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 for their valuable feedback and help during the writing of this book:

- Iftach Bar
- Xi Chen
- Wil Pannell
- Pavel Pomyerantsyev
- Mauricio Poppe
- Mika Ristimäki
- Jesus Rodriguez
- Scott Silvi

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

2015-08-24: Maintenance Release

Added a number of Angular 1.4 updates and fixed errata.

2015-07-31: Chapter 8: Filters

Added Chapter 8 and added filter support to Chapters 9 and 12.

2015-07-23: Expression Parser Rewrite

Revisited Part 2 of the book to cover the Angular 1.4+ expression parser.

2015-06-15: Chapter 21: Interpolation

Added Chapter 21.

2015-06-08: Chapter 20: Directive Transclusion

Added Chapter 20 and fixed a few errata.

2015-04-27: Chapter 19: Directive Templates

Added Chapter 19 and fixed a few errata.

2015-04-05: Chapter 14: $http

Added Chapter 14 and fixed a number of errata.

2015-02-23: Chapter 13: Promises

Added Chapter 13.

2015-01-03: Chapter 16: Controllers

Added Chapter 16 and fixed a number of errata.

2014-12-24: Chapter 15: Directive Linking And Scopes

Added Chapter 15.

2014-11-16: Angular 1.3 updates

Added coverage of several smaller changes in Angular 1.3 across all chapters. Fixed a few errata.

2014-10-26: Added $applyAsync

Added coverage of Angular 1.3 $applyAsync to Chapters 1 and 2. Fixed a few errata.

2014-10-12: Added One-Time Binding and Input Tracking

Reworked the Expressions part of the book for Angular 1.3, with its new features - both internal and user-facing. Fixed a few errata.

2014-09-21: Added `$watchGroup`

Coverage of the new Angular 1.3 `$watchGroup` feature in Chapter 1 and fixed a number of errata.

2014-08-16: Chapter 13: Directive Attributes

Added Chapter 13.

2014-08-09: Maintenance Release

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

2014-08-04: Chapter 12: DOM Compilation and Basic Directives

Added Chapter 12.

2014-06-14: Chapter 11: High-Level Dependency Injection Features.

Added Chapter 11 and fixed a number of errata.

2014-05-24: Chapter 10: Providers

Added Chapter 10.

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. A few directories are needed at this point: The project root directory, a `src` directory for sources, and a `test` directory for unit 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. This update to `package.json` was caused by the `--save-dev` flag.

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, Sinon, 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.

We're also going to use a test helper library called Sinon.JS for some of the more sophisticated mock objects we're going to need. Sinon will become particularly helpful when we start building HTTP features.

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

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

While we're at it, let's install Sinon too:

```
npm install sinon --save-dev
```

Next, load and configure Testem and Jasmine in `Gruntfile.js`, and add some additional global variables to JSHint's configuration:

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,
          afterEach: false,
          sinon: false
        },
        browser: true,
        devel: true
      }
    },
    testem: {
      unit: {
        options: {
          framework: 'jasmine2',
          launch_in_dev: ['PhantomJS'],
          before_tests: 'grunt jshint',
          serve_files: [
            'node_modules/sinon/pkg/sinon.js',
            '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 (globally, by enabling the `-g` flag) 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.

> The terminology used by Jasmine (and rSpec) comes from Behavior-Driven Development. The `describe` and `it` blocks describe the *behavior* of our code. This is also where the file suffix `_spec.js` comes from. The test files are *specifications* for our code.

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 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/index.js',
            'node_modules/jquery/dist/jquery.js',
            'node_modules/sinon/pkg/sinon.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 passes 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();
  });
};
```


> The `var self = this;` pattern is something we'll be using throughout the book to get around JavaScript's peculiar binding of `this`. There is a good A List Apart article that describes the problem and the pattern.

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;
};
```

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.

> Explicitly returning `false` in a `_.forEach` loop causes LoDash to short-circuit the loop and exit immmediately.

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. 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 $evalAsync'ed 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 `$evalAsync` will be executed in the same digest loop. But what happens if you schedule an `$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 $evalAsync'ed 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 $evalAsync'ed 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.

> Though `$evalAsync` does schedule a `$digest`, the preferred way to asynchronously execute code within a digest is using `$applyAsync`, introduced in the next section.

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.

Coalescing $apply Invocations - $applyAsync

While `$evalAsync` can be used to defer work either from inside a digest or from outside one, it is really designed for the former use case. The digest-launching `setTimeout` call is there mostly just to prevent confusion if someone was to call `$evalAsync` from outside a digest.

For the use case of `$applying` a function from outside a digest loop asynchronously, there is also a specialized function called `$applyAsync`. It is designed to be used like `$apply` is - for integrating code that's not aware of the Angular digest cycle. Unlike `$apply`, it does not evaluate the given function immediately nor does it launch a digest immediately. Instead, it schedules both of these things to happen after a short period of time.

The original motivation for adding `$applyAsync` was handling HTTP responses: Whenever the `$http` service receives a response, any response handlers are invoked and a digest is launched. That means a digest is run for each HTTP response. This may cause performance problems with applications that have a lot of HTTP traffic (as many apps do during startup) and/or an expensive digest cycle. The `$http` service can now be configured to use `$applyAsync` instead, in which case HTTP responses arriving very close to each other will be coalesced into a single digest. However, `$applyAsync` is in no way tied to the `$http` service, and you can use it for anything that might benefit from coalescing digests.

As we see in our first test for it, when we `$applyAsync` a function, it does not immediately cause anything to happen, but 50 milliseconds later it will have:

test/scope_spec.js

```
it('allows async $apply with $applyAsync', function(done) {
  scope.counter = 0;

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

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

  scope.$applyAsync(function(scope) {
    scope.aValue = 'abc';
  });
  expect(scope.counter).toBe(1);

  setTimeout(function() {
    expect(scope.counter).toBe(2);
    done();
  }, 50);
});
```


So far this is no different from `$evalAsync`, but we start to see the difference when we call `$applyAsync` from a listener function. If we used `$evalAsync`, the function would still be called during the same digest. But `$applyAsync` *always* defers the invocation:

test/scope_spec.js

```
it("never executes $applyAsync'ed function in the same cycle", function(done) {
  scope.aValue = [1, 2, 3];
  scope.asyncApplied = false;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.$applyAsync(function(scope) {
        scope.asyncApplied = true;
      });
    }
  );

  scope.$digest();
  expect(scope.asyncApplied).toBe(false);
  setTimeout(function() {
    expect(scope.asyncApplied).toBe(true);
    done();
  }, 50);
});
```

Let's begin the implementation of `$applyAsync` by introducing another queue in the Scope constructor. This is for work that has been scheduled with `$applyAsync`:

src/scope.js

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

When someone calls `$applyAsync`, we'll push a function to the queue. The function will later evaluate the given expression in the context of the scope, just like `$apply` does:

src/scope.js

```
Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
});
```


What we should also do here is actually schedule the function application. We can do this with `setTimeout` and a delay of 0. In that timeout, we `$apply` a function which drains the queue and invokes all the functions in it:

src/scope.js

```
Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
  setTimeout(function() {
    self.$apply(function() {
      while (self.$$applyAsyncQueue.length) {
        self.$$applyAsyncQueue.shift()();
      }
    });
  }, 0);
};
```

> Note that we do not `$apply` each individual item in the queue. We only `$apply` once, outside the loop. We only want to digest once here.

As we discussed, the main point of `$applyAsync` is to optimize things that happen in quick succession so that they only need a single digest. We haven't got this exactly right yet. Each call to `$applyAsync` currently schedules a new digest, which is plain to see if we increment a counter in a watch function:

test/scope_spec.js

```
it('coalesces many calls to $applyAsync', function(done) {
  scope.counter = 0;

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

  scope.$applyAsync(function(scope) {
    scope.aValue = 'abc';
  });
  scope.$applyAsync(function(scope) {
    scope.aValue = 'def';
```



```
  });

  setTimeout(function() {
    expect(scope.counter).toBe(2);
    done();
  }, 50);
});
```

We want the counter to be 2 (the watch is executed twice on the first digest), not any more than that.

What we need to do is keep track of whether a `setTimeout` to drain the queue has already been scheduled. We'll keep this information in a private scope attribute called `$$applyAsyncId`:

src/scope.js

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

We can then check this attribute when scheduling the job, and maintain its state when the job is scheduled and when it finishes:

src/scope.js

```
Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
  if (self.$$applyAsyncId === null) {
    self.$$applyAsyncId = setTimeout(function() {
      self.$apply(function() {
        while (self.$$applyAsyncQueue.length) {
          self.$$applyAsyncQueue.shift()();
        }
        self.$$applyAsyncId = null;
      });
    }, 0);
  }
};
```

Another aspect of `$applyAsync` is that it should not do a digest if one happens to be launched for some other reason before the timeout triggers. In those cases the digest should drain the queue and the `$applyAsync` timeout should be cancelled:

test/scope_spec.js

```
it('cancels and flushes $applyAsync if digested first', function(done) {
  scope.counter = 0;

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

  scope.$applyAsync(function(scope) {
    scope.aValue = 'abc';
  });
  scope.$applyAsync(function(scope) {
    scope.aValue = 'def';
  });

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

  setTimeout(function() {
    expect(scope.counter).toBe(2);
    done();
  }, 50);
});
```

Here we test that everything we have scheduled with `$applyAsync` happens immediately if we call `$digest`. That leaves nothing to be done later.

Let's first extract the flushing of the queue out of the inner function in `$applyAsync` itself, so that we can call it from multiple locations:

src/scope.js

```
Scope.prototype.$$flushApplyAsync = function() {
  while (this.$$applyAsyncQueue.length) {
    this.$$applyAsyncQueue.shift()();
  }
  this.$$applyAsyncId = null;
};
```

src/scope.js


```
Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
  if (self.$$applyAsyncId === null) {
    self.$$applyAsyncId = setTimeout(function() {
      self.$apply(_.bind(self.$$flushApplyAsync, self));
    }, 0);
  }
};
```

> The LoDash `_.bind` function is equivalent to ECMAScript 5 Function.prototype.bind, and is used to make sure the `this` receiver of the function is a known value.

Now we can also call this function from `$digest` - if there's an `$applyAsync` flush timeout currently pending, we cancel it and flush the work immediately:

src/scope.js

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

  if (this.$$applyAsyncId) {
    clearTimeout(this.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

  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);
  this.$clearPhase();
};
```

And that's it for `$applyAsync`. It is a useful little optimization for situations where you need to `$apply`, but know you'll be doing it several times within a short period of time.

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 `$evalAsync` and `$applyAsync`, `$$postDigest` schedules a function to run "later". Specifically, the function will be run *after* the next digest has finished. Similarly to `$evalAsync`, a function scheduled with `$$postDigest` is executed just once. Unlike `$evalAsync` or `$applyAsync`, 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.$$asyncQueue = [];
  this.$$applyAsyncQueue = [];
  this.$$applyAsyncId = null;
  this.$$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");

  if (this.$$applyAsyncId) {
    clearTimeout(this.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

  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 `$evalAsync` or `$applyAsync` 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 `$evalAsync`, `$applyAsync`, and `$$postDigest`. All of them are used to execute arbitrary functions in relation to the digest loop. In none of them 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 `$applyAsync`, we'll define a test case checking that a function scheduled with `$applyAsync` is invoked even if it has functions before it that throw exceptions.

test/scope_spec.js

```
it("catches exceptions in $applyAsync", function(done) {
  scope.$applyAsync(function(scope) {
    throw "Error";
  });
  scope.$applyAsync(function(scope) {
    throw "Error";
  });
  scope.$applyAsync(function(scope) {
    scope.applied = true;
  });

  setTimeout(function() {
    expect(scope.applied).toBe(true);
    done();
  }, 50);
});
```

> We use two error-throwing functions, because if we used just one, the second function would indeed run. That's because `$apply` launches `$digest`, and the `$applyAsync` queue drainage therein from a `finally` block.

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`:

src/scope.js

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

  if (this.$$applyAsyncId) {
    clearTimeout(this.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

  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);
    }
  }
};
```

Fixing `$applyAsync`, on the other hand, is done in the loop that drains the queue in `$$flushApplyAsync`:

src/scope.js


```
Scope.prototype.$$flushApplyAsync = function() {
  while (this.$$applyAsyncQueue.length) {
    try {
      this.$$applyAsyncQueue.shift()();
    } catch (e) {
      console.error(e);
    }
  }
  this.$$applyAsyncId = null;
};
```

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 middlemost watch removes itself when it is first called, leaving only the first and the third watch. 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 watcher 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);
```

```
    }
  };
};
```

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.

Watching Several Changes With One Listener: `$watchGroup`

So far we have been looking at watches and listeners as simple cause-and-effect pairs: When this changes, do that. It is not unusual, however, to want to watch *several* pieces of state and execute some code when *any one* of them changes.

Since Angular watches are just normal JavaScript functions, this is perfectly doable with the watch implementation we already have: Just craft a watch function that runs multiple checks and returns some combined value of them that causes the listener to fire.

As it happens, from Angular 1.3 onwards it is not necessary to craft these kinds of functions manually. Instead, you can use a built-in Scope feature called `$watchGroup`.

The `$watchGroup` function takes several watch functions wrapped in an array, and a single listener function. The idea is that when any of the watch functions given in the array detects a change, the listener function is invoked. The listener function is given the new and old values of the watches wrapped in arrays, in the order of the original watch functions.

Here's the first test case for this, wrapped in a new `describe` block:

test/scope_spec.js

```
describe('$watchGroup', function() {

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

  it('takes watches as an array and calls listener with arrays', function() {
    var gotNewValues, gotOldValues;

    scope.aValue = 1;
```



```
    scope.anotherValue = 2;

    scope.$watchGroup([
      function(scope) { return scope.aValue; },
      function(scope) { return scope.anotherValue; }
    ], function(newValues, oldValues, scope) {
      gotNewValues = newValues;
      gotOldValues = oldValues;
    });
    scope.$digest();

    expect(gotNewValues).toEqual([1, 2]);
    expect(gotOldValues).toEqual([1, 2]);
  });

});
```

In the test we grab the `newValues` and `oldValues` arguments received by the listener, and check that they are arrays containing the return values of the watch functions.

Let's take a first stab at implementing `$watchGroup`. We could try to just register each watch individually, reusing the listener for each one:

src/scope.js

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
    var self = this;
    _.forEach(watchFns, function(watchFn) {
        self.$watch(watchFn, listenerFn);
    });
};
```

This doesn't quite cut it though. We expect the listener to receive *arrays* of all the watch values, but now it just gets called with each watch value individually.

We'll need to define a separate internal listener function for each watch, and inside those internal listeners collect the values into arrays. We can then give those arrays to the original listener function. We'll use one array for the new values and another for the old values:

src/scope.js

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var newValues = new Array(watchFns.length);
  var oldValues = new Array(watchFns.length);
  _.forEach(watchFns, function(watchFn, i) {
    self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      listenerFn(newValues, oldValues, self);
    });
  });
};
```


`$watchGroup` always uses reference watches for change detection.

The problem with our first implementation is that it calls the listener a bit too eagerly: If there are several changes in the watch array, the listener will get called several times, and we'd like for it to get called just once. Even worse, since we're calling the listener immediately upon noticing a change, it's likely that we have a mixture of new and previous values in our `oldValues` and `newValues` arrays, causing the user to see an inconsistent combination of values.

Let's test that the listener is called just once even in the presence of multiple changes:

test/scope_spec.js

```
it('only calls listener once per digest', function() {
  var counter = 0;

  scope.aValue = 1;
  scope.anotherValue = 2;

  scope.$watchGroup([
    function(scope) { return scope.aValue; },
    function(scope) { return scope.anotherValue; }
  ], function(newValues, oldValues, scope) {
    counter++;
  });
  scope.$digest();

  expect(counter).toEqual(1);
});
```

How can we defer the listener call to a moment when all watches will have been checked? Since in `$watchGroup` we're not in charge of running the digest, there's no obvious place for us to put the listener call. But what we can do is use the `$evalAsync` function we implemented earlier in the chapter. Its purpose is to do some work later but still during the same digest - just the ticket for us!

We'll create a new internal function in `$watchGroup`, called `watchGroupListener`. This is the function that's in charge of calling the original listener with the two arrays. Then, in each individual listener we schedule a call to this function *unless one has been scheduled already*:

src/scope.js

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var oldValues = new Array(watchFns.length);
  var newValues = new Array(watchFns.length);
```



```
  var changeReactionScheduled = false;

  function watchGroupListener() {
    listenerFn(newValues, oldValues, self);
    changeReactionScheduled = false;
  }

  _.forEach(watchFns, function(watchFn, i) {
    self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      if (!changeReactionScheduled) {
        changeReactionScheduled = true;
        self.$evalAsync(watchGroupListener);
      }
    });
  });
};
```

That takes care of the basic behavior of `$watchGroup`, and we can turn our attention to a couple of special cases.

One issue is related to the requirement that when a listener is called for the very first time, both the new and old values should be the same. Now, our `$watchGroup` already does something like this, because it's built on the `$watch` function that implements this behavior. On the first invocation, the contents of the `newValues` and `oldValues` arrays will be exactly the same.

However, while the *contents* of those two arrays are the same, they are currently still two separate *array objects*. That breaks the contract of using the same exact value twice. It also means that if a user wants to compare the two values, they cannot use reference equality (`===`), but instead have to iterate the array contents and see if they match.

We want to do better, and have both the old and new values be the *same exact value* on the first invocation:

test/scope_spec.js

```
it('uses the same array of old and new values on first run', function() {
  var gotNewValues, gotOldValues;

  scope.aValue = 1;
  scope.anotherValue = 2;

  scope.$watchGroup([
    function(scope) { return scope.aValue; },
    function(scope) { return scope.anotherValue; }
  ], function(newValues, oldValues, scope) {
    gotNewValues = newValues;
    gotOldValues = oldValues;
  });
```



```
  scope.$digest();

  expect(gotNewValues).toBe(gotOldValues);
});
```

While doing this, let's also make sure we won't break what we already have, by adding a test that ensures we still get *different* arrays on subsequent listener invocations:

test/scope_spec.js

```
it('uses different arrays for old and new values on subsequent runs', function() {
  var gotNewValues, gotOldValues;

  scope.aValue = 1;
  scope.anotherValue = 2;

  scope.$watchGroup([
    function(scope) { return scope.aValue; },
    function(scope) { return scope.anotherValue; }
  ], function(newValues, oldValues, scope) {
    gotNewValues = newValues;
    gotOldValues = oldValues;
  });
  scope.$digest();

  scope.anotherValue = 3;
  scope.$digest();

  expect(gotNewValues).toEqual([1, 3]);
  expect(gotOldValues).toEqual([1, 2]);
});
```

We can implement this requirement by checking in the watch group listener whether it's being called for the first time. If it is, we'll just pass the `newValues` array to the original listener twice:

src/scope.js

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var oldValues = new Array(watchFns.length);
  var newValues = new Array(watchFns.length);
  var changeReactionScheduled = false;
  var firstRun = true;

  function watchGroupListener() {
    if (firstRun) {
      firstRun = false;
```

```
      listenerFn(newValues, newValues, self);
    } else {
      listenerFn(newValues, oldValues, self);
    }
    changeReactionScheduled = false;
  }

  _.forEach(watchFns, function(watchFn, i) {
    self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      if (!changeReactionScheduled) {
        changeReactionScheduled = true;
        self.$evalAsync(watchGroupListener);
      }
    });
  });
};
```

The other special case is a situation where the array of watches happens to be empty. It is not completely obvious what to do in this situation. Our current implementation does nothing - if there are no watches, no listeners will get fired. What Angular actually does though is make sure the listener gets called *exactly once*, with empty arrays as the values:

test/scope_spec.js

```
it('calls the listener once when the watch array is empty', function() {
  var gotNewValues, gotOldValues;

  scope.$watchGroup([], function(newValues, oldValues, scope) {
    gotNewValues = newValues;
    gotOldValues = oldValues;
  });
  scope.$digest();

  expect(gotNewValues).toEqual([]);
  expect(gotOldValues).toEqual([]);
});
```

What we'll do is check for the empty case in `$watchGroup`, schedule a call to the listener, and then return without bothering to do any further setup:

src/scope.js

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var oldValues = new Array(watchFns.length);
  var newValues = new Array(watchFns.length);
```



```
  var changeReactionScheduled = false;
  var firstRun = true;

  if (watchFns.length === 0) {
    self.$evalAsync(function() {
      listenerFn(newValues, newValues, self);
    });
    return;
  }

  function watchGroupListener() {
    if (firstRun) {
      firstRun = false;
      listenerFn(newValues, newValues, self);
    } else {
      listenerFn(newValues, oldValues, self);
    }
    changeReactionScheduled = false;
  }

  _.forEach(watchFns, function(watchFn, i) {
    self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      if (!changeReactionScheduled) {
        changeReactionScheduled = true;
        self.$evalAsync(watchGroupListener);
      }
    });
  });
};
```

The final feature we'll need for `$watchGroups` is deregistration. One should be able to deregister a watch group in exactly the same way as they deregister an individual watch: By using a removal function returned by `$watchGroup`.

test/scope_spec.js

```
it('can be deregistered', function() {
  var counter = 0;

  scope.aValue = 1;
  scope.anotherValue = 2;

  var destroyGroup = scope.$watchGroup([
    function(scope) { return scope.aValue; },
    function(scope) { return scope.anotherValue; }
  ], function(newValues, oldValues, scope) {
    counter++;
```



```
  });
  scope.$digest();

  scope.anotherValue = 3;
  destroyGroup();
  scope.$digest();

  expect(counter).toEqual(1);
});
```

Here we test that once the deregistration function has been called, further changes do not cause the listener to fire.

Since the individual watch registrations already return removal functions, all we really need to do is collect them, and then create a deregistration function that invokes all of them:

src/scope.js

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var oldValues = new Array(watchFns.length);
  var newValues = new Array(watchFns.length);
  var changeReactionScheduled = false;
  var firstRun = true;

  if (watchFns.length === 0) {
    self.$evalAsync(function() {
      listenerFn(newValues, newValues, self);
    });
    return;
  }

  function watchGroupListener() {
    if (firstRun) {
      firstRun = false;
      listenerFn(newValues, newValues, self);
    } else {
      listenerFn(newValues, oldValues, self);
    }
    changeReactionScheduled = false;
  }

  var destroyFunctions = _.map(watchFns, function(watchFn, i) {
    return self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      if (!changeReactionScheduled) {
        changeReactionScheduled = true;
        self.$evalAsync(watchGroupListener);
      }
```

```
    });
  });

  return function() {
    _.forEach(destroyFunctions, function(destroyFunction) {
      destroyFunction();
    });
  };
};
```

Since we have a special case for the situation where the watch array is empty, it needs its own watch deregistration function as well. The listener is only called once anyway in that situation, but one could still invoke the deregistration function before even the first digest occurs, in which case even that single call should be skipped:

test/scope_spec.js

```
it('does not call the zero-watch listener when deregistered first', function() {
  var counter = 0;

  var destroyGroup = scope.$watchGroup([], function(newValues, oldValues, scope) {
    counter++;
  });
  destroyGroup();
  scope.$digest();

  expect(counter).toEqual(0);
});
```

The deregistration function for this case just sets a boolean flag, which is checked before invoking the listener:

src/scope.js

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
    var self = this;
    var oldValues = new Array(watchFns.length);
    var newValues = new Array(watchFns.length);
    var changeReactionScheduled = false;
    var firstRun = true;

    if (watchFns.length === 0) {
        var shouldCall = true;
        self.$evalAsync(function() {
            if (shouldCall) {
                listenerFn(newValues, newValues, self);
            }
        });
```



```
        return function() {
            shouldCall = false;
        };
    }

    function watchGroupListener() {
        if (firstRun) {
            firstRun = false;
            listenerFn(newValues, newValues, self);
        } else {
            listenerFn(newValues, oldValues, self);
        }
        changeReactionScheduled = false;
    }

    var destroyFunctions = _.map(watchFns, function(watchFn, i) {
        return self.$watch(watchFn, function(newValue, oldValue) {
            newValues[i] = newValue;
            oldValues[i] = oldValue;
            if (!changeReactionScheduled) {
                changeReactionScheduled = true;
                self.$evalAsync(watchGroupListener);
            }
        });
    });

    return function() {
        _.forEach(destroyFunctions, function(destroyFunction) {
            destroyFunction();
        });
    };
};
```

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 `$evalAsync`, `$applyAsync`, and `$$postDigest`.
- Exception handling in the Angular digest.
- Destroying watches so they won't get executed again.
- Watching several things with a single effect using the `$watchGroup` function.

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("$watchGroup", 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 discussed 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!

> You could also use the ES5 shorthand function `Object.create(this)` to construct the child scope.

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.$$asyncQueue = [];
  this.$$applyAsyncQueue = [];
  this.$$applyAsyncId = null;
  this.$$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.$$digestOnce = function() {
  var dirty;
  var continueLoop = true;
  var self = this;
  this.$$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) {
            continueLoop = false;
            return false;
          }
        }
      } catch (e) {
        console.error(e);
      }
    });
    return continueLoop;
  });
  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.

The inner loop iterates over the scope hierarchy until all scopes have been visited or until the short-circuiting optimization kicks in. The optimization is tracked with the `continueLoop` variable. If it becomes `false`, we escape from both of the loops and the `$$digestOnce` function.

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, $evalAsync, and $applyAsync

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.$$asyncQueue = [];
  this.$$applyAsyncQueue = [];
  this.$$applyAsyncId = null;
  this.$$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 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` - and by association, `$applyAsync` - we have one more digest-triggering function to discuss, and that's `$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 our digest code 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.

We should refer to `$root.$$lastDirtyWatch` in `$watch`:

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.$$watchers.unshift(watcher);
  this.$root.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
      self.$root.$$lastDirtyWatch = null;
    }
  };
};
```

We should also do so in `$digest`:

src/scope.js

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

  if (this.$$applyAsyncId) {
    clearTimeout(this.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

  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);
    }
  }
};
```


And finally, we should do so in `$$digestOnce`:

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 directive scope linking.

Since we've now broken the prototypal inheritance chain, we need to revisit the discussions about `$digest`, `$apply`, `$evalAsync`, and `$applyAsync` 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 `$apply`, `$evalAsync`, and `$applyAsync` 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);
});
```

> Since `$applyAsync` is implemented in terms of `$apply`, it is suspect to the same problem, and will be fixed when we fix `$apply`.

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:

src/scope.js

```
Scope.prototype.$new = function(isolated) {
  var child;
  if (isolated) {
    child = new Scope();
    child.$root = this.$root;
  } 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 `$evalAsync`, `$applyAsync`, and `$$postDigest` functions. Recall that we drain the `$$asyncQueue` and the `$$postDigestQueue` in `$digest`, and `$$applyAsyncQueue` in `$$flushApplyAsync`. In neither do we take 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, `$$asyncQueue`, `$$applyAsyncQueue`, 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`, 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.$$asyncQueue = this.$$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;
};
```

For `$$applyAsyncQueue` the problem is a bit different: Since the flushing of the queue is controlled with the `$$applyAsyncId` attribute, and each scope in the hierachy may now have its *own* instance of this attribute, we have effectively several `$applyAsync` processes, one per isolated scope. This goes against the whole purpose of `$applyAsync`, which is to coalesce `$apply` invocations together.

First of all, we should share the queue between scopes, just like we did with the `$evalAsync` and `$postDigest` queues:

src/scope.js


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

Secondly, we need to share the `$$applyAsyncId` attribute. We cannot simply copy this in `$new` because we also need to be able to assign it. But what we can do is explicitly access it through `$root`:

src/scope.js

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

  if (this.$root.$$applyAsyncId) {
    clearTimeout(this.$root.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

  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);
    }
  }
};


Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
  if (self.$root.$$applyAsyncId === null) {
    self.$root.$$applyAsyncId = setTimeout(function() {
      self.$apply(_.bind(self.$$flushApplyAsync, self));
    }, 0);
  }
};

Scope.prototype.$$flushApplyAsync = function() {
  while (this.$$applyAsyncQueue.length) {
    try {
      this.$$applyAsyncQueue.shift()();
    } catch (e) {
      console.error(e);
    }
  }
  this.$root.$$applyAsyncId = null;
};
```

And finally we have everything set up properly!

Substituting The Parent Scope

In some situations it is useful to pass in some *other* scope as the parent of a new scope, while still maintaining the normal prototypical inheritance chain:

test/scope_spec.js

```
it('can take some other scope as the parent', function() {
  var prototypeParent = new Scope();
```



```
  var hierarchyParent = new Scope();
  var child = prototypeParent.$new(false, hierarchyParent);

  prototypeParent.a = 42;
  expect(child.a).toBe(42);

  child.counter = 0;
  child.$watch(function(scope) {
    scope.counter++;
  });

  prototypeParent.$digest();
  expect(child.counter).toBe(0);

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

Here we construct two "parent" scopes and then create the child. One of the parents is the normal, prototypical parent of the new scope. The other parent is the "hierarchical" parent, passed in as the second argument to `$new`.

We test that the prototypal inheritance between the prototype parent and the child works as usual, but we also test that launching a digest on the prototype parent does *not* cause a watch on the child to run. Instead, this happens when we launch a digest on the *hierarchical* parent.

We'll introduce an optional second argument to `$new`, which defaults to the current scope, `this`. We then use that scope's children to store the new child:

src/scope.js

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


> We also use `parent` to access the various queues in the isolate scope construction. Since they are all shared between all scopes anyway, it doesn't really matter whether we use `this` or `parent`, but we use the latter for clarity. The important part is the scope whose `$$children` we push the new scope into.

This feature introduces a subtle difference between the prototypal and hierarchical inheritance chains in your scope hierarchy. It is arguably of little value in most cases, and not worth the mental overhead of tracking two subtly different scope hierarchies. But, as we implement directive transclusion later in the book, we will see how it can sometimes be useful.

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 means that all of its watchers are removed and that the scope itself is removed from the `$$children` of its parent. Since the scope will no longer be referenced from anywhere, 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.)

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 (or substituted) parent will be assigned to the `$parent` attribute:

src/scope.js

```
Scope.prototype.$new = function(isolated, parent) {
  var child;
  parent = parent || this;
  if (isolated) {
    child = new Scope();
    child.$root = parent.$root;
    child.$$asyncQueue = parent.$$asyncQueue;
    child.$$postDigestQueue = parent.$$postDigestQueue;
    child.$$applyAsyncQueue = parent.$$applyAsyncQueue;
  } else {
    var ChildScope = function() { };
    ChildScope.prototype = this;
    child = new ChildScope();
  }
  parent.$$children.push(child);
  child.$$watchers = [];
  child.$$children = [];
  child.$parent = parent;
  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 find the current scope from its parent's `$$children` array and then remove it - as long as the scope is not the root scope and has a parent. It will also remove the watchers of the scope:

src/scope.js


```
Scope.prototype.$destroy = function() {
  if (this.$parent) {
    var siblings = this.$parent.$$children;
    var indexOfThis = siblings.indexOf(this);
    if (indexOfThis >= 0) {
      siblings.splice(indexOfThis, 1);
    }
  }
  this.$$watchers = null;
};
```

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

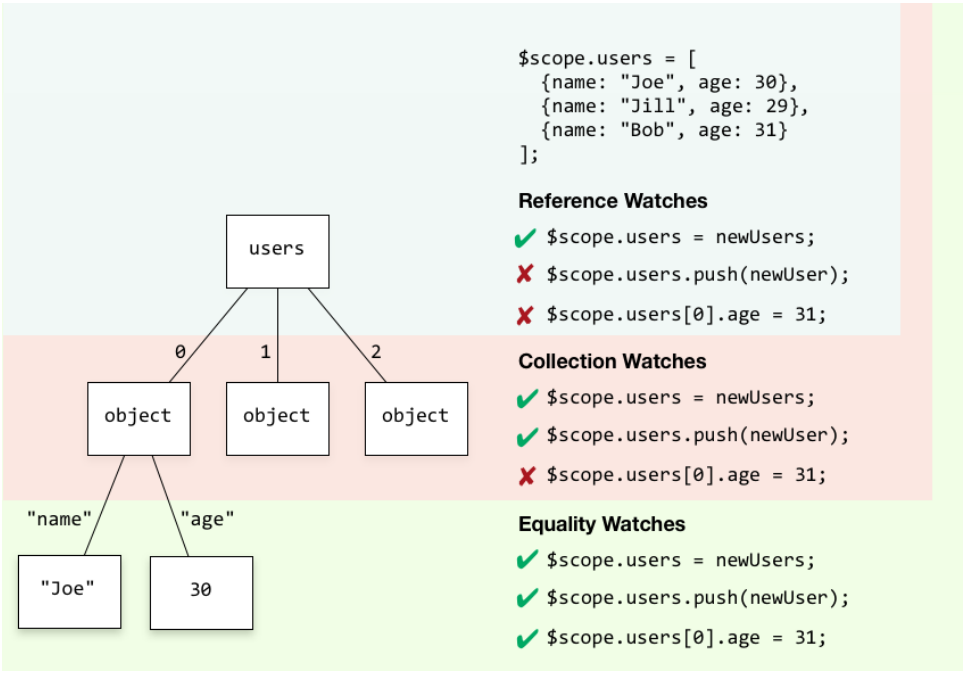

Figure 3.1: The three watch depths in Angular.js. Reference and equality/value-based watches were implemented in Chapter 1. Collection watches are the topic of this chapter.

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 calls 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.

In Chapter 1 we saw how `NaN`s can be problematic because `NaN` is not equal to `NaN`. We had to handle them specially in normal watches, and the following test case illustrates we also have to do so for collection watches:

test/scope_spec.js

```
it('does not fail on NaNs in arrays', function() {
  scope.arr = [2, NaN, 3];
  scope.counter = 0;

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

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

The test throws an exception because `NaN` is always triggering a change, causing an infinite digest. We can fix this by checking whether both old and the new value are `NaN`:

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) {
        var bothNaN = _.isNaN(newItem) && _.isNaN(oldValue[i]);
        if (!bothNaN && newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
        }
      });
    } else {

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

  return changeCount;
};
```

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) {
        var bothNaN = _.isNaN(newItem) && _.isNaN(oldValue[i]);
        if (!bothNaN && 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) {
        var bothNaN = _.isNaN(newItem) && _.isNaN(oldValue[i]);
        if (!bothNaN && 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) {
  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) {
        var bothNaN = _.isNaN(newItem) && _.isNaN(oldValue[i]);
        if (!bothNaN && newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
        }
      });
    } else {
      if (!_.isObject(oldValue) || _.isArrayLike(oldValue)) {
        changeCount++;
        oldValue = {};
      }
      _.forOwn(newValue, function(newVal, key) {
        if (oldValue[key] !== newVal) {
          changeCount++;
          oldValue[key] = newVal;
        }
      });
    }
  } 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 LoDash `_.forOwn` function iterates over an object's members, but only the ones defined for the object itself. Members inherited through the prototype chain are excluded. `$watchCollection` does not watch inherited properties in objects.

Just like with arrays, `NaN`s need special care here. When an object has an attribute with a `NaN` value, that causes an infinite digest:

test/scope_spec.js

```
it("does not fail on NaN attributes in objects", function() {
  scope.counter = 0;
  scope.obj = {a: NaN};

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

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

We need to check if both the old and new value are `NaN`, and if so, consider them to be the same value:

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) {
        var bothNaN = _.isNaN(newItem) && _.isNaN(oldValue[i]);
        if (!bothNaN && newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
        }
      });
    } else {
      if (!_.isObject(oldValue) || _.isArrayLike(oldValue)) {
        changeCount++;
        oldValue = {};
      }
```



```
      _.forOwn(newValue, function(newVal, key) {
        var bothNaN = _.isNaN(newVal) && _.isNaN(oldValue[key]);
        if (!bothNaN && oldValue[key] !== newVal) {
          changeCount++;
          oldValue[key] = newVal;
        }
      });
    }
  } else {
    if (!self.$$areEqual(newValue, oldValue, false)) {
      changeCount++;
    }
    oldValue = newValue;
  }

  return changeCount;
};
```

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) {
  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) {
        var bothNaN = _.isNaN(newItem) && _.isNaN(oldValue[i]);
        if (!bothNaN && newItem !== oldValue[i]) {
          changeCount++;
          oldValue[i] = newItem;
        }
      });
    } else {
      if (!_.isObject(oldValue) || _.isArrayLike(oldValue)) {
        changeCount++;
        oldValue = {};
      }
      _.forOwn(newValue, function(newVal, key) {
        var bothNaN = _.isNaN(newVal) && _.isNaN(oldValue[key]);
        if (!bothNaN && oldValue[key] !== newVal) {
          changeCount++;
          oldValue[key] = newVal;
        }
      });
      _.forOwn(oldValue, function(oldVal, key) {
        if (!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 `$watchCollection` 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;
    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) {
          var bothNaN = _.isNaN(newItem) && _.isNaN(oldValue[i]);
          if (!bothNaN && newItem !== oldValue[i]) {
            changeCount++;
            oldValue[i] = newItem;
          }
        });
      } else {
        if (!_.isObject(oldValue) || _.isArrayLike(oldValue)) {
          changeCount++;
          oldValue = {};
          oldLength = 0;
        }
        newLength = 0;
        _.forOwn(newValue, function(newVal, key) {
          newLength++;
          if (oldValue.hasOwnProperty(key)) {
            var bothNaN = _.isNaN(newVal) && _.isNaN(oldValue[key]);
            if (!bothNaN && oldValue[key] !== newVal) {
              changeCount++;
              oldValue[key] = newVal;
            }
          } else {
            changeCount++;
            oldLength++;
            oldValue[key] = newVal;
          }
        });
        if (oldLength > newLength) {
          changeCount++;
          _.forOwn(oldValue, function(oldVal, key) {
            if (!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 you

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 to 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.$$asyncQueue = [];
  this.$$applyAsyncQueue = [];
  this.$$applyAsyncId = null;
  this.$$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.$new = function(isolated, parent) {
  var child;
  parent = parent || this;
  if (isolated) {
    child = new Scope();
    child.$root = parent.$root;
    child.$$asyncQueue = parent.$$asyncQueue;
    child.$$postDigestQueue = parent.$$postDigestQueue;
    child.$$applyAsyncQueue = this.$$applyAsyncQueue;
  } else {
    var ChildScope = function() { };
    ChildScope.prototype = this;
    child = new ChildScope();
  }
  parent.$$children.push(child);
  child.$$watchers = [];
  child.$$listeners = {};
  child.$$children = [];
  child.$parent = parent;
  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. One way to look at it is that when the event is propagating up or down the scope hierarchy, `currentScope` points to where we *currently* are in the propagation.

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;
};
```

Since `currentScope` is meant to communicate the *current* status of the event propagation, it should also be cleared after the event propagation is done. Otherwise any code that holds on to the event after it has finished propagating will have stale information about the propagation status. We can test this by capturing the event in a listener, and seeing that after the event has been processed, `currentScope` is explicitly set to `null`:

test/scope_spec.js

```
it("sets currentScope to null after propagation on $emit", function() {
  var event;
  var scopeListener = function(evt) {
    event = evt;
  };
  scope.$on('someEvent', scopeListener);

  scope.$emit('someEvent');

  expect(event.currentScope).toBe(null);
});

it("sets currentScope to null after propagation on $broadcast", function() {
  var event;
  var scopeListener = function(evt) {
    event = evt;
  };
  scope.$on('someEvent', scopeListener);
```



```
  scope.$broadcast('someEvent');

  expect(event.currentScope).toBe(null);
});
```

This is achieved by simply setting `currentScope` to `null` at the end of `$emit`:

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);
  event.currentScope = null;
  return event;
};
```

The same is done in `$broadcast`:

src/scope.js

```
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;
  });
  event.currentScope = null;
  return event;
};
```

And 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.$destroy = function() {
  this.$broadcast('$destroy');
  if (this.$parent) {
    var siblings = this.$parent.$$children;
    var indexOfThis = siblings.indexOf(this);
    if (indexOfThis >= 0) {
      siblings.splice(indexOfThis, 1);
    }
  }
  this.$$watchers = null;
};
```

Disabling Listeners On Destroyed Scopes

In addition to firing the `$destroy` event, another effect of destroying a scope should be that its event listeners are no longer active:

test/scope_spec.js

```
it('no longers calls listeners after destroyed', function() {
  var listener = jasmine.createSpy();
  scope.$on('myEvent', listener);

  scope.$destroy();

  scope.$emit('myEvent');
  expect(listener).not.toHaveBeenCalled();
});
```

We can re-set the `$$listeners` object of the scope to an empty object, which will effectively throw all existing event listeners away:

src/scope.js


```
Scope.prototype.$destroy = function() {
  this.$broadcast('$destroy');
  if (this.$parent) {
    var siblings = this.$parent.$$children;
    var indexOfThis = siblings.indexOf(this);
    if (indexOfThis >= 0) {
      siblings.splice(indexOfThis, 1);
    }
  }
  this.$$watchers = null;
  this.$$listeners = {};
};
```

This leaves the `$$listeners` on any child scopes untouched, but since those Scopes are no longer part of the Scope hierarchy they won't be receiving any events. Unless there's a reference leak in application code, the child Scopes along with their listeners will soon be garbage collected.

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

We now have a full implementation of Scopes, which forms the core of Angular's change detection system. We'll now turn our attention to *expressions* - a feature that provides frictionless access to that core. Expressions 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 interpolation syntax `{{ }}`.

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 the magnitude of the problems associated with mixing markup and code.

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 features: The use of both `eval` and `with` is frowned upon, and `with` is actually forbidden in ECMAScript 5 strict mode.

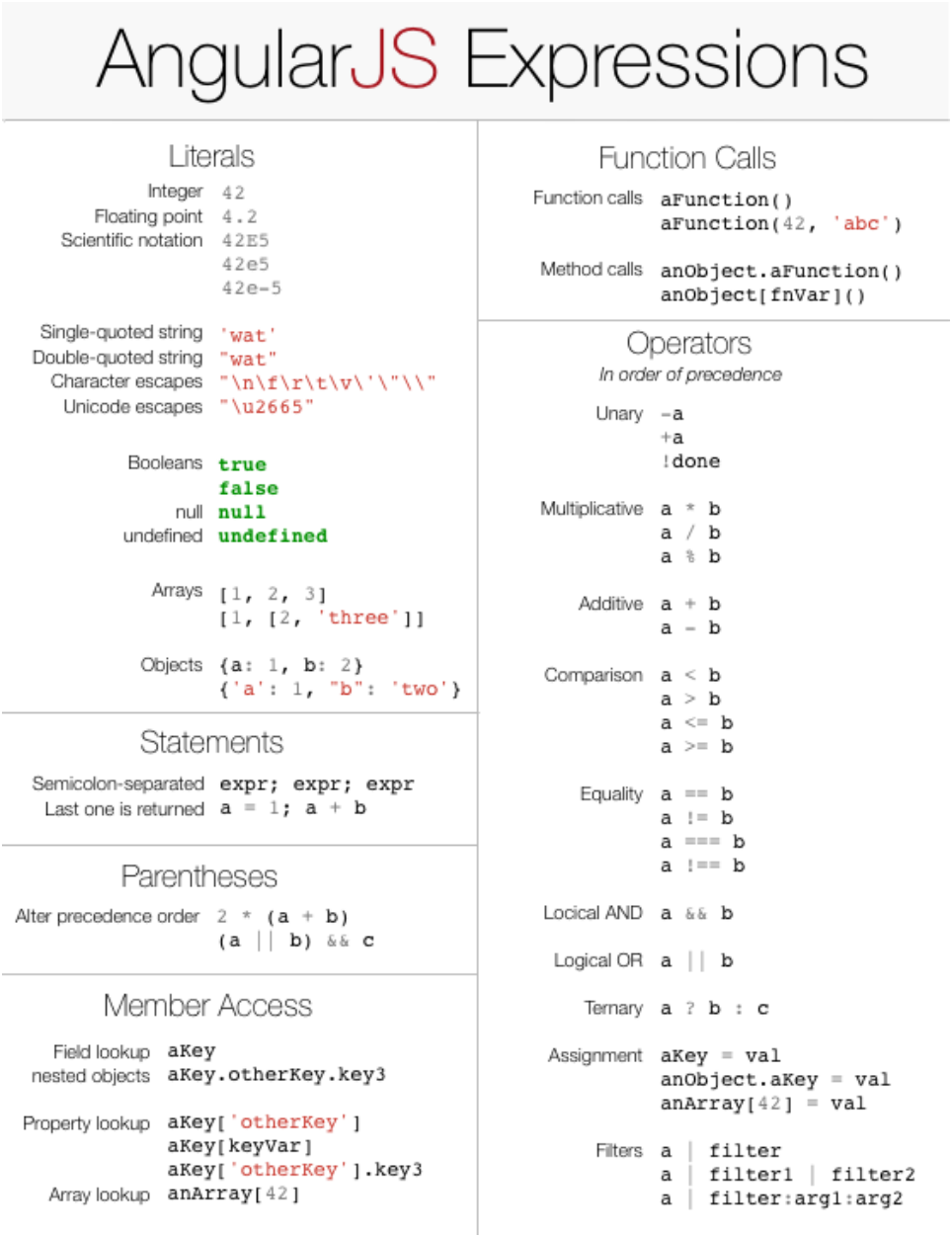

Figure 4.1: A cheatsheet of the full capabilities of the Angular expression language. PDF at http://teropa.info/blog/2014/03/23/angularjs-expressions-cheatsheet.html

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 that use the pipe character `|` are not valid JavaScript (where the pipe denotes a bitwise OR operation), 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, lexers, and abstract syntax trees - the tools of language designers.

What We Will Skip

There are a couple of things the Angular expression implementation includes that we will omit in the interest of staying closer to the essence of expressions:

- The Angular expression parser does a lot of work so that application programmers can get clear error messages when parsing goes wrong. This involves 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 switches to an *interpreted mode* from the default *compiled mode* when there is one present. We will only focus on the compiled mode in this book, which means our implementation would not work with a Content Security Policy.

Chapter 5

Literal Expressions

To begin our implementation of the Angular expression parser, let's make a version that can parse *literal* expressions - simple data expressions that represent themselves, like numbers, strings, an arrays (`"42"`, `[1, 2, 3]`), and *unlike* identifiers or function calls, (`fourtyTwo`, `a.b()`).

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 ...
}
```

> We will later turn this function into the `$parse` service, once we have the dependency injector up and running.

The file will contain four objects that do all the work of turning expression strings into functions: A *Lexer*, an *AST Builder*, an *AST Compiler*, and a *Parser*. They have distinct responsibilities in different phases of the job:

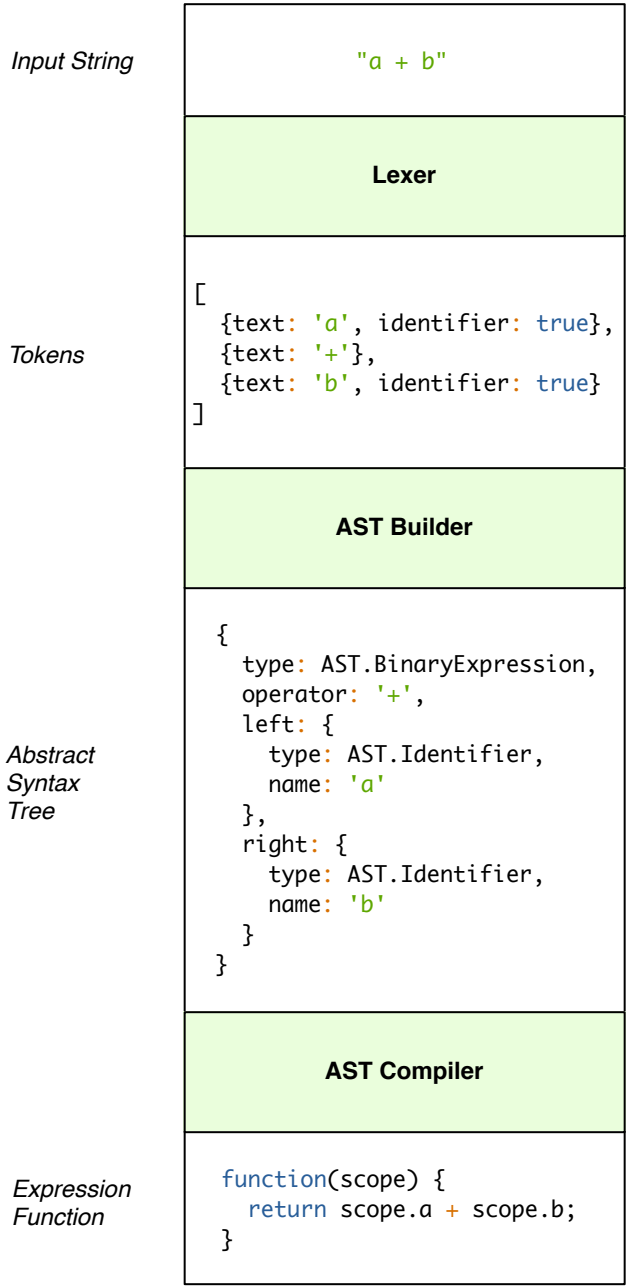

Figure 5.1: The expression parsing pipeline

- The *Lexer* takes the original expression string and returns an array of *tokens* parsed from that string. For example, the string `"a + b"` would result in tokens for `a`, `+`, and `b`.
- The *AST Builder* takes the array of tokens generated by the lexer, and builds up an *Abstract Syntax Tree* (AST) from them. The tree represents the syntactic structure of the expression as nested JavaScript objects. For example, the tokens `a`, `+`, and `b` would result in something like this:

```
{
  type: AST.BinaryExpression,
  operator: '+',
  left: {
    type: AST.Identifier,
    name: 'a'
  },
  right: {
    type: AST.Identifier,
    name: 'b'
  }
}
```

- The *AST Compiler* takes the abstract syntax tree and compiles it into a JavaScript function that evaluates the expression represented in the tree. For example, the AST above would result in something like this:

```
function(scope) {
  return scope.a + scope.b;
}
```

- The *Parser* is responsible for combining the low-level steps mentioned above. It doesn't do very much itself, but instead delegates the heavy lifting to the Lexer, the AST Builder, and the AST Compiler.

What this all means is that whenever you use expressions in Angular, JavaScript functions get generated behind the scenes. Those functions then get repeatedly executed when the expressions are evaluated during digests.

Let's set up the scaffolding for each of these. Firstly, the `Lexer` is defined as a constructor function. It includes a method called `lex`, which executes the tokenization:

src/parse.js

```
function Lexer() {
}

Lexer.prototype.lex = function(text) {
  // Tokenization will be done here
};
```

The AST Builder (denoted in the code just by `AST`) is another constructor function. It takes a Lexer as an argument. It also has an `ast` method, which will execute the AST building for the tokens of a given expression:

src/parse.js


```
function AST(lexer) {
  this.lexer = lexer;
}

AST.prototype.ast = function(text) {
  this.tokens = this.lexer.lex(text);
  // AST building will be done here
};
```

The AST Compiler is yet another constructor function, which takes an AST Builder as an argument. It has a method called `compile`, which compiles an expression into an expression function:

src/parse.js

```
function ASTCompiler(astBuilder) {
  this.astBuilder = astBuilder;
}

ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  // AST compilation will be done here
};
```

Finally, `Parser` is a constructor function that constructs the complete parsing pipeline from the pieces outlined above. It takes a Lexer as an agument, and has a method called `parse`:

src/parse.js

```
function Parser(lexer) {
  this.lexer = lexer;
  this.ast = new AST(this.lexer);
  this.astCompiler = new ASTCompiler(this.ast);
}

Parser.prototype.parse = function(text) {
  return this.astCompiler.compile(text);
};
```

We can now augment the public `parse` function so that it creates a Lexer and a Parser and then calls `Parser.parse`:

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 these functions with the features that make the magic happen.

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 AST Builder all the information it needs to construct an abstract syntax tree. At this point, we'll need just two things for our numeric literal, which are:

- The text that the token was parsed from
- The numeric value of the token

For the number 42, our token can simply be something like:

```
{
  text: '42',
  value: 42
}
```


So, let's implement number parsing in the Lexer in a way that gets us a data structure like the one above.

The `lex` function of the Lexer forms basically one big loop, which iterates over all characters in the given input 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 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++;
  }
  this.tokens.push({
    text: number,
    value: Number(number)
  });
};
```

Here we just add a new token to the `this.tokens` collection. The token's `text` attribute is the string we have read, and the `value` attribute is the numeric value converted from that string using the Number constructor.

The lexer is now doing its part for integer parsing. Next, we'll focus our attention on the AST Builder.

As discussed earlier, the AST is a nested JavaScript object structure that represents an expression in a tree-like form. Each node in the tree will have a `type` attribute that describes the syntactic structure the node represents. In addition to the type, nodes will have type-specific attributes that hold further information about the node.

For instance, our numeric literals will have a type of `AST.Literal`, and a `value` attribute that holds the value of the literal:

```
{type: AST.Literal, value: 42}
```

Every AST has a root node of type `AST.Program`. That root node has an attribute called `body` that holds the contents of the expression. Thus, our numeric literal should actually be wrapped inside an `AST.Program`:

```
{
  type: AST.Program,
  body: {
    type: AST.Literal,
    value: 42
  }
}
```

This is the AST we should now form from the Lexer output.

> There is actually an additional level of wrapping in ASTs that we're skipping for now. It has to do with expression having multiple *statements*. We will see how this works in a couple of chapters.

The top-level Program node is created by a method on the AST builder called `program`. It becomes the return value of the whole AST building process:

src/parse.js

```
AST.prototype.ast = function(text) {
  this.tokens = this.lexer.lex(text);
  return this.program();
};
AST.prototype.program = function() {
  return {type: AST.Program};
};
```

The value of the type, `AST.Program`, is a "marker constant" defined on the `AST` function. It is used to identify what type of node is being represented. Its value is a simple string:

src/parse.js

```
function AST(lexer) {
  this.lexer = lexer;
}
AST.Program = 'Program';
```

We will introduce similar marker constants for all AST node types, and use them in the AST compiler to make decisions about what kind of JavaScript code to generate.

The program should have a body, which in this case can just be a numeric literal value. Its `type` is `AST.Literal`, and it is generated by the `constant` method of the AST builder:

src/parse.js

```
AST.prototype.program = function() {
  return {type: AST.Program, body: this.constant()};
};
AST.prototype.constant = function() {
  return {type: AST.Literal, value: this.tokens[0].value};
};
```

For now we just grab the first token we have, and take its `value` attribute.

We need a marker constant for this type of node as well:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
```


This gives us the AST we need for numeric literals, and we can next focus our attention to the AST Compiler and its task of generating a JavaScript function from this AST.

What the AST Compiler will do is walk over the tree generated by the AST Builder, and build up the JavaScript source code that represents the nodes in the tree. It'll then generate a JavaScript function for the source code. For our numeric literal, that function will be very simple:

```
function() {
  return 42;
}
```

In the main `compile` function of the compiler, we'll introduce a `state` attribute into which we will collect information while walking the tree. For now we'll collect just one thing, which is the JavaScript code that forms the body of the function:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: []};
};
```

Once we've initialized the state, we'll start walking the tree, which we do with a method called `recurse`:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: []};
  this.recurse(ast);
};
ASTCompiler.prototype.recurse = function(ast) {

};
```

The goal is that once `recurse` returns, `state.body` will hold JavaScript statements that we can create a function from. That function will become our return value:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: []};
  this.recurse(ast);
  /* jshint -W054 */
  return new Function(this.state.body.join(''));
  /* jshint +W054 */
};
```


We're using the `Function` constructor to create the function. This constructor takes some JavaScript source code and compiles it into a function on the fly. 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.").

The final piece of the puzzle is to figure out what we should be doing in `recurse`. The expectation is to generate some JavaScript code and put it into `this.state.body`.

As the name implies, `recurse` is a recursive method that we will invoke for each node in the tree. Since each node has a `type`, and different types of nodes require different kind of processing, we'll introduce a `switch` statement with alternate branches for different AST node types:

src/parse.js

```
ASTCompiler.prototype.recurse = function(ast) {
  switch (ast.type) {
  case AST.Program:

  case AST.Literal:

  }
};
```

A literal is a "leaf node", which means it has no child nodes - just a value. What we can do for it is simply return the node's value:

src/parse.js

```
case AST.Literal:
  return ast.value;
```

For the Program node we need to do a bit more. We need to generate the `return` statement for the whole expression. What we should return is the value of the `body` of the Program, which we can obtain by recursively calling `recurse`:

src/parse.js

```
case AST.Program:
  this.state.body.push('return ', this.recurse(ast.body), ';');
  break;
```


In the case of our first test the body happens to be the value of the identifier `42`, resulting in the function body `return 42;`.

This makes our test pass. We have generated an expression function from the expression string `'42'`!

> You can actually inspect the generated source code by calling `fn.toString()` on the expression functions returned from `parse`. This can be a useful trick when debugging issues with more complicated expressions.

There are quite a few pieces at work here, which may seem unnecessarily complicated, but the roles of the different pieces will become more apparent as we add more features to our expression implementation.

> 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.

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 digit:

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++;
  }
  this.tokens.push({
```



```
    text: number,
    value: Number(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 upper case scientific notation", function() {
  var fn = parse('.42E2');
  expect(fn()).toBe(42);
});
```

Now that we have the specs 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++;
  }
  this.tokens.push({
    text: number,
    value: Number(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 digit, 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++;
  }
  this.tokens.push({
    text: number,
    value: Number(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++;
  }
  this.tokens.push({
    text: number,
    value: Number(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. One important difference is that before entering the `while` loop, we'll increment the character index 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({
        text: string,
        value: string
      });
      return;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

That's a good start for string parsing, but we're not done yet. Our tests are still failing because when this token ends up in the AST as a literal, its value is compiled as-is into the resulting JavaScript function. The expression `'abc'` results in a function like this:


```
function() {
  return abc;
}
```

The quotes around the string are missing, and the function is trying to look up a variable instead!

Our AST Compiler needs to be able to *escape* string values, so that they are properly quoted in the JavaScript. We'll use a method called `escape` for that:

src/parse.js

```
case AST.Literal:
  return this.escape(ast.value);
```

This method puts quotes around a value, but if and only if it is a string:

src/parse.js

```
ASTCompiler.prototype.escape = function(value) {
  if (_.isString(value)) {
    return '\'' + value + '\'';
  } else {
    return value;
  }
};
```

We're also being a bit too lenient about the start and end quotes of the input string by allowing a string to terminate with 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({
        text: string,
        value: string
      });
      return;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```

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 `\'`, and the double quote character `\"`.
- 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.

Let's consider single character escapes first. For instance, We should be able to parse strings that have quotes in them:

test/parse_spec.js

```
it('can parse a string with single quotes inside', function() {
  var fn = parse("'a\\\'b'");
  expect(fn()).toEqual('a\'b');
});

it('can parse a string with double quotes inside', function() {
  var fn = parse('"a\\\"b"');
  expect(fn()).toEqual('a\"b');
});
```

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 next characters differently:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var string = '';
  var escape = false;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    if (escape) {

    } else if (ch === quote) {
      this.index++;
      this.tokens.push({
        text: string,
        value: 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. This includes the quote characters we wrote unit tests for:

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 string = '';
  var escape = false;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    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: string,
        value: string
      });
      return;
    } else if (ch === '\\') {
      escape = true;
    } else {
      string += ch;
    }
    this.index++;
  }
  throw 'Unmatched quote';
};
```


There's still a problem when we get to the AST compilation phase, though. When the AST compiler encounters characters like ' and " in literals, it just puts them in the result, which results in invalid JavaScript code. The `escape` method of the compiler should be able to handle these characters. What we'll do is a regex replacement during the escaping:

src/parse.js

```
ASTCompiler.prototype.escape = function(value) {
  if (_.isString(value)) {
    return '\'' +
      value.replace(this.stringEscapeRegex, this.stringEscapeFn) +
      '\'';
  } else {
    return value;
  }
};
```

What we match for escaping are any characters other than a space or an alphanumeric character:

src/parse.js

```
ASTCompiler.prototype.stringEscapeRegex = /[^ a-zA-Z0-9]/g;
```

In the replacement function, we get the numeric unicode value of the character we're escaping (using charCodeAt), and convert it into the corresponding hexadecimal (base 16) unicode escape sequence that we can safely concatenate into the generated JavaScript code:

src/parse.js

```
ASTCompiler.prototype.stringEscapeFn = function(c) {
  return '\\u' + ('0000' + c.charCodeAt(0).toString(16)).slice(-4);
};
```

Finally, let's consider unicode escape sequences in the input expressions themselves:

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 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 string = '';
  var escape = false;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    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: string,
        value: 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 characters here, as unicode escape sequences are case-insensitive:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var string = '';
  var escape = false;
  while (this.index < this.text.length) {
    var ch = this.text.charAt(this.index);
    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: string,
        value: 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, meaning that 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);
});
```

We can identify an identifier in the Lexer by seeing if we have 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 identifier tokens, but they aren't yet being turned to anything useful by the AST builder. We should change that, and we can do it by making the AST aware that certain kinds of "constant" tokens represent predefined literals:

src/parse.js

```
AST.prototype.constants = {
  'null': {type: AST.Literal, value: null},
  'true': {type: AST.Literal, value: true},
  'false': {type: AST.Literal, value: false}
};
```


To plug these into the AST, let's introduce an additional intermediate function between `program`, and `constant`, called `primary`. That is, a program's body consists of a "primary" token:

src/parse.js

```
AST.prototype.program = function() {
  return {type: AST.Program, body: this.primary()};
};
AST.prototype.primary = function() {
  return this.constant();
};
AST.prototype.constant = function() {
  return {type: AST.Literal, value: this.tokens[0].value};
};
```

A primary token can be one of our predefined constants, or some other constant from our previous implementation.

src/parse.js

```
AST.prototype.primary = function() {
  if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    return this.constants[this.tokens[0].text];
  } else {
    return this.constant();
  }
};
```

The test cases for `true` and `false` are now passing, as they end up in the compiled JavaScript as-is. For `null` this isn't happening yet, because its default string representation is an empty string. We need a special case for it in the compiler's `escape` method so that the text `null` appears in the compiled code:

src/parse.js

```
ASTCompiler.prototype.escape = function(value) {
  if (_.isString(value)) {
    return '\'' +
      value.replace(this.stringEscapeRegex, this.stringEscapeFn) +
      '\'';
  } else if (_.isNull(value)) {
    return 'null';
  } else {
    return 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 multi-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
      });
      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;
};
```

In the AST builder we now have to consider a situation where the Lexer output doesn't just consist of a single token. We now have two tokens - `[` and `]` - that should cause an array node to be included in the AST.

Arrays are primary expressions, just like constants, so the handling of arrays is going to be in `AST.primary`. A primary expression may begin with an opening square bracket, in which case we handle it as an array declaration:

src/parse.js

```
AST.prototype.primary = function() {
  if (this.expect('[')) {
    return this.arrayDeclaration();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    return this.constants[this.tokens[0].text];
  } else {
    return this.constant();
  }
};
```

The `expect` function used here is something we don't have yet. It has to do with the fact that we are now working with multiple tokens. `expect` checks if the next token is what we *expect* it to be, and returns it if it is. It also removes that token from `this.tokens`, so that we "move forward" to the next token:

src/parse.js

```
AST.prototype.expect = function(e) {
  if (this.tokens.length > 0) {
    if (this.tokens[0].text === e || !e) {
      return this.tokens.shift();
    }
  }
};
```

Note that `expect` can also be called with no arguments, in which case it'll process whatever token is next.

The `arrayDeclaration` function is also new. This is where we will consume the tokens related to an array and construct the array AST node. 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

```
AST.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

```
AST.prototype.consume = function(e) {
  var token = this.expect(e);
  if (!token) {
    throw 'Unexpected. Expecting: ' + e;
  }
  return token;
};
```

If no exception is thrown, we have a valid (empty) array, and we can return the corresponding AST node. It has its own type of `ArrayExpression`:

src/parse.js

```
AST.prototype.arrayDeclaration = function() {
  this.consume(']');
  return {type: AST.ArrayExpression};
};
```

We also need to introduce this type:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
```


This new type now needs to be handled by the AST compiler. For now we can actually just emit `'[]'` for it to make our test pass:

src/parse.js

```
ASTCompiler.prototype.recurse = function(ast) {
  switch (ast.type) {
  case AST.Program:
    this.state.body.push('return ', this.recurse(ast.body), ';');
    break;
  case AST.Literal:
    return this.escape(ast.value);
  case AST.ArrayExpression:
    return '[]';
  }
};
```

So that is how a basic, empty array is produced: The Lexer emits its opening and closing square brackets as tokens, `AST.primary` notices the opening square bracket and delegates to `AST.arrayDeclaration`, which consumes the closing square bracket and emits an `ArrayExpression` AST node. The AST compiler then emits an empty JavaScript array expression.

Next, let's consider *non-empty* arrays. When array literals have elements in them, they are separated by commas. The elements may be anything, including other arrays:

test/parse_spec.js

```
it("will parse a non-empty array", function() {
  var fn = parse('[1, "two", [3], true]');
  expect(fn()).toEqual([1, 'two', [3], true]);
});
```

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
      });
      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 the AST `arrayDeclaration`. It should check whether the array is immediately closed as an empty array or not. If not, there's element processing to be done. The element nodes are collected to a local array called `elements`:

src/parse.js

```
AST.prototype.arrayDeclaration = function() {
  var elements = [];
  if (!this.peek(']')) {

  }
  this.consume(']');
  return {type: AST.ArrayExpression};
};
```

We need to define the `peek` function used above before getting into collecting the elements. It is basically the same as `expect`, but does *not* consume the token it looks at:

src/parse.js

```
AST.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

```
AST.prototype.expect = function(e) {
  var token = this.peek(e);
  if (token) {
    return this.tokens.shift();
  }
};
```

If there are elements in an array, we'll consume them in a loop. The loop terminates when we no longer see a comma following the last element. Each element is recursively processed as *another* primary node:

src/parse.js

```
AST.prototype.arrayDeclaration = function() {
  var elements = [];
  if (!this.peek(']')) {
    do {
      elements.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(']');
  return {type: AST.ArrayExpression};
};
```

Before this'll work, there's an important change we have to make to the `primary` and `constant` methods of AST that we implemented earlier. They can no longer assume there's just one token, and they need to *consume* the current token so that we're able to move forward to the next one.

In `primary` we should consume a constant value's token:

src/parse.js

```
AST.prototype.primary = function() {
  if (this.expect('[')) {
    return this.arrayDeclaration();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    return this.constants[this.consume().text];
  } else {
    return this.constant();
  }
};
```


And we should do the same in `constant`:

src/parse.js

```
AST.prototype.constant = function() {
  return {type: AST.Literal, value: this.consume().value};
};
```

The last thing the AST builder should do is attach those collected elements into the `ArraExpression` node so that we'll be able to access them in the compiler:

src/parse.js

```
AST.prototype.arrayDeclaration = function() {
  var elements = [];
  if (!this.peek(']')) {
    do {
      elements.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(']');
  return {type: AST.ArrayExpression, elements: elements};
};
```

Moving to the compiler then, what we need to do is recurse into each of the elements of an array, and collect the resulting JavaScript expressions:

src/parse.js

```
case AST.ArrayExpression:
  var elements = _.map(ast.elements, function(element) {
    return this.recurse(element);
  }, this);
  return '[]';
```

We can then emit them as the contents of the array JavaScript expression:

src/parse.js

```
case AST.ArrayExpression:
  var elements = _.map(ast.elements, function(element) {
    return this.recurse(element);
  }, this);
  return '[' + elements.join(',') + ']';
```


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 in the AST builder so that it's prepared for a situation where it doesn't have an element expression to process. If it sees the closing square bracket, it should break out early:

src/parse.js

```
AST.prototype.arrayDeclaration = function() {
  var elements = [];
  if (!this.peek(']')) {
    do {
      if (this.peek(']')) {
        break;
      }
      elements.push(this.primary());
    } while (this.expect(','));
  }
  this.consume(']');
  return {type: AST.ArrayExpression, elements: elements};
};
```

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
      });
      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
      });
      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. `AST.primary` looks out for opening curly braces and when one is seen, delegates to a new method called `object`:

src/parse.js

```
AST.prototype.primary = function() {
  if (this.expect('[')) {
    return this.arrayDeclaration();
  } else if (this.expect('{')) {
    return this.object();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    return this.constants[this.consume().text];
  } else {
    return this.constant();
  }
};
```


The `object` method's structure is basically the same as that of `arrayDeclaration`. It consumes the object, including the closing curly brace, and returns an AST node of type `ObjectExpression`:

src/parse.js

```
AST.prototype.object = function() {
  this.consume('}');
  return {type: AST.ObjectExpression};
};
```

Again, the type needs to be defined as well:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
```

The AST compiler's responsibility is to now emit an object literal when it sees an `ObjectExpression` node in `recurse`:

src/parse.js

```
case AST.ObjectExpression:
  return '{}';
```

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 string keys:

test/parse_spec.js

```
it("will parse a non-empty object", function() {
  var fn = parse('{"a key": 1, \'another-key\': 2}');
  expect(fn()).toEqual({'a key': 1, 'another-key': 2});
});
```

Just like in array AST building, in object AST building we have a `do..while` loop that consumes the keys and values separated by commas:

src/parse.js


```
AST.prototype.object = function() {
  if (!this.peek('}')) {
    do {

    } while (this.expect(','));
  }
  this.consume('}');
  return {type: AST.ObjectExpression};
};
```

Inside the loop body, we will first read in the key by consuming a constant token. From it we form another AST node of type `Property`:

src/parse.js

```
AST.prototype.object = function() {
  if (!this.peek('}')) {
    do {
      var property = {type: AST.Property};
      property.key = this.constant();

    } while (this.expect(','));
  }
  this.consume('}');
  return {type: AST.ObjectExpression};
};
```

The type needs to be declared as well:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
```

Then we'll consume the colon character that should separate the key and a value:

src/parse.js

```
AST.prototype.object = function() {
  if (!this.peek('}')) {
    do {
      var property = {type: AST.Property};
      property.key = this.constant();
      this.consume(':');

    } while (this.expect(','));
  }
  this.consume('}');
  return {type: AST.ObjectExpression};
};
```


Finally we'll consume the value, which is a whole other primary AST node that we attach onto the property:

src/parse.js

```
AST.prototype.object = function() {
  if (!this.peek('}')) {
    do {
      var property = {type: AST.Property};
      property.key = this.constant();
      this.consume(':');
      property.value = this.primary();

    } while (this.expect(','));
  }
  this.consume('}');
  return {type: AST.ObjectExpression};
};
```

We then collect these properties from the loop and attach them to the `ObjectExpression` node:

src/parse.js

```
AST.prototype.object = function() {
  var properties = [];
  if (!this.peek('}')) {
    do {
      var property = {type: AST.Property};
      property.key = this.constant();
      this.consume(':');
      property.value = this.primary();
      properties.push(property);
    } while (this.expect(','));
  }
  this.consume('}');
  return {type: AST.ObjectExpression, properties: properties};
};
```

During compilation we'll now generate the JavaScript for each property, and put it inside the generated object literal:

src/parse.js


```
case AST.ObjectExpression:
  var properties = _.map(ast.properties, function(property) {

  }, this);
  return '{' + properties.join(',') + '}';
```

The key is a `Constant` node, which has a `value` attribute that we should also escape, since it is a string:

src/parse.js

```
case AST.ObjectExpression:
  var properties = _.map(ast.properties, function(property) {
    var key = this.escape(property.key.value);

  }, this);
  return '{' + properties.join(',') + '}';
```

The value is any expression, whose value we can get using `recurse`. The combined value of the property consists of the key and value separated by a colon character:

src/parse.js

```
case AST.ObjectExpression:
  var properties = _.map(ast.properties, function(property) {
    var key = this.escape(property.key.value);
    var value = this.recurse(property.value);
    return key + ':' + value;
  }, this);
  return '{' + properties.join(',') + '}';
```

An object's keys are not always strings. They may also be identifiers where the quotes are omitted:

test/parse_spec.js

```
it("will parse an object with identifier keys", function() {
  var fn = parse('{a: 1, b: [2, 3], c: {d: 4}}');
  expect(fn()).toEqual({a: 1, b: [2, 3], c: {d: 4}});
});
```

The test fails because what the AST consumes at the position of the object keys are *identifier* tokens generated by `readIdent`, while it expects them to be constants instead. Let's first tweak `readIdent` a bit so that the fact that these are identifier tokens is marked:

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,
    identifier: true
  };

  this.tokens.push(token);
};
```

The AST builder should check this flag and form an *identifier* node instead of a constant one when it is `true`:

src/parse.js

```
AST.prototype.object = function() {
  var properties = [];
  if (!this.peek('}')) {
    do {
      var property = {type: AST.Property};
      if (this.peek().identifier) {
        property.key = this.identifier();
      } else {
        property.key = this.constant();
      }
      this.consume(':');
      property.value = this.primary();
      properties.push(property);
    } while (this.expect(','));
  }
  this.consume('}');
  return {type: AST.ObjectExpression, properties: properties};
};
```

Identifiers are a new kind of AST node, of type `Identifier`. They have a `name` attribute formed from the text of the identifier token:

src/parse.js

```
AST.prototype.identifier = function() {
  return {type: AST.Identifier, name: this.consume().text};
};
```

We need to introduce the `Identifier` type:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
```

We will later use identifier nodes elsewhere in the AST as well, but for now they only exist as object keys.

In the AST compiler we should now check whether the key is an `Identifier`. This affects which attribute we use on it to access the actual key to generate:

src/parse.js

```
case AST.ObjectExpression:
  var properties = _.map(ast.properties, function(property) {
    var key = property.key.type === AST.Identifier ?
      property.key.name :
      this.escape(property.key.value);
    var value = this.recurse(property.value);
    return key + ':' + value;
  }, this);
  return '{' + properties.join(',') + '}';
```

And finally we're able to parse all the literals that the Angular expression language supports!

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 three phases: Lexing, AST building, and AST compilation.
- That the end result of the parsing process is a generated JavaScript function.
- How the parser deals with integers, floating point numbers, and scientific notation.
- 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.

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();
});
```

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. In fact, the first change we should make is to add this argument to the generated, compiled expression function. We'll call it `s`:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: []};
  this.recurse(ast);
  /* jshint -W054 */
  return new Function('s', this.state.body.join(''));
  /* jshint +W054 */
};
```

When parsed, the expression `'aKey'` turns into an identifier token and then an `Identifier` AST node. We have already used identifier nodes as object keys. Now we will extend our identifier support to include attribute lookup.

In the AST builder, the change we have to make is to check for identifiers when we're building a primary AST node. We'll do it in the same way we did when building object property nodes: If we're looking at an identifier token, we'll make an `Identifier` node.

src/parse.js

```
AST.prototype.primary = function() {
  if (this.expect('[')) {
    return this.arrayDeclaration();
  } else if (this.expect('{')) {
    return this.object();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    return this.constants[this.consume().text];
  } else if (this.peek().identifier) {
    return this.identifier();
  } else {
    return this.constant();
  }
};
```

Similarly, the AST compiler now needs to be able to handle an identifier in its `recurse` method. When one is seen, it should be handled as a member attribute lookup from the scope - or `s`.

src/parse.js

```
case AST.Identifier:
  return this.nonComputedMember('s', ast.name);
```

The `nonComputedMember` method takes two arguments: An object to do a lookup from, and the member attribute to look up. It generates the JavaScript for a non-computed `a.b` member lookup (as opposed to a *computed* `a[b]` lookup):

src/parse.js


```
ASTCompiler.prototype.nonComputedMember = function(left, right) {
  return '(' + left + ').' + right;
};
```

And that passes our test: We're generating the function body `return s.aKey` for the expression `'aKey'`. That instantly makes the expression language much more useful than before, 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. That means, for example, that when we evaluate the expression function without any arguments, causing `s` to be `undefined`, no exceptions should be thrown:

test/parse_spec.js

```
it('returns undefined when looking up attribute from undefined', function() {
  var fn = parse('aKey');
  expect(fn()).toBeUndefined();
});
```

What this means that we'll need to generate a conditional to the JavaScript code, which first checks if `s` actually exists before trying to do a lookup from it. Essentially, we'll want to generate something like this for `aKey`:

```
function (s) {
  var v0;
  if (s) {
    v0 = s.aKey;
  }
  return v0;
}
```

For the if statement, let's introduce a method called `if_` that takes two arguments: A test expression and the "consequent" statement to execute when the expression is true. It generates the corresponding JavaScript `if` statement and appends it to the expression body:

src/parse.js

```
ASTCompiler.prototype.if_ = function(test, consequent) {
  this.state.body.push('if(', test, '){', consequent, '}');
};
```

We'll use it to check for the existence of `s` in the identifier lookup:

src/parse.js


```
case AST.Identifier:
  this.if_('s', '');
  return this.nonComputedMember('s', ast.name);
```

What we should do next is introduce a variable before the `if` block, populate it with the scope attribute inside the if block, and then return the variable name from this `recurse` call:

src/parse.js

```
case AST.Identifier:
  this.state.body.push('var v0;');
  this.if_('s', 'v0=' + this.nonComputedMember('s', ast.name) + ';');
  return 'v0';
```

Our tests now pass, but before we move forward, let's spend a moment refactoring this code to make it easier to extend later. For instance, we can introduce another helper function for variable assignment:

src/parse.js

```
ASTCompiler.prototype.assign = function(id, value) {
  return id + '=' + value + ';';
};
```

We'll use it inside the `if` statement:

src/parse.js

```
case AST.Identifier:
  this.state.body.push('var v0;');
  this.if_('s', this.assign('v0', this.nonComputedMember('s', ast.name)));
  return 'v0';
```

Also, many expressions are going to need several variables, and it's going to be difficult to figure out what names to generate for them without having them clash. For this purpose we'll introduce a running number to the state of the compiler, which forms a basis for generating unique ids. We'll initialize it to zero:

src/parse.js


```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0};
  this.recurse(ast);
  /* jshint -W054 */
  return new Function('s', this.state.body.join(''));
  /* jshint +W054 */
};
```

Then we'll make a method, also called `nextId`, that generates a variable name and increments the counter:

src/parse.js

```
ASTCompiler.prototype.nextId = function() {
  var id = 'v' + (this.state.nextId++);
  return id;
};
```

We can then use this function in identifier lookup:

src/parse.js

```
case AST.Identifier:
  var intoId = this.nextId();
  this.state.body.push('var ', intoId, ';');
  this.if_('s', this.assign(intoId, this.nonComputedMember('s', ast.name)));
  return intoId;
```

Finally, the `var` declaration shouldn't really be a part of the identifier lookup. In JavaScript, variable declarations are hoisted to the top of the function, and it would be better if we did that explicitly as well. So, we can introduce a `vars` array in the compiler state:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0, vars: []};
  this.recurse(ast);
  /* jshint -W054 */
  return new Function('s', this.state.body.join(''));
  /* jshint +W054 */
};
```

Whenever `nextId` is called, the generated variable name is added to the state:

src/parse.js


```
ASTCompiler.prototype.nextId = function() {
  var id = 'v' + (this.state.nextId++);
  this.state.vars.push(id);
  return id;
};
```

Then, a `var` declaration for all the generated vars is added to the top of the generated JavaScript:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0, vars: []};
  this.recurse(ast);
  /* jshint -W054 */
  return new Function('s',
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      '') +
    this.state.body.join(''));
  /* jshint +W054 */
};
```

Now we no longer need the `var` statement in the identifier branch:

src/parse.js

```
case AST.Identifier:
  var intoId = this.nextId();
  this.if_('s', this.assign(intoId, this.nonComputedMember('s', ast.name)));
  return intoId;
```

And there we have a general facility for introducing any number of variables inside the expression function. This'll come in very handy in the near future.

Parsing `this`

One special kind of attribute lookup we need to handle is a reference to `this`. The role of `this` in Angular expressions is similar to its role in JavaScript: It is the context in which the expression is being evaluated. The context of an expression function is always the scope it's being evaluated on:

test/parse_spec.js


```
it('will parse this', function() {
  var fn = parse('this');
  var scope = {};
  expect(fn(scope)).toBe(scope);
  expect(fn()).toBeUndefined();
});
```

From the Lexer, `this` comes out as an identifier, and in the AST builder we'll now add a special kind of AST node for it into the `constants` lookup object:

src/parse.js

```
AST.prototype.constants = {
  'null': {type: AST.Literal, value: null},
  'true': {type: AST.Literal, value: true},
  'false': {type: AST.Literal, value: false},
  'this': {type: AST.ThisExpression}
};
```

We haven't introduced `AST.ThisExpression` yet, so we need to do that:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
```

In the AST Compiler's `recurse` method we can compile an `AST.ThisExpression` node to a simple reference to `s` - the scope given to the expression function:

src/parse.js

```
case AST.ThisExpression:
  return 's';
```

Non-Computed Attribute Lookup

In addition to just referencing a Scope attribute, you can go deeper in the same expression and look up something in a nested data structure using the dot operator:

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.

More generally, an attribute lookup doesn't have to be preceded by an identifier. It might as well be some other expression, like an object literal:

test/parse_spec.js

```
it('looks up a member from an object', function() {
  var fn = parse('{aKey: 42}.aKey');
  expect(fn()).toBe(42);
});
```

We don't currently emit dot tokens from the Lexer, so before doing anything else we need to change that:

src/parse.js

```
} else if (this.is('[],{}:.')) {
  this.tokens.push({
    text: this.ch
  });
  this.index++;
```

On the AST building side of things, non-computed attribute lookup using the dot operator is considered a primary node, and handled in the `primary` method. After processing an initial primary node, we should check if it is followed by a dot token:

src/parse.js


```
AST.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    primary = this.constants[this.consume().text];
  } else if (this.peek().identifier) {
    primary = this.identifier();
  } else {
    primary = this.constant();
  }
  if (this.expect('.')) {

  }
  return primary;
};
```

If a dot is found, this primary expression becomes a `MemberExpression` node. We reuse the initial primary node as the *object* of the member expression, and expect to have an identifier after the dot, which we'll use as the property name to look up:

src/parse.js

```
AST.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    primary = this.constants[this.consume().text];
  } else if (this.peek().identifier) {
    primary = this.identifier();
  } else {
    primary = this.constant();
  }
  if (this.expect('.')) {
    primary = {
      type: AST.MemberExpression,
      object: primary,
      property: this.identifier()
    };
  }
  return primary;
};
```

The `MemberExpression` type needs to be introduced:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
AST.MemberExpression = 'MemberExpression';
```

In the AST compiler we already have all the parts we need. What we should generate is a non-computed member lookup, where the left side is the `object` of the AST node, and the right side is the `property` of the node.

Just like we did in the simple lookup, we need to guard the lookup with an `if` statement, because the left side might not exist. We'll also reuse the `intoId` variable name here so let's pull its declaration to the top of the `recurse` function while we're adding the `MemberExpression` case:

src/parse.js

```
ASTCompiler.prototype.recurse = function(ast) {
  var intoId;
  switch (ast.type) {
  case AST.Program:
    this.state.body.push('return ', this.recurse(ast.body), ';');
    break;
  case AST.Literal:
    return this.escape(ast.value);
  case AST.ArrayExpression:
    var elements = _.map(ast.elements, function(element) {
      return this.recurse(element);
    }, this);
    return '[' + elements.join(',') + ']';
  case AST.ObjectExpression:
    var properties = _.map(ast.properties, function(property) {
      var key = property.key.type === AST.Identifier ?
        property.key.name :
        this.escape(property.key.value);
      var value = this.recurse(property.value);
      return key + ':' + value;
    }, this);
    return '{' + properties.join(',') + '}';
  case AST.Identifier:
    intoId = this.nextId();
    this.if_('s', this.assign(intoId, this.nonComputedMember('s', ast.name)));
    return intoId;
```



```
  case AST.ThisExpression:
    return 's';
  case AST.MemberExpression:
    intoId = this.nextId();
    var left = this.recurse(ast.object);
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
    return intoId;
  }
};
```

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();
});
```

However many lookups there are, they're all part of the same primary expression. The trick is to turn the `if` statement in `AST.primary` into a `while` statement. As long as there are dots in the expression, new member lookups are generated. The previous member lookup always becomes the object of the next lookup:

src/parse.js

```
AST.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    primary = this.constants[this.consume().text];
  } else if (this.peek().identifier) {
    primary = this.identifier();
  } else {
    primary = this.constant();
  }
  while (this.expect('.')) {
    primary = {
      type: AST.MemberExpression,
      object: primary,
```



```
      property: this.identifier()
    };
  }
  return primary;
};
```

Before moving forward, let's take a moment to inspect what happens here, since the recursive code in the AST builder and AST compiler is already surprisingly powerful. The abstract syntax tree that gets built up for `aKey.secondKey.thirdKey.fourthKey` looks like this:

```
{
  type: AST.Program,
  body: {
    type: AST.MemberExpression,
    property: {type: AST.Identifier, name: 'fourthKey'},
    object: {
      type: AST.MemberExpresion,
      property: {type: AST.Identifier, name: 'thirdKey'},
      object: {
        type: AST.MemberExpression,
        property: {type: AST.Identifier, name: 'secondKey'},
        object: {type: AST.Identifier, name: 'aKey'}
      }
    }
  }
}
```

In the AST compiler this turns into the following JavaScript function:

```
function(s) {
  var v0, v1, v2, v3;
  if (s) {
    v3 = s.aKey;
  }
  if (v3) {
    v2 = v3.secondKey;
  }
  if (v2) {
    v1 = v2.thirdKey;
  }
  if (v1) {
    v0 = v1.fourthKey;
  }
  return v0;
}
```

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`. This argument is basically just another object, like the `scope` argument is. The contract is that parsed expressions should look things up from either the `scope` object or the `locals` object. They should try to use `locals` first and only if that fails fall back to `scope`.

This means you can effectively augment or override `scope` attributes with `locals`. In other words, you can make certain attributes available to expressions without putting them on any scope. This is sometimes useful especially in directives. For example, the `ngClick` directive lets you access the click event from your click handler expression by referring to `$event`. It does this by attaching it to the expression locals.

test/parse_spec.js

```
it('uses locals instead of scope when there is a matching key', function() {
  var fn = parse('aKey');
  var scope  = {aKey: 42};
  var locals = {aKey: 43};
  expect(fn(scope, locals)).toBe(43);
});

it('does not use locals instead of scope when no matching key', function() {
  var fn = parse('aKey');
  var scope  = {aKey: 42};
  var locals = {otherKey: 43};
  expect(fn(scope, locals)).toBe(42);
});
```

> You may recall that we've already seen something like `locals` in `Scope.prototype.$eval`, which takes a `locals` argument. In fact, this argument gets passed to the expression directly, as we'll see when we integrate expressions with watches.

The locals vs. scope rule only applies to the *first* part of the key. If the first level of a multi-level lookup 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');
  var scope  = {aKey: {anotherKey: 42}};
  var locals = {aKey: {}};
  expect(fn(scope, locals)).toBeUndefined();
});
```


Our generated JavaScript function should now take two arguments, the second of which is the locals object. We'll call it `l` in the generated code:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0, vars: []};
  this.recurse(ast);
  /* jshint -W054 */
  return new Function('s', 'l',
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) + this.state.body.join(''));
  /* jshint +W054 */
};
```

Inside the compiler, the relevant section for us to modify is the code generated for `AST.Identifier`, which is currently doing a Scope attribute lookup. We should have an additional check there to see if we should be doing a locals lookup instead of the scope lookup:

src/parse.js

```
case AST.Identifier:
  intoId = this.nextId();
  this.if_('l',
    this.assign(intoId, this.nonComputedMember('l', ast.name)));
  this.if_(this.not('l') + ' && s',
    this.assign(intoId, this.nonComputedMember('s', ast.name)));
  return intoId;
```

This version tries to check `l` first, and only looks the identifier up from `s` if there is no `l`. The `not` method is new, so we should define it. It just negates the value of a JavaScript expression:

src/parse.js

```
ASTCompiler.prototype.not = function(e) {
  return '!(' + e + ')';
};
```


Our test cases are not passing yet, and the problem is in how we're checking the locals lookup. The contract is that locals should only be used if the property actually exists in locals, whereas currently we always use locals if it merely exists. We must instead check if `l` actually has an attribute matching the identifier, which we can do with a new helper method called `getHasOwnProperty`:

src/parse.js

```
case AST.Identifier:
  intoId = this.nextId();
  this.if_(this.getHasOwnProperty('l', ast.name),
    this.assign(intoId, this.nonComputedMember('l', ast.name)));
  this.if_(this.not(this.getHasOwnProperty('l', ast.name)) + ' && s',
    this.assign(intoId, this.nonComputedMember('s', ast.name)));
  return intoId;
```

The `getHasOwnProperty` method takes an object and a property name. It first checks that the object exists, and then that the property name is included in the object. For the latter it uses JavaScript's in operator:

src/parse.js

```
ASTCompiler.prototype.getHasOwnProperty = function(object, property) {
  return object + '&&(' + this.escape(property) + ' in ' + object + ')';
};
```

Computed Attribute Lookup

We've seen how you can access attributes on scopes in a non-computed way using the dot operator. The second way you can do that in Angular expressions (as you can in JavaScript) is *computed attribute lookup* using square bracket notation:

test/parse_spec.js

```
it('parses a simple computed 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 computed 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 or *computed* in some other way (hence the name). You can't do that with the dot notation:

test/parse_spec.js

```
it('parses a computed 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* computed property accesses - as the key:

test/parse_spec.js

```
it('parses computed access with another access as property', function() {
  var fn = parse('lock[keys["aKey"]]');
  expect(fn({keys: {aKey: 'theKey'},  lock: {theKey: 42}})).toBe(42);
});
```

The expression `lock[key]` consists of four tokens: The identifier token `lock`, a single character token `'['`, the identifier token `key`, and a single character token `']'`. Together they form a primary AST node. We'll augment `AST.primary` to look out for opening square brackets in addition to dots:

src/parse.js

```
AST.prototype.primary = function() {
  var primary;
  if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    primary = this.constants[this.consume().text];
  } else if (this.peek().identifier) {
    primary = this.identifier();
  } else {
    primary = this.constant();
  }
  var next;
```



```
  while ((next = this.expect('.')) || (next = this.expect('['))) {
    primary = {
      type: AST.MemberExpression,
      object: primary,
      property: this.identifier()
    };
  }
  return primary;
};
```

To make the alternative expectations a bit more concise, we can extend `expect` and `peek` to support multiple alternative tokens. Let's go ahead and bump the number of alternatives we can use to four:

src/parse.js

```
AST.prototype.expect = function(e1, e2, e3, e4) {
  var token = this.peek(e1, e2, e3, e4);
  if (token) {
    return this.tokens.shift();
  }
};
AST.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];
    }
  }
};
```

Now we can go ahead and use the two-argument form in `primary`:

src/parse.js

```
var next;
while ((next = this.expect('.', '['))) {
  primary = {
    type: AST.MemberExpression,
    object: primary,
    property: this.identifier()
  };
}
```

If what we encounter is an opening square bracket, we should also consume a closing square bracket before we're done:

src/parse.js


```
var next;
while ((next = this.expect('.', '['))) {
  primary = {
    type: AST.MemberExpression,
    object: primary,
    property: this.identifier()
  };
  if (next.text === '[') {
    this.consume(']');
  }
}
```

Also, as we saw in our tests, what goes between the square brackets is not an identifier. It is a whole other primary expression. To support this it's actually better to branch out the computed and non-computed cases to handle them separately:

src/parse.js

```
var next;
while ((next = this.expect('.', '['))) {
  if (next.text === '[') {
    primary = {
      type: AST.MemberExpression,
      object: primary,
      property: this.primary()
    };
    this.consume(']');
  } else {
    primary = {
      type: AST.MemberExpression,
      object: primary,
      property: this.identifier()
    };
  }
}
```

The AST compiler is also going to need to know whether it is dealing with a computed or non-computed property access. Before we're done with the AST builder, let's add information about that to the AST nodes:

src/parse.js

```
var next;
while ((next = this.expect('.', '['))) {
  if (next.text === '[') {
    primary = {
      type: AST.MemberExpression,
```

```
      object: primary,
      property: this.primary(),
      computed: true
    };
    this.consume(']');
  } else {
    primary = {
      type: AST.MemberExpression,
      object: primary,
      property: this.identifier(),
      computed: false
    };
  }
}
```

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.

Moving on to the AST compiler, we now have two different types of `AST.MemberExpression`, which we should generate different kind of code for:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object);
  if (ast.computed) {

  } else {
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
  }
  return intoId;
```

Since the property for the computed lookup is an arbitrary expression itself, we should first recurse into it:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object);
  if (ast.computed) {
    var right = this.recurse(ast.property);
  } else {
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
  }
  return intoId;
```


That gives us the computed property that we should look up. We can complete the picture by doing the actual lookup and assigning the result as the result of the member expression:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object);
  if (ast.computed) {
    var right = this.recurse(ast.property);
    this.if_(left,
      this.assign(intoId, this.computedMember(left, right)));
  } else {
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
  }
  return intoId;
```

The new `computedMember` method is the last piece we need before we're done with computed lookup. It generates the JavaScript for a computed attribute access:

src/parse.js

```
ASTCompiler.prototype.computedMember = function(left, right) {
  return '(' + left + ')[' + right + ']';
};
```

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. Let's begin by adding the parentheses as character tokens that the Lexer will emit:

src/parse.js

```
} else if (this.is('[],{}:.()')) {
  this.tokens.push({
    text: this.ch
  });
  this.index++;
```

Function calls are processed as primary AST nodes, just like property accesses. In the `while` loop in `AST.primary` we'll consume not only square brackets and dots, but also opening parentheses. When we come across one, we generate a `CallExpression` node, and set the previous primary expression as the *callee* (meaning the function to call):

src/parse.js

```
var next;
while ((next = this.expect('.', '[', '('))) {
  if (next.text === '[') {
    primary = {
      type: AST.MemberExpression,
      object: primary,
      property: this.primary(),
      computed: true
    };
    this.consume(']');
  } else if (next.text === '.') {
    primary = {
      type: AST.MemberExpression,
      object: primary,
      property: this.identifier(),
      computed: false
    };
  } else if (next.text === '(') {
    primary = {type: AST.CallExpression, callee: primary};
    this.consume(')');
  }
}
```

The `CallExpression` constant needs to be defined:

src/parse.js


```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
AST.MemberExpression = 'MemberExpression';
AST.CallExpression = 'CallExpression';
```

Now we're ready to compile the call expression into JavaScript. We can do it by first obtaining the function to call by recursing to `callee`, and then generating the JavaScript for calling that function - but only if it exists:

src/parse.js

```
case AST.CallExpression:
  var callee = this.recurse(ast.callee);
  return callee + '&&' + callee + '()';
```

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 arguments 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);
});
```

Arguments 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);
});
```

Arguments 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 AST builder, we should be prepared to parse any arguments that go between the opening and closing parentheses. We'll do it with a new method called `parseArguments`:

src/parse.js

```
} else if (next.text === '(') {
  primary = {
    type: AST.CallExpression,
    callee: primary,
    arguments: this.parseArguments()
  };
  this.consume(')');
```

This method collects primary expressions until it sees a closing parenthesis, in precisely the same way as we did with array literals - with the exception that with arguments we don't support tailing commas:

src/parse.js

```
AST.prototype.parseArguments = function() {
  var args = [];
  if (!this.peek(')')) {
    do {
      args.push(this.primary());
    } while (this.expect(','));
  }
  return args;
};
```


As we compile these argument expressions into JavaScript, we can recurse to each one and collect the results into an array:

src/parse.js

```
case AST.CallExpression:
  var callee = this.recurse(ast.callee);
  var args = _.map(ast.arguments, function(arg) {
    return this.recurse(arg);
  }, this);
  return callee + '&&' + callee + '()';
```

We can then join the argument expressions into the generated function call:

src/parse.js

```
case AST.CallExpression:
  var callee = this.recurse(ast.callee);
  var args = _.map(ast.arguments, function(arg) {
    return this.recurse(arg);
  }, this);
  return callee + '&&' + callee + '(' + args.join(',') + ')';
```

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 is invoked as a *method*. What that means is that the function body will have the `this` keyword bound to the containing object. So, in these test cases, `this` in `aFunction` should point to `aMember` because of the way we call it in the expression. For computed attribute accesses:

test/parse_spec.js

```
it('calls methods accessed as computed properties', function() {
  var scope = {
    anObject: {
      aMember: 42,
      aFunction: function() {
        return this.aMember;
      }
    }
  };
  var fn = parse('anObject["aFunction"]()');
  expect(fn(scope)).toBe(42);
});
```


And for non-computed accesses:

test/parse_spec.js

```
it('calls methods accessed as non-computed properties', function() {
  var scope = {
    anObject: {
      aMember: 42,
      aFunction: function() {
        return this.aMember;
      }
    }
  };
  var fn = parse('anObject.aFunction()');
  expect(fn(scope)).toBe(42);
});
```

All the steps for making this work are going to be in the AST compiler. What we'll need to do is generate the right kind of JavaScript in the `CallExpression` branch of `recurse`, so that `this` will be bound to the object that was referenced in the original Angular expression.

The key to doing that is to introduce a "call context" object, in which information about the method call will be stored. We'll introduce such an object for call expressions. We will then pass it into the `recurse` method as the second argument when we process the callee:

src/parse.js

```
case AST.CallExpression:
  var callContext = {};
  var callee = this.recurse(ast.callee, callContext);
  var args = _.map(ast.arguments, function(arg) {
    return this.recurse(arg);
  }, this);
  return callee + '&&' + callee + '(' + args.join(',') + ')';
```

In the `recurse` method's declaration we now need to be able to receive a second argument. Here we'll just call it `context`:

src/parse.js

```
ASTCompiler.prototype.recurse = function(ast, context) {
  // ...
};
```


What we expect to happen when we pass in the context to `recurse` is that if we're dealing with a method call, three attributes will get populated on it:

- `context` - The owning object of the method. Will eventually become `this`.
- `name` - The method's property name in the owning object.
- `computed` - Whether the method was accessed as a computed property or not.

The `CallExpression` branch makes use of these three properties to form the callee that will cause the correct `this` to be used in the generated call expression:

src/parse.js

```
case AST.CallExpression:
  var callContext = {};
  var callee = this.recurse(ast.callee, callContext);
  var args = _.map(ast.arguments, function(arg) {
    return this.recurse(arg);
  }, this);
  if (callContext.name) {
    if (callContext.computed) {
      callee = this.computedMember(callContext.context, callContext.name);
    } else {
      callee = this.nonComputedMember(callContext.context, callContext.name);
    }
  }
  return callee + '&&' + callee + '(' + args.join(',') + ')';
```

What's happening here is that we are reconstrucing the computed or non-computed property access in the JavaScript code. Doing that causes the `this` binding to occur.

Now that we've seen how the call context will be used, we should populate it in the `MemberExpression` branch, which is where the callee of method call expressions is formed. The `context` attribute of the context should be the owning object of the member expression:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object);
  if (context) {
    context.context = left;
  }
  if (ast.computed) {
    var right = this.recurse(ast.property);
    this.if_(left,
      this.assign(intoId, this.computedMember(left, right)));
  } else {
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
  }
  return intoId;
```


The `name` and `computed` attributes of the context are different depending on whether the member lookup is computed or not:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object);
  if (context) {
    context.context = left;
  }
  if (ast.computed) {
    var right = this.recurse(ast.property);
    this.if_(left,
      this.assign(intoId, this.computedMember(left, right)));
    if (context) {
      context.name = right;
      context.computed = true;
    }
  } else {
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
    if (context) {
      context.name = ast.property.name;
      context.computed = false;
    }
  }
  return intoId;
```

Now we are correctly generating method calls and our test suite is happy.

A feature closely related to method calls is what happens to `this` for non-method functions. When you call a standalone function in an Angular expression, its `this` is actually bound to the Scope:

test/parse_spec.js

```
it('binds bare functions to the scope', function() {
  var scope = {
    aFunction: function() {
      return this;
    }
  };
  var fn = parse('aFunction()');
  expect(fn(scope)).toBe(scope);
});
```


An exception to this is when the function was attached to the expression locals instead of the scope. In that case, `this` points to the locals:

test/parse_spec.js

```
it('binds bare functions on locals to the locals', function() {
  var scope = {};
  var locals = {
    aFunction: function() {
      return this;
    }
  };
  var fn = parse('aFunction()');
  expect(fn(scope, locals)).toBe(locals);
});
```

We can implement this by populating the context for `Identifier` expressions as well. The context is either `l` or `s`, the name is the identifier name, and `computed` is always `false`:

src/parse.js

```
case AST.Identifier:
  intoId = this.nextId();
  this.if_(this.getHasOwnProperty('l', ast.name),
    this.assign(intoId, this.nonComputedMember('l', ast.name)));
  this.if_(this.not(this.getHasOwnProperty('l', ast.name)) + ' && s',
    this.assign(intoId, this.nonComputedMember('s', ast.name)));
  if (context) {
    context.context = this.getHasOwnProperty('l', ast.name) + '?l:s';
    context.name = ast.name;
    context.computed = false;
  }
  return intoId;
```

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);
});
```

You can assign not only simple identifiers, but also computed and non-computed properties, and nested combinations of them:

test/parse_spec.js

```
it('can assign a computed object property', function() {
  var fn = parse('anObject["anAttribute"] = 42');
  var scope = {anObject: {}};
  fn(scope);
  expect(scope.anObject.anAttribute).toBe(42);
});

it('can assign a non-computed object property', function() {
  var fn = parse('anObject.anAttribute = 42');
  var scope = {anObject: {}};
  fn(scope);
  expect(scope.anObject.anAttribute).toBe(42);
});

it('can assign a nested object property', function() {
  var fn = parse('anArray[0].anAttribute = 42');
  var scope = {anArray: [{}]};
  fn(scope);
  expect(scope.anArray[0].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 AST builder to use. This time we'll need one for the = sign that denotes assignments:

src/parse.js


```
} else if (this.is('[],{}:.()=')) {
  this.tokens.push({
    text: this.ch
  });
  this.index++;
```

Assignment is not what we've been calling a "primary" AST node, and it'll not be built by the existing `AST.primary` function. It'll have a function of its own, which we'll call `assignment`. In it, we begin by consuming the left hand side token, which is a primary node. The left hand side is then followed by an equals sign token, and then the right hand side, which is another primary node:

src/parse.js

```
AST.prototype.assignment = function() {
  var left = this.primary();
  if (this.expect('=')) {
    var right = this.primary();

  }
  return left;
};
```

Together these form an `AssignmentExpression` with `left` and `right` subexpressions:

src/parse.js

```
AST.prototype.assignment = function() {
  var left = this.primary();
  if (this.expect('=')) {
    var right = this.primary();
    return {type: AST.AssignmentExpression, left: left, right: right};
  }
  return left;
};
```

We need to define the `AssignmentExpression` constant as well:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
AST.MemberExpression = 'MemberExpression';
AST.CallExpression = 'CallExpression';
AST.AssignmentExpression = 'AssignmentExpression';
```


The remaining issue in the AST builder is that `assignment` is currently an "orphan" method: It is not called by anything.

