will introduce the two main concepts involved - modules and injectors - and see how they allow for registering application components and then injecting them to where they are needed.

Of these two concepts, *modules* is the one most application developers deal with directly. Modules are collections of application configuration information. They are where you register your services, controllers, directives, filters, and other application components.

But it is the *injector* that really brings an application to life. While modules are where you register your components, no components are actually created until you create an injector and give it some modules to instantiate.

In this chapter we'll see how to create modules and then how to create an injector that loads those modules.

The `angular` Global

If you have ever used Angular, you have probably interacted with the global `angular` object. This is the point where we introduce that object.

The reason we need the object now is that it is where information about registered Angular modules is stored. As we begin creating modules and injectors, we'll need that storage.

The framework component that deals with modules is called the *module loader* and implemented in a file called `loader.js`. This is where we'll introduce the `angular` global. But first, as always, let's add a test for it in a new test file:

test/loader_spec.js

```
/* jshint globalstrict: true */
/* global setupModuleLoader: false */
'use strict';
```

```
describe('setupModuleLoader', function() {

  it('exposes angular on the window', function() {
    setupModuleLoader(window);
    expect(window.angular).toBeDefined();
  });

});
```

This test assumes there is a global function called `setupModuleLoader` that you can call with a `window` object. When you've done that, there will be an `angular` attribute on that `window` object.

Let's then create `loader.js` and make this test pass:

src/loader.js

```
/* jshint globalstrict: true */
'use strict';

function setupModuleLoader(window) {
  var angular = window.angular = {};
}
```

That'll give us something to start from.

Initializing The Global Just Once

Since the `angular` global provides storage for registered modules, it is essentially a holder for global state. That means we need to take some measures to manage that state. First of all, we want a clean slate for each and every unit test, so we'll need to get rid of any existing `angular` globals at the beginning of every test:

test/loader_spec.js

```
beforeEach(function() {
  delete window.angular;
});
```

Also, in `setupModuleLoader` we need to be careful not to override an existing `angular` if there is one, even if someone or somehing was to call the function several times. When you call `setupModuleLoader` twice on the same `window`, after both calls the `angular` global should point to the same exact object:

test/loader_spec.js

```
it('creates angular just once', function() {
  setupModuleLoader(window);
  var ng = window.angular;
  setupModuleLoader(window);
  expect(window.angular).toBe(ng);
});
```

This can be fixed with a simple check for an existing `window.angular`:

src/loader.js

```
function setupModuleLoader(window) {
  var angular = (window.angular = window.angular || {});
}
```

We'll be reusing this "load once" pattern soon though, so let's abstract it out to a generic function called `ensure`, that takes an object, an attribute name, and a "factory function" that produces a value. The function uses the factory function to produce the attribute, but only if it does not already exist:

src/loader.js

```
function setupModuleLoader(window) {
  var ensure = function(obj, name, factory) {
    return obj[name] || (obj[name] = factory());
  };

  var angular = ensure(window, 'angular', Object);
}
```

In this case we'll assign an empty object to `window.angular` using the call `Object()`, which, to all intents and purposes, is the same as calling `new Object()`.

The `module` Method

The first method we'll introduce to `angular` is the one we'll be using in this and the following chapters a lot: `module`. Let's assert that this method in fact exists in a newly created `angular` global:

test/loader_spec.js

```
it('exposes the angular module function', function() {
  setupModuleLoader(window);
  expect(window.angular.module).toBeDefined();
});
```


Just like the global object itself, the `module` method should not be overridden when `setupModuleLoader` is called several times:

test/loader_spec.js

```
it('exposes the angular module function just once', function() {
  setupModuleLoader(window);
  var module = window.angular.module;
  setupModuleLoader(window);
  expect(window.angular.module).toBe(module);
});
```

We can now reuse our new `ensure` function to construct this method:

src/loader.js

```
function setupModuleLoader(window) {
  var ensure = function(obj, name, factory) {
    return obj[name] || (obj[name] = factory());
  };

  var angular = ensure(window, 'angular', Object);

  ensure(angular, 'module', function() {
    return function() {

    };
  });
}
```

Registering A Module

Having laid the groundwork, let's now get into what we actually want to do in this chapter, which is to register modules.

All the remaining tests in `loader_spec.js` in this chapter will work with a module loader, so let's create a nested `describe` block for them, and put the setup of the module loader into a before block. This way we won't have to repeat it for every test:

test/loader_spec.js

```
describe('modules', function() {

  beforeEach(function() {
    setupModuleLoader(window);
  });

});
```


The first behavior we'll test is that we can call the `angular.module` function and get back a module object.

The method signature of `angular.module` is that it takes a module name (a string) and an array of the module's dependencies, which may be empty. The method constructs a module object and returns it. One of the things the module object contains is its name, in the `name` attribute:

test/loader_spec.js

```
it('allows registering a module', function() {
  var myModule = window.angular.module('myModule', []);
  expect(myModule).toBeDefined();
  expect(myModule.name).toEqual('myModule');
});
```

When you register a module with the same name several times, the new module *replaces* any old ones. This also means that when we call `module` twice with the same name, it'll give us a different module object each time:

test/loader_spec.js

```
it('replaces a module when registered with same name again', function() {
  var myModule = window.angular.module('myModule', []);
  var myNewModule = window.angular.module('myModule', []);
  expect(myNewModule).not.toBe(myModule);
});
```

In our `module` method, let's delegate the work involved in creating a module to a new function called `createModule`. In that function, for now we can just create a module object and return it:

src/loader.js

```
function setupModuleLoader(window) {
  var ensure = function(obj, name, factory) {
    return obj[name] || (obj[name] = factory());
  };

  var angular = ensure(window, 'angular', Object);

  var createModule = function(name, requires) {
    var moduleInstance = {
      name: name
    };
    return moduleInstance;
  };
```



```
  ensure(angular, 'module', function() {
    return function(name, requires) {
      return createModule(name, requires);
    };
  });

}
```