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 build either an assignment expression or just a plain primary expression. This pattern of trying to build 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 `program` with a call to `assignment`, and that'll take care of all expressions we have so far, whether assignments or something else:

src/parse.js

```
AST.prototype.program = function() {
  return {type: AST.Program, body: this.assignment()};
};
```

Assignments can also be contained in arrays as items, so we should change the method we call in array expressions as well:

src/parse.js

```
AST.prototype.arrayDeclaration = function() {
  var elements = [];
  if (!this.peek(']')) {
    do {
      if (this.peek(']')) {
        break;
      }
      elements.push(this.assignment());
    } while (this.expect(','));
  }
  this.consume(']');
  return {type: AST.ArrayExpression, elements: elements};
};
```

The same goes for values in object expressions:

src/parse.js

```
AST.prototype.object = function() {
  var properties = [];
  if (!this.peek('}')) {
    do {
      var property = {type: AST.Property};
      if (this.peek().identifier) {
```



```
        property.key = this.identifier();
      } else {
        property.key = this.constant();
      }
      this.consume(':');
      property.value = this.assignment();
      properties.push(property);
    } while (this.expect(','));
  }
  this.consume('}');
  return {type: AST.ObjectExpression, properties: properties};
};
```

And it goes for function arguments as well:

src/parse.js

```
AST.prototype.parseArguments = function() {
  var args = [];
  if (!this.peek(')')) {
    do {
      args.push(this.assignment());
    } while (this.expect(','));
  }
  return args;
};
```

In the AST compiler, we'll first process the left hand side of the assignment - the identifier or member to assign. We'll make use of the context feature built in the previous section to collect information about it as we recurse into it:

src/parse.js

```
case AST.AssignmentExpression:
  var leftContext = {};
  this.recurse(ast.left, leftContext);
```

The left hand side of the generated assignment expression will be different depending on whether the left hand side was computed or not:

src/parse.js

```
case AST.AssignmentExpression:
  var leftContext = {};
  this.recurse(ast.left, leftContext);
  var leftExpr;
  if (leftContext.computed) {
    leftExpr = this.computedMember(leftContext.context, leftContext.name);
  } else {
    leftExpr = this.nonComputedMember(leftContext.context, leftContext.name);
  }
```

The assignment itself is a combination of the left hand side and the right hand side, separated by `=`. It becomes the result of the expression:

src/parse.js

```
case AST.AssignmentExpression:
  var leftContext = {};
  this.recurse(ast.left, leftContext);
  var leftExpr;
  if (leftContext.computed) {
    leftExpr = this.computedMember(leftContext.context, leftContext.name);
  } else {
    leftExpr = this.nonComputedMember(leftContext.context, leftContext.name);
  }
  return this.assign(leftExpr, this.recurse(ast.right));
```

An interesting thing about nested assignments 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 assignment path that do not exist', function() {
  var fn = parse('some["nested"].property.path = 42');
  var scope = {};
  fn(scope);
  expect(scope.some.nested.property.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.

This is done by passing a new, third argument to `recurse`, which tells it to create any missing objects there might be. We pass it as `true` when recursing the left side of an assignment:

src/parse.js

```
case AST.AssignmentExpression:
  var leftContext = {};
  this.recurse(ast.left, leftContext, true);
  // ...
```

In `recurse` we'll receive this argument:

src/parse.js


```
ASTCompiler.prototype.recurse = function(ast, context, create) {
  // ...
};
```

When handling `MemberExpressions` we'll then check if we're supposed to be creating missing objects. If we are, we assign empty objects to the members. We need to handle that separately for the computed and non-computed cases:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object);
  if (context) {
    context.context = left;
  }
  if (ast.computed) {
    var right = this.recurse(ast.property);
    if (create) {
      this.if_(this.not(this.computedMember(left, right)),
        this.assign(this.computedMember(left, right), '{}'));
    }
    this.if_(left,
      this.assign(intoId, this.computedMember(left, right)));
    if (context) {
      context.name = right;
      context.computed = true;
    }
  } else {
    if (create) {
      this.if_(this.not(this.nonComputedMember(left, ast.property.name)),
        this.assign(this.nonComputedMember(left, ast.property.name), '{}'));
    }
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
    if (context) {
      context.name = ast.property.name;
      context.computed = false;
    }
  }
  return intoId;
```

This takes care of the last member expression before the assignment, but for nested paths such as the one in our test case, we need to recursively pass the `create` flag for the next left hand side expression as well:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object, undefined, create);
  // ...
```

The path will terminate in an `Identifier` expression. It should be able to populate an empty object on the scope if one does not exist and we're supposed to be creating missing objects. We do it only when there's no match either on the locals or the scope:

src/parse.js

```
case AST.Identifier:
  intoId = this.nextId();
  this.if_(this.getHasOwnProperty('l', ast.name),
    this.assign(intoId, this.nonComputedMember('l', ast.name)));
  if (create) {
    this.if_(this.not(this.getHasOwnProperty('l', ast.name)) +
             ' && s && ' +
             this.not(this.getHasOwnProperty('s', ast.name)),
      this.assign(this.nonComputedMember('s', ast.name), '{}'));
  }
  this.if_(this.not(this.getHasOwnProperty('l', ast.name)) + ' && s',
    this.assign(intoId, this.nonComputedMember('s', ast.name)));
  if (context) {
    context.context = this.getHasOwnProperty('l', ast.name) + '?l:s';
    context.name = ast.name;
    context.computed = false;
  }
  return intoId;
```

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 AST compiler: Apart from literals you can only work on things that have been attached to the scope (or the locals). The objects with potentially dangerous operations, such as `window` simply aren't accessible.

There are a few ways around this in our current implementation though: In particular, 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 the expression functions. It turns out that if we don't take measures to prevent it, that very 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:


```
aFunction.constructor("return window;")()
```

Apart from the function constructor, there are a few other common object members that could cause security issues because calling them may have unpredictably wide effects:

- `__proto__` is a non-standard, deprecated accessor for an object's prototype. It allows not only getting but also *setting* global prototypes, making it potentially dangerous.
- `__defineGetter__`, `__lookupGetter__`, `__defineSetter__`, and `__lookupSetter__` are non-standard functions for defining properties on object in terms of getter and setter functions. Since they are not standardized and not supported in all browsers, and they potentially allow redefining well-known global properties, Angular disallows them in expressions.

Let's add tests that verify none of the above six members are available in expressions:

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();
});

it('does not allow accessing __proto__', function() {
  expect(function() {
    var fn = parse('obj.__proto__');
    fn({obj: { }});
  }).toThrow();
});

it('does not allow calling __defineGetter__', function() {
  expect(function() {
    var fn = parse('obj.__defineGetter__("evil", fn)');
    fn({obj: { }, fn: function() { }});
  }).toThrow();
});

it('does not allow calling __defineSetter__', function() {
  expect(function() {
    var fn = parse('obj.__defineSetter__("evil", fn)');
    fn({obj: { }, fn: function() { }});
  }).toThrow();
});

it('does not allow calling __lookupGetter__', function() {
  expect(function() {
    var fn = parse('obj.__lookupGetter__("evil")');
    fn({obj: { }});
  }).toThrow();
});
```



```
it('does not allow calling __lookupSetter__', function() {
  expect(function() {
    var fn = parse('obj.__lookupSetter__("evil")');
    fn({obj: { }});
  }).toThrow();
});
```

The security measure we'll take against these kinds of attacks is to simply disallow accessing any members on any objects that have these names. To that effect, we'll introduce a helper function that checks the name of a member and throws an exception if it's not allowed:

src/parse.js

```
function ensureSafeMemberName(name) {
  if (name === 'constructor' || name === '__proto__' ||
      name === '__defineGetter__' || name === '__defineSetter__' ||
      name === '__lookupGetter__' || name === '__lookupSetter__') {
    throw 'Attempting to access a disallowed field in Angular expressions!';
  }
}
```

Now we need to sprinkle calls to this function in a few places in the AST compiler. In identifiers we'll call it with the name of the identifier:

src/parse.js

```
case AST.Identifier:
  ensureSafeMemberName(ast.name);
  // ...
```

In non-computed member accesses we'll check the property name:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object, undefined, create);
  if (context) {
    context.context = left;
  }
  if (ast.computed) {
    var right = this.recurse(ast.property);
    if (create) {
      this.if_(this.not(this.computedMember(left, right)),
        this.assign(this.computedMember(left, right), '{}'));
    }
```



```
    this.if_(left,
      this.assign(intoId, this.computedMember(left, right)));
    if (context) {
      context.name = right;
      context.computed = true;
    }
  } else {
    ensureSafeMemberName(ast.property.name);
    if (create) {
      this.if_(this.not(this.nonComputedMember(left, ast.property.name)),
        this.assign(this.nonComputedMember(left, ast.property.name), '{}'));
    }
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
    if (context) {
      context.name = ast.property.name;
      context.computed = false;
    }
  }
  return intoId;
```

In non-computed member accesses we'll need to do a bit more work, as we won't know the name of the property at parse time. Instead we'll need to call `ensureSafeMemberName` at *runtime*, whenever the expression is evaluated.

First of all, we'll need to make `ensureSafeMemberName` available to expressions at runtime. Let's first refactor our generated function code so that it itself is not the expression function, but *returns* the expression function:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0, vars: []};
  this.recurse(ast);
  var fnString = 'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      '') +
    this.state.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  return new Function(fnString)();
  /* jshint +W054 */
};
```

As we now have this higher-order function, we can use it to pass in arguments that will become available inside the closure of the generated code. At this point we'll pass in `ensureSafeMemberName` so we can use it at runtime:

src/parse.js


```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0, vars: []};
  this.recurse(ast);
  var fnString = 'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) +
    this.state.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  return new Function('ensureSafeMemberName', fnString)(ensureSafeMemberName);
  /* jshint +W054 */
};
```

Next, we'll generate a call to this function for the right side of computed member accesses:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object, undefined, create);
  if (context) {
    context.context = left;
  }
  if (ast.computed) {
    var right = this.recurse(ast.property);
    this.addEnsureSafeMemberName(right);
    if (create) {
      this.if_(this.not(this.computedMember(left, right)),
        this.assign(this.computedMember(left, right), '{}'));
    }
    this.if_(left,
      this.assign(intoId, this.computedMember(left, right)));
    if (context) {
      context.name = right;
      context.computed = true;
    }
  } else {
    ensureSafeMemberName(ast.property.name);
    if (create) {
      this.if_(this.not(this.nonComputedMember(left, ast.property.name)),
        this.assign(this.nonComputedMember(left, ast.property.name), '{}'));
    }
    this.if_(left,
      this.assign(intoId, this.nonComputedMember(left, ast.property.name)));
    if (context) {
```



```
      context.name = ast.property.name;
      context.computed = false;
    }
  }
  return intoId;
```

The `addEnsureSafeMemberName` generates the call to `ensureSafeMemberName`:

src/parse.js

```
ASTCompiler.prototype.addEnsureSafeMemberName = function(expr) {
  this.state.body.push('ensureSafeMemberName(' + expr + ');');
};
```

Ensuring Safe Objects

The second security measure Angular expressions provide for us has more 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 computed property', function() {
  var fn = parse('anObject["wnd"]');
  expect(function() { fn({anObject: {wnd: window}}); }).toThrow();
});

it('does not allow accessing window as non-computed 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

```
function ensureSafeObject(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 first make this new helper function available at runtime:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0, vars: []};
  this.recurse(ast);
  var fnString = 'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) +
    this.state.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  return new Function(
    'ensureSafeMemberName',
    'ensureSafeObject',
    fnString)(
      ensureSafeMemberName,
      ensureSafeObject);
  /* jshint +W054 */
};
```

Now we can wrap the results of member accesses with calls to `ensureSafeObject`:

src/parse.js

```
case AST.MemberExpression:
  intoId = this.nextId();
  var left = this.recurse(ast.object, undefined, create);
  if (context) {
    context.context = left;
```



```
  }
  if (ast.computed) {
    var right = this.recurse(ast.property);
    this.addEnsureSafeMemberName(right);
    if (create) {
      this.if_(this.not(this.computedMember(left, right)),
        this.assign(this.computedMember(left, right), '{}'));
    }
    this.if_(left,
      this.assign(intoId,
        'ensureSafeObject(' + this.computedMember(left, right) + ')'));
    if (context) {
      context.name = right;
      context.computed = true;
    }
  } else {
    ensureSafeMemberName(ast.property.name);
    if (create) {
      this.if_(this.not(this.nonComputedMember(left, ast.property.name)),
        this.assign(this.nonComputedMember(left, ast.property.name), '{}'));
    }
    this.if_(left,
      this.assign(intoId,
        'ensureSafeObject(' +
          this.nonComputedMember(left, ast.property.name) + ')'));
    if (context) {
      context.name = ast.property.name;
      context.computed = false;
    }
  }
  return intoId;
```

One should not be allowed to pass unsafe objects in as function arguments:

src/parse.js

```
it('does not allow passing window as function argument', function() {
  var fn = parse('aFunction(wnd)');
  expect(function() {
    fn({aFunction: function() { }, wnd: window});
  }).toThrow();
});
```

We need to wrap each of the argument expressions of a call expression in `ensureSafeObject`:

src/parse.js

```
case AST.CallExpression:
  var callContext = {};
  var callee = this.recurse(ast.callee, callContext);
  var args = _.map(ast.arguments, function(arg) {
    return 'ensureSafeObject(' + this.recurse(arg) + ')';
  }, this);
  // ...
```

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 methods on window', function() {
  var fn = parse('wnd.scrollTo(0)');
  expect(function() {
    fn({wnd: window});
  }).toThrow();
});
```

In this case what we should check is the context of the method call:

src/parse.js

```
case AST.CallExpression:
  var callContext = {};
  var callee = this.recurse(ast.callee, callContext);
  var args = _.map(ast.arguments, function(arg) {
    return 'ensureSafeObject(' + this.recurse(arg) + ')';
  }, this);
  if (callContext.name) {
    this.addEnsureSafeObject(callContext.context);
    if (callContext.computed) {
      callee = this.computedMember(callContext.context, callContext.name);
    } else {
      callee = this.nonComputedMember(callContext.context, callContext.name);
    }
  }
  return callee + '&&' + callee + '(' + args.join(',') + ')';
```

The `addEnsureSafeObject` method here is new, and we should add it:

src/parse.js

```
ASTCompiler.prototype.addEnsureSafeObject = function(expr) {
  this.state.body.push('ensureSafeObject(' + expr + ');');
};
```


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();
});
```

This time we should wrap the *return value* of the function call in `ensureSafeObject`:

src/parse.js

```
case AST.CallExpression:
  var callContext = {};
  var callee = this.recurse(ast.callee, callContext);
  var args = _.map(ast.arguments, function(arg) {
    return 'ensureSafeObject(' + this.recurse(arg) + ')';
  }, this);
  if (callContext.name) {
    this.addEnsureSafeObject(callContext.context);
    if (callContext.computed) {
      callee = this.computedMember(callContext.context, callContext.name);
    } else {
      callee = this.nonComputedMember(callContext.context, callContext.name);
    }
  }
  return callee + '&&ensureSafeObject(' + callee + '(' + args.join(',') + '))';
```

It should not be allowed to assign an unsafe object on the scope:

test/parse_spec.js

```
it('does not allow assigning window', function() {
  var fn = parse('wnd = anObject');
  expect(function() {
    fn({anObject: window});
  }).toThrow();
});
```

This means we should wrap the right hand side of assignments too:

src/parse.js


```
case AST.AssignmentExpression:
  var leftContext = {};
  this.recurse(ast.left, leftContext, true);
  var leftExpr;
  if (leftContext.computed) {
    leftExpr = this.computedMember(leftContext.context, leftContext.name);
  } else {
    leftExpr = this.nonComputedMember(leftContext.context, leftContext.name);
  }
  return this.assign(leftExpr,
    'ensureSafeObject(' + this.recurse(ast.right) + ')');
```

And finally, one should not be able to reference an unsafe object using an identifier if it happens to be simply aliased on the scope:

test/parse_spec.js

```
it('does not allow referencing window', function() {
  var fn = parse('wnd');
  expect(function() {
    fn({wnd: window});
  }).toThrow();
});
```

We need to generate a safeness check for Identifiers too:

src/parse.js

```
case AST.Identifier:
  ensureSafeMemberName(ast.name);
  intoId = this.nextId();
  this.if_(this.getHasOwnProperty('l', ast.name),
    this.assign(intoId, this.nonComputedMember('l', ast.name)));
  if (create) {
    this.if_(this.not(this.getHasOwnProperty('l', ast.name)) +
             ' && s && ' +
             this.not(this.getHasOwnProperty('s', ast.name)),
      this.assign(this.nonComputedMember('s', ast.name), '{}'));
  }
  this.if_(this.not(this.getHasOwnProperty('l', ast.name)) + ' && s',
    this.assign(intoId, this.nonComputedMember('s', ast.name)));
  if (context) {
    context.context = this.getHasOwnProperty('l', ast.name) + '?l:s';
    context.name = ast.name;
    context.computed = false;
  }
  this.addEnsureSafeObject(intoId);
  return intoId;
```


`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:

src/parse.js

```
function ensureSafeObject(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 third 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


```
function ensureSafeObject(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;
}
```

The fourth and final dangerous object we should think about is the `Object` object. Apart from acting as the constructor for primitive object wrappers, it holds a number of helper functions, such as `Object.defineProperty()`, `Object.freeze()`, `Object.getOwnPropertyDescriptor()`, and `Object.setPrototypeOf()`. It is this latter role of `Object` that we are concerned about. Some of these functions are potentially dangerous if an attacker gets access to them. Thus, we will forbid referencing `Object` completely:

test/parse_spec.js

```
it('does not allow calling functions on Object', function() {
  var fn = parse('obj.create({})');
  expect(function() {
    fn({obj: Object});
  }).toThrow();
});
```

We check if the object looks like `Object` by looking at a couple of attribute names only `Object` is likely to have:

src/parse.js

```
function ensureSafeObject(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!';
    } else if (obj.getOwnPropertyNames || obj.getOwnPropertyDescriptor) {
      throw 'Referencing Object in Angular expressions is disallowed!';
    }
  }
  return obj;
}
```


Ensuring Safe Functions

We just saw how you cannot call a function in an expression if it happens to be the function constructor function. There's also another thing Angular prevents you from doing with functions, which is to rebind their receiver (`this`) to something else:

test/parse_spec.js

```
it('does not allow calling call', function() {
  var fn = parse('fun.call(obj)');
  expect(function() { fn({fun: function() { }, obj: {}}); }).toThrow();
});

it('does not allow calling apply', function() {
  var fn = parse('fun.apply(obj)');
  expect(function() { fn({fun: function() { }, obj: {}}); }).toThrow();
});
```

The methods `call` and `apply` (as well as `bind`) are all different ways to invoke a function so that its receiver is changed from the original. In all of the test cases above, `this` would be bound to `obj` within the body of `fun`. Since rebinding `this` can cause functions to behave differently than the function author originally intended, Angular simply disallows them in expressions so that this cannot be abused in injection attacks.

In the previous section we ensured the safety of invoked functions with `ensureSafeObject`. Now we're going to switch from that to a new function called `ensureSafeFunction`. The first thing it does is check that the function isn't the function constructor, just like `ensureSafeObject` does:

src/parse.js

```
function ensureSafeFunction(obj) {
  if (obj) {
    if (obj.constructor === obj) {
      throw 'Referencing Function in Angular expressions is disallowed!';
    }
  }
  return obj;
}
```

The second responsibility of `ensureSafeFunction` is to see that the function isn't `call`, `apply`, or `bind`. Let's introduce them as constant values in `parse.js` so we can compare the function to them:

src/parse.js

```
var CALL = Function.prototype.call;
var APPLY = Function.prototype.apply;
var BIND = Function.prototype.bind;
```


Now we can refer to these constants in `ensureSafeFunction`:

src/parse.js

```
function ensureSafeFunction(obj) {
  if (obj) {
    if (obj.constructor === obj) {
      throw 'Referencing Function in Angular expressions is disallowed!';
    } else if (obj === CALL || obj === APPLY || obj === BIND) {
      throw 'Referencing call, apply, or bind in Angular expressions '+
        'is disallowed!';
    }
  }
  return obj;
}
```

We should now make `ensureSafeFunction` available at runtime:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0, vars: []};
  this.recurse(ast);
  var fnString = 'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) +
    this.state.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  return new Function(
    'ensureSafeMemberName',
    'ensureSafeObject',
    'ensureSafeFunction',
    fnString)(
      ensureSafeMemberName,
      ensureSafeObject,
      ensureSafeFunction);
  /* jshint +W054 */
};
```

We should then generate a call to it for the callee of a function call expression:

src/parse.js


```
case AST.CallExpression:
  var callContext = {};
  var callee = this.recurse(ast.callee, callContext);
  var args = _.map(ast.arguments, function(arg) {
    return 'ensureSafeObject(' + this.recurse(arg) + ')';
  }, this);
  if (callContext.name) {
    this.addEnsureSafeObject(callContext.context);
    if (callContext.computed) {
      callee = this.computedMember(callContext.context, callContext.name);
    } else {
      callee = this.nonComputedMember(callContext.context, callContext.name);
    }
  }
  this.addEnsureSafeFunction(callee);
  return callee + '&&ensureSafeObject(' + callee + '(' + args.join(',') + '))';
```

Before we're done we still need to add the `addEnsureSafeFunction` helper method:

src/parse.js

```
ASTCompiler.prototype.addEnsureSafeFunction = function(expr) {
  this.state.body.push('ensureSafeFunction(' + expr + ');');
};
```

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 attributes 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 attributes the application developer has to have put them on the scope in the first place, which is something they should not be doing.

Summary

This chapter has been quite a ride. We have added a lot of code, and 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 computed and non-computed attribute lookup.
- How scope attributes can be overriden by locals
- How Angular expressions do function calls
- How Angular expressions obtain 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 the forgiving nature of Angular expressions is formed: 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 *operators*. Once we're done with that, our language will have the vocabulary for doing arithmetic, comparisons, and logical expressions.

Chapter 7

Operator Expressions

A 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 or a computed or non-computed 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 to add 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 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 the AST builder we'll introduce a new method `unary` that deals with unary operators and falls back to `primary` for everything else:

src/parse.js

```
AST.prototype.unary = function() {
  if (this.expect('+')) {

  } else {
    return this.primary();
  }
};
```

What `unary` actually does is build a `UnaryExpression` token, whose sole argument is expected to be a primary expression:

src/parse.js

```
AST.prototype.unary = function() {
  if (this.expect('+')) {
    return {
      type: AST.UnaryExpression,
      operator: '+',
      argument: this.primary()
    };
  } else {
    return this.primary();
  }
};
```


`UnaryExpression` is a new AST node type:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
AST.MemberExpression = 'MemberExpression';
AST.CallExpression = 'CallExpression';
AST.AssignmentExpression = 'AssignmentExpression';
AST.UnaryExpression = 'UnaryExpression';
```

To have unary expressions actually parsed, we need to call `unary` from somewhere. It's done from `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

```
AST.prototype.assignment = function() {
  var left = this.unary();
  if (this.expect('=')) {
    var right = this.unary();
    return {type: AST.AssignmentExpression, left: left, right: right};
  }
  return left;
};
```

Because `unary` falls back to `primary`, `assignment` now supports either of them on both the left and right hand sides.

At this point, we should revisit the `Lexer`. Our AST builder 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: We'll introduce a "constant" object called `OPERATORS` that contains tokens that we consider operators:

src/parse.js

```
var OPERATORS = {
  '+': true
};
```


All the values in this object will be `true`. We're just using an object instead of an array because objects allow us to check the presence of a key more efficiently, in constant time.

The Lexer still needs to emit the `+`. 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
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent();
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      var op = OPERATORS[this.ch];
      if (op) {
        this.tokens.push({text: this.ch});
        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.

The remaining piece of the puzzle is in the AST compiler, which now needs to learn how to compile unary expressions. What we can do there is to return a JavaScript fragment that consists of the operator of the expression, followed by the recursed value of the argument:

src/parse.js

```
case AST.UnaryExpression:
  return ast.operator + '(' + this.recurse(ast.argument) + ')';
```

The unary arithmetic operators in Angular expressions are different from JavaScript's unary operators in that they treat an undefined value as zero, whereas JavaScript would just treat it as `NaN`:

src/parse.js

```
it('replaces undefined with zero for unary +', function() {
  expect(parse('+a')({})).toBe(0);
});
```

Let's guard the operand of the unary expression with a call to a new method `ifDefined` that takes two arguments: An expression and a value to use if the expression is undefined, which in this case is `0`.

src/parse.js

```
case AST.UnaryExpression:
  return ast.operator +
    '(' + this.ifDefined(this.recurse(ast.argument), 0) + ')';
```

The `ifDefined` method generates a runtime JavaScript call to an `ifDefined` function with the same arguments:

src/parse.js

```
ASTCompiler.prototype.ifDefined = function(value, defaultValue) {
  return 'ifDefined(' + value + ',' + this.escape(defaultValue) + ')';
};
```

This function is passed in to the generated JavaScript function:

src/parse.js


```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {body: [], nextId: 0, vars: []};
  this.recurse(ast);
  var fnString = 'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) +
    this.state.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  return new Function(
    'ensureSafeMemberName',
    'ensureSafeObject',
    'ensureSafeFunction',
    'ifDefined',
    fnString)(
      ensureSafeMemberName,
      ensureSafeObject,
      ensureSafeFunction,
      ifDefined);
  /* jshint +W054 */
};
```

And finally, the actual implementation of `ifDefined` returns the value itself if it is defined, and otherwise the default value:

src/parse.js

```
function ifDefined(value, defaultValue) {
  return typeof value === 'undefined' ? defaultValue : value;
}
```

> We don't use LoDash here as it may not be available to expression code at runtime.

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:

src/parse.js

```
var OPERATORS = {
  '+': true,
  '!': true
};
```

In the AST builder we can now expect either `+` or `!` in `unary`. We also can no longer hardcode `+` in the AST node's operator, but must use the actual operator we found instead:

src/parse.js

```
AST.prototype.unary = function() {
  var token;
  if ((token = this.expect('+', '!'))) {
    return {
      type: AST.UnaryExpression,
      operator: token.text,
      argument: this.primary()
    };
  } else {
    return this.primary();
  }
};
```

This partially fixes our test case already, as we don't need to make any changes to the AST compiler to make this work. The test is still failing though, because we don't have the ability to apply `!` several times in a row. The trick to fix it is to simply expect another unary expression as the argument of `unary`:

src/parse.js

```
AST.prototype.unary = function() {
  var token;
  if ((token = this.expect('+', '!'))) {
    return {
      type: AST.UnaryExpression,
      operator: token.text,
      argument: this.unary()
    };
  } 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);
  expect(parse('-a')({})).toBe(0);
});
```

This too should first be added to the `OPERATORS` object:

src/parse.js

```
var OPERATORS = {
  '+': true,
  '!': true,
  '-': true
};
```

Then it should be expected by the AST builder in a unary position:

src/parse.js

```
AST.prototype.unary = function() {
  var token;
  if ((token = this.expect('+', '!', '-'))) {
    return {
      type: AST.UnaryExpression,
      operator: token.text,
      argument: this.unary()
    };
  } else {
    return this.primary();
  }
};
```

Now that we are matching characters with keys in the `OPERATORS` object, we have inadvertently introduced a bug into string expressions. Consider a string that contains a single exclamation mark - which also happens to be an operator when used outside of a string:

test/parse_spec.js


```
it('parses a ! in a string', function() {
  expect(parse('"!"')()).toBe('!');
});
```

This now causes an error, because the string token contains a `text` attribute with a value of `!`, which the AST builder interprets as a unary `!` operator! This should certainly not happen.

We need to modify our string tokens so that their `text` attribute does *not* hold the characters inside the string, but instead holds the *whole* original string token including the surrounding quotation marks. This way it will not be confused with operators. We'll collect the "raw" original string into a `rawString` variable in the `readString` method of the Lexer:

src/parse.js

```
Lexer.prototype.readString = function(quote) {
  this.index++;
  var string = '';
  var rawString = quote;
  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,
        value: string
      });
      return;
    } else if (ch === '\\') {
      escape = true;
    } else {
```



```
      string += ch;
    }
    this.index++;
  }
};
```

And that's it for unary 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);
});
```

First we'll put them in the collection of `OPERATORS` so that they will be emitted by the Lexer:

src/parse.js

```
var OPERATORS = {
  '+': true,
  '!': true,
  '-': true,
  '*': true,
  '/': true,
  '%': true
};
```

In the AST builder these operators are handled by a new method called `multiplicative`. It emits a `BinaryExpression` node (which means it's an expression with two arguments). Both the left and right hand arguments of the expression are expected to be unary expressions:

src/parse.js


```
AST.prototype.multiplicative = function() {
  var left = this.unary();
  var token;
  if ((token = this.expect('*', '/', '%'))) {
    left = {
      type: AST.BinaryExpression,
      left: left,
      operator: token.text,
      right: this.unary()
    };
  }
  return left;
};
```

Note that just like many of our previous AST builder methods, this method also has a fallback mode: It just returns a unary expression if it cannot match a multiplicative expression.

The new AST node type needs to be added:

src/parse.js

```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
AST.MemberExpression = 'MemberExpression';
AST.CallExpression = 'CallExpression';
AST.AssignmentExpression = 'AssignmentExpression';
AST.UnaryExpression = 'UnaryExpression';
AST.BinaryExpression = 'BinaryExpression';
```

Now, in `AST.assignment` we'll replace the call to `unary` with a call to `multiplicative` so that multiplicative operators actually get applied:

src/parse.js

```
AST.prototype.assignment = function() {
  var left = this.multiplicative();
  if (this.expect('=')) {
    var right = this.multiplicative();
    return {type: AST.AssignmentExpression, left: left, right: right};
  }
  return left;
};
```


In the AST compiler, we can now add support for binary expressions. They're basically very similar to unary expressions. The difference is just that there are two operands: One that goes on the left side of the operator and one that goes on the right.

src/parse.js

```
case AST.BinaryExpression:
  return '(' + this.recurse(ast.left) + ')' +
    ast.operator +
    '(' + this.recurse(ast.right) + ')';
```

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 AST builder functions call each other.

Right now, our "top-level" AST builder function is `assignment`. In turn, `assignment` invokes `multiplicative`, which invokes `unary`, which finally invokes `primary`. The very first thing each method does is to build 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.

We're already 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 long as there are more multiplicative operators to parse. The result of each step becomes the left hand side of the next step, increasing the depth of the syntax tree. In concrete terms there's actually nothing more to it than switching the `if` in `AST.multiplicative` to a `while`:

src/parse.js


```
AST.prototype.multiplicative = function() {
  var left = this.unary();
  var token;
  while ((token = this.expect('*', '/', '%'))) {
    left = {
      type: AST.BinaryExpression,
      left: left,
      operator: token.text,
      right: 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 these operators covered. On the AST building 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

```
AST.prototype.additive = function() {
  var left = this.multiplicative();
  var token;
  while ((token = this.expect('+')) || (token = this.expect('-'))) {
    left = {
      type: AST.BinaryExpression,
      left: left,
      operator: token.text,
      right: 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

```
AST.prototype.assignment = function() {
  var left = this.additive();
  if (this.expect('=')) {
    var right = this.additive();
    return {type: AST.AssignmentExpression, left: left, right: right};
  }
  return left;
};
```

Since we already implemented AST compilation for binary expressions when we were adding support for multiplicative operators, this actually already makes our test pass! From the compiler's point of view, there is no difference between multiplicative and additive operators - except for one thing:

As we saw with the unary operators, a missing operand for `+` or `-` is treated as zero. This is also the case for binary addition and subtraction. Either or both missing operands are replaced with zeros.

test/parse_spec.js


```
it('substitutes undefined with zero in addition', function() {
  expect(parse('a + 22')()).toBe(22);
  expect(parse('42 + a')()).toBe(42);
});

it('substitutes undefined with zero in subtraction', function() {
  expect(parse('a - 22')()).toBe(-22);
  expect(parse('42 - a')()).toBe(42);
});
```

The arguments in the compiled code need to be wrapped in `ifDefined`, but only for plus and minus:

src/parse.js

```
case AST.BinaryExpression:
  if (ast.operator === '+' || ast.operator === '-') {
    return '(' + this.ifDefined(this.recurse(ast.left), 0) + ')' +
      ast.operator +
      '(' + this.ifDefined(this.recurse(ast.right), 0) + ')';
  } else {
    return '(' + this.recurse(ast.left) + ')' +
      ast.operator +
      '(' + this.recurse(ast.right) + ')';
  }
  break;
```

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`

> As per the ECMAScript specification, `true` is coerced to `1` and `false` to `0` when used in a numeric context, which is what's happening in step 3 above.

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:

src/parse.js

```
var OPERATORS = {
  '+': true,
  '-': true,
  '!': true,
  '*': true,
  '/': true,
  '%': true,
  '==': true,
  '!=': true,
  '===': true,
  '!==': true,
  '<': true,
  '>': true,
  '<=': true,
  '>=': true
};
```

In the AST builder, 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

```
AST.prototype.equality = function() {
  var left = this.relational();
  var token;
  while ((token = this.expect('==', '!=', '===', '!==='))) {
    left = {
      type: AST.BinaryExpression,
      left: left,
      operator: token.text,
      right: this.relational()
    };
  }
```



```
  return left;
};
AST.prototype.relational = function() {
  var left = this.additive();
  var token;
  while ((token = this.expect('<', '>', '<=', '>='))) {
    left = {
      type: AST.BinaryExpression,
      left: left,
      operator: token.text,
      right: this.additive()
    };
  }
  return left;
};
```

Equality is now our lowest precedence operator after assignment, so that is what assignment should delegate to:

src/parse.js

```
AST.prototype.assignment = function() {
  var left = this.equality();
  if (this.expect('=')) {
    var right = this.equality();
    return {type: AST.AssignmentExpression, left: left, right: right};
  }
  return left;
};
```

We also 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
      });
      this.index++;
    } else if (this.isIdent(this.ch)) {
      this.readIdent();
    } else if (this.isWhitespace(this.ch)) {
      this.index++;
    } else {
      var ch = this.ch;
      var ch2 = this.ch + this.peek();
      var ch3 = this.ch + this.peek() + this.peek(2);
      var op = OPERATORS[ch];
      var op2 = OPERATORS[ch2];
      var op3 = OPERATORS[ch3];
      if (op || op2 || op3) {
        var token = op3 ? ch3 : (op2 ? ch2 : ch);
        this.tokens.push({text: token});
        this.index += token.length;
      } 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:

src/parse.js

```
var OPERATORS = {
  '+': true,
  '-': true,
  '!': true,
  '*': true,
  '/': true,
  '%': true,
  '=': true,
  '==': true,
  '!=': true,
  '===': true,
  '!==': true,
  '<': true,
  '>': true,
  '<=': true,
  '>=': true
};
```

The single equals sign is now emitted as an operator token instead of a text token. It still gets built into an assignment node, since both kinds of tokens have the text attribute with the value of `=`, which is what the AST builder is interested in.

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 two more entries:

src/parse.js

```
var OPERATORS = {
  '+': true,
  '-': true,
  '!': true,
  '*': true,
  '/': true,
  '%': true,
```



```
  '=': true,
  '==': true,
  '!=': true,
  '===': true,
  '!==': true,
  '<': true,
  '>': true,
  '<=': true,
  '>=': true,
  '&&': true,
  '||': true
};
```

In the AST builder we have two new functions that build the operators as `LogicalExpression` nodes - one for OR and one for AND:

src/parse.js

```
AST.prototype.logicalOR = function() {
  var left = this.logicalAND();
  var token;
  while ((token = this.expect('||'))) {
    left = {
      type: AST.LogicalExpression,
      left: left,
      operator: token.text,
      right: this.logicalAND()
    };
  }
  return left;
};
AST.prototype.logicalAND = function() {
  var left = this.equality();
  var token;
  while ((token = this.expect('&&'))) {
    left = {
      type: AST.LogicalExpression,
      left: left,
      operator: token.text,
      right: this.equality()
    };
  }
  return left;
};
```

The `LogicalExpression` type is new:

src/parse.js


```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
AST.MemberExpression = 'MemberExpression';
AST.CallExpression = 'CallExpression';
AST.AssignmentExpression = 'AssignmentExpression';
AST.UnaryExpression = 'UnaryExpression';
AST.BinaryExpression = 'BinaryExpression';
AST.LogicalExpression = 'LogicalExpression';
```

Once again, since we're going from the operators with higher precedence downward, these operators are inserted to the building chain right after `assignment`:

src/parse.js

```
AST.prototype.assignment = function() {
  var left = this.logicalOR();
  if (this.expect('=')) {
    var right = this.logicalOR();
    return {type: AST.AssignmentExpression, left: left, right: right};
  }
  return left;
};
```

In the AST compiler we have a new branch for `AST.LogicalExpression` that first recurses the left hand side argument and stores its value as the result of the whole expression:

src/parse.js

```
case AST.LogicalExpression:
  intoId = this.nextId();
  this.state.body.push(this.assign(intoId, this.recurse(ast.left)));
  return intoId;
```

It then generates a condition that evaluates the right hand side argument *if* the left hand side was truthy (in the case of `&&`) or falsy (in the case of `||`). If the right hand side is evaluated, its value becomes the value of the whole expression:

src/parse.js

```
case AST.LogicalExpression:
  intoId = this.nextId();
  this.state.body.push(this.assign(intoId, this.recurse(ast.left)));
  this.if_(ast.operator === '&&' ? intoId : this.not(intoId),
    this.assign(intoId, this.recurse(ast.right)));
  return intoId;
```

And there we have implemented both `&&` and `||` in terms of `if`! This special behavior is why we didn't treat AND and OR as `BinaryExpression` nodes even though they're technically binary expressions.

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');
});
```


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 `:` - its presence is more convenient to detect in the AST building phase.

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 `AST` we'll introduce a new function `ternary` that builds this operator. It consumes the three operands as well as the two parts of the operator itself, and emits a `ConditionalExpression` node:

src/parse.js

```
AST.prototype.ternary = function() {
  var test = this.logicalOR();
  if (this.expect('?')) {
    var consequent = this.assignment();
    if (this.consume(':')) {
      var alternate = this.assignment();
      return {
        type: AST.ConditionalExpression,
        test: test,
        consequent: consequent,
        alternate: alternate
      };
    }
  }
  return test;
};
```

Note that the "middle" and "right" expressions may be any expressions, since we consume them as assignments. Also note that while the method does have a fallback to `logicalOR`, once the consequent part `?` of the operator has been detected, the alternate `:` part is required and doesn't have a fallback. That's because we use `consume` for it, which throws an exception when not matched.

The `ConditionalExpression` type needs to be introduced:

src/parse.js


```
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
AST.MemberExpression = 'MemberExpression';
AST.CallExpression = 'CallExpression';
AST.AssignmentExpression = 'AssignmentExpression';
AST.UnaryExpression = 'UnaryExpression';
AST.BinaryExpression = 'BinaryExpression';
AST.LogicalExpression = 'LogicalExpression';
AST.ConditionalExpression = 'ConditionalExpression';
```

And once again we change the next operator looked at from `assignment`, now to `ternary`:

src/parse.js

```
AST.prototype.assignment = function() {
  var left = this.ternary();
  if (this.expect('=')) {
    var right = this.ternary();
    return {type: AST.AssignmentExpression, left: left, right: right};
  }
  return left;
};
```

As the `ConditionalExpression` is compiled, it first stores the value of the test expression in a variable:

src/parse.js

```
case AST.ConditionalExpression:
  var testId = this.nextId();
  this.state.body.push(this.assign(testId, this.recurse(ast.test)));
```

It then executes either the consequent or the alternate expression based on the value of the test expression. One of the former is then returned as the value of the expression:

src/parse.js

```
case AST.ConditionalExpression:
  intoId = this.nextId();
  var testId = this.nextId();
  this.state.body.push(this.assign(testId, this.recurse(ast.test)));
  this.if_(testId,
    this.assign(intoId, this.recurse(ast.consequent)));
  this.if_(this.not(testId),
    this.assign(intoId, this.recurse(ast.alternate)));
  return intoId;
```


The final precedence order of the operators can be read by looking at the order in which the AST builder's methods 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 building 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

```
AST.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 if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    primary = this.constants[this.consume().text];
  } else if (this.peek().identifier) {
    primary = this.identifier();
  } else {
    primary = this.constant();
  }
  var next;
  while ((next = this.expect('.', '[', '('))) {
    if (next.text === '[') {
      primary = {
        type: AST.MemberExpression,
        object: primary,
        property: this.primary(),
        computed: true
      };
      this.consume(']');
    } else if (next.text === '.') {
      primary = {
        type: AST.MemberExpression,
        object: primary,
        property: this.identifier(),
        computed: false
      };
    } else if (next.text === '(') {
      primary = {
        type: AST.CallExpression,
        callee: primary,
        arguments: this.parseArguments()
      };
      this.consume(')');
    }
  }
  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 AST builder. Let's add it to the growing collection of text token characters in `Lexer.lex`:

src/parse.js

```
    } else if (this.is('[],{}:.()?;')) {
```

In the AST builder, we're going to change the nature of the `AST.Program` node type so that its `body` is no longer just a single expression, but an array of expressions. We form the body array by consuming expressions as long as we're able to match semicolons between them:

src/parse.js

```
AST.prototype.program = function() {
  var body = [];
  while (true) {
    if (this.tokens.length) {
      body.push(this.assignment());
    }
    if (!this.expect(';')) {
      return {type: AST.Program, body: body};
    }
  }
};
```


When an expression has no semicolons, which is the most common case, the `body` array will hold exactly one item when the loop is done.

At compilation time, all but the last statements in the body are first generated, each one terminating in a semicolon. Then, a `return` statement for the last statement in the body is generated:

src/parse.js

```
case AST.Program:
  _.forEach(_.initial(ast.body), function(stmt) {
    this.state.body.push(this.recurse(stmt), ';');
  }, this);
  this.state.body.push('return ', this.recurse(_.last(ast.body)), ';');
  break;
```

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 AST builder
- How the precedence order may be altered with parentheses
- 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 complete 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

The expression language we have implemented has one remaining feature that we need to cover: Filters.

Filters are all about processing the value of an expression in order to turn it into something else. When you apply a filter to an expression, a filter function is called with the expression value, and the return value of that call becomes the final value of the expression.

You can apply a filter by adding a Unix-style pipe character and the name of a filter to your expression string. For example, Angular's built-in uppercase filter takes a string and converts it into uppercase:

```
myExpression | uppercase
```

Filter may be composed, by applying several of them back to back:

```
myNumbers | odd | increment
```

The key thing to understand about filters is that they're really nothing but plain functions. They take the value of the input expression, and return another value, which will become the value of the output expression (or the input of the next filter). The expression above is roughly equivalent to:

```
increment(odd(myNumbers))
```

The main difference between filters and plain functions is that you don't have to put filters on the Scope in order to invoke them. Instead, you register them into your Angular application and then just use them wherever you need them. When you use the pipe operator, Angular will find the corresponding filter from its filter registry.

Unlike all the expression features we've seen so far, filters are not something that exists in the JavaScript language. In JavaScript the pipe operator `|` denotes a bitwise OR operation, but the Angular expression language uses the same syntax for filtering instead.

In this chapter we'll implement filter support into our expression system. We will also implement one of the filters that Angular ships with: The surprisingly versatile, and interestingly named `filter` filter.

Angular ships with a number of other built-in filters, such as ones for formatting numbers and currencies. We will not be implementing the full filter suite, but if any of the other built-in filters interests you, the source code should be easy enough to follow.

Filter Registration

Before filters can be usd in expressions, they need to be registered somewhere. Angular ships with a special *filter service* for this purpose. It supplies functionality both for registering filters and for looking up filters that have been previously registered.

At this point we're going to implement a simple version of the service using a couple of standalone functions, `register` and `filter`. Later on we will further develop this implementation so that it is fully integrated with the dependency injection system, which is the topic of the next part of the book.

First of all, the filter service allows registering filters. It is done by calling the registration function with the name of the filter and a *factory function*. The factory function is expected to the return the filter. Later, the registered filter can be obtained using the `filter` function. The test for this goes into a new test file:

test/filter_spec.js

```
/* jshint globalstrict: true */
/* global filter: false, register: false */
'use strict';

describe("filter", function() {

  it('can be registered and obtained', function() {
    var myFilter = function() { };
    var myFilterFactory = function() {
      return myFilter;
    };
    register('my', myFilterFactory);
    expect(filter('my')).toBe(myFilter);
  });

});
```

We can implement these two functions in a straightforward way. They both access a storage object whose keys are filter names and values are filters. When a filter is registered, the return value of the factory function is placed in the object. The implementation for this also goes into a new file:

src/filter.js


```
var filters = {};

function register(name, factory) {
  var filter = factory();
  filters[name] = filter;
  return filter;
}

function filter(name) {
  return filters[name];
}
```

The registration function also supports a shorthand for registering several filters in a single call. You can do it by passing in an object where the keys are filter names and the values are the corresponding filter factories:

test/filter_spec.js

```
it('allows registering multiple filters with an object', function() {
  var myFilter = function() { };
  var myOtherFilter = function() { };
  register({
    my: function() {
      return myFilter;
    },
    myOther: function() {
      return myOtherFilter;
    }
  });

  expect(filter('my')).toBe(myFilter);
  expect(filter('myOther')).toBe(myOtherFilter);
});
```

In the implementation, if the first argument is an object, we recursively invoke `register` for each key-value pair in the object:

src/filter.js

```
function register(name, factory) {
  if (_.isObject(name)) {
    return _.map(name, function(factory, name) {
      return register(name, factory);
    });
  } else {
    var filter = factory();
    filters[name] = filter;
    return filter;
  }
}
```


This will do for the implementation of the filter service itself for now. As mentioned already, we will return to it once we have a dependency injection system.

Filter Expressions

We're now all set to start looking at the implementation of filters in the expression parser. What we expect to get is the ability to use pipe characters in expressions in order to modify the expression's value. After the pipe we should be able to provide the name of a filter that we have previously registered, which will then get called with the input value:

test/parse_spec.js

```
it('can parse filter expressions', function() {
  register('upcase', function() {
    return function(str) {
      return str.toUpperCase();
    };
  });
  var fn = parse('aString | upcase');
  expect(fn({aString: 'Hello'})).toEqual('HELLO');
});
```

We also need to let JSHint know that we are purposefully calling the global `register` function in our tests:

test/parse_spec.js

```
/* global parse: false, register: false */
```

We are going to handle the pipe as an operator expression, and thus we'll begin by adding it to the list of operators supported by the Lexer:

src/parse.js

```
var OPERATORS = {
  '+': true,
  '-': true,
  '!': true,
  '*': true,
  '/': true,
  '%': true,
  '=': true,
  '==': true,
  '!=': true,
```



```
  '===': true,
  '!==': true,
  '<': true,
  '>': true,
  '<=': true,
  '>=': true,
  '&&': true,
  '||': true,
  '|': true
};
```

This causes a token to be emitted by the Lexer when it encouters the single pipe character.

Next, we are going to build up an AST node for filter expressions. They're handled by a new AST builder method called `filter`. It first consumes an assignment expression (or some higher-precedence expression) as the left hand side, and then sees if there's a pipe character following it:

src/parse.js

```
AST.prototype.filter = function() {
  var left = this.assignment();
  if (this.expect('|')) {

  }
  return left;
};
```

If there is a pipe, a `CallExpression` node is created. The callee will be the filter name, which we consume as an identifier node. The sole argument to the call will be the left hand side consumed earlier:

src/parse.js

```
AST.prototype.filter = function() {
  var left = this.assignment();
  if (this.expect('|')) {
    left = {
      type: AST.CallExpression,
      callee: this.identifier(),
      arguments: [left]
    };
  }
  return left;
};
```

Earlier we discussed that filters are really just function calls. Here we see it in concrete form: We have a call expression where the function is the filter, and the argument to the function is the expression to the left of the filter.

We do need to do some work in the AST compiler as well, but before we get to it, let's add `filter` to the chain of calls in the AST builder. We can already see how `filter` falls back to `assignment`, which hints that `filter` has lower precedence than `assignment`. This is in fact the case - filter expressions are the expressions with the lowest precedence of all. Thus, they are the first thing we invoke when consuming an expression statement:

src/parse.js

```
AST.prototype.program = function() {
  var body = [];
  while (true) {
    if (this.tokens.length) {
      body.push(this.filter());
    }
    if (!this.expect(';')) {
      return {type: AST.Program, body: body};
    }
  }
};
```

Filters are also the first thing we try to parse when precedence is re-set using parentheses:

src/parse.js

```
AST.prototype.primary = function() {
  var primary;
  if (this.expect('(')) {
    primary = this.filter();
    this.consume(')');
  } else if (this.expect('[')) {
    primary = this.arrayDeclaration();
  } else if (this.expect('{')) {
    primary = this.object();
  } else if (this.constants.hasOwnProperty(this.tokens[0].text)) {
    primary = this.constants[this.consume().text];
  } else if (this.peek().identifier) {
    primary = this.identifier();
  } else {
    primary = this.constant();
  }
  // ...
};
```


Before the AST compiler will be able to do something useful with the `CallExpression`, it will need to know that it is specifically a *filter* `CallExpression`. That's because in normal call expressions, the function or method to be called (the callee) is expected to be on the Scope, which won't be the case for filters. Let's add a `filter` boolean flag to the AST node to let the compiler know about this:

src/parse.js

```
AST.prototype.filter = function() {
  var left = this.assignment();
  if (this.expect('|')) {
    left = {
      type: AST.CallExpression,
      callee: this.identifier(),
      arguments: [left],
      filter: true
    };
  }
  return left;
};
```

In the compiler we can now introduce an `if` block in which we can handle the filter call separately from other call expressions:

src/parse.js

```
case AST.CallExpression:
  var callContext, callee, args;
  if (ast.filter) {

  } else {
    callContext = {};
    callee = this.recurse(ast.callee, callContext);
    args = _.map(ast.arguments, function(arg) {
      return 'ensureSafeObject(' + this.recurse(arg) + ')';
    }, this);
    if (callContext.name) {
      this.addEnsureSafeObject(callContext.context);
      if (callContext.computed) {
        callee = this.computedMember(callContext.context, callContext.name);
      } else {
        callee = this.nonComputedMember(callContext.context, callContext.name);
      }
    }
    this.addEnsureSafeFunction(callee);
    return callee + '&&ensureSafeObject(' + callee + '(' + args.join(',') + '))';
  }
  break;
```


What we're going to do here is use a new helper method called `filter` to obtain the JavaScript we need for the filter function. Then we'll resolve the arguments using `recurse`, and return a piece of code that combines the function and the arguments:

src/parse.js

```
case AST.CallExpression:
  var callContext, callee, args;
  if (ast.filter) {
    callee = this.filter(ast.callee.name);
    args = _.map(ast.arguments, function(arg) {
      return this.recurse(arg);
    }, this);
    return callee + '(' + args + ')';
  } else {
    // ...
  }
  break;
```

We are now expecting the `filter` method of the compiler to return an expression, which at runtime evaluates to the filter function the user wanted to apply. We know we can get it using the filter service, but how do we embed it to the JavaScript generated for the expression?

Firstly, we'll expect the filter function to be available at runtime in some variable. Let's just generate a variable and return it:

src/parse.js

```
ASTCompiler.prototype.filter = function(name) {
  var filterId = this.nextId();
  return filterId;
};
```

This variable will naturally be `undefined` as we haven't written anything that would assign something to it. We need to somehow assign the filter function to it. Specifically, we'll need to generate some code that gets the filter from the filter service and puts it in this variable *at runtime*.

We will need to keep track of which filters have been used in the expression before we can make them available. We can store that information in the compiler state, in a new attribute:

src/parse.js


```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {
    body: [],
    nextId: 0,
    vars: [],
    filters: {}
  };
  // ...
};
```

When `filter` is called, we'll now store information about it in the state object. We can use the filter name as the key and the variable name that the filter should be assigned to as the value:

src/parse.js

```
ASTCompiler.prototype.filter = function(name) {
  var filterId = this.nextId();
  this.state.filters[name] = filterId;
  return filterId;
};
```

If the filter has already been used previously, we should reuse the variable name generated the last time instead of generating a new one:

src/parse.js

```
ASTCompiler.prototype.filter = function(name) {
  if (!this.state.filters.hasOwnProperty('name')) {
    this.state.filters[name] = this.nextId();
  }
  return this.state.filters[name];
};
```

At this point, once the AST has been recursed, `state.filters` will contain all the filters that were used in the expression. Now we should generate the code that populates the filter variables. For that we need to have the `filter` service at runtime, so let's pass it in to the generated function, like we've done for several other functions earlier:

src/parse.js

```
/* jshint -W054 */
return new Function(
  'ensureSafeMemberName',
  'ensureSafeObject',
  'ensureSafeFunction',
  'ifDefined',
  'filter',
  fnString)(
    ensureSafeMemberName,
    ensureSafeObject,
    ensureSafeFunction,
    ifDefined,
    filter);
/* jshint +W054 */
```

JSHint isn't happy about using `filter`, so we need to add an exclusion rule for it at the top of `parse.js`:

src/parse.js

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

This, again, is something we will no longer need to have once dependency injection is in place.

The JavaScript code for the filter lookup will be the very first thing we generate into the function. We'll do it with a helper function called `filterPrefix`:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {
    body: [],
    nextId: 0,
    vars: [],
    filters: {}
  };
  this.recurse(ast);
  var fnString = this.filterPrefix() +
    'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) +
    this.state.body.join('') +
    '}; return fn;';
  // ...
};
```


This method returns an empty string if no filters were applied in the expression:

src/parse.js

```
ASTCompiler.prototype.filterPrefix = function() {
  if (_.isEmpty(this.state.filters)) {
    return '';
  } else {

  }
};
```

If there were filters, the method will build up a collection of variable initializations and generate a `var` statement for them:

src/parse.js

```
ASTCompiler.prototype.filterPrefix = function() {
  if (_.isEmpty(this.state.filters)) {
    return '';
  } else {
    var parts = [];

    return 'var ' + parts.join(',') + ';';
  }
};
```

For each filter used, we emit the variable name generated earlier in `ASTCompiler.prototype.filter` and an initialization that looks up the filter using the `filter` service, which we now have access to in the generated code:

src/parse.js

```
ASTCompiler.prototype.filterPrefix = function() {
  if (_.isEmpty(this.state.filters)) {
    return '';
  } else {
    var parts = _.map(this.state.filters, function(varName, filterName) {
      return varName + '=' + 'filter(' + this.escape(filterName) + ')';
    }, this);
    return 'var ' + parts.join(',') + ';';
  }
};
```


There's one remaining issue here, which is that when we generated the variable names using `nextId`, we also caused them to be added into the `vars` state variable, because that's what `nextId` does. This means that they'll actually also be declared inside the expression function, which will *shadow* the filter variables we have just created. What essentially happens is that if we have an expression like this:

```
42 | increment
```

What gets generated is something like this:

```
function(ensureSafeMemberName, ensureSafeObject, ensureSafeFunction,
         ifDefined, filter) {
  var v0 = filter('increment');
  var fn = function(s, l) {
    var v0;
    return v0(42);
  };
  return fn;
}
```

That second `var v0` is the problem. What we can do is invoke `nextId` with a special flag that tells it to just generate the variable id and skip the declaration - because we're handling the declaration separately in `filterPrefix`:

src/parse.js

```
ASTCompiler.prototype.filter = function(name) {
  if (!this.state.filters.hasOwnProperty('name')) {
    this.state.filters[name] = this.nextId(true);
  }
  return this.state.filters[name];
};
```

In `nextId` we only put the generated id to `state.vars` if this flag is falsy, which it will be for everything except the variables generated from `filter`:

src/parse.js

```
ASTCompiler.prototype.nextId = function(skip) {
  var id = 'v' + (this.state.nextId++);
  if (!skip) {
    this.state.vars.push(id);
  }
  return id;
};
```

And now we have a passing test suite again. We can apply filters in expressions!

Filter Chain Expressions

An important aspect of filters is how you can compose them to form *filter chains*. That means you can apply an arbitrary number of filters back to back by just adding pipe characters to the expression:

test/parse_spec.js

```
it('can parse filter chain expressions', function() {
  register('upcase', function() {
    return function(s) {
      return s.toUpperCase();
    };
  });
  register('exclamate', function() {
    return function(s) {
      return s + '!';
    };
  });
  var fn = parse('"hello" | upcase | exclamate');
  expect(fn()).toEqual('HELLO!');
});
```

Right now we're just terminating the expression after the first filter.

We can actually quite simply just replace the `if` statement in `AST.prototype.filter` with a `while`. As long as we find pipe characters, we apply more filters. The result of one `CallExpression` becomes the argument of the next one:

src/parse.js

```
AST.prototype.filter = function() {
  var left = this.assignment();
  while (this.expect('|')) {
    left = {
      type: AST.CallExpression,
      callee: this.identifier(),
      arguments: [left],
      filter: true
    };
  }
  return left;
};
```


Additional Filter Arguments

The filters we have so far always have exactly one argument: The input expression's value. But filters can actually have additional arguments too. This is quite useful because you can then parameterize a single filter to behave differently in different situations.

For example, if you have a filter that repeats a string a number of times, you'll want to specify the number of times to repeat. You can do it by using a colon character and a number following the filter's name. The number becomes the *second* argument of the filter function:

test/parse_spec.js

```
it('can pass an additional argument to filters', function() {
  register('repeat', function() {
    return function(s, times) {
      return _.repeat(s, times);
    };
  });
  var fn = parse('"hello" | repeat:3');
  expect(fn()).toEqual('hellohellohello');
});
```

You can pass more arguments than just one, by just repeating the colon character for each additional argument:

test/parse_spec.js

```
it('can pass several additional arguments to filters', function() {
  register('surround', function() {
    return function(s, left, right) {
      return left + s + right;
    };
  });
  var fn = parse('"hello" | surround:"*":"!"');
  expect(fn()).toEqual('*hello!');
});
```

The colon character is already being emitted by the Lexer (since we've used it in the ternary operator). The AST compiler is also already all set to handle any number of filter arguments, since it emits the code for them in a loop. The only part that's missing is the building of the AST for these arguments.

We'll firstly pull out the argument array from `CallExpression` into a variable:

src/parse.js


```
AST.prototype.filter = function() {
  var left = this.assignment();
  while (this.expect('|')) {
    var args = [left];
    left = {
      type: AST.CallExpression,
      callee: this.identifier(),
      arguments: args,
      filter: true
    };
  }
  return left;
};
```

Now, we can add a loop that consumes colon characters, followed by arbitrary (non-filter) expressions as long as we find them. Each one will become an argument that we append to the `args` array:

src/parse.js

```
AST.prototype.filter = function() {
  var left = this.assignment();
  while (this.expect('|')) {
    var args = [left];
    left = {
      type: AST.CallExpression,
      callee: this.identifier(),
      arguments: args,
      filter: true
    };
    while (this.expect(':')) {
      args.push(this.assignment());
    }
  }
  return left;
};
```

This completes our parser's filter expression support!

The Filter Filter

Now that we have filter support, let's dedicate the remainder of the chapter to the implementation of one particular filter that Angular ships with: The *filter* filter.

In a nutshell, the purpose of the filter filter is to filter arrays that you use in expressions into some subset. You specify a criteria for the items that you want to match in an array,

and the result of the expression is another array of only the items that matched your criteria. It's a bit like querying arrays for items that match certain patterns.

The filter filter is often used together with `ngRepeat`, when you want to repeat a piece of DOM for items in an array, but limit it to only certain kinds of items instead of the whole array. But the filter is in no way limited to `ngRepeat` and can be used whenever you have an array in an expression.

Let's begin by asserting that the filter filter should in fact exist and be available through the filter service. This goes in a new test file:

test/filter_filter_spec.js

```
/* jshint globalstrict: true */
/* global filter: false, parse: false */
'use strict';

describe("filter filter", function() {

  it('is available', function() {
    expect(filter('filter')).toBeDefined();
  });

});
```

The factory function for the filter is introduced and registered in its own source file:

src/filter_filter.js

```
function filterFilter() {
  return function() {

  };
}

register('filter', filterFilter);
```

With the setup out of the way, let's start exploring what this filter should actually be able to do.

Filtering With Predicate Functions

The most simple way to use the filter filter - in terms of implementation - is to give it a reference to a predicate function. It will use that function to return another array with only the elements for which the predicate returns a truthy value:

test/filter_filter_spec.js


```
it('can filter an array with a predicate function', function() {
  var fn = parse('[1, 2, 3, 4] | filter:isOdd');
  var scope = {
    isOdd: function(n) {
      return n % 2 !== 0;
    }
  };
  expect(fn(scope)).toEqual([1, 3]);
});
```

The reason this is so simple to implement is that we can simply delegate to the LoDash filter function. It takes an array and a predicate function and returns a new array:

src/filter_filter.js

```
function filterFilter() {
  return function(array, filterExpr) {
    return _.filter(array, filterExpr);
  };
}
```

Filtering With Strings

Having to set up a predicate function each time you want to use the filter is not hugely convenient from the application developer's perspective. That's why the filter provides a number of conveniences for getting around that requirement. For instance, you can just give the filter a string, and it will filter the input array for items that match that string:

test/filter_filter_spec.js

```
it('can filter an array of strings with a string', function() {
  var fn = parse('arr | filter:"a"');
  expect(fn({arr: ["a", "b", "a"]})).toEqual(['a', 'a']);
});
```

We need to start checking the type of the filter expression given. If it's a function, we use it as a predicate like before, but if it's a string, we need to *create* a predicate function. If the filter expression is something we don't recognize, we just return the input array as-is since we don't know how we should filter it:

src/filter_filter.js

```
function filterFilter() {
  return function(array, filterExpr) {
    var predicateFn;
    if (_.isFunction(filterExpr)) {
      predicateFn = filterExpr;
    } else if (_.isString(filterExpr)) {
      predicateFn = createPredicateFn(filterExpr);
    } else {
      return array;
    }
    return _.filter(array, predicateFn);
  };
}
```

For now, we can just create a predicate function that compares each item to the input expression string using strict equality comparison:

src/filter_filter.js

```
function createPredicateFn(expression) {
  return function predicateFn(item) {
    return item === expression;
  };
}
```

In actuality the filter is not this strict though. It actually matches any string that *contains* the given input string:

test/filter_filter_spec.js

```
it('filters an array of strings with substring matching', function() {
  var fn = parse('arr | filter:"o"');
  expect(fn({arr: ['quick', 'brown', 'fox']})).toEqual(['brown', 'fox']);
});
```

We can change our predicate to check if each item contains the input expression:

src/filter_filter.js

```
function createPredicateFn(expression) {
  return function predicateFn(item) {
    return item.indexOf(expression) !== -1;
  };
}
```

The filter also does its comparisons in a case-insensitive manner:

test/filter_filter_spec.js

```
it('filters an array of strings ignoring case', function() {
  var fn = parse('arr | filter:"o"');
  expect(fn({arr: ['quick', 'BROWN', 'fox']})).toEqual(['BROWN', 'fox']);
});
```

We'll convert both the expression and each item into lower case before checking if there is a match:

src/filter_filter.js

```
function createPredicateFn(expression) {
  return function predicateFn(item) {
    var actual = item.toLowerCase();
    var expected = expression.toLowerCase();
    return actual.indexOf(expected) !== -1;
  };
}
```

What's even more interesting is that when you have an array of objects as the input and a string filter, *any values inside the objects will be matched.* This means we can also filter non-primitives:

test/filter_filter_spec.js

```
it('filters an array of objects where any value matches', function() {
  var fn = parse('arr | filter:"o"');
  expect(fn({arr: [
    {firstName: 'John', lastName: 'Brown'},
    {firstName: 'Jane', lastName: 'Fox'},
    {firstName: 'Mary', lastName: 'Quick'}
  ]})).toEqual([
    {firstName: 'John', lastName: 'Brown'},
    {firstName: 'Jane', lastName: 'Fox'}
  ]);
});
```

Before making this work, let's refactor the current implementation in order to separate some concerns. The responsibility of comparing two values can be extracted to its own `comparator` function:

src/filter_filter.js


```
function createPredicateFn(expression) {

  function comparator(actual, expected) {
    actual = actual.toLowerCase();
    expected = expected.toLowerCase();
    return actual.indexOf(expected) !== -1;
  }

  return function predicateFn(item) {
    return comparator(item, expression);
  };
}
```

Now we can introduce another function that takes an actual and expected value and a comparator. It knows how to "deeply compare" the values so that if the actual value is an object, it looks into that object and returns `true` if any value inside matches. Otherwise it just uses the comparator directly:

src/filter_filter.js

```
function deepCompare(actual, expected, comparator) {
  if (_.isObject(actual)) {
    return _.some(actual, function(value) {
      return comparator(value, expected);
    });
  } else {
    return comparator(actual, expected);
  }
}
```

If the predicate function now delegates to this new helper, our test starts passing:

src/filter_filter.js

```
function createPredicateFn(expression) {

  function comparator(actual, expected) {
    actual = actual.toLowerCase();
    expected = expected.toLowerCase();
    return actual.indexOf(expected) !== -1;
  }

  return function predicateFn(item) {
    return deepCompare(item, expression, comparator);
  };
}
```

We have now split the predicate's work into three functions:

- `comparator` compares two primitive values
- `deepCompare` compares two primitive values or an object of primitives to a primitive
- `predicateFn` weaves `deepCompare` and `comparator` together to form the final filter predicate.

The filter should also be able to recurse into *nested* objects into arbitrary depths:

test/filter_filter_spec.js

```
it('filters an array of objects where a nested value matches', function() {
  var fn = parse('arr | filter:"o"');
  expect(fn({arr: [
    {name: {first: 'John', last: 'Brown'}},
    {name: {first: 'Jane', last: 'Fox'}},
    {name: {first: 'Mary', last: 'Quick'}}
  ]})).toEqual([
    {name: {first: 'John', last: 'Brown'}},
    {name: {first: 'Jane', last: 'Fox'}}
  ]);
});
```

This goes for arrays too. If we have an array of arrays, the filter returns all the nested arrays in which anything was matched:

test/filter_filter_spec.js

```
it('filters an array of arrays where a nested value matches', function() {
  var fn = parse('arr | filter:"o"');
  expect(fn({arr: [
    [{name: 'John'}, {name: 'Mary'}],
    [{name: 'Jane'}]
  ]})).toEqual([
    [{name: 'John'}, {name: 'Mary'}]
  ]);
});
```

This can be enabled by turning `deepCompare` into a recursive function. All values inside objects (and arrays) are again given to `deepCompare`, and `compare` is only called when a leaf level primitive is seen:

src/filter_filter.js

```
function deepCompare(actual, expected, comparator) {
  if (_.isObject(actual)) {
    return _.some(actual, function(value) {
      return deepCompare(value, expected, comparator);
    });
  } else {
    return comparator(actual, expected);
  }
}
```


Filtering With Other Primitives

The filter expression given to the filter need not be a string. It may also be a number:

test/filter_filter_spec.js

```
it('filters with a number', function() {
  var fn = parse('arr | filter:42');
  expect(fn({arr: [
    {name: 'Mary', age: 42},
    {name: 'John', age: 43},
    {name: 'Jane', age: 44}
  ]})).toEqual([
    {name: 'Mary', age: 42}
  ]);
});
```

Or a boolean:

test/filter_filter_spec.js

```
it('filters with a boolean value', function() {
  var fn = parse('arr | filter:true');
  expect(fn({arr: [
    {name: 'Mary', admin: true},
    {name: 'John', admin: true},
    {name: 'Jane', admin: false}
  ]})).toEqual([
    {name: 'Mary', admin: true},
    {name: 'John', admin: true}
  ]);
});
```

We will create predicate functions for these types of expressions too:

src/filter_filter.js

```
function filterFilter() {
  return function(array, filterExpr) {
    var predicateFn;
    if (_.isFunction(filterExpr)) {
      predicateFn = filterExpr;
    } else if (_.isString(filterExpr) ||
               _.isNumber(filterExpr) ||
               _.isBoolean(filterExpr)) {
      predicateFn = createPredicateFn(filterExpr);
    } else {
      return array;
    }
    return _.filter(array, predicateFn);
  };
}
```


Then we'll just coerce them into strings inside the comparator:

src/filter_filter.js

```
function comparator(actual, expected) {
  actual = ('' + actual).toLowerCase();
  expected = ('' + expected).toLowerCase();
  return actual.indexOf(expected) !== -1;
}
```

It is notable that filtering with a number (or a boolean) does not mean that numeric equality is required of the items. Everything is turned into a string, and thus even string items that happen to *contain* the given number will match, as the following (passing) test case illustrates:

test/filter_filter_spec.js

```
it('filters with a substring numeric value', function() {
  var fn = parse('arr | filter:42');
  expect(fn({arr: ['contains 42']})).toEqual(['contains 42']);
});
```

You can also filter `null` values. When you do, only items that are `null` are returned. In this case, strings that contain the substring `'null'` are *not* matched:

test/filter_filter_spec.js

```
it('filters matching null', function() {
  var fn = parse('arr | filter:null');
  expect(fn({arr: [null, 'not null']})).toEqual([null]);
});
```

Flipping things around, if you filter with the string `'null'`, values that are actually `null` will *not* be matched. Only strings:

test/filter_filter_spec.js

```
it('does not match null value with the string null', function() {
  var fn = parse('arr | filter:"null"');
  expect(fn({arr: [null, 'not null']})).toEqual(['not null']);
});
```

We should create a predicate for a `null` expression too:

src/filter_filter.js


```
function filterFilter() {
  return function(array, filterExpr) {
    var predicateFn;
    if (_.isFunction(filterExpr)) {
      predicateFn = filterExpr;
    } else if (_.isString(filterExpr) ||
               _.isNumber(filterExpr) ||
               _.isBoolean(filterExpr) ||
               _.isNull(filterExpr)) {
      predicateFn = createPredicateFn(filterExpr);
    } else {
      return array;
    }
    return _.filter(array, predicateFn);
  };
}
```

In the comparator we'll introduce a special case for `null`. If either of the actual or expected values is `null`, the other one needs to be `null` too if it is to be considered a match:

src/filter_filter.js

```
function comparator(actual, expected) {
  if (_.isNull(actual) || _.isNull(expected)) {
    return actual === expected;
  }
  actual = ('' + actual).toLowerCase();
  expected = ('' + expected).toLowerCase();
  return actual.indexOf(expected) !== -1;
}
```

When there are `undefined` values in the array, they should not be matched to the string `'undefined'`:

test/filter_filter_spec.js

```
it('does not match undefined values', function() {
  var fn = parse('arr | filter:"undefined"');
  expect(fn({arr: [undefined, 'undefined']})).toEqual(['undefined']);
});
```

The rule here is that an `undefined` item never passes the filter:

src/filter_filter.js


```
function comparator(actual, expected) {
  if (_.isUndefined(actual)) {
    return false;
  }
  if (_.isNull(actual) || _.isNull(expected)) {
    return actual === expected;
  }
  actual = ('' + actual).toLowerCase();
  expected = ('' + expected).toLowerCase();
  return actual.indexOf(expected) !== -1;
}
```

Negated Filtering With Strings

It is often useful to filter an array for items that *don't* match a criterion, instead of filtering for items that do. You can do that with string filters by prefixing the string with `!`:

test/filter_filter_spec.js

```
it('allows negating string filter', function() {
  var fn = parse('arr | filter:"!o"');
  expect(fn({arr: ['quick', 'brown', 'fox']})).toEqual(['quick']);
});
```

The implementation for this is quite simple. When we get to `deepCompare` with a string criterion that begins with `!`, we call ourselves again with a string that doesn't have the `!` and negate the result:

src/filter_filter.js

```
function deepCompare(actual, expected, comparator) {
  if (_.isString(expected) && _.startsWith(expected, '!')) {
    return !deepCompare(actual, expected.substring(1), comparator);
  }
  if (_.isObject(actual)) {
    return _.some(actual, function(value, key) {
      return deepCompare(value, expected, comparator);
    });
  } else {
    return comparator(actual, expected);
  }
}
```


Filtering With Object Criteria

When you have an array of objects to filter, just using a primitive value as the filter criteria may be too blunt an instrument. For example, you may want to filter items where a specific attribute has a specific value.

You can do this by providing *an object* as the filter criteria. Here is one that looks for matches in the `name` attributes of the items in the input array. The items also have `role` attributes but the filter ignores them:

test/filter_filter_spec.js

```
it('filters with an object', function() {
  var fn = parse('arr | filter:{name: "o"}');
  expect(fn({arr: [
    {name: 'Joe', role: 'admin'},
    {name: 'Jane', role: 'moderator'}
  ]})).toEqual([
    {name: 'Joe', role: 'admin'}
  ]);
});
```

When you specify several criteria in the filter object, only items that match *all* of the criteria are returned:

test/filter_filter_spec.js

```
it('must match all criteria in an object', function() {
  var fn = parse('arr | filter:{name: "o", role: "m"}');
  expect(fn({arr: [
    {name: 'Joe', role: 'admin'},
    {name: 'Jane', role: 'moderator'}
  ]})).toEqual([
    {name: 'Joe', role: 'admin'}
  ]);
});
```

It follows from this that when the criteria object is empty, everything passes the filter:

test/filter_filter_spec.js

```
it('matches everything when filtered with an empty object', function() {
  var fn = parse('arr | filter:{}');
  expect(fn({arr: [
    {name: 'Joe', role: 'admin'},
    {name: 'Jane', role: 'moderator'}
  ]})).toEqual([
    {name: 'Joe', role: 'admin'},
    {name: 'Jane', role: 'moderator'}
  ]);
});
```


The criteria object may also contain nested objects. This allows you to reach into the data to any depth to find matches:

test/filter_filter_spec.js

```
it('filters with a nested object', function() {
  var fn = parse('arr | filter:{name: {first: "o"}}');
  expect(fn({arr: [
    {name: {first: 'Joe'}, role: 'admin'},
    {name: {first: 'Jane'}, role: 'moderator'}
  ]})).toEqual([
    {name: {first: 'Joe'}, role: 'admin'}
  ]);
});
```

You can also negate a criterion inside the criteria object, by using a `!` prefix, just like we did with primitive string criteria:

test/filter_filter_spec.js

```
it('allows negation when filtering with an object', function() {
  var fn = parse('arr | filter:{name: {first: "!o"}}');
  expect(fn({arr: [
    {name: {first: 'Joe'}, role: 'admin'},
    {name: {first: 'Jane'}, role: 'moderator'}
  ]})).toEqual([
    {name: {first: 'Jane'}, role: 'moderator'}
  ]);
});
```

That provides a nice, initial test harness for what we now need to implement. First of all, let's set things up so that a predicate function is created for object filters:

src/filter_filter.js

```
function filterFilter() {
  return function(array, filterExpr) {
    var predicateFn;
    if (_.isFunction(filterExpr)) {
      predicateFn = filterExpr;
    } else if (_.isString(filterExpr) ||
               _.isNumber(filterExpr) ||
               _.isBoolean(filterExpr) ||
               _.isNull(filterExpr) ||
               _.isObject(filterExpr)) {
      predicateFn = createPredicateFn(filterExpr);
    } else {
      return array;
    }
    return _.filter(array, predicateFn);
  };
}
```


In `deepCompare`, if we have an object as the expected value, we can't compare the actual value to it directly. We'll do something else instead:

src/filter_filter.js

```
function deepCompare(actual, expected, comparator) {
  if (_.isString(expected) && _.startsWith(expected, '!')) {
    return !deepCompare(actual, expected.substring(1), comparator);
  }
  if (_.isObject(actual)) {
    if (_.isObject(expected)) {

    } else {
      return _.some(actual, function(value, key) {
        return deepCompare(value, expected, comparator);
      });
    }
  } else {
    return comparator(actual, expected);
  }
}
```

What we do is loop over the expected criteria object, and for each value *deep-compare the corresponding value in the actual object*. If all of the criteria in the object are matched, we have a match:

src/filter_filter.js

```
function deepCompare(actual, expected, comparator) {
  if (_.isString(expected) && _.startsWith(expected, '!')) {
    return !deepCompare(actual, expected.substring(1), comparator);
  }
  if (_.isObject(actual)) {
    if (_.isObject(expected)) {
      return _.every(
        _.toPlainObject(expected),
        function(expectedVal, expectedKey) {
          return deepCompare(actual[expectedKey], expectedVal, comparator);
        }
      );
    } else {
      return _.some(actual, function(value, key) {
        return deepCompare(value, expected, comparator);
      });
    }
  } else {
    return comparator(actual, expected);
  }
}
```


This also takes care of the nested criteria objects, because of the recursive invocation to `deepCompare` that will again check if the nested value is an object.

> We additionally use a toPlainObject call for the criteria. If the criteria happens to use prototypal inheritance, this flattens the inheritance so that all inherited properties are also checked.

If some values in the criteria object happen to be `undefined`, those criteria are ignored:

test/filter_filter_spec.js

```
it('ignores undefined values in expectation object', function() {
  var fn = parse('arr | filter:{name: thisIsUndefined}');
  expect(fn({arr: [
    {name: 'Joe', role: 'admin'},
    {name: 'Jane', role: 'moderator'}
  ]})).toEqual([
    {name: 'Joe', role: 'admin'},
    {name: 'Jane', role: 'moderator'}
  ]);
});
```

Undefined values in the expectation object are considered to always be matches:

src/filter_filter.js

```
return _.every(
  _.toPlainObject(expected),
  function(expectedVal, expectedKey) {
    if (_.isUndefined(expectedVal)) {
      return true;
    }
    return deepCompare(actual[expectedKey], expectedVal, comparator);
  }
);
```

If there are nested *arrays* inside objects, the objects are considered matches if *any* item in the nested array matches. The criteria object effectively "jumps" a level here, so that you don't need to do anything special to make it match items inside arrays as well:

test/filter_filter_spec.js


```
it('filters with a nested object in array', function() {
  var fn = parse('arr | filter:{users: {name: {first: "o"}}}');
  expect(fn({arr: [
    {users: [{name: {first: 'Joe'}, role: 'admin'},
             {name: {first: 'Jane'}, role: 'moderator'}]},
    {users: [{name: {first: 'Mary'}, role: 'admin'}]}
  ]})).toEqual([
    {users: [{name: {first: 'Joe'}, role: 'admin'},
             {name: {first: 'Jane'}, role: 'moderator'}]}
  ]);
});
```

We need to introduce a special case for this in `deepCompare`. When we see an array as the actual value, we recursively call `deepCompare` for each item and return `true` if any item matched:

src/filter_filter.js

```
function deepCompare(actual, expected, comparator) {
  if (_.isString(expected) && _.startsWith(expected, '!')) {
    return !deepCompare(actual, expected.substring(1), comparator);
  }
  if (_.isArray(actual)) {
    return _.any(actual, function(actualItem) {
      return deepCompare(actualItem, expected, comparator);
    });
  }
  if (_.isObject(actual)) {
    if (_.isObject(expected)) {
      return _.every(
        _.toPlainObject(expected),
        function(expectedVal, expectedKey) {
          if (_.isUndefined(expectedVal)) {
            return true;
          }
          return deepCompare(actual[expectedKey], expectedVal, comparator);
        }
      );
    } else {
      return _.some(actual, function(value, key) {
        return deepCompare(value, expected, comparator);
      });
    }
  } else {
    return comparator(actual, expected);
  }
}
```

The object criteria matching we've implemented is now quite flexible, but it turns out it's still a little bit too lenient. When there is a criteria like `{user: {name: 'Bob'}}`, we want it to only match objects that have a `user` attribute that in turn has a `name` attribute that contains `Bob`. We do not want to match `Bob` on any other level:

test/filter_filter_spec.js

```
it('filters with nested objects on the same level only', function() {
  var items = [{user: 'Bob'},
               {user: {name: 'Bob'}},
               {user: {name: {first: 'Bob', last: 'Fox'}}}];
  var fn = parse('arr | filter:{user: {name: "Bob"}}');
  expect(fn({arr: [
    {user: 'Bob'},
    {user: {name: 'Bob'}},
    {user: {name: {first: 'Bob', last: 'Fox'}}}
  ]})).toEqual([
    {user: {name: 'Bob'}}
  ]);
});
```

This test fails because we're also matching `{user: {name: {first: 'Bob', last: 'Fox'}}}` with our criteria. Why is this happening?

The reason is that once we traverse into the `name` attribute of both the expected and actual objects, the expected value becomes the primitive string `'Bob'`. As we saw when we implemented primitive string matching, `deepCompare` matches a primitive against all nested properties of the actual object. But in this case we don't want to do that. We want to match the primitive only at the level we are currently at.

We need to extend `deepCompare` so that it can be used either to match any property of the actual object, or just the one currently being inspected. This is controlled with a new argument `matchAnyProperty`. Only when it is `true` do we traverse into the actual object to check for matches. Otherwise we attempt a simple primitive comparison:

src/filter_filter.js

```
function deepCompare(actual, expected, comparator, matchAnyProperty) {
  if (_.isString(expected) && _.startsWith(expected, '!')) {
    return !deepCompare(actual, expected.substring(1), comparator);
  }
  if (_.isArray(actual)) {
    return _.any(actual, function(actualItem) {
      return deepCompare(actualItem, expected, comparator);
    });
  }
  if (_.isObject(actual)) {
    if (_.isObject(expected)) {
      return _.every(
        _.toPlainObject(expected),
        function(expectedVal, expectedKey) {
```



```
          if (_.isUndefined(expectedVal)) {
            return true;
          }
          return deepCompare(actual[expectedKey], expectedVal, comparator);
        }
      );
    } else if (matchAnyProperty) {
      return _.some(actual, function(value, key) {
        return deepCompare(value, expected, comparator);
      });
    } else {
      return comparator(actual, expected);
    }
  } else {
    return comparator(actual, expected);
  }
}
```

From the predicate function we now need to pass in `true` for this argument to restore the default behavior where we *do* want to match any properties:

src/filter_filter.js

```
return function predicateFn(item) {
  return deepCompare(item, expression, comparator, true);
};
```

We also need to maintain this argument in all the recursive calls we make inside `deepCompare`, with the exception of the one inside the `_.every` function where we specifically *don't* want to match all properties:

src/filter_filter.js

```
function deepCompare(actual, expected, comparator, matchAnyProperty) {
  if (_.isString(expected) && _.startsWith(expected, '!')) {
    return !deepCompare(actual, expected.substring(1),
                        comparator, matchAnyProperty);
  }
  if (_.isArray(actual)) {
    return _.any(actual, function(actualItem) {
      return deepCompare(actualItem, expected,
                         comparator, matchAnyProperty);
    });
  }
  if (_.isObject(actual)) {
    if (_.isObject(expected)) {
      return _.every(
        _.toPlainObject(expected),
```



```
        function(expectedVal, expectedKey) {
          if (_.isUndefined(expectedVal)) {
            return true;
          }
          return deepCompare(actual[expectedKey], expectedVal, comparator);
        }
      );
    } else if (matchAnyProperty) {
      return _.some(actual, function(value, key) {
        return deepCompare(value, expected,
                           comparator, matchAnyProperty);
      });
    } else {
      return comparator(actual, expected);
    }
  } else {
    return comparator(actual, expected);
  }
}
```

Filtering With Object Wildcards

If you want to say "I want *any* of the properties in an object to match this value", you can use a special wildcard key `$` in the criteria object:

test/filter_filter_spec.js

```
it('filters with a wildcard property', function() {
  var fn = parse('arr | filter:{$: "o"}');
  expect(fn({arr: [
    {name: 'Joe', role: 'admin'},
    {name: 'Jane', role: 'moderator'},
    {name: 'Mary', role: 'admin'}
  ]})).toEqual([
    {name: 'Joe', role: 'admin'},
    {name: 'Jane', role: 'moderator'}
  ]);
});
```

Unlike regular object properties, the wildcard property matches values in nested objects too - on any level:

test/filter_filter_spec.js

```
it('filters nested objects with a wildcard property', function() {
  var fn = parse('arr | filter:{$: "o"}');
  expect(fn({arr: [
    {name: {first: 'Joe'}, role: 'admin'},
    {name: {first: 'Jane'}, role: 'moderator'},
    {name: {first: 'Mary'}, role: 'admin'}
  ]})).toEqual([
    {name: {first: 'Joe'}, role: 'admin'},
    {name: {first: 'Jane'}, role: 'moderator'}
  ]);
});
```

So far this doesn't really differ from using a simple primitive filter instead. Why use `{$: "o"}` and not just `"o"`? The main reason is when you nest a wildcard inside another object criterion. This scopes the wildcard inside its parent:

test/filter_filter_spec.js

```
it('filters wildcard properties scoped to parent', function() {
  var fn = parse('arr | filter:{name: {$: "o"}}');
  expect(fn({arr: [
    {name: {first: 'Joe', last: 'Fox'}, role: 'admin'},
    {name: {first: 'Jane', last: 'Quick'}, role: 'moderator'},
    {name: {first: 'Mary', last: 'Brown'}, role: 'admin'}
  ]})).toEqual([
    {name: {first: 'Joe', last: 'Fox'}, role: 'admin'},
    {name: {first: 'Mary', last: 'Brown'}, role: 'admin'}
  ]);
});
```

As we are traversing the contents of an expectation object, we should check if the key is `$`. If it is, we want to match *the whole* actual object against it, *not* just the matching key in it (which would require the actual object to include a `$` key):

src/filter_filter.js

```
return _.every(
  _.toPlainObject(expected),
  function(expectedVal, expectedKey) {
    if (_.isUndefined(expectedVal)) {
      return true;
    }
    var isWildcard = (expectedKey === '$');
    var actualVal = isWildcard ? actual : actual[expectedKey];
    return deepCompare(actualVal, expectedVal, comparator);
  }
);
```

Additionally, in this case we *do* want to match any property inside the actual object. It is precisely what the wildcard is for. Thus, we need to pass in the fourth argument to the recusive `deepCompare` call:

src/filter_filter.js

```
return _.every(
  _.toPlainObject(expected),
  function(expectedVal, expectedKey) {
    if (_.isUndefined(expectedVal)) {
      return true;
    }
    var isWildcard = (expectedKey === '$');
    var actualVal = isWildcard ? actual : actual[expectedKey];
    return deepCompare(actualVal, expectedVal, comparator, isWildcard);
  }
);
```

When you use a wildcard criterion on the top level of the criteria object, it actually matches the wildcard against arrays of primitives as well:

test/filter_filter_spec.js

```
it('filters primitives with a wildcard property', function() {
  var fn = parse('arr | filter:{$: "o"}');
  expect(fn({arr: ['Joe', 'Jane', 'Mary']})).toEqual(['Joe']);
});
```

In the predicate function we simply use the value of the `$` property of the original filter expression - if there was one - when matching against a non-object:

src/filter_filter.js

```
function createPredicateFn(expression) {
  var shouldMatchPrimitives =
    _.isObject(expression) && ('$' in expression);

  function comparator(actual, expected) {
    if (_.isUndefined(actual)) {
      return false;
    }
    if (_.isNull(actual) || _.isNull(expected)) {
      return actual === expected;
    }
    actual = ('' + actual).toLowerCase();
    expected = ('' + expected).toLowerCase();
    return actual.indexOf(expected) !== -1;
  }
```



```
  return function predicateFn(item) {
    if (shouldMatchPrimitives && !_.isObject(item)) {
      return deepCompare(item, expression.$, comparator);
    }
    return deepCompare(item, expression, comparator, true);
  };
}
```

Finally, wildcard properties can also be nested. When you do that, you require some value to be present at least some number of levels deep in the object:

test/filter_filter_spec.js

```
it('filters with a nested wildcard property', function() {
  var fn = parse('arr | filter:{$: {$: "o"}}');
  expect(fn({arr: [
    {name: {first: 'Joe'}, role: 'admin'},
    {name: {first: 'Jane'}, role: 'moderator'},
    {name: {first: 'Mary'}, role: 'admin'}
  ]})).toEqual([
    {name: {first: 'Joe'}, role: 'admin'}
  ]);
});
```

This currently also matches `role: 'moderator'` though it should only match the `'o'` at least two levels deep since that's what our criteria object specifies.

We can fix this by first passing in a *fourth* argument to `deepCompare`, called `inWildcard`. We only set it to true when recursing from a wildcard criterion:

src/filter_filter.js

```
function deepCompare(
    actual, expected, comparator, matchAnyProperty, inWildcard) {
  if (_.isString(expected) && _.startsWith(expected, '!')) {
    return !deepCompare(actual, expected.substring(1),
                        comparator, matchAnyProperty);
  }
  if (_.isArray(actual)) {
    return _.any(actual, function(actualItem) {
      return deepCompare(actualItem, expected,
                         comparator, matchAnyProperty);
    });
  }

  if (_.isObject(actual)) {
    if (_.isObject(expected)) {
```



```
      return _.every(
        _.toPlainObject(expected),
        function(expectedVal, expectedKey) {
          if (_.isUndefined(expectedVal)) {
            return true;
          }
          var isWildcard = (expectedKey === '$');
          var actualVal = isWildcard ? actual : actual[expectedKey];
          return deepCompare(actualVal, expectedVal,
                             comparator, isWildcard, isWildcard);
        }
      );
    } else if (matchAnyProperty) {
      return _.some(actual, function(value, key) {
        return deepCompare(value, expected, comparator, matchAnyProperty);
      });
    } else {
      return comparator(actual, expected);
    }
  } else {
    return comparator(actual, expected);
  }
}
```

We can then prevent object criteria from being used when in a wildcard search, by guarding the corresponding `if` statement. That means we end up in the second conditional branch (`else if (matchAnyProperty)`), which jumps into the next level of nesting. In the recursive `deepCompare` calls done there, the `inWildcard` flag becomes falsy again, and we apply the second wildcard in the correct scope:

src/filter_filter.js

```
function deepCompare(
    actual, expected, comparator, matchAnyProperty, inWildcard) {
  if (_.isString(expected) && _.startsWith(expected, '!')) {
    return !deepCompare(actual, expected.substring(1),
                        comparator, matchAnyProperty);
  }
  if (_.isArray(actual)) {
    return _.any(actual, function(actualItem) {
      return deepCompare(actualItem, expected,
                         comparator, matchAnyProperty);
    });
  }

  if (_.isObject(actual)) {
    if (_.isObject(expected) && !inWildcard) {
      return _.every(
        _.toPlainObject(expected),
```

```
        function(expectedVal, expectedKey) {
          if (_.isUndefined(expectedVal)) {
            return true;
          }
          var isWildcard = (expectedKey === '$');
          var actualVal = isWildcard ? actual : actual[expectedKey];
          return deepCompare(actualVal, expectedVal,
                             comparator, isWildcard, isWildcard);
        }
      );
    } else if (matchAnyProperty) {
      return _.some(actual, function(value, key) {
        return deepCompare(value, expected, comparator, matchAnyProperty);
      });
    } else {
      return comparator(actual, expected);
    }
  } else {
    return comparator(actual, expected);
  }
}
```

Filtering With Custom Comparators

You may also customize the strategy by which the filter compares two values by providing your own *comparator* function as the second additional argument. For example, here we provide a comparator that compares two values using the strict equality operator `===`:

test/filter_filter_spec.js

```
it('allows using a custom comparator', function() {
  var fn = parse('arr | filter:{$: "o"}:myComparator');
  expect(fn({
    arr: ['o', 'oo', 'ao', 'aa'],
    myComparator: function(left, right) {
      return left === right;
    }
  })).toEqual(['o']);
});
```

This is different from providing a *filter predicate function* as seen earlier in the chapter. Whereas a filter predicate decides, based on arbitrary criteria, whether a given item should pass the filter or not, a comparator function compares a given item to the filter value (or a part of it) and decides how they should be compared.

We need to accept this third argument to the filter function, and pass it to predicate function creation:

src/filter_filter.js


```
function filterFilter() {
  return function(array, filterExpr, comparator) {
    var predicateFn;
    if (_.isFunction(filterExpr)) {
      predicateFn = filterExpr;
    } else if (_.isString(filterExpr) ||
               _.isNumber(filterExpr) ||
               _.isBoolean(filterExpr) ||
               _.isNull(filterExpr) ||
               _.isObject(filterExpr)) {
      predicateFn = createPredicateFn(filterExpr, comparator);
    } else {
      return array;
    }
    return _.filter(array, predicateFn);
  };
}
```

In `createPredicateFn` we should now only form the custom comparator if one wasn't given already:

src/filter_filter.js

```
function createPredicateFn(expression, comparator) {
  var shouldMatchPrimitives =
    _.isObject(expression) && ('$' in expression);

  if (!_.isFunction(comparator)) {
    comparator = function(actual, expected) {
      if (_.isUndefined(actual)) {
        return false;
      }
      if (_.isNull(actual) || _.isNull(expected)) {
        return actual === expected;
      }
      actual = ('' + actual).toLowerCase();
      expected = ('' + expected).toLowerCase();
      return actual.indexOf(expected) !== -1;
    };
  }

  return function predicateFn(item) {
    if (shouldMatchPrimitives && !_.isObject(item)) {
      return deepCompare(item, expression.$, comparator);
    }
    return deepCompare(item, expression, comparator, true);
  };
}
```


You may also let the filter know that it should use strict *value equality* for comparison (instead of the more lenient substring matching), by passing the special value `true` in place of the comparator:

test/filter_filter_spec.js

```
it('allows using an equality comparator', function() {
  var fn = parse('arr | filter:{name: "Jo"}:true');
  expect(fn({arr: [
    {name: "Jo"},
    {name: "Joe"}
  ]})).toEqual([
    {name: "Jo"}
  ]);
});
```

This is useful if you indeed want to match your filter to the values *precisely* and don't want to accept partial string matches.

When the comparator value is true, `createPredicateFn` will use the LoDash _.isEqual function as the comparator. It returns true if two values are exactly equal:

src/filter_filter.js

```
function createPredicateFn(expression, comparator) {
  var shouldMatchPrimitives =
    _.isObject(expression) && ('$' in expression);

  if (comparator === true) {
    comparator = _.isEqual;
  } else if (!_.isFunction(comparator)) {
    comparator = function(actual, expected) {
      if (_.isUndefined(actual)) {
        return false;
      }
      if (_.isNull(actual) || _.isNull(expected)) {
        return actual === expected;
      }
      actual = ('' + actual).toLowerCase();
      expected = ('' + expected).toLowerCase();
      return actual.indexOf(expected) !== -1;
    };
  }

  return function predicateFn(item) {
    if (shouldMatchPrimitives && !_.isObject(item)) {
      return deepCompare(item, expression.$, comparator);
    }
    return deepCompare(item, expression, comparator, true);
  };
}
```


Summary

During this chapter we've added the final missing feature to our implementation of the Angular expression language. Filters are an often useful way to modify the resulting value of an expression using predefined filter functions that may be reused all across your application.

In this chapter you've learned:

- That filters are applied to expressions using the pipe operator `|`.
- That Angular expressions don't support bitwise operators, and that the bitwise OR would conflict with the filter operator.
- That filters are registered and obtained using the filter service.
- How you can register several filters in bulk by giving the filter service an object.
- How filter expressions are processed as call expressions by the AST builder and compiler.
- That filter expressions have the lowest precedence of all expressions.
- How the AST compiler generates JavaScript code to look up all the filters used in an expression from the filter service at runtime.
- How several filter invocations can be chained.
- How additional arguments can be passed to filters, and how they're given to the filter function as the second, third, etc. arguments.
- How the built-in filter filter works: With predicate functions, primitives, or objects as the filter expression. With nested objects and arrays. With wildcard `$` keys, and with custom comparators.

Chapter 9

Expressions And Watches

We now have a complete implementation of dirty-checking and a complete implementation of the Angular expression language, but we have not yet integrated them in any way. As all Angular users know, the true power of Angular's dirty-checking system is in the *combination* of these two things. After this chapter we'll have that combination.

This chapter concludes our coverage of expressions and scopes by putting them together. In addition to the basic integration between these two systems, we'll build a few powerful optimizations: *Constant detection*, *One-time binding* and *input tracking*. We'll then wrap things up by looking at how expressions can not only be evaluated, but also *reassigned*.

Integrating Expressions to Scopes

The external API of `Scope` should accept not only raw functions, but also *expression strings* in the following methods:

- `$watch`
- `$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 can do this in `parse`, so that if you try to parse a function, it'll just return that function back to you:

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 missing completely). 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 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 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.aValue = 42;
  scope.$watch('aValue', function(newValue, oldValue, scope) {
    theValue = newValue;
  });
  scope.$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.$$watchers.unshift(watcher);
  this.$$root.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
      self.$$root.$$lastDirtyWatch = null;
    }
  };
};
```

Since `$watch` now accepts expressions, `$watchCollection` should accept them as well. Add a new test to the `describe("$watchCollection")` test block:

test/scope_spec.js

```
it('accepts expressions for watch functions', function() {
  var theValue;

  scope.aColl = [1, 2, 3];
  scope.$watchCollection('aColl', function(newValue, oldValue, scope) {
    theValue = newValue;
  });
  scope.$digest();

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


To make this work, we also need to call `parse` for the watch function in `$watchCollection`:

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;

  watchFn = parse(watchFn);

  // The rest of the function unchanged
};
```

Next, we should also 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:

test/scope_spec.js

```
it('accepts expressions in $apply', function() {
  scope.aFunction = _.constant(42);
  expect(scope.$apply('aFunction()')).toBe(42);
});

it('accepts expressions in $evalAsync', function(done) {
  var called;
  scope.aFunction = function() {
    called = true;
  };

  scope.$evalAsync('aFunction()');

  scope.$$postDigest(function() {
    expect(called).toBe(true);
    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);
};
```

Since `$apply` and `$evalAsync` are implemented in terms of `$eval`, this change immediately adds expression support for them as well.

Note that we're passing the `locals` argument on to the expression function. As we saw when we implemented lookup expressions, it may be used to override scope field access.

Literal And Constant Expressions

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 an integer or array literal.
- `constant` - a boolean value denoting whether the expression was a constant, i.e. a literal primitive, or a literal collection of constant values. When an expression is constant, its value will never change over time.

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 it is not a constant since `aVariable` is not a constant.

Users of the `$parse` service occasionally use these two flags to make decisions about how to use the expression. The `constant` flag in particular will be used in this chapter to apply some optimizations in expression watching.

> There's also a third attribute, called `assign`, in the returned functions. We will get to know it later in this chapter.

Let's talk about the `literal` flag first as it is simpler to implement. All kinds of simple literal values, including numbers, strings, and booleans should be marked as literal:

test/parse_spec.js


```
it('marks integers literal', function() {
  var fn = parse('42');
  expect(fn.literal).toBe(true);
});

it('marks strings literal', function() {
  var fn = parse('"abc"');
  expect(fn.literal).toBe(true);
});

it('marks booleans literal', function() {
  var fn = parse('true');
  expect(fn.literal).toBe(true);
});
```

Both array and object literals should also be marked as literal:

test/parse_spec.js

```
it('marks arrays literal', function() {
  var fn = parse('[1, 2, aVariable]');
  expect(fn.literal).toBe(true);
});

it('marks objects literal', function() {
  var fn = parse('{a: 1, b: aVariable}');
  expect(fn.literal).toBe(true);
});
```

Anything else should be marked non-literal:

test/parse_spec.js

```
it('marks unary expressions non-literal', function() {
  var fn = parse('!false');
  expect(fn.literal).toBe(false);
});

it('marks binary expressions non-literal', function() {
  var fn = parse('1 + 2');
  expect(fn.literal).toBe(false);
});
```

What we'll do is check if the AST is literal using a helper function `isLiteral`. Then we'll attach the result in the compiled expression function:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {
    body: [],
    nextId: 0,
    vars: [],
    filters: {}
  };
  this.recurse(ast);
  var fnString = this.filterPrefix() +
    'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) +
    this.state.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  var fn = new Function(
    'ensureSafeMemberName',
    'ensureSafeObject',
    'ensureSafeFunction',
    'ifDefined',
    'filter',
    fnString)(
      ensureSafeMemberName,
      ensureSafeObject,
      ensureSafeFunction,
      ifDefined,
      filter);
  /* jshint +W054 */
  fn.literal = isLiteral(ast);
  return fn;
};
```

The `isLiteral` function is defined as follows:

- An empty program is literal
- A non-empty program is literal if it has just one expression whose type is a literal, an array, or an object.

Expressed in code:

src/parse.js

```
function isLiteral(ast) {
  return ast.body.length === 0 ||
      ast.body.length === 1 && (
      ast.body[0].type === AST.Literal ||
      ast.body[0].type === AST.ArrayExpression ||
      ast.body[0].type === AST.ObjectExpression);
}
```


Setting the `constant` flag is a little bit more involved. We're going to need to consider each AST node type separately to see how its "constantness" should be determined.

Let's begin from simple literals. Numbers, strings, and booleans are all constants:

test/parse_spec.js

```
it('marks integers constant', function() {
  var fn = parse('42');
  expect(fn.constant).toBe(true);
});

it('marks strings constant', function() {
  var fn = parse('"abc"');
  expect(fn.constant).toBe(true);
});

it('marks booleans constant', function() {
  var fn = parse('true');
  expect(fn.constant).toBe(true);
});
```

The generated function will have a `constant` flag, just like it has a `literal` flag. The value of the flag is read from the root AST node:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  this.state = {
    body: [],
    nextId: 0,
    vars: [],
    filters: {}
  };
  this.recurse(ast);
  var fnString = this.filterPrefix() +
    'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) +
    this.state.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  var fn = new Function(
    'ensureSafeMemberName',
    'ensureSafeObject',
```

```
    'ensureSafeFunction',
    'ifDefined',
    'filter',
    fnString)(
      ensureSafeMemberName,
      ensureSafeObject,
      ensureSafeFunction,
      ifDefined,
      filter);
  /* jshint +W054 */
  fn.literal = isLiteral(ast);
  fn.constant = ast.constant;
  return fn;
};
```

The problem is that the root AST node has no such flag yet. How does it end up there? Well, what we are going to do is pre-process the AST before compilation, using a function called `markConstantExpressions`. We expect the `ast` flag to be set during that pre-processing:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantExpressions(ast);
  this.state = {
    body: [],
    nextId: 0,
    vars: [],
    filters: {}
  };
  this.recurse(ast);
  var fnString = this.filterPrefix() +
    'var fn=function(s,l){' +
    (this.state.vars.length ?
      'var ' + this.state.vars.join(',') + ';' :
      ''
    ) +
    this.state.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  var fn = new Function(
    'ensureSafeMemberName',
    'ensureSafeObject',
    'ensureSafeFunction',
    'ifDefined',
    'filter',
    fnString)(
      ensureSafeMemberName,
      ensureSafeObject,
```

```
      ensureSafeFunction,
      ifDefined,
      filter);
  /* jshint +W054 */
  fn.literal = isLiteral(ast);
  fn.constant = ast.constant;
  return fn;
};
```

A bit like `recurse`, `markConstantExpressions` will be a recursive function that consists of one big `switch` statement. For instance, literal expressions will all be constant, so when `markConstantExpressions` is called with a literal AST node, it will set its `constant` flag to `true`:

src/parse.js

```
function markConstantExpressions(ast) {
  switch (ast.type) {
  case AST.Literal:
    ast.constant = true;
    break;
  }
}
```

Before we get our test cases to pass we need to consider the fact that the root node of an AST is always of type `Program`. A program consists of a number of subexpressions. When `markConstantExpressions` sees a program, it needs to recursively call itself for each of those subexpressions:

src/parse.js

```
function markConstantExpressions(ast) {
  switch (ast.type) {
  case AST.Program:
    _.forEach(ast.body, function(expr) {
      markConstantExpressions(expr);
    });
    break;
  case AST.Literal:
    ast.constant = true;
    break;
  }
}
```

The `Program` node can then be marked as constant if all of the subexpressions are constant:

src/parse.js


```
function markConstantExpressions(ast) {
  var allConstants;
  switch (ast.type) {
  case AST.Program:
    allConstants = true;
    _.forEach(ast.body, function(expr) {
      markConstantExpressions(expr);
      allConstants = allConstants && expr.constant;
    });
    ast.constant = allConstants;
    break;
  case AST.Literal:
    ast.constant = true;
    break;
  }
}
```

Identifier expressions are never constant - we can't know if they will change over time:

test/parse_spec.js

```
it('marks identifiers non-constant', function() {
  var fn = parse('a');
  expect(fn.constant).toBe(false);
});
```

The implementation for this is straightforward:

src/parse.js

```
case AST.Identifier:
  ast.constant = false;
  break;
```

Array expressions are constant if and only if all of their elements are constant:

test/parse_spec.js

```
it('marks arrays constant when elements are constant', function() {
  expect(parse('[1, 2, 3]').constant).toBe(true);
  expect(parse('[1, [2, [3]]]').constant).toBe(true);
  expect(parse('[1, 2, a]').constant).toBe(false);
  expect(parse('[1, [2, [a]]]').constant).toBe(false);
});
```

We can check arrays very similarly as we did programs: We recurse to each element in the array, and mark the array based on what we saw:

src/parse.js

```
case AST.ArrayExpression:
  allConstants = true;
  _.forEach(ast.elements, function(element) {
    markConstantExpressions(element);
    allConstants = allConstants && element.constant;
  });
  ast.constant = allConstants;
  break;
```

Similarly for objects, constantness depends on whether each of the values inside the object is constant. (Object keys are just strings, which are by definition always constant, so we need not consider them.)

test/parse_spec.js

```
it('marks objects constant when values are constant', function() {
  expect(parse('{a: 1, b: 2}').constant).toBe(true);
  expect(parse('{a: 1, b: {c: 3}}').constant).toBe(true);
  expect(parse('{a: 1, b: something}').constant).toBe(false);
  expect(parse('{a: 1, b: {c: something}}').constant).toBe(false);
});
```

We implement this by iterating over the object properties and marking their values:

src/parse.js

```
case AST.ObjectExpression:
  allConstants = true;
  _.forEach(ast.properties, function(property) {
    markConstantExpressions(property.value);
    allConstants = allConstants && property.value.constant;
  });
  ast.constant = allConstants;
  break;
```

`this` is not constant. It can't be, since its value is the runtime Scope:

test/parse_spec.js

```
it('marks this as non-constant', function() {
  expect(parse('this').constant).toBe(false);
});
```


The implementation of `markConstantExpressions` for `ThisExpression` is unsurprising:

src/parse.js

```
case AST.ThisExpression:
  ast.constant = false;
  break;
```

For non-computed lookup expressions, constantness is defined in terms of the constantness of the object from which we're looking things up:

test/parse_spec.js

```
it('marks non-computed lookup constant when object is constant', function() {
  expect(parse('{a: 1}.a').constant).toBe(true);
  expect(parse('obj.a').constant).toBe(false);
});
```

When we have a lookup, we first visit the object and then set the constant flag:

src/parse.js

```
case AST.MemberExpression:
  markConstantExpressions(ast.object);
  ast.constant = ast.object.constant;
  break;
```

When we extend our consideration to computed lookups, we should also see if the lookup key is constant or not:

test/parse_spec.js

```
it('marks computed lookup constant when object and key are', function() {
  expect(parse('{a: 1}["a"]').constant).toBe(true);
  expect(parse('obj["a"]').constant).toBe(false);
  expect(parse('{a: 1}[something]').constant).toBe(false);
  expect(parse('obj[something]').constant).toBe(false);
});
```

If the lookup is computed, we additionally visit the property node:

src/parse.js


```
case AST.MemberExpression:
  markConstantExpressions(ast.object);
  if (ast.computed) {
    markConstantExpressions(ast.property);
  }
  ast.constant = ast.object.constant &&
                 (!ast.computed || ast.property.constant);
  break;
```

A call expression is not constant - we cannot make such assumptions about the nature of the function being called:

test/parse_spec.js

```
it('marks function calls non-constant', function() {
  expect(parse('aFunction()').constant).toBe(false);
});
```

We can just set the flag to `false`:

src/parse.js

```
case AST.CallExpression:
  ast.constant = false;
  break;
```

A special case for this are *filters*. They are also call expressions but unlike regular function calls, they *are* considered constant if their input expressions are constant:

test/parse_spec.js

```
it('marks filters constant if arguments are', function() {
  register('aFilter', function() {
    return _.identity;
  });
  expect(parse('[1, 2, 3] | aFilter').constant).toBe(true);
  expect(parse('[1, 2, a] | aFilter').constant).toBe(false);
  expect(parse('[1, 2, 3] | aFilter:42').constant).toBe(true);
  expect(parse('[1, 2, 3] | aFilter:a').constant).toBe(false);
});
```

We can repeat the same trick here as we did with arrays and objects: Iterate over the arguments array and set the constant flag only if everything in it is constant. Note that we initialize the value based on the `filter` flag of the node, so that for non-filter calls the flag never becomes `true`:

src/parse.js


```
case AST.CallExpression:
  allConstants = ast.filter ? true : false;
  _.forEach(ast.arguments, function(arg) {
    markConstantExpressions(arg);
    allConstants = allConstants && arg.constant;
  });
  ast.constant = allConstants;
  break;
```

> There are cases where filter calls are not constant even if their arguments are. We will return to this later in the chapter.

An assignment expression is in fact constant when both of its sides are constant, even though an assignment with a constant left-hand side is otherwise nonsensical:

test/parse_spec.js

```
it('marks assignments constant when both sides are', function() {
  expect(parse('1 = 2').constant).toBe(true);
  expect(parse('a = 2').constant).toBe(false);
  expect(parse('1 = b').constant).toBe(false);
  expect(parse('a = b').constant).toBe(false);
});
```

We should visit both sides of the assignment and set the `constant` flag based on the results:

src/parse.js

```
case AST.AssignmentExpression:
  markConstantExpressions(ast.left);
  markConstantExpressions(ast.right);
  ast.constant = ast.left.constant && ast.right.constant;
  break;
```

A unary operator expression is constant if and only if its argument is constant:

test/parse_spec.js

```
it('marks unaries constant when arguments are constant', function() {
  expect(parse('+42').constant).toBe(true);
  expect(parse('+a').constant).toBe(false);
});
```


In the implementation we first visit the argument and then check the flag based on that:

src/parse.js

```
case AST.UnaryExpression:
  markConstantExpressions(ast.argument);
  ast.constant = ast.argument.constant;
  break;
```

The constantness of binary and logical expressions is based on the constantness of their left and right arguments. If both arguments are constant, the whole expression is constant:

test/parse_spec.js

```
it('marks binaries constant when both arguments are constant', function() {
  expect(parse('1 + 2').constant).toBe(true);
  expect(parse('1 + 2').literal).toBe(false);
  expect(parse('1 + a').constant).toBe(false);
  expect(parse('a + 1').constant).toBe(false);
  expect(parse('a + a').constant).toBe(false);
});

it('marks logicals constant when both arguments are constant', function() {
  expect(parse('true && false').constant).toBe(true);
  expect(parse('true && false').literal).toBe(false);
  expect(parse('true && a').constant).toBe(false);
  expect(parse('a && false').constant).toBe(false);
  expect(parse('a && b').constant).toBe(false);
});
```

In the implementation for these two we first visit the left and right argument nodes and then form the `constant` flag for the expression itself:

src/parse.js

```
case AST.BinaryExpression:
case AST.LogicalExpression:
  markConstantExpressions(ast.left);
  markConstantExpressions(ast.right);
  ast.constant = ast.left.constant && ast.right.constant;
  break;
```

Finally, the ternary operator is constant if all of the three of its operands are constant:

test/parse_spec.js

```
it('marks ternaries constant when all arguments are', function() {
  expect(parse('true ? 1 : 2').constant).toBe(true);
  expect(parse('a ? 1 : 2').constant).toBe(false);
  expect(parse('true ? a : 2').constant).toBe(false);
  expect(parse('true ? 1 : b').constant).toBe(false);
  expect(parse('a ? b : c').constant).toBe(false);
});
```

Here we visit each of the three operand nodes first, before forming the result:

src/parse.js

```
case AST.ConditionalExpression:
  markConstantExpressions(ast.test);
  markConstantExpressions(ast.consequent);
  markConstantExpressions(ast.alternate);
  ast.constant =
    ast.test.constant && ast.consequent.constant && ast.alternate.constant;
  break;
```

And now we have implemented a full tree-walking function for marking all kinds of expressions as constant or non-constant.

Optimizing Constant Expression Watching

Using expression strings in watches enables us to add a new optimization that will in some cases make the digest loop go faster. In the previous section 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. That means we can safely remove the watch so that we don't have to incur the cost of dirty-checking in future digests:

test/scope_spec.js

```
it('removes constant watches after first invocation', function() {
  scope.$watch('[1, 2, 3]', function() {});
  scope.$digest();

  expect(scope.$$watchers.length).toBe(0);
});
```

This test case throws a "10 iterations reached" exception at the moment, because the expression generates a new array each time it's evaluated, and the reference watch considers it a new value. Since `[1, 2, 3]` is a constant, it should *not* be evaluated more than once.

This optimization can be implemented with a new expression feature called *watch delegates*. A watch delegate is a function that may be attached to an expression. When an expression with a watch delegate is encountered in `Scope.$watch`, that delegate is used to bypass normal watch creation. Instead of creating a watcher, we *delegate* that job to the expression itself:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;

  watchFn = parse(watchFn);

  if (watchFn.$$watchDelegate) {
    return watchFn.$$watchDelegate(self, listenerFn, valueEq, watchFn);
  }

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

The delegate is given everything it might need to know to construct the watch correctly: The scope instance, the listener function, the value/reference equality flag, and the watch expression itself.

> As the two dollar signs in the name attest, watch delegates are designed to be an internal facility in the Angular framework, not something for application developers to use directly.

The expression parser can now attach watch delegates to expressions whenever it wants something special to happen when those expressions are used in watches. One instance of this is constant expressions, for which we can now introduce a *constant watch delegate*:

src/parse.js


```
function parse(expr) {
  switch (typeof expr) {
  case 'string':
    var lexer = new Lexer();
    var parser = new Parser(lexer);
    var parseFn = parser.parse(expr);
    if (parseFn.constant) {
      parseFn.$$watchDelegate = constantWatchDelegate;
    }
    return parseFn;
  case 'function':
    return expr;
  default:
    return _.noop;
  }
}
```

The constant watch delegate is a watcher that behaves like any other watcher, except that it removes itself immediately upon first invocation:

src/parse.js

```
function constantWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  var unwatch = scope.$watch(
    function() {
      return watchFn(scope);
    },
    function(newValue, oldValue, scope) {
      if (_.isFunction(listenerFn)) {
        listenerFn.apply(this, arguments);
      }
      unwatch();
    },
    valueEq
  );
  return unwatch;
}
```

Note that we don't use the original `watchFn` directly as the first argument to `$watch`, because if we did, `$watch` would find the `$$watchDelegate` from it again, resulting in infinite recursion. Instead we wrap it in a function that has no `$$watchDelegate`.

Note also that we return the `unwatch` function. Even though a constant watch removes itself after its first invocation, it can also be removed by using the return value from `Scope.$watch`, just like any other watch.

One-Time Expressions

As Angular application developers we often run into situations where we know that after some watch first gets a value, that value will never change. A typical example of this is something like a list of objects:

```
<li ng-repeat="user in users">
  {{user.firstName}} {{user.lastName}}
</li>
```

This snippet uses a collection watcher with `ng-repeat`, but it also uses *two more watchers per each user* - one for the first name and one for the last name.

It is quite common that in a list like this, the first and last names of a given user don't change, because they are read-only - there is no application logic that mutates them. Nevertheless, Angular now has to dirty-check them in each digest, because it doesn't know you have no intention of changing them. This kind of unnecessary dirty-checking can be a significant performance issue in large applications and on low end devices.

Angular has a feature called *one-time binding* for getting around this. When you know that the value of a watcher will never change as long as the watcher exists, you can let Angular know about that by prefixing the watch expression with two colon characters:

```
<li ng-repeat="user in users">
  {{::user.firstName}} {{::user.lastName}}
</li>
```

The one-time watching syntax is implemented in the expression engine and you can use it in any watch expression:

src/scope_test.js

```
it('accepts one-time watches', function() {
  var theValue;

  scope.aValue = 42;
  scope.$watch('::aValue', function(newValue, oldValue, scope) {
    theValue = newValue;
  });
  scope.$digest();

  expect(theValue).toBe(42);
});
```

The crucial difference between one-time watchers and regular watchers is that when a one-time watcher has been resolved, it is automatically removed so that it doesn't put any more pressure on the digest loop:

src/scope_test.js


```
it('removes one-time watches after first invocation', function() {
  scope.aValue = 42;
  scope.$watch('::aValue', function() { });
  scope.$digest();

  expect(scope.$$watchers.length).toBe(0);
});
```

One-time watching is handled completely within `parse.js`, by making use of the watch delegate system we introduced in the previous section. If the expression is preceded by two colons, a "one-time watch delegate" will be attached to it:

src/parse.js

```
function parse(expr) {
  switch (typeof expr) {
  case 'string':
    var lexer = new Lexer();
    var parser = new Parser(lexer);
    var oneTime = false;
    if (expr.charAt(0) === ':' && expr.charAt(1) === ':') {
      oneTime = true;
      expr = expr.substring(2);
    }
    var parseFn = parser.parse(expr);
    if (parseFn.constant) {
      parseFn.$$watchDelegate = constantWatchDelegate;
    } else if (oneTime) {
      parseFn.$$watchDelegate = oneTimeWatchDelegate;
    }
    return parseFn;
  case 'function':
    return expr;
  default:
    return _.noop;
  }
}
```

On the surface level, the contract of the one-time watch delegate seems exactly the same as that of the constant watch delegate: Run the watch once and then remove it. Indeed, our current test suite passes if we implement the one-time watch delegate with identical behavior as the constant watch delegate:

src/parse.js

```
function oneTimeWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  var unwatch = scope.$watch(
    function() {
      return watchFn(scope);
    }, function(newValue, oldValue, scope) {
      if (_.isFunction(listenerFn)) {
        listenerFn.apply(this, arguments);
      }
      unwatch();
    }, valueEq
  );
  return unwatch;
}
```

There is a problem with this though, which is that unlike constants, a one-time expression might not always have a value when it is evaluated for the first time. We might still be waiting for the data to come back from a server, for example. One-time expressions are much more useful if they can support these kinds of asynchronous use cases. That's why *they should only be removed when their value becomes something else than* `undefined`:

test/scope_spec.js

```
it('does not remove one-time-watches until value is defined', function() {
  scope.$watch('::aValue', function() { });

  scope.$digest();
  expect(scope.$$watchers.length).toBe(1);

  scope.aValue = 42;
  scope.$digest();
  expect(scope.$$watchers.length).toBe(0);
});
```

We can make this test pass by guarding the `unwatch()` invocation with an `if` statement:

src/parse.js

```
function oneTimeWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  var unwatch = scope.$watch(
    function() {
      return watchFn(scope);
    }, function(newValue, oldValue, scope) {
      if (_.isFunction(listenerFn)) {
        listenerFn.apply(this, arguments);
      }
      if (!_.isUndefined(newValue)) {
        unwatch();
      }
    }, valueEq
  );
  return unwatch;
}
```


This is still not quite good enough though. As we've seen, a lot of things may happen during a digest, and one of those things may be that the expression value becomes `undefined` again. Angular's one-time expressions are only removed when the value is *stabilized*, which means that it must be something else than `undefined` at the end of a digest:

test/scope_spec.js

```
it('does not remove one-time-watches until value stays defined', function() {
  scope.aValue = 42;

  scope.$watch('::aValue', function() { });
  var unwatchDeleter = scope.$watch('aValue', function() {
    delete scope.aValue;
  });

  scope.$digest();
  expect(scope.$$watchers.length).toBe(2);

  scope.aValue = 42;
  unwatchDeleter();
  scope.$digest();
  expect(scope.$$watchers.length).toBe(0);
});
```

In this test we have a second watcher that causes the value of the one-time watcher to become `undefined` during the digest. While that second watcher is still alive, the one-time watcher does not stabilize and should not be removed.

We must keep track of the *last* value seen by the one-time watch, and *after the digest* check whether it is defined. Only then do we remove the watcher. We can defer the removal using the `$$postDigest` method:

src/parse.js

```
function oneTimeWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  var lastValue;
  var unwatch = scope.$watch(
    function() {
      return watchFn(scope);
    }, function(newValue, oldValue, scope) {
      lastValue = newValue;
      if (_.isFunction(listenerFn)) {
        listenerFn.apply(this, arguments);
      }
      if (!_.isUndefined(newValue)) {
        scope.$$postDigest(function() {
          if (!_.isUndefined(lastValue)) {
            unwatch();
```



```
          }
        });
      }
    }, valueEq
  );
  return unwatch;
}
```

This is already pretty good, but there's one more special case one-time watches should handle for us: When used with a *collection literal*, such as an array or an object, one-time watches check if the values *inside* the literal are all defined before removing itself. For example, when one-time watching an array literal, the watch is only removed after the array does not have any `undefined` items in it:

test/scope_spec.js

```
it('does not remove one-time watches before all array items defined', function() {
  scope.$watch('::[1, 2, aValue]', function() { }, true);

  scope.$digest();
  expect(scope.$$watchers.length).toBe(1);

  scope.aValue = 3;
  scope.$digest();
  expect(scope.$$watchers.length).toBe(0);
});
```

The same is true for objects. A one-time watch for an object literal is only removed when the object has no `undefined` values:

src/parse.js

```
it('does not remove one-time watches before all object vals defined', function() {
  scope.$watch('::{a: 1, b: aValue}', function() { }, true);

  scope.$digest();
  expect(scope.$$watchers.length).toBe(1);

  scope.aValue = 3;
  scope.$digest();
  expect(scope.$$watchers.length).toBe(0);
});
```

> This feature enables one-time binding in directives that are configured with an object literal syntax, such as `ngClass` and `ngStyle`.

If the expression is a literal, we should use a special "one time literal watch" delegate for it, instead of the normal one time watch delegate:

src/parse.js


```
function parse(expr) {
  switch (typeof expr) {
    case 'string':
      var lexer = new Lexer();
      var parser = new Parser(lexer);
      var oneTime = false;
      if (expr.charAt(0) === ':' && expr.charAt(1) === ':') {
        oneTime = true;
        expr = expr.substring(2);
      }
      var parseFn = parser.parse(expr);

      if (parseFn.constant) {
        parseFn.$$watchDelegate = constantWatchDelegate;
      } else if (oneTime) {
        parseFn.$$watchDelegate = parseFn.literal ? oneTimeLiteralWatchDelegate :
                                                    oneTimeWatchDelegate;
      }

      return parseFn;
    case 'function':
      return expr;
    default:
      return _.noop;
  }
}
```

This new delegate is similar to the one-time watch delegate, but instead of checking if the expression value itself is defined, it assumes it is a collection and checks whether all of its contained items are defined:

src/parse.js

```
function oneTimeLiteralWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  function isAllDefined(val) {
    return !_.any(val, _.isUndefined);
  }
  var unwatch = scope.$watch(
    function() {
      return watchFn(scope);
    }, function(newValue, oldValue, scope) {
      if (_.isFunction(listenerFn)) {
        listenerFn.apply(this, arguments);
      }
      if (isAllDefined(newValue)) {
        scope.$$postDigest(function() {
          if (isAllDefined(newValue)) {
            unwatch();
```



```
          }
        });
      }
    }, valueEq
  );
  return unwatch;
}
```

Literal numbers, strings, booleans and `undefined` are also literals, but they never get passed to `oneTimeLiteralWatchDelegate` because they are also constants and thus get delegated to `constantWatchDelegate`. The one-time literal delegate is only applied for arrays and objects that hold at least one non-constant item.

Input Tracking

There's one more optimization we can do with expression watching, and that is something called *input tracking*. The idea is that when an expression is composed of one or more *input expressions* (like `'a * b'` is composed of `'a'` and `'b'`), there is no need to re-evaluate the expression unless at least one of its inputs has changed.

For example, an array literal expression should not change if none of its contained items has changed:

test/scope_spec.js

```
it('does not re-evaluate an array if its contents do not change', function() {
  var values = [];

  scope.a = 1;
  scope.b = 2;
  scope.c = 3;

  scope.$watch('[a, b, c]', function(value) {
    values.push(value);
  });

  scope.$digest();
  expect(values.length).toBe(1);
  expect(values[0]).toEqual([1, 2, 3]);

  scope.$digest();
  expect(values.length).toBe(1);

  scope.c = 4;
  scope.$digest();
  expect(values.length).toBe(2);
  expect(values[1]).toEqual([1, 2, 4]);

});
```


Here we are watching a non-constant array. We digest three times. On the first time we expect the listener to trigger with the value of the array. On the second time we expect the listener *not* to trigger because the array contents haven't changed. On the third time we mutate the contents of the array and expect the listener to trigger again.

What actually occurs is a "10 $digest iterations reached" exception, because we are using a reference watch, and the expression generates a new array reference on each invocation. It should not do that.

What we'll do is have each expression function created by the parser include information about its *input expressions* - the expressions that may cause the whole expression's value to change. We're essentially going to extend our AST compiler so that it compiles not only the full expression function, but also a collection of *input expression functions* for each of the full expression's inputs.

Let's consider the watcher side of the implementation first, before digging into the AST compiler. As we parse an expression, if it is not constant or one-time but it does have inputs, it should have an `inputs` attribute. If it does, we'll use a special `inputsWatchDelegate` to watch it:

src/parse.js

```
function parse(expr) {
  switch (typeof expr) {
    case 'string':
      var lexer = new Lexer();
      var parser = new Parser(lexer);
      var oneTime = false;
      if (expr.charAt(0) === ':' && expr.charAt(1) === ':') {
        oneTime = true;
        expr = expr.substring(2);
      }
      var parseFn = parser.parse(expr);

      if (parseFn.constant) {
        parseFn.$$watchDelegate = constantWatchDelegate;
      } else if (oneTime) {
        parseFn.$$watchDelegate = parseFn.literal ? oneTimeLiteralWatchDelegate :
                                                    oneTimeWatchDelegate;
      } else if (parseFn.inputs) {
        parseFn.$$watchDelegate = inputsWatchDelegate;
      }

      return parseFn;
    case 'function':
      return expr;
    default:
      return _.noop;
  }
}
```


The inputs watch delegate will implement a watcher in terms of the inputs of the given expression:

src/parse.js

```
function inputsWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  var inputExpressions = watchFn.inputs;

  return scope.$watch(function() {

  }, listenerFn, valueEq);
}
```

The input tracking is done by maintaining an array of the values of the input expressions. The array is initalized with some initial "unique" values (an empty function literal), and then updated on each watch run by evaluating each input expression against the scope:

src/parse.js

```
function inputsWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  var inputExpressions = watchFn.inputs;

  var oldValues = _.times(inputExpressions.length, _.constant(function() { }));

  return scope.$watch(function() {
    _.forEach(inputExpressions, function(inputExpr, i) {
      var newValue = inputExpr(scope);
      if (!expressionInputDirtyCheck(newValue, oldValues[i])) {
        oldValues[i] = newValue;
      }
    });
  }, listenerFn, valueEq);
}
```

The actual dirty-checking is delegated to helper function that does a simple `NaN`-aware reference equality check:

src/parse.js

```
function expressionInputDirtyCheck(newValue, oldValue) {
  return newValue === oldValue ||
    (typeof newValue === 'number' && typeof oldValue === 'number' &&
     isNaN(newValue) && isNaN(oldValue));
}
```


On each watch run, a `changed` flag is set if any of the inputs have actually changed:

src/parse.js

```
function inputsWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  var inputExpressions = watchFn.inputs;

  var oldValues = _.times(inputExpressions.length, _.constant(function() { }));

  return scope.$watch(function() {
    var changed = false;
    _.forEach(inputExpressions, function(inputExpr, i) {
      var newValue = inputExpr(scope);
      if (changed || !expressionInputDirtyCheck(newValue, oldValues[i])) {
        changed = true;
        oldValues[i] = newValue;
      }
    });
  }, listenerFn, valueEq);
}
```

If a change has occurred, a new value for the compound expression itself is evaluated. It is used as the return value of the watch:

src/parse.js

```
function inputsWatchDelegate(scope, listenerFn, valueEq, watchFn) {
  var inputExpressions = watchFn.inputs;

  var oldValues = _.times(inputExpressions.length, _.constant(function() { }));
  var lastResult;

  return scope.$watch(function() {
    var changed = false;
    _.forEach(inputExpressions, function(inputExpr, i) {
      var newValue = inputExpr(scope);
      if (changed || !expressionInputDirtyCheck(newValue, oldValues[i])) {
        changed = true;
        oldValues[i] = newValue;
      }
    });
    if (changed) {
      lastResult = watchFn(scope);
    }
    return lastResult;
  }, listenerFn, valueEq);
}
```


The `lastResult` variable will continue holding the same value until at least one of the input expressions changes again.

> The Angular.js implementation has one additional optimization in `inputsWatchDelegate` for expressions that have a single input only. In those cases it skips the creation of the `oldValues` array, removing some memory and computation overhead. We will skip that optimization here.

With the watch delegate taken care of, let's think about how that `inputs` array it uses comes to be. This takes us back into the AST compiler.

The process for forming `inputs` begins by determining what the inputs of an expression are. Different kinds of expressions have different kinds of inputs, so the input nodes of each AST node type need to be determined separately. This means we're going to need a similar tree-walking function as the one we created for constant checking.

In fact, we're not going to create a new tree-walking function, but extend the existing function so that it does input checking in addition to constant checking. Let's first rename it so it reflects its new purpose better:

src/parse.js

```
function markConstantAndWatchExpressions(ast) {
  var allConstants;
  switch (ast.type) {
  case AST.Program:
    allConstants = true;
    _.forEach(ast.body, function(expr) {
      markConstantAndWatchExpressions(expr);
      allConstants = allConstants && expr.constant;
    });
    ast.constant = allConstants;
    break;
  case AST.Literal:
    ast.constant = true;
    break;
  case AST.Identifier:
    ast.constant = false;
    break;
  case AST.ArrayExpression:
    allConstants = true;
    _.forEach(ast.elements, function(element) {
      markConstantAndWatchExpressions(element);
      allConstants = allConstants && element.constant;
    });
    ast.constant = allConstants;
    break;
  case AST.ObjectExpression:
    allConstants = true;
    _.forEach(ast.properties, function(property) {
      markConstantAndWatchExpressions(property.value);
```



```
      allConstants = allConstants && property.value.constant;
    });
    ast.constant = allConstants;
    break;
  case AST.ThisExpression:
    ast.constant = false;
    break;
  case AST.MemberExpression:
    markConstantAndWatchExpressions(ast.object);
    if (ast.computed) {
      markConstantAndWatchExpressions(ast.property);
    }
    ast.constant = ast.object.constant &&
                   (!ast.computed || ast.property.constant);
    break;
  case AST.CallExpression:
    allConstants = ast.filter ? true : false;
    _.forEach(ast.arguments, function(arg) {
      markConstantAndWatchExpressions(arg);
      allConstants = allConstants && arg.constant;
    });
    ast.constant = allConstants;
    break;
  case AST.AssignmentExpression:
    markConstantAndWatchExpressions(ast.left);
    markConstantAndWatchExpressions(ast.right);
    ast.constant = ast.left.constant && ast.right.constant;
    break;
  case AST.UnaryExpression:
    markConstantAndWatchExpressions(ast.argument);
    ast.constant = ast.argument.constant;
    break;
  case AST.BinaryExpression:
  case AST.LogicalExpression:
    markConstantAndWatchExpressions(ast.left);
    markConstantAndWatchExpressions(ast.right);
    ast.constant = ast.left.constant && ast.right.constant;
    break;
  case AST.ConditionalExpression:
    markConstantAndWatchExpressions(ast.test);
    markConstantAndWatchExpressions(ast.consequent);
    markConstantAndWatchExpressions(ast.alternate);
    ast.constant =
      ast.test.constant && ast.consequent.constant && ast.alternate.constant;
    break;
  }
}
```

The name change needs to be reflected in `ASTCompiler.compile` as well:

src/parse.js


```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  // ...
};
```

What we're now going to do in this function, in addition to marking constants, is gather the input nodes of each AST node into a `toWatch` attribute. We'll need a variable to gather those inputs in, so let's introduce it first:

src/parse.js

```
function markConstantAndWatchExpressions(ast) {
  var allConstants;
  var argsToWatch;
  // ...
}
```

Now, let's consider the inputs of each node type in turn. For each node type, we need to think "when would the value of this expression change?"

For literals, there is nothing to watch - simple literals never change:

src/parse.js

```
case AST.Literal:
  ast.constant = true;
  ast.toWatch = [];
  break;
```

For identifier expressions, what needs to be watched is the expression itself. There are no smaller parts that it could be broken into:

src/parse.js

```
case AST.Identifier:
  ast.constant = false;
  ast.toWatch = [ast];
  break;
```

For arrays, we need to watch all the inputs of any non-constant elements in the array:

src/parse.js


```
case AST.ArrayExpression:
  allConstants = true;
  argsToWatch = [];
  _.forEach(ast.elements, function(element) {
    markConstantAndWatchExpressions(element);
    allConstants = allConstants && element.constant;
    if (!element.constant) {
      argsToWatch.push.apply(argsToWatch, element.toWatch);
    }
  });
  ast.constant = allConstants;
  ast.toWatch = argsToWatch;
  break;
```

Ditto for objects, inputs are based on the inputs of non-constant values inside the object:

src/parse.js

```
case AST.ObjectExpression:
  allConstants = true;
  argsToWatch = [];
  _.forEach(ast.properties, function(property) {
    markConstantAndWatchExpressions(property.value);
    allConstants = allConstants && property.value.constant;
    if (!property.value.constant) {
      argsToWatch.push.apply(argsToWatch, property.value.toWatch);
    }
  });
  ast.constant = allConstants;
  ast.toWatch = argsToWatch;
  break;
```

`this` has no inputs:

src/parse.js

```
case AST.ThisExpression:
  ast.constant = false;
  ast.toWatch = [];
  break;
```

A member expression, like an identifier, has no separate inputs. What needs to be watched is the expression itself:

src/parse.js

```
case AST.MemberExpression:
  markConstantAndWatchExpressions(ast.object);
  if (ast.computed) {
    markConstantAndWatchExpressions(ast.property);
  }
  ast.constant = ast.object.constant &&
                 (!ast.computed || ast.property.constant);
  ast.toWatch = [ast];
  break;
```

A call expression's input is considered to be the call itself, unless it is a filter, in which case the inputs are formed of its non-constant arguments:

src/parse.js

```
case AST.CallExpression:
  allConstants = ast.filter ? true : false;
  argsToWatch = [];
  _.forEach(ast.arguments, function(arg) {
    markConstantAndWatchExpressions(arg);
    allConstants = allConstants && arg.constant;
    if (!arg.constant) {
      argsToWatch.push.apply(argsToWatch, arg.toWatch);
    }
  });
  ast.constant = allConstants;
  ast.toWatch = ast.filter ? argsToWatch : [ast];
  break;
```

For assignments, the input is the node itself:

src/parse.js

```
case AST.AssignmentExpression:
  markConstantAndWatchExpressions(ast.left);
  markConstantAndWatchExpressions(ast.right);
  ast.constant = ast.left.constant && ast.right.constant;
  ast.toWatch = [ast];
  break;
```

For unary operator expressions we should watch the inputs of the argument - there's no need to apply the operator again except when the argument changes:

src/parse.js

```
case AST.UnaryExpression:
  markConstantAndWatchExpressions(ast.argument);
  ast.constant = ast.argument.constant;
  ast.toWatch = ast.argument.toWatch;
  break;
```

For binary expressions we need to watch the inputs of both the left and right arguments:

src/parse.js

```
case AST.BinaryExpression:
case AST.LogicalExpression:
  markConstantAndWatchExpressions(ast.left);
  markConstantAndWatchExpressions(ast.right);
  ast.constant = ast.left.constant && ast.right.constant;
  ast.toWatch = ast.left.toWatch.concat(ast.right.toWatch);
  break;
```

We should be careful not to apply this for logical operator expressions though. If we watch both the left and right side inputs, we may end up breaking the short-circuiting behavior of AND and OR. So at this point we need to separate the implementations for `BinaryExpression` and `LogicalExpression`, and set the input of `LogicalExpression` to itself:

src/parse.js

```
case AST.BinaryExpression:
  markConstantAndWatchExpressions(ast.left);
  markConstantAndWatchExpressions(ast.right);
  ast.constant = ast.left.constant && ast.right.constant;
  ast.toWatch = ast.left.toWatch.concat(ast.right.toWatch);
  break;
case AST.LogicalExpression:
  markConstantAndWatchExpressions(ast.left);
  markConstantAndWatchExpressions(ast.right);
  ast.constant = ast.left.constant && ast.right.constant;
  ast.toWatch = [ast];
  break;
```

Finally, for conditional expressions the input is again the expression itself. Here too we need to be careful not to undo the short-circuiting:

src/parse.js

```
case AST.ConditionalExpression:
  markConstantAndWatchExpressions(ast.test);
  markConstantAndWatchExpressions(ast.consequent);
  markConstantAndWatchExpressions(ast.alternate);
  ast.constant =
    ast.test.constant && ast.consequent.constant && ast.alternate.constant;
  ast.toWatch = [ast];
  break;
```

At this point we have an implementation of `markConstantAndWatchExpressions` after which every node in the AST (except for `Program`) will have a `toWatch` array. When possible, it contains the *input nodes* of that node. When input tracking is not possible, the array will contain the node itself. In any case, we can now make use of this information to implement input tracking.

We have the `toWatch` arrays in AST nodes, and expect to have the `inputs` arrays in expression functions. What remains to be done is to connect those two. The AST compiler will need to *separately compile* the input nodes of the main expression node.

Let's first refactor the compiler so that it can actually have several compilation targets. Currently we are emitting all JavaScript into the array `this.state.body` and all variable names into `this.state.vars`. Since we're soon going to need to compile multiple functions, we need to change this so that the body and vars may go into different places depending on what we're currently compiling.

Let's "wrap" the body and vars in the compiler state into an intermediate object called `fn`:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {}
  };
  // ...
};
```

Then, let's set a `computing` attribute on the state to the value `'fn'` before calling `recurse`:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {}
  };
  this.state.computing = 'fn';
  this.recurse(ast);
```


What this says is "whatever you emit, put it in the `fn` object of `state` because that's what we're currently computing". Let's fulfill that requirement by updating all the locations where we emit code or vars to use the `computing` attribute.

In the `AST.Program` branch of `recurse`:

src/parse.js

```
case AST.Program:
  _.forEach(_.initial(ast.body), function(stmt) {
    this.state[this.state.computing].body.push(this.recurse(stmt), ';');
  }, this);
  this.state[this.state.computing].body.push(
    'return ', this.recurse(_.last(ast.body)), ';');
  break;
```

In the `AST.LogicalExpression` branch of `recurse`:

src/parse.js

```
case AST.LogicalExpression:
  intoId = this.nextId();
  this.state[this.state.computing].body.push(
    this.assign(intoId, this.recurse(ast.left)));
  this.if_(ast.operator === '&&' ? intoId : this.not(intoId),
    this.assign(intoId, this.recurse(ast.right)));
  return intoId;
```

In the `AST.ConditionalExpression` branch of `recurse`:

src/parse.js

```
case AST.ConditionalExpression:
  intoId = this.nextId();
  var testId = this.nextId();
  this.state[this.state.computing].body.push(
    this.assign(testId, this.recurse(ast.test)));
  this.if_(testId,
    this.assign(intoId, this.recurse(ast.consequent)));
  this.if_(this.not(testId),
    this.assign(intoId, this.recurse(ast.alternate)));
  return intoId;
```

In `nextId`:

src/parse.js


```
ASTCompiler.prototype.nextId = function(skip) {
  var id = 'v' + (this.state.nextId++);
  if (!skip) {
    this.state[this.state.computing].vars.push(id);
  }
  return id;
};
```

In `if_`:

src/parse.js

```
ASTCompiler.prototype.if_ = function(test, consequent) {
  this.state[this.state.computing].body.push(
    'if(', test, '){', consequent, '}');
};
```

And in `addEnsureSafeMemberName`, `addEnsureSafeObject`, and `addEnsureSafeFunction`:

src/parse.js

```
ASTCompiler.prototype.addEnsureSafeMemberName = function(expr) {
  this.state[this.state.computing].body.push(
    'ensureSafeMemberName(' + expr + ');');
};
ASTCompiler.prototype.addEnsureSafeObject = function(expr) {
  this.state[this.state.computing].body.push(
    'ensureSafeObject(' + expr + ');');
};
ASTCompiler.prototype.addEnsureSafeFunction = function(expr) {
  this.state[this.state.computing].body.push(
    'ensureSafeFunction(' + expr + ');');
};
```

Now that we've changed the location of our generated code, we need to also change where we read it from when generating the function:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
```



```
    filters: {}
  };
  this.state.computing = 'fn';
  this.recurse(ast);
  var fnString = this.filterPrefix() +
    'var fn=function(s,l){' +
    (this.state.fn.vars.length ?
      'var ' + this.state.fn.vars.join(',') + ';' :
      '') +
    this.state.fn.body.join('') +
    '}; return fn;';
  /* jshint -W054 */
  var fn = new Function(
    'ensureSafeMemberName',
    'ensureSafeObject',
    'ensureSafeFunction',
    'ifDefined',
    'filter',
    fnString)(
      ensureSafeMemberName,
      ensureSafeObject,
      ensureSafeFunction,
      ifDefined,
      filter);
  /* jshint +W054 */
  fn.literal = isLiteral(ast);
  fn.constant = ast.constant;
  return fn;
};
```

The reason we did all this refactoring is that now we can reuse all that compilation code to also compile the input functions based on the `toWatch` attribute of AST nodes. We're going to track the generated inputs using an `inputs` array in the compiler state:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    inputs: []
  };
  // ...
};
```

The compilation of input functions will be done right before we compile the main expression function itself:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    inputs: []
  };
  _.forEach(getInputs(ast.body), function(input) {

  });
  this.state.computing = 'fn';
  this.recurse(ast);
  // ...
};
```

The `getInputs` helper function used here is where we get the inputs of the top-level AST node. We only do it if the program body consists of one expression and if the expression's inputs are something else than the expression itself:

src/parse.js

```
function getInputs(ast) {
  if (ast.length !== 1) {
    return;
  }
  var candidate = ast[0].toWatch;
  if (candidate.length !== 1 || candidate[0] !== ast[0]) {
    return candidate;
  }
}
```

Inside the loop we will now compile each input expression function. We generate a unique "input key" for each input, initialize the compiler state for it, and then set it as the value of `computing`. When we then call `recurse`, the generated code will go into the right place. Finally we generate the final `return` statement for the function, and add the input key to the `inputs` array.

src/parse.js


```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    inputs: []
  };
  _.forEach(getInputs(ast.body), function(input, idx) {
    var inputKey = 'fn' + idx;
    this.state[inputKey] = {body: [], vars: []};
    this.state.computing = inputKey;
    this.state[inputKey].body.push('return ' + this.recurse(input) + ';');
    this.state.inputs.push(inputKey);
  }, this);
  this.state.computing = 'fn';
  this.recurse(ast);
  // ...
};
```

At this point we have the *names* of each generated input expression in `state.inputs` and also the generated code nested inside the state. What remains to be done is the attachment of the generated input functions into the main expression function. We'll do it using a method called `watchFns`:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    inputs: []
  };
  _.forEach(getInputs(ast.body), function(input, idx) {
    var inputKey = 'fn' + idx;
    this.state[inputKey] = {body: [], vars: []};
    this.state.computing = inputKey;
    this.state[inputKey].body.push('return ' + this.recurse(input) + ';');
    this.state.inputs.push(inputKey);
  }, this);
  this.state.computing = 'fn';
  this.recurse(ast);
  var fnString = this.filterPrefix() +
    'var fn=function(s,l){' +
    (this.state.fn.vars.length ?
```



```
      'var ' + this.state.fn.vars.join(',') + ';' :
      ''
    ) +
    this.state.fn.body.join('') +
    '};' +
    this.watchFns() +
    ' return fn;';
  // ...
};
```

What `watchFns` does is go through the `inputs` we have collected into the compiler state. It collects an array of JavaScript code fragments and joins them into a single string that it then returns:

src/parse.js

```
ASTCompiler.prototype.watchFns = function() {
  var result = [];
  _.forEach(this.state.inputs, function(inputName) {

  }, this);
  return result.join('');
};
```

A JavaScript function is generated for each of the inputs, based on the vars and body stored in the compiler state for that input key. The function is generated similarly as the main expression function: Var declarations first, then the body. The function takes a single argument, which is (presumably) a scope:

src/parse.js

```
ASTCompiler.prototype.watchFns = function() {
  var result = [];
  _.forEach(this.state.inputs, function(inputName) {
    result.push('var ', inputName, '=function(s) {',
      (this.state[inputName].vars.length ?
        'var ' + this.state[inputName].vars.join(',') + ';' :
        ''
      ),
      this.state[inputName].body.join(''),
    '};');
  }, this);
  return result.join('');
};
```

A statement that actually puts the `inputs` array in the generated main function is also generated. It contains references to all the generated input functions:

src/parse.js


```
ASTCompiler.prototype.watchFns = function() {
  var result = [];
  _.forEach(this.state.inputs, function(inputName) {
    result.push('var ', inputName, '=function(s) {',
      (this.state[inputName].vars.length ?
        'var ' + this.state[inputName].vars.join(',') + ';' :
        ''
      ),
      this.state[inputName].body.join(''),
    '};');
  }, this);
  if (result.length) {
    result.push('fn.inputs = [', this.state.inputs.join(','), '];');
  }
  return result.join('');
};
```

The one remaining factor we have to consider before our test passes is the role of locals in input expressions. As you may have noticed, the input functions generated in `watchFns` do not take the locals `l` argument. That is because in inputs we do not consider locals at all.

The problem with this is that we have some code generated in the `AST.Identifier` branch of `recurse` that requires `l` to always be present. We need to change this. Let's set an attribute on the compiler that marks whether what is currently being compiled are input functions or the main expression function:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    inputs: []
  };
  this.stage = 'inputs';
  _.forEach(getInputs(ast.body), function(input, idx) {
    var inputKey = 'fn' + idx;
    this.state[inputKey] = {body: [], vars: []};
    this.state.computing = inputKey;
    this.state[inputKey].body.push('return ' + this.recurse(input) + ';');
    this.state.inputs.push(inputKey);
  }, this);
  this.stage = 'main';
  this.state.computing = 'fn';
  this.recurse(ast);
  // ...
};
```


Now, when generating the code for `AST.Identifier`, we'll do the locals property check differently based on the compiler stage. When compiling inputs, it's always going to be `false`. When compiling the main function, it will be based on the `getHasOwnProperty` check as before:

src/parse.js

```
case AST.Identifier:
  ensureSafeMemberName(ast.name);
  intoId = this.nextId();
  var localsCheck;
  if (this.stage === 'inputs') {
    localsCheck = 'false';
  } else {
    localsCheck = this.getHasOwnProperty('l', ast.name);
  }
  this.if_(localsCheck,
    this.assign(intoId, this.nonComputedMember('l', ast.name)));
  if (create) {
    this.if_(this.not(localsCheck) +
             ' && s && ' +
             this.not(this.getHasOwnProperty('s', ast.name)),
      this.assign(this.nonComputedMember('s', ast.name), '{}'));
  }
  this.if_(this.not(localsCheck) + ' && s',
    this.assign(intoId, this.nonComputedMember('s', ast.name)));
  if (context) {
    context.context = localsCheck + '?l:s';
    context.name = ast.name;
    context.computed = false;
  }
  this.addEnsureSafeObject(intoId);
  return intoId;
```

And there we have an implementation of input tracking. It ended up involving quite a large number of changes, but it can also be a very powerful optimization. In summary, this is what now happens:

1. The compiler visits each AST node and sets its `toWatch` attribute based on its input nodes, when applicable.
2. The compiler generates a separate JavaScript function body for each input of the top-level expression. The inputs are determined based on the `toWatch` attribute populated in the previous step.
3. The compiler's `watchFns` method generates input expression functions for each of the bodies copiled in the previous step. It attaches them to the `inputs` attribute of the main expression function.
4. An inputs watch delegate is attached to the expression when it is being watched.
5. Instead of watching the main expression function, the inputs watch delegate watches each of the functions it finds in `inputs`.

Stateful Filters

As we've implemented the constant optimizations and input tracking in this chapter, we've seen how filter call expressions are considered different from regular function call expressions: Filter expressions are constant if their arguments are constant and only their inputs are watched for changes.

There is a fairly significant assumption underlying this implementation, which is that *a filter is always expected to return the same result given the same inputs*. In other words, it is expected that filters are pure functions.

For most filters - including the filter filter that we implemented earlier - this is in fact the case. It doesn't matter when you call the filter filter, or how many times you call it. It will always return the same value given the same input arguments. This is a nice property for functions to have, both because it makes them easier to understand and because it allows for the kinds of optimizations we have made: When the filter filter is used in an expression, we don't actually need to recalculate it unless its input array (or one of its other arguments) changes. This ends up being a very significant performance gain in many applications.

Sometimes, however, this assumption does not hold. It is conceivable to have a filter whose value may change even when its inputs don't. One example of such a filter is one that embeds the current time in the expression output. Angular allows you to put a special `$stateful` attribute on these kinds of filters. If you set it to `true`, the constant and input tracking optimizations are not applied to it:

test/scope_spec.js

```
it('allows $stateful filter value to change over time', function(done) {

  register('withTime', function() {
    return _.extend(function(v) {
      return new Date().toISOString() + ': ' + v;
    }, {
      $stateful: true
    });
  });

  var listenerSpy = jasmine.createSpy();
  scope.$watch('42 | withTime', listenerSpy);
  scope.$digest();
  var firstValue = listenerSpy.calls.mostRecent().args[0];

  setTimeout(function() {
    scope.$digest();
    var secondValue = listenerSpy.calls.mostRecent().args[0];
    expect(secondValue).not.toEqual(firstValue);
    done();
  }, 100);
});
```


We need to allow the global `register` function to be used in `scope_spec.js` before JSHint will allow this code to pass:

test/scope_spec.js

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

This test fails because the watch is evaluating to the same value each time. That's because it has a constant input which doesn't change. What we need to do is go to `parse.js` and disable the constant and input tracking optimization for stateful filter calls.

src/parse.js

```
case AST.CallExpression:
  var stateless = ast.filter && !filter(ast.callee.name).$stateful;
  allConstants = stateless ? true : false;
  argsToWatch = [];
  _.forEach(ast.arguments, function(arg) {
    markConstantAndWatchExpressions(arg);
    allConstants = allConstants && arg.constant;
    if (!arg.constant) {
      argsToWatch.push.apply(argsToWatch, arg.toWatch);
    }
  });
  ast.constant = allConstants;
  ast.toWatch = stateless ? argsToWatch : [ast];
  break;
```

External Assignment

When expressions are evaluated, the point is usually to obtain their current value on a Scope. That is what the whole data binding system in Angular is based on.

In some cases, a different mode of calling expressions is called for. That is to *assign new values* for them on a Scope. Such a mode is exposed through the `assign` method that expression functions have.

For example, if you have an expression like `a.b`. You can call it on a scope:

```
var exprFn = parse('a.b');
var scope = {a: {b: 42}};
exprFn(scope); // => 42
```

But you can also *assign* it on a scope, which means that we actually go and *replace* the expression's value on the given scope:

```
var exprFn = parse('a.b');
var scope = {a: {b: 42}};
exprFn.assign(scope, 43);
scope.a.b; // => 43
```

Later in the book we will make use of `assign` as we implement two-way binding for isolate scopes.

This is essentially the logic of `AssignmentExpression` exposed externally as a function. Expressed as test cases, one should be able to assign both simple identifiers and nested members:

test/parse_spec.js

```
it('allows calling assign on identifier expressions', function() {
  var fn = parse('anAttribute');
  expect(fn.assign).toBeDefined();

  var scope = {};
  fn.assign(scope, 42);
  expect(scope.anAttribute).toBe(42);
});

it('allows calling assign on member expressions', function() {
  var fn = parse('anObject.anAttribute');
  expect(fn.assign).toBeDefined();

  var scope = {};
  fn.assign(scope, 42);
  expect(scope.anObject).toEqual({anAttribute: 42});
});
```

What we're going to do is generate one more expression function, which will be attached as the `assign` method of the main expression function. It has its own compiler state and compiler stage:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    assign: {body: [], vars: []},
    inputs: []
  };
  this.stage = 'inputs';
  _.forEach(getInputs(ast.body), function(input, idx) {
```



```
    var inputKey = 'fn' + idx;
    this.state[inputKey] = {body: [], vars: []};
    this.state.computing = inputKey;
    this.state[inputKey].body.push('return ' + this.recurse(input) + ';');
    this.state.inputs.push(inputKey);
  }, this);
  this.stage = 'assign';

  this.stage = 'main';
  this.state.computing = 'fn';
  this.recurse(ast);
  // ...
};
```

The assign method will only be generated if we can form something called an "assignable AST" for the expression:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    assign: {body: [], vars: []},
    inputs: []
  };
  this.stage = 'inputs';
  _.forEach(getInputs(ast.body), function(input, idx) {
    var inputKey = 'fn' + idx;
    this.state[inputKey] = {body: [], vars: []};
    this.state.computing = inputKey;
    this.state[inputKey].body.push('return ' + this.recurse(input) + ';');
    this.state.inputs.push(inputKey);
  }, this);
  this.stage = 'assign';
  var assignable = assignableAST(ast);
  if (assignable) {

  }
  this.stage = 'main';
  this.state.computing = 'fn';
  this.recurse(ast);
  // ...
};
```

An assignable AST is formed only if the expression has just one statement whose type is either identifier or member:

src/parse.js

```
function isAssignable(ast) {
  return ast.type === AST.Identifier || ast.type == AST.MemberExpression;
}
function assignableAST(ast) {
  if (ast.body.length == 1 && isAssignable(ast.body[0])) {

  }
}
```

What the assignable AST actually is is the original expression wrapped inside an `AST.AssignmentExpression`. The right hand side of the assignment is special - its type is `AST.NGValueParameter`, and it denotes a parameterized value that is supplied at runtime (as an argument to the `assign` method):

src/parse.js

```
function assignableAST(ast) {
  if (ast.body.length == 1 && isAssignable(ast.body[0])) {
    return {
      type: AST.AssignmentExpression,
      left: ast.body[0],
      right: {type: AST.NGValueParameter}
    };
  }
}
```

Now that we (potentially) have the assignable AST, we can go ahead and compile it using `recurse`:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    assign: {body: [], vars: []},
    inputs: []
  };
  this.stage = 'inputs';
  _.forEach(getInputs(ast.body), function(input, idx) {
```



```
    var inputKey = 'fn' + idx;
    this.state[inputKey] = {body: [], vars: []};
    this.state.computing = inputKey;
    this.state[inputKey].body.push('return ' + this.recurse(input) + ';');
    this.state.inputs.push(inputKey);
  }, this);
  this.stage = 'assign';
  var assignable = assignableAST(ast);
  if (assignable) {
    this.state.computing = 'assign';
    this.state.assign.body.push(this.recurse(assignable));
  }
  this.stage = 'main';
  this.state.computing = 'fn';
  this.recurse(ast);
  // ...
};
```

From the compilation result we'll generate the assign function. It takes three arguments: A scope, the value to assign, and the locals. We put the code of this function into a new variable called `extra` and append it to the main expression function as well:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  var extra = '';
  markConstantAndWatchExpressions(ast);
  this.state = {
    nextId: 0,
    fn: {body: [], vars: []},
    filters: {},
    assign: {body: [], vars: []},
    inputs: []
  };
  this.stage = 'inputs';
  _.forEach(getInputs(ast.body), function(input, idx) {
    var inputKey = 'fn' + idx;
    this.state[inputKey] = {body: [], vars: []};
    this.state.computing = inputKey;
    this.state[inputKey].body.push('return ' + this.recurse(input) + ';');
    this.state.inputs.push(inputKey);
  }, this);
  this.stage = 'assign';
  var assignable = assignableAST(ast);
  if (assignable) {
    this.state.computing = 'assign';
    this.state.assign.body.push(this.recurse(assignable));
    extra = 'fn.assign = function(s,v,l){' +
```



```
      (this.state.assign.vars.length ?
        'var ' + this.state.assign.vars.join(',') + ';' :
        ''
      ) +
      this.state.assign.body.join('') +
      '};';
  }
  this.stage = 'main';
  this.state.computing = 'fn';
  this.recurse(ast);
  var fnString = this.filterPrefix() +
    'var fn=function(s,l){' +
    (this.state.fn.vars.length ?
      'var ' + this.state.fn.vars.join(',') + ';' :
      ''
    ) +
    this.state.fn.body.join('') +
    '};' +
    this.watchFns() +
    extra +
    ' return fn;';
  /* jshint -W054 */
  var fn = new Function(
    'ensureSafeMemberName',
    'ensureSafeObject',
    'ensureSafeFunction',
    'ifDefined',
    'filter',
    fnString)(
      ensureSafeMemberName,
      ensureSafeObject,
      ensureSafeFunction,
      ifDefined,
      filter);
  /* jshint +W054 */
  fn.literal = isLiteral(ast);
  fn.constant = ast.constant;
  return fn;
};
```

The only missing part now is the compilation of the `NGValueParameter` node in `recurse`. Its job is to denote the location in the generated code where the parameter given to `assign` should be applied. The name of that parameter in the generated code is `v`, so we can simply compile this AST node to `v`:

src/parse.js

```
case AST.NGValueParameter:
  return 'v';
```

Summary

Combining the Scope system with the expression system has resulted in a powerful and expressive dirty-checking implementation, which can be used as a standalone tool, but will also prove very useful when combined with the directive system later in the book.

The combination of scopes and expressions has also allowed for some new powerful features: Constant optimization, one-time binding, and input tracking. They provide potentially very significant performance improvements to application developers.

In this chapter you have learned:

- How `Scope` uses the expression parser to allow expressions to be used as watches.
- How `Scope` uses the expression parser to allow expressions to be evaled.
- How expressions get marked as `constant` and/or `literal`.
- That the expression parser sometimes provides *watch delegates* that bypass the normal watch behavior.
- How the constant watch delegate removes watches for constant values immediately after their first invocation.
- How one-time binding works, and how it's implemented with a one-time watch delegate that removes the watch after it has stabilized in some defined value.
- That one-time binding supports array and object literals by waiting for all of their contained items to stabilize.
- How the expression parser uses *input tracking* to minimize the evaluation of compound expressions by only doing it when their input expressions change.
- How filters can be marked as stateful, causing the constant and input tracking optimizations to be disabled for them.
- How the `assign` method on expression functions allows the value of the expression to be reassigned on a given scope.

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 and some people criticise 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 10

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 to start with 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 something 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, we'll get two different module objects:

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 nonexistent 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, a 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, which we introduce as a local variable in the function:

src/loader.js

```
var createModule = function(name, requires, modules) {
  if (name === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid module name';
  }
  var invokeQueue = [];
  var moduleInstance = {
    name: name,
    requires: requires,
    constant: function(key, value) {
      invokeQueue.push(['constant', [key, value]]);
    },
    _invokeQueue: 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() {
  angular.module('myModule', ['myOtherModule']);
  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 have 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;
    }
  };

  function invoke(fn) {
    var args = _.map(fn.$inject, function(token) {
      return cache[token];
    });
    return fn.apply(null, args);
  }

  _.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];
    },
    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

```
function invoke(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

```
function invoke(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

```
function invoke(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 the $inject annotation of a function 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;
    }
```



```
  };

  function annotate(fn) {
    return fn.$inject;
  }

  function invoke(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);
  }

  _.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];
    },
    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

```
function annotate(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

```
function annotate(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

```
function annotate(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


```
function annotate(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


```
function annotate(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 ends. 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

```
function annotate(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];
    });
  }
}
```

Strict Mode

The approach of determining a function's dependencies from its source code is one of Angular's most controversial features. This is partly because of the unexpected, magical nature of the feature, but there are also some very practical issues with this approach: If you choose to minify your JavaScript code before releasing it - as many people justifiably do - the source code of your application changes. Crucially, the *argument names* of your functions change when you run a minifier such as UglifyJS or Closure Compiler. This breaks non-annotated dependency injection.

For this reason, many people choose to not use the non-annotated dependency injection feature (or at least generate the annotations using a tool like ng-annotate). Angular can help sticking to this decision by enforcing a *strict dependency injection mode*, in which it will throw an error if you ever try to inject a function that has not been explicitly annotated.

Strict dependency injection is enabled by passing a boolean flag as a second argument to `createInjector`:

test/injector_spec.js

```
it('throws when using a non-annotated fn in strict mode', function() {
  var injector = createInjector([], true);

  var fn = function(a, b, c) { };

  expect(function() {
    injector.annotate(fn);
  }).toThrow();
});
```

The `createInjector` function accepts this second argument, which must be `true` (not just truthy) to enable strict mode:

src/injector.js

```
function createInjector(modulesToLoad, strictDi) {
  var cache = {};
  var loadedModules = {};
  strictDi = (strictDi === true);
  // ...
}
```

In `annotate`, when we get to parsing function arguments, we'll check if we're in strict mode and throw an exception if that is the case:

src/injector.js

```
function annotate(fn) {
  if (_.isArray(fn)) {
    return fn.slice(0, fn.length - 1);
  } else if (fn.$inject) {
    return fn.$inject;
  } else if (!fn.length) {
    return [];
  } else {
    if (strictDi) {
      throw 'fn is not using explicit annotation and '+
            'cannot be invoked in strict mode';
    }
    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];
    });
  }
}
```

Strict mode is a nice little feature that you'll definitely want to enable if you are minifying your code. If a non-annotated injection point sneaks through, you'll get a clear error telling you about it, instead of an obscure error about an undeclared dependency with a name you've never heard of.

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

```
function invoke(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

```
function instantiate(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

```
function instantiate(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

```
function instantiate(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 strict DI mode helps make sure you're not accidentally using non-annotated dependency injection when you don't mean to.
- 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 11

Providers

When it comes to the actual logic of dependency injection, our injector is already pretty much done. It can do all the types on injection the Angular.js injector can. For the remainder of this part of the book, we can focus on building the APIs with which you can *create* those application components that you actually inject.

So far the only way to add something to our injector has been the `constant`, which, to all intents and purposes, is little more than a direct "put" of a value to the injector's cache.

In this chapter we'll focus on the concept of *providers*. Providers are objects that know how to make dependencies. They're useful when you actually need to run some code when constructing a dependency. They're also useful when your dependency has other dependencies, which you'd also like to have injected.

Providers are the underlying mechanism for all of the other types of application components in Angular, apart from constants. Services, factories, and values will all be built on providers.

The Simplest Possible Provider: An Object with A `$get` Method

Generally speaking, what Angular calls a provider is any JavaScript object that has a method attached to it called `$get`. When you provide such an object to the injector, it will call that `$get` method and treat its return value as the actual dependency value:

test/injector_spec.js

```
it('allows registering a provider and uses its $get', function() {
  var module = angular.module('myModule', []);
  module.provider('a', {
    $get: function() {
      return 42;
    }
  });
```

```
  var injector = createInjector(['myModule']);

  expect(injector.has('a')).toBe(true);
  expect(injector.get('a')).toBe(42);
});
```

So here, `a` is 42, and the *provider* for `a` is the object `{$get: function() { return 42; }}`. This indirection isn't very useful the way we're using it now, but as we will see, it gives us a chance to conveniently *configure* `a` in ways that aren't possible for constants.

To implement the kind of provider needed by this test, let's begin with the module loader. It needs to have a method for registering a provider. That method should put the registration invocation in the invoke queue:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: function(key, value) {
    invokeQueue.push(['constant', [key, value]]);
  },
  provider: function(key, provider) {
    invokeQueue.push(['provider', [key, provider]]);
  },
  _invokeQueue: invokeQueue
};
```

There's some repetition forming in the component registration here, so let's introduce a generic "queueing function" with which we can implement `module.constant` and `module.provider` with minimal duplication of effort:

src/loader.js

```
var createModule = function(name, requires, modules) {
  if (name === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid module name';
  }
  var invokeQueue = [];

  var invokeLater = function(method) {
    return function() {
      invokeQueue.push([method, arguments]);
      return moduleInstance;
    };
  };
```



```
  var moduleInstance = {
    name: name,
    requires: requires,
    constant: invokeLater('constant'),
    provider: invokeLater('provider'),
    _invokeQueue: invokeQueue
  };

  modules[name] = moduleInstance;
  return moduleInstance;
};
```

`invokeLater` returns a function that has been preconfigured for a particular type of application component, or rather, a particular method of `$provide`. The function pushes to the invoke queue an array with that method name and any arguments given. Notice that we also return the module instance from registration, so that the chaining of registrations is possible: `module.constant('a', 42).constant('b', 43)`.

We still need the code in the injector's `$provide` object that can handle the new queue item. For now, let's simply call the `$get` method of the provider and put the return value in the cache. That satisfies our first test case:

src/injector.js

```
var $provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    cache[key] = value;
  },
  provider: function(key, provider) {
    cache[key] = provider.$get();
  }
};
```

Injecting Dependencies To The `$get` Method

At this point we've gained very little by introducing providers. We've just "hidden" the actual dependency behind a `$get` method the injector needs to call.

The benefits this buys us begin to become visible when we think of cases where constructing an application component has *its own* dependencies. That is, when our dependencies have dependencies. So far we've had no way to really model this, but as we now have the `$get` method, we can call that method *with dependency injection*:

test/injector_spec.js


```
it('injects the $get method of a provider', function() {
  var module = angular.module('myModule', []);
  module.constant('a', 1);
  module.provider('b', {
    $get: function(a) {
      return a + 2;
    }
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('b')).toBe(3);
});
```

This one is easy: We need to call `provider.$get` with dependency injection. In the last chapter we implemented the function that does exactly that: `invoke`.

src/injector.js

```
var $provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    cache[key] = value;
  },
  provider: function(key, provider) {
    cache[key] = invoke(provider.$get, provider);
  }
};
```

Note that we bind the receiver `this` to the provider object in the invocation. Because `$get` is a method of the provider object, its `this` should be bound to that object.

So, `invoke` is not only a method of the injector you can call for your own functions, but it is also used internally by the injector to inject dependencies to `provider.$get` methods.

Lazy Instantiation of Dependencies

Now that we can have dependencies between dependencies, we'll need to start thinking about the order in which things are done. At the time when we create a dependency, do we have all of *its* dependencies available? Consider the following case, where `b` depends on `a`, but `b` is registered *before* `a`:

test/injector_spec.js

```
it('injects the $get method of a provider lazily', function() {
  var module = angular.module('myModule', []);
  module.provider('b', {
    $get: function(a) {
      return a + 2;
    }
  });
  module.provider('a', {$get: _.constant(1)});

  var injector = createInjector(['myModule']);

  expect(injector.get('b')).toBe(3);
});
```

The test fails because as we try to invoke the `$get` method for `b`, its dependency is not yet available.

If this was really how the Angular injector worked, you would have to be very careful to load your code in the order of its dependencies. Fortunately, this is *not* how the injector works. Instead, the injector invokes those `$get` methods *lazily*, only when their return values are needed. So, even though `b` is registered as the first thing, its `$get` method should not be called at that point. It should be called only when we ask the injector for `b` for the first time. By that time `a` will also have been registered.

Since we can't call a provider's `$get` at the time when we drain the invoke queue, we'll need to keep the provider object around so we can call it later. We have the `cache` object for storage, but it doesn't really make sense to put the provider there, because the cache is for dependency *instances*, not the providers that produce them. What we need to do is split the cache in two: One cache for all the providers, and the other for all the instances:

src/injector.js

```
function createInjector(modulesToLoad, strictDi) {
  var providerCache = {};
  var instanceCache = {};
  var loadedModules = {};

  // ...

}
```

Then, in `$provide`, we'll put constants in `instanceCache` and providers in `providerCache`. When caching a provider, we attach a `'Provider'` suffix to the cache key. So registering the provider for `'a'` will cause the key `'aProvider'` to be put in the provider cache. We need to have a clear separation between instances and their providers and this naming enforces that separation:

src/injector.js

```
var $provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    providerCache[key + 'Provider'] = provider;
  }
};
```

As we then need to get a hold of a dependency, either for injection or because someone asked for it directly, we need to look at both of these caches. Let's create a new local method inside `createInjector` that does this. The function `getService` will, given the name of a dependency, first look for it in the instance cache, and then in the provider cache. If it finds a provider for the dependency, it invokes it, like we did earlier in `$provide.provider`:

src/injector.js

```
function getService(name) {
  if (instanceCache.hasOwnProperty(name)) {
    return instanceCache[name];
  } else if (providerCache.hasOwnProperty(name + 'Provider')) {
    var provider = providerCache[name + 'Provider'];
    return invoke(provider.$get, provider);
  }
}
```

From `invoke` we can now call our new function during the dependency lookup loop. If the lookup from `locals` fails, we look for the token in the injector's caches:

src/injector.js

```
function invoke(fn, self, locals) {
  var args = _.map(annotate(fn), function(token) {
    if (_.isString(token)) {
      return locals && locals.hasOwnProperty(token) ?
        locals[token] :
        getService(token);
    } else {
      throw 'Incorrect injection token! Expected a string, got '+token;
    }
  });
  if (_.isArray(fn)) {
    fn = _.last(fn);
  }
  return fn.apply(self, args);
}
```


Finally, we also need to update the way the `injector.get` and `injector.has` methods work. The `get` method can just be an alias to the new `getService` function. In `has` we need to do a property check in both caches, and use the `'Provider'` suffix when checking the provider cache:

src/injector.js

```
return {
  has: function(key) {
    return instanceCache.hasOwnProperty(key) ||
      providerCache.hasOwnProperty(key + 'Provider');
  },
  get: getService,
  annotate: annotate,
  invoke: invoke
};
```

So, a dependency from a provider only gets instantiated when its either injected somewhere, or explicitly asked for through `injector.get`. If no one ever asks for a dependency, it will never actually come to be - its provider's `$get` never gets called.

You can check for the existence of a dependency through `injector.has`, which does not cause the dependency to be instantiated. It just checks if there's either a dependency instance or a provider for it available.

Making Sure Everything Is A Singleton

You may have heard it said that "everything in Angular is a singleton". That is generally true: When you use the same dependency in two different places, you will have a reference to the same exact object.

This is not how our current injector implementation works though. When you ask for a provider-created component twice, you will get two results that don't necessarily point to the same thing:

test/injector_spec.js

```
it('instantiates a dependency only once', function() {
  var module = angular.module('myModule', []);
  module.provider('a', {$get: function() { return {}; }});

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(injector.get('a'));
});
```


This is because we call `$get` twice (through `getService()`) and each time it gives us a new object. This is not how it should be.

The solution to this is to simply put the return value of a provider invocation to the instance cache right after we get it. So the next time someone needs the same dependency, it will just be available in the instance cache. We will never call a provider's `$get` method twice:

src/injector.js

```
function getService(name) {
  if (instanceCache.hasOwnProperty(name)) {
    return instanceCache[name];
  } else if (providerCache.hasOwnProperty(name + 'Provider')) {
    var provider = providerCache[name + 'Provider'];
    var instance = instanceCache[name] = invoke(provider.$get);
    return instance;
  }
}
```

Circular Dependencies

As we now have dependencies depending on other dependencies, we've introduced the possibility of a *circular dependency chain* occurring: If A depends on B, B depends on C, and C depends on A, that's a problem because there's no way for us to construct A, B, or C without getting into an infinite loop - and a corresponding stack overflow error. We would perhaps like a more informative error message to be shown:

test/injector_spec.js

```
it('notifies the user about a circular dependency', function() {
  var module = angular.module('myModule', []);
  module.provider('a', {$get: function(b) { }});
  module.provider('b', {$get: function(c) { }});
  module.provider('c', {$get: function(a) { }});

  var injector = createInjector(['myModule']);

  expect(function() {
    injector.get('a');
  }).toThrowError(/Circular dependency found/);
});
```

The trick here has two parts to it: As we construct a dependency, before invoking its `$get` method we'll put a special marker value into the instance cache. The marker value says "we're currently constructing this dependency". If, then, at some point we see this marker

value when looking up a dependency, that means we're trying to look up something that we're also currently constructing and we have a circle.

We can just use an empty object as the marker value. The important thing is that it's not equal to anything else. Let's introduce the marker at the beginning of `injector.js`:

test/injector_spec.js

```
var FN_ARGS = /^function\s*[^\(]*\(\s*([^\)]*)\)/m;
var FN_ARG = /^\s*(_?)(\S+?)\1\s*$/;
var STRIP_COMMENTS = /(\/\/.*$)|(\/\*.*?\*\/)/mg;
var INSTANTIATING = { };
```

In `getService` we'll put the marker in before calling a provider, and check for it when looking up an instance:

src/injector.js

```
function getService(name) {
  if (instanceCache.hasOwnProperty(name)) {
    if (instanceCache[name] === INSTANTIATING) {
      throw new Error('Circular dependency found');
    }
    return instanceCache[name];
  } else if (providerCache.hasOwnProperty(name + 'Provider')) {
    instanceCache[name] = INSTANTIATING;
    var provider = providerCache[name + 'Provider'];
    var instance = instanceCache[name] = invoke(provider.$get);
    return instance;
  }
}
```

As the dependency is finally constructed, it'll replace the `INSTANTIATING` marker in the instance cache. But something could also go wrong during instantiation, in which case we don't want that marker to be left in the cache:

test/injector_spec.js

```
it('cleans up the circular marker when instantiation fails', function() {
  var module = angular.module('myModule', []);
  module.provider('a', {$get: function() {
    throw 'Failing instantiation!';
  }});

  var injector = createInjector(['myModule']);

  expect(function() {
    injector.get('a');
  }).toThrow('Failing instantiation!');
  expect(function() {
    injector.get('a');
  }).toThrow('Failing instantiation!');
});
```


What we're doing here is checking that if you try and fail to instantiate `a` twice, it'll try to invoke the provider on both times. Our current implementation does not do that, because on the first time it leaves the `INSTANTIATING` marker in the instance cache, and on the second time it sees that and draws the conclusion that this is a circular dependency. We should make sure we don't leave the marker in even if the invocation fails:

src/injector.js

```
function getService(name) {
  if (instanceCache.hasOwnProperty(name)) {
    if (instanceCache[name] === INSTANTIATING) {
      throw new Error('Circular dependency found');
    }
    return instanceCache[name];
  } else if (providerCache.hasOwnProperty(name + 'Provider')) {
    instanceCache[name] = INSTANTIATING;
    try {
      var provider = providerCache[name + 'Provider'];
      var instance = instanceCache[name] = invoke(provider.$get);
      return instance;
    } finally {
      if (instanceCache[name] === INSTANTIATING) {
        delete instanceCache[name];
      }
    }
  }
}
```

Just notifying the user that there's a circular dependency somewhere is not very helpful though. It would be much better to let them know where that problem actually is.

> We usually don't spend much time thinking of error messages in this book, but this is one of those cases where knowing how the error message is constructed really helps you decipher it.

What we'd like is to show the user the path to the dependency where the problem occurred. In our case, reading from right to left:

```
a <- c <- b <- a
```

Updating our existing test case to expect this in the error message:

test/injector_spec.js


```
it('notifies the user about a circular dependency', function() {
  var module = angular.module('myModule', []);
  module.provider('a', {$get: function(b) { }});
  module.provider('b', {$get: function(c) { }});
  module.provider('c', {$get: function(a) { }});

  var injector = createInjector(['myModule']);

  expect(function() {
    injector.get('a');
  }).toThrowError('Circular dependency found: a <- c <- b <- a');
});
```

What we need to do is store the current dependency path in a data structure as we do dependency resolution. Let's introduce an array for it inside `createInjector()`:

src/injector.js

```
function createInjector(modulesToLoad, strictDi) {
  var providerCache = {};
  var instanceCache = {};
  var loadedModules = {};
  var path = [];
  // ...
```

In `getService()` we can treat `path` essentially as a stack. When we start resolving a dependency, we add its name to the front of the path. When we're done, we pop it off:

src/injector.js

```
function getService(name) {
  if (instanceCache.hasOwnProperty(name)) {
    if (instanceCache[name] === INSTANTIATING) {
      throw new Error('Circular dependency found');
    }
    return instanceCache[name];
  } else if (providerCache.hasOwnProperty(name + 'Provider')) {
    path.unshift(name);
    instanceCache[name] = INSTANTIATING;
    try {
      var provider = providerCache[name + 'Provider'];
      var instance = instanceCache[name] = invoke(provider.$get);
      return instance;
    } finally {
      path.shift();
      if (instanceCache[name] === INSTANTIATING) {
        delete instanceCache[name];
      }
    }
  }
}
```


If we run into a circular dependency, we can then just use the current value of `path` to display the problematic dependency path to the user:

src/injector.js

```
function getService(name) {
  if (instanceCache.hasOwnProperty(name)) {
    if (instanceCache[name] === INSTANTIATING) {
      throw new Error('Circular dependency found: ' +
        name + ' <- ' + path.join(' <- '));
    }
    return instanceCache[name];
  } else if (providerCache.hasOwnProperty(name + 'Provider')) {
    path.unshift(name);
    instanceCache[name] = INSTANTIATING;
    try {
      var provider = providerCache[name + 'Provider'];
      var instance = instanceCache[name] = invoke(provider.$get, provider);
      return instance;
    } finally {
      path.shift();
    }
  }
}
```

Now the user can actually see where they have problem!

Provider Constructors

Earlier we defined a provider to be an object thas has a `$get` method. This is true, but when you register a provider to an injector, you can also choose to do it with a *constructor function* that *instantiates* a provider instead of just using a ready-made object:

```
function AProvider() {
  this.$get = function() { return 42; }
}
```

This is a constructor that, when instantiated, results in an object with a `$get` method - a provider. Another such constructor might be:

```
function AProvider() {
  this.value = 42;
}
AProvider.prototype.$get = function() {
  return this.value;
}
```


So using this constructor style, Angular doesn't really care where the `$get` method comes from as long as the resulting object has it. You can fully leverage JavaScript's pseudo-class-style programming facilities and inheritance when working with providers.

As a unit test, here's how this feature should work:

test/injector_spec.js

```
it('instantiates a provider if given as a constructor function', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider() {
    this.$get = function() { return 42; };
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(42);
});
```

What's more is that those constructor functions can also be *injected* with other dependencies:

test/injector_spec.js

```
it('injects the given provider constructor function', function() {
  var module = angular.module('myModule', []);

  module.constant('b', 2);
  module.provider('a', function AProvider(b) {
    this.$get = function() { return 1 + b; };
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(3);
});
```

To enable constructor-style providers, we'll need to check at registration time if the provider given is actually a function. If it is, we need to *instantiate* it, and in the previous chapter we created a function for doing just that: `instantiate`.

src/injector.js

```
provider: function(key, provider) {
  if (_.isFunction(provider)) {
    provider = instantiate(provider);
  }
  providerCache[key + 'Provider'] = provider;
}
```


What we have now are two seemingly interchangeable ways to inject things to a provider: To its constructor, or to its `$get` method. Moreover, the work we did for lazy initialization does not apply to the constructor injection. The provider constructor is instantiated right when its registered. If some of its dependencies have not been registered yet, it won't work.

The key idea with these two different injection points is that they are actually *not* interchangeable, but used for two different purposes.

Two Injectors: The Provider Injector and The Instance Injector

The first difference between the two injection points is that you can inject *other providers* to a provider constructor:

test/injector_spec.js

```
it('injects another provider to a provider constructor function', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider() {
    var value = 1;
    this.setValue = function(v) { value = v; };
    this.$get = function() { return value; };
  });
  module.provider('b', function BProvider(aProvider) {
    aProvider.setValue(2);
    this.$get = function() { };
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(2);
});
```

So far we've only been injecting instances, such as constants and the return values of provider `$get` methods. Here, we actually inject a provider: The provider for `b` has a dependency to the provider for `a`, so it expects `aProvider` to be injected. It then configures `aProvider` by calling its `setValue()` method. When we get the `a` instance from the injector, we see that the method call has in fact happened.

An initial implementation of provider injection that makes our test pass could be to just add a `providerCache` lookup to `getService`:

src/injector.js


```
function getService(name) {
  if (instanceCache.hasOwnProperty(name)) {
    if (instanceCache[name] === INSTANTIATING) {
      throw new Error('Circular dependency found: ' +
        name + ' <- ' + path.join(' <- '));
    }
    return instanceCache[name];
  } else if (providerCache.hasOwnProperty(name)) {
    return providerCache[name];
  } else if (providerCache.hasOwnProperty(name + 'Provider')) {
    path.unshift(name);
    instanceCache[name] = INSTANTIATING;
    try {
      var provider = providerCache[name + 'Provider'];
      var instance = instanceCache[name] = invoke(provider.$get);
      return instance;
    } finally {
      path.shift();
      if (instanceCache[name] === INSTANTIATING) {
        delete instanceCache[name];
      }
    }
  }
}
```

We now look at `providerCache` for two different purposes: When looking for a provider to instantiate an instance, and when just looking for a provider as-is.

It is not quite this simple, however. Turns out you can't just inject either providers or instances anywhere you please. For example, while you can inject a provider to another provider's constructor, you should not be able to inject an instance there:

test/injector_spec.js

```
it('does not inject an instance to a provider constructor function', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider() {
    this.$get = function() { return 1; };
  });

  module.provider('b', function BProvider(a) {
    this.$get = function() { return a; };
  });

  expect(function() {
    createInjector(['myModule']);
  }).toThrow();

});
```


So, while `BProvider` may depend on `aProvider`, it may not depend on `a`.

Similarly, while you can inject instances to the `$get` method, you should not be able to inject providers there:

test/injector_spec.js

```
it('does not inject a provider to a $get function', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider() {
    this.$get = function() { return 1; };
  });
  module.provider('b', function BProvider() {
    this.$get = function(aProvider) { return aProvider.$get(); };
  });

  var injector = createInjector(['myModule']);

  expect(function() {
    injector.get('b');
  }).toThrow();
});
```

You should also not be able to inject providers to a function you call using `injector.invoke`:

test/injector_spec.js

```
it('does not inject a provider to invoke', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider() {
    this.$get = function() { return 1; }
  });

  var injector = createInjector(['myModule']);

  expect(function() {
    injector.invoke(function(aProvider) { });
  }).toThrow();
});
```

Nor should you even be able to call `injector.get` to obtain access to a provider. An exception should be thrown instead:

test/injector_spec.js


```
it('does not give access to providers through get', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider() {
    this.$get = function() { return 1; };
  });

  var injector = createInjector(['myModule']);
  expect(function() {
    injector.get('aProvider');
  }).toThrow();
});
```

What we have outlined with these tests is a clear separation between two types of injection: The injection that happens between provider constructors only deals with other providers. The injection that happens between `$get` methods and the external injector API only deals with *instances*. The instances may be created using providers but the providers are not exposed.

This separation can be implemented by actually having two separate injectors: One that deals exclusively with providers, and another that deals exclusively with instances. The latter will be the one exposed through the public API, and the former will only be used internally inside `createInjector`.

Let's reorganize the code to support the two injectors. We'll introduce the changes step by step first, and finally list the full, reorganized source code of `createInjector`.

To begin with, we'll have an internal function inside `createInjector` that we'll use to create our two internal injectors. This function will take two arguments: A cache to do dependency lookups from, and a factory function to fall back to when there's nothing in the cache:

src/injector.js

```
function createInternalInjector(cache, factoryFn) {

}
```

We need to move all the dependency lookup functions inside `createInternalInjector` because they need to be scoped to the `cache` and `factoryFn`. Firstly, `getService` will now live inside `createInternalInjector`. It'll do its lookups from the given cache object:

src/injector.js

```
function createInternalInjector(cache, factoryFn) {

  function getService(name) {
    if (cache.hasOwnProperty(name)) {
      if (cache[name] === INSTANTIATING) {
```



```
        throw new Error('Circular dependency found: ' +
          name + ' <- ' + path.join(' <- '));
      }
      return cache[name];
    } else {
      path.unshift(name);
      cache[name] = INSTANTIATING;
      try {
        return (cache[name] = factoryFn(name));
      } finally {
        path.shift();
        if (cache[name] === INSTANTIATING) {
          delete cache[name];
        }
      }
    }
  }

}
```

Note that we no longer do explicit provider invocation in the `else` branch. The work that happens when there's a cache miss is delegated to the `factoryFn` given to `createInternalInjector`. We will soon see what that work entails.

The `invoke` and `instantiate` functions also need to move inside `createInternalInjector`, because they depend on `getService`. The function implementations themselves do not require changes:

src/injector.js

```
function createInternalInjector(cache, factoryFn) {

  function getService(name) {
    if (cache.hasOwnProperty(name)) {
      if (cache[name] === INSTANTIATING) {
        throw new Error('Circular dependency found: ' +
          name + ' <- ' + path.join(' <- '));
      }
      return cache[name];
    } else {
      path.unshift(name);
      cache[name] = INSTANTIATING;
      try {
        return (cache[name] = factoryFn(name));
      } finally {
        path.shift();
        if (cache[name] === INSTANTIATING) {
          delete cache[name];
        }
      }
```



```
    }
  }

  function invoke(fn, self, locals) {
    var args = annotate(fn).map(function(token) {
      if (_.isString(token)) {
        return locals && locals.hasOwnProperty(token) ?
          locals[token] :
          getService(token);
      } else {
        throw 'Incorrect injection token! Expected a string, got '+token;
      }
    });
    if (_.isArray(fn)) {
      fn = _.last(fn);
    }
    return fn.apply(self, args);
  }

  function instantiate(Type, locals) {
    var instance = Object.create((_.isArray(Type) ? _.last(Type) : Type).prototype);
    invoke(Type, instance, locals);
    return instance;
  }

}
```

The last part of `createInternalInjector` is the creation of the injector object to return. This is the same object as the one we had as the return value of `createInjector` earlier:

src/injector.js

```
function createInternalInjector(cache, factoryFn) {

  // ...

  return {
    has: function(name) {
      return cache.hasOwnProperty(name) ||
          providerCache.hasOwnProperty(name + 'Provider');
    },
    get: getService,
    annotate: annotate,
    invoke: invoke,
    instantiate: instantiate
  };
}
```

The `get`, `invoke`, and `instantiate` methods refer to the functions we now have inside the `createInternalInjector` closure. The `annotate` method refers to the `annotate` function that hasn't changed. The `has` method checks for the existence of a dependency in the local cache as well as the provider cache.

Now that we have `createInternalInjector`, we can create our two injectors with it. The *provider injector* works with the provider cache. Its fallback function will throw an exception letting the user know the dependency they're looking for doesn't exist:

src/injector.js

```
function createInjector(modulesToLoad) {
  var providerCache = {};
  var providerInjector = createInternalInjector(providerCache, function() {
    throw 'Unknown provider: '+path.join(' <- ');
  });

  // ...
}
```

The instance injector correspondingly works with the instance cache. It falls back to a function that looks for a provider and uses it to construct the dependency. This is the logic we had in the `else` branch of `getService` earlier:

src/injector.js

```
function createInjector(modulesToLoad) {
  var providerCache = {};
  var providerInjector = createInternalInjector(providerCache, function() {
    throw 'Unknown provider: '+path.join(' <- ');
  });
  var instanceCache = {};
  var instanceInjector = createInternalInjector(instanceCache, function(name) {
    var provider = providerInjector.get(name + 'Provider');
    return instanceInjector.invoke(provider.$get, provider);
  });

  // ...

}
```

Note that we get the provider from the provider injector, but we invoke its `$get` method using the instance injector. That's how we make sure only instances get injected to `$get`.

As we now instantiate providers, we'll use the provider injector's `instantiate` method. That way it'll only have access to other providers, which was another one of our requirements:

src/injector.js


```
provider: function(key, provider) {
  if (_.isFunction(provider)) {
    provider = providerInjector.instantiate(provider);
  }
  providerCache[key + 'Provider'] = provider;
}
```

Constants are a special case in that we put a reference to them to both the provider and instance caches. Constants are available everywhere:

src/injector.js

```
constant: function(key, value) {
  if (key === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid constant name!';
  }
  providerCache[key] = value;
  instanceCache[key] = value;
},
```

Finally, the instance injector is what we actually return to the caller of `createInjector`:

src/injector.js

```
function createInjector(modulesToLoad, strictDi) {

  // ...

  return instanceInjector;
}
```

Here's the full, updated implementation of `createInjector`:

src/injector.js

```
function createInjector(modulesToLoad, strictDi) {
  var providerCache = {};
  var providerInjector = createInternalInjector(providerCache, function() {
    throw 'Unknown provider: '+path.join(' <- ');
  });
  var instanceCache = {};
  var instanceInjector = createInternalInjector(instanceCache, function(name) {
    var provider = providerInjector.get(name + 'Provider');
    return instanceInjector.invoke(provider.$get, provider);
  });
  var loadedModules = {};
  var path = [];
```



```
strictDi = (strictDi === true);

var $provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  }
};

function annotate(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];
    });
  }
}

function createInternalInjector(cache, factoryFn) {

  function getService(name) {
    if (cache.hasOwnProperty(name)) {
      if (cache[name] === INSTANTIATING) {
        throw new Error('Circular dependency found: ' +
          name + ' <- ' + path.join(' <- '));
      }
      return cache[name];
    } else {
      path.unshift(name);
      cache[name] = INSTANTIATING;
      try {
        return (cache[name] = factoryFn(name));
      } finally {
        path.shift();
```



```
        if (cache[name] === INSTANTIATING) {
          delete cache[name];
        }
      }
    }
  }

  function invoke(fn, self, locals) {
    var args = _.map(annotate(fn), function(token) {
      if (_.isString(token)) {
        return locals && locals.hasOwnProperty(token) ?
          locals[token] :
          getService(token);
      } else {
        throw 'Incorrect injection token! Expected a string, got '+token;
      }
    });
    if (_.isArray(fn)) {
      fn = _.last(fn);
    }
    return fn.apply(self, args);
  }

  function instantiate(Type, locals) {
    var UnwrappedType = _.isArray(Type) ? _.last(Type) : Type;
    var instance = Object.create(UnwrappedType.prototype);
    invoke(Type, instance, locals);
    return instance;
  }

  return {
    has: function(name) {
      return cache.hasOwnProperty(name) ||
          providerCache.hasOwnProperty(name + 'Provider');
    },
    get: getService,
    annotate: annotate,
    invoke: invoke,
    instantiate: instantiate
  };
}

_.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 instanceInjector;
}
```

The two injectors we now have implement two different *phases* of dependency injection.

1. *Provider* injection happens when providers are registered from a module's invoke queue. After that, there will be no more changes to `providerCache`.
2. At runtime there's *instance injection*, which happens whenever someone calls the injector's external API. The instance cache is populated as dependencies are instantiated, which happens in the fallback function of `instanceInjector`.

Unshifting Constants in The Invoke Queue

As we saw earlier, constructing instances lazily has the nice property that it frees the application developer from having to register things in the order of their dependencies. You can register A after B, even if A has a dependency on B.

With provider constructors there's no such freedom. Since provider constructors are invoked when the provider is registered (when the invoke queue is processed), you actually do need to register A before B if `BProvider` has a dependency to `AProvider`. Angular will make no effort to reorder the invoke queue to help you with this.

In the case of constants Angular does help you a bit though. Constants *will always be registered first*, so you can have a provider depending on a constant that's registered later:

test/injector_spec.js

```
it('registers constants first to make them available to providers', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider(b) {
    this.$get = function() { return b; };
  });
  module.constant('b', 42);

  var injector = createInjector(['myModule']);
  expect(injector.get('a')).toBe(42);
});
```

When constants are registered to a module, the module loader always adds them to the *front of the invoke queue*. That's how they get registered first. This is a safe reordering of the queue for Angular to make, since constants cannot depend on anything else.