Apart from the name, the new module should also have a reference to the array of required modules:

test/loader_spec.js

```
it('attaches the requires array to the registered module', function() {
  var myModule = window.angular.module('myModule', ['myOtherModule']);
  expect(myModule.requires).toEqual(['myOtherModule']);
});
```

We can just attach the given `requires` array to the module object to satisfy this requirement:

src/loader.js

```
var createModule = function(name, requires) {
  var moduleInstance = {
    name: name,
    requires: requires
  };
  return moduleInstance;
};
```

Getting A Registered Module

The other behavior provided by `angular.module` is getting hold of a module object that has been registered earlier. This you can do by omitting the second argument (the `requires` array). What that should give you is the same exact module object that was created when the module was registered:

test/loader_spec.js

```
it('allows getting a module', function() {
  var myModule = window.angular.module('myModule', []);
  var gotModule = window.angular.module('myModule');

  expect(gotModule).toBeDefined();
  expect(gotModule).toBe(myModule);
});
```


We'll introduce another private function for getting the module, called `getModule`. We'll also need some place to store the registered modules in. This we will do in a private object within the closure of the `ensure` call. We'll pass it in to both `createModule` and `getModule`:

src/loader.js

```
ensure(angular, 'module', function() {
  var modules = {};
  return function(name, requires) {
    if (requires) {
      return createModule(name, requires, modules);
    } else {
      return getModule(name, modules);
    }
  };
});
```

In `createModule` we must now store the newly created module object in `modules`:

src/loader.js

```
var createModule = function(name, requires, modules) {
  var moduleInstance = {
    name: name,
    requires: requires
  };
  modules[name] = moduleInstance;
  return moduleInstance;
};
```

In `getModule` we can then just look it up:

src/loader.js

```
var getModule = function(name, modules) {
  return modules[name];
};
```

We're basically retaining the memory of all modules registered inside the local `modules` variable. This is why it is so important to only define `angular` and `angular.module` once. Otherwise the local variable could be wiped out.

Right now, when you try to get a module that does not exist, you'll just get `undefined` as the return value. What should happen instead is an exception. Angular makes a big noise when you refer to a non-existing module so that it's clear that this is happening:

test/loader_spec.js


```
it('throws when trying to get a nonexisting module', function() {
  expect(function() {
    window.angular.module('myModule');
  }).toThrow();
});
```

In `getModule` we should check for the module's existence before trying to return it:

src/loader.js

```
var getModule = function(name, modules) {
  if (modules.hasOwnProperty(name)) {
    return modules[name];
  } else {
    throw 'Module '+name+' is not available!';
  }
};
```

Finally, as we are now using the `hasOwnProperty` method to check for a module's existence, we must be careful not to override that method in our module cache. That is, it should not be allowed to register a module called `hasOwnProperty` that would override the method:

test/loader_spec.js

```
it('does not allow a module to be called hasOwnProperty', function() {
  expect(function() {
    window.angular.module('hasOwnProperty', []);
  }).toThrow();
});
```

This check is needed in `createModule`:

src/loader.js

```
var createModule = function(name, requires, modules) {
  if (name === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid module name';
  }
  var moduleInstance = {
    name: name,
    requires: requires
  };
  modules[name] = moduleInstance;
  return moduleInstance;
};
```


The Injector

Let's shift gears a bit and lay the foundation for the other major player in Angular's dependency injection: The injector.

The injector is not part of the module loader, but an independent service in itself, so we'll put the code and tests for it in new files. In the tests, we will always assume a fresh module loader has been set up:

test/injector_spec.js

```
/* jshint globalstrict: true */
/* global createInjector: false, setupModuleLoader: false, angular: false */
'use strict';

describe('injector', function() {

  beforeEach(function() {
    delete window.angular;
    setupModuleLoader(window);
  });

});
```

One can create an injector by calling the function `createInjector`, which takes an array of module names and returns the injector object:

test/injector_spec.js

```
it('can be created', function() {
  var injector = createInjector([]);
  expect(injector).toBeDefined();
});
```

For now we can get away with an implementation that simply returns an empty object literal:

src/injector.js

```
/* jshint globalstrict: true */
/* global angular: false */
'use strict';

function createInjector(modulesToLoad) {
  return {};
}
```


Registering A Constant

The first type of Angular application component we will implement is *constants*. With `constant` you can register a simple value, such as a number, as string, an object, or a function, to an Angular module.

After we've registered a constant to a module and created an injector, we can use the injector's `has` method to check that it indeed knows about the constant:

test/injector_spec.js

```
it('has a constant that has been registered to a module', function() {
  var module = angular.module('myModule', []);
  module.constant('aConstant', 42);
  var injector = createInjector(['myModule']);
  expect(injector.has('aConstant')).toBe(true);
});
```

Here we see for the first time the full sequence of defining a module and then an injector for it. An interesting observation to make about it is that as we create an injector, we don't give it direct references to module objects. Instead we give it the *names* of the module objects, and expect it to look them up from `angular.module`.

As a sanity check, let's also make sure that the `has` method returns `false` for things that have not been registered:

test/injector_spec.js

```
it('does not have a non-registered constant', function() {
  var module = angular.module('myModule', []);
  var injector = createInjector(['myModule']);
  expect(injector.has('aConstant')).toBe(false);
});
```

So, how does a `constant` registered in a module become available in an injector? First of all, we'll need the registration method to exist in module objects:

src/loader.js

```
var createModule = function(name, requires, modules) {
  if (name === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid module name';
  }
  var moduleInstance = {
    name: name,
    requires: requires,
    constant: function(key, value) {
    }
  };
  modules[name] = moduleInstance;
  return moduleInstance;
};
```


A general rule about modules and injectors is that modules don't actually contain any application components. They just contain the *recipes* for creating application components, and the injector is where they will actually become concrete.