How this works is the module loader's `invokeLater` function takes an optional argument that specifies which method of `Array` to use when adding the queue item. It defaults to `push`, which adds the item at the end of the queue:

src/loader.js


```
var invokeLater = function(method, arrayMethod) {
  return function() {
    invokeQueue[arrayMethod || 'push']([method, arguments]);
    return moduleInstance;
  };
};
```

Constant registration will override the method to `unshift`, whereas provider registration uses the default:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('constant', 'unshift'),
  provider: invokeLater('provider'),
  _invokeQueue: invokeQueue
};
```

Summary

This chapter has shown how providers are a crucial building block of dependency injection. They make it possible for dependencies to have other dependencies, and they allow dependencies to be instantiated by running some code lazily when the dependency is first needed.

Provider injection happens on a different plane from instance injection, and in this chapter we've seen how this is implemented using two separate injector objects inside what externally looks like a single one.

In this chapter you've learned:

- How provider objects and their `$get` methods work, and how their dependencies are injected using `injector.invoke`.
- How the module loader API allows chaining of registration method calls by returning the module instance.
- That `$get` methods are called lazily, only when someone needs the dependency.
- How all dependencies are singletons because `$get` methods are called at most once and their results are then cached.
- How circular dependencies are handled and circular dependency error messages constructed.
- How providers can be registered as plain objects or constructor functions.
- How provider constructor functions are instantiated using dependency injection.

- How the two phases of dependency injection are separated inside the injector by having two dependency caches and two internal injector objects.
- How the instance injector falls back to the provider injector on cache misses.
- How constants are always registered first to loosen up the requirements in registration order.

The next chapter will finalize our implementation of dependency injection, by adding the high-level facilities most application developers use most often, such as factories and services. We will see how their implementations build on the foundation we've laid out.

Chapter 12

High-Level Dependency Injection Features

Our implementation of the Angular dependency injector is already very capable. In fact, pretty much everything Angular can do, we can now also do. The problem is that the API is still a bit austere. Providers, while powerful, are not exactly streamlined for the most common use cases application developers have. There's also not much in the way of runtime configurability in our injector.

In this chapter we can reap the benefits of the work we've done with injectors and providers, by adding the layer of features most useful for application developers. It will involve some higher-level abstractions for dependency creation, as well as a few additional ways to configure and use the injector. By the end of the chapter we'll have the full capabilities of the Angular.js DI framework.

We'll also run into a scenario where we need a data structure that's lacking in JavaScript: The hash map. Angular ships with a hash map implementation, and we'll build it in this chapter.

Injecting The $injectors

After you create an injector object, you can use its public API to introspect its contents and inject dependencies to functions and constructors. For an application developer, perhaps the most interesting method of the injector is `get`, since with it you can obtain a dependency dynamically, without having to even know its name until at runtime. This can be very useful when you need it.

It would make sense, then, to make it easy for an application developer to get access to the injector itself. This is in fact what Angular does. The injector is available as a dependency called `$injector`:

test/injector_spec.js

```
it('allows injecting the instance injector to $get', function() {
  var module = angular.module('myModule', []);

  module.constant('a', 42);
  module.provider('b', function BProvider() {
    this.$get = function($injector) {
      return $injector.get('a');
    };
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('b')).toBe(42);
});
```

The object you get when you inject `$injector` is simply the instance injector object we already have inside `createInjector`. We can make it available by adding it to the instance cache:

src/injector.js

```
var instanceInjector = instanceCache.$injector =
    createInternalInjector(instanceCache, function(name) {
  var provider = providerInjector.get(name + 'Provider');
  return instanceInjector.invoke(provider.$get, provider);
});
```

Similarly, you can inject `$injector` to a provider constructor. As you may recall from the last chapter, provider constructors only have other providers and constants available for injection. This rule is also enforced by `$injector` injection: What you get in this case is the *provider injector*:

test/injector_spec.js

```
it('allows injecting the provider injector to provider', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider() {
    this.value = 42;
    this.$get = function() { return this.value; };
  });
  module.provider('b', function BProvider($injector) {
    var aProvider = $injector.get('aProvider');
    this.$get = function() {
      return aProvider.value;
    };
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('b')).toBe(42);
});
```


The provider injector is made available in exactly the same way as the instance injector - by putting it in the cache:

src/injector.js

```
var providerInjector = providerCache.$injector =
    createInternalInjector(providerCache, function() {
  throw 'Unknown provider: '+path.join(' <- ');
});
```

So, when you ask for `$injector`, you may get the instance injector or the provider injector, depending on where you are injecting it.

Injecting $provide

The injector object lets you introspect and access dependencies that have been configured, but it does not let you change anything. It's a read-only API. If you want to *add* some dependencies and, for one reason or another, can't just add them to a module, you can use an object called `$provide`.

By injecting `$provide` you gain direct access to the methods we've been calling through the module invoke queue. For example, a provider constructor can use `$provide` to smuggle a constant into the injector:

test/injector_spec.js

```
it('allows injecting the $provide service to providers', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider($provide) {
    $provide.constant('b', 2);
    this.$get = function(b) { return 1 + b; };
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(3);
});
```

Crucially though, `$provide` is only available through the provider injector. At runtime, when what you have is the instance injector, you can no longer inject `$provide` and thus you can no longer add dependencies:

test/injector_spec.js


```
it('does not allow injecting the $provide service to $get', function() {
  var module = angular.module('myModule', []);

  module.provider('a', function AProvider() {
    this.$get = function($provide) { };
  });

  var injector = createInjector(['myModule']);

  expect(function() {
    injector.get('a');
  }).toThrow();
});
```

The `$provide` object you inject is in fact the `$provide` object we already have - the one with the `constant` and `provider` methods. That's why we gave it the peculiar name `$provide`. Now we just need to put it in the provider cache, like we did with the provider injector itself earlier:

src/injector.js

```
providerCache.$provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  }
};
```

Since we just removed the local variable `$provide`, we need to update the way we access `$provide` during the processing of the invoke queues:

src/injector.js

```
_.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];
      providerCache.$provide[method].apply(providerCache.$provide, args);
    });
  }
});
```

Through `$injector` and `$provide`, the injector gives you direct access to most of its internal machinery. Many applications will never end up needing them since all they need is modules and dependency injection. But they can come in very handy when you do need that bit of extra configurability or introspection.

Config Blocks

One of the big reasons for using a provider instead of higher-level factories or services, which we'll introduce later in this chapter, is configurability. Since you can access a provider *before* its `$get` method is called, you can affect how dependency instantiation happens. An example of this is the `$route` facility of Angular's ngRoute module. The `$route` service is responsible for routing URLs to controllers in your application. The way you configure it is by using the *provider* of `$route`. That is, `$routeProvider`:

```
$routeProvider.when('/someUrl', {
  templateUrl: '/my/view.html',
  controller: 'MyController'
});
```

The only problem is that to get `$routeProvider` you need provider injection, and the only place we currently have it is in the constructors of other providers. Defining a provider constructor just to be able to configure some other provider is awkward. What we really need is a way to execute arbitrary "configuration functions" at module loading time, and a way to inject providers to those functions. For this purpose, Angular has *config blocks*.

You can define a config block by calling the `config` function of a module. You give it a function, and that function will be executed when the injector is created:

test/injector_spec.js

```
it('runs config blocks when the injector is created', function() {
  var module = angular.module('myModule', []);

  var hasRun = false;
  module.config(function() {
    hasRun = true;
  });

  createInjector(['myModule']);

  expect(hasRun).toBe(true);
});
```


The function you provide may be injected (using any of the three dependency injection mechanisms from Chapter 9). You can, for instance, inject `$provide`:

test/injector_spec.js

```
it('injects config blocks with provider injector', function() {
  var module = angular.module('myModule', []);

  module.config(function($provide) {
    $provide.constant('a', 42);
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(42);
});
```

So a config block is an arbitrary function that has its dependencies injected from the provider cache. We can meet these requirements by calling the config block using `injector.invoke()` from the provider injector.

First we need the API for registering a config block on a module. We need to queue up a task that will cause the provider injector's `invoke` method to be called. The first problem is that our invoke queue code currently only supports calling methods on `$provide`, not `$injector`. We need to extend the queue items so that both are supported. Queue items should actually be three-item arrays: 1) The object whose method to call, 2) The name of the method to call, and 3) The method arguments:

src/loader.js

```
var invokeLater = function(service, method, arrayMethod) {
  return function() {
    var item = [service, method, arguments];
    invokeQueue[arrayMethod || 'push'](item);
    return moduleInstance;
  };
};
```

Then we need to update our existing queueing methods to specify the `$provide` object as the first item of the array:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  _invokeQueue: invokeQueue
};
```


The second change we need to make to `invokeLater` is related to the actual queue used: Our current implementation puts everything in one queue, and the invocations will get executed in the same order as they were registered. If we now take config blocks into account, this is not optimal. We would prefer for all *registration invocations* to run before any *config blocks*. That way all the providers of a module are available to a config block regardless of the order in which they were added:

test/injector_spec.js

```
it('allows registering config blocks before providers', function() {
  var module = angular.module('myModule', []);

  module.config(function(aProvider) { });
  module.provider('a', function() {
    this.$get = _.constant(42);
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(42);
});
```

To introduce this order-independence, we need to introduce a *second* queue just for the config blocks. We'll then modify the `invokeLater` function to take an optional fourth argument for specifying which queue to use. It will default to the invoke queue:

src/loader.js

```
var createModule = function(name, requires, modules, configFn) {
  if (name === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid module name';
  }

  var invokeQueue = [];
  var configBlocks = [];

  var invokeLater = function(service, method, arrayMethod, queue) {
    return function() {
      queue = queue || invokeQueue;
      queue[arrayMethod || 'push']([service, method, arguments]);
      return moduleInstance;
    };
  };

  // ...
}
```


Now we can add the new queueing method, which will queue up a call to `$injector.invoke`. We will also attach the config block queue to the module instance so that the injector can drain it:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks
};
```

We now need to iterate two queues in the injector. Let's first extract the iteration code to a function and call it for both queues:

src/injector.js

```
function runInvokeQueue(queue) {
  _.forEach(queue, function(invokeArgs) {
    var method = invokeArgs[0];
    var args = invokeArgs[1];
    providerCache.$provide[method].apply(providerCache.$provide, args);
  });
}

_.forEach(modulesToLoad, function loadModule(moduleName) {
  if (!loadedModules.hasOwnProperty(moduleName)) {
    loadedModules[moduleName] = true;
    var module = angular.module(moduleName);
    _.forEach(module.requires, loadModule);
    runInvokeQueue(module._invokeQueue);
    runInvokeQueue(module._configBlocks);
  }
});
```

As we iterate the queues, we also need to dynamically look up the object to call rather than just assume it's going to be `$provide`. The `invokeArgs` array now has three elements:

src/injector.js

```
function runInvokeQueue(queue) {
  _.forEach(queue, function(invokeArgs) {
    var service = providerInjector.get(invokeArgs[0]);
    var method = invokeArgs[1];
    var args = invokeArgs[2];
    service[method].apply(service, args);
  });
}
```


So, you can register a config block by calling `config()` on a module instance. Another way to register a config block is to supply one as the third argument when first creating a module instance using `angular.module`:

test/injector_spec.js

```
it('runs a config block added during module registration', function() {
  var module = angular.module('myModule', [], function($provide) {
    $provide.constant('a', 42);
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(42);
});
```

If a function like this is given, we need to first pass it from `angular.module` to the internal `createModule` function in the loader:

src/loader.js

```
ensure(angular, 'module', function() {
  var modules = {};
  return function(name, requires, configFn) {
    if (requires) {
      return createModule(name, requires, modules, configFn);
    } else {
      return getModule(name, modules);
    }
  };
});
```

In `createModule` we can then just call `config` on the new module instance:

src/loader.js

```
var createModule = function(name, requires, modules, configFn) {
  if (name === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid module name';
  }
  var invokeQueue = [];
  var configBlocks = [];

  var invokeLater = function(service, method, arrayMethod, queue) {
    return function() {
      queue = queue || invokeQueue;
```



```
      queue[arrayMethod || 'push']([service, method, arguments]);
      return moduleInstance;
    };
  };

  var moduleInstance = {
    name: name,
    requires: requires,
    constant: invokeLater('$provide', 'constant', 'unshift'),
    provider: invokeLater('$provide', 'provider'),
    config: invokeLater('$injector', 'invoke', 'push', configBlocks),
    _invokeQueue: invokeQueue,
    _configBlocks: configBlocks
  };

  if (configFn) {
    moduleInstance.config(configFn);
  }

  modules[name] = moduleInstance;
  return moduleInstance;
};
```

Run Blocks

Run blocks are a close cousin of config blocks. Just like config blocks, they are arbitrary functions that are invoked at injector construction time:

test/injector_spec.js

```
it('runs run blocks when the injector is created', function() {
  var module = angular.module('myModule', []);

  var hasRun = false;
  module.run(function() {
    hasRun = true;
  });

  createInjector(['myModule']);

  expect(hasRun).toBe(true);
});
```

The main difference between config blocks and run blocks is that run blocks are injected from the *instance cache*:

test/injector_spec.js


```
it('injects run blocks with the instance injector', function() {
  var module = angular.module('myModule', []);

  module.provider('a', {$get: _.constant(42)});

  var gotA;
  module.run(function(a) {
    gotA = a;
  });

  createInjector(['myModule']);

  expect(gotA).toBe(42);
});
```

The purpose of run blocks is not to configure providers - you can't even inject them here - but to just run some arbitrary code you want to hook on to the Angular startup process. To implement run blocks, all we really need to do is collect them in the module loader, and then invoke them after the injector has been created.

Unlike config blocks, run blocks aren't put into a module's invoke queue. They're stored in their own collection on the module instance:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks,
  _runBlocks: []
};
```

Let's try the simplest possible implementation for running the run blocks. We can just iterate over them after iterating the invoke queue, and call each one using the instance injector:

src/injector.js

```
_.forEach(modulesToLoad, function loadModule(moduleName) {
  if (!loadedModules.hasOwnProperty(moduleName)) {
    loadedModules[moduleName] = true;
    var module = angular.module(moduleName);
    _.forEach(module.requires, loadModule);
    runInvokeQueue(module._invokeQueue);
    runInvokeQueue(module._configBlocks);
    _.forEach(module._runBlocks, function(runBlock) {
      instanceInjector.invoke(runBlock);
    });
  }
});
```

This satisfies the existing test suite, but there's a problem. Run blocks should be run once module loading is complete. When your injector loads several modules, any run blocks from any of them should be deferred to until all modules are loaded. This is not true in our first implementation, as we can see when trying to inject dependencies from other modules:

test/injector_spec.js

```
it('configures all modules before running any run blocks', function() {
  var module1 = angular.module('myModule', []);
  module1.provider('a', {$get: _.constant(1)});
  var result;
  module1.run(function(a, b) {
    result = a + b;
  });

  var module2 = angular.module('myOtherModule', []);
  module2.provider('b', {$get: _.constant(2)});

  createInjector(['myModule', 'myOtherModule']);

  expect(result).toBe(3);
});
```

The trick is to collect all run blocks to an array, and only invoke them after the outer module loader loop has finished and everything has been loaded:

src/injector.js

```
var runBlocks = [];
_.forEach(modulesToLoad, function loadModule(moduleName) {
  if (!loadedModules.hasOwnProperty(moduleName)) {
    loadedModules[moduleName] = true;
    var module = angular.module(moduleName);
    _.forEach(module.requires, loadModule);
    runInvokeQueue(module._invokeQueue);
```



```
    runInvokeQueue(module._configBlocks);
    runBlocks = runBlocks.concat(module._runBlocks);
  }
});
_.forEach(runBlocks, function(runBlock) {
  instanceInjector.invoke(runBlock);
});
```

To sum up, config blocks are executed *during* module loading and run blocks are executed *immediately after* it.

Function Modules

As we have seen, a module is an object into which you can register application components. Internally it holds a queue of these component registrations that will be executed when the module is loaded.

There is also an alternative way you can define a module: A module can be just a function, which will be injected from the provider injector when loaded.

Here we define 'myModule'' as a normal module object. It has one dependency, which is to a function module:

test/injector_spec.js

```
it('runs a function module dependency as a config block', function() {
  var functionModule = function($provide) {
    $provide.constant('a', 42);
  };
  angular.module('myModule', [functionModule]);

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(42);
});
```

You can also use array-style dependency annotation with function modules:

test/injector_spec.js

```
it('runs a function module with array injection as a config block', function() {
  var functionModule = ['$provide', function($provide) {
    $provide.constant('a', 42);
  }];
  angular.module('myModule', [functionModule]);

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(42);
});
```


Function modules are essentially exactly the same thing as config blocks: Functions that are injected with providers. The only difference is where you define them: Config blocks are attached *to a module*, whereas function modules are *dependencies of other modules*.

As we do module loading, we can now no longer assume that a module is a string to be looked up from `angular.module`. It can also be a function or an array, in which case we should "load" it by invoking it with provider injection:

src/injector.js

```
_.forEach(modulesToLoad, function loadModule(module) {
  if (_.isString(module)) {
    if (!loadedModules.hasOwnProperty(module)) {
      loadedModules[module] = true;
      module = angular.module(module);
      _.forEach(module.requires, loadModule);
      runInvokeQueue(module._invokeQueue);
      runInvokeQueue(module._configBlocks);
      runBlocks = runBlocks.concat(module._runBlocks);
    }
  } else if (_.isFunction(module) || _.isArray(module)) {
    providerInjector.invoke(module);
  }
});
```

When you have a function module, you can also return a value from it. That value will be executed as a run block. This little detail allows for a very concise way to define ad-hoc modules and corresponding run blocks, which may be particularly useful in unit tests.

test/injector_spec.js

```
it('supports returning a run block from a function module', function() {
  var result;
  var functionModule = function($provide) {
    $provide.constant('a', 42);
    return function(a) {
      result = a;
    };
  };
  angular.module('myModule', [functionModule]);

  createInjector(['myModule']);

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


When a function module is executed, we need to add its return value to the collection of run blocks. Because returning a run block from a function module is still optional though, we need to be prepared for it to be `undefined` as well. We can defensively remove falsy run blocks before iterating them, using the LoDash `_.compact` function:

src/injector.js

```
var runBlocks = [];
_.forEach(modulesToLoad, function loadModule(module) {
  if (_.isString(module)) {
    if (!loadedModules.hasOwnProperty(module)) {
      loadedModules[module] = true;
      module = angular.module(module);
      _.forEach(module.requires, loadModule);
      runInvokeQueue(module._invokeQueue);
      runInvokeQueue(module._configBlocks);
      runBlocks = runBlocks.concat(module._runBlocks);
    }
  } else if (_.isFunction(module) || _.isArray(module)) {
    runBlocks.push(providerInjector.invoke(module));
  }
});
_.forEach(_.compact(runBlocks), function(runBlock) {
  instanceInjector.invoke(runBlock);
});
```

We still have a slight problem with loading function modules, though. As discussed in Chapter 9, each module should be loaded only once even if required multiple times. To that effect we added the `loadedModules` object and stored the names of loaded modules in it.

We can't use this object with function modules, and indeed are not currently checking repeated loads of them in any way. A function module required twice is executed twice:

test/injector_spec.js

```
it('only loads function modules once', function() {
  var loadedTimes = 0;
  var functionModule = function() {
    loadedTimes++;
  };

  angular.module('myModule', [functionModule, functionModule]);
  createInjector(['myModule']);

  expect(loadedTimes).toBe(1);
});
```


We can't put function modules to the `loadedModules` object, since JavaScript object keys can only be strings, not functions. What we need is a general purpose key-value data structure: A hash map.

JavaScript does not have such a data structure (until ECMAScript 6), so Angular ships with one, which it uses for the `loadedModules` variable. As we will see later, it is also used by certain built-in directives. While there are existing key-value data structure libraries for JavaScript that we could leverage, the one Angular bundles is peculiar enough that it makes sense for us to study how it works. So, we will now take a slight detour from dependency injection and build it.

Hash Keys And Hash Maps

The first function we'll implement here is called `hashKey`. This function takes any JavaScript value and returns a string "hash key" for it. The purpose is similar to Java's Object.hashCode() and Ruby's Object#hash, for example: A value's hash key uniquely identifies it, and no other value of any type should have the same hash key. We'll use the `hashKey` function in our hash map implementation in a moment, and it will also be used in a couple of other situations in the book.

The hash key and hash map implementations will go in a new file called `src/apis.js`. The corresponding test file will be `test/apis_spec.js`. Let's start specifying `hashKey` there.

Generally, a value's hash key has two parts to it: The first part designates the type of the value and the second part designates the value's string representation. The two parts are separated with a colon. For the `undefined` value, both of these parts will be `'undefined'`:

test/apis_spec.js

```
/* global hashKey: false, HashMap: false */

describe('apis', function() {
  'use strict';

  describe('hashKey', function() {

    it('is undefined:undefined for undefined', function() {
      expect(hashKey(undefined)).toEqual('undefined:undefined');
    });

  });

});
```

For `null`, the type is `'object'` and the string representation is `'null'`:

test/apis_spec.js

```
it('is object:null for null', function() {
  expect(hashKey(null)).toEqual('object:null');
});
```


For boolean values, the type is `'boolean'` and the string representation is `'true'` or `'false'`:

test/apis_spec.js

```
it('is boolean:true for true', function() {
  expect(hashKey(true)).toEqual('boolean:true');
});

it('is boolean:false for false', function() {
  expect(hashKey(false)).toEqual('boolean:false');
});
```

For numbers, the type is `'number'` and the string representation is just the number as a string:

test/apis_spec.js

```
it('is number:42 for 42', function() {
  expect(hashKey(42)).toEqual('number:42');
});
```

For strings, the type is `'string'` and the string representation is - unsurprisingly - the string. Here we see why we need to encode the type to the hash key. Otherwise the hash keys for the number `42` and the string `'42'` would be the same:

test/apis_spec.js

```
it('is string:42 for "42"', function() {
  expect(hashKey('42')).toEqual('string:42');
});
```

All of the tests so far can be satisfied with a simple implementation of `hashKey` that takes the type of the value with the `typeof` operator for the first part, and a coerced string representation of the value for the second part:

src/apis.js

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

function hashKey(value) {
  var type = typeof value;
  return type + ':' + value;
}
```


When we start dealing with functions and objects (and arrays), things get a bit more interesting. Generally, the hash key of an object consists of the string `'object'` as the first part, and a unique identifier as the second part:

test/apis_spec.js

```
it('is object:[unique id] for objects', function() {
  expect(hashKey({})).toMatch(/^object:\S+$/);
});
```

We should expect the hash key to be stable, so that it is the same value when generated for the same object twice:

test/apis_spec.js

```
it('is the same key when asked for the same object many times', function() {
  var obj = {};
  expect(hashKey(obj)).toEqual(hashKey(obj));
});
```

Interestingly though, the hash key of an object is stable even if the same object *mutates* between the two `hashKey` calls:

test/apis_spec.js

```
it('does not change when object value changes', function() {
  var obj = {a: 42};
  var hash1 = hashKey(obj);
  obj.a = 43;
  var hash2 = hashKey(obj);
  expect(hash1).toEqual(hash2);
});
```

This means that the `hashKey` function *does not* use value semantics for generating the hash key of an object, but it is the *identity* of the object that matters. This also means that two *different* objects whose values happen to be the same do not have the same hash code:

test/apis_spec.js

```
it('is not the same for different objects even with the same value', function() {
  var obj1 = {a: 42};
  var obj2 = {a: 42};
  expect(hashKey(obj1)).not.toEqual(hashKey(obj2));
});
```


For functions, which is what we'll be putting in hash maps in this chapter, the same rules apply. The hash key of a function is the string `'function'` followed by a numeric id:

test/apis_spec.js

```
it('is function:[unique id] for functions', function() {
  var fn = function(a) { return a; };
  expect(hashKey(fn)).toMatch(/^function:\S+$/);
});
```

The hash key stays the same when asked for the same function multiple times:

test/apis_spec.js

```
it('is the same key when asked for the same function many times', function() {
  var fn = function() { };
  expect(hashKey(fn)).toEqual(hashKey(fn));
});
```

The hashkey is *not* the same for two functions even if they are identical:

test/apis_spec.js

```
it('is not the same for different identical functions', function() {
  var fn1 = function() { return 42; };
  var fn2 = function() { return 42; };
  expect(hashKey(fn1)).not.toEqual(hashKey(fn2));
});
```

So the `hashKey` function is not strictly a hash key function at all, in the same sense as the Java and Ruby methods are. In the case of functions and compound data structures it is just a unique identifier that is based on object identity rather than value.

The way this works is that, given an object or a function, `hashKey` actually populates a special key `$$hashKey` on it, which holds the unique id of the object (the part after the colon in the return value of `hashKey`). You have probably seen these keys appear in your objects if you've used them with certain Angular directives.

test/apis_spec.js

```
it('stores the hash key in the $$hashKey attribute', function() {
  var obj = {a: 42};
  var hash = hashKey(obj);
  expect(obj.$$hashKey).toEqual(hash.match(/^object:(\S+)$/)[1]);
});
```


If there already is a `$$hashKey` in the given object, its value is used instead of generating one. That's how the value is kept stable:

test/apis_spec.js

```
it('uses preassigned $$hashKey', function() {
  expect(hashKey({$$hashKey: 42})).toEqual('object:42');
});
```

We need to branch the `hashKey` function to handle the object and non-object cases differently. In the object (or function) case we'll look at the `$$hashKey` attribute, and if needed, populate it with a unique identifier. We can use the unique id generator function that comes with LoDash:

src/apis.js

```
function hashKey(value) {
  var type = typeof value;
  var uid;
  if (type === 'function' ||
      (type === 'object' && value !== null)) {
    uid = value.$$hashKey;
    if (uid === undefined) {
      uid = value.$$hashKey = _.uniqueId();
    }
  } else {
    uid = value;
  }
  return type + ':' + uid;
}
```

Finally, if you want to plug in your own behavior for generating hash keys for objects, you can preassign a *function* as the value of the `$$hashKey` attribute. If Angular sees a function there, it will call it as a method to obtain the concrete hash key:

test/apis_spec.js

```
it('supports a function $$hashKey', function() {
  expect(hashKey({$$hashKey: _.constant(42)})).toEqual('object:42');
});

it('calls the function $$hashKey as a method with the correct this', function() {
  expect(hashKey({
    myKey: 42,
    $$hashKey: function() {
      return this.myKey;
    }
  })).toEqual('object:42');
});
```


The implementation needs to do a `typeof` check for the `$$hashKey` and call it as a method if it is a function:

src/apis.js

```
function hashKey(value) {
  var type = typeof value;
  var uid;
  if (type === 'function' ||
       (type === 'object' && value !== null)) {
    uid = value.$$hashKey;
    if (typeof uid === 'function') {
      uid = value.$$hashKey();
    } else if (uid === undefined) {
      uid = value.$$hashKey = _.uniqueId();
    }
  } else {
    uid = value;
  }
  return type + ':' + uid;
}
```

With `hashKey` now taken care of, we can implement the *hash map*. An Angular hash map is an object created using the `HashMap` constructor. It implements an associative data structure whose keys can be of any type. Like its distant cousin, the Java HashMap, it supports the `put` and `get` methods:

test/apis_spec.js

```
describe('HashMap', function() {

  it('supports put and get of primitives', function() {
    var map = new HashMap();
    map.put(42, 'fourty two');
    expect(map.get(42)).toEqual('fourty two');
  });

});
```

We can expect the key semantics of `HashMap` to be what we just defined for `hashKey`. That is, for objects it is the identity that matters, not the value (making the implementation similar to ES6 Maps, and also making the name `HashMap` a bit of a misnomer):

test/apis_spec.js


```
it('supports put and get of objects with hashKey semantics', function() {
  var map = new HashMap();
  var obj = {};
  map.put(obj, 'my value');
  expect(map.get(obj)).toEqual('my value');
  expect(map.get({})).toBeUndefined();
});
```

The implementation of `put` and `get` is quite simple now that we have `hashKey`. Since the values returned by `hashKey` are strings, the actual storage used by `HashMap` can be a regular JavaScript object. Instead of making a separate storage object though, we'll use the `HashMap` instance itself:

src/apis.js

```
function HashMap() {
}

HashMap.prototype = {
  put: function(key, value) {
    this[hashKey(key)] = value;
  },
  get: function(key) {
    return this[hashKey(key)];
  }
};
```

As a side effect of this implementation, notice that you can also access a hash map's contents directly with attribute accessors (`map['number:42']`), though it is probably not a good idea to rely on it in application code.

The third and final method supported by `HashMap` is `remove`. It is used to remove a key-value pair from the map:

test/apis_spec.js

```
it('supports remove', function() {
  var map = new HashMap();
  map.put(42, 'fourty two');
  map.remove(42);
  expect(map.get(42)).toBeUndefined();
});
```

The `remove` method also returns the value of the key that was just removed, as a convenience:

test/apis_spec.js

```
it('returns value from remove', function() {
  var map = new HashMap();
  map.put(42, 'fourty two');
  expect(map.remove(42)).toEqual('fourty two');
});
```

In the implementation of `remove` we obtain the hash key from the given key and then just use the `delete` operator to remove it from the underlying storage:

src/apis.js

```
HashMap.prototype = {
  put: function(key, value) {
    this[hashKey(key)] = value;
  },
  get: function(key) {
    return this[hashKey(key)];
  },
  remove: function(key) {
    key = hashKey(key);
    var value = this[key];
    delete this[key];
    return value;
  }
};
```

Function Modules Redux

Armed with our new implementation of `HashMap` we can now fix the failing test in `injector_spec.js`, and make sure function modules are also loaded at most once.

First we need to add `HashMap` to the allowed globals in the JSHint directive on the top of the file:

src/injector.js

```
/* global angular: false, HashMap: false */
```

Next, we'll convert the `loadedModules` variable from an object literal to a `HashMap` instance:

src/injector.js

```
var loadedModules = new HashMap();
```


And finally, in the module loading loop we can generalize the `loadedModules` check to account not only for string modules but all kinds of modules:

src/injector.js

```
_.forEach(modulesToLoad, function loadModule(module) {
  if (!loadedModules.get(module)) {
    loadedModules.put(module, true);
    if (_.isString(module)) {
      module = angular.module(module);
      _.forEach(module.requires, loadModule);
      runInvokeQueue(module._invokeQueue);
      runInvokeQueue(module._configBlocks);
      runBlocks = runBlocks.concat(module._runBlocks);
    } else if (_.isFunction(module) || _.isArray(module)) {
      runBlocks.push(providerInjector.invoke(module));
    }
  }
});
```

Factories

We are now ready to add the high-level component registration functions that application developers actually use the most: Those for factories, values, and services. As we'll see, there's actually not that much work involved because of the foundation we have built in the past few chapters.

First, let's look at factories. A factory is a function that produces a dependency. Here we register a factory function that returns `42`. When we ask for the corresponding dependency, we'll get `42`:

test/injector_spec.js

```
it('allows registering a factory', function() {
  var module = angular.module('myModule', []);

  module.factory('a', function() { return 42; });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(42);
});
```

A factory function can be injected with dependencies, which is the main added value it brings over constants. Specifically, it is injected with *instance dependencies*, as opposed to provider dependencies. Here, the factory for `b` is dependent on `a`:

test/injector_spec.js


```
it('injects a factory function with instances', function() {
  var module = angular.module('myModule', []);

  module.factory('a', function() { return 1; });
  module.factory('b', function(a) { return a + 2; });

  var injector = createInjector(['myModule']);

  expect(injector.get('b')).toBe(3);
});
```

Just like dependencies made by providers, dependencies made by factories are singletons. We expect a factory function to be invoked at most once, so that each time we need the dependency we get the same exact value:

test/injector_spec.js

```
it('only calls a factory function once', function() {
  var module = angular.module('myModule', []);

  module.factory('a', function() { return {}; });

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(injector.get('a'));
});
```

To implement factories, we first need to queue the registration up in the module loader, by adding a new queueing function:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  factory: invokeLater('$provide', 'factory'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks
  _runBlocks: []
};
```

Then we need the corresponding method in the `$provide` object of the injector. And what should it actually do? Consider that a factory's functionality is pretty much the same as the `$get` method of a provider: It returns the dependency, is injected with instances, and is called at most once.

In fact, the `$get` method of a provider is *exactly* what a factory is implemented as:

src/injector.js

```
providerCache.$provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  },
  factory: function(key, factoryFn) {
    this.provider(key, {$get: factoryFn});
  }
};
```

A factory is really a provider. When you register a factory, what happens is a provider object is created on the fly, and the `$get` method of that provider will be the original factory function you registered. All the work we did to implement providers comes into play here, and there's not much more that we need to do to implement factories.

What factories don't give you is the full configurability of providers. If you register the factory `a`, you *can* then get access to `aProvider` through provider injection. But that provider will be the one created on the fly, and will not have any configuration methods attached to it. So while `aProvider` will exist, there won't be much point accessing it directly.

There's one additional error check we should do for factories, which is to make sure they actually return something. This is to prevent bugs in application code that would otherwise be difficult to pin down.

test/injector_spec.js

```
it('forces a factory to return a value', function() {
  var module = angular.module('myModule', []);

  module.factory('a', function() {  });
  module.factory('b', function() { return null; });
```



```
  var injector = createInjector(['myModule']);

  expect(function() {
    injector.get('a');
  }).toThrow();
  expect(injector.get('b')).toBeNull();
});
```

Here we see that an `undefined` return value is not accepted, but `null` still is.

What we can do is wrap the given factory function in another function and do the check there:

src/injector.js

```
function enforceReturnValue(factoryFn) {
  return function() {
    var value = instanceInjector.invoke(factoryFn);
    if (_.isUndefined(value)) {
      throw 'factory must return a value';
    }
    return value;
  };
}

providerCache.$provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  },
  factory: function(key, factoryFn) {
    this.provider(key, {$get: enforceReturnValue(factoryFn)});
  }
};
```

What we now have as the body of the `$get` method is a function with zero arguments. That function invokes the original factory using instance dependency injection and returns its return value. If the return value is `undefined`, we let the user know by throwing an exception.

Values

Values are a bit like constants. You register one to a module by just providing the value without any generator functions involved:

test/injector_spec.js

```
it('allows registering a value', function() {
  var module = angular.module('myModule', []);

  module.value('a', 42);

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBe(42);
});
```

The difference between a value and a constant is that values are *not* available to providers or config blocks. They are strictly for instances only:

test/injector_spec.js

```
it('does not make values available to config blocks', function() {
  var module = angular.module('myModule', []);

  module.value('a', 42);
  module.config(function(a) {
  });

  expect(function() {
    createInjector(['myModule']);
  }).toThrow();

});
```

Again, we first need the queueing function for values in the module loader:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  factory: invokeLater('$provide', 'factory'),
  value: invokeLater('$provide', 'value'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
```



```
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks,
  _runBlocks: []
};
```

The actual implementation is simple: We create a factory function that has no dependencies and always returns the given value.

src/injector.js

```
providerCache.$provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  },
  factory: function(key, factoryFn) {
    this.provider(key, {$get: enforceReturnValue(factoryFn)});
  },
  value: function(key, value) {
    this.factory(key, _.constant(value));
  }
};
```

Values may also be `undefined`, but this is something our current implementation does not let us do:

test/injector_spec.js

```
it('allows an undefined value', function() {
  var module = angular.module('myModule', []);

  module.value('a', undefined);

  var injector = createInjector(['myModule']);

  expect(injector.get('a')).toBeUndefined();
});
```


So the return value enforcement of factories should not always be applied. Let's add a third optional argument to `factory` for controlling whether to do it or not:

src/injector.js

```
providerCache.$provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  },
  factory: function(key, factoryFn, enforce) {
    this.provider(key, {
      $get: enforce === false ? factoryFn : enforceReturnValue(factoryFn)
    });
  },
  value: function(key, value) {
    this.factory(key, _.constant(value), false);
  }
};
```

If the third argument to `factory` is explicitly set to `false`, it will not enforce a return value for the factory. We use this option in `value`.
So basically, a value is implemented with a factory, which in turn is implemented with a provider. By the looks of it, we could just as well have simply stored the value in the instance cache, as we do for constants. But doing it this way allows for *decoration* - a feature we'll look at in a few moments.

Services

Whereas a factory is a plain old function, a *service* is a *constructor function*. When you register a service, the function you give is treated as a constructor and an instance of it will be created:

test/injector_spec.js


```
it('allows registering a service', function() {
  var module = angular.module('myModule', []);

  module.service('aService', function MyService() {
    this.getValue = function() { return 42; };
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('aService').getValue()).toBe(42);
});
```

The constructor function can also be dependency-injected with instances:

test/injector_spec.js

```
it('injects service constructors with instances', function() {
  var module = angular.module('myModule', []);

  module.value('theValue', 42);
  module.service('aService', function MyService(theValue) {
    this.getValue = function() { return theValue; };
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('aService').getValue()).toBe(42);
});
```

And, just like everything else, a service is a singleton, meaning that the constructor will be used at most once and the resulting object is cached for subsequent uses:

test/injector_spec.js

```
it('only instantiates services once', function() {
  var module = angular.module('myModule', []);

  module.service('aService', function MyService() {
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('aService')).toBe(injector.get('aService'));
});
```

For implementing services, the same pattern repeats yet again: We first add the queueing method to the module loader.

src/loader.js


```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  factory: invokeLater('$provide', 'factory'),
  value: invokeLater('$provide', 'value'),
  service: invokeLater('$provide', 'service'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks,
  _runBlocks: []
};
```

Then we implement the method in `$provide`. It takes the dependency key and the constructor function. What we'll do is create a factory function on the fly, in which we'll do our work:

src/injector.js

```
providerCache.$provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  },
  factory: function(key, factoryFn, enforce) {
    this.provider(key, {
      $get: enforce === false ? factoryFn : enforceReturnValue(factoryFn)
    });
  },
  value: function(key, value) {
    this.factory(key, _.constant(value), false);
  },
  service: function(key, Constructor) {
    this.factory(key, function() {
```



```
    });
  }
};
```

The work we should do involves creating an instance of `Constructor`. It also involves injecting any dependencies it may have. As it happens, in Chapter 9 we implemented the injector method that does *exactly* that: `instantiate`.

src/injector.js

```
providerCache.$provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  },
  factory: function(key, factoryFn, enforce) {
    this.provider(key, {
      $get: enforce === false ? factoryFn : enforceReturnValue(factoryFn)
    });
  },
  value: function(key, value) {
    this.factory(key, _.constant(value), false);
  },
  service: function(key, Constructor) {
    this.factory(key, function() {
      return instanceInjector.instantiate(Constructor);
    });
  }
};
```

Now we see how it all builds on the same foundation. All of factories, values, and services are really just providers under the hood. They give application developers a *high-level API* for making providers that are specialized for certain kinds of patterns. Because of this, application developers actually rarely need to use raw providers. But they are always there, and the low-level access is available when needed.

Decorators

The final dependency injection feature still missing is *decorators*. Decorators are a bit different from all the other features we have seen, in that you do not use a decorator to define a dependency as such. You use a decorator to *modify some existing dependency*.

Decorators are an implementation of the object-oriented Decorator design pattern, which is where the name comes from.

What's neat about decorators is how you can use them to modify dependencies you yourself did not create. You can register a decorator for a dependency provided by some library, or by core Angular itself, as Brian Ford has described.

Let's see how a decorator could work. In this example we create a factory, and then decorate that factory with a decorator with the same name. The decorator is a function, that can be dependency-injected like a factory, but that also has a special `$delegate` argument available to it. The `$delegate` is the original dependency that is being decorated. In this case, it is the object returned by the `aValue` factory:

test/injector_spec.js

```
it('allows changing an instance using a decorator', function() {
  var module = angular.module('myModule', []);
  module.factory('aValue', function() {
    return {aKey: 42};
  });
  module.decorator('aValue', function($delegate) {
    $delegate.decoratedKey = 43;
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('aValue').aKey).toBe(42);
  expect(injector.get('aValue').decoratedKey).toBe(43);
});
```

You can also have several decorators on a single dependency. When you do, all of them will be applied to the dependency in turn:

test/injector_spec.js

```
it('allows multiple decorators per service', function() {
  var module = angular.module('myModule', []);
  module.factory('aValue', function() {
    return {};
  });
  module.decorator('aValue', function($delegate) {
    $delegate.decoratedKey = 42;
  });
  module.decorator('aValue', function($delegate) {
    $delegate.otherDecoratedKey = 43;
```



```
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('aValue').decoratedKey).toBe(42);
  expect(injector.get('aValue').otherDecoratedKey).toBe(43);
});
```

As we discussed, decorator functions can be dependency-injected also with other things than `$delegate`:

test/injector_spec.js

```
it('uses dependency injection with decorators', function() {
  var module = angular.module('myModule', []);
  module.factory('aValue', function() {
    return {};
  });
  module.constant('a', 42);
  module.decorator('aValue', function(a, $delegate) {
    $delegate.decoratedKey = a;
  });

  var injector = createInjector(['myModule']);

  expect(injector.get('aValue').decoratedKey).toBe(42);
});
```

So, what we need is a `decorator` method on the module API. It will, just like the other module API methods, queue up a call to a corresponding method on `$provide`. That method takes two arguments: The name of the dependency to decorate and the decorator function.

What we need to do is hook into the creation of the dependency that's being decorated. For that purpose, we need to first obtain its provider:

src/injector.js

```
providerCache.$provide = {
  constant: function(key, value) {
    if (key === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid constant name!';
    }
    providerCache[key] = value;
    instanceCache[key] = value;
  },
  provider: function(key, provider) {
    if (_.isFunction(provider)) {
```



```
      provider = providerInjector.instantiate(provider);
    }
    providerCache[key + 'Provider'] = provider;
  },
  factory: function(key, factoryFn, enforce) {
    this.provider(key, {
      $get: enforce === false ? factoryFn : enforceReturnValue(factoryFn)
    });
  },
  value: function(key, value) {
    this.factory(key, _.constant(value), false);
  },
  service: function(key, Constructor) {
    this.factory(key, function() {
      return instanceInjector.instantiate(Constructor);
    });
  },
  decorator: function(serviceName, decoratorFn) {
    var provider = providerInjector.get(serviceName + 'Provider');
  }
};
```

When the dependency is created, we need to grab that provider's return value and modify it somehow. What we'll do to make that possible is *override* the provider's `$get` method, with our own, decorated version:

src/injector.js

```
decorator: function(serviceName, decoratorFn) {
  var provider = providerInjector.get(serviceName + 'Provider');
  var original$get = provider.$get;
  provider.$get = function() {
    var instance = instanceInjector.invoke(original$get, provider);
    // Modifications will be done here
    return instance;
  };
}
```

In this first version we just call through to the original `$get` method and return its return value. We've overridden `$get` but are not yet doing anything special in our overridden version.

The final step is to actually apply the decorator function. Remember that it's a function, called with dependency injection, that has the additional `$delegate` argument available to it. That means we can call it with the instance injector's invoke, passing the delegate in with the locals argument that we implemented in Chapter 9:

src/injector.js

```
decorator: function(serviceName, decoratorFn) {
  var provider = providerInjector.get(serviceName + 'Provider');
  var original$get = provider.$get;
  provider.$get = function() {
    var instance = instanceInjector.invoke(original$get, provider);
    instanceInjector.invoke(decoratorFn, null, {$delegate: instance});
    return instance;
  };
}
```

Now we just need to expose the `decorator` method through the module API, which will make our tests pass:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  factory: invokeLater('$provide', 'factory'),
  value: invokeLater('$provide', 'value'),
  service: invokeLater('$provide', 'service'),
  decorator: invokeLater('$provide', 'decorator'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks,
  _runBlocks: []
};
```

And that's all there is to decorators, and all there is to dependency injection in Angular!

Integrating Scopes, Expressions, and Filters with The Injector

To conclude our coverage of dependency injection, we are going to revisit the code written earlier in the book and retrofit it to integrate with modules and dependency injection. This is important because otherwise these features won't be exposed to application developers.

At the beginning of Chapter 9 we saw how the `angular` global and the module loader come to be when you invoke `setupModuleLoader()` from `loader.js`. Now we're going to add another layer on top of that. In this layer we set up the module loader *and* register some

core Angular components to it. This is where we'll plug in the expression parser, the root scope, the filter service, and the filter filter.

> In the final part of the book we'll complete this picture by implementing the full Angular bootstrapping process.

The core component registration will go into a new file, called `src/angular_public.js`. Let's write the first test for it in `test/angular_public_spec.js`:

test/angular_public_spec.js

```
describe('angularPublic', function() {
  'use strict';

  it('sets up the angular object and the module loader', function() {
    publishExternalAPI();

    expect(window.angular).toBeDefined();
    expect(window.angular.module).toBeDefined();
  });

});
```

This actually checks nothing else than that the `publishExternalAPI` function should call through to `setupModuleLoader`. This is easy enough to implement:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);
}
```

What the function should also do is set up a module called `ng`:

test/angular_public_spec.js

```
it('sets up the ng module', function() {
  publishExternalAPI();

  expect(createInjector(['ng'])).toBeDefined();
});
```

It does this by calling `angular.module`:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
}
```

The `ng` module is where all the services, directives, filters, and other components provided by Angular itself will be. As we'll see when we build the bootstrapping process, this module is automatically included to every Angular application, so as an application developer you don't really even need to be aware of its existence. But it is how Angular exposes its own services to each other and to applications.

> In the process of reorganizing code to use dependency injection, we'll be temporarily breaking most of our unit tests, so you will be seeing a lot of red in the Testem runner. By the end of the process everything should be working again.

The first thing we'll put into the `ng` module is `$filter`, our filter service:

test/angular_public_spec.js

```
it('sets up the $filter service', function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  expect(injector.has('$filter')).toBe(true);
});
```

It is registered as a provider:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
}
```

So we expect there to be a `$FilterProvider` constructor that makes a provider for `$filter`. This constructor will be set up in `filter.js`, together with the implementation itself.

In Part 2 of the book, what we did in `filter.js` was to just set up global functions called `register` and `filter`. Now we need to wrap those into a provider. The `register` function becomes a method on the provider, and the `filter` function becomes the return value of the `$get` method of the provider. In other words, it becomes the `$filter` service:

src/filter.js


```
function $FilterProvider() {

  var filters = {};

  this.register = function(name, factory) {
    if (_.isObject(name)) {
      return _.map(name, function(factory, name) {
        return this.register(name, factory);
      }, this);
    } else {
      var filter = factory();
      filters[name] = filter;
      return filter;
    }
  };

  this.$get = function() {
    return function filter(name) {
      return filters[name];
    };
  };

}
```

We now need to revisit `filter_spec.js` so that it uses the filter provider from the `ng` module instead of assuming the `register` and `filter` functions are globally available:

test/filter_spec.js

```
/* jshint globalstrict: true */
/* global angular: false, publishExternalAPI: false, createInjector: false */
'use strict';

describe("filter", function() {

  beforeEach(function() {
    publishExternalAPI();
  });

  it('can be registered and obtained', function() {
    var myFilter = function() { };
    var myFilterFactory = function() {
      return myFilter;
    };
    var injector = createInjector(['ng', function($filterProvider) {
      $filterProvider.register('my', myFilterFactory);
    }]);
    var $filter = injector.get('$filter');
```

```
    expect($filter('my')).toBe(myFilter);
  });

  it('allows registering multiple filters with an object', function() {
    var myFilter = function() { };
    var myOtherFilter = function() { };
    var injector = createInjector(['ng', function($filterProvider) {
      $filterProvider.register({
        my: function() {
          return myFilter;
        },
        myOther: function() {
          return myOtherFilter;
        }
      });
    }]);

    var $filter = injector.get('$filter');
    expect($filter('my')).toBe(myFilter);
    expect($filter('myOther')).toBe(myOtherFilter);
  });

});
```

An additional thing that the `$filter` service does is to *instantiate filters with dependency injection and make them available as regular factories*. If you register a filter called `my`, it will be available not only through the `$filter` service but also as a regular dependency by the name of `myFilter`:

test/filter_spec.js

```
it('is available through injector', function() {
  var myFilter = function() { };
  var injector = createInjector(['ng', function($filterProvider) {
    $filterProvider.register('my', function() {
      return myFilter;
    });
  }]);
  expect(injector.has('myFilter')).toBe(true);
  expect(injector.get('myFilter')).toBe(myFilter);
});
```

The filter factory may also have dependencies, which are injected to it:

test/filter_spec.js

```
it('may have dependencies in factory', function() {
  var injector = createInjector(['ng', function($provide, $filterProvider) {
    $provide.constant('suffix', '!');
    $filterProvider.register('my', function(suffix) {
      return function(v) {
        return suffix + v;
      };
    });
  }]);
  expect(injector.has('myFilter')).toBe(true);
});
```

The `$FilterProvider` makes this possible by registering each filter into the `$provide` service as a regular factory:

src/filter.js

```
function $FilterProvider($provide) {

  var filters = {};

  this.register = function(name, factory) {
    if (_.isObject(name)) {
      return _.map(name, function(factory, name) {
        return register(name, factory);
      });
    } else {
      return $provide.factory(name + 'Filter', factory);
    }
  };

  this.$get = function() {
    return function filter(name) {
      return filters[name];
    };
  };

}
$FilterProvider.$inject = ['$provide'];
```

At runtime the `$filter` service uses the `$injector` to obtain the filter function. We no longer need the internal `filters` object at this point, as everything is stored in the DI system:

src/filter.js

```
function $FilterProvider($provide) {

  this.register = function(name, factory) {
    if (_.isObject(name)) {
      return _.map(name, function(factory, name) {
        return register(name, factory);
      });
    } else {
      return $provide.factory(name + 'Filter', factory);
    }
  };

  this.$get = ['$injector', function($injector) {
    return function filter(name) {
      return $injector.get(name + 'Filter');
    };
  }];

}
$FilterProvider.$inject = ['$provide'];
```

Angular also provides a shortcut for registering filters through the public module API. It has a `filter` method, using which filters can be registered. This is usually much more convenient than using the `$filterProvider` through a config block:

test/filter_spec.js

```
it('can be registered through module API', function() {
  var myFilter = function() { };
  var module = angular.module('myModule', [])
    .filter('my', function() {
      return myFilter;
    });

  var injector = createInjector(['ng', 'myModule']);

  expect(injector.has('myFilter')).toBe(true);
  expect(injector.get('myFilter')).toBe(myFilter);
});
```

In the module loader we introduce the `filter` method:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
```



```
  provider: invokeLater('$provide', 'provider'),
  factory: invokeLater('$provide', 'factory'),
  value: invokeLater('$provide', 'value'),
  service: invokeLater('$provide', 'service'),
  decorator: invokeLater('$provide', 'decorator'),
  filter: invokeLater('$filterProvider', 'register'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks,
  _runBlocks: []
};
```

Unlike all the other registration methods, the `filter` method does not queue up a `$provide` method call. Instead, it queues up a method call to `$filterProvider`, which is the provider defined in `filter.js`.

We also need to revisit the filter filter, which has a test that is still relying on a public `filter` method. Let's fix that:

test/filter_filter_spec.js

```
/* jshint globalstrict: true */
/* global publishExternalAPI: false, createInjector: false, parse: false */
'use strict';

describe("filter filter", function() {

  beforeEach(function() {
    publishExternalAPI();
  });

  it('is available', function() {
    var injector = createInjector(['ng']);
    expect(injector.has('filterFilter')).toBe(true);
  });

  // ...

});
```

> The rest of the tests in this file are still broken, as we've yet to fix the `parse` service. We'll return to this shortly.

The filter filter is still expecting to be able to register itself using a public `register` method. Let's flip things around so that the filter filter does not actually register itself, but is registered in the filter service instead. Remove the following line from `filter_filter.js`:


```
register('filter', filterFilter);
```

Then add the registration into `filter.js`:

src/loader.js

```
function $FilterProvider($provide) {

  this.register = function(name, factory) {
    if (_.isObject(name)) {
      return _.map(name, function(factory, name) {
        return register(name, factory);
      });
    } else {
      return $provide.factory(name + 'Filter', factory);
    }
  };

  this.$get = ['$injector', function($injector) {
    return function filter(name) {
      return $injector.get(name + 'Filter');
    };
  }];

  this.register('filter', filterFilter);

}
$FilterProvider.$inject = ['$provide'];
```

Moving on to the expression parser, it is also going to be in the `ng` module:

test/angular_public_spec.js

```
it('sets up the $parse service', function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  expect(injector.has('$parse')).toBe(true);
});
```

It is registered as a provider:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
  ngModule.provider('$parse', $ParseProvider);
}
```


Like with the filter provider, we expect there to be a `$ParseProvider` constructor that makes a provider for `$parse`. This constructor will be set up in `parse.js`, where it wraps the previous global `parse` function:

src/parse.js

```
function $ParseProvider() {

  this.$get = function() {
    return function(expr) {
      switch (typeof expr) {
        case 'string':
          var lexer = new Lexer();
          var parser = new Parser(lexer);
          var oneTime = false;
          if (expr.charAt(0) === ':' && expr.charAt(1) === ':') {
            oneTime = true;
            expr = expr.substring(2);
          }
          var parseFn = parser.parse(expr);

          if (parseFn.constant) {
            parseFn.$$watchDelegate = constantWatchDelegate;
          } else if (oneTime) {
            parseFn.$$watchDelegate = parseFn.literal ?
                                        oneTimeLiteralWatchDelegate :
                                        oneTimeWatchDelegate;
          } else if (parseFn.inputs) {
            parseFn.$$watchDelegate = inputsWatchDelegate;
          }

          return parseFn;
        case 'function':
          return expr;
        default:
          return _.noop;
      }
    };

  };

}
```

So what was previously `function parse()` became the return value of the provider's `$get` method. That return value is what you will get when you inject `$parse`.

Another thing that we need to do here is to obtain the `$filter` service, which is used internally in the parser. First, let's inject it and give it to the `Parser` constructor:

src/parse.js


```
this.$get = ['$filter', function($filter) {
  return function(expr) {
    switch (typeof expr) {
      case 'string':
        var lexer = new Lexer();
        var parser = new Parser(lexer, $filter);
        var oneTime = false;
        if (expr.charAt(0) === ':' && expr.charAt(1) === ':') {
          oneTime = true;
          expr = expr.substring(2);
        }
        var parseFn = parser.parse(expr);

        if (parseFn.constant) {
          parseFn.$$watchDelegate = constantWatchDelegate;
        } else if (oneTime) {
          parseFn.$$watchDelegate = parseFn.literal ? oneTimeLiteralWatchDelegate :
                                                      oneTimeWatchDelegate;
        } else if (parseFn.inputs) {
          parseFn.$$watchDelegate = inputsWatchDelegate;
        }

        return parseFn;
      case 'function':
        return expr;
      default:
        return _.noop;
    }
  };
}];
```

The `Parser` constructor passes `$filter` forward to the `ASTCompiler` constructor:

src/parse.js

```
function Parser(lexer, $filter) {
  this.lexer = lexer;
  this.ast = new AST(this.lexer);
  this.astCompiler = new ASTCompiler(this.ast, $filter);
}
```

The `ASTCompiler` constructor stores it in an attribute:

src/parse.js

```
function ASTCompiler(astBuilder, $filter) {
  this.astBuilder = astBuilder;
  this.$filter = $filter;
}
```


The first location where we now need this attribute is for the `filter` function that gets passed into the generated code. What we pass is in fact the `$filter` service. That is where filters will be obtained from at runtime:

src/parse.js

```
/* jshint -W054 */
var fn = new Function(
  'ensureSafeMemberName',
  'ensureSafeObject',
  'ensureSafeFunction',
  'ifDefined',
  'filter',
  fnString)(
    ensureSafeMemberName,
    ensureSafeObject,
    ensureSafeFunction,
    ifDefined,
    this.$filter);
/* jshint +W054 */
```

We also need to pass `$filter` to `markConstantAndWatchExpressions`, which has until now been using the global `filter` function:

src/parse.js

```
ASTCompiler.prototype.compile = function(text) {
  var ast = this.astBuilder.ast(text);
  var extra = '';
  markConstantAndWatchExpressions(ast, this.$filter);
  // ...
};
```

In the implementation of `markConstantAndWatchExpressions` we need to accept this argument and pass it along in all the recursive invocations, so that we can finally use it in `CallExpression`s if we end up needing it:

src/parse.js

```
function markConstantAndWatchExpressions(ast, $filter) {
  var allConstants;
  var argsToWatch;
  switch (ast.type) {
  case AST.Program:
    allConstants = true;
    _.forEach(ast.body, function(expr) {
```



```
      markConstantAndWatchExpressions(expr, $filter);
      allConstants = allConstants && expr.constant;
    });
    ast.constant = allConstants;
    break;
  case AST.Literal:
    ast.constant = true;
    ast.toWatch = [];
    break;
  case AST.Identifier:
    ast.constant = false;
    ast.toWatch = [ast];
    break;
  case AST.ArrayExpression:
    allConstants = true;
    argsToWatch = [];
    _.forEach(ast.elements, function(element) {
      markConstantAndWatchExpressions(element, $filter);
      allConstants = allConstants && element.constant;
      if (!element.constant) {
        argsToWatch.push.apply(argsToWatch, element.toWatch);
      }
    });
    ast.constant = allConstants;
    ast.toWatch = argsToWatch;
    break;
  case AST.ObjectExpression:
    allConstants = true;
    argsToWatch = [];
    _.forEach(ast.properties, function(property) {
      markConstantAndWatchExpressions(property.value, $filter);
      allConstants = allConstants && property.value.constant;
      if (!property.value.constant) {
        argsToWatch.push.apply(argsToWatch, property.value.toWatch);
      }
    });
    ast.constant = allConstants;
    ast.toWatch = argsToWatch;
    break;
  case AST.ThisExpression:
    ast.constant = false;
    ast.toWatch = [];
    break;
  case AST.MemberExpression:
    markConstantAndWatchExpressions(ast.object, $filter);
    if (ast.computed) {
      markConstantAndWatchExpressions(ast.property, $filter);
    }
    ast.constant = ast.object.constant &&
                   (!ast.computed || ast.property.constant);
    ast.toWatch = [ast];
```



```
    break;
  case AST.CallExpression:
    var stateless = ast.filter && !$filter(ast.callee.name).$stateful;
    allConstants = stateless ? true : false;
    argsToWatch = [];
    _.forEach(ast.arguments, function(arg) {
      markConstantAndWatchExpressions(arg, $filter);
      allConstants = allConstants && arg.constant;
      if (!arg.constant) {
        argsToWatch.push.apply(argsToWatch, arg.toWatch);
      }
    });
    ast.constant = allConstants;
    ast.toWatch = stateless ? argsToWatch : [ast];
    break;
  case AST.AssignmentExpression:
    markConstantAndWatchExpressions(ast.left, $filter);
    markConstantAndWatchExpressions(ast.right, $filter);
    ast.constant = ast.left.constant && ast.right.constant;
    ast.toWatch = [ast];
    break;
  case AST.UnaryExpression:
    markConstantAndWatchExpressions(ast.argument, $filter);
    ast.constant = ast.argument.constant;
    ast.toWatch = ast.argument.toWatch;
    break;
  case AST.BinaryExpression:
    markConstantAndWatchExpressions(ast.left, $filter);
    markConstantAndWatchExpressions(ast.right, $filter);
    ast.constant = ast.left.constant && ast.right.constant;
    ast.toWatch = ast.left.toWatch.concat(ast.right.toWatch);
    break;
  case AST.LogicalExpression:
    markConstantAndWatchExpressions(ast.left, $filter);
    markConstantAndWatchExpressions(ast.right, $filter);
    ast.constant = ast.left.constant && ast.right.constant;
    ast.toWatch = [ast];
    break;
  case AST.ConditionalExpression:
    markConstantAndWatchExpressions(ast.test, $filter);
    markConstantAndWatchExpressions(ast.consequent, $filter);
    markConstantAndWatchExpressions(ast.alternate, $filter);
    ast.constant =
      ast.test.constant && ast.consequent.constant && ast.alternate.constant;
    ast.toWatch = [ast];
    break;
  }
}
```

While this fixes the new unit test we added to `parse_spec.js`, the other parsing unit tests

are still broken because they rely on the global function `parse` being present. We need to change `parse_spec.js` so that instead of using that global function, it'll actually make an injector and get the `$parse` service from it in a `beforeEach` block. Here's the new preamble for the test file

test/parse_spec.js

```
/* jshint globalstrict: true */
/* global publishExternalAPI: false, createInjector: false */
'use strict';

describe("parse", function() {

  var parse;

  beforeEach(function() {
    publishExternalAPI();
    parse = createInjector(['ng']).get('$parse');
  });

  // ...

});
```

The tests that register and use filters also need to be updated so that they create their own injectors and do the registration through the filter provider:

test/parse_spec.js

```
it('can parse filter expressions', function() {
  parse = createInjector(['ng', function($filterProvider) {
    $filterProvider.register('upcase', function() {
      return function(str) {
        return str.toUpperCase();
      };
    });
  }]).get('$parse');
  var fn = parse('aString | upcase');
  expect(fn({aString: 'Hello'})).toEqual('HELLO');
});

it('can parse filter chain expressions', function() {
  parse = createInjector(['ng', function($filterProvider) {
    $filterProvider.register('upcase', function() {
      return function(s) {
        return s.toUpperCase();
      };
    });
    $filterProvider.register('exclamate', function() {
      return function(s) {
```



```
        return s + '!';
      };
    });
  }]).get('$parse');
  var fn = parse('"hello" | upcase | exclamate');
  expect(fn()).toEqual('HELLO!');
});

it('can pass an additional argument to filters', function() {
  parse = createInjector(['ng', function($filterProvider) {
    $filterProvider.register('repeat', function() {
      return function(s, times) {
        return _.repeat(s, times);
      };
    });
  }]).get('$parse');
  var fn = parse('"hello" | repeat:3');
  expect(fn()).toEqual('hellohellohello');
});

it('can pass several additional arguments to filters', function() {
  parse = createInjector(['ng', function($filterProvider) {
    $filterProvider.register('surround', function() {
      return function(s, left, right) {
        return left + s + right;
      };
    });
  }]).get('$parse');
  var fn = parse('"hello" | surround:"*":"!"');
  expect(fn()).toEqual('*hello!');
});

// ...

it('marks filters constant if arguments are', function() {
  parse = createInjector(['ng', function($filterProvider) {
    $filterProvider.register('aFilter', function() {
      return _.identity;
    });
  }]).get('$parse');
  expect(parse('[1, 2, 3] | aFilter').constant).toBe(true);
  expect(parse('[1, 2, a] | aFilter').constant).toBe(false);
  expect(parse('[1, 2, 3] | aFilter:42').constant).toBe(true);
  expect(parse('[1, 2, 3] | aFilter:a').constant).toBe(false);
});
```

We can now also go back to the filter filter test suite and fix the rest of the tests there by obtaining the `$parse` service:

test/filter_filter_spec.js

```
/* jshint globalstrict: true */
/* global publishExternalAPI: false, createInjector: false */
'use strict';

describe("filter filter", function() {

  var parse;

  beforeEach(function() {
    publishExternalAPI();
    parse = createInjector(['ng']).get('$parse');
  });

  // ...

});
```

With `$parse` now taken care of, let's turn our attention to `scope.js`. There are already some unit tests for it failing, because they also rely on the presence of global `parse` and `register` functions that no longer exist. We'll fix this momentarily.

Just like `$filter` and `$parse`, the ng module should have a scope instance included in it - the `$rootScope`:

test/angular_public_spec.js

```
it('sets up the $rootScope', function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  expect(injector.has('$rootScope')).toBe(true);
});
```

And just like with `$filter` and `$parse`, the `$rootScope` is registered using a provider:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
}
```

The way we should implement this in `scope.js` is to introduce the `$RootScopeProvider` and its `$get` method and then *move all existing code from `scope.js` so it's inside the `$get` method*:

src/scope.js

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

function $RootScopeProvider() {

  this.$get = function() {

    //  Move all previous code from scope.js here.

  };

}
```

We also need to return a value from `$get`, so as the last thing we do we'll make an instance of `Scope` and return it:

src/scope.js

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

function $RootScopeProvider() {

  this.$get = function() {

    // All previous code from scope.js goes here.

    var $rootScope = new Scope();
    return $rootScope;
  };

}
```

That returned scope object will be the `$rootScope`. Notice how everything in `scope.js` is now private: The `Scope` constructor or any of its supporting functions are not accessible outside of the provider. The `$rootScope` is the only thing we expose.

To fix the issue with the `parse` function, we need to modify our code to use the `$parse` service instead. Since we now have a provider and a `$get` method, we can inject `$parse`:

src/scope.js


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

function $RootScopeProvider() {

  this.$get = ['$parse', function($parse) {

    // All previous code from scope.js goes here.

    var $rootScope = new Scope();
    return $rootScope;
  }];

}
```

Now we can change the `$watch` function to use `$parse` instead of `parse`:

src/scope.js

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  // ...
  watchFn = $parse(watchFn);
  // ...
};
```

We'll also do the same for `$watchCollection`:

src/scope.js

```
Scope.prototype.$watchCollection = function(watchFn, listenerFn) {
  // ...
  watchFn = $parse(watchFn);
  // ...
};
```

And we'll do the same for `$eval`:

src/scope.js

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

And there we have the implementation of `$rootScope`! The tests for it are still broken though. Let's fix them.

First, update the JSHint directive on the top of `scope_spec.js`. We should no longer need global access to anything but the module and injector setup functions:

test/scope_spec.js

```
/* global publishExternalAPI: false, createInjector: false */
```

Next, sadly, the very first unit test we wrote in Chapter 1 will now need to go:

test/scope_spec.js

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

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

A `Scope` can in fact *not* be constructed with the constructor, since that constructor isn't available, so this test should be removed. How you *can* get access to a scope is via `$rootScope`, and we already have that tested in `angular_public_spec.js`.

Now we need to go through each of the nested `describe` blocks in `scope_spec.js` and set up a scope object using an injector, so that the existing tests have access to one.

In the `describe('digest')` block, we'll make an injector to get the root scope, and then we'll just use that in the tests:

test/scope_spec.js

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

  var scope;

  beforeEach(function() {
    publishExternalAPI();
    scope = createInjector(['ng']).get('$rootScope');
  });

  // ...

});
```

In the `describe('inheritance')` block, we'll add a new `beforeEach` block to obtain the root scope, which we'll use as *the parent scope*:

test/scope_spec.js


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

  var parent;

  beforeEach(function() {
    publishExternalAPI();
    parent = createInjector(['ng']).get('$rootScope');
  });

  // ...

});
```

In every test inside the `describe('inheritance')` block, we now need to remove the first line that sets up the parent scope (`var parent = new Scope()`). These are now replaced by the parent scope created in the `beforeEach` block. So, for example, the first unit test just uses `parent`, but does not construct it:

test/scope_spec.js

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

  var child = parent.$new();

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

Repeat the same trick for all the tests in the block.

The are two exceptions to this, the first being the "can be nested at any depth" spec, in which we'll just use `parent` as the value of `a`:

test/scope_spec.js

```
it("can be nested at any depth", function() {
  var a = parent;
  // ...
});
```

The second exception is the "can take some other scope as the parent" test, in which we make the two parents from the root scope:

test/scope_spec.js


```
it('can take some other scope as the parent', function() {
  var prototypeParent = parent.$new();
  var hierarchyParent = parent.$new();
  var child = prototypeParent.$new(false, hierarchyParent);
  // ...
});
```

In the `describe('$watchCollection')` block, we'll again get the root scope using an injector, instead of using the `Scope` constructor:

test/scope_spec.js

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

  var scope;

  beforeEach(function() {
    publishExternalAPI();
    scope = createInjector(['ng']).get('$rootScope');
  });

  // ...

});
```

In the `describe('Events')` block, we'll set the parent scope to be the root scope and keep the rest of the set up as it was:

test/scope_spec.js

```
describe("Events", function() {
  var parent;
  var scope;
  var child;
  var isolatedChild;

  beforeEach(function() {
    publishExternalAPI();
    parent = createInjector(['ng']).get('$rootScope');
    scope = parent.$new();
    child = scope.$new();
    isolatedChild = scope.$new(true);
  });

  // ..

});
```

Finally, there's one test that registers a filter in `scope_spec.js`, which we should modify to set up its own injector and use the `$filterProvider`:

test/scope_spec.js

```
it('allows $stateful filter value to change over time', function(done) {
  var injector = createInjector(['ng', function($filterProvider) {
    $filterProvider.register('withTime', function() {
      return _.extend(function(v) {
        return new Date().toISOString() + ': ' + v;
      }, {
        $stateful: true
      });
    });
  }]);
  scope = injector.get('$rootScope');

  var listenerSpy = jasmine.createSpy();
  scope.$watch('42 | withTime', listenerSpy);
  scope.$digest();
  var firstValue = listenerSpy.calls.mostRecent().args[0];

  setTimeout(function() {
    scope.$digest();
    var secondValue = listenerSpy.calls.mostRecent().args[0];
    expect(secondValue).not.toEqual(firstValue);
    done();
  }, 100);
});
```

At this point, all our tests should be passing again.

Making a Configurable Provider: Digest TTL

As we've now wrapped `$rootScope` in a provider, we're able to implement some configurability for it. Back in Chapter 1 we introduced the concept of the digest TTL, which is the number of digest iterations we attempt to do before throwing an exception if things still aren't stable. We set the TTL to the constant value 10, but it should actually be configurable by application developers. This will be our first use case for using raw providers.

A user can set the TTL using a configuration method from `$rootScopeProvider`. Let's add a test for it in a new `describe` block in `scope_spec.js`:

test/scope_spec.js

```
describe('TTL configurability', function() {

  beforeEach(function() {
    publishExternalAPI();
  });

  it('allows configuring a shorter TTL', function() {
    var injector = createInjector(['ng', function($rootScopeProvider) {
      $rootScopeProvider.digestTtl(5);
    }]);
    var scope = injector.get('$rootScope');

    scope.counterA = 0;
    scope.counterB = 0;

    scope.$watch(
      function(scope) { return scope.counterA; },
      function(newValue, oldValue, scope) {
        if (scope.counterB < 5) {
          scope.counterB++;
        }
      }
    );
    scope.$watch(
      function(scope) { return scope.counterB; },
      function(newValue, oldValue, scope) {
        scope.counterA++;
      }
    );

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

});
```

This is an adaptation of the TTL test we wrote in Chapter 1. We again have two interdependent watchers, but this time we stop incrementing one of the counters when it reaches the value 5. With the default TTL value of 10 these watches will not be a problem.

But we do not use the default TTL value. Instead, we configure `$rootScopeProvider` (in a function module) to use a TTL of 5. This should cause the digest to throw an exception.

In `$RootScopeProvider` we now need to introduce this method. It will set the value of a local variable called `TTL`, whose default value is 10:

src/scope.js

```
function $RootScopeProvider() {

  var TTL = 10;
```



```
  this.digestTtl = function(value) {
    if (_.isNumber(value)) {
      TTL = value;
    }
    return TTL;
  };

  // ...

}
```

Notice that the method can also be used to get the current value of the TTL.

In `$digest` we should now use the TTL value instead of a hard-coded value of 10:

src/scope.js

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

  if (this.$root.$$applyAsyncId) {
    clearTimeout(this.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

    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 TTL + " digest iterations reached";
        }
    } while (dirty || this.$$asyncQueue.length);
    this.$clearPhase();

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



```
        }
    }
};
```

Now we see a bit more clearly what the provider abstraction is useful for. As application developers, we can configure `$rootScope` without having access to its code, and without having to override anything with decorators. The `$rootScopeProvider` provides an API for configuring the `$rootScope` TTL. We can set it to a number higher than 10 if we have long chains of dependent watches. We can also set it to a number lower than 10 if we want to enforce fewer digest iterations for performance reasons.

Summary

You now know pretty much everything there is to know about the AngularJS dependency injection framework, because you've completed a full implementation of it. In this chapter we topped things off by adding the high-level features that most application developers use every day, and saw how they are actually quite simple given the foundational pieces we've laid out in earlier chapters.

In this chapter you have learned:

- How you can inject an `$injector` and how that `$injector` may be either the provider injector or the instance injector depending on where you're injecting it.
- How you can inject `$provide` and register additional components in config blocks, function modules.
- How config blocks work, and how they are implemented by simply invoking them with the provider injector.
- How a config block can be specified as the third argument to `angular.module`.
- How run blocks work, and how they are implemented by simply invoking them with the instance injector.
- That run blocks are deferred to a moment when *all* modules are loaded.
- How you can define a function module, which is essentially the same as a config block.
- How Angular's internal hash key and hash map implementations work, and how they deal with compound data structures by adding a `$$hashKey` attribute.
- How factories are implemented on top of providers.
- How values are implemented on top of factories.
- How services are implemented on top of factories.
- How decorators work, by overriding the `$get` method of the provider they decorate.
- How decorators use the `locals` argument of `injector.invoke` to make the decorated `$delegate` available for injection.
- That there's an `ng` module that holds Angular's core components in every Angular application.
- How filters, scopes, and the expression parser integrate with the dependency injection features.
- How you can adjust the digest TTL by calling a method on `$rootScopeProvider`.

Part IV

Utilities

In this part of the book, we are going to build a few essential utilities that Angular ships with. They are familiar to Angular application developers, but also used by Angular internally. We will need them as we implement the directive system in the next part.

Chapter 13

Promises

The web browser's JavaScript environment is a place where almost everything happens *asynchronously*: We set up code that sits there dormant, waiting for things to happen. The code only activates when the user does something, when data is received from the server, or when a timer fires. This fundamentally affects how JavaScript applications are constructed.

For the longest time, the preferred tool for dealing with asynchronous computation in JavaScript has been *callback functions*. You pass an API a function that it will call later when something happens. Almost all of the built-in asynchronous APIs of the JavaScript language and the browser environment use callbacks:

```
element.addEventListener('click', function(evt) {
  console.log('clicked!', evt);
});
```

However, programming with callbacks has some issues that may result in a lot of unnecessary complexity:

- Callback-style APIs conflate business logic with implementation details. For example, your actual function arguments are mixed with callback function arguments as in `computeBalance(from, to, onDone)`. What you want is `computeBalance(from, to)` but you have to add the third argument when the computation is asynchronous.
- When there are many asynchronous callbacks involved, control flow becomes difficult to follow, as well as cumbersome to write and maintain. This may cause problems like the infamous pyramid of doom.
- There is no established approach to dealing with errors. Catching errors from asynchronous code relies on a convention of having special error arguments in callbacks, and the pattern must be repeated for every callback.

Promises

Promises are an attempt to address these problems in certain situations. They are essentially a mechanism that bundles the future results of an asynchronous call into an object. An

asynchronous function does not return a value, but the *promise* that there will be a value at some point. The Promise object gets you access to that value once it becomes available.

> The value that a Promise resolves to may also be `undefined`, which is analogous to a function that doesn't return anything. Such a Promise can still be useful for signalling that the computation is finished.

Promises address the conflation of business logic with callbacks by separating callback arguments from regular function arguments. Your function does not *take* a callback. It *returns* a Promise that takes the callback:

```
computeBalance(from, to).then(function(balance) {
  // ...
});
```

Promises address the "pyramid of doom" problem by supporting *chaining*. Registering a Promise callback returns a new Promise, allowing for a chained, flat control flow:

```
computeBalance(a, b)
  .then(storeBalance)
  .then(displayResults);
```

Promises address the problem of ad hoc error handling by having an explicit API for errors:

```
computeBalance(a, b)
  .then(storeBalance)
  .catch(handleError);
```

Errors also propagate through Promise chains, so that errors in any step can be caught by one error handler - much like one `try..catch` in synchronous code handles any errors that happen within, no matter how many function calls deep they may be:

```
computeBalance(a, b)
  .then(storeBalance)
  .then(displayResults)
  .catch(handleError);
```

Promise Implementations

Promises as a concept have been around since the 1970s, and they have also been present in many libraries in JavaScript for several years. Among the most well known implementations are jQuery's Deferreds and the Q library by Kris Kowal. Today there are tens of alternative promise-like libraries available in JavaScript.

Since there are so many alternative implementations available, there are also ongoing efforts to support *interoperability* between them. The goal of those efforts is to make it easy for people to mix libraries that happen to use different Promise implementations. Most significantly, a community standard called Promises/A+ defines a specification and a test suite for how the `then` method of a Promise should behave. Many Promise libraries

implement this standard. For instance, Q is compliant, as are jQuery Deferreds starting from jQuery 3.0.

There is also a Promise implementation built right into the JavaScript language. It is defined in the ECMAScript 6 proposal and is already implemented by several browsers. Some of the newer DOM APIs such as Service Workers support native Promises right out of the gate. At the time of writing, it still remains to be seen how the roll-out of native ES6 Promises will play out and how it will affect the many existing Promise libraries out there.

Promises in AngularJS

Angular also ships with an implementation of Promises, in the built-in `$q` service. Angular's promises are modeled after the Q library (hence the name `$q`), and can be considered a stripped-down version of Q.

The `$q` service is Promises/A+ compliant. It can also be used in a way that's very similar to native ES6 Promises, as we will see in this chapter.

Perhaps the most important distinguishing feature of `$q` when compared to other Promise implementations is its integration with the digest loop. With `$q` everything happens within an Angular digest, so you don't have to worry about calling `scope.$apply`. Furthermore, whereas many Promise libraries use things like `setTimeout` to make things asynchronous, `$q` can simply use `$evalAsync`.

Further Reading

Matt Greer's JavaSript Promises... In Wicked Detail builds up a (non-Angular) Promise implementation with an approach very similar to this book.

Kyle Simpson's Promises chapter in "You Don't Know JS: Async & Performance is a well-written, thorough guide to using Promises in JavaScript. Though Angular's Promises have a slightly different API to the ES6 Promises used here, pretty much all of the content translates directly to `$q`.

Kris Kowal's A General Theory of Reactivity does a great job putting Promises in the bigger context of "reactivity", and compares and contrasts them with other asynchronous computation primitives, such as Streams.

The $q Provider

In this chapter we'll build up `$q` in its entirety, to form a complete picture of what Promises in Angular do. But first, we need to lay the groundwork. We should make sure that `$q` actually exists as part of the built-in `ng` module:

test/angular_public_spec.js

```
it('sets up $q', function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  expect(injector.has('$q')).toBe(true);
});
```


To make `$q` available, we're going to use a provider, just like we've done with `$parse` and `$rootScope`. The provider will live in a file called `q.js`:

src/q.js

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

function $QProvider() {

  this.$get = function() {

  };

}
```

To bring `$q` alive and to make the test pass we still need to register this provider to the `ng` module:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
}
```

Creating Deferreds

The first construct we are going to build into `$q` is not actually Promises, but a closely related concept called *Deferreds*.

If *Promise* is a promise that some value will become available in the future, a *Deferred* is the computation that makes that value available. The two always come in pairs, but are usually accessed by different parts of your code.

If you think of it in terms of data flow, the *producer* of the data has a Deferred, and the *consumer* of the data has a Promise. At some point, when the producer *resolves* the Deferred, the consumer will receive the Promise's *value*.

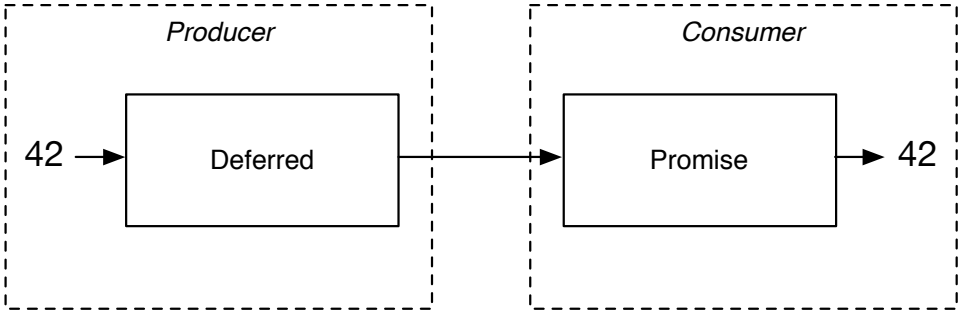

Figure 13.1: The relationship between a Deferred and a Promise

> ECMAScript 6 standard Promises do not have a concept of Deferreds and instead use plain functions to handle the producer side of the equation. AngularJS also supports this style, and we'll see how that works at the end of the chapter. Using Deferreds is still the more fundamental - and at the time of writing the more widely used - mechanism.

You construct a Deferred by calling the `defer` method of `$q`. Let's kick off our `$q` test suite with a test for that:

test/q_spec.js

```
/* jshint globalstrict: true */
/* global publishExternalAPI: false, createInjector: false */
'use strict';

describe("$q", function() {

  var $q;

  beforeEach(function() {
    publishExternalAPI();
    $q = createInjector(['ng']).get('$q');
  });

  it('can create a Deferred', function() {
    var d = $q.defer();
    expect(d).toBeDefined();
  });

});
```

Inside `$q`, Deferreds are created by a constructor function called `Deferred`. However, this is just an implementation detail of `$q`, because the constructor itself is not exposed to the outside world, whereas the `defer` method is:

src/q.js


```
this.$get = function() {

  function Deferred() {
  }

  function defer() {
    return new Deferred();
  }

  return {
    defer: defer
  };

};
```

Accessing The Promise of A Deferred

Deferreds and Promises always come in pairs. Whenever you create a Deferred, you also create a Promise. The Promise is accessible through the `promise` attribute of the Deferred:

test/q_spec.js

```
it('has a promise for each Deferred', function() {
  var d = $q.defer();
  expect(d.promise).toBeDefined();
});
```

Within `$q`, there's also an internal constructor function for Promises. The `Deferred` constructor invokes the `Promise` constructor to create a new Promise:

src/q.js

```
this.$get = function() {

  function Promise() {
  }

  function Deferred() {
    this.promise = new Promise();
  }


  function defer() {
    return new Deferred();
  }
```



```
  return {
    defer: defer
  };

};
```

Resolving A Deferred

Now that we know how to make Deferreds and Promises, we can talk about the actual point of their existence, which is to make asynchronous programming easier.

When we have a Deferred and a Promise, we can attach a *callback* on the Promise. Then, after the Deferred is resolved to a value, the callback on the Promise will *at some point in the future* get invoked with that value. We can use a Jasmine spy function as a callback when testing:

test/q_spec.js

```
it('can resolve a promise', function(done) {
  var deferred = $q.defer();
  var promise  = deferred.promise;

  var promiseSpy = jasmine.createSpy();
  promise.then(promiseSpy);

  deferred.resolve('a-ok');

  setTimeout(function() {
    expect(promiseSpy).toHaveBeenCalledWith('a-ok');
    done();
  }, 1);
});
```

We'll come back to what "some point in the future" means exactly in this context, but for the time being let's make this first test pass.

Promises store their internal state in an attribute called `$$state`:

src/q.js

```
function Promise() {
  this.$$state = {};
}
```

When a Promise's `then` method is called, the given callback is stored inside that state:

src/q.js

```
function Promise() {
  this.$$state = {};
}
Promise.prototype.then = function(onFulfilled) {
  this.$$state.pending = onFulfilled;
};
```

Then, as the paired Deferred is resolved, we can invoke the callback held by the Promise:

src/q.js

```
function Deferred() {
  this.promise = new Promise();
}
Deferred.prototype.resolve = function(value) {
  this.promise.$$state.pending(value);
};
```

This implementation satisfies the contract enforced by our test case, but is still way too simplistic for what we need. For one thing, this version is a bit too strict about the order in which things are done. It should be perfectly OK to register a Promise callback when the Deferred is *already resolved*, and still have that callback be invoked:

test/q_spec.js

```
it('works when resolved before promise listener', function(done) {
  var d = $q.defer();
  d.resolve(42);

  var promiseSpy = jasmine.createSpy();
  d.promise.then(promiseSpy);

  setTimeout(function() {
    expect(promiseSpy).toHaveBeenCalledWith(42);
    done();
  }, 0);
});
```

More generally, the order independence is achieved by the fact that when you call resolve, *Promise callbacks are not invoked immediately*, as we alluded to earlier. This is why we've been using `setTimeout` in our first tests.

test/q_spec.js

```
it('does not resolve promise immediately', function() {
  var d = $q.defer();

  var promiseSpy = jasmine.createSpy();
  d.promise.then(promiseSpy);

  d.resolve(42);

  expect(promiseSpy).not.toHaveBeenCalled();
});
```

So, if Promise callbacks are not called immediately, when are they called? This is where the digest cycle comes in handy. Promise callbacks get called *during the next digest after resolving the Deferred*. To test this, we need a Scope in the test file, so let's get one from the injector:

test/q_spec.js

```
var $q, $rootScope;

beforeEach(function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  $q = injector.get('$q');
  $rootScope = injector.get('$rootScope');
});
```

Now we can test that a Promise callback gets invoked during the digest:

test/q_spec.js

```
it('resolves promise at next digest', function() {
  var d = $q.defer();

  var promiseSpy = jasmine.createSpy();
  d.promise.then(promiseSpy);

  d.resolve(42);
  $rootScope.$apply();

  expect(promiseSpy).toHaveBeenCalledWith(42);
});
```

Instead of calling the Promise callback immediately in `Deferred.resolve`, let's store the resolved value and invoke a helper function whose job is to call the Promise callback later:

src/q.js

```
Deferred.prototype.resolve = function(value) {
  this.promise.$$state.value = value;
  scheduleProcessQueue(this.promise.$$state);
};
```

This helper function taps into the digest with the `$evalAsync` function of the root scope. Recall that this function can be used to attach callbacks that run during the next digest, and that it also schedules a digest to happen with `setTimeout` if one hasn't been scheduled already. Our callback will invoke another helper function:

src/q.js

```
function scheduleProcessQueue(state) {
  $rootScope.$evalAsync(function() {
    processQueue(state);
  });
}
```

In order to invoke `$evalAsync`, we need to get a hold of `$rootScope` in `$QProvider`, so let's inject it to the `$get` method:

src/q.js

```
this.$get = ['$rootScope', function($rootScope) {

  // ...

}];
```

And finally, we can define the `processQueue` function that gets called during the digest. Here we invoke the actual promise callback:

src/q.js

```
function processQueue(state) {
  state.pending(state.value);
}
```

> Most promise libraries, including Q, behave so that when a Deferred is resolved, Promise callbacks are not immediately invoked. Of course, generic Promise libraries usually don't have anything equivalent to the digest cycle, so they just use something like `setTimeout`.
> Since Angular resolves Promises at digest time, it can potentially lead to better-performing code: While your Promise handler gets called asynchronously, if the value is immediately available, it still gets called during the same turn of the JavaScript event loop. We don't relinquish control back to the browser, which would give it a chance to do other work in between, which is what happens with Promise libraries that use `setTimeout`.

Preventing Multiple Resolutions

One important feature of Deferreds is that they only ever get resolved once. When a Deferred has been resolved to a value, it will never be resolved to any other value. Promise callbacks also get invoked at most once. If you try to resolve a Deferred a second time, the call is ignored and no callbacks are invoked:

test/q_spec.js

```
it('may only be resolved once', function() {
  var d = $q.defer();

  var promiseSpy = jasmine.createSpy();
  d.promise.then(promiseSpy);

  d.resolve(42);
  d.resolve(43);

  $rootScope.$apply();

  expect(promiseSpy.calls.count()).toEqual(1);
  expect(promiseSpy).toHaveBeenCalledWith(42);
});
```

This is true even if there's a digest in between two resolve calls. The resolution of a Deferred is permanent:

test/q_spec.js

```
it('may only ever be resolved once', function() {
  var d = $q.defer();

  var promiseSpy = jasmine.createSpy();
  d.promise.then(promiseSpy);

  d.resolve(42);
  $rootScope.$apply();
  expect(promiseSpy).toHaveBeenCalledWith(42);

  d.resolve(43);
  $rootScope.$apply();
  expect(promiseSpy.calls.count()).toEqual(1);
});
```

We control this by adding a `status` flag to the internal `$$state` object of the Promise. Once the Deferred is resolved, the Promises's status is set to `1` (for "resolved"). On then next call we'll notice that it's been set and return immediately:

src/q.js


```
Deferred.prototype.resolve = function(value) {
  if (this.promise.$$state.status) {
    return;
  }
  this.promise.$$state.value = value;
  this.promise.$$state.status = 1;
  scheduleProcessQueue(this.promise.$$state);
};
```

Ensuring that Callbacks Get Invoked

As things stand, we are scheduling the execution of Promise callbacks when a Deferred gets resolved. That works great, but we have a blind spot in one situation: If a Promise callback is registered *after a Deferred was already resolved and a digest was run*, that callback is never invoked:

test/q_spec.js

```
it('resolves a listener added after resolution', function() {
  var d = $q.defer();
  d.resolve(42);
  $rootScope.$apply();

  var promiseSpy = jasmine.createSpy();
  d.promise.then(promiseSpy);
  $rootScope.$apply();

  expect(promiseSpy).toHaveBeenCalledWith(42);
});
```

What we should do is check the status flag at callback registration time. If we're already resolved, we should just schedule the callback invocation:

src/q.js

```
Promise.prototype.then = function(onFulfilled) {
  this.$$state.pending = onFulfilled;
  if (this.$$state.status > 0) {
    scheduleProcessQueue(this.$$state);
  }
};
```


Registering Multiple Promise Callbacks

While a Deferred always has exactly one Promise, a Promise should be able to have any number of callbacks. Our current implementation does not support this:

test/q_spec.js

```
it('may have multiple callbacks', function() {
  var d = $q.defer();

  var firstSpy = jasmine.createSpy();
  var secondSpy = jasmine.createSpy();
  d.promise.then(firstSpy);
  d.promise.then(secondSpy);

  d.resolve(42);
  $rootScope.$apply();

  expect(firstSpy).toHaveBeenCalledWith(42);
  expect(secondSpy).toHaveBeenCalledWith(42);
});
```

The way this works is that whenever a digest runs and the Promise has been resolved, any callbacks that haven't been invoked yet are invoked. Each callback gets called just once, as we already asserted earlier:

test/q_spec.js

```
it('invokes callbacks once', function() {
  var d = $q.defer();

  var firstSpy = jasmine.createSpy();
  var secondSpy = jasmine.createSpy();

  d.promise.then(firstSpy);
  d.resolve(42);
  $rootScope.$apply();
  expect(firstSpy.calls.count()).toBe(1);
  expect(secondSpy.calls.count()).toBe(0);

  d.promise.then(secondSpy);
  expect(firstSpy.calls.count()).toBe(1);
  expect(secondSpy.calls.count()).toBe(0);

  $rootScope.$apply();
  expect(firstSpy.calls.count()).toBe(1);
  expect(secondSpy.calls.count()).toBe(1);
});
```


Instead of storing just one pending callback as we're currently doing, we're going to have to support storing multiple pending callbacks, so we need an array:

src/q.js

```
Promise.prototype.then = function(onFulfilled) {
  this.$$state.pending = this.$$state.pending || [];
  this.$$state.pending.push(onFulfilled);
  if (this.$$state.status > 0) {
    scheduleProcessQueue(this.$$state);
  }
};
```

Then, when we get to invoking the callbacks, we need to loop over them:

src/q.js

```
function processQueue(state) {
  _.forEach(state.pending, function(onFulfilled) {
    onFulfilled(state.value);
  });
}
```

Also, to make sure each callback does in fact only get called once, we need to *clear* the pending callbacks as we invoke them. At each digest, pending callbacks are cleared out. If more callbacks get registered later, the array will get reinitialized.

src/q.js

```
function processQueue(state) {
  var pending = state.pending;
  delete state.pending;
  _.forEach(pending, function(onFulfilled) {
    onFulfilled(state.value);
  });
}
```

> The order in which we do things is significant here. If some of the callbacks happen to register more callbacks, we don't want them to be cleared just yet.

Rejecting Deferreds And Catching Rejections

As discussed in the beginning of the chapter, one of the big difficulties with callback-style programming is the lack of any general error handling mechanism. Errors are passed through arbitrarily chosen callback parameters, and the way that's done varies from library to library and application to application.

Promises solve this issue by having error handling built-in. There are two pieces to that solution, the first of which being the way you can signal to a Deferred that something has gone wrong. This is called *rejecting* the Deferred (and the Promise). The second piece is getting notified that a rejection has occurred. You can do this by providing a second argument to the `then` method of your Promise. That second argument is a callback that gets invoked if the promise gets rejected:

test/q_spec.js

```
it('can reject a deferred', function() {
  var d = $q.defer();

  var fulfillSpy = jasmine.createSpy();
  var rejectSpy  = jasmine.createSpy();
  d.promise.then(fulfillSpy, rejectSpy);

  d.reject('fail');
  $rootScope.$apply();

  expect(fulfillSpy).not.toHaveBeenCalled();
  expect(rejectSpy).toHaveBeenCalledWith('fail');
});
```

A Promise can get rejected at most once:

test/q_spec.js

```
it('can reject just once', function() {
  var d = $q.defer();

  var rejectSpy  = jasmine.createSpy();
  d.promise.then(null, rejectSpy);

  d.reject('fail');
  $rootScope.$apply();
  expect(rejectSpy.calls.count()).toBe(1);

  d.reject('fail again');
  $rootScope.$apply();
  expect(rejectSpy.calls.count()).toBe(1);
});
```


You also cannot resolve something you have already rejected (and vice versa). Effectively, a Promise can only ever have one outcome, which is a resolution or a rejection:

test/q_spec.js

```
it('cannot fulfill a promise once rejected', function() {
  var d = $q.defer();

  var fulfillSpy = jasmine.createSpy();
  var rejectSpy  = jasmine.createSpy();
  d.promise.then(fulfillSpy, rejectSpy);

  d.reject('fail');
  $rootScope.$apply();

  d.resolve('success');
  $rootScope.$apply();

  expect(fulfillSpy).not.toHaveBeenCalled();
});
```

The `reject` method of `Deferred` looks very similar to `resolve`. The primary difference is that the status of the promise is set to `2` for ("rejected") and not `1`:

src/q.js

```
Deferred.prototype.reject = function(reason) {
  if (this.promise.$$state.status) {
    return;
  }
  this.promise.$$state.value = reason;
  this.promise.$$state.status = 2;
  scheduleProcessQueue(this.promise.$$state);
};
```

The `then` method of `Promise` now needs to be able to receive two callbacks - the success callback and the rejection errback. The items in `pending` now become arrays of callbacks. We put the success callback in index `1` of the array and the rejection errback in index `2` of the array. This way the indexes match the Promise status codes we're using:

src/q.js

```
Promise.prototype.then = function(onFulfilled, onRejected) {
  this.$$state.pending = this.$$state.pending || [];
  this.$$state.pending.push([null, onFulfilled, onRejected]);
  if (this.$$state.status > 0) {
    scheduleProcessQueue(this.$$state);
  }
};
```


In `processQueue` we can now access the correct handler using the `status` field, and we don't actually need to do much to cover both the success and failure cases:

src/q.js

```
function processQueue(state) {
  var pending = state.pending;
  delete state.pending;
  _.forEach(pending, function(handlers) {
    var fn = handlers[state.status];
    fn(state.value);
  });
}
```

There is one problem with this implementation, which is that it assumes that each call to `then` will supply both success and failure callbacks. It should actually be possible to omit one or the other (or technically even both):

test/q_spec.js

```
it('does not require a failure handler each time', function() {
  var d = $q.defer();

  var fulfillSpy = jasmine.createSpy();
  var rejectSpy  = jasmine.createSpy();
  d.promise.then(fulfillSpy);
  d.promise.then(null, rejectSpy);

  d.reject('fail');
  $rootScope.$apply();

  expect(rejectSpy).toHaveBeenCalledWith('fail');
});

it('does not require a success handler each time', function() {
  var d = $q.defer();

  var fulfillSpy = jasmine.createSpy();
  var rejectSpy  = jasmine.createSpy();
  d.promise.then(fulfillSpy);
  d.promise.then(null, rejectSpy);

  d.resolve('ok');
  $rootScope.$apply();

  expect(fulfillSpy).toHaveBeenCalledWith('ok');
});
```


The fix for this is to simply guard the callback invocation with a check to see that it is in fact a function:

src/q.js

```
function processQueue(state) {
  var pending = state.pending;
  delete state.pending;
  _.forEach(pending, function(handlers) {
    var fn = handlers[state.status];
    if (_.isFunction(fn)) {
      fn(state.value);
    }
  });
}
```

There is also a shortcut method for the case where you just want to supply a rejection errback. You could do it with `promise.then(null, callback)`, but you can also use a method called `catch`:

test/q_spec.js

```
it('can register rejection handler with catch', function() {
  var d = $q.defer();

  var rejectSpy  = jasmine.createSpy();
  d.promise.catch(rejectSpy);
  d.reject('fail');
  $rootScope.$apply();

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

This method is implemented in terms of `then`. There's nothing special in it - it is just there for convenience:

src/q.js

```
Promise.prototype.catch = function(onRejected) {
  return this.then(null, onRejected);
};
```


Cleaning Up At The End: `finally`

In a traditional, synchronous `try..catch` you can add a `finally` block at the end for executing code that needs to always run *whether the task succeeds or fails*. Promises have a similar construct, and it's available via the `finally` method of `Promise`.

The method takes a callback, in which you can do whatever cleanup work you need to do at the end of your asynchronous task. The callback is invoked when the Promise is resolved, and it will *not* receive any arguments:

test/q_spec.js

```
it('invokes a finally handler when fulfilled', function() {
  var d = $q.defer();

  var finallySpy  = jasmine.createSpy();
  d.promise.finally(finallySpy);
  d.resolve(42);
  $rootScope.$apply();

  expect(finallySpy).toHaveBeenCalledWith();
});
```

> We use `toHaveBeenCalledWith` with zero arguments to check that the finally callback did not receive any arguments.

The `finally` callback also gets invoked if the Promise is rejected:

test/q_spec.js

```
it('invokes a finally handler when rejected', function() {
  var d = $q.defer();

  var finallySpy  = jasmine.createSpy();
  d.promise.finally(finallySpy);
  d.reject('fail');
  $rootScope.$apply();

  expect(finallySpy).toHaveBeenCalledWith();
});
```

We can implement `finally` in terms of `then`, just like we did with `catch`. We'll register both a success and a failure callback, both of which will delegate to the original callback. It will get called no matter what happens:

src/q.js


```
Promise.prototype.finally = function(callback) {
  return this.then(function() {
    callback();
  }, function() {
    callback();
  });
};
```

We'll return to `finally` later in this chapter to make sure it plays well with chained Promises. But before we go there, let's take a look at what chained Promises actually look like.

Promise Chaining

We now have a pretty good understanding of how a Deferrend-Promise pair behaves. This in itself can be a useful thing, but where Promises truly come to their own is when you have several asynchronous tasks to do, and you need to compose a workflow for them. This is where *Promise chaining* comes in.

The simplest form of Promise chaining is just attaching several `then` callbacks back to back. Each callback receives the return value of the previous one as its argument:

test/q_spec.js

```
it('allows chaining handlers', function() {
  var d = $q.defer();

  var fulfilledSpy = jasmine.createSpy();
  d.promise.then(function(result) {
    return result + 1;
  }).then(function(result) {
    return result * 2;
  }).then(fulfilledSpy);

  d.resolve(20);
  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith(42);
});
```

What's happening here is that each call to `then` returns another Promise that can be used for further callbacks. It is important to understand, however, that each time this happens a *new* Promise is created. Other callbacks on the original Promise are not affected:

test/q_spec.js

```
it('does not modify original resolution in chains', function() {
  var d = $q.defer();

  var fulfilledSpy = jasmine.createSpy();

  d.promise.then(function(result) {
    return result + 1;
  }).then(function(result) {
    return result * 2;
  });
  d.promise.then(fulfilledSpy);

  d.resolve(20);
  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith(20);
});
```

What we need to do in `then` is to create a new Deferred - one that represents the computation in the `onFulfilled` callback. We then return its Promise to the caller:

src/q.js

```
Promise.prototype.then = function(onFulfilled, onRejected) {
  var result = new Deferred();
  this.$$state.pending = this.$$state.pending || [];
  this.$$state.pending.push([null, onFulfilled, onRejected]);
  if (this.$$state.status > 0) {
    scheduleProcessQueue(this.$$state);
  }
  return result.promise;
};
```

Then we need to pass this Deferred to where the `onFulfilled` callback actually gets invoked, so that the results can be delivered. We'll set it as the first item of the array in `$$state.pending`:

src/q.js

```
Promise.prototype.then = function(onFulfilled, onRejected) {
  var result = new Deferred();
  this.$$state.pending = this.$$state.pending || [];
  this.$$state.pending.push([result, onFulfilled, onRejected]);
  if (this.$$state.status > 0) {
    scheduleProcessQueue(this.$$state);
  }
  return result.promise;
};
```


As we then invoke the callback in `processQueue`, we also pass its return value to the Deferred:

src/q.js

```
function processQueue(state) {
  var pending = state.pending;
  delete state.pending;
  _.forEach(pending, function(handlers) {
    var deferred = handlers[0];
    var fn = handlers[state.status];
    if (_.isFunction(fn)) {
      deferred.resolve(fn(state.value));
    }
  });
}
```

This is how chained `then` calls work. Each one creates a new Deferred and a new Promise. The new Deferred is independent from the original, but is *resolved* when the original one is resolved.

Note that a `then` call will always create a new Deferred and return a Promise. At some point you'll be at the end of your chain and you don't actually use the last Promise for anything. It will still exist, but it just gets ignored.

Another important aspect of chains is how they transitively pass forward a value until a callback handler is found. For instance, in the following we have a Deferred that we reject. We attach only a success callback to it, but to the next, chained Promise we do add a rejection callback. The rejection is passed through to that second Promise. So you can have a construct like `one.then(two).catch(three)`, and rely on `three` to catch errors from both `one` and `two`:

test/q_spec.js

```
it('catches rejection on chained handler', function() {
  var d = $q.defer();

  var rejectedSpy = jasmine.createSpy();
  d.promise.then(_.noop).catch(rejectedSpy);

  d.reject('fail');
  $rootScope.$apply();

  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```


This happens not only with rejections, but also with resolutions. When a Promise only has an error handler, its resolution is passed to the next Promise in the chain, whose success callbacks will get the resolved value:

test/q_spec.js

```
it('fulfills on chained handler', function() {
  var d = $q.defer();

  var fulfilledSpy = jasmine.createSpy();
  d.promise.catch(_.noop).then(fulfilledSpy);

  d.resolve(42);
  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith(42);
});
```

All of this is made possible with a simple modification to the `processQueue` function. For each handler, if there is no callback for the status we're in, we just resolve or reject the chained Deferred with the current Promise's value:

src/q.js

```
function processQueue(state) {
  var pending = state.pending;
  delete state.pending;
  _.forEach(pending, function(handlers) {
    var deferred = handlers[0];
    var fn = handlers[state.status];
    if (_.isFunction(fn)) {
      deferred.resolve(fn(state.value));
    } else if (state.status === 1) {
      deferred.resolve(state.value);
    } else {
      deferred.reject(state.value);
    }
  });
}
```

Another point about chaining is that when there is a rejection, the next `catch` handler will *actually catch* it, and the `catch` handler's own return value will be treated as a *resolution*, not a rejection. Our implementation already behaves correctly in this manner, but this may not be obvious. The trick is that we *always* call `d.resolve` in `processQueue` when we have a callback function for the current state, regardless of whether it's a resolution or a rejection.

This should make more sense if you think of it as the equivalent of a traditional, synchronous `catch` block. The `catch` handles the error and it is no longer propagated. Normal execution resumes.

test/q_spec.js


```
it('treats catch return value as resolution', function() {
  var d = $q.defer();

  var fulfilledSpy = jasmine.createSpy();
  d.promise
    .catch(function() {
      return 42;
    })
    .then(fulfilledSpy);

  d.reject('fail');
  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith(42);
});
```

Exception Handling

Explicitly rejecting a Promise is one thing, but something may also just go wrong with your code when executing a Promise callback. When an exception is thrown from a Promise callback, this should cause the next rejection handler in the chain to be invoked. It should receive the thrown exception:

test/q_spec.js

```
it('rejects chained promise when handler throws', function() {
  var d = $q.defer();

  var rejectedSpy = jasmine.createSpy();
  d.promise.then(function() {
    throw 'fail';
  }).catch(rejectedSpy);
  d.resolve(42);

  $rootScope.$apply();

  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```

The propagation of the exception happens in `processQueue`, where we invoke our handlers. If an exception is thrown, it is caught and the next Deferred in the chain is rejected with the exception:

src/q.js


```
function processQueue(state) {
  var pending = state.pending;
  delete state.pending;
  _.forEach(pending, function(handlers) {
    var deferred = handlers[0];
    var fn = handlers[state.status];
    try {
      if (_.isFunction(fn)) {
        deferred.resolve(fn(state.value));
      } else if (state.status === 1) {
        deferred.resolve(state.value);
      } else {
        deferred.reject(state.value);
      }
    } catch (e) {
      deferred.reject(e);
    }
  });
}
```

It is important to notice that an exception does not reject the Deferred whose Promise handler it is thrown from, but *the next one in the chain*. If we are executing a Promise handler, the respective Deferred must already have been resolved (or rejected), and we don't go back and change it. This means that the following test case passes, as it should:

test/q_spec.js

```
it('does not reject current promise when handler throws', function() {
  var d = $q.defer();

  var rejectedSpy = jasmine.createSpy();
  d.promise.then(function() {
    throw 'fail';
  });
  d.promise.catch(rejectedSpy);
  d.resolve(42);

  $rootScope.$apply();

  expect(rejectedSpy).not.toHaveBeenCalled();
});
```

The difference between this test case and the previous one is that here we set the rejection handler on the original Promise instead of chaining it on the success handler. This is an important distinction.

Callbacks Returning Promises

So, when a Promise callback returns a value, that value becomes the resolution of the next Promise in the chain. And when a Promise callback throws an exception, that exception becomes the rejection of the next Promise in the chain. In both of these cases, the value or exception from the callback comes *synchronously*. But what if you actually want to do some more *asynchronous* work in Promise callbacks? In other words, what if a Promise callback returns *another* Promise?

The answer is that we should connect the Promise returned by the callback to the next callback in the chain:

test/q_spec.js

```
it('waits on promise returned from handler', function() {
  var d = $q.defer();
  var fulfilledSpy = jasmine.createSpy();

  d.promise.then(function(v) {
    var d2 = $q.defer();
    d2.resolve(v + 1);
    return d2.promise;
  }).then(function(v) {
    return v * 2;
  }).then(fulfilledSpy);
  d.resolve(20);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith(42);
});
```

Here we have three Promise callbacks in a chain. The first of them goes off to do some "asynchronous" work itself, and only when it is resolved do we proceed to the second callback in the chain.

Another way to combine asynchronous workflows is to resolve one Promise with another Promise. In this case one Promise's resolution becomes the resolution of the other:

test/q_spec.js

```
it('waits on promise given to resolve', function() {
  var d = $q.defer();
  var d2 = $q.defer();
  var fulfilledSpy = jasmine.createSpy();

  d.promise.then(fulfilledSpy);
  d2.resolve(42);
  d.resolve(d2.promise);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith(42);
});
```


Something similar also happens with rejections. When there's an inner Promise and it is rejected, that rejection is propagated on the chain:

test/q_spec.js

```
it('rejects when promise returned from handler rejects', function() {
  var d = $q.defer();
  var rejectedSpy = jasmine.createSpy();
  d.promise.then(function() {
    var d2 = $q.defer();
    d2.reject('fail');
    return d2.promise;
  }).catch(rejectedSpy);
  d.resolve('ok');

  $rootScope.$apply();

  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```

What all of this effectively means is that a Deferred may be resolved with another Promise, and in that case the Deferred's resolution is dependent on the resolution of the other Promise. So, when we are given a value in `resolve`, we must check if that value itself looks like a Promise. If we decide it does, we do not resolve immediately, but instead attach callbacks to it, that will later call our own `resolve` (or `reject`) again:

src/q.js

```
Deferred.prototype.resolve = function(value) {
  if (this.promise.$$state.status) {
    return;
  }
  if (value && _.isFunction(value.then)) {
    value.then(
      _.bind(this.resolve, this),
      _.bind(this.reject, this)
    );
  } else {
    this.promise.$$state.value = value;
    this.promise.$$state.status = 1;
    scheduleProcessQueue(this.promise.$$state);
  }
};
```


Now, our `resolve` (or `reject`) method will later get invoked again when that Promise resolves. This actually makes all our test cases pass, since all of them go through `resolve`.

> Notice that we didn't add a strict type check for `value` in the implementation above. We don't require that it's actually an instance of `Promise`, but just something that has a `then` method (sometimes called a "thenable" object). This means that the nested Promise may actually be from some other Promise implementation than Angular's, which can be useful when integrating other libraries to your app.

Chaining Handlers on `finally`

When we implemented `finally` earlier, we discussed that we were going to come back to it when we have Promise chaining support. Now that we do, let's see how Promise chaining and `finally` work together.

An important aspect of `finally` is that *its return value should be ignored*. Finally is only meant for cleaning up resources and it does not participate in the formation of a Promise chain's eventual value. That means that any value a Promise chain has been resolved to should flow through intermediate `finally` handlers untouched:

test/q_spec.js

```
it('allows chaining handlers on finally, with original value', function() {
  var d = $q.defer();

  var fulfilledSpy = jasmine.createSpy();
  d.promise.then(function(result) {
    return result + 1;
  }).finally(function(result) {
    return result * 2;
  }).then(fulfilledSpy);
  d.resolve(20);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith(21);
});
```

The same exact behavior is needed for rejections. When a Promise is rejected, the rejection flows through any finally handlers to the next rejection handler in the chain:

test/q_spec.js

```
it('allows chaining handlers on finally, with original rejection', function() {
  var d = $q.defer();

  var rejectedSpy = jasmine.createSpy();
```



```
  d.promise.then(function(result) {
    throw 'fail';
  }).finally(function() {
  }).catch(rejectedSpy);
  d.resolve(20);

  $rootScope.$apply();

  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```

So, in the `then` handler that we set up for `finally`, we should just return the value given to us, so that the next handler in the chain receives it. Whatever our own callback may return is ignored:

src/q.js

```
Promise.prototype.finally = function(callback) {
  return this.then(function(value) {
    callback();
    return value;
  }, function() {
    callback();
  });
};
```

To do the same for the rejection, we can't simply return it because that would propagate to a *success* handler. We need to create a new rejected Promise for the same rejection:

src/q.js

```
Promise.prototype.finally = function(callback) {
  return this.then(function(value) {
    callback();
    return value;
  }, function(rejection) {
    callback();
    var d = new Deferred();
    d.reject(rejection);
    return d.promise;
  });
};
```

Since the resource cleanup done in a `finally` handler may be asynchronous itself, we should also support `finally` handlers that return Promise. When a Promise is returned from a `finally` handler, we should wait for it to resolve before continuing with the chain. Howerver, we should *still ignore its resolved value* and just use the original one:

test/q_spec.js


```
it('resolves to orig value when nested promise resolves', function() {
  var d = $q.defer();

  var fulfilledSpy = jasmine.createSpy();
  var resolveNested;

  d.promise.then(function(result) {
    return result + 1;
  }).finally(function(result) {
    var d2 = $q.defer();
    resolveNested = function() {
      d2.resolve('abc');
    };
    return d2.promise;
  }).then(fulfilledSpy);
  d.resolve(20);

  $rootScope.$apply();
  expect(fulfilledSpy).not.toHaveBeenCalled();

  resolveNested();
  $rootScope.$apply();
  expect(fulfilledSpy).toHaveBeenCalledWith(21);
});
```

Here we test that when there's some async work done in a finally handler, the chain does not resolve immediately, but only when that async work is resolved. The value of that async work is ignored.

Again, we need to provide the same behavior for rejections: If a Promise is rejected, any asynchronous `finally` handlers between it and the next rejection handler are resolved first:

test/q_spec.js

```
it('rejects to original value when nested promise resolves', function() {
  var d = $q.defer();

  var rejectedSpy = jasmine.createSpy();
  var resolveNested;

  d.promise.then(function(result) {
    throw 'fail';
  }).finally(function(result) {
    var d2 = $q.defer();
    resolveNested = function() {
      d2.resolve('abc');
    };
    return d2.promise;
```

```
  }).catch(rejectedSpy);
  d.resolve(20);

  $rootScope.$apply();
  expect(rejectedSpy).not.toHaveBeenCalled();

  resolveNested();
  $rootScope.$apply();
  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```

There is one case in which the result of a `finally` handler is *not* ignored, and that is when it *itself* is rejected. When that happens, the rejection takes over from any previous value in the chain. We don't want failures in resource cleanup to go unnoticed:

test/q_spec.js

```
it('rejects when nested promise rejects in finally', function() {
  var d = $q.defer();

  var fulfilledSpy = jasmine.createSpy();
  var rejectedSpy = jasmine.createSpy();
  var rejectNested;

  d.promise.then(function(result) {
    return result + 1;
  }).finally(function(result) {
    var d2 = $q.defer();
    rejectNested = function() {
      d2.reject('fail');
    };
    return d2.promise;
  }).then(fulfilledSpy, rejectedSpy);
  d.resolve(20);

  $rootScope.$apply();
  expect(fulfilledSpy).not.toHaveBeenCalled();

  rejectNested();
  $rootScope.$apply();
  expect(fulfilledSpy).not.toHaveBeenCalled();
  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```

So, once we get to a `finally` handler, we should check if the value we get from it looks like a Promise. If it does, we chain a handler on it. The chained handler always resolves to the original value, but it does not have a rejection callback, which means that it will propagate any rejections from the `finally` handler:

src/q.js


```
Promise.prototype.finally = function(callback) {
  return this.then(function(value) {
    var callbackValue = callback();
    if (callbackValue && callbackValue.then) {
      return callbackValue.then(function() {
        return value;
      });
    } else {
      return value;
    }
  }, function(rejection) {
    callback();
    var d = new Deferred();
    d.reject(rejection);
    return d.promise;
  });
};
```

We should do a similar trick if we were originally passed a rejection: If the `finally` callback returned a Promise, wait for it and then send the rejection forward. In this branch the end result is always a rejected Promise - either one with the original rejection or one with a rejection from `finally`:

src/q.js

```
Promise.prototype.finally = function(callback) {
  return this.then(function(value) {
    var callbackValue = callback();
    if (callbackValue && callbackValue.then) {
      return callbackValue.then(function() {
        return value;
      });
    } else {
      return value;
    }
  }, function(rejection) {
    var callbackValue = callback();
    if (callbackValue && callbackValue.then) {
      return callbackValue.then(function() {
        var d = new Deferred();
        d.reject(rejection);
        return d.promise;
      });
    } else {
      var d = new Deferred();
      d.reject(rejection);
      return d.promise;
    }
  });
};
```


Now we have a fully functional implementation of `finally`, but it is a bit on the verbose side. Let's add a couple of helper functions to break it down a bit.

First, we could use a general helper function that takes a value and a boolean flag, and returns a Promise that either resolves or rejects with the value, based on the boolean flag:

src/q.js

```
function makePromise(value, resolved) {
  var d = new Deferred();
  if (resolved) {
    d.resolve(value);
  } else {
    d.reject(value);
  }
  return d.promise;
}
```

We can now use this new helper function in a few places in `finally`:

src/q.js

```
Promise.prototype.finally = function(callback) {
  return this.then(function(value) {
    var callbackValue = callback();
    if (callbackValue && callbackValue.then) {
      return callbackValue.then(function() {
        return makePromise(value, true);
      });
    } else {
      return value;
    }
  }, function(rejection) {
    var callbackValue = callback();
    if (callbackValue && callbackValue.then) {
      return callbackValue.then(function() {
        return makePromise(rejection, false);
      });
    } else {
      return makePromise(rejection, false);
    }
  });
};
```

Another thing we can do is make a general handler for the finally callback, which we can use in both the resolution and rejection branches since they are quite similar:

src/q.js

```
Promise.prototype.finally = function(callback) {
  return this.then(function(value) {
    return handleFinallyCallback(callback, value, true);
  }, function(rejection) {
    return handleFinallyCallback(callback, rejection, false);
  });
};
```

The helper function invokes the `finally` callback, and then returns a Promise for the original resolution or rejection. That is, unless the `finally` callback itself rejects, in which case the returned Promise returned always rejects:

src/q.js

```
function handleFinallyCallback(callback, value, resolved) {
  var callbackValue = callback();
  if (callbackValue && callbackValue.then) {
    return callbackValue.then(function() {
      return makePromise(value, resolved);
    });
  } else {
    return makePromise(value, resolved);
  }
}
```

So, in summary, whenever a `finally` returns a Promise, we wait for it to become resolved before continuing. We ignore that Promise's resolution in favor of the original one, except when it rejects, in which case we pass the rejection forward in the chain.

Notifying Progress

Sometimes, when you are performing asynchronous work, you have some information available of what kind of progress you've made or how much work there's left. While you're not ready to resolve the Promise yet, you may want to send information downstream about what's going on.

Angular's `$q` has a built-in feature for sending this kind of information: The `notify` method on `Deferred`. You can invoke it with some value, and that gets passed to anyone listening. You can register to receive progress information by supplying a *third* callback argument to `then`:

test/q_spec.js


```
it('can report progress', function() {
  var d = $q.defer();
  var progressSpy = jasmine.createSpy();
  d.promise.then(null, null, progressSpy);

  d.notify('working...');
  $rootScope.$apply();

  expect(progressSpy).toHaveBeenCalledWith('working...');
});
```

The big difference between `notify` on the one hand and `resolve` and `reject` on the other is that you can call `notify` several times, and your callback may also get invoked several times. This is not the case with `resolve` and `reject` because a Promise never resolves or rejects more than once.

test/q_spec.js

```
it('can report progress many times', function() {
  var d = $q.defer();
  var progressSpy = jasmine.createSpy();
  d.promise.then(null, null, progressSpy);

  d.notify('40%');
  $rootScope.$apply();

  d.notify('80%');
  d.notify('100%');
  $rootScope.$apply();

  expect(progressSpy.calls.count()).toBe(3);
});
```

How we can make this work is by first grabbing the progress callback in `then`. We'll just add it as the fourth item in the pending array:

src/q.js

```
Promise.prototype.then = function(onFulfilled, onRejected, onProgress) {
  var result = new Deferred();
  this.$$state.pending = this.$$state.pending || [];
  this.$$state.pending.push([result, onFulfilled, onRejected, onProgress]);
  if (this.$$state.status > 0) {
    scheduleProcessQueue(this.$$state);
  }
  return result.promise;
};
```

Now we can implement the `notify` method. It iterates over all the pending handlers and invokes their progress callbacks. To comply with the general contract of promises, the callbacks don't get called *immediately* when `notify` is called, but asynchronously a bit later. To achieve this we again use the `$evalAsync` function from the `$rootScope`:

src/q.js

```
Deferred.prototype.notify = function(progress) {
  var pending = this.promise.$$state.pending;
  if (pending && pending.length) {
    $rootScope.$evalAsync(function() {
      _.forEach(pending, function(handlers) {
        var progressBack = handlers[3];
        if (_.isFunction(progressBack)) {
          progressBack(progress);
        }
      });
    });
  }
};
```

The disposable nature of Promises does come into effect once we consider what happens *after* Promise resolution. At that point, `notify` should *not* cause progress callbacks to be invoked anymore:

test/q_spec.js

```
it('does not notify progress after being resolved', function() {
  var d = $q.defer();
  var progressSpy = jasmine.createSpy();
  d.promise.then(null, null, progressSpy);

  d.resolve('ok');
  d.notify('working...');
  $rootScope.$apply();

  expect(progressSpy).not.toHaveBeenCalled();
});
```

The same is true after rejection - `notify` does nothing at that point:

test/q_spec.js

```
it('does not notify progress after being rejected', function() {
  var d = $q.defer();
  var progressSpy = jasmine.createSpy();
  d.promise.then(null, null, progressSpy);

  d.reject('fail');
  d.notify('working...');
  $rootScope.$apply();

  expect(progressSpy).not.toHaveBeenCalled();
});
```


We can satisfy these test cases by adding an additional check to `notify` for the status of the associated promise. If there's any kind of truthy value in it, the notification is skipped:

src/q.js

```
Deferred.prototype.notify = function(progress) {
  var pending = this.promise.$$state.pending;
  if (pending && pending.length && !this.promise.$$state.status) {
    $rootScope.$evalAsync(function() {
      _.forEach(pending, function(handlers) {
        var progressBack = handlers[3];
        if (_.isFunction(progressBack)) {
          progressBack(progress);
        }
      });
    });
  }
};
```

When it comes to Promise chains, the notification system has some interesting properties. First of all, notifications are propagated through chains. When you `notify` on one Deferred, any chained Promises get the notification too, so you can do something like this:

test/q_spec.js

```
it('can notify progress through chain', function() {
  var d = $q.defer();
  var progressSpy = jasmine.createSpy();

  d.promise
    .then(_.noop)
    .catch(_.noop)
    .then(null, null, progressSpy);

  d.notify('working...');
  $rootScope.$apply();

  expect(progressSpy).toHaveBeenCalledWith('working...');
});
```

To implement this, we can make use of the fact that for each item in the array of pending handlers, the chained `Deferred` will be stored as the first item. From `notify` we just call `notify` on the chained `Deferred`:

src/q.js


```
Deferred.prototype.notify = function(progress) {
  var pending = this.promise.$$state.pending;
  if (pending && pending.length &&
      !this.promise.$$state.status) {
    $rootScope.$evalAsync(function() {
      _.forEach(pending, function(handlers) {
        var deferred = handlers[0];
        var progressBack = handlers[3];
        if (_.isFunction(progressBack)) {
          progressBack(progress);
        }
        deferred.notify(progress);
      });
    });
  }
};
```

Where it gets more interesting is when you actually have several progress callbacks along the chain. What happens then is the return value of each progress handler becomes the notification for the next. You can effectively *transform* the progress information as it passes through the chain:

test/q_spec.js

```
it('transforms progress through handlers', function() {
  var d = $q.defer();
  var progressSpy = jasmine.createSpy();

  d.promise
    .then(_.noop)
    .then(null, null, function(progress) {
      return '***' + progress + '***';
    })
    .catch(_.noop)
    .then(null, null, progressSpy);

  d.notify('working...');
  $rootScope.$apply();

  expect(progressSpy).toHaveBeenCalledWith('***working...***');
});
```

We can do this by simply notifying the next Deferred with the return value of the current progress callback, *if* there is one:

src/q.js

```
Deferred.prototype.notify = function(progress) {
  var pending = this.promise.$$state.pending;
  if (pending && pending.length &&
      !this.promise.$$state.status) {
    $rootScope.$evalAsync(function() {
      _.forEach(pending, function(handlers) {
        var deferred = handlers[0];
        var progressBack = handlers[3];
        deferred.notify(_.isFunction(progressBack) ?
                        progressBack(progress) :
                        progress
        );
      });
    });
  }
};
```

When one of the progress callbacks happens to throw an exception, we should not let that interfere with other callbacks:

test/q_spec.js

```
it('recovers from progressback exceptions', function() {
  var d = $q.defer();
  var progressSpy  = jasmine.createSpy();
  var fulfilledSpy = jasmine.createSpy();

  d.promise.then(null, null, function(progress) {
    throw 'fail';
  });
  d.promise.then(fulfilledSpy, null, progressSpy);

  d.notify('working...');
  d.resolve('ok');
  $rootScope.$apply();

  expect(progressSpy).toHaveBeenCalledWith('working...');
  expect(fulfilledSpy).toHaveBeenCalledWith('ok');
});
```

So we wrap the invocation of each promise callback into a `try..catch`. If an exception occurs, we don't do anything special apart from logging the error out:

src/q.js

```
Deferred.prototype.notify = function(progress) {
  var pending = this.promise.$$state.pending;
  if (pending && pending.length &&
      !this.promise.$$state.status) {
    $rootScope.$evalAsync(function() {
      _.forEach(pending, function(handlers) {
        var deferred = handlers[0];
        var progressBack = handlers[3];
        try {
          deferred.notify(_.isFunction(progressBack) ?
                          progressBack(progress) :
                          progress
          );
        } catch (e) {
          console.log(e);
        }
      });
    });
  }
};
```

> When one of these exceptions occurs, the notification stops propagating in the chain, since we never call `notify` in the next Deferred. Other notification callbacks on the *same* Promise are still invoked, which is what our test verifies.

Notifications also work over asynchronous Promise handlers. When one Promise is waiting for another one to resolve, and the Promise we're waiting for sends out a notification, the notification is propagated to chained callbacks:

test/q_spec.js

```
it('can notify progress through promise returned from handler', function() {
  var d = $q.defer();

  var progressSpy = jasmine.createSpy();
  d.promise.then(null, null, progressSpy);

  var d2 = $q.defer();
  // Resolve original with nested promise
  d.resolve(d2.promise);
  // Notify on the nested promise
  d2.notify('working...');

  $rootScope.$apply();

  expect(progressSpy).toHaveBeenCalledWith('working...');
});
```

All we need to do for this one is to bind our `notify` method as a progress callback to the Promise, like we are already doing with `resolve` and `reject`:

src/q.js

```
Deferred.prototype.resolve = function(value) {
  if (this.promise.$$state.status) {
    return;
  }
  if (value && _.isFunction(value.then)) {
    value.then(
      _.bind(this.resolve, this),
      _.bind(this.reject, this),
      _.bind(this.notify, this)
    );
  } else {
    this.promise.$$state.value = value;
    this.promise.$$state.status = 1;
    scheduleProcessQueue(this.promise.$$state);
  }
};
```

And finally, you can also attach a progress callback when using `finally`. You give it as the second argument, after the finally handler itself:

test/q_spec.js

```
it('allows attaching progressback in finally', function() {
  var d = $q.defer();
  var progressSpy = jasmine.createSpy();
  d.promise.finally(null, progressSpy);

  d.notify('working...');
  $rootScope.$apply();

  expect(progressSpy).toHaveBeenCalledWith('working...');
});
```

The `finally` implementation can just pass this argument through to `then`:

src/q.js

```
Promise.prototype.finally = function(callback, progressBack) {
  return this.then(function(value) {
    return handleFinallyCallback(callback, value, true);
  }, function(rejection) {
    return handleFinallyCallback(callback, rejection, false);
  }, progressBack);
};
```


Immediate Rejection - $q.reject

Sometimes, when writing a function that's expected to return a Promise, you know right away that things are not going to work out. In this case you want to return a rejected Promise without actually doing anything asynchronously. This can certainly be done with our existing API:

```
var d = $q.defer();
d.reject('fail');
return d.promise;
```

This is a fairly verbose way to do it though, and as a pattern this is common enough that $q provides a helper method that does the same in a more succinct way. It's called `reject`:

test/q_spec.js

```
it('can make an immediately rejected promise', function() {
  var fulfilledSpy = jasmine.createSpy();
  var rejectedSpy  = jasmine.createSpy();

  var promise = $q.reject('fail');
  promise.then(fulfilledSpy, rejectedSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).not.toHaveBeenCalled();
  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```

The implementation of `reject` does exactly what our example code did above:

src/q.js

```
function reject(rejection) {
  var d = defer();
  d.reject(rejection);
  return d.promise;
}
```

And now we just need to expose `reject` as part of `$q`'s public API:

src/q.js

```
return {
  defer: defer,
  reject: reject
};
```


Immediate Resolution - $q.when

Whereas `$q.reject` lets you easily create a rejected Promise, sometimes you need to create an immediately *resolved* Promise. This is quite common in, for example, caching functions that may or may not need to do something asynchronous but should predictably return a Promise in either case. For this purpose, `$q.when` provides the mirror image of `$q.reject`:

test/q_spec.js

```
it('can make an immediately resolved promise', function() {
  var fulfilledSpy = jasmine.createSpy();
  var rejectedSpy  = jasmine.createSpy();

  var promise = $q.when('ok');
  promise.then(fulfilledSpy, rejectedSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith('ok');
  expect(rejectedSpy).not.toHaveBeenCalled();
});
```

That's not all that `$q.when` can do, however. It can also adopt another promise-like object and turn it into a native Angular Promise. Here we have an ad-hoc "Promise" implementation - an object with a `then` method. After giving it to `$q.when` we can treat it as any `$q` promise, which is what it is at that point:

test/q_spec.js

```
it('can wrap a foreign promise', function() {
  var fulfilledSpy = jasmine.createSpy();
  var rejectedSpy  = jasmine.createSpy();

  var promise = $q.when({
    then: function(handler) {
      $rootScope.$evalAsync(function() {
        handler('ok');
      });
    }
  });
  promise.then(fulfilledSpy, rejectedSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith('ok');
  expect(rejectedSpy).not.toHaveBeenCalled();
});
```


Here's how we can implement `when`:

src/q.js

```
function when(value) {
  var d = defer();
  d.resolve(value);
  return d.promise;
}
```

And here's its addition to `$q`'s public API:

src/q.js

```
return {
  defer: defer,
  reject: reject,
  when: when
};
```

Notice that we didn't really have to do anything to make the adoption of foreign Promises work. That's because our `then` implementation already knows how to wrap a foreign Promise, and `when` merely wraps `then`. However, wrapping foreign Promises with `$q.when` is such a common pattern that we highlight it here.

An additional trick that `$q.when` has up its sleeve is that it can take the three supported promise callbacks - resolved, rejected, and notify - directly as additional arguments. You can choose to give those directly to `when`, instead of doing an additional `then` invocation on the Promise that it returns:

test/q_spec.js

```
it('takes callbacks directly when wrapping', function() {
  var fulfilledSpy = jasmine.createSpy();
  var rejectedSpy  = jasmine.createSpy();
  var progressSpy  = jasmine.createSpy();

  var wrapped = $q.defer();
  $q.when(
    wrapped.promise,
    fulfilledSpy,
    rejectedSpy,
    progressSpy
  );

  wrapped.notify('working...');
  wrapped.resolve('ok');
  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith('ok');
  expect(rejectedSpy).not.toHaveBeenCalled();
  expect(progressSpy).toHaveBeenCalledWith('working...');
});
```


This can be implemented with chaining. We'll add a `then` handler to the promise right in `when`, and return its chained promise to the caller:

src/q.js

```
function when(value, callback, errback, progressback) {
  var d = defer();
  d.resolve(value);
  return d.promise.then(callback, errback, progressback);
}
```

The same exact functionality provided by `$q.when` is also made available under the name `$q.resolve`:

test/q_spec.js

```
it('makes an immediately resolved promise with resolve', function() {
  var fulfilledSpy = jasmine.createSpy();
  var rejectedSpy  = jasmine.createSpy();

  var promise = $q.resolve('ok');
  promise.then(fulfilledSpy, rejectedSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith('ok');
  expect(rejectedSpy).not.toHaveBeenCalled();
});
```

> This alias is added because ES6 Promises have a Promise.resolve method with a similar purpose.

We can just make the `when` implementation available through `resolve` as well:

src/q.js

```
return {
  defer: defer,
  reject: reject,
  when: when,
  resolve: when
};
```


Working with Promise Collections - $q.all

When you have multiple asynchronous tasks to do, treating them as collections of Promises can be really useful. Since Promises are just regular JavaScript objects, you can easily do that with any functions and libraries that produce, consume, and transform collections.

However, sometimes it's useful to have collection processing methods that are Promise-aware, and that can combine and process the asynchronous operations in them. There are libraries that specialize in this, but AngularJS also ships with one particular method that deals with Promise collections: `$q.all`.

The `$q.all` method takes a collection of Promises as its input. It returns a single Promise that resolves to an array of results. The resulting array has an item for each of the Promises in the argument array. This makes `$q.all` a very useful tool for waiting on a number of simultaneous asynchronous tasks to finish:

test/q_spec.js

```
describe('all', function() {

  it('can resolve an array of promises to array of results', function() {
    var promise = $q.all([$q.when(1), $q.when(2), $q.when(3)]);
    var fulfilledSpy = jasmine.createSpy();
    promise.then(fulfilledSpy);

    $rootScope.$apply();

    expect(fulfilledSpy).toHaveBeenCalledWith([1, 2, 3]);
  });

});
```

This is a new function that we'll expose as a public method of `$q`:

src/q.js

```
function all(promises) {

}

return {
  defer: defer,
  reject: reject,
  when: when,
  resolve: when,
  all: all
};
```


The function creates an array of results, and adds a callback to each of the given Promises. Each callback populates the result array in the corresponding index:

src/q.js

```
function all(promises) {
  var results = [];
  _.forEach(promises, function(promise, index) {
    promise.then(function(value) {
      results[index] = value;
    });
  });
}
```

The function also maintains an integer counter that's incremented each time a Promise callback is added, and decremented each time a callback is invoked:

src/q.js

```
function all(promises) {
  var results = [];
  var counter = 0;
  _.forEach(promises, function(promise, index) {
    counter++;
    promise.then(function(value) {
      results[index] = value;
      counter--;
    });
  });
}
```

What this means is that when the counter reaches zero in one of the callback handlers, all of the Promises will have been resolved. If we just construct a Deferred that is resolved to the result array at that point, we'll have a working `$q.all` implementation:

src/q.js

```
function all(promises) {
  var results = [];
  var counter = 0;
  var d = defer();
  _.forEach(promises, function(promise, index) {
    counter++;
    promise.then(function(value) {
      results[index] = value;
      counter--;
      if (!counter) {
        d.resolve(results);
      }
    });
  });
  return d.promise;
}
```


`$q.all` works not only with arrays, but also with objects. If you give it an object where values are Promises, it returns a Promise that resolves to an object with the same keys. In the result object the values are the resolutions of the original Promises:

test/q_spec.js

```
it('can resolve an object of promises to an object of results', function() {
  var promise = $q.all({a: $q.when(1), b: $q.when(2)});
  var fulfilledSpy = jasmine.createSpy();
  promise.then(fulfilledSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith({a: 1, b: 2});
});
```

When initializing the results collection, we should check if the incoming collection is an array or an object, and make the results collection match the type. The rest of our code will just work, since `_.forEach` works uniformly for arrays and objects:

src/q.js

```
function all(promises) {
  var results = _.isArray(promises) ? [] : {};
  var counter = 0;
  var d = defer();
  _.forEach(promises, function(promise, index) {
    counter++;
    promise.then(function(value) {
      results[index] = value;
      counter--;
      if (!counter) {
        d.resolve(results);
      }
    });
  });
  return d.promise;
}
```

We do have one problem with our current implementation of `$q.all`, which is that if the incoming array happens to be empty, the resulting Promise never resolves. A user might expect it to resolve to an empty array instead:

test/q_spec.js

```
it('resolves an empty array of promises immediately', function() {
  var promise = $q.all([]);
  var fulfilledSpy = jasmine.createSpy();
  promise.then(fulfilledSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith([]);
});
```

The same holds for empty objects. If we give one to `$q.all`, we never hear anything back:

test/q_spec.js

```
it('resolves an empty object of promises immediately', function() {
  var promise = $q.all({});
  var fulfilledSpy = jasmine.createSpy();
  promise.then(fulfilledSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith({});
});
```

So empty collections need some special care. What we can do is add an additional check right after looping over the collection of Promises: If the counter is already zero there was nothing to do, and we can resolve immediately.

src/q.js

```
function all(promises) {
  var results = _.isArray(promises) ? [] : {};
  var counter = 0;
  var d = defer();
  _.forEach(promises, function(promise, index) {
    counter++;
    promise.then(function(value) {
      results[index] = value;
      counter--;
      if (!counter) {
        d.resolve(results);
      }
    });
  });
  if (!counter) {
    d.resolve(results);
  }
  return d.promise;
}
```


As we have seen, not everything always goes as we plan it, and Promises may be rejected instead of being resolved. What should `$q.all` do when one or more of the Promises given to it are rejected?

What it does is reject the returned Promise. You effectively lose the results of all the Promises if one of them rejects:

test/q_spec.js

```
it('rejects when any of the promises rejects', function() {
  var promise = $q.all([$q.when(1), $q.when(2), $q.reject('fail')]);
  var fulfilledSpy = jasmine.createSpy();
  var rejectedSpy  = jasmine.createSpy();
  promise.then(fulfilledSpy, rejectedSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).not.toHaveBeenCalled();
  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```

We can do this by supplying a failure callback to each of our Promises. The moment one of those failure callbacks gets invoked, we immediately reject our Deferred. Any further incoming results will simply be ignored:

src/q.js

```
function all(promises) {
  var results = _.isArray(promises) ? [] : {};
  var counter = 0;
  var d = defer();
  _.forEach(promises, function(promise, index) {
    counter++;
    promise.then(function(value) {
      results[index] = value;
      counter--;
      if (!counter) {
        d.resolve(results);
      }
    }, function(rejection) {
      d.reject(rejection);
    });
  });
  if (!counter) {
    d.resolve(results);
  }
  return d.promise;
}
```


We're almost done with `$q.all`, but there's just one more little feature it has: Not every item in the collection given to `$q.all` actually has to be a Promise. Some or even all of the values in the collection may be just plain values, and they'll end up in the resulting collection untouched:

test/q_spec.js

```
it('wraps non-promises in the input collection', function() {
  var promise = $q.all([$q.when(1), 2, 3]);
  var fulfilledSpy = jasmine.createSpy();
  promise.then(fulfilledSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith([1, 2, 3]);
});
```

We can implement this by simply passing each value to `when` before attaching the callback to it. Recall that `when` can take either a plain value or a promise-like object, and always returns a Promise:

src/q.js

```
function all(promises) {
  var results = _.isArray(promises) ? [] : {};
  var counter = 0;
  var d = defer();
  _.forEach(promises, function(promise, index) {
    counter++;
    when(promise).then(function(value) {
      results[index] = value;
      counter--;
      if (!counter) {
        d.resolve(results);
      }
    }, function(rejection) {
      d.reject(rejection);
    });
  });
  if (!counter) {
    d.resolve(results);
  }
  return d.promise;
}
```

And that's `$q.all`! It's not only useful in itself, but also serves as an example of how easy it is to compose Promises. Other similar Promise collection utility methods, like `filter` or `reduce`, could easily be built in a similar way.

ES6-Style Promises

As discussed at the beginning of the chapter, ECMAScript 6 - the next major version of the JavaScript language - comes with a built-in Promise implementation. Our Promise implementation in `$q` already interoperates with the standard implementation nicely, because it can chain and compose any Promise-like objects that have a `then` method, and ES6 Promises have one.

If you want to go a little bit further and use a style of creating Promises that's closer to the standard implementation, `$q` supports that as well. The main difference to what we've seen earlier is that ES6 standard Promises do not have Deferreds as an explicit concept. Instead, you just create a new Promise, giving it a function as an argument. The function receives two callbacks, `resolve` and `reject`, that you can call when you're ready to resolve or reject, respectively.

With ES6 promises you replace this:

```
var deferred = Q.defer();
doAsyncStuff(function(err) {
  if (err) {
    deferred.reject(err);
  } else {
    deferred.resolve();
  }
});
return deferred.promise;
```

With this:

```
return new Promise(function(resolve, reject) {
  doAsyncStuff(function(err) {
    if (err) {
      reject(err);
    } else {
      resolve();
    }
  });
});
```

So the Deferred object itself is replaced with these nested callbacks, leaving `Promise` as the only explicit API concept.

Let's see how `$q` can support a similar kind of API. First of all, `$q` is actually a function itself:

test/q_spec.js

```
describe('ES6 style', function() {

  it('is a function', function() {
    expect($q instanceof Function).toBe(true);
  });

});
```


Let's create a function called `$Q` inside our provider, and then justall the methods we've seen so far as attributes of that function:

src/q.js

```
var $Q = function Q() {

};

return _.extend($Q, {
  defer: defer,
  reject: reject,
  when: when,
  resolve: when,
  all: all
});
```

Like an ES6 Promise constructor, this new function expects to get a function as an argument. We'll call it the *resolver function*, and it is mandatory:

test/q_spec.js

```
it('expects a function as an argument', function() {
  expect($q).toThrow();
  $q(_.noop); // Just checking that this doesn't throw
});
```

src/q.js

```
var $Q = function Q(resolver) {
  if (!_.isFunction(resolver)) {
    throw 'Expected function, got ' + resolver;
  }
};
```

What you get back from this function is a Promise:

test/q_spec.js

```
it('returns a promise', function() {
  expect($q(_.noop)).toBeDefined();
  expect($q(_.noop).then).toBeDefined();
});
```


Internally we can implement this with our existing `defer` function:

src/q.js

```
var $Q = function Q(resolver) {
  if (!_.isFunction(resolver)) {
    throw 'Expected function, got ' + resolver;
  }
  var d = defer();

  return d.promise;
};
```

Just like in the ES6 Promise example we saw earlier, the resolver function gets invoked with a `resolve` callback argument. When that callback is invoked, the Promise becomes resolved:

test/q_spec.js

```
it('calls function with a resolve function', function() {
  var fulfilledSpy = jasmine.createSpy();

  $q(function(resolve) {
    resolve('ok');
  }).then(fulfilledSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).toHaveBeenCalledWith('ok');
});
```

We can actually implement that `resolve` function by just passing in the `resolve` method of our `Deferred` - as long as we bind its `this` value correctly before passing it in:

src/q.js

```
var $Q = function Q(resolver) {
  if (!_.isFunction(resolver)) {
    throw 'Expected function, got ' + resolver;
  }
  var d = defer();
  resolver(_.bind(d.resolve, d));
  return d.promise;
};
```

With `reject` we have a similar situation: It is passed in as the second argument to the resolver function. If it is invoked by the resolver, the Promise gets rejected:

test/q_spec.js


```
it('calls function with a reject function', function() {
  var fulfilledSpy = jasmine.createSpy();
  var rejectedSpy  = jasmine.createSpy();

  $q(function(resolve, reject) {
    reject('fail');
  }).then(fulfilledSpy, rejectedSpy);

  $rootScope.$apply();

  expect(fulfilledSpy).not.toHaveBeenCalled();
  expect(rejectedSpy).toHaveBeenCalledWith('fail');
});
```

We implement the rejection callback exactly like the resolve callback: It is the `reject` method of the Deferred, pre-bound:

src/q.js

```
var $Q = function Q(resolver) {
  if (!_.isFunction(resolver)) {
    throw 'Expected function, got ' + resolver;
  }
  var d = defer();
  resolver(
    _.bind(d.resolve, d),
    _.bind(d.reject, d)
  );
  return d.promise;
};
```

And there we have an ES6-esque API for `$q`! It is just a thin veneer on top of the `Deferred` implementation built in this chapter, but it is there for you to use if you prefer an API that's closer to standard ES6 Promises.

Promises Without $digest Integration: $$q

We'll end the chapter with an interesting little-known feature Angular ships with: The `$$q` service. This is a Promise implementation, like `$q`, but instead of integrating resolutions to the `$rootScope` digest, it resolves things with a browser timeout and doesn't involve a digest at all. In that sense it is much closer to non-Angular Promise implementations than `$q`.

> As the double-dollar prefix in the name implies, `$$q` is a private service in Angular, and using it comes with the usual warnings: You take the risk of your app breaking when you upgrade Angular, as `$$q` may be removed or changed without much notice in future versions.

`$$q` is used by Angular internally by the `$timeout` and `$interval` services when you invoke them with the `skipApply` flag. It is also used by `ngAnimate` for some of its async work.

Before we can start testing `$$q`, we should get it injected into our `$q` test suite, so let's get it from the injector. This will also temporarily break a whole lot of test cases:

test/q_spec.js

```
var $q, $$q, $rootScope;

beforeEach(function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  $q = injector.get('$q');
  $$q = injector.get('$$q');
  $rootScope = injector.get('$rootScope');
});
```

To restore those test cases, let's first make a provider for `$$q` in the `src/q.js` file:

src/q.js

```
function $$QProvider() {
  this.$get = function() {

  };
}
```

Then we can publish that in the `ng` module:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
}
```


And now we're ready to add our first test cases for `$$q`. The first thing we'll do is test that you can create Deferreds and Promises with it, just like you can with `$q`, but that they do *not* resolve when you run a digest:

test/q_spec.js

```
describe('$$q', function() {

  it('uses deferreds that do not resolve at digest', function() {
    var d = $$q.defer();
    var fulfilledSpy = jasmine.createSpy();
    d.promise.then(fulfilledSpy);
    d.resolve('ok');

    $rootScope.$apply();

    expect(fulfilledSpy).not.toHaveBeenCalled();
  });

});
```

Instead, those Promises get resolved when some time has passed. We can test this quite easily by using the fake clock feature that comes with Jasmine. It lets us essentially move the clock forward and see what happens:

test/q_spec.js

```
describe('$$q', function() {

  beforeEach(function() {
    jasmine.clock().install();
  });
  afterEach(function() {
    jasmine.clock().uninstall();
  });

  it('uses deferreds that do not resolve at digest', function() {
    var d = $$q.defer();
    var fulfilledSpy = jasmine.createSpy();
    d.promise.then(fulfilledSpy);
    d.resolve('ok');

    $rootScope.$apply();

    expect(fulfilledSpy).not.toHaveBeenCalled();
  });
```



```
  it('uses deferreds that resolve later', function() {
    var d = $$q.defer();
    var fulfilledSpy = jasmine.createSpy();
    d.promise.then(fulfilledSpy);
    d.resolve('ok');

    jasmine.clock().tick(1);

    expect(fulfilledSpy).toHaveBeenCalledWith('ok');
  });

});
```

Here we forward the clock by one millisecond and see that we have a resolution afterwards.

Also, an important performance-related characteristic of `$$q` is that when it does resolve its Promises, it does not *cause* a digest to run:

test/q_spec.js

```
it('does not invoke digest', function() {
  var d = $$q.defer();
  d.promise.then(_.noop);
  d.resolve('ok');

  var watchSpy = jasmine.createSpy();
  $rootScope.$watch(watchSpy);

  jasmine.clock().tick(1);

  expect(watchSpy).not.toHaveBeenCalled();
});
```

So, how do we create `$$q`? Do we need to reimplement everything we did for `$q` to have this alternative implementation? The answer is no, we do not. If you think about it, the only difference between `$q` and `$$q` is the way in which the resolution of Deferreds is postponed: `$q` does it with `$evalAsync`, and for `$$q` we should use `setTimeout`.

So, what we'll do is wrap everything that we currently have inside `$QProvider.$get` into a helper function, called `qFactory`. We can then give that function the "postponing strategy" as an argument. Here's how our updated `$QProvider` and `$$QProvider` implementations should look:

src/q.js

```
function $QProvider() {
  this.$get = ['$rootScope', function($rootScope) {
    return qFactory(function(callback) {
      $rootScope.$evalAsync(callback);
```



```
    });
  }];
}

function $$QProvider() {
  this.$get = function() {
    return qFactory(function(callback) {
      setTimeout(callback, 0);
    });
  };
}
```

All of our existing `$q` setup code now goes into `qFactory`, which, instead of using `$evalAsync`, uses the supplied "call later" function. Here's the final, full source code of `q.js`:

src/q.js

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

function qFactory(callLater) {

  function processQueue(state) {
    var pending = state.pending;
    delete state.pending;
    _.forEach(pending, function(handlers) {
      var deferred = handlers[0];
      var fn = handlers[state.status];
      try {
        if (_.isFunction(fn)) {
          deferred.resolve(fn(state.value));
        } else if (state.status === 1) {
          deferred.resolve(state.value);
        } else {
          deferred.reject(state.value);
        }
      } catch (e) {
        deferred.reject(e);
      }
    });
  }

  function scheduleProcessQueue(state) {
    callLater(function() {
      processQueue(state);
    });
  }

  function makePromise(value, resolved) {
```



```
  var d = new Deferred();
  if (resolved) {
    d.resolve(value);
  } else {
    d.reject(value);
  }
  return d.promise;
}

function handleFinallyCallback(callback, value, resolved) {
  var callbackValue = callback();
  if (callbackValue && callbackValue.then) {
    return callbackValue.then(function() {
      return makePromise(value, resolved);
    });
  } else {
    return makePromise(value, resolved);
  }
}

function Promise() {
  this.$$state = {};
}
Promise.prototype.then = function(onFulfilled, onRejected, onProgress) {
  var result = new Deferred();
  this.$$state.pending = this.$$state.pending || [];
  this.$$state.pending.push([result, onFulfilled, onRejected, onProgress]);
  if (this.$$state.status > 0) {
    scheduleProcessQueue(this.$$state);
  }
  return result.promise;
};
Promise.prototype.catch = function(onRejected) {
  return this.then(null, onRejected);
};
Promise.prototype.finally = function(callback, progressBack) {
  return this.then(function(value) {
    return handleFinallyCallback(callback, value, true);
  }, function(rejection) {
    return handleFinallyCallback(callback, rejection, false);
  }, progressBack);
};

function Deferred() {
  this.promise = new Promise();
}
Deferred.prototype.resolve = function(value) {
  if (this.promise.$$state.status) {
    return;
  }
  if (value && _.isFunction(value.then)) {
```

```
      value.then(
        _.bind(this.resolve, this),
        _.bind(this.reject, this),
        _.bind(this.notify, this)
      );
    } else {
      this.promise.$$state.value = value;
      this.promise.$$state.status = 1;
      scheduleProcessQueue(this.promise.$$state);
    }
  };
  Deferred.prototype.reject = function(reason) {
    if (this.promise.$$state.status) {
      return;
    }
    this.promise.$$state.value = reason;
    this.promise.$$state.status = 2;
    scheduleProcessQueue(this.promise.$$state);
  };
  Deferred.prototype.notify = function(progress) {
    var pending = this.promise.$$state.pending;
    if (pending && pending.length &&
        !this.promise.$$state.status) {
      callLater(function() {
        _.forEach(pending, function(handlers) {
          var deferred = handlers[0];
          var progressBack = handlers[3];
          try {
            deferred.notify(_.isFunction(progressBack) ?
                            progressBack(progress) :
                            progress
            );
          } catch (e) {
            console.log(e);
          }
        });
      });
    }
  };

  function defer() {
    return new Deferred();
  }

  function reject(rejection) {
    var d = defer();
    d.reject(rejection);
    return d.promise;
  }

  function when(value, callback, errback, progressback) {
```



```
    var d = defer();
    d.resolve(value);
    return d.promise.then(callback, errback, progressback);
  }

  function all(promises) {
    var results = _.isArray(promises) ? [] : {};
    var counter = 0;
    var d = defer();
    _.forEach(promises, function(promise, index) {
      counter++;
      when(promise).then(function(value) {
        results[index] = value;
        counter--;
        if (!counter) {
          d.resolve(results);
        }
      }, function(rejection) {
        d.reject(rejection);
      });
    });
    if (!counter) {
      d.resolve(results);
    }
    return d.promise;
  }

  var $Q = function Q(resolver) {
    if (!_.isFunction(resolver)) {
      throw 'Expected function, got ' + resolver;
    }
    var d = defer();
    resolver(
      _.bind(d.resolve, d),
      _.bind(d.reject, d)
    );
    return d.promise;
  };

  return _.extend($Q, {
    defer: defer,
    reject: reject,
    when: when,
    resolve: when,
    all: all
  });

}

function $QProvider() {
  this.$get = ['$rootScope', function($rootScope) {
```

```
    return qFactory(function(callback) {
      $rootScope.$evalAsync(callback);
    });
  }];
}

function $$QProvider() {
  this.$get = function() {
    return qFactory(function(callback) {
      setTimeout(callback, 0);
    });
  };
}
```

Simple function composition gets us a lot of mileage here! We essentially get two services for the price of one.

Summary

`$q` is a relatively simple service, at least compared to some of the other Angular components we build in this book. But since asynchronous programming itself can be tricky, using `$q` is not always as straightforward as one might like, and reasoning about workflows and failure modes can be difficult.

Since you now know exactly how `$q` works, you are hopefully well-equipped to use everything it has to offer to make working with asynchrony much less painful than it used to be.

In this chapter you have learned:

- What kinds of problems Promises are designed to solve.
- About some of the existing Promise implementations for JavaScript.
- How AngularJS Promises compare to some of the other existing implementations.
- That Promises are made available in Angular by the `$q` and `$$q` services.
- That Promises are always paired with Deferreds, and whereas a Promise is accessed by the consumer of an asynchronously produced value, a Deferred is accessed by its producer.
- That when a Deferred is resolved, the associated Promise callbacks get invoked with `$evalAsync`.
- That each Promise is resolved at most once.
- That each Promise callback is invoked at most once.
- How a Promise callback is invoked even when registered after the Promise was already resolved.
- How Promises can be either resolved or *rejected*.
- That rejections can be caught by rejection errbacks.
- That you can execute resource cleanup code in finally callbacks, which are callbacks that are invoked for both rejected and resolved Promises.

- How you can chain several `then`, `catch`, and `finally` calls together, and each call creates a new Promise chained to the previous one.
- How even the last (or only) `then` in your Promise chain creates another Promise, but that it is simply ignored.
- How an exception thrown in a Promise handler rejects the next Promise in the chain.
- How an exception caught by a rejection errback resolves the next Promise in the chain.
- That a Promise callback may return *another* Promise, and how that Promise's eventual resolution or rejection is linked to the Promise chain.
- That finally handlers can also return Promises and they are resolved before continuing with the chain, but that the values they resolve to are ignored.
- How Deferreds can notify about progress, and how progress callbacks are registered.
- That, unlike all other Promise callbacks, notification callbacks may get called several times.
- How notification messages can be transformed in a Promise chain.
- How an immediately rejected Promise can be created with `$q.reject`.
- How an immediately resolved Promise can be created with `$q.when`.
- How `$q.when` can also be used to adopt a foreign Promise.
- How an array or object of Promises can be resolved to an array or object of values with `$q.all`.
- That `$q.all` is rejected if even one of its argument Promises is rejected.
- That not all of the items in the collection given to `$q.all` have to be Promises.
- How `$q` also supports an alternative ES6-style Promise API.

Chapter 14

$http

Before moving on to the final major theme of the book - the directive system - there's one more crucial piece of infrastructure we need to have, and that is the `$http` service. This is the service that handles all HTTP communication in Angular applications.

Most Angular applications use `$http` either directly or indirectly through `$resource` or other API wrappers. Angular also uses `$http` internally to load things like HTML templates, as we will see when we get into the directive implementation.

In this chapter we'll see what `$http` does and how it uses core services like `$rootScope` and `$q` to provide its features.

What We Will Skip

We're going to omit a few things that are not central to our discussion:

- The integration between `$http` and the `$cache` service
- The cross-site request forgery support in `$http`
- The JSONP support in `$http`

If you are interested in any of these features, picking them up from the AngularJS source code should be straightforward once you have gone through this chapter and learned how the `$http` service is constructed.

The Providers

Just as we did with `$q`, we'll begin by setting up a Provider that will make `$http` available. In this instance, we will actually set up *two* providers, since the work related to HTTP communication is divided among two services in AngularJS: There's `$http` itself, and there's something called `$httpBackend`.

The division of labor between these two services is such that `$httpBackend` handles the low-level XMLHttpRequest integration, while `$http` handles the high-level, user-facing

features. From an application developer's point of view, this division does not come up that often since most of the time `$http` is the only service used directly, while `$httpBackend` is only used internally by `$http`.

The distinction does become useful when you want to use some alternative implementation for the HTTP transport. For example, the ngMock module overrides `$httpBackend` to replace actual HTTP calls with fake HTTP calls for testing purposes.

In any case, we can expect both of these services to be available in our injector through the `ng` module:

test/angular_public_spec.js

```
it('sets up $http and $httpBackend', function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  expect(injector.has('$http')).toBe(true);
  expect(injector.has('$httpBackend')).toBe(true);
});
```

The two services will live in two separate files. `$httpBackend` will be in `http_backend.js`, where it is created by a provider:

src/http_backend.js

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

function $HttpBackendProvider() {

  this.$get = function() {

  };

}
```

The `$http` service itself is in `http.js`, and is also created by a provider. This provider's `$get` method has a dependency to the `$httpBackend`, though we won't actually use it just yet:

src/http.js

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

function $HttpProvider() {

  this.$get = ['$httpBackend', function() {

  }];

}
```


Now we can register both of these providers into the `ng` module:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
  ngModule.provider('$httpBackend', $HttpBackendProvider);
  ngModule.provider('$http', $HttpProvider);

}
```

Sending HTTP Requests

With the providers out of the way, we can start thinking about what `$http` is actually supposed to do. The core of it is to send HTTP requests to remote servers and give back the responses. So let's take that as our goal for this section.

First of all, the `$http` should be a function one can call to make a request. Let's add a test case for that, into a new test file `http_spec.js`:

test/http_spec.js

```
/* jshint globalstrict: true */
/* global publishExternalAPI: false, createInjector: false */
'use strict';

describe('$http', function() {

  var $http;

  beforeEach(function() {
    publishExternalAPI();
    var injector = createInjector(['ng']);
    $http = injector.get('$http');
  });

  it('is a function', function() {
    expect($http instanceof Function).toBe(true);
  });

});
```


The `$http` function should make an HTTP request and return a response. But since HTTP requests are asynchronous, the function can't just return a response directly. It needs to return a *Promise* for a response instead:

test/http_spec.js

```
it('returns a Promise', function() {
  var result = $http({});
  expect(result).toBeDefined();
  expect(result.then).toBeDefined();
});
```

We can make these first two test cases pass by just having `$httpProvider.$get` return a function that creates a Deferred and then returns its Promise. We need to inject `$q` in order to do that:

src/http.js

```
this.$get = ['$httpBackend', '$q', function($httpBackend, $q) {

  return function $http() {
    var deferred = $q.defer();
    return deferred.promise;
  };

}];
```

So that's how `$http` receives a request and eventually returns a response, but what should happen in between? We should make the actual HTTP request here, and for that we're going to use the standard XMLHttpRequest object that all browsers provide.

We will want to unit test all of this too, but we don't want to be making any actual network requests from our unit tests, since that would necessitate having a server. Here's where the SinonJS library we installed back in Chapter 0 becomes handy. Sinon ships with a fake XMLHttpRequest implementation that can temporarily replace the browser's built-in XMLHttpRequest. It can be used to introspect what requests have been made and to return fake responses without ever leaving the browser's JavaScript execution environment.

To enable Sinon's fake XMLHttpRequest, we need to set it up in a `beforeEach` function, and then remove it in an `afterEach` function so that we have a fresh environment after each test:

test/http_spec.js


```
describe('$http', function() {

  var $http;
  var xhr;

  beforeEach(function() {
    publishExternalAPI();
    var injector = createInjector(['ng']);
    $http = injector.get('$http');
  });

  beforeEach(function() {
    xhr = sinon.useFakeXMLHttpRequest();
  });
  afterEach(function() {
    xhr.restore();
  });

  // ...

});
```

Let's also make it easier for ourselves to check what requests have been made. For each request sent, Sinon will call the `onCreate` function of the fake XHR. If we attach a function to `onCreate` we can collect all of those requests into an array:

test/http_spec.js

```
describe('$http', function() {

  var $http;
  var xhr, requests;

  beforeEach(function() {
    publishExternalAPI();
    var injector = createInjector(['ng']);
    $http = injector.get('$http');
  });

  beforeEach(function() {
    xhr = sinon.useFakeXMLHttpRequest();
    requests = [];
    xhr.onCreate = function(req) {
      requests.push(req);
    };
  });
  afterEach(function() {
    xhr.restore();
  });
```



```
  // ...

});
```

Now we're ready to create our first test for request sending. If we call **$http** with an object that says "send a POST request with data 'hello' to http://teropa.info", we can check that an asynchronous XMLHttpRequest with those parameters is indeed sent:

test/http_spec.js

```
it('makes an XMLHttpRequest to given URL', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 'hello'
  });
  expect(requests.length).toBe(1);
  expect(requests[0].method).toBe('POST');
  expect(requests[0].url).toBe('http://teropa.info');
  expect(requests[0].async).toBe(true);
  expect(requests[0].requestBody).toBe('hello');
});
```

As discussed earlier, **$http** will delegate the actual network communication tasks to the **$httpBackend** service, so that is where the XMLHttpRequest will actually be created. Let's just assume that **$httpBackend** is a function, and call it with the method, URL, and data arguments unpacked from the request config:

src/http.js

```
return function $http(config) {
  var deferred = $q.defer();
  $httpBackend(config.method, config.url, config.data);
  return deferred.promise;
};
```

In **$httpBackend** we can now make a standard XMLHttpRequest, open it with the arguments given, and send the data:

src/http_backend.js

```
this.$get = function() {
  return function(method, url, post) {
    var xhr = new window.XMLHttpRequest();
    xhr.open(method, url, true);
    xhr.send(post || null);
  };
};
```


The three arguments to `xhr.open` are the HTTP method to use, the URL to send to, and whether the request is asynchronous (which it always is in AngularJS).

The one argument to `xhr.send` is the data to send. Not all requests have data, and when there isn't any we send an explicit `null` value.

This satisfies our current test suite, but there's one crucial step we're missing: The Promise we are returning is never actually getting resolved, because we haven't connected it to the XMLHttpRequest in any way. That Promise should be resolved with a response object that gives the user access to the HTTP response status and data, as well as the original request configuration:

test/http_spec.js

```
it('resolves promise when XHR result received', function() {
  var requestConfig = {
    method: 'GET',
    url: 'http://teropa.info'
  };

  var response;
  $http(requestConfig).then(function(r) {
    response = r;
  });

  requests[0].respond(200, {}, 'Hello');

  expect(response).toBeDefined();
  expect(response.status).toBe(200);
  expect(response.statusText).toBe('OK');
  expect(response.data).toBe('Hello');
  expect(response.config.url).toEqual('http://teropa.info');
});
```

In tests we can use Sinon's `respond` method to respond to a fake XMLHttpRequest. The three arguments given to `respond` are the HTTP status code, the HTTP response headers, and the response body.

The way this works between `$http` and `$httpBackend` is that `$httpBackend` does not use any Deferreds or Promises. Instead, it receives a traditional callback function. It then attaches an `onload` handler to the XMLHttpRequest, and invokes the callback when that handler fires:

src/http_backend.js

```
this.$get = function() {
  return function(method, url, post, callback) {
    var xhr = new window.XMLHttpRequest();
    xhr.open(method, url, true);
```



```
    xhr.send(post || null);
    xhr.onload = function() {
      var response = ('response' in xhr) ? xhr.response :
                                           xhr.responseText;
      var statusText = xhr.statusText || '';
      callback(xhr.status, response, statusText);
    };
  };
};
```

Inside `onload`, we try to get the response body primarily from `xhr.response` and secondarily from `xhr.responseText`. Some browsers support one, some the other, so we need to try both. We also get the numeric and textual status from the response, and then pass everything we have to the callback function.

Back in `$http` we can now tie all of this together. We need to construct that callback, which we'll call `done`. When it is invoked, it resolves the Promise constructed earlier. The resolution value is the *response object* - an object that collects all the information we have about the response:

src/http.js

```
return function $http(config) {
  var deferred = $q.defer();

  function done(status, response, statusText) {
    deferred.resolve({
      status: status,
      data: response,
      statusText: statusText,
      config: config
    });
  }

  $httpBackend(config.method, config.url, config.data, done);
  return deferred.promise;
};
```

As far as our test goes, this isn't quite enough yet though. The problem is related to the Promise resolution: You may recall from the previous chapter that when a Promise is resolved, callbacks are not executed immediately, but only during the next digest.

What we should do in `$http` is to *kick off a digest if one isn't already running*. We can do this using the `$apply` function of `$rootScope`:

src/http.js

```
this.$get = ['$httpBackend', '$q', '$rootScope',
  function($httpBackend, $q, $rootScope) {

  return function $http(config) {
    var deferred = $q.defer();

    function done(status, response, statusText) {
      deferred.resolve({
        status: status,
        data: response,
        statusText: statusText,
        config: config
      });
      if (!$rootScope.$$phase) {
        $rootScope.$apply();
      }
    }

    $httpBackend(config.method, config.url, config.data, done);
    return deferred.promise;
  };

}];
```

And now our test is passing!

This is one of the reasons it's nice to use `$http` when doing Ajax in Angular: You don't need to care about calling `$apply` because the framework does it for you.

Another reason is related to what happens when things go wrong, and with HTTP requests, it is quite common that things go wrong. Servers may return HTTP statuses that indicate failures, or fail to respond at all. In these cases, the built-in error management that we have in Promises is very useful. We can just *reject* the Promise instead of resolving it. This is the case, for example, when a server responds with `401`:

test/http_spec.js

```
it('rejects promise when XHR result received with error status', function() {
  var requestConfig = {
    method: 'GET',
    url: 'http://teropa.info'
  };

  var response;
  $http(requestConfig).catch(function(r) {
    response = r;
  });

  requests[0].respond(401, {}, 'Fail');
```



```
  expect(response).toBeDefined();
  expect(response.status).toBe(401);
  expect(response.statusText).toBe('Unauthorized');
  expect(response.data).toBe('Fail');
  expect(response.config.url).toEqual('http://teropa.info');
});
```

Note that the data given to the Promise handler is the same as it was with the successful response: The response object. The only difference is which handler gets called: `then` or `catch`.

We can dynamically select the Deferred method to invoke, based on the status code:

src/http.js

```
function done(status, response, statusText) {
  deferred[isSuccess(status) ? 'resolve' : 'reject']({
    status: status,
    data: response,
    statusText: statusText,
    config: config
  });
  if (!$rootScope.$$phase) {
    $rootScope.$apply();
  }
}
```

The new `isSuccess` helper function used here will return `true` if the status code is between 200 and 299, and `false` otherwise:

src/http.js

```
function isSuccess(status) {
  return status >= 200 && status < 300;
}
```

> This function would consider redirect responses, such as 302, as errors. However, redirects do not trigger errors since they are handled internally by the web browser and never get to our JavaScript code.

Another reason for an `$http` Promise to become rejected is if the request fails completely, so that there is no response. There are various reasons why this could happen: The network could be down, there could be a Cross-Origin Resource Sharing problem, or the request could be explicitly aborted.

To unit tests such a situation, we can just invoke the `onerror` handler of the (fake) request directly, since that is what gets called in real XMLHttpRequests when things go wrong:

test/http_spec.js


```
it('rejects promise when XHR result errors/aborts', function() {
  var requestConfig = {
    method: 'GET',
    url: 'http://teropa.info'
  };

  var response;
  $http(requestConfig).catch(function(r) {
    response = r;
  });

  requests[0].onerror();

  expect(response).toBeDefined();
  expect(response.status).toBe(0);
  expect(response.data).toBe(null);
  expect(response.config.url).toEqual('http://teropa.info');
});
```

In this case, we expect the status code of the response to become `0`, and the response data to become `null`.

In `$httpBackend` we should attach an `onerror` handler to catch errors from the native XMLHttpRequest. When it's called, we invoke the callback with a `-1` status code, a `null` response, and an empty status text:

src/http_backend.js

```
return function(method, url, post, callback) {
  var xhr = new window.XMLHttpRequest();
  xhr.open(method, url, true);
  xhr.send(post || null);
  xhr.onload = function() {
    var response = ('response' in xhr) ? xhr.response :
                                         xhr.responseText;
    var statusText = xhr.statusText || '';
    callback(xhr.status, response, statusText);
  };
  xhr.onerror = function() {
    callback(-1, null, '');
  };
};
```

All that remains to be done in `$http` is a "normalization" of the status code. In error responses, `$httpBackend` may return negative status codes, but `$http` never resolves to anything smaller than `0`:

src/http.js

```
function done(status, response, statusText) {
  status = Math.max(status, 0);
  deferred[isSuccess(status) ? 'resolve' : 'reject']({
    status: status,
    data: response,
    statusText: statusText,
    config: config
  });
  if (!$rootScope.$$phase) {
    $rootScope.$apply();
  }
}
```

Default Request Configuration

As we have already seen, the `$http` function takes a request configuration object as its one and only argument. That object includes all the attributes needed to make the request: The URL, the HTTP method, the content, and so on. Not all of these attributes are always required, however. There are also *default* values that are used for attributes that are omitted from the request config.

At this point, we'll set up the first of these default values: The request method. If no method is supplied when a request is made, it is assumed to be a GET.

test/http_spec.js

```
it('uses GET method by default', function() {
  $http({
    url: 'http://teropa.info'
  });
  expect(requests.length).toBe(1);
  expect(requests[0].method).toBe('GET');
});
```

We can apply defaults like these by just constructing a "default config" object in the `$http` function, and then extending it with the config given as an argument, allowing it to override the defaults:

src/http.js

```
return function $http(requestConfig) {
  var deferred = $q.defer();

  var config = _.extend({
    method: 'GET'
  }, requestConfig);

  // ...

};
```


Not everything in `$http` can be preconfigured as default values, but there are a few more defaults we'll add during the course of this chapter.

Request Headers

When sending requests to HTTP servers, it is vital to have the ability to attach *headers*. Headers are needed for all kinds of information we want the server to know about, like authentication tokens, preferred content types, and HTTP cache control.

The `$http` service has full support for HTTP headers, through a `headers` object that you can attach to the request configuration object:

test/http_spec.js

```
it('sets headers on request', function() {
  $http({
    url: 'http://teropa.info',
    headers: {
      'Accept': 'text/plain',
      'Cache-Control': 'no-cache'
    }
  });
  expect(requests.length).toBe(1);
  expect(requests[0].requestHeaders.Accept).toBe('text/plain');
  expect(requests[0].requestHeaders['Cache-Control']).toBe('no-cache');
});
```

The `$httpBackend` service will do most of the work here. From `$http` we just need to pass the headers object along:

src/http.js

```
return function $http(requestConfig) {
  // ...

  $httpBackend(
    config.method,
    config.url,
    config.data,
    done,
    config.headers
  );
  return deferred.promise;
};
```


What `$httpBackend` should do is take all the headers given to it and set them on the XMLHttpRequest, using its `setRequestHeader()` method:

src/http_backend.js

```
return function(method, url, post, callback, headers) {
  var xhr = new window.XMLHttpRequest();
  xhr.open(method, url, true);
  _.forEach(headers, function(value, key) {
    xhr.setRequestHeader(key, value);
  });
  xhr.send(post || null);
  // ...
};
```

There are also some default headers that are set, even when omitted from the request config object. Most importantly, the Accept header is by default set to a value that tells the server we primarily prefer JSON responses, and secondarily plain text responses:

test/http_spec.js

```
it('sets default headers on request', function() {
  $http({
    url: 'http://teropa.info'
  });
  expect(requests.length).toBe(1);
  expect(requests[0].requestHeaders.Accept).toBe(
    'application/json, text/plain, */*');
});
```

Let's store these default headers in a variable called `defaults`, set up in the scope of the `$HttpProvider` constructor. The variable points to an object that has a `headers` key. In it we have a nested key called `common`, to store all headers that are common across all HTTP methods. The Accept header is one of them:

src/http.js

```
function $HttpProvider() {

  var defaults = {
    headers: {
      common: {
        Accept: 'application/json, text/plain, */*'
      }
    }
  };

  // ...

}
```


We need to merge these defaults with any headers actually given in the request config object. We'll do this in a new helper function called `mergeHeaders`:

src/http.js

```
return function $http(requestConfig) {
  var deferred = $q.defer();

  var config = _.extend({
    method: 'GET'
  }, requestConfig);
  config.headers = mergeHeaders(requestConfig);

  // ...

};
```

For now, this function will just create a new object, into which it pours all the common default headers from the `defaults` object, and then all the headers given in the request configuration object:

src/http.js

```
function mergeHeaders(config) {
  return _.extend(
    {},
    defaults.headers.common,
    config.headers
  );
}
```

Not all default headers are common across all HTTP methods. POSTs, for example, should have a default `Content-Type` header, set to a JSON content type, whereas GET methods should not. This is because GET requests do not have a body, so setting their content type is not appropriate.

test/http_spec.js

```
it('sets method-specific default headers on request', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: '42'
  });
  expect(requests.length).toBe(1);
  expect(requests[0].requestHeaders['Content-Type']).toBe(
    'application/json;charset=utf-8');
});
```


The `defaults` variable also contains these method-specific defaults. Let's set the JSON `Content-Type` header for the standard HTTP methods that have a body: POST, PUT, and PATCH:

src/http.js

```
var defaults = {
  headers: {
    common: {
      Accept: 'application/json, text/plain, */*'
    },
    post: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    put: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    patch: {
      'Content-Type': 'application/json;charset=utf-8'
    }
  }
};
```

We can now extend `mergeHeaders` to also include any method-specific default headers:

src/http.js

```
function mergeHeaders(config) {
  return _.extend(
    {},
    defaults.headers.common,
    defaults.headers[(config.method || 'get').toLowerCase()],
    config.headers
  );
}
```

Another aspect of default headers is that as an application developer, you can also modify them. They are exposed through the `defaults` attribute of the `$http` service, and we can just mutate the object in that attribute to set application-global defaults:

test/http_spec.js


```
it('exposes default headers for overriding', function() {
  $http.defaults.headers.post['Content-Type'] = 'text/plain;charset=utf-8';
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: '42'
  });
  expect(requests.length).toBe(1);
  expect(requests[0].requestHeaders['Content-Type']).toBe(
    'text/plain;charset=utf-8');
});
```

We can attach this attribute to the `$http` function, but to do so we need to also reorganize our code so that the `$http` function declaration is separate from the `return` statement:

src/http.js

```
function $http(requestConfig) {
  var deferred = $q.defer();

  var config = _.extend({
    method: 'GET'
  }, requestConfig);
  config.headers = mergeHeaders(requestConfig);

  function done(status, response, statusText) {
    status = Math.max(status, 0);
    deferred[isSuccess(status) ? 'resolve' : 'reject']({
      status: status,
      data: response,
      statusText: statusText,
      config: config
    });
    if (!$rootScope.$$phase) {
      $rootScope.$apply();
    }
  }

  $httpBackend(config.method, config.url, config.data, done, config.headers);
  return deferred.promise;
}
$http.defaults = defaults;
return $http;
```

The headers are available not only in `$http` at runtime, but also in `$httpProvider` at configuration time. We can test this by creating another injector with a custom function module:

test/http_spec.js


```
it('exposes default headers through provider', function() {
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.defaults.headers.post['Content-Type'] =
      'text/plain;charset=utf-8';
  }]);
  $http = injector.get('$http');

  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: '42'
  });
  expect(requests.length).toBe(1);
  expect(requests[0].requestHeaders['Content-Type']).toBe(
    'text/plain;charset=utf-8');
});
```

We can satisfy this requirement by simply attaching the defaults to `this` as we introduce them within the `$HttpProvider` constructor:

src/http.js

```
function $HttpProvider() {

  var defaults = this.defaults = {
    // ...
  };

  // ...

}
```

> We now have default values declared in two places: Default headers are in the *defaults* variable, and the default request method is in the object created on the fly inside the `$http` function. The difference is that the former is exposed for modification and the latter is not. The default HTTP method can't be set from application code.

If you're familiar with how HTTP works, you'll know that the names of headers are actually processed case-insensitively: `Content-Type` and `content-type` should be treated interchangeably. The way we currently merge default headers with request-specific headers is not consistent with this: We may end up having the same header twice, with different capitalization. We should instead merge headers in a case-insensitive fashion so that this does not happen.

test/http_spec.js


```
it('merges default headers case-insensitively', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: '42',
    headers: {
      'content-type': 'text/plain;charset=utf-8'
    }
  });
  expect(requests.length).toBe(1);
  expect(requests[0].requestHeaders['content-type']).toBe(
    'text/plain;charset=utf-8');
  expect(requests[0].requestHeaders['Content-Type']).toBeUndefined();
});
```

This means we need to change how the `mergeHeaders` function works. We'll first split things into two objects: One called `reqHeaders` for all the headers that came with the request configuration object, and one called `defHeaders` for default headers - both common and HTTP method specific:

src/http.js

```
function mergeHeaders(config) {
  var reqHeaders = _.extend(
    {},
    config.headers
  );
  var defHeaders = _.extend(
    {},
    defaults.headers.common,
    defaults.headers[(config.method || 'get').toLowerCase()]
  );
}
```

Now we should combine these two objects. We'll take `reqHeaders` as a starting point, and apply everything from `defHeaders` to it. For each default header, we check if we already have a header with that name, with a case-insensitive check. We add the header to the result only if we don't already have it:

src/http.js

```
function mergeHeaders(config) {
  var reqHeaders = _.extend(
    {},
    config.headers
  );
  var defHeaders = _.extend(
    {},
```



```
    defaults.headers.common,
    defaults.headers[(config.method || 'get').toLowerCase()]
  );
  _.forEach(defHeaders, function(value, key) {
    var headerExists = _.any(reqHeaders, function(v, k) {
      return k.toLowerCase() === key.toLowerCase();
    });
    if (!headerExists) {
      reqHeaders[key] = value;
    }
  });
  return reqHeaders;
}
```

That takes care of header merging, and we can turn our attention to a couple of further special cases that have to do with header processing.

We've seen the Content-Type header in action, and how it is set to JSON by default. But we should additionally make sure that Content-Type is *not* set if there's no actual body in the request. If there's no body, any `Content-Type` would be misleading, and we should omit it even if it was configured:

test/http_spec.js

```
it('does not send content-type header when no data', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    headers: {
      'Content-Type': 'application/json;charset=utf-8'
    }
  });
  expect(requests.length).toBe(1);
  expect(requests[0].requestHeaders['Content-Type']).not.toBe(
    'application/json;charset=utf-8');
});
```

We can do this by just iterating over the headers and removing the one for `content-type` if there's no data to send:

src/http.js

```
function $http(requestConfig) {
  var deferred = $q.defer();

  var config = _.extend({
    method: 'GET'
  }, requestConfig);
```



```
  config.headers = mergeHeaders(requestConfig);

  if (_.isUndefined(config.data)) {
    _.forEach(config.headers, function(v, k) {
      if (k.toLowerCase() === 'content-type') {
        delete config.headers[k];
      }
    });
  }

  // ...
}
```

The final aspect of request headers we'll talk about is that a header's value may actually not always be a string, but instead *a function that produces a string*. If `$http` encounters a function as a header value, it invokes that function, giving it the request configuration object as an argument. This can be useful if you want to set default headers, but still dynamically form them separately for each request:

test/http_spec.js

```
it('supports functions as header values', function() {
  var contentTypeSpy = jasmine.createSpy().and.returnValue(
    'text/plain;charset=utf-8');
  $http.defaults.headers.post['Content-Type'] = contentTypeSpy;

  var request = {
    method: 'POST',
    url: 'http://teropa.info',
    data: 42
  };
  $http(request);

  expect(contentTypeSpy).toHaveBeenCalledWith(request);
  expect(requests[0].requestHeaders['Content-Type']).toBe(
    'text/plain;charset=utf-8');
});
```

At the end of `mergeHeaders`, we'll invoke a new helper function called `executeHelperFns`, which will resolve any of these function headers:

src/http.js

```
function mergeHeaders(config) {
  // ...
  return executeHeaderFns(reqHeaders, config);
}
```

This function iterates over the headers object using `_.transform`, and replaces any function value with the return value of the function, when called with the request config:

src/http.js

```
function executeHeaderFns(headers, config) {
  return _.transform(headers, function(result, v, k) {
    if (_.isFunction(v)) {
      result[k] = v(config);
    }
  }, headers);
}
```

The one case where the result of the header function should not be attached to the request is when it is `null` or `undefined`. Those values are omitted:

test/http_spec.js

```
it('ignores header function value when null/undefined', function() {
  var cacheControlSpy = jasmine.createSpy().and.returnValue(null);
  $http.defaults.headers.post['Cache-Control'] = cacheControlSpy;

  var request = {
    method: 'POST',
    url: 'http://teropa.info',
    data: 42
  };
  $http(request);

  expect(cacheControlSpy).toHaveBeenCalledWith(request);
  expect(requests[0].requestHeaders['Cache-Control']).toBeUndefined();
});
```

The `executeHeaderFns` function explicitly checks if the return value of the function is `null` or `undefined`, and removes the header if it is:

src/http.js

```
function executeHeaderFns(headers, config) {
  return _.transform(headers, function(result, v, k) {
    if (_.isFunction(v)) {
      v = v(config);
      if (_.isNull(v) || _.isUndefined(v)) {
        delete result[k];
      } else {
        result[k] = v;
      }
    }
  }, headers);
}
```


Response Headers

We're now able to set headers on HTTP requests in several ways, and can turn our focus to the other major task of header processing: Response headers.

The response headers returned by HTTP servers are all made available to application code in Angular. They are included in the response object you eventually get from `$http`. The response object will have a `headers` attribute that points to a *header getter function*. That function takes header names and returns the corresponding header values:

test/http_spec.js

```
it('makes response headers available', function() {
  var response;
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 42
  }).then(function(r) {
    response = r;
  });

  requests[0].respond(200, {'Content-Type': 'text/plain'}, 'Hello');

  expect(response.headers).toBeDefined();
  expect(response.headers instanceof Function).toBe(true);
  expect(response.headers('Content-Type')).toBe('text/plain');
  expect(response.headers('content-type')).toBe('text/plain');
});
```

Note that just like with request headers, we expect response header handling to be case-insensivite: Accessing the `Content-Type` header and the `content-type` header should both have exactly the same effect, regardless of what sort of capitalization the server actually used.

We begin the implementation of response headers from the HTTP backend. As it invokes the response callback, it should provide all the headers returned by the server. They are available via the `getAllResponseHeaders()` method of the XMLHttpRequest:

src/http_backend.js

```
xhr.onload = function() {
  var response = ('response' in xhr) ? xhr.response :
                                       xhr.responseText;
  var statusText = xhr.statusText || '';
  callback(
    xhr.status,
    response,
    xhr.getAllResponseHeaders(),
    statusText
  );
};
```


Over in `$http` we now have the headers (still as an unparsed string at this point), and we can create the header getter function for the response object. We'll use a new helper function called `headersGetter` to make that function:

src/http.js

```
function done(status, response, headersString, statusText) {
  status = Math.max(status, 0);
  deferred[isSuccess(status) ? 'resolve' : 'reject']({
    status: status,
    data: response,
    statusText: statusText,
    headers: headersGetter(headersString),
    config: config
  });
  if (!$rootScope.$$phase) {
    $rootScope.$apply();
  }
}
```

The `headersGetter` function takes the headers string and returns a function that resolves header names to header values. This'll be the function that application code will invoke when it needs to access a header:

src/http.js

```
function headersGetter(headers) {
  return function(name) {

  };
}
```

To gain access to individual headers, we'll need to parse the combined header string that we have at the moment. We'll do this *lazily*, so that nothing is parsed until the first header is actually requested. We then cache the parse result for subsequent calls:

src/http.js

```
function headersGetter(headers) {
  var headersObj;
  return function(name) {
    headersObj = headersObj || parseHeaders(headers);
    return headersObj[name.toLowerCase()];
  };
}
```


With this pattern we ensure the cost of parsing headers is not incurred unless someone actually needs header information.

The actual work of parsing the headers is done by another helper function, `parseHeaders`. It'll take the header string and return an object of the individual headers. This is done by first splitting the headers string to separate lines (HTTP headers always have one name-value pair per line), and then by iterating over those lines:

src/http.js

```
function parseHeaders(headers) {
  var lines = headers.split('\n');
  return _.transform(lines, function(result, line) {

  }, {});
}
```

Each of the header lines will have a header name, followed by a colon character `':'`, followed by the header value. We just need to grab the parts before and after the colon character, trim any surrounding whitespace, downcase the header name, and then put what we have into the result object

src/http.js

```
function parseHeaders(headers) {
  var lines = headers.split('\n');
  return _.transform(lines, function(result, line) {
    var separatorAt = line.indexOf(':');
    var name = _.trim(line.substr(0, separatorAt)).toLowerCase();
    var value = _.trim(line.substr(separatorAt + 1));
    if (name) {
      result[name] = value;
    }
  }, {});
}
```

The `headers` function may also be called with zero arguments, in which case it should return the full, parsed headers object:

test/http_spec.js

```
it('may returns all response headers', function() {
  var response;
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 42
  }).then(function(r) {
    response = r;
```



```
  });

  requests[0].respond(200, {'Content-Type': 'text/plain'}, 'Hello');

  expect(response.headers()).toEqual({'content-type': 'text/plain'});
});
```

This is achieved by checking the argument given to the headers getter. If no header name was given, just return the full parse result:

src/http.js

```
function headersGetter(headers) {
  var headersObj;
  return function(name) {
    headersObj = headersObj || parseHeaders(headers);
    if (name) {
      return headersObj[name.toLowerCase()];
    } else {
      return headersObj;
    }
  };
}
```

And there's our response header processing!

Allow CORS Authorization: withCredentials

Making XMLHttpRequests to domains other than the current page's origin has some security related restrictions involved. These days, the management of these restrictions is most often done with cross-origin resource sharing (CORS).

Most of the things that need to be done with CORS don't actually involve JavaScript code: It's pretty much all done between the web server and the web browser. But there is one thing we should build into `$http` to fully support CORS. By default, cross-domain requests do *not* include any cookies or authentication headers. If either of those is needed, the `withCredentials` flag needs to be set on the XMLHttpRequest.

With Angular, you can make this happen by attaching a `withCredentials` flag on the request object:

test/http_spec.js

```
it('allows setting withCredentials', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
```

```
    data: 42,
    withCredentials: true
  });

  expect(requests[0].withCredentials).toBe(true);
});
```

This flag is extracted in `$http` and given as an argument to the `$httpBackend` call:

src/http.js

```
$httpBackend(
  config.method,
  config.url,
  config.data,
  done,
  config.headers,
  config.withCredentials
);
```

In the backend, the flag is set on the XMLHttpRequest if its value is truthy:

src/http_backend.js

```
return function(method, url, post, callback, headers, withCredentials) {
  var xhr = new window.XMLHttpRequest();
  xhr.open(method, url, true);
  _.forEach(headers, function(value, key) {
    xhr.setRequestHeader(key, value);
  });
  if (withCredentials) {
    xhr.withCredentials = true;
  }
  xhr.send(post || null);
  // ...
};
```

The `withCredentials` flag can also be set globally, using the `defaults` configuration:

test/http_spec.js

```
it('allows setting withCredentials from defaults', function() {
  $http.defaults.withCredentials = true;

  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 42
  });

  expect(requests[0].withCredentials).toBe(true);
});
```


When the request configuration is constructed in `$http`, the default value is used, but only if the value given in the actual request config is `undefined`:

src/http.js

```
function $http(requestConfig) {
  var deferred = $q.defer();

  var config = _.extend({
    method: 'GET'
  }, requestConfig);
  config.headers = mergeHeaders(requestConfig);

  if (_.isUndefined(config.withCredentials) &&
      !_.isUndefined(defaults.withCredentials)) {
    config.withCredentials = defaults.withCredentials;
  }

  // ...

}
```

Request Transforms

When you communicate with a server, you often need to preprocess your data somehow so that it is in a format that the server can understand, such as JSON, XML, or some custom format.

When you have such preprocessing needs in your Angular application, it is of course completely possible to just do it separately for every request you make: You make sure that what you put in the `data` attribute of the request is in a format the server can handle. But having to repeat such preprocessing code isn't optimal. It would be useful to separate that kind of preprocessing from your actual application logic. This is where *request transforms* come in.

A request transform is a function that will be invoked with the request's *body* before it's sent out. The return value of the transform will replace the original request body.

> Transforms are not to be confused with *interceptors*, which we will cover later in this chapter.

One way to specify a request transform is to attach a `transformRequest` attribute to your request object:

test/http_spec.js


```
it('allows transforming requests with functions', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 42,
    transformRequest: function(data) {
      return '*' + data + '*';
    }
  });

  expect(requests[0].requestBody).toBe('*42*');
});
```

Transforms are applied in `$http`, where we invoke a helper function called `transformData` before sending the request. The helper function is given the request data and the `transformRequest` attribute's value. The return value is then used as the actual request data:

src/http.js

```
function $http(requestConfig) {
  var deferred = $q.defer();

  var config = _.extend({
    method: 'GET'
  }, requestConfig);
  config.headers = mergeHeaders(requestConfig);

  if (_.isUndefined(config.withCredentials) &&
      !_.isUndefined(defaults.withCredentials)) {
    config.withCredentials = defaults.withCredentials;
  }

  var reqData = transformData(config.data, config.transformRequest);

  if (_.isUndefined(reqData)) {
    _.forEach(config.headers, function(v, k) {
      if (k.toLowerCase() === 'content-type') {
        delete config.headers[k];
      }
    });
  }

  function done(status, response, headersString, statusText) {
    status = Math.max(status, 0);
    deferred[isSuccess(status) ? 'resolve' : 'reject']({
      status: status,
      data: response,
      statusText: statusText,
```



```
      headers: headersGetter(headersString),
      config: config
    });
    if (!$rootScope.$$phase) {
      $rootScope.$apply();
    }
  }

  $httpBackend(
    config.method,
    config.url,
    reqData,
    done,
    config.headers,
    config.withCredentials
  );
  return deferred.promise;
}
```

The `transformData` function invokes the transform if there is one. Otherwise it just returns the original request data:

src/http.js

```
function transformData(data, transform) {
  if (_.isFunction(transform)) {
    return transform(data);
  } else {
    return data;
  }
}
```

It is also possible to have a chain of *several* request transforms, which you can do by pointing the `transformRequest` attribute to an array of transforms. They will be invoked in order:

test/http_spec.js

```
it('allows multiple request transform functions', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 42,
    transformRequest: [function(data) {
      return '*' + data + '*';
    }, function(data) {
      return '-' + data + '-';
    }]
  });

  expect(requests[0].requestBody).toBe('-*42*-');
});
```


We can support this by *reducing* the request data in `transformData` with the array of transforms. We'll also rely on the fact that `_.reduce` returns the original value if there are *no* transforms given:

src/http.js

```
function transformData(data, transform) {
  if (_.isFunction(transform)) {
    return transform(data);
  } else {
    return _.reduce(transform, function(data, fn) {
      return fn(data);
    }, data);
  }
}
```

Attaching `transformRequest` in each request object can be useful, but arguably it is much more common to do it through the default configuration. If you attach transforms into `$http.defaults`, you can say "run this function for every request before it is sent", allowing for a much improved separation of concerns - you don't have to think about transforms every time that you make a request:

test/http_spec.js

```
it('allows settings transforms in defaults', function() {
  $http.defaults.transformRequest = [function(data) {
    return '*' + data + '*';
  }];
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 42
  });

  expect(requests[0].requestBody).toBe('*42*');
});
```

We can insert the default `transformRequest` when we construct the request config object:

src/http.js

```
function $http(requestConfig) {
  var deferred = $q.defer();

  var config = _.extend({
    method: 'GET',
    transformRequest: defaults.transformRequest
  }, requestConfig);
  config.headers = mergeHeaders(requestConfig);

  // ...
}
```


When you have default request transforms, they may need more information than just the request body to do their job - perhaps some transforms should only be applied when certain HTTP content type headers are present, for example. For this purpose, the transforms are also given the request headers as a second argument. They are wrapped in a function that takes header names and returns their values:

test/http_spec.js

```
it('passes request headers getter to transforms', function() {
  $http.defaults.transformRequest = [function(data, headers) {
    if (headers('Content-Type') === 'text/emphasized') {
      return '*' + data + '*';
    } else {
      return data;
    }
  }];
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 42,
    headers: {
      'content-type': 'text/emphasized'
    }
  });

  expect(requests[0].requestBody).toBe('*42*');
});
```

So we need to pass the request headers to `transformData`. Let's do exactly that, but let's also pass them through our `headersGetter` function first, because it can make the kind of header getter function we need. `headersGetter` does not yet know how to handle request headers, but we'll fix that in a moment.

src/http.js

```
var reqData = transformData(
  config.data,
  headersGetter(config.headers),
  config.transformRequest
);
```

In `transformData` we can now pass the headers to each of the individual request transforms:

src/http.js


```
function transformData(data, headers, transform) {
  if (_.isFunction(transform)) {
    return transform(data, headers);
  } else {
    return _.reduce(transform, function(data, fn) {
      return fn(data, headers);
    }, data);
  }
}
```

To complete the puzzle, we need to teach `headersGetter`, or more precisely the `parseHeaders` function it uses, what to do with the request headers we're giving to it.

Earlier we implemented `parseHeaders` so that it takes a response headers string and parses it into an object. With request headers we *already have* an object. The only thing we need to do is to "normalize" the headers, so that they can be accessed case-insensitively and so that any extra whitespace has been removed. This is what we'll do in `parseHeaders` if its argument is already an object:

src/http.js

```
function parseHeaders(headers) {
  if (_.isObject(headers)) {
    return _.transform(headers, function(result, v, k) {
      result[_.trim(k.toLowerCase())] = _.trim(v);
    }, {});
  } else {
    var lines = headers.split('\n');
    return _.transform(lines, function(result, line) {
      var separatorAt = line.indexOf(':');
      var name = _.trim(line.substr(0, separatorAt)).toLowerCase();
      var value = _.trim(line.substr(separatorAt + 1));
      if (name) {
        result[name] = value;
      }
    }, {});
  }
}
```

Response Transforms

Just as it is useful to transform requests before they're sent to a server, it can be useful to transform *responses* when they arrive from the server, before they are handed to application code. A typical use case for this would be parsing data from some serialization format to live JavaScript objects.

Response transforms work symmetrically with respect to request transforms. You can attach a `transformResponse` attribute to your request config, and it'll be invoked with the response body:

test/http_spec.js


```
it('allows transforming responses with functions', function() {
  var response;
  $http({
    url: 'http://teropa.info',
    transformResponse: function(data) {
      return '*' + data + '*';
    }
  }).then(function(r) {
    response = r;
  });

  requests[0].respond(200, {'Content-Type': 'text/plain'}, 'Hello');

  expect(response.data).toEqual('*Hello*');
});
```

Just like request transforms, response transforms get access to headers as a second argument. This time they are the *response* headers instead of the request headers though:

test/http_spec.js

```
it('passes response headers to transform functions', function() {
  var response;
  $http({
    url: 'http://teropa.info',
    transformResponse: function(data, headers) {
      if (headers('content-type') === 'text/decorated') {
        return '*' + data + '*';
      } else {
        return data;
      }
    }
  }).then(function(r) {
    response = r;
  });

  requests[0].respond(200, {'Content-Type': 'text/decorated'}, 'Hello');

  expect(response.data).toEqual('*Hello*');
});
```

Also, just like request transforms, response transforms can be set in the `$http` defaults so that you don't have to set them individually for all requests:

test/http_spec.js


```
it('allows setting default response transforms', function() {
  $http.defaults.transformResponse = [function(data) {
    return '*' + data + '*';
  }];
  var response;
  $http({
    url: 'http://teropa.info'
  }).then(function(r) {
    response = r;
  });

  requests[0].respond(200, {'Content-Type': 'text/plain'}, 'Hello');

  expect(response.data).toEqual('*Hello*');
});
```

Before we start getting these tests to green, let's take a moment to reorganize the code in `$http` a little bit. The `$http` function itself has gotten pretty large, and we should split it into two steps: *Preparing* the request and *sending* it. Let's keep the code for preparing the request in the `$http` function, but extract the code for sending it to a new function called `sendReq`:

src/http.js

```
function sendReq(config, reqData) {
  var deferred = $q.defer();

  function done(status, response, headersString, statusText) {
    status = Math.max(status, 0);
    deferred[isSuccess(status) ? 'resolve' : 'reject']({
      status: status,
      data: response,
      statusText: statusText,
      headers: headersGetter(headersString),
      config: config
    });
    if (!$rootScope.$$phase) {
      $rootScope.$apply();
    }
  }

  $httpBackend(
    config.method,
    config.url,
    reqData,
    done,
    config.headers,
    config.withCredentials
  );
```



```
  return deferred.promise;
}

function $http(requestConfig) {
  var config = _.extend({
    method: 'GET',
    transformRequest: defaults.transformRequest
  }, requestConfig);
  config.headers = mergeHeaders(requestConfig);

  if (_.isUndefined(config.withCredentials) &&
      !_.isUndefined(defaults.withCredentials)) {
    config.withCredentials = defaults.withCredentials;
  }

  var reqData = transformData(
    config.data,
    headersGetter(config.headers),
    config.transformRequest
  );

  if (_.isUndefined(reqData)) {
    _.forEach(config.headers, function(v, k) {
      if (k.toLowerCase() === 'content-type') {
        delete config.headers[k];
      }
    });
  }

  return sendReq(config, reqData);
}
```

It is now a little bit easier to attach response transformation to this code, so let's make a function whose job it is to do just that, and attach it as a Promise callback to the return value of `sendReq`:

src/http.js

```
function $http(requestConfig) {
  var config = _.extend({
    method: 'GET',
    transformRequest: defaults.transformRequest,
  }, requestConfig);
  config.headers = mergeHeaders(requestConfig);

  if (_.isUndefined(config.withCredentials) &&
      !_.isUndefined(defaults.withCredentials)) {
    config.withCredentials = defaults.withCredentials;
```

```
  }

  var reqData = transformData(
    config.data,
    headersGetter(config.headers),
    config.transformRequest
  );

  if (_.isUndefined(reqData)) {
    _.forEach(config.headers, function(v, k) {
      if (k.toLowerCase() === 'content-type') {
        delete config.headers[k];
      }
    });
  }

  function transformResponse(response) {
  }

  return sendReq(config, reqData)
    .then(transformResponse);
}
```

This function takes a response and replaces its `data` attribute with the result of running response transformers on it. We already have a function that can run transformers - `transformData` - and we'll reuse it here:

src/http.js

```
function transformResponse(response) {
  if (response.data) {
    response.data = transformData(response.data, response.headers,
      config.transformResponse);
  }
  return response;
}
```

We should also add the support for default response transforms, so that they get attached to `config.transformResponse`:

src/http.js

```
function $http(requestConfig) {
  var config = _.extend({
    method: 'GET',
    transformRequest: defaults.transformRequest,
    transformResponse: defaults.transformResponse
  }, requestConfig);

  // ...

}
```


We should be able to transform not only successful responses but also error responses, because they may also have bodies that need transforming:

test/http_spec.js

```
it('transforms error responses also', function() {
  var response;
  $http({
    url: 'http://teropa.info',
    transformResponse: function(data) {
      return '*' + data + '*';
    }
  }).catch(function(r) {
    response = r;
  });

  requests[0].respond(401, {'Content-Type': 'text/plain'}, 'Fail');

  expect(response.data).toEqual('*Fail*');
});
```

The `transformResponse` function is already perfectly capable of doing this, but since it is used as a promise handler, it needs to be able to re-reject failed responses. Otherwise the failures would be considered "caught" and application code would receive error responses in success handlers. Let's fix this:

src/http.js

```
function transformResponse(response) {
  if (response.data) {
    response.data = transformData(response.data, response.headers,
      config.transformResponse);
  }
  if (isSuccess(response.status)) {
    return response;
  } else {
    return $q.reject(response);
  }
}

return sendReq(config, reqData)
  .then(transformResponse, transformResponse);
```

The last aspect of response transformers is that they actually receive one additional argument that request transformers don't: The HTTP status code of the response. It is passed in as the third argument to each transformer.

test/http_spec.js


```
it('passes HTTP status to response transformers', function() {
  var response;
  $http({
    url: 'http://teropa.info',
    transformResponse: function(data, headers, status) {
      if (status === 401) {
        return 'unauthorized';
      } else {
        return data;
      }
    }
  }).catch(function(r) {
    response = r;
  });

  requests[0].respond(401, {'Content-Type': 'text/plain'}, 'Fail');

  expect(response.data).toEqual('unauthorized');
});
```

In `transformResponse` we can extract the status and pass it into `transformData`:

src/http.js

```
function transformResponse(response) {
  if (response.data) {
    response.data = transformData(
      response.data,
      response.headers,
      response.status,
      config.transformResponse
    );
  }
  if (isSuccess(response.status)) {
    return response;
  } else {
    return $q.reject(response);
  }
}
```

We've just added an additional argument to `transformData`, and that function is also called for request transforms, so we need to update that code to provide an explicit `undefined` status - requests don't have statuses:

src/http.js


```
var reqData = transformData(
  config.data,
  headersGetter(config.headers),
  undefined,
  config.transformRequest
);
```

In `transformData` we can now receive this argument and pass it right on to the transform functions:

src/http.js

```
function transformData(data, headers, status, transform) {
  if (_.isFunction(transform)) {
    return transform(data, headers, status);
  } else {
    return _.reduce(transform, function(data, fn) {
      return fn(data, headers, status);
    }, data);
  }
}
```

JSON Serialization And Parsing

For most Angular applications and for most of the time, both request and response data will be in JSON format. Because of this, Angular does what it can to make working with JSON easy: If your requests and responses are indeed in JSON, you never need to explicitly do any serialization or parsing, and can rely on the framework doing it for you.

For requests, this means that if you attach a JavaScript object as the request data, what goes into the actual request is a JSON-serialized representation of that object:

test/http_spec.js

```
it('serializes object data to JSON for requests', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: {aKey: 42}
  });

  expect(requests[0].requestBody).toBe('{"aKey":42}');
});
```

The same is true of arrays: A JavaScript array attached to a request is serialized into JSON.

test/http_spec.js


```
it('serializes array data to JSON for requests', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: [1, 'two', 3]
  });

  expect(requests[0].requestBody).toBe('[1,"two",3]');
});
```

Angular does this using the request transform feature we just implemented in the previous section. The default value for `transformRequest` holds a function that serializes the request data into JSON, if it is an Object (which includes Arrays):

src/http.js

```
var defaults = this.defaults = {
  headers: {
    common: {
      Accept: 'application/json, text/plain, */*'
    },
    post: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    put: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    patch: {
      'Content-Type': 'application/json;charset=utf-8'
    }
  },
  transformRequest: [function(data) {
    if (_.isObject(data)) {
      return JSON.stringify(data);
    } else {
      return data;
    }
  }]
};
```

There are a couple of very important exceptions to this rule though. If the response data is a Blob, presumably containing some raw binary or textual data, we shouldn't touch it, but send it out as-is and let the XMLHttpRequest deal with it:

test/http_spec.js


```
it('does not serialize blobs for requests', function() {
  var BlobBuilder = window.BlobBuilder || window.WebKitBlobBuilder ||
    window.MozBlobBuilder || window.MSBlobBuilder;
  var bb = new BlobBuilder();
  bb.append('hello');
  var blob = bb.getBlob('text/plain');
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: blob
  });

  expect(requests[0].requestBody).toBe(blob);
});
```

We should also skip JSON serialization for FormData objects. Like Blobs, FormData objects are something XMLHttpRequest already knows how to handle, and we shouldn't be trying to turn them into JSON:

test/http_spec.js

```
it('does not serialize form data for requests', function() {
  var formData = new FormData();
  formData.append('aField', 'aValue');
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: formData
  });

  expect(requests[0].requestBody).toBe(formData);
});
```

In our transformer we should guard the serialization call with a check to see if it is one of these objects. We also guard for a third type of object - File - though we don't have a unit test for it because, to be frank, constructing one is more trouble than it's worth:

src/http.js

```
transformRequest: [function(data) {
  if (_.isObject(data) && !isBlob(data) &&
      !isFile(data) && !isFormData(data)) {
    return JSON.stringify(data);
  } else {
    return data;
  }
}]
```

We are using three new helper function here, each of which looks at the String representation of the object and checks if its type is the one we're interested in:

src/http.js

```
function isBlob(object) {
  return object.toString() === '[object Blob]';
}
function isFile(object) {
  return object.toString() === '[object File]';
}
function isFormData(object) {
  return object.toString() === '[object FormData]';
}
```

This is all we need for JSON requests. The other half of $http's JSON support is responses: If the server indicates a JSON content type in the response, what you get as the response data is a JavaScript data structure parsed from the response.

test/http_spec.js

```
it('parses JSON data for JSON responses', function() {
  var response;
  $http({
    method: 'GET',
    url: 'http://teropa.info'
  }).then(function(r) {
    response = r;
  });
  requests[0].respond(
    200,
    {'Content-Type': 'application/json'},
    '{"message":"hello"}'
  );

  expect(_.isObject(response.data)).toBe(true);
  expect(response.data.message).toBe('hello');
});
```

Just like the requests, this is also done with a transform. Let's add a response transform function to the defaults:

src/http.js

```
var defaults = this.defaults = {
  headers: {
    common: {
      Accept: 'application/json, text/plain, */*'
```



```
    },
    post: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    put: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    patch: {
      'Content-Type': 'application/json;charset=utf-8'
    }
  },
  transformRequest: [function(data) {
    if (_.isObject(data) && !isBlob(data) &&
        !isFile(data) && !isFormData(data)) {
      return JSON.stringify(data);
    } else {
      return data;
    }
  }],
  transformResponse: [defaultHttpResponseTransform]
};
```

This is a function that - like any response transform - takes the response data and headers as arguments:

src/http.js

```
function defaultHttpResponseTransform(data, headers) {

}
```

What the function does is check if the response data is a String, and if the indicated content type is `application/json`. When those two conditions are true, the response data is parsed as JSON, and otherwise returned as-is:

src/http.js

```
function defaultHttpResponseTransform(data, headers) {
  if (_.isString(data)) {
    var contentType = headers('Content-Type');
    if (contentType && contentType.indexOf('application/json') === 0) {
      return JSON.parse(data);
    }
  }
  return data;
}
```

Angular actually attempts to be a little bit more clever than this though: It will try to parse responses as JSON if they *look like JSON*, even when the server fails to indicate their content type as JSON. So, for example, the following response data is parsed even though there is no Content-Type header in the response:

test/http_spec.js

```
it('parses a JSON object response without content type', function() {
  var response;
  $http({
    method: 'GET',
    url: 'http://teropa.info'
  }).then(function(r) {
    response = r;
  });
  requests[0].respond(200, {}, '{"message":"hello"}');

  expect(_.isObject(response.data)).toBe(true);
  expect(response.data.message).toBe('hello');
});
```

This also happens with arrays - a String representing a JSON array is parsed even without a content type:

test/http_spec.js

```
it('parses a JSON array response without content type', function() {
  var response;
  $http({
    method: 'GET',
    url: 'http://teropa.info'
  }).then(function(r) {
    response = r;
  });
  requests[0].respond(200, {}, '[1, 2, 3]');

  expect(_.isArray(response.data)).toBe(true);
  expect(response.data).toEqual([1, 2, 3]);
});
```

So, in our JSON response transformer, we should not only be looking at the content type, but also the data itself - does it look like JSON?

src/http.js


```
function defaultHttpResponseTransform(data, headers) {
  if (_.isString(data)) {
    var contentType = headers('Content-Type');
    if ((contentType && contentType.indexOf('application/json') === 0) ||
        isJsonLike(data)) {
      return JSON.parse(data);
    }
  }
  return data;
}
```

We could simply consider something that begins with a curly brace or a square bracket to be JSON-like:

src/http.js

```
function isJsonLike(data) {
  return data.match(/^\{/) || data.match(/^\[/);
}
```

Angular tries to be a little bit more smart about this though. A response that looks almost like valid JSON but isn't because the start and end characters are not the same, should not result in an error:

test/http_spec.js

```
it('does not choke on response resembling JSON but not valid', function() {
  var response;
  $http({
    method: 'GET',
    url: 'http://teropa.info'
  }).then(function(r) {
    response = r;
  });
  requests[0].respond(200, {}, '{1, 2, 3]');

  expect(response.data).toEqual('{1, 2, 3]');
});
```

Another case is a response string that begins with two curly braces - it kind of looks like JSON but isn't. The reason we take this special case into account is that `$http` is also used for loading Angular templates, and it is not uncommon for those to begin with an `{{interpolation expression}}`:

test/http_spec.js

```
it('does not try to parse interpolation expr as JSON', function() {
  var response;
  $http({
    method: 'GET',
    url: 'http://teropa.info'
  }).then(function(r) {
    response = r;
  });
  requests[0].respond(200, {}, '{{expr}}');

  expect(response.data).toEqual('{{expr}}');
});
```

Here's an updated test for JSON-likeness that takes these two cases into account. If the data begins with a curly brace, it should also end with a curly brace, and the same is true for square brackets. We also check that an opening curly brace is not immediately followed by another curly brace, using a lookahead expression:

src/http.js

```
function isJsonLike(data) {
  if (data.match(/^\{(?!\{)/)) {
    return data.match(/\}$/);
  } else if (data.match(/^\[/)) {
    return data.match(/\]$/);
  }
}
```

URL Parameters

We've seen how we can use $http to send information to a server in three parts: The request URL, the request headers, and the request body. The final way to attach information to an HTTP request that we'll look at is URL *query parameters*. That is, the key-value pairs that are attached to the URL after a `?` sign: `?a=1&b=2`.