What the module should hold, then, is a collection of tasks - such as "register a constant" - that the injector should carry out when it loads the module. This collection of tasks is called the *invoke queue*. Every module has an invoke queue, and when the module is loaded by an injector, the injector runs the tasks from that module's invoke queue.

For now, we'll define the invoke queue as an array of arrays. Each array in the queue has two items: The type of application component that should be registered, and the arguments for registering that component. An invoke queue that defines a single constant - the one in our unit test - looks like this:

```
[
  ['constant', ['aConstant', 42]]
]
```

The invoke queue is stored in a module attribute called `_invokeQueue` (the underscore prefix denoting it should be considered private to the module). From the `constant` function we will now push an item to the queue:

src/loader.js

```
var createModule = function(name, requires, modules) {
  if (name === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid module name';
  }
  var moduleInstance = {
    name: name,
    requires: requires,
    constant: function(key, value) {
      moduleInstance._invokeQueue.push(['constant', [key, value]]);
    },
    _invokeQueue: []
  };
  modules[name] = moduleInstance;
  return moduleInstance;
};
```

As we then create the injector, we should iterate over all the module names given, look up the corresponding module objects, and then drain their invoke queues:

src/injector.js


```
function createInjector(modulesToLoad) {

  _.forEach(modulesToLoad, function(moduleName) {
    var module = angular.module(moduleName);
    _.forEach(module._invokeQueue, function(invokeArgs) {

    });
  });

  return {};
}
```

Inside the injector we will have some code that knows how to handle each of the items that an invoke queue might hold. We will put this code in an object called `$provide` (for reasons that will become clear later). As we iterate over the items in the invoke queue, we look up a method from `$provide` that corresponds to the first item in each invocation array (e.g. `'constant'`). We then call the method with the arguments stored in the *second* item of the invocation array:

src/injector.js

```
function createInjector(modulesToLoad) {

  var $provide = {
    constant: function(key, value) {

    }
  };

  _.forEach(modulesToLoad, function(moduleName) {
    var module = angular.module(moduleName);
    _.forEach(module._invokeQueue, function(invokeArgs) {
      var method = invokeArgs[0];
      var args = invokeArgs[1];
      $provide[method].apply($provide, args);
    });
  });

  return {};
}
```

So, when you call a method such as `constant` on a module, that will cause the same method with the same arguments to be called in the `$provide` object inside `createInjector`. It's just that this does not happen immediately, but only later as the module is loaded. In the meantime, the information about the method invocation is stored in the invoke queue.

What still remains is the actual logic of registering a constant. Generally, all application components will be cached by the injector. A constant is a simple value that we can plop right into the cache. We can then implement the injector's `has` method to check for the corresponding key in the cache:

src/injector.js


```
function createInjector(modulesToLoad) {
  var cache = {};

  var $provide = {
    constant: function(key, value) {
      cache[key] = value;
    }
  };

  _.forEach(modulesToLoad, function(moduleName) {
    var module = angular.module(moduleName);
    _.forEach(module._invokeQueue, function(invokeArgs) {
      var method = invokeArgs[0];
      var args = invokeArgs[1];
      $provide[method].apply($provide, args);
    });
  });

  return {
    has: function(key) {
      return cache.hasOwnProperty(key);
    }
  };
}
```

We now once again have a situation where we need to guard the `hasOwnProperty` property of an object. One should not be able to register a constant called `hasOwnProperty`:

test/injector_spec.js

```
it('does not allow a constant called hasOwnProperty', function() {
  var module = angular.module('myModule', []);
  module.constant('hasOwnProperty', _.constant(false));
  expect(function() {
    createInjector(['myModule']);
  }).toThrow();
});
```

We disallow this by checking the key in the `constant` method of `$provide`:

src/injector.js

```
constant: function(key, value) {
  if (key === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid constant name!';
  }
  cache[key] = value;
}
```


In addition to just checking whether an application component exists, the injector also gives you the means to obtain the component itself. For that, we'll introduce a method called `get`:

test/injector_spec.js

```
it('can return a registered constant', function() {
  var module = angular.module('myModule', []);
  module.constant('aConstant', 42);
  var injector = createInjector(['myModule']);
  expect(injector.get('aConstant')).toBe(42);
});
```

The method simply looks up the key from the cache:

src/injector.js

```
return {
  has: function(key) {
    return cache.hasOwnProperty(key);
  },
  get: function(key) {
    return cache[key];
  }
};
```

> Most of the Angular dependency injection features are implemented as a collaboration between the module loader in `loader.js` and the injector in `injector.js`. We'll put the tests for this kind of functionality in `injector_spec.js`, leaving `loader_spec.js` for tests that strictly deal with the module loader alone.

Requiring Other Modules

Thus far we've been creating an injector from a single module, but it is also possible to create one that loads multiple modules. The most straightforward way to do this is to just provide more than one module name in the array given to `createInjector`. Application components from all given modules will be registered:

test/injector_spec.js


```
it('loads multiple modules', function() {
  var module1 = angular.module('myModule', []);
  var module2 = angular.module('myOtherModule', []);
  module1.constant('aConstant', 42);
  module2.constant('anotherConstant', 43);
  var injector = createInjector(['myModule', 'myOtherModule']);

  expect(injector.has('aConstant')).toBe(true);
  expect(injector.has('anotherConstant')).toBe(true);
});
```

This we have already covered, because we iterate over the `modulesToLoad` array in `createInjector`.

Another way to cause several modules to be loaded is to *require* modules from other modules. When one registers a module using `angular.module`, there is that second array argument that we've kept empty so far, but that can hold the names of the required modules. When the module is loaded, its required modules are also loaded:

test/injector_spec.js

```
it('loads the required modules of a module', function() {
  var module1 = angular.module('myModule', []);
  var module2 = angular.module('myOtherModule', ['myModule']);
  module1.constant('aConstant', 42);
  module2.constant('anotherConstant', 43);
  var injector = createInjector(['myOtherModule']);

  expect(injector.has('aConstant')).toBe(true);
  expect(injector.has('anotherConstant')).toBe(true);
});
```

The same also works transitively, i.e. the modules required by the modules you require are also loaded, ad infinitum:

test/injector_spec.js

```
it('loads the transitively required modules of a module', function() {
  var module1 = angular.module('myModule', []);
  var module2 = angular.module('myOtherModule', ['myModule']);
  var module3 = angular.module('myThirdModule', ['myOtherModule']);
  module1.constant('aConstant', 42);
  module2.constant('anotherConstant', 43);
  module3.constant('aThirdConstant', 44);
  var injector = createInjector(['myThirdModule']);

  expect(injector.has('aConstant')).toBe(true);
  expect(injector.has('anotherConstant')).toBe(true);
  expect(injector.has('aThirdConstant')).toBe(true);
});
```


The way this works is actually quite simple. As we load a module, *before* iterating the module's invoke queue, we iterate its required modules, recursively loading each of them. We also need to give the module loading function a name so that we can call it recursively:

src/injector.js

```
_.forEach(modulesToLoad, function loadModule(moduleName) {
  var module = angular.module(moduleName);
  _.forEach(module.requires, loadModule);
  _.forEach(module._invokeQueue, function(invokeArgs) {
    var method = invokeArgs[0];
    var args = invokeArgs[1];
    $provide[method].apply($provide, args);
  });
});
```

When you have modules requiring other modules, it's quite easy to get into a situation where you have a circular dependency between two or more modules:

test/injector_spec.js

```
it('loads each module only once', function() {
  var module1 = angular.module('myModule', ['myOtherModule']);
  var module2 = angular.module('myOtherModule', ['myModule']);

  createInjector(['myModule']);
});
```

Our current implementation blows the stack while trying to load this, as it always recurses into the next module without checking if it actually should.

What we need to do to deal with circular dependencies is to make sure each module is loaded exactly once. This will also have the effect that when there are two (non-circular) paths to the same module, it will not be loaded twice, so the unnecessary extra work is avoided.

We'll introduce an object in which we keep track of the modules that have been loaded. Before we load a module we then check that it isn't already loaded:

src/injector.js

```
function createInjector(modulesToLoad) {
  var cache = {};
  var loadedModules = {};

  var $provide = {
    constant: function(key, value) {
```



```
      if (key === 'hasOwnProperty') {
        throw 'hasOwnProperty is not a valid constant name!';
      }
      cache[key] = value;
    }
  };

  _.forEach(modulesToLoad, function loadModule(moduleName) {
    if (!loadedModules.hasOwnProperty(moduleName)) {
      loadedModules[moduleName] = true;
      var module = angular.module(moduleName);
      _.forEach(module.requires, loadModule);
      _.forEach(module._invokeQueue, function(invokeArgs) {
        var method = invokeArgs[0];
        var args = invokeArgs[1];
        $provide[method].apply($provide, args);
      });
    }
  });

  return {
    has: function(key) {
      return cache.hasOwnProperty(key);
    },
    get: function(key) {
      return cache[key];
    }
  };
}
```

Dependency Injection

We now a semi-useful registry for application components, into which we can load things and from which we can look things up. But the real purpose of the injector is to do actual *dependency injection*. That is, to invoke functions and construct objects and automatically look up the dependencies they need. For the remainder of this chapter, we'll focus on the dependency injection features of the injector.

The basic idea is this: We'll give the injector a function and ask it to invoke that function. We'll also expect it to figure out what arguments that function needs and provide them to it.

So, how can the injector figure out what arguments a given function needs? The easiest approach is to supply that information explicitly, using an attribute called `$inject` attached to the function. That attribute can hold an array of the names of the function's dependencies. The injector will look those dependencies up and invoke the function with them:

test/injector_spec.js


```
it('invokes an annotated function with dependency injection', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.constant('b', 2);
  var injector = createInjector(['myModule']);

  var fn = function(one, two) { return one + two; };
  fn.$inject = ['a', 'b'];

  expect(injector.invoke(fn)).toBe(3);
});
```

This can be implemented by simply looking up each item of the `$inject` array from the injector's cache, where we hold the mapping from dependency names to their values:

src/injector.js

```
function createInjector(modulesToLoad) {
  var cache = {};
  var loadedModules = {};

  var $provide = {
    constant: function(key, value) {
      if (key === 'hasOwnProperty') {
        throw 'hasOwnProperty is not a valid constant name!';
      }
      cache[key] = value;
    }
  };

  _.forEach(modulesToLoad, function loadModule(moduleName) {
    if (!loadedModules.hasOwnProperty(moduleName)) {
      loadedModules[moduleName] = true;
      var module = angular.module(moduleName);
      _.forEach(module.requires, loadModule);
      _.forEach(module._invokeQueue, function(invokeArgs) {
        var method = invokeArgs[0];
        var args = invokeArgs[1];
        $provide[method].apply($provide, args);
      });
    }
  });

  var invoke = function(fn) {
    var args = _.map(fn.$inject, function(token) {
      return cache[token];
    });
    return fn.apply(null, args);
  };
```



```
  return {
    has: function(key) {
      return cache.hasOwnProperty(key);
    },
    get: function(key) {
      return cache[key];
    },
    invoke: invoke
  };
}
```

This is the most low-level of the dependency annotation approaches you can use with Angular.