Granted, using query parameters is already possible with our current implementation, since you can just attach them to the URL string you give to HTTP. But having to serialize your parameters and keep track of the separator characters is a bit cumbersome, so you'd probably rather have Angular do this for you. It will do that if you use the `params` attribute in the request configuration:

test/http_spec.js

```
it('adds params to URL', function() {
  $http({
    url: 'http://teropa.info',
    params: {
```



```
      a: 42
    }
  });

  expect(requests[0].url).toBe('http://teropa.info?a=42');
});
```

The implementation is smart enough to see if the URL string itself already had parameters, and just appends the `params` to them if so:

test/http_spec.js

```
it('adds additional params to URL', function() {
  $http({
    url: 'http://teropa.info?a=42',
    params: {
      b: 42
    }
  });

  expect(requests[0].url).toBe('http://teropa.info?a=42&b=42');
});
```

In `$http`, before we give the request to the HTTP backend, we'll now construct the request URL using two helper functions called `serializeParams` and `buildUrl`. The first of these takes the parameters of the request and serializes them into a string, and the second combines the request URL and the serialized params:

src/http.js

```
var url = buildUrl(config.url, serializeParams(config.params));

$httpBackend(
  config.method,
  url,
  reqData,
  done,
  config.headers,
  config.withCredentials
);
```

Let's go ahead and create these functions. The `serializeParams` function iterates over the parameters object and forms a string for each parameter. The string contains the key and value separated by an equals sign. When the function has gone over all the params, it joins up the parts with the `&` character:

src/http.js

```
function serializeParams(params) {
  var parts = [];
  _.forEach(params, function(value, key) {
    parts.push(key + '=' + value);
  });
  return parts.join('&');
}
```

The `buildUrl` function appends the serialized parameters into the given URL. It checks whether it should use the `?` or `&` delimiter based of what the URL string already contains:

src/http.js

```
function buildUrl(url, serializedParams) {
  if (serializedParams.length) {
    url += (url.indexOf('?') === -1) ? '?' : '&';
    url += serializedParams;
  }
  return url;
}
```

Some URL parameters will have characters in them that are not safe to append to a URL directly. This includes characters like `=` and `&` because they would be confused by the parameter separators themselves. For this reason, both the names and values of the parameters need to be *escaped* before they're appended:

test/http_spec.js

```
it('escapes url characters in params', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      '==': '&&'
    }
  });

  expect(requests[0].url).toBe('http://teropa.info?%3D%3D=%26%26');
});
```

We can use JavaScript's built-in encodeURIComponent function to do the escaping for us:

src/http.js

```
function serializeParams(params) {
  var parts = [];
  _.forEach(params, function(value, key) {
    parts.push(
      encodeURIComponent(key) + '=' + encodeURIComponent(value));
  });
  return parts.join('&');
}
```


> The actual AngularJS implementation does not use encodeURIComponent directly, but instead an internal utility called encodeUriQuery, which doesn't escape quite as much. For example, it leaves characters like @ and : unescaped.

If there are any parameters included whose values are `null` or `undefined`, they are left out of the resulting URL:

test/http_spec.js

```
it('does not attach null or undefined params', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      a: null,
      b: undefined
    }
  });

  expect(requests[0].url).toBe('http://teropa.info');
});
```

We can just add a check for these two values into the loop, and skip a parameter when its value matches:

src/http.js

```
function serializeParams(params) {
  var parts = [];
  _.forEach(params, function(value, key) {
    if (_.isNull(value) || _.isUndefined(value)) {
      return;
    }
    parts.push(
      encodeURIComponent(key) + '=' + encodeURIComponent(value));
  });
  return parts.join('&');
}
```

HTTP supports having multiple values for a given parameter name in query parameters. This is done by just repeating the parameter name for each of the values. Angular also supports having multiple values for a parameter, when you use an array as the parameter's value:

test/http_spec.js


```
it('attaches multiple params from arrays', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      a: [42, 43]
    }
  });

  expect(requests[0].url).toBe('http://teropa.info?a=42&a=43');
});
```

We should have an inner loop that iterates over each value inside our parameter loop. For simplicity's sake we'll do this for *all* parameters, and just wrap any parameter values that aren't already arrays:

src/http.js

```
function serializeParams(params) {
  var parts = [];
  _.forEach(params, function(value, key) {
    if (_.isNull(value) || _.isUndefined(value)) {
      return;
    }
    if (!_.isArray(value)) {
      value = [value];
    }
    _.forEach(value, function(v) {
      parts.push(
        encodeURIComponent(key) + '=' + encodeURIComponent(v));
    });
  });
  return parts.join('&');
}
```

So arrays have a special meaning when used as parameter values. But what happens to *objects*? There's no "nested parameter" support in HTTP, so what happens is that objects are just serialized into JSON (and then URL-escaped). So, it is in fact possible to transport some JSON in query parameters, if the server is prepared to deserialize it as such:

test/http_spec.js

```
it('serializes objects to json', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      a: {b: 42}
    }
  });

  expect(requests[0].url).toBe('http://teropa.info?a=%7B%22b%22%3A42%7D');
});
```


In our inner value loop, we should check if the value is an object and stringify it if so:

src/http.js

```
function serializeParams(params) {
  var parts = [];
  _.forEach(params, function(value, key) {
    if (_.isNull(value) || _.isUndefined(value)) {
      return;
    }
    if (!_.isArray(value)) {
      value = [value];
    }
    _.forEach(value, function(v) {
      if (_.isObject(v)) {
        v = JSON.stringify(v);
      }
      parts.push(
        url += encodeURIComponent(key) + '=' + encodeURIComponent(v));
    });
  });
  return parts.join('&');
}
```

> The JSON serialization support will also take care of Dates, because a JavaScript Date will turn into its ISO 8601 string representation when given to `JSON.stringify`.

This is how URL parameter serialization works by default. But as an application user you can actually also substitute this approach with your own. You may attach a `paramSerializer` key to the request configuration or the predefined `$http` defaults configuration. It should be a function that takes a params object and returns the serialized params string. It essentially has the same contract as our existing `serializeParams` function:

test/http_spec.js

```
it('allows substituting param serializer', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      a: 42,
      b: 43
    },
    paramSerializer: function(params) {
      return _.map(params, function(v, k) {
        return k + '=' + v + 'lol';
```



```
      }).join('&');
    }
  });

  expect(requests[0].url)
    .toEqual('http://teropa.info?a=42lol&b=43lol');
});
```

So, when building the URL we use the `paramSerializer` function of the request config instead of invoking the `serializeParams` function directly:

src/http.js

```
var url = buildUrl(config.url,
                   config.paramSerializer(config.params));

$httpBackend(
  config.method,
  url,
  reqData,
  done,
  config.headers,
  config.withCredentials
);
```

In the defaults we set the default `serializeParams` function we implemented earlier. That's what we want to use if nothing else is configured:

src/http.js

```
var defaults = this.defaults = {
  // ...
  paramSerializer: serializeParams
};
```

When forming each actual request config, we should also pull in the default param serializer:

src/http.js

```
function $http(requestConfig) {
  var config = _.extend({
    method: 'GET',
    transformRequest: defaults.transformRequest,
    transformResponse: defaults.transformResponse,
    paramSerializer: defaults.paramSerializer
  }, requestConfig);
  // ...
}
```


There's actually an even more convenient way to supply your own param serializer: You can make one available in the dependency injector and then just refer to its name in the request configuration. Here's a custom serializer defined with a factory:

test/http_spec.js

```
it('allows substituting param serializer through DI', function() {
  var injector = createInjector(['ng', function($provide) {
    $provide.factory('mySpecialSerializer', function() {
      return function(params) {
        return _.map(params, function(v, k) {
          return k + '=' + v + 'lol';
        }).join('&');
      };
    });
  }]);
  injector.invoke(function($http) {
    $http({
      url: 'http://teropa.info',
      params: {
        a: 42,
        b: 43
      },
      paramSerializer: 'mySpecialSerializer'
    });

    expect(requests[0].url)
      .toEqual('http://teropa.info?a=42lol&b=43lol');
  });
});
```

We're going to need the `$injector` in the `$http` service, so let's inject it in:

src/http.js

```
this.$get = ['$httpBackend', '$q', '$rootScope', '$injector',
             function($httpBackend, $q, $rootScope, $injector) {
  // ...
}];
```

In the `$http` function we can then use the `$injector` to obtain the param serializer function if what we have in the configuration is just a string:

src/http.js

```
function $http(requestConfig) {
  var config = _.extend({
    method: 'GET',
    transformRequest: defaults.transformRequest,
    transformResponse: defaults.transformResponse,
    paramSerializer: defaults.paramSerializer
  }, requestConfig);
  config.headers = mergeHeaders(requestConfig);
  if (_.isString(config.paramSerializer)) {
    config.paramSerializer = $injector.get(config.paramSerializer);
  }
  // ...
}
```

In fact, the default param serializer itself is available in the dependency injector, by the name of `$httpParamSerializer`. This means you can also use it for other purposes or decorate it:

test/http_spec.js

```
it('makes default param serializer available through DI', function() {
  var injector = createInjector(['ng']);
  injector.invoke(function($httpParamSerializer) {
    var result = $httpParamSerializer({a: 42, b: 43});
    expect(result).toEqual('a=42&b=43');
  });
});
```

In `http.js` we can change the `serializeParams` function so that it is no longer a top level function, but the return value of a new provider instead:

src/http.js

```
function $HttpParamSerializerProvider() {
  this.$get = function() {
    return function serializeParams(params) {
      var parts = [];
       _.forEach(params, function(value, key) {
        if (_.isNull(value) || _.isUndefined(value)) {
          return;
        }
        if (!_.isArray(value)) {
          value = [value];
        }
        _.forEach(value, function(v) {
          if (_.isObject(v)) {
            v = JSON.stringify(v);
          }
```



```
          parts.push(
            encodeURIComponent(key) + '=' + encodeURIComponent(v));
        });
      });
      return parts.join('&');
    };
  };
}
```

We will then register this provider into the `ng` module:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
  ngModule.provider('$httpBackend', $HttpBackendProvider);
  ngModule.provider('$http', $HttpProvider);
  ngModule.provider('$httpParamSerializer', $HttpParamSerializerProvider);
}
```

In the `$http` config defaults, we now refer to the name of the default serializer since we no longer have the standalone `serializeParams` function:

src/http.js

```
var defaults = this.defaults = {
  // ...
  paramSerializer: '$httpParamSerializer'
};
```

Angular ships with one alternative to the `$httpParamSerializer` by default: Instead of using the default, you can enable jQuery compatible serialization by using the `$httpParamSerializerJQLike` serializer. This may be useful if you have an existing backend that has been built to consume jQuery-serialized forms, or if you just need to send nested data structures but can't use JSON.

As we'll soon see, it serializes collections in a special way, but for primitives it behaves exactly like the default serializer:

test/http_spec.js


```
describe('JQ-like param serialization', function() {

  it('is possible', function() {
    $http({
      url: 'http://teropa.info',
      params: {
        a: 42,
        b: 43
      },
      paramSerializer: '$httpParamSerializerJQLike'
    });

    expect(requests[0].url).toEqual('http://teropa.info?a=42&b=43');
  });

});
```

This serializer is defined by another provider in `http.js`:

src/http.js

```
function $HttpParamSerializerJQLikeProvider() {
  this.$get = function() {
    return function(params) {
      var parts = [];
      _.forEach(params, function(value, key) {
        parts.push(
          encodeURIComponent(key) + '=' + encodeURIComponent(value));
      });
      return parts.join('&');
    };
  };
}
```

This provider too is registered into the `ng` module:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
```



```
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
  ngModule.provider('$httpBackend', $HttpBackendProvider);
  ngModule.provider('$http', $HttpProvider);
  ngModule.provider('$httpParamSerializer', $HttpParamSerializerProvider);
  ngModule.provider('$httpParamSerializerJQLike',
    $HttpParamSerializerJQLikeProvider);
}
```

The values `null` and `undefined` are also skipped by this serializer:

src/http.js

```
function $HttpParamSerializerJQLikeProvider() {
  this.$get = function() {
    return function(params) {
      var parts = [];
      _.forEach(params, function(value, key) {
        if (_.isNull(value) || _.isUndefined(value)) {
          return;
        }
        parts.push(
          encodeURIComponent(key) + '=' + encodeURIComponent(value));
      });
      return parts.join('&');
    };
  };
}
```

Where this serializer starts to differ from the default is when we look at how it deals with arrays. It appends a square bracket suffix `[]` to the param names that originate from an array value:

test/http_spec.js

```
it('uses square brackets in arrays', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      a: [42, 43]
    },
    paramSerializer: '$httpParamSerializerJQLike'
  });

  expect(requests[0].url).toEqual('http://teropa.info?a%5B%5D=42&a%5B%5D=43');
});
```

The opening square bracket becomes `%5B` when URL-encoded, and the closing square bracket becomes `%5D`.

Here's how we can handle the array case:

src/http.js

```
function $HttpParamSerializerJQLikeProvider() {
  this.$get = function() {
    return function(params) {
      var parts = [];
      _.forEach(params, function(value, key) {
        if (_.isNull(value) || _.isUndefined(value)) {
          return;
        }
        if (_.isArray(value)) {
          _.forEach(value, function(v) {
            parts.push(
              encodeURIComponent(key + '[]') + '=' +  encodeURIComponent(v));
          });
        } else {
          parts.push(
            encodeURIComponent(key) + '=' + encodeURIComponent(value));
        }
      });
      return parts.join('&');
    };
  };
}
```

When serializing objects, square brackets are also used. The difference to arrays is that here the key the key used in the object is put in between the square brackets:

test/http_spec.js

```
it('uses square brackets in objects', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      a: {b: 42, c: 43}
    },
    paramSerializer: '$httpParamSerializerJQLike'
  });

  expect(requests[0].url).toEqual('http://teropa.info?a%5Bb%5D=42&a%5Bc%5D=43');
});
```

We'll handle object in its own `else if` branch. While Dates are also objects, we want to handle them as primitives from the serialization point of view. For that reason we add a special check for them:

src/http.js


```
function $HttpParamSerializerJQLikeProvider() {
  this.$get = function() {
    return function(params) {
      var parts = [];
      _.forEach(params, function(value, key) {
        if (_.isNull(value) || _.isUndefined(value)) {
          return;
        }
        if (_.isArray(value)) {
          _.forEach(value, function(v) {
            parts.push(
              encodeURIComponent(key + '[]') + '=' + encodeURIComponent(v));
          });
        } else if (_.isObject(value) && !_.isDate(value)) {
          _.forEach(value, function(v, k) {
            parts.push(
              encodeURIComponent(key + '[' + k + ']') + '=' +
              encodeURIComponent(v));
          });
        } else {
          parts.push(
            encodeURIComponent(key) + '=' + encodeURIComponent(value));
        }
      });
      return parts.join('&');
    };
  };
}
```

These square bracket prefixes also work recursively, so that when you have nested objects, the square brackets are repeated for each one, so that `{a: {b: {c: 42}}}` becomes `a[b][c]=42`:

test/http_spec.js

```
it('supports nesting in objects', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      a: {b: {c: 42}}
    },
    paramSerializer: '$httpParamSerializerJQLike'
  });

  expect(requests[0].url).toEqual('http://teropa.info?a%5Bb%5D%5Bc%5D=42');
});
```


We're going to need to make our implementation recusive so that we can support arbitrary depths of nesting. Let's reorganize our implementation so that it uses an internal `serialize` function that always takes a prefix to prepend to the key, and may recusively invoke itself to append more information to the prefix. Nothing is actually added to the `parts` array until a leaf-level value (i.e. something that is not an array or an object) is reached:

src/http.js

```
function $HttpParamSerializerJQLikeProvider() {
  this.$get = function() {
    return function(params) {
      var parts = [];

      function serialize(value, prefix) {
        if (_.isNull(value) || _.isUndefined(value)) {
          return;
        }
        if (_.isArray(value)) {
          _.forEach(value, function(v) {
            serialize(v, prefix + '[]');
          });
        } else if (_.isObject(value) && !_.isDate(value)) {
          _.forEach(value, function(v, k) {
            serialize(v, prefix + '[' + k + ']');
          });
        } else {
          parts.push(
            encodeURIComponent(prefix) + '=' +  encodeURIComponent(value));
        }
      }

      _.forEach(params, function(value, key) {
        if (_.isNull(value) || _.isUndefined(value)) {
          return;
        }
        if (_.isArray(value)) {
          _.forEach(value, function(v) {
            serialize(v, key + '[]');
          });
        } else if (_.isObject(value) && !_.isDate(value)) {
          _.forEach(value, function(v, k) {
            serialize(v, key + '[' + k + ']');
          });
        } else {
          parts.push(
            encodeURIComponent(key) + '=' +
            encodeURIComponent(value));
        }
      });
      return parts.join('&');
    };
```



```
  };
}
```

This works quite nicely, but we are now duplicating a lot of code here. We have almost the same logic repeated once for the top-level params object and once for nested values. The only difference between the two cases is that the square bracket syntax is *not* used on the top level. If we introduce a `topLevel` flag to `serialize` instead, we can get rid of the duplication. We can just pass the top-level params object to `serialize` and set the flag to `true`. When the flag is `true`, we don't append the square brackets to object keys:

src/http.js

```
function $HttpParamSerializerJQLikeProvider() {
  this.$get = function() {
    return function(params) {
      var parts = [];

      function serialize(value, prefix, topLevel) {
        if (_.isNull(value) || _.isUndefined(value)) {
          return;
        }
        if (_.isArray(value)) {
          _.forEach(value, function(v) {
            serialize(v, prefix + '[]');
          });
        } else if (_.isObject(value) && !_.isDate(value)) {
          _.forEach(value, function(v, k) {
            serialize(v, prefix +
                          (topLevel ? '' : '[') +
                          k +
                          (topLevel ? '' : ']'));
          });
        } else {
          parts.push(
            encodeURIComponent(prefix) + '=' + encodeURIComponent(value));
        }
      }

      serialize(params, '', true);

      return parts.join('&');
    };
  };
}
```

Finally, if the items contained in an array are objects (or nested arrays) themselves, we emit the array index between the square brackets, so that `{a: [{b: 42}]}` does not become `a[][b]=42` but `a[0][b]=42`:

test/http_spec.js


```
it('appends array indexes when items are objects', function() {
  $http({
    url: 'http://teropa.info',
    params: {
      a: [{b: 42}]
    },
    paramSerializer: '$httpParamSerializerJQLike'
  });

  expect(requests[0].url).toEqual('http://teropa.info?a%5B0%5D%5Bb%5D=42');
});
```

We can use the loop index to put between the square brackets in this case:

src/http.js

```
function $HttpParamSerializerJQLikeProvider() {
  this.$get = function() {
    return function(params) {
      var parts = [];

      function serialize(value, prefix, topLevel) {
        if (_.isNull(value) || _.isUndefined(value)) {
          return;
        }
        if (_.isArray(value)) {
          _.forEach(value, function(v, i) {
            serialize(v, prefix +
                         '[' +
                         (_.isObject(v) ? i : '') +
                         ']');
          });
        } else if (_.isObject(value)) {
          _.forEach(value, function(v, k) {
            serialize(v, prefix +
                         (topLevel ? '' : '[') +
                         k +
                         (topLevel ? '' : ']'));
          });
        } else {
          parts.push(
            encodeURIComponent(prefix) + '=' + encodeURIComponent(value));
        }
      }

      serialize(params, '', true);

      return parts.join('&');
    };
  };
}
```


Shorthand Methods

We've pretty much covered everything about $http that affects how the actual HTTP responses are made, and for the remainder of the chapter we'll focus on a few things that relate to the API $http provides to your application, and the asynchronous Promise workflow that takes place with requests.

One of the conveniences $http provides to applications comes in the form of a few shorthand methods that enable making requests in a more streamlined way than using the raw `$http` function. For instance, there is a method called `get` that takes a request URL and an optional request config, and issues a GET request:

test/http_spec.js

```
it('supports shorthand method for GET', function() {
  $http.get('http://teropa.info', {
    params: {q: 42}
  });

  expect(requests[0].url).toBe('http://teropa.info?q=42');
  expect(requests[0].method).toBe('GET');
});
```

The method calls the lower-level `$http` function, making sure that the configuration object has the given URL and the GET method attached:

src/http.js

```
$http.defaults = defaults;
$http.get = function(url, config) {
  return $http(_.extend(config || {}, {
    method: 'GET',
    url: url
  }));
};
return $http;
```

The exact same kind of shorthand method is also provided for HEAD and DELETE requests:

test/http_spec.js


```
it('supports shorthand method for HEAD', function() {
  $http.head('http://teropa.info', {
    params: {q: 42}
  });

  expect(requests[0].url).toBe('http://teropa.info?q=42');
  expect(requests[0].method).toBe('HEAD');
});

it('supports shorthand method for DELETE', function() {
  $http.delete('http://teropa.info', {
    params: {q: 42}
  });

  expect(requests[0].url).toBe('http://teropa.info?q=42');
  expect(requests[0].method).toBe('DELETE');
});
```

The implementations for these methods are also exactly the same as for GET - with the exception of the method HTTP method that's configured:

src/http.js

```
$http.head = function(url, config) {
  return $http(_.extend(config || {}, {
    method: 'HEAD',
    url: url
  }));
};
$http.delete = function(url, config) {
  return $http(_.extend(config || {}, {
    method: 'DELETE',
    url: url
  }));
};
```

In fact, the three methods are so similar that we might as well generate them in a loop, to avoid all that repetition in the code:

src/http.js

```
$http.defaults = defaults;
_.forEach(['get', 'head', 'delete'], function(method) {
  $http[method] = function(url, config) {
    return $http(_.extend(config || {}, {
      method: method.toUpperCase(),
      url: url
    }));
  };
});
return $http;
```


There are three more HTTP methods for which a shorthand method is provided: POST, PUT, and PATCH. The difference with these three to the previous three is that they support *request bodies*, which is to say you can set the `data` attribute on the requests. So this time the shorthand methods take three arguments: The URL, the (optional) request data, and the (optional) request configuration object:

test/http_spec.js

```
it('supports shorthand method for POST with data', function() {
  $http.post('http://teropa.info', 'data', {
    params: {q: 42}
  });

  expect(requests[0].url).toBe('http://teropa.info?q=42');
  expect(requests[0].method).toBe('POST');
  expect(requests[0].requestBody).toBe('data');
});

it('supports shorthand method for PUT with data', function() {
  $http.put('http://teropa.info', 'data', {
    params: {q: 42}
  });

  expect(requests[0].url).toBe('http://teropa.info?q=42');
  expect(requests[0].method).toBe('PUT');
  expect(requests[0].requestBody).toBe('data');
});

it('supports shorthand method for PATCH with data', function() {
  $http.patch('http://teropa.info', 'data', {
    params: {q: 42}
  });

  expect(requests[0].url).toBe('http://teropa.info?q=42');
  expect(requests[0].method).toBe('PATCH');
  expect(requests[0].requestBody).toBe('data');
});
```

Let's generate these methods in another loop just below the one we added for GET, HEAD, and DELETE:

src/http.js

```
_.forEach(['post', 'put', 'patch'], function(method) {
  $http[method] = function(url, data, config) {
    return $http(_.extend(config || {}, {
      method: method.toUpperCase(),
```



```
      url: url,
      data: data
    }));
  };
});
```

Interceptors

Earlier in the chapter we discussed request and response transforms and how they allow modifying request and response bodies, mostly for the purposes of serialization and deserialization. Now we'll turn our attention to another feature that allows similar, generic processing steps to be attached to HTTP requests and responses: Interceptors.

Interceptors are a more high-level and fully-featured API than transforms, and really allow for any arbitrary execution logic to be attached to HTTP request and response processing. With interceptors you can freely modify or replace requests and responses. Since interceptors are Promise-based, you can do asynchronous work in them - something you cannot do with transforms.

Interceptors are created by factory functions. To register an interceptor, you need to append its factory function into the `interceptors` array held by `$httpProvider`. This means interceptor registration must happen at configuration time. Once the `$http` service is created, all registered interceptor factory functions are invoked:

test/http_spec.js

```
it('allows attaching interceptor factories', function() {
  var interceptorFactorySpy = jasmine.createSpy();
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.interceptors.push(interceptorFactorySpy);
  }]);
  $http = injector.get('$http');

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

We'll set up this array in the `$HttpProvider` constructor, and expose it through the `interceptors` attribute:

src/http.js

```
function $HttpProvider() {

  var interceptorFactories = this.interceptors = [];

  // ...

}
```


Then, when `$http` itself is created (when `$httpProvider.$get` is invoked), we'll call all registered factories, which gives us an array of all the interceptors:

src/http.js

```
this.$get = ['$httpBackend', '$q', '$rootScope', '$injector'
              function($httpBackend, $q, $rootScope, $injector) {

  var interceptors = _.map(interceptorFactories, function(fn) {
    return fn();
  });

  // ...

}];
```

Interceptor factories may also be integrated to the dependency injection system. If a factory function has arguments, those arguments are injected. The factory may also be wrapped in an array-style dependency injection wrapper `['a', 'b', function(a, b) { }]`. Here we have an interceptor factory that has a dependency to `$rootScope`:

test/http_spec.js

```
it('uses DI to instantiate interceptors', function() {
  var interceptorFactorySpy = jasmine.createSpy();
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.interceptors.push(['$rootScope', interceptorFactorySpy]);
  }]);
  $http = injector.get('$http');
  var $rootScope = injector.get('$rootScope');
  expect(interceptorFactorySpy).toHaveBeenCalledWith($rootScope);
});
```

We'll use the injector's `invoke` method to instantiate all interceptors, instead of calling the factories directly:

src/http.js

```
this.$get = ['$httpBackend', '$q', '$rootScope', '$injector',
              function($httpBackend, $q, $rootScope, $injector) {

  var interceptors = _.map(interceptorFactories, function(fn) {
    return $injector.invoke(fn);
  });

  // ...

}];
```


So far we have been appending interceptor factories directly into the `$httpProvider.interceptors` array, but there's also another way to register an interceptor. You can first register a regular Angular factory, and then push its *name* into `$httpProvider.interceptors`. This makes it easier to treat interceptors as first-class Angular components - "it's just a factory":

test/http_spec.js

```
it('allows referencing existing interceptor factories', function() {
  var interceptorFactorySpy = jasmine.createSpy().and.returnValue({});
  var injector = createInjector(['ng', function($provide, $httpProvider) {
    $provide.factory('myInterceptor', interceptorFactorySpy);
    $httpProvider.interceptors.push('myInterceptor');
  }]);
  $http = injector.get('$http');

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

Upon interceptor creation, we must now check whether an interceptor has been registered as a string or as a function. If it's a string, we can obtain the corresponding interceptor using `$injector.get` (which will invoke the factory), and if it's a function we can just invoke it like before:

src/http.js

```
var interceptors = _.map(interceptorFactories, function(fn) {
  return _.isString(fn) ? $injector.get(fn) :
                          $injector.invoke(fn);
});
```

Now that we know how interceptors are registered, we can start talking about what they actually are and how they are integrated into the `$http`'s request processing.

Interceptors make heavy use of Promises. They are attached to `$http`'s processing logic as Promise handlers, and they may also return Promises. We already have Promise integration in `$http` (because it returns a Promise), but before we can integrate interceptors to it, we need to reorganize things a little bit.

First of all, some of the code we currently have in the `$http` function will need to run *before* any interceptors, and some after. The part that runs after interceptors will eventually need to run in a Promise callback, but let's first just extract it to a new function, which we'll call `serverRequest`. No new behavior is added at this point - the code is just moved from one function to another. Where we draw the division is just after the `config` object has been created and the headers attached to it:

src/http.js


```
function serverRequest(config) {
  if (_.isUndefined(config.withCredentials) &&
      !_.isUndefined(defaults.withCredentials)) {
    config.withCredentials = defaults.withCredentials;
  }

  var reqData = transformData(
    config.data,
    headersGetter(config.headers),
    undefined,
    config.transformRequest
  );

  if (_.isUndefined(reqData)) {
    _.forEach(config.headers, function(v, k) {
      if (k.toLowerCase() === 'content-type') {
        delete config.headers[k];
      }
    });
  }

  function transformResponse(response) {
    if (response.data) {
      response.data = transformData(
        response.data,
        response.headers,
        response.status,
        config.transformResponse
      );
    }
    if (isSuccess(response.status)) {
      return response;
    } else {
      return $q.reject(response);
    }
  }

  return sendReq(config, reqData)
    .then(transformResponse, transformResponse);
}

function $http(requestConfig) {
  var config = _.extend({
    method: 'GET',
    transformRequest: defaults.transformRequest,
    transformResponse: defaults.transformResponse,
    paramSerializer: defaults.paramSerializer
  }, requestConfig);
  if (_.isString(config.paramSerializer)) {
    config.paramSerializer = $injector.get(config.paramSerializer);
  }
```



```
  config.headers = mergeHeaders(requestConfig);

  return serverRequest(config);
}
```

Next, let's tweak the way we create the `$http` Promise. Previously we just let `sendReq` create the Promise that's eventually returned, but now we'll create one from the `config` object right in the `$http` function, and then invoke `serverRequest` as a *Promise handler*. This also does not yet alter the behavior of `$http`, but will make it easier to attach interceptors, which we'll be doing momentarily:

src/http.js

```
function $http(requestConfig) {
  var config = _.extend({
    method: 'GET',
    transformRequest: defaults.transformRequest,
    transformResponse: defaults.transformResponse,
    paramSerializer: defaults.paramSerializer
  }, requestConfig);
  if (_.isString(config.paramSerializer)) {
    config.paramSerializer = $injector.get(config.paramSerializer);
  }
  config.headers = mergeHeaders(requestConfig);

  var promise = $q.when(config);
  return promise.then(serverRequest);
}
```

Though we haven't really changed the behavior of $http yet, you may have noticed that this change breaks a large number of tests. That's because with the way we're now executing the request, *it isn't actually sent until the next digest after `$http` is called*. The reason is that `serverRequest` is invoked in a Promise callback, and Promise callbacks are only executed during digests.

This is indeed exactly what happens in Angular, but it does mean we need to make changes to our tests. Basically, in each existing `$http` test case, we need to invoke `$rootScope.$apply()` after we've invoked `$http()` before we can make any assertions about the request.

In order to call `$apply` we first need to get a handle of `$rootScope` at test setup:

test/http_spec.js

```
var $http, $rootScope;
var xhr, requests;

beforeEach(function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  $http = injector.get('$http');
  $rootScope = injector.get('$rootScope');
});
```


Then we need to add a `$rootScope.$apply()` call after each `$http` call. Here's an example of the first test case where it's needed:

test/http_spec.js

```
it('makes an XMLHttpRequest to given URL', function() {
  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: 'hello'
  });
  $rootScope.$apply();
  expect(requests.length).toBe(1);
  expect(requests[0].method).toBe('POST');
  expect(requests[0].url).toBe('http://teropa.info');
  expect(requests[0].async).toBe(true);
  expect(requests[0].requestBody).toBe('hello');
});
```

Go ahead and repeat this for each test case in `http_spec.js`. Note that some of the test cases use their own injector, so for them we need to get the `$rootScope` from that injector as well. Otherwise we would be calling an unrelated `$rootScope`:

test/http_spec.js

```
it('exposes default headers through provider', function() {
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.defaults.headers.post['Content-Type'] =
      'text/plain;charset=utf-8';
  }]);
  $http = injector.get('$http');
  $rootScope = injector.get('$rootScope');

  $http({
    method: 'POST',
    url: 'http://teropa.info',
    data: '42'
  });
  $rootScope.$apply();

  expect(requests.length).toBe(1);
  expect(requests[0].requestHeaders['Content-Type']).toBe(
    'text/plain;charset=utf-8');
});

// ...
```



```
it('allows substituting param serializer through DI', function() {
  var injector = createInjector(['ng', function($provide) {
    $provide.factory('mySpecialSerializer', function() {
      return function(params) {
        return _.map(params, function(v, k) {
          return k + '=' + v + 'lol';
        }).join('&');
      };
    });
  }]);
  injector.invoke(function($http, $rootScope) {
    $http({
      url: 'http://teropa.info',
      params: {
        a: 42,
        b: 43
      },
      paramSerializer: 'mySpecialSerializer'
    });
    $rootScope.$apply();

    expect(requests[0].url)
      .toEqual('http://teropa.info?a=42lol&b=43lol');
  });
});
```

And finally we are ready to go ahead with the interceptor implementation!

Interceptors are objects that have one or more of four keys: `request`, `requestError`, `response`, and `responseError`. The values of those keys are functions that get called at different points during the processing of an HTTP request.

The first key we'll look at is `request`. If an interceptor defines a `request` method, it will get called *before a request is sent out*, and is expected to return a *modified request*. That means it can basically transform or replace the request before it's sent - similar to what transforms do:

test/http_spec.js

```
it('allows intercepting requests', function() {
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.interceptors.push(function() {
      return {
        request: function(config) {
          config.params.intercepted = true;
          return config;
        }
      };
    });
  }]);
```



```
  $http = injector.get('$http');
  $rootScope = injector.get('$rootScope');

  $http.get('http://teropa.info', {params: {}});
  $rootScope.$apply();
  expect(requests[0].url).toBe('http://teropa.info?intercepted=true');
});
```

The interceptor function may also return a *Promise for* a modified request, which means that whatever it does, it can also do asynchronously. This feature alone makes interceptors much more powerful than transforms:

test/http_spec.js

```
it('allows returning promises from request intercepts', function() {
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.interceptors.push(function($q) {
      return {
        request: function(config) {
          config.params.intercepted = true;
          return $q.when(config);
        }
      };
    });
  }]);
  $http = injector.get('$http');
  $rootScope = injector.get('$rootScope');

  $http.get('http://teropa.info', {params: {}});
  $rootScope.$apply();
  expect(requests[0].url).toBe('http://teropa.info?intercepted=true');
});
```

With the way we've now set up things in `$http`, integrating request interceptors is actually quite simple. Each one is just another link in the Promise handler chain, with the actual request sending being the final link:

src/http.js

```
var promise = $q.when(config);
_.forEach(interceptors, function(interceptor) {
  promise = promise.then(interceptor.request);
});
return promise.then(serverRequest);
```

Response interceptors work similarly to request interceptors - their argument is just the response instead of the request. They can modify or replace the response and their return value will be used as the response for further interceptors (and finally for application code):

test/http_spec.js


```
it('allows intercepting responses', function() {
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.interceptors.push(_.constant({
      response: function(response) {
        response.intercepted = true;
        return response;
      }
    }));
  }]);
  $http = injector.get('$http');
  $rootScope = injector.get('$rootScope');

  var response;
  $http.get('http://teropa.info').then(function(r) {
    response = r;
  });
  $rootScope.$apply();

  requests[0].respond(200, {}, 'Hello');
  expect(response.intercepted).toBe(true);
});
```

We can add the response interceptors by continuing the promise chain *after* the `serverRequest` callback has been attached. Another difference to request interceptors is that this time we iterate the interceptors in *reverse order*, where interceptors that were registered last get invoked first. This makes sense if you think about the request-response cycle: When we're handling the response, we're coming *back up* the interceptor chain.

src/http.js

```
var promise = $q.when(config);
_.forEach(interceptors, function(interceptor) {
  promise = promise.then(interceptor.request);
});
promise = promise.then(serverRequest);
_.forEachRight(interceptors, function(interceptor) {
  promise = promise.then(interceptor.response);
});
return promise;
```

The final two methods supported by interceptors have to do with error handling. The `requestError` methods are invoked when something goes wrong *before* the request is actually sent out, which is to say, there's an error in one of the preceding interceptors:

test/http_spec.js

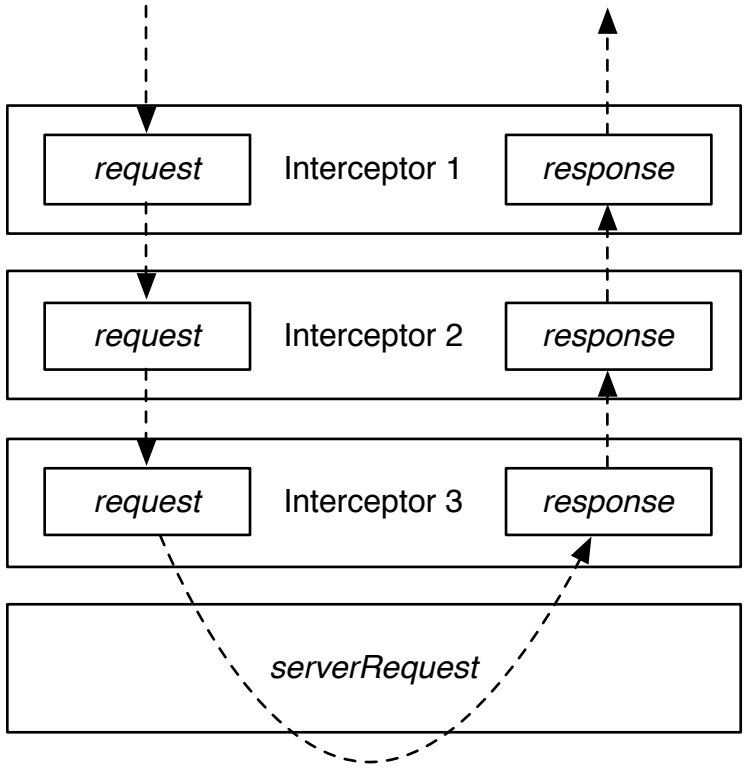

Figure 14.1: Interceptors are invoked in order for request and in reverse order for responses


```
it('allows intercepting request errors', function() {
  var requestErrorSpy = jasmine.createSpy();
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.interceptors.push(_.constant({
      request: function(config) {
        throw 'fail';
      }
    }));
    $httpProvider.interceptors.push(_.constant({
      requestError: requestErrorSpy
    }));
  }]);
  $http = injector.get('$http');
  $rootScope = injector.get('$rootScope');

  $http.get('http://teropa.info');
  $rootScope.$apply();

  expect(requests.length).toBe(0);
  expect(requestErrorSpy).toHaveBeenCalledWith('fail');
});
```

We can integrate `requestError` interceptors at the same time as we are integrating `request` interceptors. `requestError` functions are added as *failure callbacks* to the Promise chain:

src/http.js

```
var promise = $q.when(config);
_.forEach(interceptors, function(interceptor) {
  promise = promise.then(interceptor.request, interceptor.requestError);
});
promise = promise.then(serverRequest);
_.forEachRight(interceptors, function(interceptor) {
  promise = promise.then(interceptor.response);
});
return promise;
```

Response error interceptors will catch errors that happen after we have an HTTP response. Just like `response` interceptors, they are invoked in *reverse order*, so they receive errors from either the actual HTTP response, or from response interceptors registered *before* them:

test/http_spec.js

```
it('allows intercepting response errors', function() {
  var responseErrorSpy = jasmine.createSpy();
  var injector = createInjector(['ng', function($httpProvider) {
    $httpProvider.interceptors.push(_.constant({
```



```
      responseError: responseErrorSpy
    }));
    $httpProvider.interceptors.push(_.constant({
      response: function() {
        throw 'fail';
      }
    }));
  }]);
  $http = injector.get('$http');
  $rootScope = injector.get('$rootScope');

  $http.get('http://teropa.info');
  $rootScope.$apply();

  requests[0].respond(200, {}, 'Hello');
  $rootScope.$apply();

  expect(responseErrorSpy).toHaveBeenCalledWith('fail');
});
```

Registering `responseError` interceptors is completely symmetrical with registering `requestError` interceptors: They're added as failure callbacks at the same time when `response` interceptors are registered:

src/http.js

```
var promise = $q.when(config);
_.forEach(interceptors, function(interceptor) {
  promise = promise.then(interceptor.request, interceptor.requestError);
});
promise = promise.then(serverRequest);
_.forEachRight(interceptors, function(interceptor) {
  promise = promise.then(interceptor.response, interceptor.responseError);
});
return promise;
```

There we have a complete implementation of interceptors. There's a lot of power in them, but using them effectively requires an understanding of how they integrate to the Promise chain used by `$http`. We now have that understanding.

Promise Extensions

As a user of `$http`, you may have noticed that the Promises it returns have some methods on them that we did not implement in the previous chapter. Those methods are `success` and `error`, and they are in fact methods that *only* exist in Promises you get from `$http`.

The purpose of these extensions if to make it a little bit easier to work with HTTP responses. With a normal `then` or `catch` handler, what you get as the argument is the full response object. What `success` gives you instead is the response unpacked into four separate arguments: The response data, the status code, the headers, and the original request configuration:

test/http_spec.js

```
it('allows attaching success handlers', function() {
  var data, status, headers, config;
  $http.get('http://teropa.info').success(function(d, s, h, c) {
    data = d;
    status = s;
    headers = h;
    config = c;
  });
  $rootScope.$apply();

  requests[0].respond(200, {'Cache-Control': 'no-cache'}, 'Hello');
  $rootScope.$apply();

  expect(data).toBe('Hello');
  expect(status).toBe(200);
  expect(headers('Cache-Control')).toBe('no-cache');
  expect(config.method).toBe('GET');
});
```

While this isn't a hugely important feature, it can be convenient, especially if all you care about is the response body - you can just declare a single-argument `success` handler and you don't need to care about the format of the response object.

A completely identical extension is also available for error responses. That's the `error` handler:

test/http_spec.js

```
it('allows attaching error handlers', function() {
  var data, status, headers, config;
  $http.get('http://teropa.info').error(function(d, s, h, c) {
    data = d;
    status = s;
    headers = h;
    config = c;
  });
  $rootScope.$apply();

  requests[0].respond(401, {'Cache-Control': 'no-cache'}, 'Fail');
  $rootScope.$apply();

  expect(data).toBe('Fail');
```



```
  expect(status).toBe(401);
  expect(headers('Cache-Control')).toBe('no-cache');
  expect(config.method).toBe('GET');
});
```

These two extensions are attached to the final `Promise` object that we get after all the interceptors have been applied. Each of them attaches a normal `then` or `catch` handler to the promise, and includes a callback that unpacks the response into the separate arguments we saw in our test cases:

src/http.js

```
var promise = $q.when(config);
_.forEach(interceptors, function(interceptor) {
  promise = promise.then(interceptor.request, interceptor.requestError);
});
promise = promise.then(serverRequest);
_.forEachRight(interceptors, function(interceptor) {
  promise = promise.then(interceptor.response, interceptor.responseError);
});
promise.success = function(fn) {
  promise.then(function(response) {
    fn(response.data, response.status, response.headers, config);
  });
  return promise;
};
promise.error = function(fn) {
  promise.catch(function(response) {
    fn(response.data, response.status, response.headers, config);
  });
  return promise;
};
return promise;
```

Request Timeouts

With network requests, sometimes things just take too long: A server may take a very long time to respond, or you may get into a situation where the server *never* responds because of some network hiccup.

Browsers, servers, and proxies often have built-mechanisms for dealing with timeouts, but you may also want to have some control in your application into how long you're prepared to wait for a response. Angular has some features that help with this.

Firstly, you can attach a `timeout` attribute on your request configuration object, and set a Promise as the value of that attribute. Angular will then abort the request if the Promise resolves before a response is received.

This is quite a powerful feature, since you can control when a request times out based on your application logic: For example, the user might navigate to another route, or close a dialog, and you can arrange things so that the application isn't waiting for responses that will never be used.

test/http_spec.js

```
it('allows aborting a request with a Promise', function() {
  var timeout = $q.defer();
  $http.get('http://teropa.info', {
    timeout: timeout.promise
  });
  $rootScope.$apply();

  timeout.resolve();
  $rootScope.$apply();

  expect(requests[0].aborted).toBe(true);
});
```

For this test case we need to inject `$q` into the test suite:

test/http_spec.js

```
var $http, $rootScope, $q;
var xhr, requests;

beforeEach(function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  $http = injector.get('$http');
  $rootScope = injector.get('$rootScope');
  $q = injector.get('$q');
});
```

Timeout management is handled by the HTTP backend. All we do in `$http` is pass in the timeout attribute from the configuration:

src/http.js

```
$httpBackend(
  config.method,
  url,
  reqData,
  done,
  config.headers,
  config.timeout,
  config.withCredentials
);
```


In `$httpBackend` we receive that argument, and attach a Promise handler to it if one was given. That handler simply aborts the ongoing XMLHttpRequest:

src/http_backend.js

```
return function(method, url, post, callback, headers, timeout, withCredentials) {
  var xhr = new window.XMLHttpRequest();
  xhr.open(method, url, true);
  _.forEach(headers, function(value, key) {
    xhr.setRequestHeader(key, value);
  });
  if (withCredentials) {
    xhr.withCredentials = true;
  }
  xhr.send(post || null);
  xhr.onload = function() {
    var response = ('response' in xhr) ? xhr.response :
                                         xhr.responseText;
    var statusText = xhr.statusText || '';
    callback(
      xhr.status,
      response,
      xhr.getAllResponseHeaders(),
      statusText
    );
  };
  xhr.onerror = function() {
    callback(-1, null, '');
  };
  if (timeout) {
    timeout.then(function() {
      xhr.abort();
    });
  }
};
```

In addition to a Promise-based timeout, you can also simply supply a number as the value of the `timeout` attribute. Angular will then abort the request after that amount of time (in milliseconds) has passed.

We need the Jasmine clock feature that we also used in the previous chapter so that we can manipulate JavaScript's internal clock. Add the following before and after callbacks to the `$http` test suite:

test/http_spec.js


```
beforeEach(function() {
  jasmine.clock().install();
});
afterEach(function() {
  jasmine.clock().uninstall();
});
```

Now we can see that the request is aborted after the number of milliseconds specified in the request config has passed:

test/http_spec.js

```
it('allows aborting a request after a timeout', function() {
  $http.get('http://teropa.info', {
    timeout: 5000
  });
  $rootScope.$apply();

  jasmine.clock().tick(5001);

  expect(requests[0].aborted).toBe(true);
});
```

In `$httpBackend`, if we see a numeric timeout, as opposed to a Promise-like timeout, we'll now use a native `setTimeout` to abort the request:

src/http_backend.js

```
if (timeout && timeout.then) {
  timeout.then(function() {
    xhr.abort();
  });
} else if (timeout > 0) {
  setTimeout(function() {
    xhr.abort();
  }, timeout);
}
```

We should also make sure that we cancel that timeout if the request actually finishes before it fires. We don't want to call `abort` in the XMLHttpRequest when we shouldn't be touching it anymore:

src/http_backend.js

```
return function(method, url, post, callback, headers, timeout, withCredentials) {
  var xhr = new window.XMLHttpRequest();
  var timeoutId;
  xhr.open(method, url, true);
  _.forEach(headers, function(value, key) {
    xhr.setRequestHeader(key, value);
  });
  if (withCredentials) {
    xhr.withCredentials = true;
  }
  xhr.send(post || null);
  xhr.onload = function() {
    if (!_.isUndefined(timeoutId)) {
      clearTimeout(timeoutId);
    }
    var response = ('response' in xhr) ? xhr.response :
                                         xhr.responseText;
    var statusText = xhr.statusText || '';
    callback(
      xhr.status,
      response,
      xhr.getAllResponseHeaders(),
      statusText
    );
  };
  xhr.onerror = function() {
    if (!_.isUndefined(timeoutId)) {
      clearTimeout(timeoutId);
    }
    callback(-1, null, '');
  };
  if (timeout && timeout.then) {
    timeout.then(function() {
      xhr.abort();
    });
  } else if (timeout > 0) {
    timeoutId = setTimeout(function() {
      xhr.abort();
    }, timeout);
  }
};
```

Pending Requests

Sometimes it is useful to inspect information about requests that are currently in flight. This is mostly useful for debugging and tooling, but may have other use cases as well.

Such a feature could be implemented using interceptors, but there is also a built-in feature for it in `$http`: Any ongoing requests are going to be in an array accessible via the attribute

`$http.pendingRequests`. A request is added there when it is sent, and removed when a response is received - regardless of whether that's a success or an error response:

test/http_spec.js

```
describe('pending requests', function() {

  it('are in the collection while pending', function() {
    $http.get('http://teropa.info');
    $rootScope.$apply();

    expect($http.pendingRequests).toBeDefined();
    expect($http.pendingRequests.length).toBe(1);
    expect($http.pendingRequests[0].url).toBe('http://teropa.info');

    requests[0].respond(200, {}, 'OK');
    $rootScope.$apply();

    expect($http.pendingRequests.length).toBe(0);
  });

  it('are also cleared on failure', function() {
    $http.get('http://teropa.info');
    $rootScope.$apply();

    requests[0].respond(404, {}, 'Not found');
    $rootScope.$apply();

    expect($http.pendingRequests.length).toBe(0);
  });

});
```

We'll initialize this array into the `$http` object in `$httpProvider.$get`:

src/http.js

```
$http.defaults = defaults;
$http.pendingRequests = [];
```

Now we can add the addition and removal of requests to this array, which we'll do in the `sendReq` method. A request is pushed into the array when it's sent, and removed when the Promise is resolved or rejected:

src/http.js


```
function sendReq(config, reqData) {
  var deferred = $q.defer();
  $http.pendingRequests.push(config);
  deferred.promise.then(function() {
    _.remove($http.pendingRequests, config);
  }, function() {
    _.remove($http.pendingRequests, config);
  });

  // ...

}
```

Note that since we do this in `sendReq` and not in `$http`, a request is not considered "pending" while it is in the interceptor pipeline. Only when it is actually in flight.

Integrating $http and $applyAsync

We'll conclude the chapter by discussing a useful optimization that `$http` supports.

Back in Part 1 of the book, as we implemented Scopes, we discussed the `$applyAsync` feature that shipped with Angular 1.3. It essentially allows scheduling a function to be run not immediately, but after "a little bit" of time has passed. The idea is that you can limit the number of digests that happen in close succession, by postponing them and potentially combining them: If multiple `$applyAsync` calls are made in the same few milliseconds, they'll all be invoked in the same digest.

The original motivation for `$applyAsync` was to use it together with `$http`. It is very common, especially as an application starts up, to make several HTTP requests to the server at the same time, to obtain different resources. If the server is fast, it is also likely that the responses for these requests will arrive in quick succession. When this happens, Angular will kick off a new digest for each of those responses, because that's how `$http` works.

The optimization we can apply here is to kick off the digest that results from an HTTP response using `$applyAsync`. Then, if several requests do arrive close to each other, the changes they trigger will all happen in one digest. Depending on the app, this can result in very significant time savings at critical points of the application lifecycle - particularly when it's first loading.

The `$applyAsync` optimization is *not* enabled by default. You have to explicitly call `useApplyAsync(true)` on the `$httpProvider` at configuration time to enable it. We'll do that in a `beforeEach` block for our test cases:

test/http_spec.js

```
describe('useApplyAsync', function() {

  beforeEach(function() {
```



```
    var injector = createInjector(['ng', function($httpProvider) {
      $httpProvider.useApplyAsync(true);
    }]);
    $http = injector.get('$http');
    $rootScope = injector.get('$rootScope');
  });

});
```

When this optimization is enabled, a response handler should *not* be invoked immediately when a response arrives. This is in contrast to what we've seen earlier:

test/http_spec.js

```
it('does not resolve promise immediately when enabled', function() {
  var resolvedSpy = jasmine.createSpy();
  $http.get('http://teropa.info').then(resolvedSpy);
  $rootScope.$apply();

  requests[0].respond(200, {}, 'OK');
  expect(resolvedSpy).not.toHaveBeenCalled();
});
```

What'll happen instead is that the response handler has been called after we let some time pass:

teest/http_spec.js

```
it('resolves promise later when enabled', function() {
  var resolvedSpy = jasmine.createSpy();
  $http.get('http://teropa.info').then(resolvedSpy);
  $rootScope.$apply();

  requests[0].respond(200, {}, 'OK');
  jasmine.clock().tick(100);

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

In `$httpProvider` we'll set up the `useApplyAsync` method and an internal `useApplyAsync` boolean flag. The method can be invoked in two ways: When invoked with an argument it sets the flag, and when invoked without an argument, it returns the current value of the flag:

src/http.js

```
function $HttpProvider() {

  var interceptorFactories = this.interceptors = [];

  var useApplyAsync = false;
  this.useApplyAsync = function(value) {
    if (_.isUndefined(value)) {
      return useApplyAsync;
    } else {
      useApplyAsync = !!value;
      return this;
    }
  };

  // ...

}
```

In our `done` handler we'll now extract the code that actually resolves the Promise to a little helper function called `resolvePromise`. Based on the state of the `useApplyAsync` flag we'll call that function immediately (followed by `$rootScope.$apply()`) or call it later with `$rootScope.$applyAsync()`:

src/http.js

```
function done(status, response, headersString, statusText) {
  status = Math.max(status, 0);

  function resolvePromise() {
    deferred[isSuccess(status) ? 'resolve' : 'reject']({
      status: status,
      data: response,
      statusText: statusText,
      headers: headersGetter(headersString),
      config: config
    });
  }

  if (useApplyAsync) {
    $rootScope.$applyAsync(resolvePromise);
  } else {
    resolvePromise();
    if (!$rootScope.$$phase) {
      $rootScope.$apply();
    }
  }
}
```

Summary

Angular's `$http` service looks deceptively simple: It's really just a function that executes HTTP request using the browser's built-in XMLHttpRequest support.

But as we've seen in this chapter, there's more to `$http` than meets the eye. From error management to Promise integration to the interceptor pipeline, `$http` really provides a lot for application and library developers to base their networking code on.

In this chapter you have learned:

- That HTTP requests in Angular are handled by two collaborating services: `$http` and `$httpBackend`. The backend may be overriden to provide an alternative transport or to mock it out during testing.
- That `$http` is a function that takes a request configuration object and returns a Promise that will later be resolved (or rejected) when a response arrives.
- That `$http` kicks off a digest on the `$rootScope` when the response arrives - unless `useApplyAsync` is enabled, in which case it uses `$rootScope.$applyAsync` to do it later.
- That `$http.defaults` (and `$httpProvider.defaults`) allows setting some default configurations that'll be used for all requests.
- How you can supply HTTP headers by providing a `headers` object in the request configuration.
- How Angular has some default headers, such as `Accept` and `Content-Type`, but that you can override them using `$http.defaults`.
- That default and request-specific headers are merged case-insensitively, since that's how headers work according to the HTTP standard.
- That request header values may also be functions that return the concrete header values when called.
- That response headers are available through the `headers` function attached to the response object, and how that function treats header names case-insensitively.
- That response headers are parsed lazily only when requested.
- How CORS authorization can be enabled by supplying the `withCredentials` flag in request objects.
- That both request and response bodies can be *transformed* using transforms that are registered with the `transformRequest` and `transformResponse` attributes, which may be either supplied in `$http.defaults` or individual request objects
- That Angular uses request and response transforms to do JSON serialization and parsing by default.
- How request data is serialized to JSON when it is an array or an object, unless it is a special object like a Blob, File, or FormData.
- How response data is parsed as JSON when it's either specified as such with a `Content-Type` header, or when it merely *looks like* JSON.
- How single- and multi-value URL parameters can be supplied, and how they are escaped and attached to the request URL.
- That objects are serialized into JSON when used as URL parameters.
- That Dates are serialized into their ISO 8601 representation when used as URL parameters.
- That three short-hand methods are provided for HTTP methods without bodies: `$http.get`, `$http.head`, and `$http.delete`.

- That three short-hand methods are provided for HTTP methods that do have bodies: `$http.post`, `$http.put`, and `$http.patch`.
- How interceptor factories can be registered directly to the `$httpProvider` or as references to existing factories.
- That interceptors are objects with one or more of the following methods: `request`, `requestError`, `response`, and `responseErrors`.
- How interceptors form a Promise-based pipeline for processing the HTTP request and response.
- That response interceptors are invoked in reverse order compared to how they were registered.
- How `$http` extends Promise with two methods useful for dealing with HTTP responses: `success` and `error`.
- How a request can be aborted with a Promise-based or a numeric timeout attribute.
- That all ongoing requests are made available through the array `$http.pendingRequests`.
- How `$http` can use the `$applyAsync` feature of Scope to skip unnecessary digests when several HTTP responses arrive in quick succession.
- That the `$applyAsync` optimization is not enabled by default, and you have to call `$httpProvider.useApplyAsync(true)` to enable it.

Part V

Directives

At this point of the book we're ready to build what is considered by many to be the core feature of AngularJS: The directive system.

Of the three distinguishing features of Angular - digests, dependency injection, and directives - this is the one that is most directly involved with making applications for web browsers. Whereas digests and DI are more or less general purpose features, directives are designed to work specifically with the Document Object Model as browsers implement it. They are the most important building block of Angular's "view layer".

The idea of directives is simple but powerful: An extensible DOM. In addition to the HTML elements and attributes implemented by browsers and standardized by the W3C, we can make our own HTML elements and attributes with special meaning and functionality. You could say that with directives we can build Domain-Specific Languages (DSLs) on top of the standard DOM. Those DSLs may be application-specific, or they may be something we share within our companies, or they may even be distributed as open source projects.

In many ways, the idea of an extensible DOM resembles the much older idea of extensible programming languages. This idea has been most prevalent in the Lisp family of languages. In Lisps, it is widespread practice to not just write your program in the programming language, but rather *extend the programming language to better fit your problem and then write the program in that new language*. Paul Graham, in his 1993 essay Programming Bottom-Up, put it like this:

> Experienced Lisp programmers divide up their programs differently. As well as top-down design, they follow a principle which could be called bottom-up design– changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient.

If you're used to writing Angular directives, this will probably sound familiar. This is exactly what we do in Angular: We build the DOM up toward our application. We think: "I wish browsers had such-and-such an element." So we go and write it.

The idea of an extensible DOM does exist outside of Angular as well, in particular with the Web Components standard currently being finalized and implemented. Web Components are in many ways an evolution of Angular directives, and they go further in making things modular, by, for example, scoping CSS rules at component boundaries. Angular will, in its 2.0 version, also make use of the Web Components standard, but the current 1.x versions do not yet do so.

The directive compiler is also the most complex part of the Angular codebase. To some extent, this is due to the richness of features provided - the compiler does *a lot*, as we will see. Some of the complexity comes from having to deal with the often verbose and complicated DOM standard. But mostly the compiler is just showing its age. It is an implementation with several years of churn behind it, with all sorts of additions and fixes applied over time.

If you're familiar with the directive API, you'll probably have experienced the frustration of trying to figure it out.

This surface complexity makes it all the more important for us to understand what's going on inside. That is why we'll spend the next several chapters going through the directive compiler implentation in great detail. At the end of it we will surface with supercharged skills in authoring effective directives.

Chapter 15

DOM Compilation and Basic Directives

In this chapter we'll make a very simple but fully functional directive compiler. With it, we'll be able to attach behavior to the DOM using special elements, attributes, classes, and comments. We'll also learn how to apply directives to DOM fragments that span several elements.

The process of applying directives to the DOM is called *compilation*, and there is a good reason for that: Just like a programming language compiler takes source code as input and produces machine code or bytecode as output, the DOM compiler takes DOM structures as input and produces transformed DOM structures as output. The transformed DOM could be something completely different from the input DOM. Or it could be identical to the input, but have completely new kind of behavior. The compiler applies arbitrary JavaScript code to the DOM, so pretty much anything is possible.

The directive compiler lives in a file called `compile.js`. That is where we'll be spending most of our time in this part of the book.

Creating The $compile Provider

Directive compilation happens using a function called `$compile`. Just like `$rootScope` and `$parse`, it is a built-in component provided by the injector. When we create an injector with the `ng` module, we can expect `$compile` to be there:

test/angular_public_spec.js

```
it('sets up $compile', function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  expect(injector.has('$compile')).toBe(true);
});
```

`$compile` is defined as a provider in a new file `compile.js`:

src/compile.js

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

function $CompileProvider() {

  this.$get = function() {

  };

}
```

We include `$compile` to the `ng` module in `angular_public.js`, just like we do with `$parse` and `$rootScope`:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
  ngModule.provider('$httpBackend', $HttpBackendProvider);
  ngModule.provider('$http', $HttpProvider);
  ngModule.provider('$httpParamSerializer', $HttpParamSerializerProvider);
  ngModule.provider('$httpParamSerializerJQLike',
    $HttpParamSerializerJQLikeProvider);
  ngModule.provider('$compile', $CompileProvider);
}
```

Registering Directives

The main job of `$compile` will be to apply directives to the DOM. For that to work it will need to have some directives to apply. That means we need a way to *register* directives.

Directive registration happens through modules, just like the registration of services, factories, and other components. A directive is registered using the `directive` method of a module object. Whenever a directive is registered, it automatically gets the `Directive` suffix, so that when we register the directive `abc`, the injector will have a directive called `abcDirective`:

test/compile_spec.js


```
describe('$compile', function() {

  beforeEach(function() {
    delete window.angular;
    publishExternalAPI();
  });

  it('allows creating directives', function() {
    var myModule = window.angular.module('myModule', []);
    myModule.directive('testing', function() { });
    var injector = createInjector(['ng', 'myModule']);
    expect(injector.has('testingDirective')).toBe(true);
  });

});
```

The `directive` method for module objects is similar to the one we made for filters earlier. It queues up a call to the `directive` method of the `$compileProvider`:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  factory: invokeLater('$provide', 'factory'),
  value: invokeLater('$provide', 'value'),
  service: invokeLater('$provide', 'service'),
  decorator: invokeLater('$provide', 'decorator'),
  filter: invokeLater('$filterProvider', 'register'),
  directive: invokeLater('$compileProvider', 'directive'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks
  _runBlocks: []
};
```

For now, all we need this new method to do is call back to `$provide` to register the directive factory. For this purpose, we need to inject `$provide` to `$CompileProvider`. We also add the `$inject` attribute so that the injection will be minification-safe:

src/compile.js


```
function $CompileProvider($provide) {

  this.directive = function(name, directiveFactory) {
    $provide.factory(name + 'Directive', directiveFactory);
  };

  this.$get = function() {

  };

}
$CompileProvider.$inject = ['$provide'];
```

So when we register a directive, what goes into the injector is just a factory. There is one special aspect of directive factories that other kinds of factories don't have though. There may be several directives with the same name:

test/compile_spec.js

```
it('allows creating many directives with the same name', function() {
  var myModule = window.angular.module('myModule', []);
  myModule.directive('testing', _.constant({d: 'one'}));
  myModule.directive('testing', _.constant({d: 'two'}));
  var injector = createInjector(['ng', 'myModule']);

  var result = injector.get('testingDirective');
  expect(result.length).toBe(2);
  expect(result[0].d).toEqual('one');
  expect(result[1].d).toEqual('two');
});
```

In this test we register two directives called `testing`, and then see what we get back when we get `testingDirective` from the injector. What we expect to have is an array of two directives.

> The directive themselves don't do anything at this point. They're just dummy object literals.

So, unlike with other kinds of components, you cannot override a directive by just providing another directive with the same name. To alter an existing directive you'll need to use a decorator.

The reason for allowing multiple directives with the same name is that directive names are used for matching them with DOM elements and attributes. If Angular enforced uniqueness of directive names, you could not have two directives that both match the same element. This would be similar to a jQuery implemementation that didn't allow the same selector to be used for two different purposes. That would be severely restrictive indeed.

What we must do in `$CompileProvider.directive` is introduce an internal registry of directives, where each directive name points to an array of directive factories:

src/compile.js


```
function $CompileProvider($provide) {

  var hasDirectives = {};

  this.directive = function(name, directiveFactory) {
    if (!hasDirectives.hasOwnProperty(name)) {
      hasDirectives[name] = [];
    }
    hasDirectives[name].push(directiveFactory);
  };

  this.$get = function() {

  };
}
```

What we then register to the `$provider` is a function that looks up the directive factories from the internal registry and invokes each one using `$injector`. Whoever asked for the directive will receive an array of the results of the invocations:

src/compile.js

```
this.directive = function(name, directiveFactory) {
  if (!hasDirectives.hasOwnProperty(name)) {
    hasDirectives[name] = [];
    $provide.factory(name + 'Directive', ['$injector', function($injector) {
      var factories = hasDirectives[name];
      return _.map(factories, $injector.invoke);
    }]);
  }
  hasDirectives[name].push(directiveFactory);
};
```

As application developers we rarely need to inject directives to our own code since the normal approach for applying directives is through DOM compilation. But, if you do need to obtain a directive by injecting it, it is possible. You'll just always get it wrapped in an array because of how the directive factory function is implemented.

One special case we still need to handle in this code is related to the use of the `hasOwnProperty` method. Just like we've done in earlier chapters, we again need to prevent anyone from registering a directive by that name because it would override the method in the `hasDirectives` object:

test/compile_spec.js


```
it('does not allow a directive called hasOwnProperty', function() {
  var myModule = window.angular.module('myModule', []);
  myModule.directive('hasOwnProperty', function() { });
  expect(function() {
    createInjector(['ng', 'myModule']);
  }).toThrow();
});
```

This restriction can be implemented as a straightforward string comparison in the `directive` method:

src/compile.js

```
this.directive = function(name, directiveFactory) {
  if (name === 'hasOwnProperty') {
    throw 'hasOwnProperty is not a valid directive name';
  }
  if (!hasDirectives.hasOwnProperty(name)) {
    hasDirectives[name] = [];
    $provide.factory(name + 'Directive', ['$injector', function($injector) {
      var factories = hasDirectives[name];
      return _.map(factories, $injector.invoke);
    }]);
  }
  hasDirectives[name].push(directiveFactory);
};
```

There's one more special feature of directive registration we need to handle: A shorthand method for registering several directives at once. This can be done by giving the `directive` method a single object as an argument. The object's keys are interpreted as directive names and the values as the factories:

test/compile_spec.js

```
it('allows creating directives with object notation', function() {
  var myModule = window.angular.module('myModule', []);
  myModule.directive({
    a: function() { },
    b: function() { },
    c: function() { }
  });
  var injector = createInjector(['ng', 'myModule']);

  expect(injector.has('aDirective')).toBe(true);
  expect(injector.has('bDirective')).toBe(true);
  expect(injector.has('cDirective')).toBe(true);
});
```

In the `directive` method we need to check which approach the caller is using:

src/compile.js

```
this.directive = function(name, directiveFactory) {
  if (_.isString(name)) {
    if (name === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid directive name';
    }
    if (!hasDirectives.hasOwnProperty(name)) {
      hasDirectives[name] = [];
      $provide.factory(name + 'Directive', ['$injector', function($injector) {
        var factories = hasDirectives[name];
        return _.map(factories, $injector.invoke);
      }]);
    }
    hasDirectives[name].push(directiveFactory);
  } else {

  }
};
```

In the case where we've been given an object, we iterate over that object's members and recursively call the registration function for each one:

src/compile.js

```
this.directive = function(name, directiveFactory) {
  if (_.isString(name)) {
    if (name === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid directive name';
    }
    if (!hasDirectives.hasOwnProperty(name)) {
      hasDirectives[name] = [];
      $provide.factory(name + 'Directive', ['$injector', function($injector) {
        var factories = hasDirectives[name];
        return _.map(factories, $injector.invoke);
      }]);
    }
    hasDirectives[name].push(directiveFactory);
  } else {
    _.forEach(name, function(directiveFactory, name) {
      this.directive(name, directiveFactory);
    }, this);
  }
};
```


Compiling The DOM with Element Directives

Now that we have the ability to register some directives, we can get into the business of applying them. That process is called *DOM compilation*, and it is the primary responsibility of `$compile`.

Let's say we have a directive called `myDirective`. We can implement this directive as a function that returns an object:

```
myModule.directive('myDirective', function() {
  return {
  };
});
```

That object is the *directive definition object*. Its keys and values will configure the directive's behavior. One such key is called `compile`. With it, we can define the directive's *compilation function*. It is a function `$compile` will call while it is traversing the DOM. It will receive one argument, which is the element the directive is being applied to:

```
myModule.directive('myDirective', function() {
  return {
    compile: function(element) {

    }
  };
});
```

When we have a directive like this one, we can apply it to the DOM by adding an element that matches the directive's name:

```
<my-directive></my-directive>
```

Let's codify all of this as a unit test. In the test we'll need to create an injector with a directive in it. We're going to do a whole lot of that in this part of the book, so let's go ahead and add a helper function that makes it easier:

test/compile_spec.js

```
function makeInjectorWithDirectives() {
  var args = arguments;
  return createInjector(['ng', function($compileProvider) {
    $compileProvider.directive.apply($compileProvider, args);
  }]);
}
```

This function creates an injector with two modules: The `ng` module and a function module wherein a directive is registered using the `$compileProvider`.

We can put this function to use right away in our new unit test:

test/compile_spec.js


```
it('compiles element directives from a single element', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<my-directive></my-directive>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

This test does several things:

1. It creates a module with the `myDirective` directive and an injector for it.
2. It uses jQuery to parse a DOM fragment with the `<my-directive>` element
3. It gets the `$compile` function from the injector and invokes it with the jQuery object from step two.

Inside the directive's compilation function we just assign a data attribute to the element, giving us a way to check at the end of the test that the directive has indeed been applied.

> As discussed in the book's introduction, we'll be using jQuery in this book to provide low-level DOM inspection and manipulation capabilities. AngularJS does not itself have a hard dependency on jQuery, but ships with a minimal subset called jqLite instead. Since DOM esoterica is not the focus of this book, we won't delve into it and just use jQuery instead. That means that whereas AngularJS gives you jqLite objects when you work with directives, this book's implementation gives you jQuery objects.

The argument given to `$compile` is by no means limited to a single DOM element. It can, for example, be a collection of several elements. Here we apply `myDirective` to two siblings and see that the compile function is separately invoked for each of them:

test/compile_spec.js

```
it('compiles element directives found from several elements', function() {
  var idx = 1;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', idx++);
      }
    };
```



```
  });
  injector.invoke(function($compile) {
    var el = $('<my-directive></my-directive><my-directive></my-directive>');
    $compile(el);
    expect(el.eq(0).data('hasCompiled')).toBe(1);
    expect(el.eq(1).data('hasCompiled')).toBe(2);
  });
});
```

To make this all work we'll need to introduce several additions to our code. We'll go over them one by one, and at the end we'll look at the full source code of the updated `CompileProvider`.

To begin with, the `$get` method of `CompileProvider` will need to return something. That something is the `$compile` function we've just invoked in our tests:

src/compile.js

```
this.$get = function() {

  function compile($compileNodes) {

  }

  return compile;
};
```

As we saw in the tests, the function receives the DOM node(s) to compile as its argument.

> Within `$compile` the dollar prefix is often used in variable names to separate jQuery (jqLite) wrapped DOM nodes from raw DOM nodes. This is a convention picked up from the jQuery world. Unfortunately, since the dollar prefix is also used to denote framework-provided components in AngularJS, it is sometimes difficult to make out which reason the dollar prefix is being used for. In the example above, `$compileNodes` denotes a jQuery wrapped DOM node or a collection thereof.

What we'll do inside `compile` is to invoke another local function called `compileNodes`. For now, this seems like an unnecessary bit of indirection, but as we'll soon see, we do need the distinction.

src/compile.js

```
this.$get = function() {

  function compile($compileNodes) {
    return compileNodes($compileNodes);
  }
```



```
  function compileNodes($compileNodes) {

  }

  return compile;
};
```

In `compileNodes`, we will iterate over the given jQuery object to handle every given node separately. For each node, we'll look up any and all directives that should be applied to that node using a new function called `collectDirectives`:

src/compile.js

```
function compileNodes($compileNodes) {
  _.forEach($compileNodes, function(node) {
    var directives = collectDirectives(node);
  });
}

function collectDirectives(node) {

}
```

The job of `collectDirectives` is to, given a DOM node, figure out what directives apply to it and return them. For now, we'll just use one strategy to do that, which is to find directives that apply to the element's name:

src/compile.js

```
function collectDirectives(node) {
  var directives = [];
  var normalizedNodeName = _.camelCase(nodeName(node).toLowerCase());
  addDirective(directives, normalizedNodeName);
  return directives;
}
```

This code uses two helper functions that don't exist yet: `nodeName` and `addDirective`, so we'll introduce them now.

`nodeName` is a function defined in `compile.js` that returns the name of the given DOM node, which may be a raw DOM node or a jQuery-wrapped one:

src/compile.js

```
function nodeName(element) {
  return element.nodeName ? element.nodeName : element[0].nodeName;
}
```


The `addDirective` function is implemented in `compile.js`, within the closure formed by the `$get` method. It takes an array of directives, and the name of a directive. It checks if the local `hasDirectives` array has directives with that name. If it does, the corresponding directive functions are obtained from the injector and added to the array.

src/compile.js

```
function addDirective(directives, name) {
  if (hasDirectives.hasOwnProperty(name)) {
    directives.push.apply(directives, $injector.get(name + 'Directive'));
  }
}
```

Note the use of `push.apply` here. We expect `$injector` to give us an array of directives because of the way we set things up earlier. By using `apply` we basically concatenate that array to `directives`.

We're using `$injector` in `addDirective`, but don't currently have it injected to our code. We need to inject it to the wrapping `$get` method:

src/compile.js

```
this.$get = ['$injector', function($injector) {

  // ...

}];
```

Back in `compileNodes`, once we have collected the directives for the node, we'll apply them to it, for which we'll use another new function:

src/compile.js

```
function compileNodes($compileNodes) {
  _.forEach($compileNodes, function(node) {
    var directives = collectDirectives(node);
    applyDirectivesToNode(directives, node);
  });
}

function applyDirectivesToNode(directives, compileNode) {

}
```


This function iterates the directives, and calls the `compile` function from each one, giving it a jQuery-wrapped element as an argument. This is where we invoke the `compile` function of the directive definition object set up in the test case:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode) {
  var $compileNode = $(compileNode);
  _.forEach(directives, function(directive) {
    if (directive.compile) {
      directive.compile($compileNode);
    }
  });
}
```

And now we're successfully applying some directives to the DOM! The process basically iterates over each node given and repeats two steps:

1. Find all the directives that apply to the node
2. Apply those directives to the node by invoking their `compile` functions.

Here's the full code of `compile.js` at the end of the process:

src/compile.js

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

function nodeName(element) {
  return element.nodeName ? element.nodeName : element[0].nodeName;
}

function $CompileProvider($provide) {

  var hasDirectives = {};

  this.directive = function(name, directiveFactory) {
    if (_.isString(name)) {
      if (name === 'hasOwnProperty') {
        throw 'hasOwnProperty is not a valid directive name';
      }
      if (!hasDirectives.hasOwnProperty(name)) {
        hasDirectives[name] = [];
        $provide.factory(name + 'Directive', ['$injector', function($injector) {
          var factories = hasDirectives[name];
          return _.map(factories, $injector.invoke);
        }]);
      }
      hasDirectives[name].push(directiveFactory);
    } else {
```



```
      _.forEach(name, function(directiveFactory, name) {
        this.directive(name, directiveFactory);
      }, this);
    }
  };

  this.$get = ['$injector', function($injector) {

    function compile($compileNodes) {
      return compileNodes($compileNodes);
    }

    function compileNodes($compileNodes) {
      _.forEach($compileNodes, function(node) {
        var directives = collectDirectives(node);
        applyDirectivesToNode(directives, node);
      });
    }

    function collectDirectives(node) {
      var directives = [];
      var normalizedNodeName = _.camelCase(nodeName(node).toLowerCase());
      addDirective(directives, normalizedNodeName);
      return directives;
    }

    function addDirective(directives, name) {
      if (hasDirectives.hasOwnProperty(name)) {
        directives.push.apply(directives, $injector.get(name + 'Directive'));
      }
    }

    function applyDirectivesToNode(directives, compileNode) {
      var $compileNode = $(compileNode);
      _.forEach(directives, function(directive) {
        if (directive.compile) {
          directive.compile($compileNode);
        }
      });
    }

    return compile;
  }];

}
$CompileProvider.$inject = ['$provide'];
```

Recursing to Child Elements

Our simple directive implementation currently does nothing but iterate the top-level elements given to it. It would be reasonable to expect it to be able to also compile any *children* of those top-level elements:

test/compile_spec.js

```
it('compiles element directives from child elements', function() {
  var idx = 1;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', idx++);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div><my-directive></my-directive></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBeUndefined();
    expect(el.find('> my-directive').data('hasCompiled')).toBe(1);
  });
});
```

Here we check that the child `<my-directive>` has been compiled. As a sanity check, we also see that the parent `<div>` has *not* been compiled.

Our compiler should also be able to compile several *nested* directive elements:

test/compile_spec.js

```
it('compiles nested directives', function() {
  var idx = 1;
  var injector = makeInjectorWithDirectives('myDir', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', idx++);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<my-dir><my-dir><my-dir/></my-dir></my-dir>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(1);
    expect(el.find('> my-dir').data('hasCompiled')).toBe(2);
    expect(el.find('> my-dir > my-dir').data('hasCompiled')).toBe(3);
  });
});
```


Satisfying this requirement ends up being quite simple. All we need to do is recurse to each node's child nodes from `compileNodes`:

src/compile.js

```
function compileNodes($compileNodes) {
  _.forEach($compileNodes, function(node) {
    var directives = collectDirectives(node);
    applyDirectivesToNode(directives, node);
    if (node.childNodes && node.childNodes.length) {
      compileNodes(node.childNodes);
    }
  });
}
```

The order in which things are done is significant here: We compile the parent first, then the children.

Using Prefixes with Element Directives

We've seen how directives can be applied by simply matching their names to names of elements in the DOM. In the next several sections of this chapter we'll see some other ways of doing the matching.

Firstly, when matching directives to element names, Angular lets you use the prefixes `x` and `data` in the DOM:

```
<x-my-directive></x-my-directive>
<data-my-directive></data-my-directive>
```

Also, in addition to hyphens, you can use colons or underscores as the delimiter between the prefix and the directive name:

```
<x:my-directive></x:my-directive>
<x_my-directive></x_my-directive>
```

Combining these two options, there are in all six alternative ways to add prefixes to elements. To unit test them all, we'll loop over them and generate a test block for each combination:

test/compile_spec.js

```
_.forEach(['x', 'data'], function(prefix) {
  _.forEach([':', '-', '_'], function(delim) {

    it('compiles element directives with '+prefix+delim+' prefix', function() {
      var injector = makeInjectorWithDirectives('myDir', function() {
        return {
          compile: function(element) {
```

```
            element.data('hasCompiled', true);
          }
        };
      });
      injector.invoke(function($compile) {
        var el = $('<'+prefix+delim+'my-dir></'+prefix+delim+'my-dir>');
        $compile(el);
        expect(el.data('hasCompiled')).toBe(true);
      });
    });

  });
});
```

We'll introduce a new helper function on the top level of `compile.js` for handling the prefix matching. It takes the name of a DOM element as an argument and returns a "normalized" directive name. That process entails both camel-casing the name and removing any prefixes:

src/compile.js

```
function directiveNormalize(name) {
  return _.camelCase(name.replace(PREFIX_REGEXP, ''));
}
```

The prefix regexp matches either the `x` or the `data` prefix case-insensitively, followed by one of the three delimiter characters:

src/compile.js

```
var PREFIX_REGEXP = /(x[\:\-_]|data[\:\-_])/i;
```

In `collectDirectives` we'll now replace the call to `_.camelCase` with a call to the new `directiveNormalize`:

src/compile.js

```
function collectDirectives(node) {
  var directives = [];
  var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
  addDirective(directives, normalizedNodeName);
  return directives;
}
```

Applying Directives to Attributes

Matching element names to directive names is not the only way to couple directives with the DOM. The second approach we'll look up is matching by *attribute names*. This is perhaps the most common way directives are applied in Angular applications:

test/compile_spec.js

```
it('compiles attribute directives', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

The same prefix rules we just implemented for element names apply to attribute names as well. For example, the `x:` prefix is allowed:

test/compile_spec.js

```
it('compiles attribute directives with prefixes', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div x:my-directive></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

It is naturally possible to apply several attribute directives to the same element:

test/compile_spec.js


```
it('compiles several attribute directives in an element', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        compile: function(element) {
          element.data('hasCompiled', true);
        }
      };
    },
    mySecondDirective: function() {
      return {
        compile: function(element) {
          element.data('secondCompiled', true);
        }
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive my-second-directive></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
    expect(el.data('secondCompiled')).toBe(true);
  });
});
```

We can also combine both element and attribute directives in the same element:

test/compile_spec.js

```
it('compiles both element and attributes directives in an element', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        compile: function(element) {
          element.data('hasCompiled', true);
        }
      };
    },
    mySecondDirective: function() {
      return {
        compile: function(element) {
          element.data('secondCompiled', true);
        }
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<my-directive my-second-directive></my-directive>');
    $compile(el);
```

```
    expect(el.data('hasCompiled')).toBe(true);
    expect(el.data('secondCompiled')).toBe(true);
  });
});
```

The function responsible for going through the attributes is `collectDirectives`, where we're already doing matching of directives by element name. Here, we'll iterate over the current node's attributes and add any directives that might match them - in a case-insensitive fashion:

src/compile.js

```
function collectDirectives(node) {
  var directives = [];
  var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
  addDirective(directives, normalizedNodeName);
  _.forEach(node.attributes, function(attr) {
    var normalizedAttrName = directiveNormalize(attr.name.toLowerCase());
    addDirective(directives, normalizedAttrName);
  });
  return directives;
}
```

Angular also let's us use a special `ng-attr` prefix when applying directives through attributes:

test/compile_spec.js

```
it('compiles attribute directives with ng-attr prefix', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div ng-attr-my-directive></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

The `ng-attr` prefix can also be combined with one of the other prefixes we saw earlier:

test/compile_spec.js

```
it('compiles attribute directives with data:ng-attr prefix', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div data:ng-attr-my-directive></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

We'll handle this case by, after normalizing the directive name, checking whether it begins with `ngAttr` followed by an uppercase character. If so, we remove that part and downcase the new first character:

src/compile.js

```
function collectDirectives(node) {
  var directives = [];
  var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
  addDirective(directives, normalizedNodeName);
  _.forEach(node.attributes, function(attr) {
    var normalizedAttrName = directiveNormalize(attr.name.toLowerCase());
    if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
      normalizedAttrName =
        normalizedAttrName[6].toLowerCase() +
        normalizedAttrName.substring(7);
    }
    addDirective(directives, normalizedAttrName);
  });
  return directives;
}
```

Applying Directives to Classes

The third method for applying directives to the DOM is by matching them to the CSS class names of elements. We can simply use a class name that (when normalized) matches a directive name:

test/compile_spec.js

```
it('compiles class directives', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div class="my-directive"></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

Just as with attribute directives, several classes of the same element may be applied to directives:

test/compile_spec.js

```
it('compiles several class directives in an element', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        compile: function(element) {
          element.data('hasCompiled', true);
        }
      };
    },
    mySecondDirective: function() {
      return {
        compile: function(element) {
          element.data('secondCompiled', true);
        }
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div class="my-directive my-second-directive"></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
    expect(el.data('secondCompiled')).toBe(true);
  });
});
```

Finally, class names may have the same kinds of prefixes we have seen with elements and attributes (though they may not use the `ng-attr` prefix):

test/compile_spec.js


```
it('compiles class directives with prefixes', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div class="x-my-directive"></div>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

To implement class-based matching, we'll just iterate over each node's `classList` attribute and match directives to each normalized class name:

src/compile.js

```
function collectDirectives(node) {
  var directives = [];
  var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
  addDirective(directives, normalizedNodeName);
  _.forEach(node.attributes, function(attr) {
    var normalizedAttrName = directiveNormalize(attr.name.toLowerCase());
    if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
      normalizedAttrName =
        normalizedAttrName[6].toLowerCase() +
        normalizedAttrName.substring(7);
    }
    addDirective(directives, normalizedAttrName);
  });
  _.forEach(node.classList, function(cls) {
    var normalizedClassName = directiveNormalize(cls);
    addDirective(directives, normalizedClassName);
  });
  return directives;
}
```

> AngularJS does not use `classList` because it's a HTML5 feature not widely supported in older browsers. Instead, it manually tokenizes the `className` attribute of the element.

Applying Directives to Comments

The final directive matching mechanism Angular has is perhaps the most esoteric one: Applying directives to HTML comments. This is possible by crafting a comment that begins with the text `directive:`, followed by the directive's name:

```
<!-- directive: my-directive -->
```

Here's the same in a unit test:

test/compile_spec.js

```
it('compiles comment directives', function() {
  var hasCompiled;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        hasCompiled = true;
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<!-- directive: my-directive -->');
    $compile(el);
    expect(hasCompiled).toBe(true);
  });
});
```

> Here we use a test variable `hasCompiled` instead of a data attribute on the node because of the limitations of attaching jQuery data to comment nodes.

The object we have at hand when we enter the `collectDirectives` function is a DOM node. It may be an element or it may be something else, such as a comment. What we need to do is execute different code based on the type of the node. The type can be determined by looking at the `nodeType` attribute of the node.

We'll wrap the code we've written thus far in an `if` branch for elements and introduce a new one for comments:

src/compile.js

```
function collectDirectives(node) {
  var directives = [];
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName);
    _.forEach(node.attributes, function(attr) {
      var normalizedAttrName = directiveNormalize(attr.name.toLowerCase());
      if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
```



```
        normalizedAttrName =
          normalizedAttrName[6].toLowerCase() +
          normalizedAttrName.substring(7);
      }
      addDirective(directives, normalizedAttrName);
    });
    _.forEach(node.classList, function(cls) {
      var normalizedClassName = directiveNormalize(cls);
      addDirective(directives, normalizedClassName);
    });
  } else if (node.nodeType === Node.COMMENT_NODE) {

  }
  return directives;
}
```

What we do in the node branch is match a regular expression with the comment's text value, and see if it begins with `directive:`. If it does, we take the directive name that follows, normalize it, and find any directives that match it.

src/compile.js

```
function collectDirectives(node) {
  var directives = [];
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName);
    _.forEach(node.attributes, function(attr) {
      var normalizedAttrName = directiveNormalize(attr.name.toLowerCase());
      if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
        normalizedAttrName =
          normalizedAttrName[6].toLowerCase() +
          normalizedAttrName.substring(7);
      }
      addDirective(directives, normalizedAttrName);
    });
    _.forEach(node.classList, function(cls) {
      var normalizedClassName = directiveNormalize(cls);
      addDirective(directives, normalizedClassName);
    });
  } else if (node.nodeType === Node.COMMENT_NODE) {
    var match = /^\s*directive\:\s*([\d\w\-_]+)/.exec(node.nodeValue);
    if (match) {
      addDirective(directives, directiveNormalize(match[1]));
    }
  }
  return directives;
}
```

Note that the regular expression allows whitespace at the beginning of the comment as well as after the `directive:` prefix.

Restricting Directive Application

So there are four ways in which Angular matches directives with the DOM: By element name, by attribute name, by class name, and by special comments.

This does not mean, however, that any given directive can be matched using whatever approach the application developer chooses. Directive authors do have the possibility to *restrict* which of the four matching modes the directive can be used with. This is useful because it would make little sense to apply a directive that implements a custom element to a comment, for example. It just wouldn't work.

The restriction can be done by specifying a `restrict` attribute on the directive definition object. That attribute carries a tiny domain-specific language consisting of single-character codes and combinations thereof:

- `E` means "match with element names"
- `A` means "match with attribute names"
- `C` means "match with class names"
- `M` means "match with comments"
- `EA` means "match with element names and attribute names"
- `MCA` means "match with comments, class names, and attribute names"
- etc.

Let's use some generative testing techniques so that we can cover a lot of ground without having to write tens of test cases. We'll set up a data structure with different combinations of `restrict` and the expected outcomes of using the combination in each of the four modes. We'll then loop over that data structure and generate `describe` blocks:

test/compile_spec.js

```
_.forEach({
  E:    {element: true,  attribute: false, class: false, comment: false},
  A:    {element: false, attribute: true,  class: false, comment: false},
  C:    {element: false, attribute: false, class: true,  comment: false},
  M:    {element: false, attribute: false, class: false, comment: true},
  EA:   {element: true,  attribute: true,  class: false, comment: false},
  AC:   {element: false, attribute: true,  class: true,  comment: false},
  EAM:  {element: true,  attribute: true,  class: false, comment: true},
  EACM: {element: true,  attribute: true,  class: true,  comment: true},
}, function(expected, restrict) {

  describe('restricted to '+restrict, function() {

  });

});
```

Inside this loop we can then add *another* loop that iterates over the four kinds of DOM structures we can use:

test/compile_spec.js


```
_.forEach({
  E:    {element: true,  attribute: false, class: false, comment: false},
  A:    {element: false, attribute: true,  class: false, comment: false},
  C:    {element: false, attribute: false, class: true,  comment: false},
  M:    {element: false, attribute: false, class: false, comment: true},
  EA:   {element: true,  attribute: true,  class: false, comment: false},
  AC:   {element: false, attribute: true,  class: true,  comment: false},
  EAM:  {element: true,  attribute: true,  class: false, comment: true},
  EACM: {element: true,  attribute: true,  class: true,  comment: true},
}, function(expected, restrict) {

  describe('restricted to '+restrict, function() {

    _.forEach({
      element:   '<my-directive></my-directive>',
      attribute: '<div my-directive></div>',
      class:     '<div class="my-directive"></div>',
      comment:   '<!-- directive: my-directive -->'
    }, function(dom, type) {

      it((expected[type] ? 'matches' : 'does not match')+' on '+type, function() {
        var hasCompiled = false;
        var injector = makeInjectorWithDirectives('myDirective', function() {
          return {
            restrict: restrict,
            compile: function(element) {
              hasCompiled = true;
            }
          };
        });
        injector.invoke(function($compile) {
          var el = $(dom);
          $compile(el);
          expect(hasCompiled).toBe(expected[type]);
        });
      });

    });

  });

});
```

All in all these loops add 32 new test cases. If you want to add more combinations to the data structure, that's also easy to do.

The restriction of directives will happen in the `addDirectives` function. Before we go there, however, we need to modify `collectDirectives` so that it lets `addDirectives` know which of the four matching modes is currently being used:

src/compile.js

```
function collectDirectives(node) {
  var directives = [];
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName, 'E');
    _.forEach(node.attributes, function(attr) {
      var normalizedAttrName = directiveNormalize(attr.name.toLowerCase());
      if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
        normalizedAttrName =
          normalizedAttrName[6].toLowerCase() +
          normalizedAttrName.substring(7);
      }
      addDirective(directives, normalizedAttrName, 'A');
    });
    _.forEach(node.classList, function(cls) {
      var normalizedClassName = directiveNormalize(cls);
      addDirective(directives, normalizedClassName, 'C');
    });
  } else if (node.nodeType === Node.COMMENT_NODE) {
    var match = /^\s*directive\:\s*([\d\w\-_]+)/.exec(node.nodeValue);
    if (match) {
      addDirective(directives, directiveNormalize(match[1]), 'M');
    }
  }
  return directives;
}
```

The `addDirective` function can now filter the array of matching directives to those whose `restrict` attribute has a character for the current mode:

src/compile.js

```
function addDirective(directives, name, mode) {
  if (hasDirectives.hasOwnProperty(name)) {
    var foundDirectives = $injector.get(name + 'Directive');
    var applicableDirectives = _.filter(foundDirectives, function(dir) {
      return dir.restrict.indexOf(mode) !== -1;
    });
    directives.push.apply(directives, applicableDirectives);
  }
}
```

After you make this change, you'll notice an unfortunate effect: We have now broken most of our existing directive test cases. This is because they don't have the `restrict` attribute which we now require. We need to add one. Beginning from the test case `'compiles element directives from a single element'`, add a `restrict` attribute with value `'EACM'` to each one until the previous tests pass:

test/compile_spec.js


```
it('compiles element directives from a single element', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      restrict: 'EACM',
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<my-directive></my-directive>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

Finally, the `restrict` attribute does have a default value of `EA`. That is, if you do not define `restrict`, your directive is matched using element and attribute names only:

test/compile_spec.js

```
it('applies to attributes when no restrict given', function() {
  var hasCompiled = false;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        hasCompiled = true;
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive></div>');
    $compile(el);
    expect(hasCompiled).toBe(true);
  });
});

it('applies to elements when no restrict given', function() {
  var hasCompiled = false;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        hasCompiled = true;
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<my-directive></my-directive>');
```



```
    $compile(el);
    expect(hasCompiled).toBe(true);
  });
});

it('does not apply to classes when no restrict given', function() {
  var hasCompiled = false;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        hasCompiled = true;
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div class="my-directive"></div>');
    $compile(el);
    expect(hasCompiled).toBe(false);
  });
});
```

We'll apply the default value back in the directive factory function, where we first obtain the directive function itself:

src/compile.js

```
this.directive = function(name, directiveFactory) {
  if (_.isString(name)) {
    if (name === 'hasOwnProperty') {
      throw 'hasOwnProperty is not a valid directive name';
    }
    if (!hasDirectives.hasOwnProperty(name)) {
      hasDirectives[name] = [];
      $provide.factory(name + 'Directive', ['$injector', function($injector) {
        var factories = hasDirectives[name];
        return _.map(factories, function(factory) {
          var directive = $injector.invoke(factory);
          directive.restrict = directive.restrict || 'EA';
          return directive;
        });
      }]);
    }
    hasDirectives[name].push(directiveFactory);
  } else {
    _.forEach(name, function(directiveFactory, name) {
      this.directive(name, directiveFactory);
    }, this);
  }
};
```

Prioritizing Directives

When multiple directives are used on an element, the order in which they're applied often makes a big difference. One directive may depend on the effects of another to be already applied.

Instead of putting the responsibility of correct directive order on the application programmer's shoulders, Angular directives have an internal *priority* configuration that controls the order in which they're applied. Every directive definition object has a `priority` attribute, and for each node all the matched directives are sorted by this attribute before they're compiled. Priorities are numeric, and a larger number means higher priority - i.e. for compilation, directives are sorted in *descending* order by priority.

Expressed as test cases, what this means is that when there are two directives with specific priorities on an element, they're compiled in priority order:

test/compile_spec.js

```
it('applies in priority order', function() {
  var compilations = [];
  var injector = makeInjectorWithDirectives({
    lowerDirective: function() {
      return {
        priority: 1,
        compile: function(element) {
          compilations.push('lower');
        }
      };
    },
    higherDirective: function() {
      return {
        priority: 2,
        compile: function(element) {
          compilations.push('higher');
        }
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div lower-directive higher-directive></div>');
    $compile(el);
    expect(compilations).toEqual(['higher', 'lower']);
  });
});
```

When two directives have the same priority, the tie is broken by comparing by *name*, so that even when priorities are the same, the application order is stable and predictable:

test/compile_spec.js

```
it('applies in name order when priorities are the same', function() {
  var compilations = [];
  var injector = makeInjectorWithDirectives({
    firstDirective: function() {
      return {
        priority: 1,
        compile: function(element) {
          compilations.push('first');
        }
      };
    },
    secondDirective: function() {
      return {
        priority: 1,
        compile: function(element) {
          compilations.push('second');
        }
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div second-directive first-directive></div>');
    $compile(el);
    expect(compilations).toEqual(['first', 'second']);
  });
});
```

When there are two directives for which both priorities *and* names are the same, they are applied in the order in which they were registered:

test/compile_spec.js

```
it('applies in registration order when names are the same', function() {
  var compilations = [];
  var myModule = window.angular.module('myModule', []);
  myModule.directive('aDirective', function() {
    return {
      priority: 1,
      compile: function(element) {
        compilations.push('first');
      }
    };
  });
  myModule.directive('aDirective', function() {
    return {
      priority: 1,
      compile: function(element) {
        compilations.push('second');
      }
```

```
    };
  });
  var injector = createInjector(['ng', 'myModule']);
  injector.invoke(function($compile) {
    var el = $('<div a-directive></div>');
    $compile(el);
    expect(compilations).toEqual(['first', 'second']);
  });
});
```

This test case will pass immediately, but we include it for the sake of completeness.

What we're going to do is once we've collected all the directives for a given element in `collectDirectives`, we'll sort the result just before returning it to the caller. We can use the built-in `sort` method of JavaScript arrays, and provide a custom compare function that we'll introduce momentarily:

src/compile.js

```
function collectDirectives(node) {
  var directives = [];

  // ...

  directives.sort(byPriority);
  return directives;
}
```

The compare function `byPriority` takes two directives:

src/compile.js

```
function byPriority(a, b) {

}
```

The function will primarily look at the priorities of the directives, and return a negative or positive number depending on whether the first or the second of the priorities is larger ("higher"):

src/compile.js

```
function byPriority(a, b) {
  return b.priority - a.priority;
}
```


When the priorities are the same, the tie is broken by comparing by name. We use the `<` operator, which compares strings lexicographically:

src/compile.js

```
function byPriority(a, b) {
  var diff = b.priority - a.priority;
  if (diff !== 0) {
    return diff;
  } else {
    return (a.name < b.name ? -1 : 1);
  }
}
```

To get the directive name we're referencing its `name` attribute, which is something we don't actually have yet. We should set it during directive registration, in the `$compileProvider.directive` method:

src/compile.js

```
$provide.factory(name + 'Directive', ['$injector', function($injector) {
  var factories = hasDirectives[name];
  return _.map(factories, function(factory) {
    var directive = $injector.invoke(factory);
    directive.restrict = directive.restrict || 'EA';
    directive.name = directive.name || name;
    return directive;
  });
}]);
```

As the final priority rule, to break ties consistently even if the names of the directives match, we use the registration order:

src/compile.js

```
function byPriority(a, b) {
  var diff = b.priority - a.priority;
  if (diff !== 0) {
    return diff;
  } else {
    if (a.name !== b.name) {
      return (a.name < b.name ? -1 : 1);
    } else {
      return a.index - b.index;
    }
  }
}
```


The `index` attribute is also something we don't have yet. We can add that too during directive registration:

src/compile.js

```
$provide.factory(name + 'Directive', ['$injector', function($injector) {
  var factories = hasDirectives[name];
  return _.map(factories, function(factory, i) {
    var directive = $injector.invoke(factory);
    directive.restrict = directive.restrict || 'EA';
    directive.name = directive.name || name;
    directive.index = i;
    return directive;
  });
}]);
```

As a directive author you don't necessarily need to specify a priority every time. If you do not give one, a default value of `0` is used:

test/compile_spec.js

```
it('uses default priority when one not given', function() {
  var compilations = [];
  var myModule = window.angular.module('myModule', []);
  myModule.directive('firstDirective', function() {
    return {
      priority: 1,
      compile: function(element) {
        compilations.push('first');
      }
    };
  });
  myModule.directive('secondDirective', function() {
    return {
      compile: function(element) {
        compilations.push('second');
      }
    };
  });
  var injector = createInjector(['ng', 'myModule']);
  injector.invoke(function($compile) {
    var el = $('<div second-directive first-directive></div>');
    $compile(el);
    expect(compilations).toEqual(['first', 'second']);
  });
});
```


This is also something we can set up during directive registration. We either use the defined priority or zero:

src/compile.js

```
$provide.factory(name + 'Directive', ['$injector', function($injector) {
  var factories = hasDirectives[name];
  return _.map(factories, function(factory, i) {
    var directive = $injector.invoke(factory);
    directive.restrict = directive.restrict || 'EA';
    directive.priority = directive.priority || 0;
    directive.name = directive.name || name;
    directive.index = i;
    return directive;
  });
}]);
```

Terminating Compilation

Usually when you give Angular some DOM element to compile, the whole DOM subtree starting from that element gets compiled right away. That's because of the recursive nature of the compilation we have implemented in this chapter. There are cases where not everything will get compiled, however. One of them is when one of the directives used in the DOM is a *terminal directive*.

A directive can be marked as terminal, by setting the `terminal` key in its definition object to `true`. What happens then is that when this directive is compiled, it *terminates the compilation* right away, and any further directives on the element are *not* compiled.

The most common use case for `terminal` is directives that want to delay compilation. For example, Angular's built-in `ng-if` is a terminal directive, and using it stops compilation for that DOM subtree. The directive then later launches *another* compilation for its contents when its conditional expression becomes truthy:

```
<div ng-if="condition">
  <!-- Contents compiled later when condition is true -->
  <div some-other-directive></div>
</div>
```

As discussed, when there's a terminal directive on a node, other directives with lower priorities will not get compiled:

test/compile_spec.js

```
it('stops compiling at a terminal directive', function() {
  var compilations = [];
  var myModule = window.angular.module('myModule', []);
  myModule.directive('firstDirective', function() {
    return {
```



```
      priority: 1,
      terminal: true,
      compile: function(element) {
        compilations.push('first');
      }
    };
  });
  myModule.directive('secondDirective', function() {
    return {
      priority: 0,
      compile: function(element) {
        compilations.push('second');
      }
    };
  });
  var injector = createInjector(['ng', 'myModule']);
  injector.invoke(function($compile) {
    var el = $('<div first-directive second-directive></div>');
    $compile(el);
    expect(compilations).toEqual(['first']);
  });
});
```

However, if there are other directives with the *same priority* as the terminal directive's priority, they are still compiled:

test/compile_spec.js

```
it('still compiles directives with same priority after terminal', function() {
  var compilations = [];
  var myModule = window.angular.module('myModule', []);
  myModule.directive('firstDirective', function() {
    return {
      priority: 1,
      terminal: true,
      compile: function(element) {
        compilations.push('first');
      }
    };
  });
  myModule.directive('secondDirective', function() {
    return {
      priority: 1,
      compile: function(element) {
        compilations.push('second');
      }
    };
  });
  var injector = createInjector(['ng', 'myModule']);
```



```
  injector.invoke(function($compile) {
    var el = $('<div first-directive second-directive></div>');
    $compile(el);
    expect(compilations).toEqual(['first', 'second']);
  });
});
```

In `applyDirectivesToNode`, where we do the actual directive compilation, we should track the priority of any terminal directives we may have seen. We initialize the "terminal priority" with the lowest possible number in JavaScript and update it if we see a terminal directive:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  _.forEach(directives, function(directive) {
    if (directive.compile) {
      directive.compile($compileNode, attrs);
    }
    if (directive.terminal) {
      terminalPriority = directive.priority;
    }
  });
}
```

If we then encounter a directive whose priority is lower than our terminal priority, we'll exit the directive loop early by returning `false`, effectively causing compilation for this node to end:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  _.forEach(directives, function(directive) {
    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
      directive.compile($compileNode, attrs);
    }
    if (directive.terminal) {
      terminalPriority = directive.priority;
    }
  });
}
```


What should also happen when a terminal directive is encountered is that child nodes are not compiled. We're still currently compiling them even when compilation should terminate:

test/compile_spec.js

```
it('stops child compilation after a terminal directive', function() {
  var compilations = [];
  var myModule = window.angular.module('myModule', []);
  myModule.directive('parentDirective', function() {
    return {
      terminal: true,
      compile: function(element) {
        compilations.push('parent');
      }
    };
  });
  myModule.directive('childDirective', function() {
    return {
      compile: function(element) {
        compilations.push('child');
      }
    };
  });
  var injector = createInjector(['ng', 'myModule']);
  injector.invoke(function($compile) {
    var el = $('<div parent-directive><div child-directive></div></div>');
    $compile(el);
    expect(compilations).toEqual(['parent']);
  });
});
```

For now, what we can do is return a "terminal" flag from `applyDirectivesToNode`, which will be `true` when there was a terminal directive on the node:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  _.forEach(directives, function(directive) {
    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
```



```
      directive.compile($compileNode, attrs);
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });
  return terminal;
}
```

This is a flag we can check at `compileNodes`. If it is set for some node, we skip the compilation of its children:

src/compile.js

```
function compileNodes($compileNodes) {
  _.forEach($compileNodes, function(node) {
    var attrs = {};
    var directives = collectDirectives(node, attrs);
    var terminal = applyDirectivesToNode(directives, node, attrs);
    if (!terminal && node.childNodes && node.childNodes.length) {
      compileNodes(node.childNodes);
    }
  });
}
```

We will later revisit the details of how this flag is managed, but in any case we have now implemented the gist of this functionality.

Applying Directives Across Multiple Nodes

So far we've seen how directives can be matched to single elements with four different mechanisms. There is one more mechanism to cover, which is matching a directive to a collection of several sibling elements, by explicitly denoting the start and end elements of the directive:

```
<div my-directive-start>
</div>
<some-other-html/>
<div my-directive-end>
</div>
```

Not all directives can be applied this way. The directive author must explicitly set a `multiElement` flag on the directive definition object to enable the behavior. Furthermore, this mechanism applies to attribute matching only.

When you apply an attribute directive like this, what you get as an argument to the directive's `compile` function is a jQuery/jqLite object with *both the start and end elements and all the elements in between*:

test/compile_spec.js


```
it('allows applying a directive to multiple elements', function() {
  var compileEl = false;
  var injector = makeInjectorWithDirectives('myDir', function() {
    return {
      multiElement: true,
      compile: function(element) {
        compileEl = element;
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div my-dir-start></div><span></span><div my-dir-end></div>');
    $compile(el);
    expect(compileEl.length).toBe(3);
  });
});
```

The handling of start/end attributes begins in `collectDirectives`, where we first iterate each element's attributes. What we should do here is see if we're dealing with a multi-element directive. This is done in three steps:

1. Form a version of the directive name that doesn't have any Start or End suffixes
2. See if there's a multi-element directive registered by that name, and
3. See if the current attribute name actually does end with `Start`.

If 2 and 3 are both true, we are dealing with a multi-element directive application:

src/compile.js

```
_.forEach(node.attributes, function(attr) {
  var normalizedAttrName = directiveNormalize(attr.name.toLowerCase());
  if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
    normalizedAttrName =
      normalizedAttrName[6].toLowerCase() +
      normalizedAttrName.substring(7);
  }
  var directiveNName = normalizedAttrName.replace(/(Start|End)$/, '');
  if (directiveIsMultiElement(directiveNName)) {
    if (/Start$/.test(normalizedAttrName)) {
    }
  }
  addDirective(directives, normalizedAttrName, 'A');
});
```

The new helper function `directiveIsMultiElement` sees first whether there are directives registered with the given name. If there are, it gets them from the injector and checks if any of them have the `multiElement` flag set to `true`:

src/compile.js


```
function directiveIsMultiElement(name) {
  if (hasDirectives.hasOwnProperty(name)) {
    var directives = $injector.get(name + 'Directive');
    return _.any(directives, {multiElement: true});
  }
  return false;
}
```

If this is indeed a multi-element directive application, we're going to store the start and end attribute names along with the directive, so that we can use them for matching elements later in the compilation process. We'll introduce variables for the start and end attribute names, populate them if we've matched a start attribute, and pass them along to `addDirective`:

src/compile.js

```
_.forEach(node.attributes, function(attr) {
  var attrStartName, attrEndName;
  var normalizedAttrName = directiveNormalize(attr.name.toLowerCase());
  if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
    normalizedAttrName =
      normalizedAttrName[6].toLowerCase() +
      normalizedAttrName.substring(7);
  }
  var directiveNName = normalizedAttrName.replace(/(Start|End)$/, '');
  if (directiveIsMultiElement(directiveNName)) {
    if (/Start$/.test(normalizedAttrName)) {
      attrStartName = normalizedAttrName;
      attrEndName =
        normalizedAttrName.substring(0, normalizedAttrName.length - 5) + 'End';
      normalizedAttrName =
        normalizedAttrName.substring(0, normalizedAttrName.length - 5);
    }
  }
  addDirective(directives, normalizedAttrName, 'A', attrStartName, attrEndName);
});
```

Before we move on, there's one problem we need to fix, though. What we are now passing to `addDirective` are normalized, camel-cased attribute names that are no longer in the exact format they were in the DOM. It is going to be difficult to match against them when looking at the DOM later on.

What we need to do is use the *original, non-normalized attribute names* instead. What's peculiar about how Angular does this, however, is that if there is also an `ng-attr-` prefix applied to the start attribute, that is *not* stored. So, basically, the `ng-attr-` prefix and the `-start` suffix cannot be used at the same time.

What we'll do is, after the possible removal of the `ng-attr-` prefix, "denormalize" the attribute name again by hyphenizing it, and then use that to store the start and end attribute names:

src/compile.js


```
_.forEach(node.attributes, function(attr) {
  var attrStartName, attrEndName;
  var name = attr.name;
  var normalizedAttrName = directiveNormalize(name.toLowerCase());
  if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
    name = _.kebabCase(
      normalizedAttrName[6].toLowerCase() +
      normalizedAttrName.substring(7)
    );
  }
  var directiveNName = normalizedAttrName.replace(/(Start|End)$/, '');
  if (directiveIsMultiElement(directiveNName)) {
    if (/Start$/.test(normalizedAttrName)) {
      attrStartName = name;
      attrEndName = name.substring(0, name.length - 5) + 'end';
      name = name.substring(0, name.length - 6);
    }
  }
  normalizedAttrName = directiveNormalize(name.toLowerCase());
  addDirective(directives, normalizedAttrName, 'A', attrStartName, attrEndName);
});
```

Now that we have an acceptable implementation of `collectDirectives`, let's look at `addDirective`, which now has two new (optional) arguments: The start and end attributes used with the directive.

What we'll do here is, if the start and end attributes are given, attach that to the directive object with the special keys `$$start` and `$$end`. We don't want to contaminate the original directive object with these keys though, so we'll make an extended version for our own purposes. This is important since a directive may be applied several times, sometimes with start/end tag separation and sometimes without it:

src/compile.js

```
function addDirective(directives, name, mode, attrStartName, attrEndName) {
  if (hasDirectives.hasOwnProperty(name)) {
    var foundDirectives = $injector.get(name + 'Directive');
    var applicableDirectives = _.filter(foundDirectives, function(dir) {
      return dir.restrict.indexOf(mode) !== -1;
    });
    _.forEach(applicableDirectives, function(directive) {
      if (attrStartName) {
        directive = _.create(directive, {
          $$start: attrStartName,
          $$end: attrEndName
        });
      }
      directives.push(directive);
    });
  }
}
```


The next step of the process is taken in `applyDirectivesToNode`, which is where we actually call the `compile` method of the directive. Before we do that, we have to see if this directive application had the start/end tag separation and if it did, replace the nodes passed to `compile` with the start and end nodes and any siblings in between. We expect that set of elements to be available using a function called `groupScan`

src/compile.js

```
function applyDirectivesToNode(directives, compileNode) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }

    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
      directive.compile($compileNode);
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });
  return terminal;
}
```

This is a new function that takes three arguments: A node to begin the search from, and the start and end attribute names:

src/compile.js

```
function groupScan(node, startAttr, endAttr) {

}
```

The function checks if the initial node has the start attribute. If it does not, the function gives up and just returns a result with the initial node

src/compile.js


```
function groupScan(node, startAttr, endAttr) {
  var nodes = [];
  if (startAttr && node && node.hasAttribute(startAttr)) {

  } else {
    nodes.push(node);
  }
  return $(nodes);
}
```

If the initial node *does* have the start attribute, the function begins collecting the group. The group consists of the initial node and all siblings that come after it, up to and including the sibling that has the end attribute:

src/compile.js

```
function groupScan(node, startAttr, endAttr) {
  var nodes = [];
  if (startAttr && node && node.hasAttribute(startAttr)) {
    var depth = 0;
    do {
      if (node.nodeType === Node.ELEMENT_NODE) {
        if (node.hasAttribute(startAttr)) {
          depth++;
        } else if (node.hasAttribute(endAttr)) {
          depth--;
        }
      }
      nodes.push(node);
      node = node.nextSibling;
    } while (depth > 0);
  } else {
    nodes.push(node);
  }
  return $(nodes);
}
```

The loop maintains a `depth` variable, which it increments when a start attribute is encountered and decremented when an end attribute is encountered. This is done so that we'll handle cases with "nested" groups correctly:

```
<div my-dir-start></div>
<div my-dir-start></div>
<div my-dir-end></div>
<div my-dir-end></div>
```

And finally we're also able to apply directives to multi-element spans!

Summary

We now have a passable directive engine, with which we can define directives and then apply them on DOM structures. This is already quite a useful thing, and can, for example, be used in many contexts where you would traditionally use jQuery to selects elements from the DOM and attach behavior to them.

When it comes to achieving the full power of Angular directives, however, we are just getting started.

In this chapter you have learned:

- How the `$compile` provider is constructed
- That directive registration happens outside of the core dependency injection mechanisms but integrates with it.
- That directives are available from the injector though they're usually applied via DOM compilation.
- That an Angular application may have several directives with the same name.
- How several directives can be registered with one `module.directive()` invocation using the shorthand object notation.
- How the DOM compilation happens: For each node in the compiled DOM, matching directives are collected and then applied to the node.
- How the compiler recurses into child nodes after their parents are compiled.
- How the directive name can be prefixed in different ways when specified in the DOM.
- How the different matching modes (element, attribute, class, comment) work.
- How directives can be restricted to only match certain modes, where the default is `'A'` for attribute matching.
- How directives can be applied over a group of sibling nodes by having `-start` and `-end` attributes in separate elements.

In the next chapter we'll look at the `Attributes` object that provides several facilities for directive authors to work with DOM attributes.

Chapter 16

Directive Attributes

Dealing with DOM attributes is an important part of working with directives. This is true whether we actually use attribute *directives* or not. Element and class directives often have a need to interact with attributes as well. Even comment directives may have attributes associated with them, as we'll see in this chapter.

Attributes can be used to configure directives and to pass information to them. Directives also often manipulate the attributes of their elements to change how they look or behave in the browser. In addition to this, attributes also provide a means for directive-to-directive communication, using a mechanism called *observing*.

In this chapter we'll implement the directive attribute system in full. At the end of it you'll know all the secrets of the `attrs` argument that the `compile` and `link` functions of your directives receive.

Passing Attributes to the `compile` Function

Our directive implementation from the previous chapter supports DOM compilation, and the directive `compile` function. We saw how that function receives the jQuery (jqLite) wrapped DOM element as an argument. That element naturally also provides access to the element's DOM attributes. However, there's an even easier way to get to those attributes, which is using a *second* argument to the compile function. That second argument is an object whose properties are the element's DOM attributes, with their names camelCased:

test/compile_spec.js

```
describe('attributes', function() {

  it('passes the element attributes to the compile function', function() {
    var injector = makeInjectorWithDirectives('myDirective', function() {
      return {
        restrict: 'E',
        compile: function(element, attrs) {
          element.data('givenAttrs', attrs);
```

```
        }
      };
    });
    injector.invoke(function($compile) {
      var el = $('<my-directive my-attr="1" my-other-attr="two"></my-directive>');
      $compile(el);

      expect(el.data('givenAttrs').myAttr).toEqual('1');
      expect(el.data('givenAttrs').myOtherAttr).toEqual('two');
    });
  });

});
```

In this test we expect the `compile` function to receive a second argument, which we attach to the element as a data attribute. We then inspect it to see that it has attributes that correspond to what was present in the DOM.

Apart from camelCasing the names, another thing that will have been done for us is whitespace removal. Any whitespace the attribute values may have had will be gone:

test/compile_spec.js

```
it('trims attribute values', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      restrict: 'E',
      compile: function(element, attrs) {
        element.data('givenAttrs', attrs);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<my-directive my-attr=" val "></my-directive>');
    $compile(el);

    expect(el.data('givenAttrs').myAttr).toEqual('val');
  });
});
```

Let's build support for this to our DOM compiler. We'll need to construct an attributes object for each node we're compiling. This happens inside the node loop in `compileNodes`. We make an attributes object and then pass it first to `collectDirectives` and then to `applyDirectivesToNode`:

src/compile.js

```
function compileNodes($compileNodes) {
  _.forEach($compileNodes, function(node) {
    var attrs = {};
    var directives = collectDirectives(node, attrs);
    var terminal = applyDirectivesToNode(directives, node, attrs);
    if (!terminal && node.childNodes && node.childNodes.length) {
      compileNodes(node.childNodes);
    }
  });
}
```

In `applyDirectivesToNode` the object goes straight to the directive's compile function, which is where our test cases will grab it:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }

    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
      directive.compile($compileNode, attrs);
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });
  return terminal;
}
```

The actual work of collecting the attributes and putting them to the object happens in `collectDirectives`. Since we're already iterating attributes in that function (for the purpose of matching them to directives), we can just add all the attributes to the object during that same iteration:

src/compile.js

```
function collectDirectives(node, attrs) {
  var directives = [];
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName, 'E');
    _.forEach(node.attributes, function(attr) {
      var attrStartName, attrEndName;
      var name = attr.name;
      var normalizedAttrName = directiveNormalize(name.toLowerCase());
      if (/^ngAttr[A-Z]/.test(normalizedAttrName)) {
        name = _.kebabCase(
          normalizedAttrName[6].toLowerCase() +
          normalizedAttrName.substring(7)
        );
      }
      var directiveNName = normalizedAttrName.replace(/(Start|End)$/, '');
      if (directiveIsMultiElement(directiveNName)) {
        if (/Start$/.test(normalizedAttrName)) {
          attrStartName = name;
          attrEndName = name.substring(0, name.length - 5) + 'end';
          name = name.substring(0, name.length - 6);
        }
      }
      normalizedAttrName = directiveNormalize(name.toLowerCase());
      addDirective(directives,
        normalizedAttrName, 'A', attrStartName, attrEndName);
      attrs[normalizedAttrName] = attr.value.trim();
    });
    _.forEach(node.classList, function(cls) {
      var normalizedClassName = directiveNormalize(cls);
      addDirective(directives, normalizedClassName, 'C');
    });
  } else if (node.nodeType === Node.COMMENT_NODE) {
    var match = /^\s*directive\:\s*([\d\w\-_]+)/.exec(node.nodeValue);
    if (match) {
      addDirective(directives, directiveNormalize(match[1]), 'M');
    }
  }
  directives.sort(byPriority);
  return directives;
}
```

Introducing A Test Helper

Before we go any further, there's one thing we could do to make unit testing a lot less repetitive in this chapter. The common pattern of unit tests here is:

1. Register a directive

2. Compile a DOM fragment
3. Grab the attributes object
4. Run some checks on it.

We can introduce a helper function in the `describe` block for attributes that does most of this work:

test/compile_spec.js

```
function registerAndCompile(dirName, domString, callback) {
  var givenAttrs;
  var injector = makeInjectorWithDirectives(dirName, function() {
    return {
      restrict: 'EACM',
      compile: function(element, attrs) {
        givenAttrs = attrs;
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $(domString);
    $compile(el);
    callback(el, givenAttrs);
  });
}
```

This function takes three arguments: A directive name to register, a DOM string to parse and compile, and a callback to invoke when it's all done. The callback will receive the element and the attributes object as arguments.

Now we can change our first two unit tests to the following, much terser format:

test/compile_spec.js

```
it('passes the element attributes to the compile function', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive my-attr="1" my-other-attr="two"></my-directive>',
    function(element, attrs) {
      expect(attrs.myAttr).toEqual('1');
      expect(attrs.myOtherAttr).toEqual('two');
    }
  );
});

it('trims attribute values', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive my-attr=" val "></my-directive>',
    function(element, attrs) {
      expect(attrs.myAttr).toEqual('val');
    }
  );
});
```


Handling Boolean Attributes

Some attributes in HTML are so-called boolean attributes. An example of this is the `disabled` attribute of input fields. Boolean attributes are special in that they don't really have an explicit value. Their mere presence in an element means they should be interpreted as "true".

Since in JavaScript we have a notion of truthiness and falsiness, it would be convenient if we could work with boolean attributes in a way that supports that notion. And in fact, we can do that. Angular coerces the values of boolean attributes, so that they are always `true` in the attributes object:

test/compile_spec.js

```
it('sets the value of boolean attributes to true', function() {
  registerAndCompile(
    'myDirective',
    '<input my-directive disabled>',
    function(element, attrs) {
      expect(attrs.disabled).toBe(true);
    }
  );
});
```

Importantly though, this coercion doesn't happen to any old attribute you add to an element. It only applies to attributes that are defined as boolean in standard HTML. Others will get no special treatment:

test/compile_spec.js

```
it('does not set the value of custom boolean attributes to true', function() {
  registerAndCompile(
    'myDirective',
    '<input my-directive whatever>',
    function(element, attrs) {
      expect(attrs.whatever).toEqual('');
    }
  );
});
```

In `collectDirectives`, where we set the attribute value to the attributes object, we should just use the value `true` if we deem this to be a boolean attribute:

src/compile.js


```
attrs[normalizedAttrName] = attr.value.trim();
if (isBooleanAttribute(node, normalizedAttrName)) {
  attrs[normalizedAttrName] = true;
}
```

The new `isBooleanAttribute` function checks two things: Whether this attribute name is one of the standard boolean attribute names, and whether the element's name is one where boolean attributes are used:

src/compile.js

```
function isBooleanAttribute(node, attrName) {
  return BOOLEAN_ATTRS[attrName] && BOOLEAN_ELEMENTS[node.nodeName];
}
```

The `BOOLEAN_ATTRS` constant contains the (normalized) standard boolean attribute names:

src/compile.js

```
var BOOLEAN_ATTRS = {
  multiple: true,
  selected: true,
  checked: true,
  disabled: true,
  readOnly: true,
  required: true,
  open: true
};
```

The `BOOLEAN_ELEMENTS` constant contains the element names we want to match. The names are in uppercase because that is how the DOM reports node names:

src/compile.js

```
var BOOLEAN_ELEMENTS = {
  INPUT: true,
  SELECT: true,
  OPTION: true,
  TEXTAREA: true,
  BUTTON: true,
  FORM: true,
  DETAILS: true
};
```

Overriding attributes with ng-attr

We've seen that you can prefix an attribute with `ng-attr-`, and the prefix will be stripped as the attributes are collected. But what happens when an element has the same attribute declared both with and without the `ng-attr-` prefix? With our current implementation it depends on which of them happens to be declared first, but Angular actually has order-independent behavior for this: An `ng-attr-` prefixed attribute will always override a non-prefixed one.

test/compile_spec.js

```
it('overrides attributes with ng-attr- versions', function() {
  registerAndCompile(
    'myDirective',
    '<input my-directive ng-attr-whatever="42" whatever="41">',
    function(element, attrs) {
      expect(attrs.whatever).toEqual('42');
    }
  );
});
```

As we go through the attributes, we'll set a flag if we're looking at one that has the `ng-attr-` prefix. Then, as we store the attribute, we first check that it hasn't been stored already. That is, unless it has the `ng-attr-` prefix, in which case it is stored anyway - and thus overrides any previous value:

src/compile.js

```
function collectDirectives(node, attrs) {
  var directives = [];
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName, 'E');
    _.forEach(node.attributes, function(attr) {
      var attrStartName, attrEndName;
      var name = attr.name;
      var normalizedAttrName = directiveNormalize(name.toLowerCase());
      var isNgAttr = /^ngAttr[A-Z]/.test(normalizedAttrName);
      if (isNgAttr) {
        name = _.kebabCase(
          normalizedAttrName[6].toLowerCase() +
          normalizedAttrName.substring(7)
        );
      }
      var directiveNName = normalizedAttrName.replace(/(Start|End)$/, '');
      if (directiveIsMultiElement(directiveNName)) {
        if (/Start$/.test(normalizedAttrName)) {
          attrStartName = name;
          attrEndName = name.substring(0, name.length - 5) + 'end';
```



```
          name = name.substring(0, name.length - 6);
        }
      }
      normalizedAttrName = directiveNormalize(name.toLowerCase());
      addDirective(directives, normalizedAttrName, 'A',
        attrStartName, attrEndName);
      if (isNgAttr || !attrs.hasOwnProperty(normalizedAttrName)) {
        attrs[normalizedAttrName] = attr.value.trim();
        if (isBooleanAttribute(node, normalizedAttrName)) {
          attrs[normalizedAttrName] = true;
        }
      }
    });
    _.forEach(node.classList, function(cls) {
      var normalizedClassName = directiveNormalize(cls);
      addDirective(directives, normalizedClassName, 'C');
    });
  } else if (node.nodeType === Node.COMMENT_NODE) {
    var match = /^\s*directive\:\s*([\d\w\-_]+)/.exec(node.nodeValue);
    if (match) {
      addDirective(directives, directiveNormalize(match[1]), 'M');
    }
  }
  directives.sort(byPriority);
  return directives;
}
```

Setting Attributes

An object of normalized attributes is moderately useful in itself, but things become much more powerful when we add an ability to not only read but also *write* attributes through that object. For that purpose, there is a `$set` method on the object:

test/compile_spec.js

```
it('allows setting attributes', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive attr="true"></my-directive>',
    function(element, attrs) {
      attrs.$set('attr', 'false');
      expect(attrs.attr).toEqual('false');
    }
  );
});
```


The attributes object now has a method, and it would make sense to define that method in its prototype. That would suggest using a constructor. That is exactly what Angular does: There is an `Attributes` constructor defined within the `$get` method of the compile provider. It takes an element as its argument:

src/compile.js

```
this.$get = ['$injector', function($injector) {

  function Attributes(element) {
    this.$$element = element;
  }

  // ...

}];
```

We can now switch the attributes construction in `compileNodes` to use the new constructor instead of using an ad-hoc object literal:

src/compile.js

```
function compileNodes($compileNodes) {
  _.forEach($compileNodes, function(node) {
    var attrs = new Attributes($(node));
    var directives = collectDirectives(node, attrs);
    var terminal = applyDirectivesToNode(directives, node, attrs);
    if (!terminal && node.childNodes && node.childNodes.length) {
      compileNodes(node.childNodes);
    }
  });
}
```

The one method we need for the time being is `$set`. To make our first test pass it can just set the attribute's new value on itself:

src/compile.js

```
function Attributes(element) {
  this.$$element = element;
}

Attributes.prototype.$set = function(key, value) {
  this[key] = value;
};
```


When you set an attribute, it is fair to expect that it should flush the corresponding attribute to the DOM as well, and not just change it in the JavaScript object:

test/compile_spec.js

```
it('sets attributes to DOM', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive attr="true"></my-directive>',
    function(element, attrs) {
      attrs.$set('attr', 'false');
      expect(element.attr('attr')).toEqual('false');
    }
  );
});
```

The `$set` method does do this, by using the element that was given to the `Attributes` constructor:

src/compile.js

```
Attributes.prototype.$set = function(key, value) {
  this[key] = value;
  this.$$element.attr(key, value);
};
```

You can prevent this behavior though, by passing a third argument to `$set` with a value of `false` (any falsy value will not do here - it needs to be a concrete `false`):

test/compile_spec.js

```
it('does not set attributes to DOM when flag is false', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive attr="true"></my-directive>',
    function(element, attrs) {
      attrs.$set('attr', 'false', false);
      expect(element.attr('attr')).toEqual('true');
    }
  );
});
```

In the implementation we should make a decision about flushing the value to the DOM by comparing the third argument to `false`:

src/compile.js


```
Attributes.prototype.$set = function(key, value, writeAttr) {
  this[key] = value;
  if (writeAttr !== false) {
    this.$$element.attr(key, value);
  }
};
```

And why would such a feature be useful? Why would you want to set an attribute on an element, but not really change it in the DOM? Here we touch on the other main reason why the `Attributes` object exists, apart from DOM manipulation: Directive-to-directive communication.

Because of the way we constructed the attributes object, the same exact object is shared by all the directives of an element:

test/compile_spec.js

```
it('shares attributes between directives', function() {
  var attrs1, attrs2;
  var injector = makeInjectorWithDirectives({
    myDir: function() {
      return {
        compile: function(element, attrs) {
          attrs1 = attrs;
        }
      };
    },
    myOtherDir: function() {
      return {
        compile: function(element, attrs) {
          attrs2 = attrs;
        }
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div my-dir my-other-dir></div>');
    $compile(el);
    expect(attrs1).toBe(attrs2);
  });
});
```

Because they share the same attributes object, the directives can use that object to send information to each other.

Since DOM access is generally much more expensive than pure JavaScript code, Angular provides the optional third argument on `$set` for optimization purposes, for those cases where you don't actually care about the DOM but only want to let other directives know about the attribute change.

Setting Boolean Properties

When you set an attribute, another thing Angular does is to set it as a property using jQuery's prop function, if it looks like a boolean attribute:

test/compile_spec.js

```
it('sets prop for boolean attributes', function() {
  registerAndCompile(
    'myDirective',
    '<input my-directive>',
    function(element, attrs) {
      attrs.$set('disabled', true);
      expect(element.prop('disabled')).toBe(true);
    }
  );
});
```

Crucially, this also happens when we choose not to flush things to the DOM. This is useful when we want to change DOM *properties* (such as disabled), but not necessarily DOM/HTML *attributes*:

test/compile_spec.js

```
it('sets prop for boolean attributes even when not flushing', function() {
  registerAndCompile(
    'myDirective',
    '<input my-directive>',
    function(element, attrs) {
      attrs.$set('disabled', true, false);
      expect(element.prop('disabled')).toBe(true);
    }
  );
});
```

In the `$set` method we set the prop if the attribute looks like a boolean one. We do this regardless of the `writeAttr` flag's value:

src/compile.js

```
Attributes.prototype.$set = function(key, value, writeAttr) {
  this[key] = value;

  if (isBooleanAttribute(this.$$element[0], key)) {
    this.$$element.prop(key, value);
  }

  if (writeAttr !== false) {
    this.$$element.attr(key, value);
  }
};
```


Another reason for using the `prop` function is that not all versions of jQuery have dealt with attrs and props consistently. This does not really apply to our project because we control the version of jQuery, but it is good to know this is happening in your Angular applications.

Denormalizing Attribute Names for The DOM

With the attributes object we are using *normalized* attribute names: The names are not exactly as they are in the DOM, but instead they're made more convenient for JavaScript consumption. Most notably, instead of hyphenated-attribute-names we use camelCasedAttributeNames. Also, as we saw in the previous chapter, several special prefixes that the attributes may have had will be gone.

When we *set* an attribute, this becomes a problem. We can't set a normalized attribute name on the DOM, because that's unlikely to work as we expect (apart from simple names like `'attr'` that we have been using so far). We need to *denormalize* the attribute name when we flush it to the DOM.

There are several ways to do this attribute name denormalization. The most straightforward of these is to just supply a denormalized attribute name when we call `$set`. It can be given as the fourth argument:

test/compile_spec.js

```
it('denormalizes attribute name when explicitly given', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive some-attribute="42"></my-directive>',
    function(element, attrs) {
      attrs.$set('someAttribute', 43, true, 'some-attribute');
      expect(element.attr('some-attribute')).toEqual('43');
    }
  );
});
```

In `$set` we'll use the fourth argument as the name when setting the DOM attribute. If a fourth argument wasn't given, we'll use the first argument as before:

src/compile.js

```
Attributes.prototype.$set = function(key, value, writeAttr, attrName) {
  this[key] = value;

  if (isBooleanAttribute(this.$$element[0], key)) {
    this.$$element.prop(key, value);
```



```
  }

  if (!attrName) {
    attrName = key;
  }

  if (writeAttr !== false) {
    this.$$element.attr(attrName, value);
  }
};
```

This works, but it's not a very user-friendly API. The caller of `$set` has to supply the two versions of the attribute name each time, which is far from optimal.

Another way to denormalize the attribute name is to just snake-case it if not explicitly supplied:

test/compile_spec.js

```
it('denormalizes attribute by snake-casing', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive some-attribute="42"></my-directive>',
    function(element, attrs) {
      attrs.$set('someAttribute', 43);
      expect(element.attr('some-attribute')).toEqual('43');
    }
  );
});
```

For this we can just use the `_.kebabCase` function provided by LoDash:

src/compile.js

```
Attributes.prototype.$set = function(key, value, writeAttr, attrName) {
  this[key] = value;

  if (isBooleanAttribute(this.$$element[0], key)) {
    this.$$element.prop(key, value);
  }

  if (!attrName) {
    attrName = _.kebabCase(key, '-');
  }

  if (writeAttr !== false) {
    this.$$element.attr(attrName, value);
  }
};
```


Even this is not quite optimal though: As we discussed, Angular supports various prefixes on DOM attributes, which are stripped during normalization. It is likely that when you `$set` an attribute, you want the DOM attribute with the original prefix to be updated. But `_.kebabCase` knows nothing about any prefixes the attribute name may have had. We should be able to support the original prefix:

test/compile_spec.js

```
it('denormalizes attribute by using original attribute name', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive x-some-attribute="42"></my-directive>',
    function(element, attrs) {
      attrs.$set('someAttribute', '43');
      expect(element.attr('x-some-attribute')).toEqual('43');
    }
  );
});
```

The one exception is the `ng-attr-` prefix, which is *not* retained when you `$set` an attribute:

test/compile_spec.js

```
it('does not use ng-attr- prefix in denormalized names', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive ng-attr-some-attribute="42"></my-directive>',
    function(element, attrs) {
      attrs.$set('someAttribute', 43);
      expect(element.attr('some-attribute')).toEqual('43');
    }
  );
});
```

We'll need to store a *mapping* of the normalized attribute names to their original names before normalization. This mapping will be stored in a field called `$attr` in `Attributes` instances:

src/compile.js

```
function Attributes(element) {
  this.$$element = element;
  this.$attr = {};
}
```


In `$set` we'll look up the attribute name from `$attr` if it wasn't explicitly given to the function. Only as a last resort will we use `_.kebabCase`, in which case we'll also store the hyphenized version to `$attr` for the benefit of future `$set` invocations:

src/compile.js

```
Attributes.prototype.$set = function(key, value, writeAttr, attrName) {
  this[key] = value;

  if (isBooleanAttribute(this.$$element[0], key)) {
    this.$$element.prop(key, value);
  }

  if (!attrName) {
    if (this.$attr[key]) {
      attrName = this.$attr[key];
    } else {
      attrName = this.$attr[key] = _.kebabCase(key);
    }
  }

  if (writeAttr !== false) {
    this.$$element.attr(attrName, value);
  }
};
```

The mapping object is populated inside `collectDirectives`. It has access to the attributes object, and it'll directly use its `$attr` property to set the normalized-to-denormalized mapping for each of the element's attributes. This is done in the `_.forEach(node.attributes)` loop after processing the `ng-attr-` prefix:

src/compile.js

```
function collectDirectives(node, attrs) {
  var directives = [];
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName, 'E');
    _.forEach(node.attributes, function(attr) {
      var attrStartName, attrEndName;
      var name = attr.name;
      var normalizedAttrName = directiveNormalize(name.toLowerCase());
      var isNgAttr = /^ngAttr[A-Z]/.test(normalizedAttrName);
      if (isNgAttr) {
        name = _.kebabCase(
          normalizedAttrName[6].toLowerCase() +
          normalizedAttrName.substring(7)
        );
        normalizedAttrName = directiveNormalize(name.toLowerCase());
      }
```



```
      attrs.$attr[normalizedAttrName] = name;

      // ...

    });

    // ...

  } // ...
}
```

Finally, when you do supply an explicit argument name as the fourth argument to `$set`, what also happens is the mapping to the denormalized attribute name will be overwritten with the argument you gave. Any calls to `$set` after that will use the denormalized name you explicitly supplied, and the original denormalized name is no longer used:

test/compile_spec.js

```
it('uses new attribute name after once given', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive x-some-attribute="42"></my-directive>',
    function(element, attrs) {
      attrs.$set('someAttribute', 43, true, 'some-attribute');
      attrs.$set('someAttribute', 44);

      expect(element.attr('some-attribute')).toEqual('44');
      expect(element.attr('x-some-attribute')).toEqual('42');
    }
  );
});
```

So, if an `attrName` *is* supplied to `$set`, it also updates it into the `$attr` object:

src/compile.js

```
Attributes.prototype.$set = function(key, value, writeAttr, attrName) {
  this[key] = value;

  if (isBooleanAttribute(this.$$element[0], key)) {
    this.$$element.prop(key, value);
  }

  if (!attrName) {
    if (this.$attr[key]) {
      attrName = this.$attr[key];
    } else {
      attrName = this.$attr[key] = _.kebabCase(key);
```



```
    }
  } else {
    this.$attr[key] = attrName;
  }

  if (writeAttr !== false) {
    this.$$element.attr(attrName, value);
  }
};
```

Observing Attributes

As we've discussed, the `Attributes` object provides a means of communication between the directives of a single element. Attribute changes made by one directive can be seen by another.

It would also be useful for one directive to get *notified* of an attribute change made by another directive, so that when a change happens we would know right away. This could be done by `$watch`ing an attribute value, but there is also a dedicated mechanism Angular provides this purpose, which is a possibility to `$observe` an attribute value:

test/compile_spec.js

```
it('calls observer immediately when attribute is $set', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive some-attribute="42"></my-directive>',
    function(element, attrs) {

      var gotValue;
      attrs.$observe('someAttribute', function(value) {
        gotValue = value;
      });

      attrs.$set('someAttribute', '43');

      expect(gotValue).toEqual('43');
    }
  );
});
```

With `$observe` you can attach a function to `Attributes`, that will be called immediately whenever someone `$set`s that attribute, as we see here.

The `Attributes` object maintains a registry object of observers, where the keys are attribute names and the values are arrays of observer functions for that attribute:

src/compile.js


```
Attributes.prototype.$observe = function(key, fn) {
  this.$$observers = this.$$observers || Object.create(null);
  this.$$observers[key] = this.$$observers[key] || [];
  this.$$observers[key].push(fn);
};
```

> We use `Object.create` with a `null` prototype instead of a simple object literal, to avoid clashes with attribute names that exist on the built-in Object prototype, such as `toString` and `constructor`.

At the end of `$set` we invoke all observers that have been registered for the given attribute:

src/compile.js

```
Attributes.prototype.$set = function(key, value, writeAttr, attrName) {
  this[key] = value;

  if (isBooleanAttribute(this.$$element[0], key)) {
    this.$$element.prop(key, value);
  }

  if (!attrName) {
    if (this.$attr[key]) {
      attrName = this.$attr[key];
    } else {
      attrName = this.$attr[key] = _.kebabCase(key);
    }
  } else {
    this.$attr[key] = attrName;
  }

  if (writeAttr !== false) {
    this.$$element.attr(attrName, value);
  }

  if (this.$$observers) {
    _.forEach(this.$$observers[key], function(observer) {
      try {
        observer(value);
      } catch (e) {
        console.log(e);
      }
    });
  }
};
```

Note that we wrap the observer call in a `try..catch` block. We do this for the same reason we did with `$watch`es and event listeners in Part 1 of the book: If one observer throws an exception, that should not cause other observers to be skipped.

Attribute observing is a traditional application of the Observer Pattern. While the same goals could be achieved by using `$watch`es, what's nice about `$observer`s is that they do not put any pressure on the `$digest`. Whereas a `$watch` function needs to be executed on every digest, an `$observer` only ever executes when the attribute it is observing is set. For the rest of the time it does not cost us any CPU cycles.

Note that `$observer`s do *not* react to attribute changes that happen outside of the `Attributes` object. If you set an attribute to the underlying element using jQuery or raw DOM access, no `$observer`s will fire. Attribute observing is tied to the `$set` function. That's the cost of the performance gain of not using a `$watch`.

So `$observer`s run whenever the corresponding attribute is `$set`, but they are also guaranteed to run once after initially registered. This happens on the first `$digest` that happens after registration:

test/compile_spec.js

```
it('calls observer on next $digest after registration', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive some-attribute="42"></my-directive>',
    function(element, attrs, $rootScope) {

      var gotValue;
      attrs.$observe('someAttribute', function(value) {
        gotValue = value;
      });

      $rootScope.$digest();

      expect(gotValue).toEqual('42');
    }
  );
});
```

The third argument, `$rootScope`, to the test callback function is a new one. We need to pass it from `registerAndCompile`:

test/compile_spec.js

```
function registerAndCompile(dirName, domString, callback) {
  var givenAttrs;
  var injector = makeInjectorWithDirectives(dirName, function() {
    return {
      restrict: 'EACM',
      compile: function(element, attrs) {
        givenAttrs = attrs;
```



```
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $(domString);
    $compile(el);
    callback(el, givenAttrs, $rootScope);
  });
}
```

In order to tap into the `$digest` from `Attributes`, we need access to a Scope there as well. We can just inject `$rootScope` to the `$get` method of the `CompileProvider` and use that:

src/compile.js

```
this.$get = ['$injector', '$rootScope', function($injector, $rootScope) {

  // ...

}];
```

Now we can add a callback for the next `$digest` with `$evalAsync`. In the callback we just call the observer function with the current value of the attribute:

src/compile.js

```
Attributes.prototype.$observe = function(key, fn) {
  var self = this;
  this.$$observers = this.$$observers || Object.create(null);
  this.$$observers[key] = this.$$observers[key] || [];
  this.$$observers[key].push(fn);
  $rootScope.$evalAsync(function() {
    fn(self[key]);
  });
};
```

Even though observers generally have nothing to do with the digest, they still use it for the initial invocation. `Scope.$evalAsync` just provides a convenient way to do the initial invocation asynchronously. The same could also have been achieved with a timeout, but this is how Angular does it.

Finally, an observer can be removed in the same way a watcher or an event listener can: By invoking a deregistration function that you get as the return value from `$observe`.

test/compile_spec.js

```
it('lets observers be deregistered', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive some-attribute="42"></my-directive>',
    function(element, attrs) {

      var gotValue;
      var remove = attrs.$observe('someAttribute', function(value) {
        gotValue = value;
      });

      attrs.$set('someAttribute', '43');
      expect(gotValue).toEqual('43');

      remove();
      attrs.$set('someAttribute', '44');
      expect(gotValue).toEqual('43');
    }
  );
});
```

The implementation of this function follows the same pattern that we've seen before: Find the index of the function in the observers, and then remove it by splicing the observers at that index:

src/compile.js

```
Attributes.prototype.$observe = function(key, fn) {
  var self = this;
  this.$$observers = this.$$observers || Object.create(null);
  this.$$observers[key] = this.$$observers[key] || [];
  this.$$observers[key].push(fn);
  $rootScope.$evalAsync(function() {
    fn(self[key]);
  });
  return function() {
    var index = self.$$observers[key].indexOf(fn);
    if (index >= 0) {
      self.$$observers[key].splice(index, 1);
    }
  };
};
```

Providing Class Directives As Attributes

We have now covered how Angular makes an element's attributes available through the `Attributes` object. That is not the only content the `Attributes` object may have, though.

There are a few special cases related to class and comment directives that cause attributes to be populated.

Firstly, when there is a class directive, that directive's name will be present in attributes:

test/compile_spec.js

```
it('adds an attribute from a class directive', function() {
  registerAndCompile(
    'myDirective',
    '<div class="my-directive"></div>',
    function(element, attrs) {
      expect(attrs.hasOwnProperty('myDirective')).toBe(true);
    }
  );
});
```

In `collectDirectives` we'll put the directive name into the attributes:

src/compile.js

```
_.forEach(node.classList, function(cls) {
  var normalizedClassName = directiveNormalize(cls);
  addDirective(directives, normalizedClassName, 'C');
  attrs[normalizedClassName] = undefined;
});
```

The value of this attribute will be `undefined` but it is still *present* in the attributes object, so the `hasOwnProperty` check returns `true`.

The class should not *always* be put in the attributes though. It should only happen for classes that actually match directives, but our current implementation adds *all* classes:

test/compile_spec.js

```
it('does not add attribute from class without a directive', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive class="some-class"></my-directive>',
    function(element, attrs) {
      expect(attrs.hasOwnProperty('someClass')).toBe(false);
    }
  );
});
```

As we add the attribute for the class, we'll need to first check whether a directive was actually matched on it. We can expect the `addDirective` function to return information about this:

src/compile.js


```
_.forEach(node.classList, function(cls) {
  var normalizedClassName = directiveNormalize(cls);
  if (addDirective(directives, normalizedClassName, 'C')) {
    attrs[normalizedClassName] = undefined;
  }
});
```

So in `addDirective` we'll need to return a value, telling the caller whether a directive was added or not:

src/compile.js

```
function addDirective(directives, name, mode, attrStartName, attrEndName) {
  var match;
  if (hasDirectives.hasOwnProperty(name)) {
    var foundDirectives = $injector.get(name + 'Directive');
    var applicableDirectives = _.filter(foundDirectives, function(dir) {
      return dir.restrict.indexOf(mode) !== -1;
    });
    _.forEach(applicableDirectives, function(directive) {
      if (attrStartName) {
        directive = _.create(directive, {$$start: attrStartName, $$end: attrEndName});
      }
      directives.push(directive);
      match = directive;
    });
  }
  return match;
}
```

The concrete value the function returns will be either `undefined` or one of the matched directive definition objects, although in this use case we only check if the value is truthy or not.

So the class attribute value is `undefined` by default, but it doesn't necessarily have to be. You can also provide a *value* for the attribute by adding a colon after the class name. The value may even have whitespace in it:

test/compile_spec.js

```
it('supports values for class directive attributes', function() {
  registerAndCompile(
    'myDirective',
    '<div class="my-directive: my attribute value"></div>',
    function(element, attrs) {
      expect(attrs.myDirective).toEqual('my attribute value');
    }
  );
});
```


The attribute value by default consumes the rest of the `class` attribute, but it can also be terminated using a semicolon. After the semicolon you can add other CSS classes, which may or may not be related to directives:

test/compile_spec.js

```
it('terminates class directive attribute value at semicolon', function() {
  registerAndCompile(
    'myDirective',
    '<div class="my-directive: my attribute value; some-other-class"></div>',
    function(element, attrs) {
      expect(attrs.myDirective).toEqual('my attribute value');
    }
  );
});
```

What we now have in the element's CSS `class` attribute isn't straightforwardly consumable with the `classList` array, which is what we used in the previous chapter. We need to switch to regex matching the `className` string in order to support the attribute value syntax.

First, let's reorganize `collectDirectives` just slightly to set things up. We're going to need a local variable for holding regex matches. For that we'll use one called `match`. Such a variable is already used in comment processing, so we'll pull the variable declaration up to the top-level of the function. Then we'll replace the `classList` forEach loop with a placeholder for where we'll do the class parsing - we only do so if the element in fact has a non-empty `className` property:

src/compile.js

```
function collectDirectives(node, attrs) {
  var directives = [];
  var match;
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName, 'E');
    _.forEach(node.attributes, function(attr) {
      var attrStartName, attrEndName;
      var name = attr.name;
      var normalizedAttrName = directiveNormalize(name.toLowerCase());
      var isNgAttr = /^ngAttr[A-Z]/.test(normalizedAttrName);
      if (isNgAttr) {
        name = _.kebabCase(
          normalizedAttrName[6].toLowerCase() +
          normalizedAttrName.substring(7)
        );
        normalizedAttrName = directiveNormalize(name.toLowerCase());
      }
```



```
      attrs.$attr[normalizedAttrName] = name;

      var directiveNName = normalizedAttrName.replace(/(Start|End)$/, '');
      if (directiveIsMultiElement(directiveNName)) {
        if (/Start$/.test(normalizedAttrName)) {
          attrStartName = name;
          attrEndName = name.substring(0, name.length - 5) + 'end';
          name = name.substring(0, name.length - 6);
        }
      }
      normalizedAttrName = directiveNormalize(name.toLowerCase());
      addDirective(directives, normalizedAttrName, 'A', attrStartName, attrEndName);
      if (isNgAttr || !attrs.hasOwnProperty(normalizedAttrName)) {
        attrs[normalizedAttrName] = attr.value.trim();
        if (isBooleanAttribute(node, normalizedAttrName)) {
          attrs[normalizedAttrName] = true;
        }
      }

    });
    var className = node.className;
    if (_.isString(className) && !_.isEmpty(className)) {

    }
  } else if (node.nodeType === Node.COMMENT_NODE) {
    match = /^\s*directive\:\s*([\d\w\-_]+)/.exec(node.nodeValue);
    if (match) {
      addDirective(directives, directiveNormalize(match[1]), 'M');
    }
  }
  directives.sort(byPriority);
  return directives;
}
```

You may have guessed that just like function argument parsing in `$injector`, this is a task that calls for some fairly involved regular expression work. We need to craft a regex that:

- Matches class names - several of them separated with whitespace
- Optionally matches a value for each class name if there is a colon following it. The value may contain whitespace.
- Terminates the (optional) value at semicolon and matches the next class name(s).

Without further ado, here is the regex we'll be using:

```
/([\d\w\-_]+)(?:\:([^;]+))?;?/
```

- `([\d\w\-_]+)` matches one or more digits, word characters, hyphens, or underscores and captures them in a group. This matches a class name.

- `(?:` begins a non-capturing group for the value of the attribute. At the end of the expression `)?` closes the non-capturing group and marks it as optional, and `;?` adds an optional semi-colon at the end.
- Inside the non-capturing group, `\:` matches a colon character and `([^;]+)` matches one or more characters other than semicolon, and captures them in another group. This is the value of the attribute.

If we apply this regular expression to the class name in a loop, we can consume the class name, each iteration either consuming just a class name or a class name with followed by a value:

src/compile.js

```
className = node.className;
if (_.isString(className) && !_.isEmpty(className)) {
  while ((match = /([\d\w\-_]+)(?:\:([^;]+))?;?/.exec(className))) {

    className = className.substr(match.index + match[0].length);
  }
}
```

In order to not create an infinite loop, at each iteration we need to replace the value of `className` with the remainder that follows whatever was just matched. We do that by taking a substring of `className` that begins from the end of the match.

To pull things together, we can now find the directive, as well as the attribute and its value inside the loop:

src/compile.js

```
className = node.className;
if (isString(className) && className !== '') {
  while ((match = /([\d\w\-_]+)(?:\:([^;]+))?;?/.exec(className))) {
    var normalizedClassName = directiveNormalize(match[1]);
    if (addDirective(directives, normalizedClassName, 'C')) {
      attrs[normalizedClassName] = match[2] ? match[2].trim() : undefined;
    }
    className = className.substr(match.index + match[0].length);
  }
}
```

The directive name will be in the second item of the match array (because it is matched by the first capturing group). The attribute value, should there be any, will be in the third item.

With this, both our old and new test cases for class directives are passing!

Adding Comment Directives As Attributes

Just like class directives end up in the attributes object, so do comment directives. And just like the class directive attributes may be associated with a value, so may comment directives be:

test/compile_spec.js

```
it('adds an attribute with a value from a comment directive', function() {
  registerAndCompile(
    'myDirective',
    '<!-- directive: my-directive and the attribute value -->',
    function(element, attrs) {
      expect(attrs.hasOwnProperty('myDirective')).toBe(true);
      expect(attrs.myDirective).toEqual('and the attribute value');
    }
  );
});
```

Comment directives are easier to process than class directives because there may only be one directive per comment node. Still, just like with class directives, we need to handle this with a regular expression. We already have one for parsing comment directive and we just need to change it a bit. After matching the directive name, we'll allow some whitespace, and then we'll capture the rest of the comment into a group that'll become the attribute's value:

src/compile.js

```
} else if (node.nodeType === Node.COMMENT_NODE) {
  match = /^\s*directive\:\s*([\d\w\-_]+)\s*(.*)$/.exec(node.nodeValue);
  if (match) {
    var normalizedName = directiveNormalize(match[1]);
    if (addDirective(directives, normalizedName, 'M')) {
      attrs[normalizedName] = match[2] ? match[2].trim() : undefined;
    }
  }
}
```

The pattern here is the same as with classes: *If* a matching directive is found for the comment, the normalized directive name is added to attributes.

Manipulating Classes

Apart from making attributes available and observable, the `Attributes` object also provides some helper methods for manipulating the element's CSS classes: Adding them, removing

them, and updating them. These features are not related to class directives in any way - their purpose is just to help with updating the `class` attribute of the DOM element.

> The class manipulation in `Attributes` also integrates tightly with the Angular animation system, but this is a discussion for another time and won't be covered in this chapter.

You can add a CSS class to the element with the `$addClass` method on `Attributes` and remove one with the `$removeClass` method:

test/compile_spec.js

```
it('allows adding classes', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive></my-directive>',
    function(element, attrs) {
      attrs.$addClass('some-class');
      expect(element.hasClass('some-class')).toBe(true);
    }
  );
});

it('allows removing classes', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive class="some-class"></my-directive>',
    function(element, attrs) {
      attrs.$removeClass('some-class');
      expect(element.hasClass('some-class')).toBe(false);
    }
  );
});
```

You could, of course, just as well add classes using the jQuery/jqLite element. This is actually all that our new methods need to do:

src/compile.js

```
Attributes.prototype.$addClass = function(classVal) {
  this.$$element.addClass(classVal);
};

Attributes.prototype.$removeClass = function(classVal) {
  this.$$element.removeClass(classVal);
};
```


The third and final class manipulation method provided by `Attributes` is more interesting. It takes two arguments: A set of new classes for an element, and a set of old classes. It then diffs those two sets and adds all classes that are in the first set but not the second, and removes all classes that are in the second but not the first:

test/compile_spec.js

```
it('allows updating classes', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive class="one three four"></my-directive>',
    function(element, attrs) {
      attrs.$updateClass('one two three', 'one three four');
      expect(element.hasClass('one')).toBe(true);
      expect(element.hasClass('two')).toBe(true);
      expect(element.hasClass('three')).toBe(true);
      expect(element.hasClass('four')).toBe(false);
    }
  );
});
```

So the new function takes the new classes and old classes as arguments, both of them as strings. The first thing we do is split those strings at whitespace to get arrays of individual class names:

src/compile.js

```
Attributes.prototype.$updateClass = function(newClassVal, oldClassVal) {
  var newClasses = newClassVal.split(/\s+/);
  var oldClasses = oldClassVal.split(/\s+/);

};
```

Then we take the set difference of those arrays in both directions, and finally apply the differences to the element accordingly:

src/compile.js

```
Attributes.prototype.$updateClass = function(newClassVal, oldClassVal) {
  var newClasses = newClassVal.split(/\s+/);
  var oldClasses = oldClassVal.split(/\s+/);
  var addedClasses = _.difference(newClasses, oldClasses);
  var removedClasses = _.difference(oldClasses, newClasses);
  if (addedClasses.length) {
    this.$addClass(addedClasses.join(' '));
  }
  if (removedClasses.length) {
    this.$removeClass(removedClasses.join(' '));
  }
};
```


Summary

That second argument to `compile` (and the third argument to `link`) looks innocent but packs a punch: It makes the attributes of the current element conveniently available in a normalized manner. It also allows setting those attributes using the normalized names, which facilitates communication between directives and also provides a nice API for setting attributes on the DOM. It allows observing changes on those attributes, so that you can know immediately when some other directive changes an attribute. Finally, it has some convenience methods for manipulating the element's CSS classes.

In this chapter you have learned:

- How an element's attribute values are made available to directives using the `Attributes` object
- That boolean attribute values are always `true` when present, but only for the standard HTML boolean attributes.
- That the same `Attributes` object is shared by all the directives of an element.
- How attributes can be `$set`, and how the caller can decide whether to also change the DOM or only the corresponding JavaScript property.
- What the attribute name denormalization rules are when `$set`tting an attribute on the DOM.
- How attribute changes can be observed, and how that only applies to attributes set using `$set` and not to attributes changed by other means.
- That class directives also become attributes and may optionally have values attached to them.
- That comment directives also become attributes and may optionally have values attached to them.
- How the `Attributes` object provides some convenience methods for adding, removing, and updating the CSS classes of the element.

In the next chapter we will begin exploring how Scopes and Directives are tied together using a process called *linking*.

Chapter 17

Directive Linking and Scopes

A couple of chapters ago we built up DOM compilation, which is the first piece of the puzzle in Angular's directive system. It consisted of walking over the DOM tree and finding directives to apply to certain elements. But that is not the whole picture. As you may know, there is actually a second crucial part to Angular's directive system, in addition to compilation: *Linking*. That is the topic of this chapter.

Linking is the process of combining the compiled DOM tree with scope objects. By association, this combines the DOM tree with all the application data and functions attached to those scope objects, as well as the watch-based dirty checking system, which we created in Part 1 of the book, and which is a crucial part of Angular's data binding implementation.

So, in this chapter we will bring directives and scopes together. We'll see how new scopes can be requested by directives and how the resulting scope hierarchy typically follows the structure of the DOM hierarchy. We'll also see how *isolate scopes* work, and how they provide a fine-grained and flexible mechanism for passing information around. Let's get to it.

The Public Link Function

In general, applying Angular directives to a DOM tree is a two-step process:

1. *Compile* the DOM tree
2. *Link* the compiled DOM tree to Scopes

The first step we have already covered, so we can focus our attention to the second step.

We have a service called `$compile` for compilation, so one might expect that there is a similar service called `$link` for linking. But this is not the case. There are no top-level facilities for the directive linking process in Angular. Instead, it is all built into `compile.js`.

While both compilation and linking are implemented in the same file, they are still separated from each other. When you call `$compile`, no linking occurs. Instead, when you

call `$compile`, it returns you a *function that you can call later to initiate linking*. This function is called the *public link function*:

test/compile_spec.js

```
it('returns a public link function from compile', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {compile: _.noop};
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive></div>');
    var linkFn = $compile(el);
    expect(linkFn).toBeDefined();
    expect(_.isFunction(linkFn)).toBe(true);
    });
  });
```

So, we need to return such a function from the public `compile` function of `$compile`:

src/compile.js

```
function compile($compileNodes) {
  compileNodes($compileNodes);

  return function publicLinkFn() {

  };
}
```

And what does this function do? It does many things, as we will see, but the very first thing it does is to attach some debug information to the DOM. Specifically, the function *takes a scope object as an argument* and attaches it to the DOM node(s) as jQuery/jqLite data.

Let's create a new `describe` block for this and all the following test cases that deal with linking:

test/compile_spec.js

```
describe('linking', function() {

  it('takes a scope and attaches it to elements', function() {
    var injector = makeInjectorWithDirectives('myDirective', function() {
      return {compile: _.noop};
    });
    injector.invoke(function($compile, $rootScope) {
      var el = $('<div my-directive></div>');
      $compile(el)($rootScope);
      expect(el.data('$scope')).toBe($rootScope);
    });
  });

});
```


In this case we give the `$rootScope` to the public link function and verify that it becomes the `$scope` data attribute of the (top-level) elements given to `$compile`.

Passing this test is easy enough. We can simply attach the data to the elements originally given to `compile`:

src/compile.js

```
function compile($compileNodes) {
  compileNodes($compileNodes);

  return function publicLinkFn(scope) {
    $compileNodes.data('$scope', scope);
  };
}
```

> The attachment of various data attributes and CSS classes to nodes is something you can actually turn off in AngularJS, by calling the `debugInfoEnabled` function of the `$compileProvider`. Angular lets you do this because such information is not always needed in production builds and disabling it gets you some extra performance. Such optimization is not really relevant to our discussion and we will skip it in this book.

Directive Link Functions

If all the public link function did was attach the `$scope` data attribute to the element, it wouldn't be very interesting. But that is certainly not all that it does. The *main* job of the public link function is to initiate the actual linking of directives to the DOM. This is where *directive link functions* come in.

Every directive may include its own link function. If you have ever authored Angular directives, you'll know that this is the case, and you'll also know that this is actually the part of the directive API most often used.

The directive link function is similar to the directive compile function, with two important differences:

1. The two functions get called at different points in time. Directive compilation functions get called during compilation, and linking functions get called during linking. The difference is mostly relevant in terms of what *other directives* do during these two steps. For example, in the presence of DOM-altering directives like `ngRepeat`, your directive will get compiled *once* but linked separately for *each repetition* introduced by `ngRepeat`.
2. The compile function has access to the DOM element and the Attributes object, as we have seen. The link function has access to not only these, but also the Scope object being linked to. This is often where application data and functionality gets attached to the directive.

There are several ways to define directive linking functions. The most straightforward for us to begin with is the most low-level one: When a directive has a `compile` function, it is *expected to return the link function as its return value*. Let's make a test that does this and checks the arguments that the link function should receive:

test/compile_spec.js

```
it('calls directive link function with scope', function() {
  var givenScope, givenElement, givenAttrs;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function() {
        return function link(scope, element, attrs) {
          givenScope = scope;
          givenElement = element;
          givenAttrs = attrs;
        };
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(givenScope).toBe($rootScope);
    expect(givenElement[0]).toBe(el[0]);
    expect(givenAttrs).toBeDefined();
    expect(givenAttrs.myDirective).toBeDefined();
  });
});
```

The link function takes three arguments:

1. A scope, which we expect to be the same scope that we gave to the public link function.
2. An element, which we expect to be the exact element the directive was applied to
3. An Arguments object for the arguments of that element.

The directive API is often criticized for its complexity - and often rightly so - but there *is* a nice bit of symmetry here: Just like the public compile function returns the public link function, an individual directive's compile function returns its link function. This is a pattern repeated on all levels of the compilation and linking process.

Let's make this test pass by connecting the public link function and the directive link functions together. There will be a couple of intermediate steps between the two that we need to take care of.

Our public `compile` function calls the `compileNodes` function, which compiles a collection of nodes. Here's the first of the intermediate steps: The `compileNodes` function should return another linking function for us. We'll call this the *composite link function*, because it will be a *composite* of individual node linking functions. The composite link function is called by the public link function:

src/compile.js


```
function compile($compileNodes) {
  var compositeLinkFn = compileNodes($compileNodes);

  return function publicLinkFn(scope) {
    $compileNodes.data('$scope', scope);
    compositeLinkFn(scope, $compileNodes);
  };
}
```

The composite link function receives two arguments: The scope to link, and the DOM elements to link. The latter is currently exactly the same as the elements that we compiled, but this will not always be the case, as we'll see later.

So, in `compileNodes` we should introduce the composite link function and return it:

src/compile.js

```
function compileNodes($compileNodes) {
  _.forEach($compileNodes, function(node) {
    var attrs = new Attributes($(node));
    var directives = collectDirectives(node, attrs);
    var terminal = applyDirectivesToNode(directives, node, attrs);
    if (!terminal && node.childNodes && node.childNodes.length) {
      compileNodes(node.childNodes);
    }
  });

  function compositeLinkFn(scope, linkNodes) {

  }

  return compositeLinkFn;
}
```

The composite link function's job is to link all the individual nodes. For each of them, there is yet another level of link functions needed: Each node will have a *node link function*, which is returned by the `applyDirectivesToNode` function.

Note that this means `applyDirectivesToNode` will no longer return the `terminal` flag. Instead, the `terminal` flag will be an *attribute* of the node link function:

src/compile.js

```
function compileNodes($compileNodes) {
  _.forEach($compileNodes, function(node) {
    var attrs = new Attributes($(node));
    var directives = collectDirectives(node, attrs);
    var nodeLinkFn;
```



```
    if (directives.length) {
      nodeLinkFn = applyDirectivesToNode(directives, node, attrs);
    }
    if ((!nodeLinkFn || !nodeLinkFn.terminal) &&
        node.childNodes && node.childNodes.length) {
      compileNodes(node.childNodes);
    }
  });

  function compositeLinkFn(scope, linkNodes) {

  }

  return compositeLinkFn;
}
```

Let's collect those node link functions to an array while compiling, along with the index to where we currently are in the node collection:

src/compile.js

```
function compileNodes($compileNodes) {
  var linkFns = [];
  _.forEach($compileNodes, function(node, i) {
    var attrs = new Attributes($(node));
    var directives = collectDirectives(node, attrs);
    var nodeLinkFn;
    if (directives.length) {
      nodeLinkFn = applyDirectivesToNode(directives, node, attrs);
    }
    if ((!nodeLinkFn || !nodeLinkFn.terminal) &&
        node.childNodes && node.childNodes.length) {
      compileNodes(node.childNodes);
    }
    if (nodeLinkFn) {
      linkFns.push({
        nodeLinkFn: nodeLinkFn,
        idx: i
      });
    }
  });

  function compositeLinkFn(scope, linkNodes) {

  }

  return compositeLinkFn;
}
```

What we have at the end of that loop is a collection of objects that store node link functions and indexes. We only collect them for nodes for which there are directives.

In the composite link function we can now *invoke* all the node link functions that we collected:

src/compile.js

```
function compileNodes($compileNodes) {
  var linkFns = [];
  _.forEach($compileNodes, function(node, i) {
    var attrs = new Attributes($(node));
    var directives = collectDirectives(node, attrs);
    var nodeLinkFn;
    if (directives.length) {
      nodeLinkFn = applyDirectivesToNode(directives, node, attrs);
    }
    if ((!nodeLinkFn || !nodeLinkFn.terminal) &&
        node.childNodes && node.childNodes.length) {
      compileNodes(node.childNodes);
    }
    if (nodeLinkFn) {
      linkFns.push({
        nodeLinkFn: nodeLinkFn,
        idx: i
      });
    }
  });

  function compositeLinkFn(scope, linkNodes) {
    _.forEach(linkFns, function(linkFn) {
      linkFn.nodeLinkFn(scope, linkNodes[linkFn.idx]);
    });
  }

  return compositeLinkFn;
}
```

We're expecting a 1-to-1 correspondence with compile nodes and link nodes here, because we expect the indexes to match. This is an assumption that won't hold forever, but it'll do for now.

Finally, as we get to the level of the individual node link functions, we reach the point where we can link the directives themselves. We'll need to collect the directive link functions - which are the results of calling each directive's `compile` function:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
```



```
  var terminal = false;
  var linkFns = [];
  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }

    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
      var linkFn = directive.compile($compileNode, attrs);
      if (linkFn) {
        linkFns.push(linkFn);
      }
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });
  return terminal;
}
```

We can now construct the node link function and return it. The function invokes the directive link functions. We also set the `terminal` flag on it as an attribute, so that `compileNodes` is able to check its value:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var linkFns = [];
  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }

    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
      var linkFn = directive.compile($compileNode, attrs);
      if (linkFn) {
        linkFns.push(linkFn);
```

```
      }
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });

  function nodeLinkFn(scope, linkNode) {
    _.forEach(linkFns, function(linkFn) {
      var $element = $(linkNode);
      linkFn(scope, $element, attrs);
    });
  }
  nodeLinkFn.terminal = terminal;
  return nodeLinkFn;
}
```

Our test case finally passes, and we're successfully doing some linking! As we've seen, there are several steps involved but each one has a specific purpose:

- The public link function is used to link the whole DOM tree that we're compiling
  - The composite link function links a collection of nodes
    - The node link function links all the directives of a single node
      - The directive link function links a single directive.

The first and last of these link functions are the ones that we come into touch with as application developers. The two in the middle are part of the internal machinery of `compile.js`.

Plain Directive Link Functions

It is quite common to have a directive that does nothing in the `compile` function but instead defers all of its work to the `link` function. There's an API shortcut for this in the directive definition object, where you can just introduce the `link` attribute with the link function directly, skipping `compile`:

src/compile.js

```
it('supports link function in directive definition object', function() {
  var givenScope, givenElement, givenAttrs;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      link: function(scope, element, attrs) {
        givenScope = scope;
        givenElement = element;
```


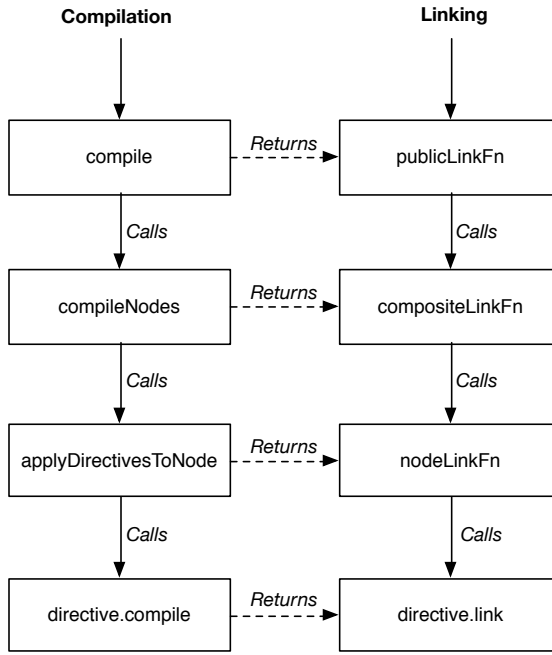

Figure 17.1: The relationships between the compile and link functions

```
        givenAttrs = attrs;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(givenScope).toBe($rootScope);
    expect(givenElement[0]).toBe(el[0]);
    expect(givenAttrs).toBeDefined();
    expect(givenAttrs.myDirective).toBeDefined();
  });
});
```

We can handle this while we're registering the directive factories. If a directive definition object has no `compile` attribute, but does have a `link` attribute, we'll substitute the compile function with a dummy function that just returns the link function. The directive compiler will never know the difference:

src/compile.js

```
$provide.factory(name + 'Directive', ['$injector', function($injector) {
  var factories = hasDirectives[name];
```

```
  return _.map(factories, function(factory, i) {
    var directive = $injector.invoke(factory);
    directive.restrict = directive.restrict || 'EA';
    directive.priority = directive.priority || 0;
    if (directive.link && !directive.compile) {
      directive.compile = _.constant(directive.link);
    }
    directive.name = directive.name || name;
    directive.index = i;
    return directive;
  });
}]);
```

Linking Child Nodes

So far we have been focusing on the linking of nodes on a single level of the DOM hierarchy. Linking should really happen for the whole *DOM tree* that is being compiled, including all the descendants, so let's get that taken care of as well.

When you have a DOM tree with directives on several levels, the directives on lower levels will actually get linked *first*:

test/compile_spec.js

```
it('links directive on child elements first', function() {
  var givenElements = [];
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      link: function(scope, element, attrs) {
        givenElements.push(element);
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive><div my-directive></div></div>');
    $compile(el)($rootScope);
    expect(givenElements.length).toBe(2);
    expect(givenElements[0][0]).toBe(el[0].firstChild);
    expect(givenElements[1][0]).toBe(el[0]);
  });
});
```

In this test we collect all the elements linked to instances of `myDirective`. We apply the directive to two elements: A parent and a child. We then see that the child element was linked before the parent element.

During compilation we handle child nodes by having a recursive call inside `compileNodes` for each node's `childNodes`. Since `compileNodes` now returns a composite link function,

we need to grab hold of the recursive call's return value in order to link the child nodes. So for each node, we need to collect potentially two link functions: The node link function and the composite link function of its children. If either is present, we'll add an element to the `linkFns` array:

src/compile.js

```
function compileNodes($compileNodes) {
  var linkFns = [];
  _.forEach($compileNodes, function(node, idx) {
    var attrs = new Attributes($(node));
    var directives = collectDirectives(node, attrs);
    var nodeLinkFn;
    if (directives.length) {
      nodeLinkFn = applyDirectivesToNode(directives, node, attrs);
    }
    var childLinkFn;
    if ((!nodeLinkFn || !nodeLinkFn.terminal) &&
        node.childNodes && node.childNodes.length) {
      childLinkFn = compileNodes(node.childNodes);
    }
    if (nodeLinkFn || childLinkFn) {
      linkFns.push({
        nodeLinkFn: nodeLinkFn,
        childLinkFn: childLinkFn,
        idx: i
      });
    }
  });

  // ...

}
```

Now, in the composite link function, where we call the node link functions, we'll add an additional argument: The child link function. We expect the node link function to handle linking of the node's children using it.

src/compile.js

```
function compositeLinkFn(scope, linkNodes) {
  _.forEach(linkFns, function(linkFn) {
    linkFn.nodeLinkFn(
      linkFn.childLinkFn,
      scope,
      linkNodes[linkFn.idx]
    );
  });
}
```


The child link function is actually the *first* argument to the node link function, which changes the contract of the node link function and thus breaks some of our existing test cases. Let's fix this by updating `nodeLinkFn` to take the new argument and call it *before the node itself is linked*:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  if (childLinkFn) {
    childLinkFn(scope, linkNode.childNodes);
  }
  _.forEach(linkFns, function(linkFn) {
    var $element = $(linkNode);
    linkFn(scope, $element, attrs);
  });
}
```

Now we've made the test pass, and we're linking children. When a node is linked, its child nodes are linked too.

The problem with this approach is that when a node does not have any directives applied to it, it will not be linked, and neither will its children even if they *do* have directives applied. If we add a test case for this, it will fail:

test/compile_spec.js

```
it('links children when parent has no directives', function() {
  var givenElements = [];
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      link: function(scope, element, attrs) {
        givenElements.push(element);
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div><div my-directive></div></div>');
    $compile(el)($rootScope);
    expect(givenElements.length).toBe(1);
    expect(givenElements[0][0]).toBe(el[0].firstChild);
  });
});
```

In the test we expect the child element to get linked, but that isn't happening. In fact, the test is throwing an error because we're trying to invoke a non-existing node link function from the composite link function.

We can fix that by adding a check for whether there is a node link function or not. If there is, we do what we did before: Invoke it and expect it to link the children. But if there isn't, we're going to call the child link function directly from the composite link function.

src/compile.js


```
function compositeLinkFn(scope, linkNodes) {
  _.forEach(linkFns, function(linkFn) {
    if (linkFn.nodeLinkFn) {
      linkFn.nodeLinkFn(
        linkFn.childLinkFn,
        scope,
        linkNodes[linkFn.idx]
      );
    } else {
      linkFn.childLinkFn(
        scope,
        linkNodes[linkFn.idx].childNodes
      );
    }
  });
}
```

Recall that `childLinkFn` is the *composite link function* of the children, and thus takes two arguments: The scope and the nodes to link.

Pre- And Post-Linking

It may seem peculiar that an element's children get linked before the element itself. There is a good explanation for this, which is that there are actually two different kinds of link functions, and we've only been looking at one of them so far: There are both *prelink* functions and *postlink* functions. The difference between the two is the order in which they get invoked. Prelink functions are called *before* child nodes get linked, and postlink functions are called *after* that.

What we've been calling link functions so far are actually postlink functions, as that is the default when you don't specify one or the other. Another, more explicit way to express what we've done so far is to have a nested `post` key in the directive definition object:

test/compile_spec.js

```
it('supports link function objects', function() {
  var linked;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      link: {
        post: function(scope, element, attrs) {
          linked = true;
        }
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
```



```
    var el = $('<div><div my-directive></div></div>');
    $compile(el)($rootScope);
    expect(linked).toBe(true);
  });
});
```

As we compile a node, we'll need to see if we have a direct link function, or an object of link functions, in which case we'll access the `post` key of the object to find the function itself:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var linkFns = [];
  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }

    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
      var linkFn = directive.compile($compileNode, attrs);
      if (_.isFunction(linkFn)) {
        linkFns.push(linkFn);
      } else if (linkFn) {
        linkFns.push(linkFn.post);
      }
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });

  // ...

}
```

The real reason for having this object notation for link functions is that now we can support both pre- and postlink functions. Let's construct a test case with two levels of nodes and check the order in which linkings get invoked:

test/compile_spec.js

```
it('supports prelinking and postlinking', function() {
  var linkings = [];
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      link: {
        pre: function(scope, element) {
          linkings.push(['pre', element[0]]);
        },
        post: function(scope, element) {
          linkings.push(['post', element[0]]);
        }
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive><div my-directive></div></div>');
    $compile(el)($rootScope);
    expect(linkings.length).toBe(4);
    expect(linkings[0]).toEqual(['pre',  el[0]]);
    expect(linkings[1]).toEqual(['pre',  el[0].firstChild]);
    expect(linkings[2]).toEqual(['post', el[0].firstChild]);
    expect(linkings[3]).toEqual(['post', el[0]]);
  });
});
```

Here we are making sure that the order of the link function invocations is:

1. Parent prelink
2. Child prelink
3. Child postlink
4. Parent postlink

At the moment, the test fails as the prelink functions are not getting invoked at all.

Let's change `applyDirectivesToNode` so that it collects link functions to two separate arrays: `preLinkFns` and `postLinkFns`:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = [], postLinkFns = [];
  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }
```



```
    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
      var linkFn = directive.compile($compileNode, attrs);
      if (_.isFunction(linkFn)) {
        postLinkFns.push(linkFn);
      } else if (linkFn) {
        if (linkFn.pre) {
          preLinkFns.push(linkFn.pre);
        }
        if (linkFn.post) {
          postLinkFns.push(linkFn.post);
        }
      }
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });

  // ..
```

Then let's change the node link function to support the order of invocations we want: First invoke the prelink functions, then invoke the child link functions, and finally invoke the postlink functions:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  var $element = $(linkNode);

  _.forEach(preLinkFns, function(linkFn) {
    linkFn(scope, $element, attrs);
  });
  if (childLinkFn) {
    childLinkFn(scope, linkNode.childNodes);
  }
  _.forEach(postLinkFns, function(linkFn) {
    linkFn(scope, $element, attrs);
  });
}
```

In the previous section we passed the child link function to the node link function instead of just calling it there. Now we see the main reason why: This gives the node link function a chance to call its own prelink functions before invoking the child link function.

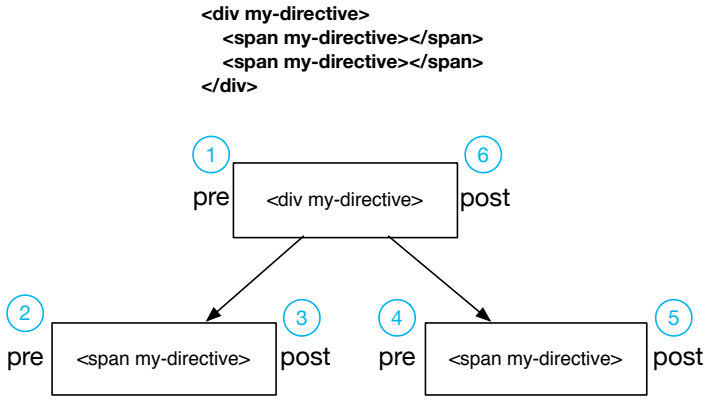

Figure 17.2: The order of link function invocations

There's one more difference between pre- and postlink functions, which has to do with the order in which they're called *within one element*: Prelink functions are called in the directive priority order, but postlink functions should actually be called in *reverse directive priority order*. This is a general rule about postlink functions, both between elements and within a single element: They are invoked in reverse order compared to compilation.

Our current implementation still calls both kinds of link functions in priority order, since we're just calling them in the order in which we collected them during compilation. The following test will not yet pass:

test/compile_spec.js

```
it('reverses priority for postlink functions', function() {
  var linkings = [];
  var injector = makeInjectorWithDirectives({
    firstDirective: function() {
      return {
        priority: 2,
        link: {
          pre: function(scope, element) {
            linkings.push('first-pre');
          },
          post: function(scope, element) {
            linkings.push('first-post');
          }
        }
      };
    },
    secondDirective: function() {
      return {
        priority: 1,
        link: {
          pre: function(scope, element) {
            linkings.push('second-pre');
```



```
          },
          post: function(scope, element) {
            linkings.push('second-post');
          }
        }
      };
    },
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div first-directive second-directive></div>');
    $compile(el)($rootScope);
    expect(linkings).toEqual([
      'first-pre',
      'second-pre',
      'second-post',
      'first-post'
    ]);
  });
});
```

We can fix this by changing the iterator function we use for postlink functions, so that they're iterated from right to left:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  var $element = $(linkNode);

  _.forEach(preLinkFns, function(linkFn) {
    linkFn(scope, $element, attrs);
  });
  if (childLinkFn) {
    childLinkFn(scope, linkNode.childNodes);
  }
  _.forEachRight(postLinkFns, function(linkFn) {
    linkFn(scope, $element, attrs);
  });
}
```

Keeping The Node List Stable for Linking

As we already touched on earlier, the way we have set up the composite link function requires a one-to-one correspondence between compile nodes and link nodes and isn't very robust when it comes to changes in the underlying DOM. Since DOM manipulation is often exactly what directives are used for, this can be a problem. For example, the linking process breaks if we have a directive that inserts new siblings to the elements being linked:

test/compile_spec.js


```
it('stabilizes node list during linking', function() {
  var givenElements = [];
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      link: function(scope, element, attrs) {
        givenElements.push(element[0]);
        element.after('<div></div>');
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div><div my-directive></div><div my-directive></div></div>');
    var el1 = el[0].childNodes[0], el2 = el[0].childNodes[1];
    $compile(el)($rootScope);
    expect(givenElements.length).toBe(2);
    expect(givenElements[0]).toBe(el1);
    expect(givenElements[1]).toBe(el2);
  });
});
```

In this test we have a directive that inserts a new element after the current element during linking. We apply the element to two siblings, and expect both to be linked. What happens instead is one of the *inserted* elements gets linked, and for the second application of the directive, the element used for compilation is different from the element used for linking. This is definitely not what we want.

We can fix this by making our own copy of the node collection before we start running the node link functions. Unlike the raw DOM node collection, this collection will be protected from elements shifting around during linking. We can create the collection by iterating over the indexes in the `linkFns` array:

src/compile.js

```
function compositeLinkFn(scope, linkNodes) {
  var stableNodeList = [];
  _.forEach(linkFns, function(linkFn) {
    var nodeIdx = linkFn.idx;
    stableNodeList[nodeIdx] = linkNodes[nodeIdx];
  });

  _.forEach(linkFns, function(linkFn) {
    if (linkFn.nodeLinkFn) {
      linkFn.nodeLinkFn(
        linkFn.childLinkFn,
        scope,
        stableNodeList[linkFn.idx]
      );
    } else {
```



```
      linkFn.childLinkFn(
        scope,
        stableNodeList[linkFn.idx].childNodes
      );
    }
  });
}
```

Linking Directives Across Multiple Nodes

A couple of chapters ago we saw how directives can be configured as `multiElement` directives and then applied with the `-start` and `-end` suffixes in the DOM. These cases need a bit of special attention while linking, because you would expect the link function of those directives to receive a *collection* of elements from the start to the end element, and currently it's just receiving the start element:

test/compile_spec.js

```
it('invokes multi-element directive link functions with whole group', function() {
  var givenElements;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      multiElement: true,
      link: function(scope, element, attrs) {
        givenElements = element;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $(
      '<div my-directive-start></div>'+
      '<p></p>'+
      '<div my-directive-end></div>'
    );
    $compile(el)($rootScope);
    expect(givenElements.length).toBe(3);
  });
});
```

What we're going to do is add some logic to `applyDirectivesToNode` that knows what to do with multi-element directive applications. But first let's do a tiny bit of refactoring, by introducing a helper function that collects the link functions of a node, so that the directive `_.forEach` loop doesn't grow too large:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = [], postLinkFns = [];

  function addLinkFns(preLinkFn, postLinkFn) {
    if (preLinkFn) {
      preLinkFns.push(preLinkFn);
    }
    if (postLinkFn) {
      postLinkFns.push(postLinkFn);
    }
  }

  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }

    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.compile) {
      var linkFn = directive.compile($compileNode, attrs);
      if (_.isFunction(linkFn)) {
        addLinkFns(null, linkFn);
      } else if (linkFn) {
        addLinkFns(linkFn.pre, linkFn.post);
      }
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });

  // ...

}
```

The new `addLinkFns` function will know some tricks about multi-element directives but first we have to let it know when we're actually dealing with one. We already have the `$$start` and `$$end` attributes that we are attaching to the directive object in these cases (which we do in `addDirective`). These attributes hold the attribute names that were used in the DOM to mark the beginning and end of the directive application. Let's pass them along to `addLinkFns`:

src/compile.js


```
_.forEach(directives, function(directive) {
  if (directive.$$start) {
    $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
  }
  if (directive.compile) {
    var linkFn = directive.compile($compileNode, attrs);
    var attrStart = directive.$$start;
    var attrEnd = directive.$$end;
    if (_.isFunction(linkFn)) {
      addLinkFns(null, linkFn, attrStart, attrEnd);
    } else if (linkFn) {
      addLinkFns(linkFn.pre, linkFn.post, attrStart, attrEnd);
    }
  }
  if (directive.terminal) {
    terminal = true;
    terminalPriority = directive.priority;
  }
});
```

In `addLinkFns` we are going to check if these arguments actually have defined values. If they do, we are going to *wrap* the link functions with special wrappers that know how to resolve the start and end to the collection of elements:

src/compile.js

```
function addLinkFns(preLinkFn, postLinkFn, attrStart, attrEnd) {
  if (preLinkFn) {
    if (attrStart) {
      preLinkFn = groupElementsLinkFnWrapper(preLinkFn, attrStart, attrEnd);
    }
    preLinkFns.push(preLinkFn);
  }
  if (postLinkFn) {
    if (attrStart) {
      postLinkFn = groupElementsLinkFnWrapper(postLinkFn, attrStart, attrEnd);
    }
    postLinkFns.push(postLinkFn);
  }
}
```

The new `groupElementsLinkFnWrapper` returns a wrapped link function, that replaces the `element` given to the original link function with the full group of elements. For collecting that group we already have the function we need: The `groupScan` function that we are using for doing the same thing in the compilation phase:

src/compile.js

```
function groupScan(node, startAttr, endAttr) {
  // ..
}

function groupElementsLinkFnWrapper(linkFn, attrStart, attrEnd) {
  return function(scope, element, attrs) {
    var group = groupScan(element[0], attrStart, attrEnd);
    return linkFn(scope, group, attrs);
  };
}
```

So, what we have for multi-element directives is one additional level of indirection between the public link function and the directive link function: A wrapper that knows how to resolve the element group, given the start element and the start and end attribute names.

Linking And Scope Inheritance

Having gotten the basics of the linking process in order, we arrive at the other crucial theme of this chapter: The ways in which new scopes are created during the directive linking process.

Our code so far takes a Scope as an argument to the public link function, and gives that exact same scope to all the directive link functions. A single scope is shared by all directives in the DOM tree. While this may occur in actual Angular applications too, it is far more common to have directives that request their *own* scope, using the inheritance mechanisms introduced back in Chapter 2. Let's look at how this happens.

A directive may ask for an inherited scope, by introducing a `scope` attribute on the directive definition object, and setting its value to `true`:

test/compile_spec.js

```
it('makes new scope for element when directive asks for it', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: true,
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(givenScope.$parent).toBe($rootScope);
  });
});
```


A `false` value for `scope` is considered equivalent to `undefined`, i.e. the directive should just receive the scope from its environment, which is what we've been doing so far.

When there is at least one directive on an element that requests an inherited scope, *all* directives on that element will receive that inherited scope:

test/compile_spec.js

```
it('gives inherited scope to all directives on element', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        scope: true
      };
    },
    myOtherDirective: function() {
      return {
        link: function(scope) {
          givenScope = scope;
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    $compile(el)($rootScope);
    expect(givenScope.$parent).toBe($rootScope);
  });
});
```

Here we apply two directives on the same element, one of which requests an inherited scope. We check that even the directive that didn't ask for an inherited scope now gets one.

When there is scope inheritance involved with an element, two things are attached to the element:

- An `ng-scope` CSS class
- The new Scope object as jQuery/jqLite data

test/compile_spec.js


```
it('adds scope class and data for element with new scope', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: true,
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(el.hasClass('ng-scope')).toBe(true);
    expect(el.data('$scope')).toBe(givenScope);
  });
});
```

Let's make these test cases pass. The first thing that needs to happen is the detection of directives that request a new scope. We can do this in `applyDirectivesToNode`, where, if we encounter at least one directive with its `scope` attribute set to `true`, we will set a similar `scope` attribute on the node link function:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = [], postLinkFns = [];
  var newScopeDirective;

  function addLinkFns(preLinkFn, postLinkFn, attrStart, attrEnd) {
    // ...
  }

  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }

    if (directive.priority < terminalPriority) {
      return false;
    }

    if (directive.scope) {
      newScopeDirective = newScopeDirective || directive;
    }
    if (directive.compile) {
```



```
      var linkFn = directive.compile($compileNode, attrs);
      var attrStart = directive.$$start;
      var attrEnd = directive.$$end;
      if (_.isFunction(linkFn)) {
        addLinkFns(null, linkFn, attrStart, attrEnd);
      } else if (linkFn) {
        addLinkFns(linkFn.pre, linkFn.post, attrStart, attrEnd);
      }
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });

  function nodeLinkFn(childLinkFn, scope, linkNode) {
    // ...
  }
  nodeLinkFn.terminal = terminal;
  nodeLinkFn.scope = newScopeDirective && newScopeDirective.scope;

  return nodeLinkFn;
}
```

In the composite link function we will now make a new scope if the node link function is marked as one that requests it - i.e. the node had at least one directive that wanted an inherited scope:

src/compile.js

```
_.forEach(linkFns, function(linkFn) {
  if (linkFn.nodeLinkFn) {
    if (linkFn.nodeLinkFn.scope) {
      scope = scope.$new();
    }
    linkFn.nodeLinkFn(
      linkFn.childLinkFn,
      scope,
      stableNodeList[linkFn.idx]
    );
  } else {
    linkFn.childLinkFn(
      scope,
      stableNodeList[linkFn.idx].childNodes
    );
  }
});
```

We should also set the CSS class and data accordingly if there is an inherited scope. The CSS class is added already during compilation - not during linking:

src/compile.js

```
_.forEach($compileNodes, function(node, i) {
  var attrs = new Attributes($(node));
  var directives = collectDirectives(node, attrs);
  var nodeLinkFn;
  if (directives.length) {
    nodeLinkFn = applyDirectivesToNode(directives, node, attrs);
  }
  var childLinkFn;
  if ((!nodeLinkFn || !nodeLinkFn.terminal) &&
      node.childNodes && node.childNodes.length) {
    childLinkFn = compileNodes(node.childNodes);
  }
  if (nodeLinkFn && nodeLinkFn.scope) {
    attrs.$$element.addClass('ng-scope');
  }
  if (nodeLinkFn || childLinkFn) {
    linkFns.push({
      nodeLinkFn: nodeLinkFn,
      childLinkFn: childLinkFn,
      idx: i
    });
  }
});
```

The `$scope` jQuery data is added during linking - since we don't have the scope object until then:

src/compile.js

```
_.forEach(linkFns, function(linkFn) {
  var node = stableNodeList[linkFn.idx];
  if (linkFn.nodeLinkFn) {
    if (linkFn.nodeLinkFn.scope) {
      scope = scope.$new();
      $(node).data('$scope', scope);
    }
    linkFn.nodeLinkFn(
      linkFn.childLinkFn,
      scope,
      node
    );
  } else {
    linkFn.childLinkFn(
      scope,
      node.childNodes
    );
  }
});
```


And that's it!

In Part 1 of the book we discussed that scope inheritance often follows the structure of the DOM tree. Now we see how that actually happens: Directives may ask new scopes to be created, in which case the elements where the directives are applied - as well as all of their children - get the inherited scope.

Isolate Scopes

Back in Chapter 2 we saw the two alternative ways in which you can do Scope inheritance: Prototypal inheritance, and non-prototypal, isolated inheritance. We've already seen how the first approach ties into directives. The remainder of the chapter will focus on the second approach.

As we've learned, isolate scopes are scopes that participate in the scope hierarchy but do *not* inherit the attributes of their parents. They participate in event propagation and digest cycles along with other scopes, but you cannot use them to share arbitrary data from parents to descendants. When you use an isolate scope with a directive, it is easier to make the directive more *modular*, as you can make sure that your directive is more or less isolated from its surrounding environment.

Isolated scopes are not *completely* isolated from their context, however. The directive system allows you to tie attributes from the surrounding environment to isolate scopes, by using *isolate scope bindings*. The main difference between this and normal scope inheritance is that everything you pass into an isolate scope must be explicitly defined as an isolate binding, whereas with normal inheritance all the parent attributes come through the JavaScript object prototype, whether you want them or not.

Before we get into the bindings however, let's get the basics out of the way. An isolate scope is requested by using an object as the value of a directive's `scope` attribute. The scope of that directive will be a child of the scope from the surrounding context, but it will not prototypally inherit from it:

test/compile_spec.js

```
it('creates an isolate scope when requested', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {},
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(givenScope.$parent).toBe($rootScope);
```



```
    expect(Object.getPrototypeOf(givenScope)).not.toBe($rootScope);
  });
});
```

An important point about isolate scope directives, which makes them different from plain inherited scope directives, is that if one directive uses an isolate scope, that scope is not given to other directives on the same element. The scope is isolated *for the directive*, not for the whole element:

test/compile_spec.js

```
it('does not share isolate scope with other directives', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        scope: {}
      };
    },
    myOtherDirective: function() {
      return {
        link: function(scope) {
          givenScope = scope;
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    $compile(el)($rootScope);
    expect(givenScope).toBe($rootScope);
  });
});
```

As we see here, when there are two directives on an element and one of them uses an isolate scope, the second one is still using the scope from the surrounding context.

The same rule also applies to the children of the element. The children are *not* given the isolate scope:

test/compile_spec.js

```
it('does not use isolate scope on child elements', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        scope: {}
```



```
      };
    },
    myOtherDirective: function() {
      return {
        link: function(scope) {
          givenScope = scope;
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive><div my-other-directive></div></div>');
    $compile(el)($rootScope);
    expect(givenScope).toBe($rootScope);
  });
});
```

> There are exceptions to this final rule: Sometimes the children of an element do share the isolate scope. This happens when the children are created by the isolated directive's own template - something we'll look at in later chapters.

Armed with our basic isolate scope test suite, let's start building up the things that we need.

In `applyDirectivesToNode` we're going to detect when a directive requests an isolate scope, and pass that information to `addLinkFns`:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = [], postLinkFns = [];
  var newScopeDirective, newIsolateScopeDirective;

  function addLinkFns(preLinkFn, postLinkFn, attrStart, attrEnd, isolateScope) {
    // ...
  }

  _.forEach(directives, function(directive) {
    if (directive.$$start) {
      $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
    }

    if (directive.priority < terminalPriority) {
      return false;
    }
```

```
    if (directive.scope) {
      if (_.isObject(directive.scope)) {
        newIsolateScopeDirective = directive;
      } else {
        newScopeDirective = newScopeDirective || directive;
      }
    }
    if (directive.compile) {
      var linkFn = directive.compile($compileNode, attrs);
      var isolateScope = (directive === newIsolateScopeDirective);
      var attrStart = directive.$$start;
      var attrEnd = directive.$$end;
      if (_.isFunction(linkFn)) {
        addLinkFns(null, linkFn, attrStart, attrEnd, isolateScope);
      } else if (linkFn) {
        addLinkFns(linkFn.pre, linkFn.post, attrStart, attrEnd, isolateScope);
      }
    }
    if (directive.terminal) {
      terminal = true;
      terminalPriority = directive.priority;
    }
  });

  // ...
}
```

In `addLinkFns` we'll just attach the `isolateScope` flag to the pre and post link functions:

src/compile.js

```
function addLinkFns(preLinkFn, postLinkFn, attrStart, attrEnd, isolateScope) {
  if (preLinkFn) {
    if (attrStart) {
      preLinkFn = groupElementsLinkFnWrapper(preLinkFn, attrStart, attrEnd);
    }
    preLinkFn.isolateScope = isolateScope;
    preLinkFns.push(preLinkFn);
  }
  if (postLinkFn) {
    if (attrStart) {
      postLinkFn = groupElementsLinkFnWrapper(postLinkFn, attrStart, attrEnd);
    }
    postLinkFn.isolateScope = isolateScope;
    postLinkFns.push(postLinkFn);
  }
}
```

Then, in the node link function, we're going to do the actual creation of the isolate scope, which we do if any directive requested it for the element:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  var $element = $(linkNode);

  var isolateScope;
  if (newIsolateScopeDirective) {
    isolateScope = scope.$new(true);
  }

  _.forEach(preLinkFns, function(linkFn) {
    linkFn(scope, $element, attrs);
  });
  if (childLinkFn) {
    childLinkFn(scope, linkNode.childNodes);
  }
  _.forEachRight(postLinkFns, function(linkFn) {
    linkFn(scope, $element, attrs);
  });
}
```

Then we'll pass that isolate scope to any link functions that have the `isolateScope` flag. This means that the link function(s) of the isolate scope directive receive the isolate scope, but others receive the surrounding scope. Also, the child link function never receives the isolate scope:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  var $element = $(linkNode);

  var isolateScope;
  if (newIsolateScopeDirective) {
    isolateScope = scope.$new(true);
  }

  _.forEach(preLinkFns, function(linkFn) {
    linkFn(linkFn.isolateScope ? isolateScope : scope, $element, attrs);
  });
  if (childLinkFn) {
    childLinkFn(scope, linkNode.childNodes);
  }
  _.forEachRight(postLinkFns, function(linkFn) {
    linkFn(linkFn.isolateScope ? isolateScope : scope, $element, attrs);
  });
}
```


As we saw, the isolate scope is not shared with other directives on the same element or child elements. Furthermore, only one directive on an element is allowed to make an isolate scope for itself. Trying to use more than that will throw during compilation:

test/compile_spec.js

```
it('does not allow two isolate scope directives on an element', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        scope: {}
      };
    },
    myOtherDirective: function() {
      return {
        scope: {}
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    expect(function() {
      $compile(el);
    }).toThrow();
  });
});
```

Actually, if there is a directive with an isolate scope on an element, other directives are not allowed to have even *non-isolated*, inherited scopes:

test/compile_spec.js

```
it('does not allow both isolate and inherited scopes on an element', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        scope: {}
      };
    },
    myOtherDirective: function() {
      return {
        scope: true
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
```



```
    expect(function() {
      $compile(el);
    }).toThrow();
  });
});
```

We are going to check these conditions in `applyDirectivesToNode`, where we have two ways in which the rules could be broken:

1. We're encountering an isolate scope directive and have already encountered another isolate scope or inherited scope directive earlier.
2. We're encountering an inherited scope directive and have already encoutered an isolate scope directive earlier.

In both cases, we'll throw with a message many Angular application developers have seen before:

src/compile.js

```
_.forEach(directives, function(directive) {
  if (directive.$$start) {
    $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
  }

  if (directive.priority < terminalPriority) {
    return false;
  }

  if (directive.scope) {
    if (_.isObject(directive.scope)) {
      if (newIsolateScopeDirective || newScopeDirective) {
        throw 'Multiple directives asking for new/inherited scope';
      }
      newIsolateScopeDirective = directive;
    } else {
      if (newIsolateScopeDirective) {
        throw 'Multiple directives asking for new/inherited scope';
      }
      newScopeDirective = newScopeDirective || directive;
    }
  }

  // ...

});
```

Finally, applying an isolate scope directive causes the element to receive the `ng-isolate-scope` CSS class (whereas it will *not* receive the `ng-scope` class) and the scope object itself as jQuery data with the `$isolateScope` key:

test/compile_spec.js


```
it('adds class and data for element with isolated scope', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {},
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(el.hasClass('ng-isolate-scope')).toBe(true);
    expect(el.hasClass('ng-scope')).toBe(false);
    expect(el.data('$isolateScope')).toBe(givenScope);
  });
});
```

Both of these are done in the node link function right when the isolate scope object is created:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  var $element = $(linkNode);

  var isolateScope;
  if (newIsolateScopeDirective) {
    isolateScope = scope.$new(true);
    $element.addClass('ng-isolate-scope');
    $element.data('$isolateScope', isolateScope);
  }

  // ...

}
```

Isolate Attribute Bindings

We now have isolate scopes, but they are all totally blank. That limits their usefulness quite a bit, and as we discussed earlier, there are a few ways you can actually tie data onto them.

One of those ways - the first that we are going to implement - is to have scope attributes that are bound to values in the *element's* attributes. These scope attributes will be *observed*

using the attribute observer mechanism we built in the previous chapter, so that whenever the element's attribute is `$set`, the scope attribute gets updated.

Attribute bindings can bse useful in a couple of ways: You can get easy access to attributes defined on the underlying DOM/HTML element, and you can communicate from other directives to the isolated one by setting attributes that it has bound in this way. This is because all the directives of an element, isolated or not, share the same Attributes object.

Attribute bindings are defined on the directive definition's scope object. The key defines the name of the attribute, and the value is the character `@`, which is short for "attribute binding". Once we have added that, and we `$set` the attribute, it pops onto the isolate scope:

test/compile_spec.js

```
it('allows observing attribute to the isolate scope', function() {
  var givenScope, givenAttrs;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        anAttr: '@'
      },
      link: function(scope, element, attrs) {
        givenScope = scope;
        givenAttrs = attrs;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);

    givenAttrs.$set('anAttr', '42');
    expect(givenScope.anAttr).toEqual('42');
  });
});
```

Let's go ahead and implement this. The first part of processing isolate bindings happens during directive registration. If the directive has an isolate scope definition, we are going to parse its contents for further processing later:

src/compile.js

```
$provide.factory(name + 'Directive', ['$injector', function($injector) {
  var factories = hasDirectives[name];
  return _.map(factories, function(factory, i) {
    var directive = $injector.invoke(factory);
    directive.restrict = directive.restrict || 'EA';
    directive.priority = directive.priority || 0;
    if (directive.link && !directive.compile) {
      directive.compile = _.constant(directive.link);
```



```
    }
    if (_.isObject(directive.scope)) {
      directive.$$isolateBindings = parseIsolateBindings(directive.scope);
    }
    directive.name = directive.name || name;
    directive.index = i;
    return directive;
  });
}]);
```

`parseIsolateBindings` is a new function which we can add to the top level of `compile.js`. It takes the scope definition object and returns an object of the parsed binding rules from that object. For now, we'll just take the definition almost as-is. The function takes a scope definition like

```
{
  anAttr: '@'
}
```

And returns the following:

```
{
  anAttr: {
    mode: '@'
  }
}
```

We'll build up more features later, but here's the implementation we need for now:

src/compile.js

```
function parseIsolateBindings(scope) {
  var bindings = {};
  _.forEach(scope, function(definition, scopeName) {
    bindings[scopeName] = {
      mode: definition
    };
  });
  return bindings;
}
```

The second part of processing isolate bindings is to actually do the binding when the directive is linked. This happens in the node link function, where we iterate over the `$$isolateBindings` object created earlier:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  var $element = $(linkNode);

  var isolateScope;
  if (newIsolateScopeDirective) {
    isolateScope = scope.$new(true);
    $element.addClass('ng-isolate-scope');
    $element.data('$isolateScope', isolateScope);
    _.forEach(newIsolateScopeDirective.$$isolateBindings,
      function(definition, scopeName) {

    });
  }

  // ...

}
```

At this point, we check if the mode of the binding is `@` for attribute binding, and if so, add an observer on the element's attributes for the corresponding attribute. The observer puts the attribute value on the scope.

src/compile.js

```
_.forEach(newIsolateScopeDirective.$$isolateBindings,
  function(definition, scopeName) {
  switch (definition.mode) {
    case '@':
      attrs.$observe(scopeName, function(newAttrValue) {
        isolateScope[scopeName] = newAttrValue;
      });
      break;
  }
});
```

Since `$observe` is called only the next time the attribute changes, and we stilll want to have the value on the scope if it never actually does, we'll want to put the initial value of the attribute on the scope *immediately*, so that it is already there when the link function is run:

test/compile_spec.js

```
it('sets initial value of observed attr to the isolate scope', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        anAttr: '@'
```

```
      },
      link: function(scope, element, attrs) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive an-attr="42"></div>');
    $compile(el)($rootScope);
    expect(givenScope.anAttr).toEqual('42');
  });
});
```

We can do this at the time when we register the observer:

src/compile.js

```
_.forEach(newIsolateScopeDirective.$$isolateBindings,
  function(definition, scopeName) {
  switch (definition.mode) {
    case '@':
      attrs.$observe(scopeName, function(newAttrValue) {
        isolateScope[scopeName] = newAttrValue;
      });
      if (attrs[scopeName]) {
        isolateScope[scopeName] = attrs[scopeName];
      }
      break;
  }
});
```

At this point we always have a one-to-one correspondence between the name of the attribute on the element and the name of the attribute on the isolate scope. But you can also specify a different name for the scope attribute. This happens by using the *scope attribute name* as the key in the scope definition, and specifying the *element attribute name* as a suffix of the '@' character in the value:

test/compile_spec.js

```
it('allows aliasing observed attribute', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        aScopeAttr: '@anAttr'
      },
      link: function(scope, element, attrs) {
        givenScope = scope;
```

```
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive an-attr="42"></div>');
    $compile(el)($rootScope);
    expect(givenScope.aScopeAttr).toEqual('42');
  });
});
```

We need to do some parsing of the scope definition values to grab the scope aliases. As with dependency injection, a regex will come in handy here. We can use the following, which matches an `@` character, and then zero or more word characters, which it captures into a group. It also supports whitespace around and between all the characters that we're interested in:

```
/\s*@\s*(\w*)\s*/
```

Using this regex, we'll set an `attrName` key on the binding. Since the alias is optional, the `attrName` can also just be the scope name - which is the use case we were looking at earlier:

src/compile.js

```
function parseIsolateBindings(scope) {
  var bindings = {};
  _.forEach(scope, function(definition, scopeName) {
    var match = definition.match(/\s*@\s*(\w*)\s*/);
    bindings[scopeName] = {
      mode: '@',
      attrName: match[1] || scopeName
    };
  });
  return bindings;
}
```

Now we must use the `attrName` when accessing the element attribute while setting up the isolate binding:

src/compile.js

```
_.forEach(newIsolateScopeDirective.$$isolateBindings,
  function(definition, scopeName) {
  var attrName = definition.attrName;
  switch (definition.mode) {
    case '@':
      attrs.$observe(attrName, function(newAttrValue) {
        isolateScope[scopeName] = newAttrValue;
      });
```



```
      if (attrs[attrName]) {
        isolateScope[scopeName] = attrs[attrName];
      }
      break;
  }
});
```

Bi-Directional Data Binding

While attribute binding can often be useful, probably the most widely used isolate scope binding mode is *bi-directional data binding*: Connecting scope attributes on the isolate scope to expressions evaluated on the parent scope.