Rejecting Non-String DI Tokens

We've seen how the `$inject` array should contain the dependency names. If someone was to put something invalid, like, say, a number to the `$inject` array, our current implementation would just map that to `undefined`. We should throw an exception instead, to let the user know they're doing something wrong:

test/injector_spec.js

```
it('does not accept non-strings as injection tokens', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  var injector = createInjector(['myModule']);

  var fn = function(one, two) { return one + two; };
  fn.$inject = ['a', 2];

  expect(function() {
    injector.invoke(fn);
  }).toThrow();
});
```

This can be done with a simple type check inside the dependency mapping function:

src/injector.js

```
var invoke = function(fn) {
  var args = _.map(fn.$inject, function(token) {
    if (_.isString(token)) {
      return cache[token];
    } else {
      throw 'Incorrect injection token! Expected a string, got '+token;
    }
  });
  return fn.apply(null, args);
};
```


Binding `this` in Injected Functions

Sometimes the functions you want to inject are actually methods attached to objects. In such methods, the value of `this` may be significant. When called directly, the JavaScript language takes care of binding `this`, but when called indirectly through `injector.invoke` there's no such automatic binding. Instead, we can give `injector.invoke` the `this` value as an optional second argument, which it will bind when it invokes the function:

test/injector_spec.js

```
it('invokes a function with the given this context', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  var injector = createInjector(['myModule']);

  var obj = {
    two: 2,
    fn: function(one) { return one + this.two; }
  };
  obj.fn.$inject = ['a'];

  expect(injector.invoke(obj.fn, obj)).toBe(3);
});
```

As we are already using `Function.apply` for invoking the function, we can just pass the value along to it (where we previously supplied `null`):

src/injector.js

```
var invoke = function(fn, self) {
  var args = _.map(fn.$inject, function(token) {
    if (_.isString(token)) {
      return cache[token];
    } else {
      throw 'Incorrect injection token! Expected a string, got '+token;
    }
  });
  return fn.apply(self, args);
};
```


Providing Locals to Injected Functions

Most often you just want the injector to provide all the arguments to a function, but there may be cases where you want to explicitly provide some of the arguments during invocation. This may be because you want to override some of the arguments, or because some of the arguments may not be registered to the injector at all.

For this purpose, `injector.invoke` takes an optional *third* argument, which is an object of local mappings from dependency names to values. If supplied, dependency lookup is done primarily from this object, and secondarily from the injector itself. This is a similar approach as the one we've seen earlier in `$scope.$eval`:

test/injector_spec.js

```
it('overrides dependencies with locals when invoking', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.constant('b', 2);
  var injector = createInjector(['myModule']);

  var fn = function(one, two) { return one + two; };
  fn.$inject = ['a', 'b'];

  expect(injector.invoke(fn, undefined, {b: 3})).toBe(4);
});
```

In the dependency mapping function, we'll look at the locals first, if given, and fall back to `cache` if there's nothing to be found in `locals`:

src/injector.js

```
var invoke = function(fn, self, locals) {
  var args = _.map(fn.$inject, function(token) {
    if (_.isString(token)) {
      return locals && locals.hasOwnProperty(token) ?
        locals[token] :
        cache[token];
    } else {
      throw 'Incorrect injection token! Expected a string, got '+token;
    }
  });
  return fn.apply(self, args);
};
```


Array-Style Dependency Annotation

While you can always annotate an injected function using the `$inject` attribute, you might not always *want* to do that because of the verbosity of the approach. A slightly less verbose option for providing the dependency names is to supply `injector.invoke` *an array* instead of a function. In that array, you first give the names of the dependencies, and as the last item the actual function to invoke:

```
['a', 'b', function(one, two) {
  return one + two;
}]
```

Since we are now talking about several different approaches to annotating a function, a method that is able to extract any type of annotation is called for. Such a method is in fact provided by the injector. It is called `annotate`.

Let's specify this method's behavior in a new nested `describe` block. Firstly, when given a function with an `$inject` attribute, `annotate` just returns that attribute's value:

test/injector_spec.js

```
describe('annotate', function() {

  it('returns a functions $inject annotation when it has one', function() {
    var injector = createInjector([]);

    var fn = function() { };
    fn.$inject = ['a', 'b'];

    expect(injector.annotate(fn)).toEqual(['a', 'b']);
  });

});
```

We'll introduce a local function in the `createInjector` closure, exposed as a property of the injector:

src/injector.js

```
function createInjector(modulesToLoad) {
  var cache = {};
  var loadedModules = {};

  var $provide = {
    constant: function(key, value) {
      if (key === 'hasOwnProperty') {
        throw 'hasOwnProperty is not a valid constant name!';
      }
      cache[key] = value;
    }
```



```
  };

  _.forEach(modulesToLoad, function loadModule(moduleName) {
    if (!loadedModules.hasOwnProperty(moduleName)) {
      loadedModules[moduleName] = true;
      var module = angular.module(moduleName);
      _.forEach(module.requires, loadModule);
      _.forEach(module._invokeQueue, function(invokeArgs) {
        var method = invokeArgs[0];
        var args = invokeArgs[1];
        $provide[method].apply($provide, args);
      });
    }
  });

  var annotate = function(fn) {
    return fn.$inject;
  };

  var invoke = function(fn, self, locals) {
    var args = _.map(fn.$inject, function(token) {
      if (_.isString(token)) {
        return locals && locals.hasOwnProperty(token) ?
          locals[token] :
          cache[token];
      } else {
        throw 'Incorrect injection token! Expected a string, got '+token;
      }
    });
    return fn.apply(self, args);
  };

  return {
    has: function(key) {
      return cache.hasOwnProperty(key);
    },
    get: function(key) {
      return cache[key];
    },
    annotate: annotate,
    invoke: invoke
  };
}
```

When given an array, `annotate` extracts the dependency names from that array, based on our definition of array-style injection:

test/injector_spec.js

```
it('returns the array-style annotations of a function', function() {
  var injector = createInjector([]);

  var fn = ['a', 'b', function() { }];

  expect(injector.annotate(fn)).toEqual(['a', 'b']);
});
```

So, if `fn` is an array, `annotate` should return an array of all but the last item of it:

src/injector.js

```
var annotate = function(fn) {
  if (_.isArray(fn)) {
    return fn.slice(0, fn.length - 1);
  } else {
    return fn.$inject;
  }
};
```

Dependency Annotation from Function Arguments

The third and final, and perhaps the most interesting way to define a function's dependencies is to not actually define them at all. When the injector is given a function without an `$inject` attribute and without an array wrapping, it will attempt to extract the dependency names from the *function itself*.

Let's handle the easy case of a function with zero arguments first:

test/injector_spec.js

```
it('returns an empty array for a non-annotated 0-arg function', function() {
  var injector = createInjector([]);

  var fn = function() { };

  expect(injector.annotate(fn)).toEqual([]);
});
```

From `annotate` we'll return an empty array if the function is not annoted. This'll make the test pass:

src/injector.js

```
var annotate = function(fn) {
  if (_.isArray(fn)) {
    return fn.slice(0, fn.length - 1);
  } else if (fn.$inject) {
    return fn.$inject;
  } else {
    return [];
  }
};
```

If the function does have arguments though, we'll need to figure out a way to extract them so that the following test will also pass:

test/injector_spec.js

```
it('returns annotations parsed from function args when not annotated', function() {
  var injector = createInjector([]);

  var fn = function(a, b) { };

  expect(injector.annotate(fn)).toEqual(['a', 'b']);
});
```

The trick is to *read in the source code of the function and extract the argument declarations using a regular expression.* In JavaScript, you can get a function's source code by calling the `toString` method of the function:

```
(function(a, b) { }).toString() // => "function (a, b) { }"
```

Since the source code contains the function's argument list, we can grab it using the following regexp, which we'll define as a "constant" at the top of `injector.js`:

src/injector.js

```
var FN_ARGS = /^function\s*[^\(]*\(\s*([^\)]*)\)/m;
```

The regexp can be broken down as follows:

`/^`	We begin by anchoring the match to the beginning of input
`function`	Every function begins with the `function` keyword...
`\s*`	...followed by (optionally) some whitespace...
`[^\(]*`	...followed by the (optional) function name - characters other than '('...
`\(`	...followed by the opening parenthesis of the argument list...
`\s*`	...followed by (optionally) some whitespace...

`(`	...followed by the argument list, which we capture in a capturing group...
`[^\)]*`	...into which we read a succession of any characters other than ')'...
`)`	...and when done reading we close the capturing group...
`\)`	...and still match the closing parenthesis of the argument list...
`/m`	...and define the whole regular expression to match over multiple lines.

When we match using this regexp in `annotate`, we'll get the argument list from the capturing group as the second item of the match result. By then splitting that at `','` we'll get the array of argument names. For the function with no arguments (detected using the `Function.length` attribute) we'll add a special case of the empty array:

src/injector.js

```
var annotate = function(fn) {
  if (_.isArray(fn)) {
    return fn.slice(0, fn.length - 1);
  } else if (fn.$inject) {
    return fn.$inject;
  } else if (!fn.length) {
    return [];
  } else {
    var argDeclaration = fn.toString().match(FN_ARGS);
    return argDeclaration[1].split(',');
  }
};
```

As you implement this, you'll notice that our test is still failing. That's because there's some extra whitespace in the second dependency name: `' b'`. Our regexp gets rid of the whitespace at the beginning of the argument list, but not of the whitespace *between* argument names. To get rid of that whitespace, we'll need to iterate over the argument names before returning them.

The following regexp will match any heading and trailing whitespace in a string, and capture the non-whitespace section in between in a capturing group:

src/injector.js

```
var FN_ARG = /^\s*(\S+)\s*$/;
```

By mapping the argument names to the second match result of this regexp we can get the cleaned-up argument names:

src/injector.js

```
var annotate = function(fn) {
  if (_.isArray(fn)) {
    return fn.slice(0, fn.length - 1);
  } else if (fn.$inject) {
    return fn.$inject;
  } else if (!fn.length) {
    return [];
  } else {
    var argDeclaration = fn.toString().match(FN_ARGS);
    return _.map(argDeclaration[1].split(','), function(argName) {
      return argName.match(FN_ARG)[1];
    });
  }
};
```

The simple case of "on-the-fly" dependency annotation now works, but what happens if there are some commented-out arguments in the function declaration:

test/injector_spec.js

```
it('strips comments from argument lists when parsing', function() {
  var injector = createInjector([]);

  var fn = function(a, /*b,*/ c) { };

  expect(injector.annotate(fn)).toEqual(['a', 'c']);
});
```

> Here we run into some differences between web browsers. Some browsers return the source code from `Function.toString()` with the comments stripped, and some leave them in. The WebKit in the current version of PhantomJS strips the comments, so this test may pass immediately in your environment. Chrome does not strip the comments, so if you want to see the failing tests, connect Testem to Chrome at least for the duration of this section.

Before extracting the function arguments, we'll need to preprocess the function source code to strip away any comments it might contain. This regexp is our first attempt at doing so:

src/injector.js

```
var STRIP_COMMENTS = /\/\*.*\*\//;
```

The rexep matches the characters `/*`, then a succession of any characters, and then the characters `*/`. By replacing the match result of this regexp with an empty string we can strip the comment:

src/injector.js


```
var annotate = function(fn) {
  if (_.isArray(fn)) {
    return fn.slice(0, fn.length - 1);
  } else if (fn.$inject) {
    return fn.$inject;
  } else if (!fn.length) {
    return [];
  } else {
    var source = fn.toString().replace(STRIP_COMMENTS, '');
    var argDeclaration = source.match(FN_ARGS);
    return _.map(argDeclaration[1].split(','), function(argName) {
      return argName.match(FN_ARG)[1];
    });
  }
};
```

This first attempt doesn't quite cut it when there are *several* commented-out sections in the argument list:

src/injector.js

```
it('strips several comments from argument lists when parsing', function() {
  var injector = createInjector([]);

  var fn = function(a, /*b,*/ c/*, d*/) { };

  expect(injector.annotate(fn)).toEqual(['a', 'c']);
});
```

What's happening is the regexp matches everything between the first opening `/*` and the last closing `*/`, so that any non-comment sections in between are lost. We'll need to convert the quantifier between the opening and closing comments to a lazy one, so that it'll consume as little as possible. We also need to add the `g` modifier to the regexp to have it match multiple comments in the string:

src/injector.js

```
var STRIP_COMMENTS = /\/\*.*?\*\//g;
```

Then there's still the other kind of comments an argument list spread over multiple lines might include: `//` style comments that comment out the remainder of a line:

test/injector_spec.js


```
it('strips // comments from argument lists when parsing', function() {
  var injector = createInjector([]);

  var fn = function(a, //b,
                    c) { };

  expect(injector.annotate(fn)).toEqual(['a', 'c']);
});
```

To strip these comments out, we'll have our `STRIP_COMMENTS` regexp match two different kinds of input: The input we defined earlier, and an input that begins with two forward slashes `//` and is followed by any characters until the line endsf. We'll also add the `m` modifier to the regexp to make it match over multiple lines:

src/injector.js

```
var STRIP_COMMENTS = /(\/\/.*$)|(\/\*.*?\*\/)/mg;
```

And this takes care of all kinds of comments inside argument lists!

The final feature we need to take care of when parsing argument names is stripping surrounding underscore characters from them. Angular lets you put an underscore character on both sides of an argument name, which it will then ignore, so that the following pattern of capturing an injected argument to a local variable with the same name is possible:

```
var aVariable;
injector.invoke(function(_aVariable_) {
  aVariable = _aVariable_;
});
```

So, if an argument is surrounded by underscores on both sides, they should be stripped from the resulting dependency name. If there's an underscore on just one side of the argument name, or somewhere in the middle, it should be left in as-is:

test/injector_spec.js

```
it('strips surrounding underscores from argument names when parsing', function() {
  var injector = createInjector([]);

  var fn = function(a, _b_, c_, _d, an_argument) { };

  expect(injector.annotate(fn)).toEqual(['a', 'b', 'c_', '_d', 'an_argument']);
});
```

Underscore stripping is done by the `FN_ARG` regexp we've previously used for stripping whitespace. It should also match an optional underscore before the argument name, and then match the same thing after the argument name using a backreference:

src/injector.js


```
var FN_ARG = /^\s*(_?)(\S+?)\1\s*$/;
```

Now that we've added a new capturing group to the regexp, the actual argument name will be in the *third* item of the match result, not the second:

src/injector.js

```
var annotate = function(fn) {
  if (_.isArray(fn)) {
    return fn.slice(0, fn.length - 1);
  } else if (fn.$inject) {
    return fn.$inject;
  } else if (!fn.length) {
    return [];
  } else {
    var source = fn.toString().replace(STRIP_COMMENTS, '');
    var argDeclaration = source.match(FN_ARGS);
    return _.map(argDeclaration[1].split(','), function(argName) {
      return argName.match(FN_ARG)[2];
    });
  }
};
```

Integrating Annotation with Invocation

We are now able to extract dependency names using the three different methods that Angular supports: `$inject`, array wrapper, and function source extraction. What we still need to do is to integrate this dependency name lookup to `injector.invoke`. You should be able to give it an array-annotated function and expect it to do the right thing:

test/injector_spec.js

```
it('invokes an array-annotated function with dependency injection', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.constant('b', 2);
  var injector = createInjector(['myModule']);

  var fn = ['a', 'b', function(one, two) { return one + two; }];

  expect(injector.invoke(fn)).toBe(3);
});
```


In exactly the same way, you should be able to give it a non-annotated function and expect it to parse the dependency annotations from the source:

test/injector_spec.js

```
it('invokes a non-annotated function with dependency injection', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.constant('b', 2);
  var injector = createInjector(['myModule']);

  var fn = function(a, b) { return a + b; };

  expect(injector.invoke(fn)).toBe(3);
});
```

In `invoke` we'll need to do two things: Firstly, we need to look up the dependency names using `annotate()` instead of accessing `$inject` directly. Secondly, we need to check if the function given was wrapped into an array, and unwrap it if necessary before trying to invoke it:

src/injector.js

```
var invoke = function(fn, self, locals) {
  var args = _.map(annotate(fn), function(token) {
    if (_.isString(token)) {
      return locals && locals.hasOwnProperty(token) ?
                locals[token] :
                cache[token];
    } else {
      throw 'Incorrect injection token! Expected a string, got '+token;
    }
  });
  if (_.isArray(fn)) {
    fn = _.last(fn);
  }
  return fn.apply(self, args);
};
```

And now we can feed any of the three kinds of functions to invoke!

Instantiating Objects with Dependency Injection

We'll conclude this chapter by adding one more capability to the injector: Injecting not only plain functions but also constructor functions.

When you have a constructor function and want to instantiate an object using that function, while also injecting its dependencies, you can use `injector.instantiate`. It can handle a constructor that has an explicit `$inject` annotation attached:

test/injector_spec.js


```
it('instantiates an annotated constructor function', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.constant('b', 2);
  var injector = createInjector(['myModule']);

  function Type(one, two) {
    this.result =  one + two;
  }
  Type.$inject = ['a', 'b'];

  var instance = injector.instantiate(Type);
  expect(instance.result).toBe(3);
});
```

You can also use array-wrapper style annotations:

test/injector_spec.js

```
it('instantiates an array-annotated constructor function', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.constant('b', 2);
  var injector = createInjector(['myModule']);

  function Type(one, two) {
    this.result = one + two;
  }

  var instance = injector.instantiate(['a', 'b', Type]);
  expect(instance.result).toBe(3);
});
```

And, just like for plain functions, the injector should be able to extract the dependency names from the constructor function itself:

test/injector_spec.js

```
it('instantiates a non-annotated constructor function', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.constant('b', 2);
  var injector = createInjector(['myModule']);

  function Type(a, b) {
    this.result = a + b;
  }

  var instance = injector.instantiate(Type);
  expect(instance.result).toBe(3);
});
```


Let's introduce the new `instantiate` method in the injector. It points to a local function, which we'll introduce momentarily:

src/injector.js

```
return {
  has: function(key) {
    return cache.hasOwnProperty(key);
  },
  get: function(key) {
    return cache[key];
  },
  annotate: annotate,
  invoke: invoke,
  instantiate: instantiate
};
```

A simplistic implementation of `instantiate` could just make a new object, invoke the constructor function with the new object bound to `this`, and then return the new object:

src/injector.js

```
var instantiate = function(Type) {
  var instance = {};
  invoke(Type, instance);
  return instance;
};
```

This does indeed make our existing tests pass. But there's one important behavior of using a constructor function that we're forgetting: When you construct an object with `new`, you also set up the *prototype chain* of the object based on the prototype chain of the constructor. We should respect this behavior in `injector.instantiate`.

For example, if the constructor we're instantiating has a prototype where some additional behavior is defined, that behavior should be available to the resulting object through inheritance:

test/injector_spec.js

```
it('uses the prototype of the constructor when instantiating', function() {
  function BaseType() { }
  BaseType.prototype.getValue = _.constant(42);

  function Type() { this.v = this.getValue(); }
  Type.prototype = BaseType.prototype;

  var module = angular.module('myModule', []);
  var injector = createInjector(['myModule']);

  var instance = injector.instantiate(Type);
  expect(instance.v).toBe(42);
});
```


To set up the prototype chain, we can construct the object using the ES5.1 Object.create function instead of just making a simple literal. We also need to remember to unwrap the constructor because it might use array dependency annotations:

src/injector.js

```
var instantiate = function(Type) {
  var UnwrappedType = _.isArray(Type) ? _.last(Type) : Type;
  var instance = Object.create(UnwrappedType.prototype);
  invoke(Type, instance);
  return instance;
};
```

> Angular.js does not use `Object.create` here because it isn't supported by some older browsers we don't care about. Instead, it goes through some manual contortions to achieve the same result.

Finally, just like `injector.invoke` supports supplying a `locals` object, so should `injector.instantiate`. It can be given as an optional second argument:

test/injector_spec.js

```
it('supports locals when instantiating', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.constant('b', 2);
  var injector = createInjector(['myModule']);

  function Type(a, b) {
    this.result = a + b;
  }

  var instance = injector.instantiate(Type, {b: 3});
  expect(instance.result).toBe(4);
});
```

We just need to take the `locals` argument and pass it along to `invoke` as its third argument:

src/injector.js

```
var instantiate = function(Type, locals) {
  var UnwrappedType = _.isArray(Type) ? _.last(Type) : Type;
  var instance = Object.create(UnwrappedType.prototype);
  invoke(Type, instance, locals);
  return instance;
};
```


Summary

We've now begun our journey towards the fully featured Angular.js dependency injection framework. At this point we already have a perfectly serviceable module system and an injector into which you can register constants, and using which you can inject functions and constructors.

In this chapter you have learned:

- How the `angular` global variable and its `module` method come to be.
- How modules can be registered.
- That the `angular` global will only ever be registered once per window, but any given module can be overridden by a later registration with the same name.
- How previously registered modules can be looked up.
- How an injector comes to be.
- How the injector is given names of modules to instantiate, which it will look up from the `angular` global.
- How application component registrations in modules are queued up and only instantiated when the injector loads the module.
- How modules can require other modules and that the required modules are loaded first by the injector.
- That the injector loads each module only once to prevent unnecessary work and problems with circular requires.
- How the injector can be used to invoke a function and how it can look up its arguments from its `$inject` annotation.
- How the injected function's `this` keyword can be bound by supplying it to `injector.invoke`.
- How a function's dependencies can be overridden or augmented by supplying a locals object to `injector.invoke`.
- How array-wrapper style function annotation works.
- How function dependencies can be looked up from the function's source code.
- How the dependencies of any given function can be extracted using `injector.annotate`.
- How objects can be instantiated with dependency injection using `injector.instantiate`.

In the next chapter we'll focus on *Providers* - one of the central building blocks of the Angular DI system, on which many of the high-level features are built.

Chapter 10

Providers

Chapter 11

High-Level Dependency Injection Features

Part IV

Utilities

Part V

Directives

Part VI

The Directive Library

Part VII

The Core Extension Modules