Bi-directional data binding allows for some of the same kind of data sharing between parent and child scopes as non-isolated inherited scopes do, but there are a few crucial differences:

Scope Inheritance	Bi-Directional Data Binding
Everything is shared from the parent to the child.	Only attributes explicitly mentioned in expressions are shared.
One-to-one correspondence between parent and child attributes.	Child attributes may not have matching parent attributes, but but can be any expressions instead.
Assigning an attribute on the child scope *shadows* the parent attribute.	Assigning an attribute in the child *updates* the parent attribute - hence *bi-directional data binding*.

Let's start exploring how this works. The simplest bi-directional data binding configuration you can make is to use the plain `'='` character in the scope definition object. What this says is: "Evaluate this attribute as an expression on the parent scope". The expression itself is not defined in the scope definition object, but in the DOM attribute when the directive is applied. Here's an example where `anAttr` is bound this way, and applied to the expression `'42'`, which evaluates to the number 42:

test/compile_spec.js

```
it('allows binding expression to isolate scope', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        anAttr: '='
      },
      link: function(scope) {
        givenScope = scope;
      }
```



```
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive an-attr="42"></div>');
    $compile(el)($rootScope);

    expect(givenScope.anAttr).toBe(42);
  });
});
```

We expect the corresponding scope attribute to be present after we have linked the directive.

As with attribute binding, you can alias the bi-directional data binding expression, so that the scope attribute doesn't have to be called the same as the DOM element attribute. The aliasing syntax is similar to what we did with attribute binding:

test/compile_spec.js

```
it('allows aliasing expression attribute on isolate scope', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myAttr: '=theAttr'
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive the-attr="42"></div>');
    $compile(el)($rootScope);

    expect(givenScope.myAttr).toBe(42);
  });
});
```

The expressions used in bi-directional data binding are certainly not limited to constant literals like `42`. Where this mechanism really becomes useful is when you reference some attributes of the parent scope in the expressions. This is how you can pass data from a parent scope to an isolated scope - either directly or by deriving a new value in the expression, as we do here:

test/compile_spec.js

```
it('evaluates isolate scope expression on parent scope', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myAttr: '='
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    $rootScope.parentAttr = 41;
    var el = $('<div my-directive my-attr="parentAttr + 1"></div>');
    $compile(el)($rootScope);

    expect(givenScope.myAttr).toBe(42);
  });
});
```

Let's see how we can make these test cases pass. Before anything else, we need to teach our isolate binding parser some new tricks: It needs to be able to parse either attribute bindings (`@`) or two-day data bindings (`=`). The following regular expression will do the job:

```
/\s*([@=])\s*(\w*)\s*/
```

This extends our previous regex by accepting either `@` or `=` as the first character and capturing it into a group so we can grab it later.

If we apply this regex in `parseIsolateBindings`, we can set the `mode` attribute of each binding to this character for later reference:

src/compile.js

```
function parseIsolateBindings(scope) {
  var bindings = {};
  _.forEach(scope, function(definition, scopeName) {
    var match = definition.match(/\s*([@=])\s*(\w*)\s*/);
    bindings[scopeName] = {
      mode: match[1],
      attrName: match[2] || scopeName
    };
  });
  return bindings;
}
```

Notice that the group index of the attribute name shifts as we added a new group before it.

Now we need to handle the new `=` mode bindings as we link the node and create the isolate scope. What we'll do is:

1. Get the expression string applied for this binding in the DOM
2. Parse that string as an Angular expression
3. Evaluate the parsed expression in the context of the parent scope
4. Set the result of the evaluation as an attribute on the isolate scope

Here are those four steps in code:

src/compile.js

```
_.forEach(newIsolateScopeDirective.$$isolateBindings,
  function(definition, scopeName) {
  var attrName = definition.attrName;
  switch (definition.mode) {
    case '@':
      attrs.$observe(attrName, function(newAttrValue) {
        isolateScope[scopeName] = newAttrValue;
      });
      if (attrs[attrName]) {
        isolateScope[scopeName] = attrs[attrName];
      }
      break;
    case '=':
      var parentGet = $parse(attrs[attrName]);
      isolateScope[scopeName] = parentGet(scope);
      break;
  }
});
```

We're using the `$parse` service implemented earlier to parse the expression, but in order to use it we need to inject it to the `$get` function of `CompileProvider`. Let's do that too:

src/compile.js

```
this.$get = ['$injector', '$parse', '$rootScope',
  function($injector, $parse, $rootScope) {

  // ...

};
```

With our first set of unit tests for bi-directional data binding now passing, let's start extending it to cover more ground. One very important aspect of bi-directional data binding is that it does not only bind the data once as our current implementation does, but *watches* the expression and potentially updates the isolate scope attribute to a new value on every digest.

test/compile_spec.js


```
it('watches isolated scope expressions', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myAttr: '='
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-attr="parentAttr + 1"></div>');
    $compile(el)($rootScope);

    $rootScope.parentAttr = 41;
    $rootScope.$digest();
    expect(givenScope.myAttr).toBe(42);
  });
});
```

As we set a parent scope attribute and trigger a digest, we expect the binding expression to be evaluated and and the resulting value updated on isolate scope. This is exactly what watchers are for, so let's add one for the expression we have parsed:

src/compile.js

```
case '=':
  var parentGet = $parse(attrs[attrName]);
  isolateScope[scopeName] = parentGet(scope);
  scope.$watch(parentGet, function(newValue) {
    isolateScope[scopeName] = newValue;
  });
  break;
```

And now we arrive at the actual *bi-directional* part of bi-directional data binding: When you *assign* an attribute bound like this on the isolate scope, it may also affect the other side of the binding on the parent scope. This is something we haven't seen before. Here's an example:

test/compile_spec.js


```
it('allows assigning to isolated scope expressions', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myAttr: '='
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-attr="parentAttr"></div>');
    $compile(el)($rootScope);

    givenScope.myAttr = 42;
    $rootScope.$digest();
    expect($rootScope.parentAttr).toBe(42);
  });
});
```

In the test we have bound the attribute `myAttr` on the isolate scope to an attribute called `parentAttr` on the parent scope. We test that when we assign a value on the *child scope* and run a digest, the same value gets updated on the parent scope.

As soon as data binding works in two ways as it does here, the question of precedence becomes relevant: What if *both* the parent and the child attributes change during the same digest? Which one wins and becomes the value of both the parent and the child? Angular gives precedence to the parent:

test/compile_spec.js

```
it('gives parent change precedence when both parent and child change', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myAttr: '='
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-attr="parentAttr"></div>');
    $compile(el)($rootScope);

    $rootScope.parentAttr = 42;
```



```
    givenScope.myAttr = 43;
    $rootScope.$digest();
    expect($rootScope.parentAttr).toBe(42);
    expect(givenScope.myAttr).toBe(42);
  });
});
```

So this is basically how bi-directional data binding should work. Let's go ahead and build it up. It isn't hugely complicated but there are a few subtle details we need to address.

Firstly, we need a bit more control on what happens when changes occur than a plain watch-listener pair can give us. Instead, to make things easier, we'll *only* register a watch function and omit the listener function completely. We rely on the watch function being called in each digest and do our own change detection in it:

src/compile.js

```
case '=':
  var parentGet = $parse(attrs[attrName]);
  isolateScope[scopeName] = parentGet(scope);
  var parentValueWatch = function() {
    var parentValue = parentGet(scope);
    if (isolateScope[scopeName] !== parentValue) {
      isolateScope[scopeName] = parentValue;
    }
    return parentValue;
  };
  scope.$watch(parentValueWatch);
  break;
```

This implementation still only passes our old unit tests and not the new one, but the code is now in a shape better suited for the bi-directional part of bi-directional data binding.

Right now we're tracking *if* the current value of the watch is different from what's on the isolate scope, but we're not tracking *where* the change has occurred if it has indeed occurred. To help with this, we'll introduce a new variable `lastValue`, which will always store *the value that the parent scope had after the last digest*:

src/compile.js

```
case '=':
  var parentGet = $parse(attrs[attrName]);
  var lastValue = isolateScope[scopeName] = parentGet(scope);
  var parentValueWatch = function() {
    var parentValue = parentGet(scope);
    if (isolateScope[scopeName] !== parentValue) {
      if (parentValue !== lastValue) {
        isolateScope[scopeName] = parentValue;
      }
```



```
    }
    lastValue = parentValue;
    return lastValue;
  };
  scope.$watch(parentValueWatch);
  break;
```

The purpose of this new variable becomes more clear when we consider the situation where the isolate scope attribute's current value is not equal to the parent scope attribute's current value, but *is* equal to `lastValue`? That means the value has changed on the isolate scope and we should update the parent:

src/compile.js

```
case '=':
  var parentGet = $parse(attrs[attrName]);
  var lastValue = isolateScope[scopeName] = parentGet(scope);
  var parentValueWatch = function() {
    var parentValue = parentGet(scope);
    if (isolateScope[scopeName] !== parentValue) {
      if (parentValue !== lastValue) {
        isolateScope[scopeName] = parentValue;
      } else {

      }
    }
    lastValue = parentValue;
    return lastValue;
  };
  scope.$watch(parentValueWatch);
  break;
```

And how can we update the parent? Well, when we implemented expressions we saw how some expressions are *assignable*, meaning that they can not only be evaluated for a value, but can also be updated to a new value using the `assign` function attached to the expression. That is what we are going to use to send the new value to the parent scope:

src/compile.js

```
case '=':
  var parentGet = $parse(attrs[attrName]);
  var lastValue = isolateScope[scopeName] = parentGet(scope);
  var parentValueWatch = function() {
    var parentValue = parentGet(scope);
    if (isolateScope[scopeName] !== parentValue) {
      if (parentValue !== lastValue) {
        isolateScope[scopeName] = parentValue;
      } else {
```



```
        parentValue = isolateScope[scopeName];
        parentGet.assign(scope, parentValue);
      }
    }
    lastValue = parentValue;
    return lastValue;
  };
  scope.$watch(parentValueWatch);
  break;
```

Notice that in addition to using `assign`, we update the local `parentValue` variable, and thus also the `lastValue` variable to the value we have assigned. Everything will be in sync for the next digest.

Notice also how the precedence rule is now implemented here: We have an if-else block where we first see if the parent scope attribute has changed, and only when it hasn't do we consider any changes in the child scope. When both the parent and child have changed, the child change gets ignored and overwritten.

You may have noticed that bi-directional data binding uses *reference watches* to detect value change. While it is not possible to change this behavior to use value-bases watching, it *is* possible to shallow-watch collection changes. Using a special syntax in the scope definition object we can tell the framework that it should use `$watchCollection` instead of `$watch` for the bi-directional data binding. This is useful when, for example, we have a binding to a function call that returns a new array every time:

test/compile_spec.js

```
it('throws when isolate scope expression returns new arrays', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myAttr: '='
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    $rootScope.parentFunction = function() {
      return [1, 2, 3];
    };
    var el = $('<div my-directive my-attr="parentFunction()"></div>');
    $compile(el)($rootScope);
    expect(function() {
      $rootScope.$digest();
    }).toThrow();
  });
});
```


Normal reference watches can't cope with this: The watch sees a new array every time and considers it a new value. The digest runs to the iteration limit and then throws.

To fix this we can introduce a collection watch using the `=*` syntax in the scope definition:

test/compile_spec.js

```
it('can watch isolated scope expressions as collections', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myAttr: '=*'
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    $rootScope.parentFunction = function() {
      return [1, 2, 3];
    };
    var el = $('<div my-directive my-attr="parentFunction()"></div>');
    $compile(el)($rootScope);
    $rootScope.$digest();
    expect(givenScope.myAttr).toEqual([1, 2, 3]);
  });
});
```

We need a new extension to our parsing function again. It should take an optional asterisk after the bi-directional data binding character '=':

```
/\s*(@|=(\*?))\s*(\w*)\s*/
```

This now effectively matches the beginning of the expression as "`@` or (`=` and optionally `*`)". The combination `@*` is not supported because it would not make any sense.

Using this regex, `parseIsolateBindings` can populate a `collection` flag on the binding, based on whether an asterisk was seen or not. Notice that we need to change the match indexes again:

src/compile.js

```
function parseIsolateBindings(scope) {
  var bindings = {};
  _.forEach(scope, function(definition, scopeName) {
    var match = definition.match(/\s*(@|=(\*?))\s*(\w*)\s*/);
```



```
    bindings[scopeName] = {
      mode: match[1][0],
      collection: match[2] === '*',
      attrName: match[3] || scopeName
    };
  });
  return bindings;
}
```

And now we can simply choose to use either `$watch` or `$watchCollection` based on the value of the `collection` flag. The rest of our implementation can remain unchanged:

src/compile.js

```
case '=':
  var parentGet = $parse(attrs[attrName]);
  var lastValue = isolateScope[scopeName] = parentGet(scope);
  var parentValueWatch = function() {
    var parentValue = parentGet(scope);
    if (isolateScope[scopeName] !== parentValue) {
      if (parentValue !== lastValue) {
        isolateScope[scopeName] = parentValue;
      } else {
        parentValue = isolateScope[scopeName];
        parentGet.assign(scope, parentValue);
      }
    }
    lastValue = parentValue;
    return lastValue;
  };
  if (definition.collection) {
    scope.$watchCollection(attrs[attrName], parentValueWatch);
  } else {
    scope.$watch(parentValueWatch);
  }
  break;
```

Note that for the `$watchCollection` case we register our function as the *listener function* and not the watch function. This is mainly because `$watchCollection` does not support omitting the listener function like `$watch` does. This is OK, because we don't really need to do any work as long as the parent attribute keeps pointing to the same array or object: Because we've copied a reference to the same array or object, any mutation within it is automatically "synced". It is only when the parent starts pointing to a *new* array or object that we need to react, and at that point the listener function will fire.

You can also make bi-directional bindings *optional*, which means that if the attribute referenced by the binding does not exist on the DOM element, no watcher will be created. This is done by using the binding syntax `=?` instead of just `=`:

test/compile_spec.js


```
it('does not watch optional missing isolate scope expressions', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myAttr: '=?'
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect($rootScope.$$watchers.length).toBe(0);
  });
});
```

Here we are testing that `$rootScope` has no watchers set up afer the directive is linked. Our current implementation creates a watcher for the expression `undefined`, which is certainly not a catastrophy, but still adds a bit of unnecessary overhead to the application.

Another extension to the binding syntax regexp is called for. It should optionally support a question mark character after the binding specifier:

```
/\s*(@|=(\*?))(\??)\s*(\w*)\s*/
```

> The `?` suffix is also syntactically supported in attribute bindings, but observers will still be added for them even if the attributes don't exist during linking.

Applying this regex, we'll set an `optional` flag on the binding object. Note that the attribute name now switches to index `4` in the match result:

src/compile.js

```
function parseIsolateBindings(scope) {
  var bindings = {};
  _.forEach(scope, function(definition, scopeName) {
    var match = definition.match(/\s*(@|=(\*?))(\??)\s*(\w*)\s*/);
    bindings[scopeName] = {
      mode: match[1][0],
      collection: match[2] === '*',
      optional: match[3],
      attrName: match[4] || scopeName
    };
  });
  return bindings;
}
```


Now, if the attribute is `undefined` during linking and the binding is optional, we will skip the creation of the watcher:

src/compile.js

```
case '=':
  if (definition.optional && !attrs[attrName]) {
    break;
  }
  // ...
```

The last aspect of bi-directional data binding we'll look at is cleaning up after ourselves. Since we have set up a watcher, we'll need to make sure that we also deregister that watcher when the isolate scope is destroyed. Otherwise we would be introducing a memory leak, because the watcher is on the *parent scope*, which may not be destroyed when the isolate scope is.

src/compile.js

```
case '=':
  if (definition.optional && !attrs[attrName]) {
    break;
  }
  var parentGet = $parse(attrs[attrName]);
  var lastValue = isolateScope[scopeName] = parentGet(scope);
  var parentValueWatch = function() {
    var parentValue = parentGet(scope);
    if (isolateScope[scopeName] !== parentValue) {
      if (parentValue !== lastValue) {
        isolateScope[scopeName] = parentValue;
      } else {
        parentValue = isolateScope[scopeName];
        parentGet.assign(scope, parentValue);
      }
    }
    lastValue = parentValue;
    return lastValue;
  };
  var unwatch;
  if (definition.collection) {
    unwatch = scope.$watchCollection(attrs[attrName], parentValueWatch);
  } else {
    unwatch = scope.$watch(parentValueWatch);
  }
  isolateScope.$on('$destroy', unwatch);
  break;
```

And there we have bi-directional data binding!

Expression Binding

The third and final way to bind something on an isolate scope is to bind an expression, which happens by using the `'&'` character in the scope definition object. It is a bit different from the other two in that it is primarily designed to bind *behavior* instead of data: When you apply the directive, you supply an expression the directive can invoke when something happens. This is useful particularly in event-driven directives such as `ngClick`, but has general applicability as well.

The bound expression will be present on the isolate scope as a function, which we can call from, say, the link function:

test/compile_spec.js

```
it('allows binding an invokable expression on the parent scope', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myExpr: '&'
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    $rootScope.parentFunction = function() {
      return 42;
    };
    var el = $('<div my-directive my-expr="parentFunction() + 1"></div>');
    $compile(el)($rootScope);
    expect(givenScope.myExpr()).toBe(43);
  });
});
```

The `myExpr` function on the isolate scope is actually the expression function of the `'parentFunction() + 1'` expression - which invokes `parentFunction` on the parent scope and adds 1 to the result.

To make this work, we need to revisit the isolate scope definition parsing function one more time. In this instance, we add the `&` character as one of the allowed characters for the `mode` attribute:

src/compile.js

```
function parseIsolateBindings(scope) {
  var bindings = {};
  _.forEach(scope, function(definition, scopeName) {
    var match = definition.match(/\s*([@&]|=(\*?))(\??)\s*(\w*)\s*/);
```



```
    bindings[scopeName] = {
      mode: match[1][0],
      collection: match[2] === '*',
      optional: match[3],
      attrName: match[4] || scopeName
    };
  });
  return bindings;
}
```

The rest is actually quite simple. When we encounter an `&`-mode binding, we parse the corresponding attribute into an expression function. We then attach a wrapper function for it on the isolate scope. All expression functions take a scope as the first argument, and the wrapper function will supply that. Crucially, the expression is invoked in the context of the *parent scope*, not the isolate scope. This makes sense since the expression is defined by the user of the directive, not the directive itself:

src/compile.js

```
_.forEach(newIsolateScopeDirective.$$isolateBindings,
  function(definition, scopeName) {
  var attrName = definition.attrName;
  switch (definition.mode) {
    case '@':
      // ...
      break;
    case '=':
      // ...
      break;
    case '&':
      var parentExpr = $parse(attrs[attrName]);
      isolateScope[scopeName] = function() {
        return parentExpr(scope);
      };
      break;
  }
});
```

Our current implementation allows calling functions on the parent scope, but it does not allow any arguments to be passed, which is a bit limiting. We can fix this by making a few changes. The way this works is a bit different from how straight-up function calls work though. Consider the following expression on the parent scope:

```
<div my-expr="parentFunction(a, b)"></div>
```

One might expect to be able to call this from the isolate scope as

```
scope.myExpr(1, 2);
```


But this is not the case. If you think of it from the perspective of the directive's user, `a` and `b` are not necessarily arguments you expect to receive from inside the directive, but might also be attributes on the *parent scope itself*. It would be pretty limiting if you could not use them in these expressions.

So, how can we implement a solution where arguments to isolate scope expressions *may or may not be* supplied from inside the isolate scope. Well, what we can do is *named arguments*, defined as an object:

```
scope.myExpr({a: 1, b: 2});
```

Then, if your system is designed so that only `b` comes from the isolate scope and `a` actually refers to something on the parent scope, that can easily be accomplished.

> When Angular itself uses arguments in isolate scope expressions, it uses the `$` prefix to distinguish them from your own variables. For example, directives like ngClick use `$event` for passing in the DOM event.

Here's the idea expressed as a unit test:

test/compile_spec.js

```
it('allows passing arguments to parent scope expression', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myExpr: '&'
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var gotArg;
    $rootScope.parentFunction = function(arg) {
      gotArg = arg;
    };
    var el = $('<div my-directive my-expr="parentFunction(argFromChild)"></div>');
    $compile(el)($rootScope);
    givenScope.myExpr({argFromChild: 42});
    expect(gotArg).toBe(42);
  });
});
```

Here we have a function defined on the parent scope, which we call from the `myExpr` expression bound to the isolate scope. In the expression we refer to an argument called `argFromChild`, and that is what we pass in as a named argument from the isolate scope.

Let's go ahead and implement this. This is actually remarkably simple, and that's because we already have an existing solution for passing named arguments to expressions: It is the optional second `locals` argument that expression functions take. Our wrapper function takes one argument - the locals - and passes it into the expression function as the *second* argument:

src/compile.js

```
case '&':
  var parentExpr = $parse(attrs[attrName]);
  isolateScope[scopeName] = function(locals) {
    return parentExpr(scope, locals);
  };
  break;
```

So, when the expression `'parentFunction(argFromChild)'` is evaluated, the `argFromChild` lookup is evaluated as part of it. If there is a matching attribute on the locals object (which corresponds to the "named arguments" object passed from the isolate scope), it is used as the value of `argFromChild`. If there is no such attribute on the locals, `argFromChild` is looked up from the (parent) scope.

Finally, an expression binding may also be marked optional in the isolate scope specification. If that is done and the expression is not supplied by the directive user, there will be no function to call on the scope:

test/compile_spec.js

```
it('sets missing optional parent scope expression to undefined', function() {
  var givenScope;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        myExpr: '&?'
      },
      link: function(scope) {
        givenScope = scope;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var gotArg;
    $rootScope.parentFunction = function(arg) {
      gotArg = arg;
    };
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(givenScope.myExpr).toBeUndefined();
  });
});
```


Recall from the expression chapters that when `$parse` is given something it doesn't know how to parse (such as `undefined` or `null`), it just returns the LoDash no-op function `_.noop`. We can make use of that fact and skip creating the binding if that function is what we see for an optional binding:

src/compile.js

```
case '&':
  var parentExpr = $parse(attrs[attrName]);
  if (parentExpr === _.noop && definition.optional) {
    break;
  }
  isolateScope[scopeName] = function(locals) {
    return parentExpr(scope, locals);
  };
  break;
```

Summary

By this point we have both of the core processes of the directive system implemented: Compilation and linking. We understand how directives and scopes interact and how the directive system creates new scopes.

In this chapter you have learned:

- How linking is built into the functions returned by the compile functions.
- How the public link function, the composite link functions, the node link functions, and the directive link functions are all return values of their respective compile functions, and how they are chained together during linking.
- That a directive's compile function should return its link function.
- That you can omit a directive's compile function and supply the link function directly.
- How child nodes get linked whether the parent nodes have directives or not.
- That prelink functions are invoked before child node linking, and postlink functions after it.
- That a link function is always a postlink function unless explicitly defined otherwise.
- How the linking process protects itself from DOM mutations that occur during linking.
- How the nodes for multi-element directives are resolved during linking.
- How directives can request new, inherited scopes.
- That inherited scopes are shared by all the directives in the same element and its children.
- How CSS classes and jQuery data are added to elements that have directives with inherited scopes.
- How directives can request new isolate scopes.
- That isolate scopes are not shared between directives in the same element or its children.
- That there can only be one isolate scope directive per element.

- That there cannot be inherited scope directives on an element when there is an isolate scope directive.
- How element attributes can be bound as observed values on an isolate scope.
- How bi-directional data bindings can be attached on an isolate scope.
- That when both the parent and child change simultaneously in a bi-directional data binding, the parent takes precedence.
- How collections are supported in bi-directional data bindings.
- How invokable expressions can be attached on an isolate scope.
- How named arguments can be used with invokable expressions.

In the next chapter we'll start building on the core directive implementation by introducing *controllers*.

Chapter 18

Controllers

Controllers in AngularJS are an interesting beast. Given the huge popularity of so-called "Model-View Controller" style JavaScript frameworks, and the common (if slightly misleading) characterization of AngularJS as one of those frameworks, it would seem that controllers are one of the most fundamental building blocks of Angular.

In actual fact, controllers do not have the top-level role in Angular that they seem to have at first blush. Yes, controllers are important, even crucially so in many applications. But they are in fact just a part of the directive system. Directives get top billing in Angular, and controllers have a supporting role in helping directives do their job. Standalone controllers, as in the case of `ngController` et al., are mostly just a side effect of the directive system, as we will soon see.

This does not mean that controllers are not interesting or that they are not useful. Quite the opposite, as we will learn in this chapter. There are three pieces to the controller puzzle, and we will look at all of them: The `$controller` provider, the controller integration of the directive compiler, and the `ngController` directive.

The `$controller` provider

It all begins with the ability to bring controller objects to life. There is a specialized service for that, and it is called `$controller`. This service as well as its provider are the first things that we need to add.

The `$controller` service comes as part of the `ng` module, and we can create a test for its presence:

test/angular_public_spec.js

```
it('sets up $controller', function() {
  publishExternalAPI();
  var injector = createInjector(['ng']);
  expect(injector.has('$controller')).toBe(true);
});
```

The provider has its own file - `src/controller.js` - and is set up just like the other providers we have already created. There's a provider constructor, and a `$get` method that will return the concrete `$controller` service:

src/controller.js

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

function $ControllerProvider() {

  this.$get = function() {

  };

}
```

Now we can reference this provider as we register `$controller` as part of the `ng` module:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
  ngModule.provider('$httpBackend', $HttpBackendProvider);
  ngModule.provider('$http', $HttpProvider);
  ngModule.provider('$httpParamSerializer', $HttpParamSerializerProvider);
  ngModule.provider('$httpParamSerializerJQLike',
    $HttpParamSerializerJQLikeProvider);
  ngModule.provider('$compile', $CompileProvider);
  ngModule.provider('$controller', $ControllerProvider);
}
```


Controller Instantiation

Controllers in AngularJS are always created using constructor functions. That is, the kinds of functions you typically denote with a CapitalizedFunctionNames and that are instantiated with the `new` operator:

```
function MyController() {
  this.someField = 42;
}
```

The division of labor is such that the application developer provides the constructor function and the framework's `$controller` service instantiates it when required. The simplest way to do this is to just give `$controller` a constructor function as an argument and expect an instance of that constructor as the return value. This constitutes our first test case for `$controller`, so let's add a new test file for it:

test/controller_spec.js

```
describe('$controller', function() {

  beforeEach(function() {
    delete window.angular;
    publishExternalAPI();
  });

  it('instantiates controller functions', function() {
    var injector = createInjector(['ng']);
    var $controller = injector.get('$controller');

    function MyController() {
      this.invoked = true;
    }

    var controller = $controller(MyController);

    expect(controller).toBeDefined();
    expect(controller instanceof MyController).toBe(true);
    expect(controller.invoked).toBe(true);
  });

});
```

Our test checks that the object we get back is indeed a prototypal instance of the given constructor, and that the constructor was invoked on that object.

At this point we have nothing we could not easily do ourselves by just using the `new` operator directly on the controller constructor. The job of `$controller` gets a bit more interesting when we consider a controller that has dependencies. The constructor is, in fact, invoked with dependency injection:

test/controller_spec.js


```
it('injects dependencies to controller functions', function() {
  var injector = createInjector(['ng', function($provide) {
    $provide.constant('aDep', 42);
  }]);
  var $controller = injector.get('$controller');

  function MyController(aDep) {
    this.theDep = aDep;
  }

  var controller = $controller(MyController);

  expect(controller.theDep).toBe(42);
});
```

Based on these tests, we see that `$controller` is a function, since we are just calling it directly. So the return value of the provider's `$get` method should be a function:

src/controller.js

```
function $ControllerProvider() {

  this.$get = function() {

    return function() {

    };

  };

}
```

This function can take a constructor function as an argument, and return an instantiated version of that constructor, with dependencies injected. In our `$injector` service we have something that does exactly that - `instantiate`. We can use it here:

src/controller.js

```
this.$get = ['$injector', function($injector) {

  return function(ctrl) {
    return $injector.instantiate(ctrl);
  };

}];
```


Not all of the controller constructor's arguments need to be registered to the injector beforehand. We can augment the pre-registered dependencies by supplying an object of *locals* to `$controller` as the second argument:

test/controller_spec.js

```
it('allows injecting locals to controller functions', function() {
  var injector = createInjector(['ng']);
  var $controller = injector.get('$controller');

  function MyController(aDep) {
    this.theDep = aDep;
  }

  var controller = $controller(MyController, {aDep: 42});

  expect(controller.theDep).toBe(42);
});
```

As it happens, `$injector.instantiate` has built-in support for this too:

src/controller.js

```
this.$get = ['$injector', function($injector) {

  return function(ctrl, locals) {
    return $injector.instantiate(ctrl, locals);
  };

}];
```

Controller Registration

Though it's a good start, our fledgling `$controller` provider doesn't yet have much going for it. It's nothing but a wrapper for `$injector.instantiate`, really. This will start changing as we consider a more typical use case for the provider: You can register controllers at *configuration time* and then look them up at runtime.

Here we use a new method called `register` on the provider to register a controller constructor in a config block. Then later we ask for an instance of that controller by name. Just like before, we expect to get something that's an instance of the controller constructor:

test/controller_spec.js


```
it('allows registering controllers at config time', function() {
  function MyController() {
  }
  var injector = createInjector(['ng', function($controllerProvider) {
    $controllerProvider.register('MyController', MyController);
  }]);
  var $controller = injector.get('$controller');

  var controller = $controller('MyController');
  expect(controller).toBeDefined();
  expect(controller instanceof MyController).toBe(true);
});
```

The `$controller` provider remembers the registered constructors in an internal object, whose keys are controller names and values are the constructor functions. You can add one using the `register` method of the provider:

src/controller.js

```
function $ControllerProvider() {

  var controllers = {};

  this.register = function(name, controller) {
    controllers[name] = controller;
  };

  this.$get = ['$injector', function($injector) {

    return function(ctrl, locals) {
      return $injector.instantiate(ctrl, locals);
    };

  }];

}
```

The actual `$controller` function can now check whether it should instantiate a constructor directly or look up a previously registered one, by checking the type of the first argument:

src/controller.js

```
this.$get = ['$injector', function($injector) {

  return function(ctrl, locals) {
    if (_.isString(ctrl)) {
      ctrl = controllers[ctrl];
    }
    return $injector.instantiate(ctrl, locals);
  };

}];
```


Much like you can with directives, you can also register several controllers with one call to `$controllerProvider.register`, if you give it an object where the keys are controller names and the values are their constructor functions:

test/controller_spec.js

```
it('allows registering several controllers in an object', function() {
  function MyController() { }
  function MyOtherController() { }
  var injector = createInjector(['ng', function($controllerProvider) {
    $controllerProvider.register({
      MyController: MyController,
      MyOtherController: MyOtherController
    });
  }]);
  var $controller = injector.get('$controller');

  var controller = $controller('MyController');
  var otherController = $controller('MyOtherController');

  expect(controller instanceof MyController).toBe(true);
  expect(otherController instanceof MyOtherController).toBe(true);
});
```

If `register` is given an object, it can simply extend the internal `controllers` object with that, since both objects have the same structure:

src/controller.js

```
this.register = function(name, controller) {
  if (_.isObject(name)) {
    _.extend(controllers, name);
  } else {
    controllers[name] = controller;
  }
};
```

As an Angular application developer, the `register` function of `$controllerProvider` may not actually be that familiar to you. This is because the more common approach to register controller constructors is by doing it on modules. Modules have a `controller` method, using which you can register a controller on the module:

test/controller_spec.js


```
it('allows registering controllers through modules', function() {
  var module = angular.module('myModule', []);
  module.controller('MyController', function MyController() { });

  var injector = createInjector(['ng', 'myModule']);
  var $controller = injector.get('$controller');
  var controller = $controller('MyController');

  expect(controller).toBeDefined();
});
```

What we have on module objects is simply a queued-up invocation of the `$controllerProvider.register` method we have just created. When you call `module.controller`, `$controllerProvider.register` will get called for you:

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  factory: invokeLater('$provide', 'factory'),
  value: invokeLater('$provide', 'value'),
  service: invokeLater('$provide', 'service'),
  decorator: invokeLater('$provide', 'decorator'),
  filter: invokeLater('$filterProvider', 'register'),
  directive: invokeLater('$compileProvider', 'directive'),
  controller: invokeLater('$controllerProvider', 'register'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks,
  _runBlocks: []
};
```

Global Controller Lookup

The method that we just saw for registering controllers is the preferred way to do it in Angular applications. But there is another approach supported by `$controller`, which is to look the constructor up from the global `window` object. However, this is *not* enabled by default, and such a lookup will normally raise an exception:

test/controller_spec.js


```
it('does not normally look controllers up from window', function() {
  window.MyController = function MyController() { };
  var injector = createInjector(['ng']);
  var $controller = injector.get('$controller');

  expect(function() {
    $controller('MyController');
  }).toThrow();
});
```

If, on the other hand, you call a special function called `allowGlobals` on `$controllerProvider` at config time, suddenly `$controller` *will* find the constructor from `window` and use it:

test/controller_spec.js

```
it('looks up controllers from window when so configured', function() {
  window.MyController = function MyController() { };
  var injector = createInjector(['ng', function($controllerProvider) {
    $controllerProvider.allowGlobals();
  }]);

  var $controller = injector.get('$controller');
  var controller = $controller('MyController');
  expect(controller).toBeDefined();
  expect(controller instanceof window.MyController).toBe(true);
});
```

Using this configuration option is not recommended practice, as it relies on global state, which does not bode well for modularity. It should only really be used in the simplest of example applications, and in my opinion it is of dubious value even in those. But it is there nevertheless, and here's how it works: The `allowGlobals` function sets an internal `globals` flag within the provider to `true`. When looking up controllers, we attempt to do a lookup on `window` if the normal lookup fails - but only if the `globals` flag has been set:

src/controller.js

```
function $ControllerProvider() {

  var controllers = {};
  var globals = false;

  this.allowGlobals = function() {
    globals = true;
  };

  this.register = function(name, controller) {
    if (_.isObject(name)) {
      _.extend(controllers, name);
```



```
    } else {
      controllers[name] = controller;
    }
  };

  this.$get = ['$injector', function($injector) {

    return function(ctrl, locals) {
      if (_.isString(ctrl)) {
        if (controllers.hasOwnProperty(ctrl)) {
          ctrl = controllers[ctrl];
        } else if (globals) {
          ctrl = window[ctrl];
        }
      }
      return $injector.instantiate(ctrl, locals);
    };

  }];

}
```

Directive Controllers

Now that we have a `$controller` service that knows how to register, look up, and instantiate controllers, we can start looking at situations where controllers are actually used. This is where directives come in.

You can attach a controller to a directive by specifying a `controller` key in the directive's definition object, and providing a controller constructor function as the value. That controller constructor will get instantiated when the directive is linked.

Let's add a test for that, as we ll as a new `describe` block in `compile_spec.js`, into which we can place all the controller-related tests in this chapter:

test/compile_spec.js

```
describe('controllers', function() {

  it('can be attached to directives as functions', function() {
    var controllerInvoked;
    var injector = makeInjectorWithDirectives('myDirective', function() {
      return {
        controller: function MyController() {
          controllerInvoked = true;
        }
      };
    });
    injector.invoke(function($compile, $rootScope) {
```

```
      var el = $('<div my-directive></div>');
      $compile(el)($rootScope);
      expect(controllerInvoked).toBe(true);
    });
  });

});
```

The `controller` key can also point to a string that references the name of a previously registered controller constructor:

test/compile_spec.js

```
it('can be attached to directives as string references', function() {
  var controllerInvoked;
  function MyController() {
    controllerInvoked = true;
  }
  var injector = createInjector(['ng',
      function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {controller: 'MyController'};
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(controllerInvoked).toBe(true);
  });
});
```

Controllers are instantiated for each directive individually, and there are no restrictions about having several directives with different controllers on the same element. Here we have an element with two directives applied, both of which have their own controller:

test/compile_spec.js

```
it('can be applied in the same element independent of each other', function() {
  var controllerInvoked;
  var otherControllerInvoked;
  function MyController() {
    controllerInvoked = true;
  }
  function MyOtherController() {
    otherControllerInvoked = true;
  }
  var injector = createInjector(['ng',
```

```
      function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $controllerProvider.register('MyOtherController', MyOtherController);
    $compileProvider.directive('myDirective', function() {
      return {controller: 'MyController'};
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {controller: 'MyOtherController'};
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    $compile(el)($rootScope);
    expect(controllerInvoked).toBe(true);
    expect(otherControllerInvoked).toBe(true);
  });
});
```

There are also no restrictions about using the *same* controller constructor several times. Each directive application gets its own instance of the controller, even if the same constructor is used twice on the same element:

test/compile_spec.js

```
it('can be applied to different directives, as different instances', function() {
  var invocations = 0;
  function MyController() {
    invocations++;
  }
  var injector = createInjector(['ng',
      function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {controller: 'MyController'};
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {controller: 'MyController'};
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    $compile(el)($rootScope);
    expect(invocations).toBe(2);
  });
});
```

Let's get this series of unit tests to green. First, as we iterate over directives during compilation in `applyDirectivesToNode`, we should collect all the directives that have controllers:

src/compile.js


```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = [], postLinkFns = [];
  var newScopeDirective, newIsolateScopeDirective;
  var controllerDirectives;

  function addLinkFns(preLinkFn, postLinkFn, attrStart, attrEnd, isolateScope) {
    // ...
  }

  _.forEach(directives, function(directive) {

    // ...

    if (directive.controller) {
      controllerDirectives = controllerDirectives || {};
      controllerDirectives[directive.name] = directive;
    }
  });

  // ...
}
```

Here we are building a `controllerDirectives` object where the keys are directive names and the values are the corresponding directive objects.

With the help of this object, in the linking phase we now have knowledge about all the controllers that should be instantiated when a node is linked. We can accomplish this using our new `$controller` service. It should be able to handle any value given for a directive controller, whether it's a constructor function or the name of one:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  var $element = $(linkNode);

  if (controllerDirectives) {
    _.forEach(controllerDirectives, function(directive) {
      $controller(directive.controller);
    });
  }

  // ...

}
```


Before this code will work we need to inject the `$controller` service into `$compileProvider.$get`:

src/compile.js

```
this.$get = ['$injector', '$parse', '$controller', '$rootScope',
  function($injector, $parse, $controller, $rootScope) {
```

This gets all of our test cases to pass. Directive controllers are instantiated using `$controller`, and doing that is just a matter of wiring `$compile` and `$controller` together.

An interesting additional feature of directive-controller integration is that when you have an attribute directive and specify the controller name as the string `'@'`, the controller is looked up *using the value of the directive attribute in the DOM*. This can be useful when you want to specify the directive controller not when the directive is registered, but when the directive is used. Effectively, this allows you to plug in different controllers for the same directive.

test/compile_spec.js

```
it('can be aliased with @ when given in directive attribute', function() {
  var controllerInvoked;
  function MyController() {
    controllerInvoked = true;
  }
  var injector = createInjector(['ng',
      function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {controller: '@'};
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive="MyController"></div>');
    $compile(el)($rootScope);
    expect(controllerInvoked).toBe(true);
  });
});
```

The support for this is built into the node link function, where the controller name is substituted with the value of the DOM attribute if it's defined as `'@'`:

src/compile.js

```
_.forEach(controllerDirectives, function(directive) {
  var controllerName = directive.controller;
  if (controllerName === '@') {
    controllerName = attrs[directive.name];
  }
  $controller(controllerName);
});
```


Locals in Directive Controllers

While we now know how to *instantiate* a controller for a directive, the connection between the directive and the controller is next to non-existent: The controller just happens to be instantiated with the directive but it doesn't really have access to any information about that directive, which greatly diminishes its value.

We can strengthen this connection by making a few things available to the controller:

- `$scope` - The directive's scope object
- `$element` - The element the directive is being applied to
- `$attrs` - The Attributes object of the element the directive is being applied to

These are all available to the directive constructor:

test/compile_spec.js

```
it('gets scope, element, and attrs through DI', function() {
  var gotScope, gotElement, gotAttrs;
  function MyController($element, $scope, $attrs) {
    gotElement = $element;
    gotScope = $scope;
    gotAttrs = $attrs;
  }
  var injector = createInjector(['ng',
      function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {controller: 'MyController'};
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive an-attr="abc"></div>');
    $compile(el)($rootScope);
    expect(gotElement[0]).toBe(el[0]);
    expect(gotScope).toBe($rootScope);
    expect(gotAttrs).toBeDefined();
    expect(gotAttrs.anAttr).toEqual('abc');
  });
});
```

> We purposefully put `$element` before `$scope` in the constructor arguments to highlight the fact that these are dependency-injected arguments and the order does not matter. In link functions the order is always `scope, element, attrs`, because link functions do not use dependency injection.

We can use the `locals` support we added to `$controller` earlier to pass these objects into the controller constructor. We just need to make a suitable locals object in the controller loop and pass it into `$controller`. This makes `$scope`, `$element`, and `$attrs` available for injection:

src/compile.js


```
_.forEach(controllerDirectives, function(directive) {
  var locals = {
    $scope: scope,
    $element: $element,
    $attrs: attrs
  };
  var controllerName = directive.controller;
  if (controllerName === '@') {
    controllerName = attrs[directive.name];
  }
  $controller(controllerName, locals);
});
```

And now the controller is much more connected to the directive. In fact, given those three objects, you can do anything in the directive controller that you can do in the directive's link functions. Many people actually choose to organize their directive code so that the link function doesn't do much at all, and everything is in the controller instead. This has the benefit that the controller is a separate component that can be unit tested without having to instantiate the directive - something you can't really do with link functions.

Attaching Directive Controllers on The Scope

We know how to pass the scope object to the controller. You can also do the inverse of that, which is to attach the controller object onto the scope. This enables the application pattern of publishing controller data and functions on `this` instead of `$scope` while still making them available to interpolation expressions in the DOM, as well as child directives and controllers.

> This application pattern has been described well in an article by Todd Motto. We defer further discussion about the pattern to that article and focus here on the infrastructure that enables it.

When a `controllerAs` key is defined on the directive definition object, it specifies the key by which the controller object will be attached to the scope:

test/compile_spec.js

```
it('can be attached on the scope', function() {
  function MyController() { }
  var injector = createInjector(['ng',
      function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {
        controller: 'MyController',
        controllerAs: 'myCtrl'
```

```
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect($rootScope.myCtrl).toBeDefined();
    expect($rootScope.myCtrl instanceof MyController).toBe(true);
  });
});
```

In this case the directive doesn't request an inherited or isolated scope, so the scope that gets the controller is `$rootScope`, making things a bit easier to test.

To support this use case, the `$controller` function takes an additional, optional argument which defines the "identifier" of the controller on the scope. In the node link function we can use the value of `controllerAs` for this as we call `$controller`:

src/compile.js

```
_.forEach(controllerDirectives, function(directive) {
  var locals = {
    $scope: scope,
    $element: $element,
    $attrs: attrs
  };
  var controllerName = directive.controller;
  if (controllerName === '@') {
    controllerName = attrs[directive.name];
  }
  $controller(controllerName, locals, directive.controllerAs);
});
```

> This argument is not really meant to be used by application developers directly. It is there just to support the `controllerAs` feature of the directive compiler.
> Also, this will actually be the *fourth* argument to `$controller`, not the third. The third argument is reserved for another optional argument, which we'll introduce later in this chapter.

If this optional argument is given to `$controller`, it invokes an internal helper function that will attach the controller instance to the scope:

src/controller.js

```
return function(ctrl, locals, identifier) {
  if (_.isString(ctrl)) {
    if (controllers.hasOwnProperty(ctrl)) {
```

```
      ctrl = controllers[ctrl];
    } else if (globals) {
      ctrl = window[ctrl];
    }
  }
  var instance = $injector.instantiate(ctrl, locals);
  if (identifier) {
    addToScope(locals, identifier, instance);
  }
  return instance;
};
```

The `addToScope` function finds the scope - which will be on the given `locals` object - and puts the controller instance on it using the identifier. If an identifier has been given but there's no `$scope` in the locals object, an exception is thrown.

This function can be defined on the top level of `controller.js`:

src/controller.js

```
function addToScope(locals, identifier, instance) {
  if (locals && _.isObject(locals.$scope)) {
    locals.$scope[identifier] = instance;
  } else {
    throw 'Cannot export controller as ' + identifier +
      '! No $scope object provided via locals';
  }
}
```

Controllers on Isolate Scope Directives

At first blush, using controllers with isolate scope directives doesn't look too different from using them in non-isolated contexts. However, some of the features related to isolate scopes do provide us with obstacles that require special attention.

Before we go there, let's cover some of the basics: When a directive has an isolate scope, the `$scope` argument injected to the controller should be the isolate scope and not the surrounding scope.

test/compile_spec.js

```
it('gets isolate scope as injected $scope', function() {
  var gotScope;
  function MyController($scope) {
    gotScope = $scope;
  }
  var injector = createInjector(['ng',
      function($controllerProvider, $compileProvider) {
```

```
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: 'MyController'
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(gotScope).not.toBe($rootScope);
  });
});
```

To support this behavior, we'll need to shift our code in the node link function around a bit. Any isolate scope should be created *before* instantiating controllers. We'll still do the rest of the isolate scope setup *after* controller instantiation, for reasons that will become apparent in a moment.

Once we have the isolate scope object, we pass it into the directive controller. We should be careful to use the isolate scope only for the directive that actually requested it. All other controllers active on the node still receive the non-isolated scope:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {
  var $element = $(linkNode);

  var isolateScope;
  if (newIsolateScopeDirective) {
    isolateScope = scope.$new(true);
    $element.addClass('ng-isolate-scope');
    $element.data('$isolateScope', isolateScope);
  }

  if (controllerDirectives) {
    _.forEach(controllerDirectives, function(directive) {
      var locals = {
        $scope: directive === newIsolateScopeDirective ? isolateScope : scope,
        $element: $element,
        $attrs: attrs
      };
      var controllerName = directive.controller;
      if (controllerName === '@') {
        controllerName = attrs[directive.name];
      }
      $controller(controllerName, locals, directive.controllerAs);
    });
  }
```



```
  if (newIsolateScopeDirective) {
    _.forEach(newIsolateScopeDirective.$$isolateBindings,
        function(definition, scopeName) {
      // ...
    }};
  }

  // ...

}
```

We could have just moved all of the isolate scope setup code above the controller instantiation code and our test case would still have passed. So why did we split it in two like we did?

The reason is a feature related to isolate scopes that we'll implement next, called `bindToController`. This is a flag that can be set in the directive definition object, which controls *where all the isolate scope bindings will be attached*. In the previous chapter we saw that all the bindings introduced with `@`, `=`, or `&` are attached on the isolate scope. However, when the `bindToController` flag is set on a directive, those bindings should be placed on the *controller object* instead of the isolate scope. This is particularly useful in conjunction with the `controllerAs` option, which then makes the controller with all those isolate bindings available to child elements.

The catch is that now we have a chicken-and-egg problem when it comes to setting things up:

1. *Isolate scope bindings should exist before the controller constructor is called*, because the constructor may expect such bindings to already exist when it runs.
2. If `bindToController` is used, the isolate scope bindings must be attached on the `controller object`, which means *we must have the controller object before we can set up the isolate bindings.*

This means that we need to have the controller object before we actually call the controller constructor. Given the flexibility of the JavaScript language, it is actually possible for us to do that, but we'll have to spend some effort to get there.

Let's first add a couple of unit tests that illustrate where we are going with this. When a controller constructor is invoked, all isolate scope bindings must already be on the scope:

test/compile_spec.js

```
it('has isolate scope bindings available during construction', function() {
  var gotMyAttr;
  function MyController($scope) {
    gotMyAttr = $scope.myAttr;
  }
  var injector = createInjector(['ng',
      function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
```



```
      return {
        scope: {
          myAttr: '@myDirective'
        },
        controller: 'MyController'
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive="abc"></div>');
    $compile(el)($rootScope);
    expect(gotMyAttr).toEqual('abc');
  });
});
```

On the other hand, if `bindToController` is enabled, the isolate scope bindings will be on the controller instance, not on `$scope`. *When the controller constructor is called, there will already be attributes on `this`*:

test/compile_spec.js

```
it('can bind isolate scope bindings directly to self', function() {
  var gotMyAttr;
  function MyController() {
    gotMyAttr = this.myAttr;
  }
  var injector = createInjector(['ng',
      function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {
          myAttr: '@myDirective'
        },
        controller: 'MyController',
        bindToController: true
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive="abc"></div>');
    $compile(el)($rootScope);
    expect(gotMyAttr).toEqual('abc');
  });
});
```

Both of these test cases are initially failing, and will continue to do so until we're done. So let's go ahead and dive into the details.

The `$controller` function takes an optional third argument, called `later`, that causes the function to return a "semi-constructed" controller instead of a fully constructed one.

The meaning of "semi-constructed" is that the controller object already exists, but the controller constructor has not yet been invoked. In concrete terms, in this case the return value of `$controller` has the following characteristics:

- It is a function which, when called, will invoke the controller constructor
- It has an attribute called `instance` that points to the controller object.

This kind of deferred construction gives the caller of `$controller` a chance to do work between creating the controller object and calling its constructor - just what we need to set up the isolate scope bindings.

test/controller_spec.js

```
it('can return a semi-constructed controller', function() {
  var injector = createInjector(['ng']);
  var $controller = injector.get('$controller');

  function MyController() {
    this.constructed = true;
    this.myAttrWhenConstructed = this.myAttr;
  }

  var controller = $controller(MyController, null, true);

  expect(controller.constructed).toBeUndefined();
  expect(controller.instance).toBeDefined();

  controller.instance.myAttr = 42;
  var actualController = controller();

  expect(actualController.constructed).toBeDefined();
  expect(actualController.myAttrWhenConstructed).toBe(42);
});
```

As we introduce the `later` argument to `$controller`, the `identifier` argument we introduced earlier is pushed to be the fourth argument:

src/controller.js

```
this.$get = ['$injector', function($injector) {

  return function(ctrl, locals, later, identifier) {
    // ...
  };

}];
```


We want to keep our existing test suite passing, so let's temporarily supply `false` as the value for `later` in `compile.js`. We'll come back and change this in a few moments:

src/compile.js

```
_.forEach(controllerDirectives, function(directive) {
  var locals = {
    $scope: directive === newIsolateScopeDirective ? isolateScope : scope,
    $element: $element,
    $attrs: attrs
  };
  var controllerName = directive.controller;
  if (controllerName === '@') {
    controllerName = attrs[directive.name];
  }
  $controller(controllerName, locals, false, directive.controllerAs);
});
```

> Both `later` and `identifier` are designed to be used only by the framework internally, and not directly by application developers. If you do use them, beware that they may change or go away in the future since they are considered to be implementation details.

Inside `$controller` we can now decide based on this flag whether we should do normal instantiation or something else:

src/controller.js

```
return function(ctrl, locals, later, identifier) {
  if (_.isString(ctrl)) {
    if (controllers.hasOwnProperty(ctrl)) {
      ctrl = controllers[ctrl];
    } else if (globals) {
      ctrl = window[ctrl];
    }
  }
  var instance;
  if (later) {

  } else {
    instance = $injector.instantiate(ctrl, locals);
    if (identifier) {
      addToScope(locals, identifier, instance);
    }
    return instance;
  }
};
```


What we do has two steps to it:

1. Create a new object whose prototype is based on the constructor function. `Object.create` comes in handy here.
2. Return the "semi-constructed" controller: A function that can be used to actually call the constructor later, and that has the object instance available in the `instance` attribute.

When we do finally call the constructor, we should not do it with `$injector.instantiate` because we're not actually instantiating anything at that point. We can call it as a regular dependency-injected function with `$injector.invoke`. We just need to pass the constructor object as the `self` argument so that `this` is bound correctly.

src/controller.js

```
return function(ctrl, locals, later, identifier) {
  if (_.isString(ctrl)) {
    if (controllers.hasOwnProperty(ctrl)) {
      ctrl = controllers[ctrl];
    } else if (globals) {
      ctrl = window[ctrl];
    }
  }
  var instance;
  if (later) {
    instance = Object.create(ctrl);
    return _.extend(function() {
      $injector.invoke(ctrl, instance, locals);
      return instance;
    }, {
      instance: instance
    });
  } else {
    instance = $injector.instantiate(ctrl, locals);
    if (identifier) {
      addToScope(locals, identifier, instance);
    }
    return instance;
  }
};
```

Since `$controller` supports dependency injection, the first argument given to it may actually be an array-style dependency injection wrapper instead of a plain function. It should still support the `later` flag:

test/controller_spec.js


```
it('can return a semi-constructed ctrl when using array injection', function() {
  var injector = createInjector(['ng', function($provide) {
    $provide.constant('aDep', 42);
  }]);
  var $controller = injector.get('$controller');

  function MyController(aDep) {
    this.aDep = aDep;
    this.constructed = true;
  }

  var controller = $controller(['aDep', MyController], null, true);
  expect(controller.constructed).toBeUndefined();
  var actualController = controller();
  expect(actualController.constructed).toBeDefined();
  expect(actualController.aDep).toBe(42);
});
```

As we pass in the prototype to `Object.create`, we need to unwrap the dependency injection wrapper array if there is one:

src/controller.js

```
var ctrlConstructor = _.isArray(ctrl) ? _.last(ctrl) : ctrl;
instance = Object.create(ctrlConstructor.prototype);
return _.extend(function() {
  $injector.invoke(ctrl, instance, locals);
  return instance;
}, {
  instance: instance
});
```

Now, combining the third `later` argument with the fourth `identifier` argument, we can also expect `$controller` to bind the semi-constructed controller object on the scope if we ask it to:

test/controller_spec.js

```
it('can bind semi-constructed controller to scope', function() {
  var injector = createInjector(['ng']);
  var $controller = injector.get('$controller');

  function MyController() {
  }
  var scope = {};

  var controller = $controller(MyController, {$scope: scope}, true, 'myCtrl');
  expect(scope.myCtrl).toBe(controller.instance);
});
```


> We're not using an actual Scope object here - just a plain object. For the purposes of the test, there is no difference.

This is accomplished by calling the same `addToScope` helper function from this branch that we already call in the "eager construction" branch:

src/controller.js

```
if (later) {
  var ctrlConstructor = _.isArray(ctrl) ? _.last(ctrl) : ctrl;
  instance = Object.create(ctrlConstructor.prototype);
  if (identifier) {
    addToScope(locals, identifier, instance);
  }
  return _.extend(function() {
    $injector.invoke(ctrl, instance, locals);
    return instance;
  }, {
    instance: instance
  });
} else {
  instance = $injector.instantiate(ctrl, locals);
  if (identifier) {
    addToScope(locals, identifier, instance);
  }
  return instance;
}
```

Now that we have the infrastructure we need in `$controller`, we can tie things together by making a few changes in `$compile`.

First, let's introduce a variable in which we can store the semi-constructed controller functions. This goes into the top level of the `applyDirectivesToNode` function:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var preLinkFns = [], postLinkFns = [], controllers = {};
  // ...
}
```

As we construct the controllers, we'll store the semi-constructed functions in this object. Note that we now pass in `true` as the third argument to `$controller` to trigger the `later` flag:

src/compile.js


```
_.forEach(controllerDirectives, function(directive) {
  var locals = {
    $scope: directive === newIsolateScopeDirective ? isolateScope : scope,
    $element: $element,
    $attrs: attrs
  };
  var controllerName = directive.controller;
  if (controllerName === '@') {
    controllerName = attrs[directive.name];
  }
  controllers[directive.name] =
    $controller(controllerName, locals, true, directive.controllerAs);
});
```

Then, *after* we have set up the isolate scope bindings, but *before* we call the prelink functions, we invoke the semi-constructed controller functions, which triggers the invocation of the actual controller constructors.

src/compile.js

```
if (newIsolateScopeDirective) {
  _.forEach(newIsolateScopeDirective.$$isolateBindings,
      function(definition, scopeName) {
    // ...
  });
}

_.forEach(controllers, function(controller) {
  controller();
});

_.forEach(preLinkFns, function(linkFn) {
  linkFn(linkFn.isolateScope ? isolateScope : scope, $element, attrs);
});
```

Effectively, we now set up the isolate scope binding while the controllers are in the semi-constructed state.

This has the first of the two high-level test cases passing, but the one for `bindToController` is still failing. What we need to do is extend the isolate binding initialization so that it can deal with two different situations: Regular isolate bindings and bindings to the *controller* instead of the scope (when `bindToController` is `true`).

To make this easier, let's change the parsing code for the bindings slightly. What we are currently doing is setting an `$$isolateBindings` attribute on the directive when we initialize it. We should generalize this into a `$$bindings` attribute with which we can handle both scope and controller bindings. To initialize it, we'll use a new helper function called `parseDirectiveBindings`:

src/compile.js


```
return _.map(factories, function(factory, i) {
  var directive = $injector.invoke(factory);
  directive.restrict = directive.restrict || 'EA';
  directive.priority = directive.priority || 0;
  if (directive.link && !directive.compile) {
    directive.compile = _.constant(directive.link);
  }
  directive.$$bindings = parseDirectiveBindings(directive);
  directive.name = directive.name || name;
  directive.index = i;
  return directive;
});
```

A first version of `parseDirectiveBindings` can merely call the existing `parseIsolateBindings` function, and set the return value into the `isolateScope` attribute of our new bindings object:

src/compile.js

```
function parseDirectiveBindings(directive) {
  var bindings = {};
  if (_.isObject(directive.scope)) {
    bindings.isolateScope = parseIsolateBindings(directive.scope);
  }
  return bindings;
}
```

Moving then to the *initialization* of these bindings that happens during linking, we should do a bit of refactoring here too. Let's extract the binding initialization code to a separate function, away from `nodeLinkFn`. We can call it `initializeDirectiveBindings`. It contains the initialization loop code we've written earlier and takes the arguments it needs to do its work:

src/compile.js

```
function initializeDirectiveBindings(scope, attrs, bindings, isolateScope) {
  _.forEach(bindings, function(definition, scopeName) {
    var attrName = definition.attrName;
    switch (definition.mode) {
      case '@':
        attrs.$observe(attrName, function(newAttrValue) {
          isolateScope[scopeName] = newAttrValue;
        });
        if (attrs[attrName]) {
          isolateScope[scopeName] = attrs[attrName];
        }
        break;
```



```
      case '=':
        if (definition.optional && !attrs[attrName]) {
          break;
        }
        var parentGet = $parse(attrs[attrName]);
        var lastValue = isolateScope[scopeName] = parentGet(scope);
        var parentValueWatch = function() {
          var parentValue = parentGet(scope);
          if (isolateScope[scopeName] !== parentValue) {
            if (parentValue !== lastValue) {
              isolateScope[scopeName] = parentValue;
            } else {
              parentValue = isolateScope[scopeName];
              parentGet.assign(scope, parentValue);
            }
          }
          lastValue = parentValue;
          return lastValue;
        };
        var unwatch;
        if (definition.collection) {
          unwatch = scope.$watchCollection(attrs[attrName], parentValueWatch);
        } else {
          unwatch = scope.$watch(parentValueWatch);
        }
        isolateScope.$on('$destroy', unwatch);
        break;
      case '&':
        var parentExpr = $parse(attrs[attrName]);
        if (parentExpr === _.noop && definition.optional) {
          break;
        }
        isolateScope[scopeName] = function(locals) {
          return parentExpr(scope, locals);
        };
        break;
    }
  });
}
```

In `nodeLinkFn` itself, all that is now left to do is to call this function instead of having the big loop. As the bindings, we give it the isolate scope bindings we created in the new `parseDirectiveBindings` function:

src/compile.js

```
if (newIsolateScopeDirective) {
  initializeDirectiveBindings(
    scope,
```



```
    attrs,
    newIsolateScopeDirective.$$bindings.isolateScope,
    isolateScope
  );
}
```

Now we're ready to extend this to work with `bindToController` as well. In `parseDirectiveBindings`, if this flag is `true`, we'll put the bindings in the `bindToController` key and not the `isolateScope` key:

src/compile.js

```
function parseDirectiveBindings(directive) {
  var bindings = {};
  if (_.isObject(directive.scope)) {
    if (directive.bindToController) {
      bindings.bindToController = parseIsolateBindings(directive.scope);
    } else {
      bindings.isolateScope = parseIsolateBindings(directive.scope);
    }
  }
  return bindings;
}
```

These new bindings are then initialized just before the controllers are instantiated in the node link function. We only do so if we have an isolate scope directive that also has a controller. We can reuse the `initializeDirectiveBindings` function we extracted earlier:

src/compile.js

```
if (newIsolateScopeDirective && controllers[newIsolateScopeDirective.name]) {
  initializeDirectiveBindings(
    scope,
    attrs,
    newIsolateScopeDirective.$$bindings.bindToController,
    isolateScope
  );
}

_.forEach(controllers, function(controller) {
  controller();
});
```

The remaining issue with this is that these bindings are still being attached to the isolate scope object, when the whole point is that they should be attached to the controller! The `initializeDirectiveBindings` function should accept one more argument, which we'll call `destination`, that points to the object on which all the data should be bound. It is used as the target across the different binding types:

src/compile.js


```
function initializeDirectiveBindings(
    scope, attrs, destination, bindings, newScope) {
  _.forEach(bindings, function(definition, scopeName) {
    var attrName = definition.attrName;
    switch (definition.mode) {
      case '@':
        attrs.$observe(attrName, function(newAttrValue) {
          destination[scopeName] = newAttrValue;
        });
        if (attrs[attrName]) {
          destination[scopeName] = attrs[attrName];
        }
        break;
      case '=':
        if (definition.optional && !attrs[attrName]) {
          break;
        }
        var parentGet = $parse(attrs[attrName]);
        var lastValue = destination[scopeName] = parentGet(scope);
        var parentValueWatch = function() {
          var parentValue = parentGet(scope);
          if (destination[scopeName] !== parentValue) {
            if (parentValue !== lastValue) {
              destination[scopeName] = parentValue;
            } else {
              parentValue = destination[scopeName];
              parentGet.assign(scope, parentValue);
            }
          }
          lastValue = parentValue;
          return lastValue;
        };
        var unwatch;
        if (definition.collection) {
          unwatch = scope.$watchCollection(attrs[attrName], parentValueWatch);
        } else {
          unwatch = scope.$watch(parentValueWatch);
        }
        newScope.$on('$destroy', unwatch);
        break;
      case '&':
        var parentExpr = $parse(attrs[attrName]);
        if (parentExpr === _.noop && definition.optional) {
          break;
        }
        destination[scopeName] = function(locals) {
          return parentExpr(scope, locals);
        };
        break;
    }
```



```
  });
}
```

In the case of regular isolate bindings, the destination is the isolate scope itself:

src/compile.js

```
if (newIsolateScopeDirective) {
  initializeDirectiveBindings(
    scope,
    attrs,
    isolateScope,
    newIsolateScopeDirective.$$bindings.isolateScope,
    isolateScope
  );
}
```

And in the case of `bindToController`, the destination is the controller instance object:

src/compile.js

```
if (newIsolateScopeDirective && controllers[newIsolateScopeDirective.name]) {
  initializeDirectiveBindings(
    scope,
    attrs,
    controllers[newIsolateScopeDirective.name].instance,
    newIsolateScopeDirective.$$bindings.bindToController,
    isolateScope
  );
}
```

And finally all the tests are passing!

This implementation also unlocks a couple of convenient patterns we can easily enable for application developers. Firstly, people might prefer specifying the directive bindings as the value of `bindToController` instead of `isolateScope`, since the controller will be where they end up. It makes the API just a bit more convenient:

test/compile_spec.js

```
it('can bind iso scope bindings through bindToController', function() {
  var gotMyAttr;
  function MyController() {
    gotMyAttr = this.myAttr;
  }
  var injector = createInjector(['ng',
                  function($controllerProvider, $compileProvider) {
```

```
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: 'MyController',
        bindToController: {
          myAttr: '@myDirective'
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive="abc"></div>');
    $compile(el)($rootScope);
    expect(gotMyAttr).toEqual('abc');
  });
});
```

We can easily handle this in the `parseDirectiveBindings` function. If the value of `bindToController` is an object, it is parsed just like the value of `scope` is parsed:

src/compile.js

```
function parseDirectiveBindings(directive) {
  var bindings = {};
  if (_.isObject(directive.scope)) {
    if (directive.bindToController) {
      bindings.bindToController = parseIsolateBindings(directive.scope);
    } else {
      bindings.isolateScope = parseIsolateBindings(directive.scope);
    }
  }
  if (_.isObject(directive.bindToController)) {
    bindings.bindToController =
      parseIsolateBindings(directive.bindToController);
  }
  return bindings;
}
```

Secondly, we can extend this implementation so that you don't actually need an isolate scope at all in order to make bindings! You can just use a combination of a regular inherited scope and an object value for `bindToController`:

test/compile_spec.js

```
it('can bind through bindToController without iso scope', function() {
  var gotMyAttr;
  function MyController() {
    gotMyAttr = this.myAttr;
  }
  var injector = createInjector(['ng',
                    function($controllerProvider, $compileProvider) {
    $controllerProvider.register('MyController', MyController);
    $compileProvider.directive('myDirective', function() {
      return {
        scope: true,
        controller: 'MyController',
        bindToController: {
          myAttr: '@myDirective'
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive="abc"></div>');
    $compile(el)($rootScope);
    expect(gotMyAttr).toEqual('abc');
  });
});
```

Here we can extend the code we have in the node link function, so that it supports making the controller binding initialization for either the isolate scope directive or the new scope directive, whichever the current node has:

src/compile.js

```
var scopeDirective = newIsolateScopeDirective || newScopeDirective;
if (scopeDirective && controllers[scopeDirective.name]) {
  initializeDirectiveBindings(
    scope,
    attrs,
    controllers[scopeDirective.name].instance,
    scopeDirective.$$bindings.bindToController,
    isolateScope
  );
}
```

And now our controller and isolate scope implementations are fully compatible, and actually support each other in interesting ways.

It's quite a bit of infrastructure for a seemingly simple feature, but the application pattern enabled by this is quite useful: The combination of `controllerAs` and `bindToController`

lets application developers write a lot of code without ever explicitly using the `$scope` object, which is preferred by many people.

> See this article for some further discussion on this pattern.

Requiring Controllers

Controllers are a handy alternative to link functions for carrying the logic of your directives, but that's not all controllers are useful for. Controllers can also be used to provide yet another channel for communication between different directives. This is perhaps the most powerful of the "cross-directive communication" facilities in Angular: Requiring controllers from other directives.

A directive can "require" some other directive by name by specifying the `require` key in the directive definition object. When this is done, and the required directive is indeed present on the same element, the required directive's controller is given to the requiring directive's link function as the fourth argument.

Effectively, one directive can gain access to the controller of another directive, and thus all the data and functions it makes available. This is the case even if one of the directives uses an isolate scope:

test/compile_spec.js

```
it('can be required from a sibling directive', function() {
  function MyController() { }
  var gotMyController;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: MyController
      };
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {
        require: 'myDirective',
        link: function(scope, element, attrs, myController) {
          gotMyController = myController;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    $compile(el)($rootScope);
    expect(gotMyController).toBeDefined();
    expect(gotMyController instanceof MyController).toBe(true);
  });
});
```


Here we have two directives, `myDirective` and `myOtherDirective`, used on the same element. `myDirective` defines both a controller and an isolate scope. `myOtherDirective` defines neither, but it *requires* `myDirective`. We check that this causes the controller of `myDirective` to be passed to `myOtherDirective`'s link function.

Let's first attach some information about a directive's `require` flag to its link function. As we invoke `addLinkFns` in the directive loop of `applyDirectivesToNode`, let's pass in the `require` attribute of the directive:

src/compile.js

```
if (directive.compile) {
  var linkFn = directive.compile($compileNode, attrs);
  var isolateScope = (directive === newIsolateScopeDirective);
  var attrStart = directive.$$start;
  var attrEnd = directive.$$end;
  var require = directive.require;
  if (_.isFunction(linkFn)) {
    addLinkFns(null, linkFn, attrStart, attrEnd, isolateScope, require);
  } else if (linkFn) {
    addLinkFns(
      linkFn.pre, linkFn.post, attrStart, attrEnd, isolateScope, require);
  }
}
```

In `addLinkFns`, let's use this argument to populate the `require` attribute of both the pre- and postlink functions:

src/compile.js

```
function addLinkFns(preLinkFn, postLinkFn, attrStart, attrEnd,
                    isolateScope, require) {
  if (preLinkFn) {
    if (attrStart) {
      preLinkFn = groupElementsLinkFnWrapper(preLinkFn, attrStart, attrEnd);
    }
    preLinkFn.isolateScope = isolateScope;
    preLinkFn.require = require;
    preLinkFns.push(preLinkFn);
  }
  if (postLinkFn) {
    if (attrStart) {
      postLinkFn = groupElementsLinkFnWrapper(postLinkFn, attrStart, attrEnd);
    }
    postLinkFn.isolateScope = isolateScope;
    postLinkFn.require = require;
    postLinkFns.push(postLinkFn);
  }
}
```


As we now invoke those pre- and postlink functions from the node link function, we can check whether they have the `require` attribute set. If they do, we'll pass in a fourth argument to the link functions. The value of this argument will be the return value of a new function called `getControllers`, which we'll define in a moment:

src/compile.js

```
_.forEach(preLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require)
  );
});
if (childLinkFn) {
  childLinkFn(scope, linkNode.childNodes);
}
_.forEachRight(postLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require)
  );
});
```

The `getControllers` function should be defined inside `applyDirectivesToNode`, where it has access to the `controllers` variable that stores the (semi-constructed) controllers of the current node:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = [], postLinkFns = [], controllers = {};
  var newScopeDirective, newIsolateScopeDirective;
  var controllerDirectives;

  function getControllers(require) {

  }

  // ...

}
```


What the function does is look up the required controller and return it. If no controller is found, it throws an exception:

src/compile.js

```
function getControllers(require) {
  var value;
  if (controllers[require]) {
    value = controllers[require].instance;
  }
  if (!value) {
    throw 'Controller '+require+' required by directive, cannot be found!';
  }
  return value;
}
```

Note that what's stored in `controllers` are the semi-constructed controller functions from the previous section, so we need to access the `instance` attribute to get to the actual controller object that the link functions will then receive. By that point it will be fully constructed.

Requiring Multiple Controllers

You can actually require not just one, but several other directive controllers to your directive. If you define an array of strings as the value of `require`, the fourth argument to your link functions will be an array of the corresponding controller objects:

test/compile_spec.js

```
it('can be required from multiple sibling directives', function() {
  function MyController() { }
  function MyOtherController() { }
  var gotControllers;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        scope: true,
        controller: MyController
      };
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {
        scope: true,
        controller: MyOtherController
      };
    });
```



```
    $compileProvider.directive('myThirdDirective', function() {
      return {
        require: ['myDirective', 'myOtherDirective'],
        link: function(scope, element, attrs, controllers) {
          gotControllers = controllers;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive my-third-directive></div>');
    $compile(el)($rootScope);
    expect(gotControllers).toBeDefined();
    expect(gotControllers.length).toBe(2);
    expect(gotControllers[0] instanceof MyController).toBe(true);
    expect(gotControllers[1] instanceof MyOtherController).toBe(true);
  });
});
```

In this case we have three directives on the same element. The first two both define controllers, and the third one requires the first two. We then check that the third directive's link function receives both controllers.

Making this work is actually quite simple. If the argument given to `getControllers` is an array, we'll return a recursive mapping of the values in that array to `getControllers` - an array of controllers:

src/compile.js

```
function getControllers(require) {
  if (_.isArray(require)) {
    return _.map(require, getControllers);
  } else {
    var value;
    if (controllers[require]) {
      value = controllers[require].instance;
    }
    if (!value) {
      throw 'Controller '+require+' required by directive, cannot be found!';
    }
    return value;
  }
}
```

Self-Requiring Directives

When a directive defines its own controller, but does not require any other directive controllers, it receives its own controller object as the fourth argument to its link functions. It's as if the directive requires itself - a convenient little feature:

test/compile_spec.js

```
it('is passed to link functions if there is no require', function() {
  function MyController() { }
  var gotMyController;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: MyController,
        link: function(scope, element, attrs, myController) {
          gotMyController = myController;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(gotMyController).toBeDefined();
    expect(gotMyController instanceof MyController).toBe(true);
  });
});
```

The directive "requiring itself" is actually exactly what happens. As the directive definition is registered, if it doesn't have a `require` attribute but does have a `controller` attribute, the `require` attribute's value is set to be the name of the directive itself. This causes the directive's own controller to be looked up, which passes our test case:

src/compile.js

```
$provide.factory(name + 'Directive', ['$injector', function($injector) {
  var factories = hasDirectives[name];
  return _.map(factories, function(factory, i) {
    var directive = $injector.invoke(factory);
    directive.restrict = directive.restrict || 'EA';
    directive.priority = directive.priority || 0;
    if (directive.link && !directive.compile) {
      directive.compile = _.constant(directive.link);
    }
    directive.$$bindings = parseDirectiveBindings(directive);
    directive.name = directive.name || name;
    directive.index = i;
    directive.require = directive.require || (directive.controller && name);
    return directive;
  });
}]);
```


Requiring Controllers in Multi-Element Directives

Requiring controllers should work just as well for directive with grouped elements:

test/compile_spec.js

```
it('is passed through grouped link wrapper', function() {
  function MyController() { }
  var gotMyController;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        multiElement: true,
        scope: {},
        controller: MyController,
        link: function(scope, element, attrs, myController) {
          gotMyController = myController;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive-start></div><div my-directive-end></div>');
    $compile(el)($rootScope);
    expect(gotMyController).toBeDefined();
    expect(gotMyController instanceof MyController).toBe(true);
  });
});
```

> In this test we're using the "self-require" facility from the last section: The fourth argument to the link function should be the directive's own controller. We do this just to simplify the test a bit, as we don't have to define additional directives.

The reason this test case does not pass immediately is that the `groupElementsLinkFnWrapper` function - which is used to wrap the link functions of multi-element directives - is not aware of the fourth argument to link functions and thus does not pass it forward. Fixing this is easy enough:

src/compile.js

```
function groupElementsLinkFnWrapper(linkFn, attrStart, attrEnd) {
  return function(scope, element, attrs, ctrl) {
    var group = groupScan(element[0], attrStart, attrEnd);
    return linkFn(scope, group, attrs, ctrl);
  };
}
```


Requiring Controllers from Parent Elements

Requiring a controller from a sibling directive is one thing, but it is quite limited: We don't currently have any way for directives that are applied on a *family* of elements to collaborate (outside of sharing things on a scope object).

The `require` flag is in fact more flexible than what we have previously seen, because it allows you to require controllers not only from the current element but also its parents. This style of lookup is enabled if you prefix the required directive name with `^`:

test/compile_spec.js

```
it('can be required from a parent directive', function() {
  function MyController() { }
  var gotMyController;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: MyController
      };
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {
        require: '^myDirective',
        link: function(scope, element, attrs, myController) {
          gotMyController = myController;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive><div my-other-directive></div></div>');
    $compile(el)($rootScope);
    expect(gotMyController).toBeDefined();
    expect(gotMyController instanceof MyController).toBe(true);
  });
});
```

Here `myOtherDirective` requires `^myDirective`, which is found from the parent element.

When the `^` prefix is used, the required directive is searched not only from parent elements, but also from the current element as well (in fact the current element is looked at *first*). The exact meaning of `^` is "current element or one of its ancestors".

test/compile_spec.js

```
it('finds from sibling directive when requiring with parent prefix', function() {
  function MyController() { }
  var gotMyController;
```



```
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: MyController
      };
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {
        require: '^myDirective',
        link: function(scope, element, attrs, myController) {
          gotMyController = myController;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    $compile(el)($rootScope);
    expect(gotMyController).toBeDefined();
    expect(gotMyController instanceof MyController).toBe(true);
  });
});
```

To get this kind of `require` system working, we can't rely on the `controllers` object we have in `applyDirectivesToNode` alone, because that only knows about controllers on the *current* element. Our controller lookup code will need to become aware of the structure of the DOM as well. The first step towards this is to pass the current element to the `getControllers` function:

src/compile.js

```
_.forEach(preLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element)
  );
});
if (childLinkFn) {
  childLinkFn(scope, linkNode.childNodes);
}
_.forEachRight(postLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element)
  );
});
```


Before `getControllers` can find what it needs, we need to attach some information to the DOM as we create controllers. Every controller that we create should also be added as jQuery data to the respective DOM node:

src/compile.js

```
_.forEach(controllerDirectives, function(directive) {
  var locals = {
    $scope: directive === newIsolateScopeDirective ? isolateScope : scope,
    $element: $element,
    $attrs: attrs
  };
  var controllerName = directive.controller;
  if (controllerName === '@') {
    controllerName = attrs[directive.name];
  }
  var controller =
    $controller(controllerName, locals, true, directive.controllerAs);
  controllers[directive.name] = controller;
  $element.data('$' + directive.name + 'Controller', controller.instance);
});
```

Now, `getControllers` should try to match the `require` argument to a regular expression that captures any preceding `^` character. If there is no such prefix we can do the lookup as we did before, but if there is, we need to do something else:

src/compile.js

```
function getControllers(require, $element) {
  if (_.isArray(require)) {
    return _.map(require, getControllers);
  } else {
    var value;
    var match = require.match(/^(\^)?/);
    require = require.substring(match[0].length);
    if (match[1]) {

    } else {
      if (controllers[require]) {
        value = controllers[require].instance;
      }
    }
    if (!value) {
      throw 'Controller '+require+' required by directive, cannot be found!';
    }
    return value;
  }
}
```


What we do when there is a ^ prefix is walk up the DOM until we find the jQuery data that matches the required directive name (or until we reach the root of the DOM tree):

src/compile.js

```
function getControllers(require, $element) {
  if (_.isArray(require)) {
    return _.map(require, getControllers);
  } else {
    var value;
    var match = require.match(/^(\^)?/);
    require = require.substring(match[0].length);
    if (match[1]) {
      while ($element.length) {
        value = $element.data('$' + require + 'Controller');
        if (value) {
          break;
        } else {
          $element = $element.parent();
        }
      }
    } else {
      if (controllers[require]) {
        value = controllers[require].instance;
      }
    }
    if (!value) {
      throw 'Controller '+require+' required by directive, cannot be found!';
    }
    return value;
  }
}
```

In addition to ^, the `require` attribute also supports a ^^ prefix. It is very similar to ^ in that it finds a directive controller from a parent element:

test/compile_spec.js

```
it('can be required from a parent directive with ^^', function() {
  function MyController() { }
  var gotMyController;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: MyController
      };
```



```
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {
        require: '^^myDirective',
        link: function(scope, element, attrs, myController) {
          gotMyController = myController;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive><div my-other-directive></div></div>');
    $compile(el)($rootScope);
    expect(gotMyController).toBeDefined();
    expect(gotMyController instanceof MyController).toBe(true);
  });
});
```

The difference is that `^^` does *not* look for the controller from sibling directives and starts its search directly for the parent element:

test/compile_spec.js

```
it('does not find from sibling directive when requiring with ^^', function() {
  function MyController() { }
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: MyController
      };
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {
        require: '^^myDirective',
        link: function(scope, element, attrs, myController) {
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    expect(function() {
      $compile(el)($rootScope);
    }).toThrow();
  });
});
```

In the test we expect the linking process to throw since we're requiring a directive that exists on a sibling directive and it will never be found because of the `^^` prefix.

The regex in `getControllers` should optionally match that second `^` in the prefix, and if one is found, start the search from the current element's parent instead of the element itself:

src/compile.js

```
function getControllers(require, $element) {
  if (_.isArray(require)) {
    return _.map(require, getControllers);
  } else {
    var value;
    var match = require.match(/^(\^\^?)?/);
    require = require.substring(match[0].length);
    if (match[1]) {
      if (match[1] === '^^') {
        $element = $element.parent();
      }
      while ($element.length) {
        value = $element.data('$' + require + 'Controller');
        if (value) {
          break;
        } else {
          $element = $element.parent();
        }
      }
    } else {
      if (controllers[require]) {
        value = controllers[require].instance;
      }
    }
    if (!value) {
      throw 'Controller '+require+' required by directive, cannot be found!';
    }
    return value;
  }
}
```

Optionally Requiring Controllers

Our `require` implementation currently always throws an exception when the required controller cannot be found. You can in fact choose not to be that strict about the requirement by prefixing the require statement with a question mark. If you do that, instead of throwing an exception, you'll just get `null` as the value of that controller:

test/compile_spec.js


```
it('does not throw on required missing controller when optional', function() {
  var gotCtrl;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        require: '?noSuchDirective',
        link: function(scope, element, attrs, ctrl) {
          gotCtrl = ctrl;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(gotCtrl).toBe(null);
  });
});
```

The regex in `getControllers` optionally matches this question mark. It only throws the exception when the question mark was *not* given. We should also adjust the `return` statement so that it returns an actual `null` instead of `undefined` when no controller was found:

src/compile.js

```
function getControllers(require, $element) {
  if (_.isArray(require)) {
    return _.map(require, getControllers);
  } else {
    var value;
    var match = require.match(/^(\^\^?)?(\?)?/);
    var optional = match[2];
    require = require.substring(match[0].length);
    if (match[1]) {
      if (match[1] === '^^') {
        $element = $element.parent();
      }
      while ($element.length) {
        value = $element.data('$' + require + 'Controller');
        if (value) {
          break;
        } else {
          $element = $element.parent();
        }
      }
    } else {
      if (controllers[require]) {
        value = controllers[require].instance;
```



```
      }
    }
    if (!value && !optional) {
      throw 'Controller '+require+' required by directive, cannot be found!';
    }
    return value || null;
  }
}
```

The final bit of `require` logic we are going to look at has to do with the syntax of the `^`, `^^`, and `?` prefixes. Angular actually lets you specify `?` *either as the suffix or the prefix* of `^` and `^^`. That means that both `?^` and `^?` are equally valid:

test/compile_spec.js

```
it('allows optional marker after parent marker', function() {
  var gotCtrl;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        require: '^?noSuchDirective',
        link: function(scope, element, attrs, ctrl) {
          gotCtrl = ctrl;
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(gotCtrl).toBe(null);
  });
});

it('allows optional marker before parent marker', function() {
  function MyController() { }
  var gotMyController;
  var injector = createInjector(['ng', function($compileProvider) {
    $compileProvider.directive('myDirective', function() {
      return {
        scope: {},
        controller: MyController
      };
    });
    $compileProvider.directive('myOtherDirective', function() {
      return {
        require: '?^myDirective',
        link: function(scope, element, attrs, ctrl) {
          gotMyController = ctrl;
```



```
        }
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');
    $compile(el)($rootScope);
    expect(gotMyController).toBeDefined();
    expect(gotMyController instanceof MyController).toBe(true);
  });
});
```

Our regular expression must try to match the ^ characters both before and after the question mark. We should use one or the other interchangeably:

src/compile.js

```
function getControllers(require, $element) {
  if (_.isArray(require)) {
    return _.map(require, getControllers);
  } else {
    var value;
    var match = require.match(/^(\^\^?)?(\?)?(\^\^?)?/);
    var optional = match[2];
    require = require.substring(match[0].length);
    if (match[1] || match[3]) {
      if (match[3] && !match[1]) {
        match[1] = match[3];
      }
      if (match[1] === '^^') {
        $element = $element.parent();
      }
      while ($element.length) {
        value = $element.data('$' + require + 'Controller');
        if (value) {
          break;
        } else {
          $element = $element.parent();
        }
      }
    } else {
      if (controllers[require]) {
        value = controllers[require].instance;
      }
    }
    if (!value && !optional) {
      throw 'Controller '+require+' required by directive, cannot be found!';
    }
    return value || null;
  }
}
```


And there we have a complete implementation of the `require` mechanism of Angular directives!

The ngController Directive

We'll wrap up the chapter with a discussion and implementation of `ngController` - a directive that's very familiar to pretty much every Angular application developer. For example, the second code example on the angularjs.org website uses it:

```
<div ng-controller="TodoController">
  <!-- ... -->
</div>
```

To begin our treatise on ngController, let's test that when we use it with the name of a registered controller constructor, that controller is indeed instantiated. This goes in a new test file, called `test/directives/ng_controller_spec.js`:

test/directives/ng_controller_spec.js

```
describe('ngController', function() {

  beforeEach(function() {
    delete window.angular;
    publishExternalAPI();
  });

  it('is instantiated during compilation & linking', function() {
    var instantiated;
    function MyController() {
      instantiated = true;
    }
    var injector = createInjector(['ng', function($controllerProvider) {
      $controllerProvider.register('MyController', MyController);
    }]);
    injector.invoke(function($compile, $rootScope) {
      var el = $('<div ng-controller="MyController"></div>');
      $compile(el)($rootScope);
      expect(instantiated).toBe(true);
    });
  });

});
```

Let's also test that the controller also receives `$scope`, `$element`, and `$attrs` as dependency-injected arguments if it requests them:

test/directives/ng_controller_spec.js


```
it('may inject scope, element, and attrs', function() {
  var gotScope, gotElement, gotAttrs;
  function MyController($scope, $element, $attrs) {
    gotScope = $scope;
    gotElement = $element;
    gotAttrs = $attrs;
  }
  var injector = createInjector(['ng', function($controllerProvider) {
    $controllerProvider.register('MyController', MyController);
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div ng-controller="MyController"></div>');
    $compile(el)($rootScope);
    expect(gotScope).toBeDefined();
    expect(gotElement).toBeDefined();
    expect(gotAttrs).toBeDefined();
  });
});
```

And while we're at it, let's test that the scope received by the constructor is a scope *inherited* from the surrounding scope - meaning that `ngController` should create a new (non-isolated) scope:

test/directives/ng_controller_spec.js

```
it('has an inherited scope', function() {
  var gotScope;
  function MyController($scope, $element, $attrs) {
    gotScope = $scope;
  }
  var injector = createInjector(['ng', function($controllerProvider) {
    $controllerProvider.register('MyController', MyController);
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div ng-controller="MyController"></div>');
    $compile(el)($rootScope);
    expect(gotScope).not.toBe($rootScope);
    expect(gotScope.$parent).toBe($rootScope);
    expect(Object.getPrototypeOf(gotScope)).toBe($rootScope);
  });
});
```

Now, let's create something that passes these tests. We'll put it in a new file called `src/directives/ng_controller.js`. The implementation is remarkably simple. Here it is in its entirety:

src/directives/ng_controller.js


```
var ngControllerDirective = function() {
  'use strict';

  return {
    restrict: 'A',
    scope: true,
    controller: '@'
  };
};
```

To make our tests pass, all we need to do is include this new directive as part of the `ng` module in `angular_public.js`:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
  ngModule.provider('$httpBackend', $HttpBackendProvider);
  ngModule.provider('$http', $HttpProvider);
  ngModule.provider('$httpParamSerializer', $HttpParamSerializerProvider);
  ngModule.provider('$httpParamSerializerJQLike',
    $HttpParamSerializerJQLikeProvider);
  ngModule.provider('$compile', $CompileProvider);
  ngModule.provider('$controller', $ControllerProvider);
  ngModule.directive('ngController', ngControllerDirective);
}
```

The simplicity of `ngController` may be surprising. That's because it is so prevalently used in application code that it seems that it must be a major part of the Angular framework architecture. In actual fact, the whole implementation of `ngController` just falls out of the implementation of the `$controller` service and the support for controllers in `$compile`.

Attaching Controllers on The Scope

Earlier in the chapter we saw how you can attach a controller on the scope by defining the `controllerAs` attribute on the directive definition object.

There is actually another way you can accomplish this attachment, which is most often used in conjunction with `ngController`. That is to define the alias right in the string that defines the controller constructor name:


```
<div ng-controller="TodoController as todoCtrl">
<!-- ... -->
</div>
```

Let's add a test case for this into the `ngController` test suite:

test/directives/ng_controller_spec.js

```
it('allows aliasing controller in expression', function() {
  var gotScope;
  function MyController($scope) {
    gotScope = $scope;
  }
  var injector = createInjector(['ng', function($controllerProvider) {
    $controllerProvider.register('MyController', MyController);
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div ng-controller="MyController as myCtrl"></div>');
    $compile(el)($rootScope);
    expect(gotScope.myCtrl).toBeDefined();
    expect(gotScope.myCtrl instanceof MyController).toBe(true);
  });
});
```

Now, even though we have the test in `ng_controller_spec.js`, the implementation of this feature is not actually in ngController. It is in the `$controller` service. As the controller is looked up, `$controller` first extracts the actual controller name and the optional alias from the given string.

src/controller.js

```
return function(ctrl, locals, later, identifier) {
  if (_.isString(ctrl)) {
    var match = ctrl.match(/^(\S+)(\s+as\s+(\w+))?/);
    ctrl = match[1];
    if (controllers.hasOwnProperty(ctrl)) {
      ctrl = controllers[ctrl];
      } else if (globals) {
      ctrl = window[ctrl];
    }
  }
  // ...
}
```

The regex matches a group of non-whitespace characters as the controller name, and then optionally the word `'as'` surrounded by whitespace, followed by a group of word characters that specify the identifier.

If there is an identifier, we should assign it to the `identifier` variable - unless a value for it was already explicitly given by the caller. This triggers our existing logic in `addToScope` and attaches the controller on the scope, making our test pass.

src/controller.js


```
return function(ctrl, locals, later, identifier) {
  if (_.isString(ctrl)) {
    var match = ctrl.match(/^(\S+)(\s+as\s+(\w+))?/);
    ctrl = match[1];
    identifier = identifier || match[3];
    if (controllers.hasOwnProperty(ctrl)) {
      ctrl = controllers[ctrl];
    } else if (globals) {
      ctrl = window[ctrl];
    }
  }
  // ...
}
```

Note that while the "controller as" syntax is most often used in conjunction with ngController, that doesn't have to be the case. Since the feature is implemented by the $controller service, you could just as well use it with directive controllers, specifying something like `'MyCtrl as myCtrl'` as the value of the `controller` attribute instead of specifying both `controller` and `controllerAs` separately.

Looking Up A Controller Constructor from The Scope

There's one more thing that the controller expressions given to `$controller` can do, which is to refer to a controller constructor function *attached to the scope* instead of a controller constructor function registered to `$controllerProvider`.

Here we have a test for a controller called `MyCtrlOnScope` being used by `ngController`. In fact, no controller has been registered by that name *but* there is a function on the scope matching that key. That function will be found and used to construct the controller:

test/directives/ng_controller_spec.js

```
it('allows looking up controller from surrounding scope', function() {
  var gotScope;
  function MyController($scope) {
    gotScope = $scope;
  }
  var injector = createInjector(['ng']);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div ng-controller="MyCtrlOnScope as myCtrl"></div>');
    $rootScope.MyCtrlOnScope = MyController;
    $compile(el)($rootScope);
    expect(gotScope.myCtrl).toBeDefined();
    expect(gotScope.myCtrl instanceof MyController).toBe(true);
  });
});
```


When `$controller` is trying to find the controller, it will now need to also look at the `$scope` attribute of the given `locals` object (if there is one), before resorting to a global lookup:

src/controller.js

```
return function(ctrl, locals, later, identifier) {
  if (_.isString(ctrl)) {
    var match = ctrl.match(/^(\S+)(\s+as\s+(\w+))?/);
    ctrl = match[1];
    identifier = identifier || match[3];
    if (controllers.hasOwnProperty(ctrl)) {
      ctrl = controllers[ctrl];
    } else {
      ctrl = (locals && locals.$scope && locals.$scope[ctrl]) ||
             (globals && window[ctrl]);
    }
  }
  // ...
};
```

Summary

Controllers are an important part of Angular applications, and in this chapter we have learned pretty much everything there is to know about them.

The implementation of controllers is perhaps surprisingly tightly coupled with directives. Even when "standalone" controllers are used, as with `ngController`, it's really just an application of the directive controller system.

In this chapter you've learned:

- How controllers can be instantiated by the `$controller` service.
- How controller instantiation supports dependency injection.
- How controllers can be pre-registered in the config phase using `$controllerProvider` or modules.
- That controllers can be optionally looked up from `window`, even though doing it is not recommended.
- How controllers can be attached to directives.
- That every directive gets its own controller instance, and that there can be many directives with controllers on the same DOM node.
- How the special `@` value can be used to defer the directive controller resolution to the directive user.
- How `$scope`, `$element`, and `$attrs` are made available for dependency injection in directive controllers.

- How a directive controller can be attached to the directive's scope using `controllerAs`.
- How isolate scope bindings can be attached to controller objects instead of the isolate scope using `bindToController`.
- How bindings can be attached using `bindToController` even without using an isolate scope.
- How controllers are instantiated in a deferred fashion inside `$compile` to support `bindToController`.
- How a sibling directive's controller can be required by specifying a `require` attribute whose value is the sibling directive's name.
- How multiple `require`s can be specified using arrays.
- That if a directive has a controller and does not require any other directives, it is as if the directive requires itself.
- How parent directives can be required with `^` and `^^`.
- How requires can be made optional with `?`.
- How the `ngController` directive works.
- How controllers can also be attached on the scope by giving a `ControllerConstructor as scopeName` style expression to `$controller`.
- How `$controller` also tries to find controller constructors from the scope, in addition to its own register and `window`.

Chapter 19

Directive Templates

Attaching directives to an existing DOM is one thing, but it is also very common to have directives that construct their *own* DOM: A directive may not only decorate an existing element, but also actually populate the element's contents with its own. For instance, if you use a directive like `<login-form>`, you might expect it to actually render a login form with inputs, buttons, and labels.

On some level our directive system already supports this, since you can do arbitrary DOM manipulation in directive `compile` and `link` functions. You can construct whatever content the directive needs using the JavaScript DOM API.

That is not very convenient though. It would be preferable to just supply the framework with a string of HTML and say "when the directive is applied, populate the element's contents with this". That's precisely what *directive templates* are for, and that is what we will implement in this chapter.

> Directive templates can be especially useful when combined with isolate scopes, allowing you to construct your application UI from "components". This is where Angular 2 is headed, though you can also adopt the style in Angular 1.x apps.

What We Will Skip

There are a couple of features in Angular's directive template support that won't be covered in this book.

Firstly, whenever you use a directive template, that template is used to populate the *children* of the current element. It is possible to modify this behavior so that the template actually *replaces* the element on which you applied the directive. You just need to configure `replace: true` on the directive definition object. However, this behavior is deprecated and the Angular team does not encourage its use. Because of this, we're not going to implement `replace`.

Secondly, we're also going to skip some of the caching functionality that's built into asynchronously loaded templates. We're just going to use `$http` directly instead.

Basic Templating

The basic idea behind directive templates is simple: A directive may define a `template` attribute in its directive definition object. That attribute holds a string of HTML code. When the directive is compiled into an element in the DOM, the contents of the element will be populated with that HTML code.

Here's this behavior in action:

test/compile_spec.js

```
describe('template', function() {

  it('populates an element during compilation', function() {
    var injector = makeInjectorWithDirectives('myDirective', function() {
      return {
        template: '<div class="from-template"></div>'
      };
    });
    injector.invoke(function($compile) {
      var el = $('<div my-directive></div>');
      $compile(el);
      expect(el.find('> .from-template').length).toBe(1);
    });
  });

});
```

If you use a template directive on an element that already has some contents, those contents will be replaced by the template. The previous child elements no longer exist after the template has been applied:

test/compile_spec.js

```
it('replaces any existing children', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      template: '<div class="from-template"></div>'
    };
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive><div class="existing"></div></div>');
    $compile(el);
    expect(el.find('> .existing').length).toBe(0);
  });
});
```

The contents of the template are not just static HTML. They are also compiled so that any directives used in templates also get applied. We can check this by applying a directive on an element inside the template and spying on its compile function:

test/compile_spec.js


```
it('compiles template contents also', function() {
  var compileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        template: '<div my-other-directive></div>'
      };
    },
    myOtherDirective: function() {
      return {
        compile: compileSpy
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive></div>');
    $compile(el);
    expect(compileSpy).toHaveBeenCalled();
  });
});
```

That's the basic behavior of templates. Let's see how we can make it work.

Templates are applied during compilation, in the `applyDirectivesToNode` function. In this function, as we iterate over each directive, we can just check if one of them has a template. If we see one, we'll replace the element's inner HTML with that template:

src/compile.js

```
_.forEach(directives, function(directive) {
  if (directive.$$start) {
    $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
  }

  if (directive.priority < terminalPriority) {
    return false;
  }

  if (directive.scope) {
    if (_.isObject(directive.scope)) {
      if (newIsolateScopeDirective || newScopeDirective) {
        throw 'Multiple directives asking for new/inherited scope';
      }
      newIsolateScopeDirective = directive;
    } else {
      if (newIsolateScopeDirective) {
        throw 'Multiple directives asking for new/inherited scope';
      }
```



```
      newScopeDirective = newScopeDirective || directive;
    }
  }
  if (directive.compile) {
    var linkFn = directive.compile($compileNode, attrs);
    var isolateScope = (directive === newIsolateScopeDirective);
    var attrStart = directive.$$start;
    var attrEnd = directive.$$end;
    var require = directive.require;
    if (_.isFunction(linkFn)) {
      addLinkFns(null, linkFn, attrStart, attrEnd, isolateScope, require);
    } else if (linkFn) {
      addLinkFns(linkFn.pre, linkFn.post, attrStart, attrEnd, isolateScope, require)
    }
  }
  if (directive.controller) {
    controllerDirectives = controllerDirectives || {};
    controllerDirectives[directive.name] = directive;
  }
  if (directive.template) {
    $compileNode.html(directive.template);
  }
  if (directive.terminal) {
    terminal = true;
    terminalPriority = directive.priority;
  }
});
```

This immediately passes all our tests! Any existing contents of the element are replaced by the template, and when we eventually go compile the element's children, the elements from the template will already be there.

For the remainder of the chapter we're going to deal with all kinds of subtle details related to directive templates, but it is good to remember that underneath it all, both the idea and the implementation of directive templates are very simple.

Disallowing More Than One Template Directive Per Element

Since a template directive replaces the element's contents, it makes very little sense to apply two or more directives with templates on the same element. Only the last one's template would remain.

Angular explicitly throws an exception during compilation when you try to do this, so that the problem is obvious:

test/compile_spec.js

```
it('does not allow two directives with templates', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {template: '<div></div>'};
    },
    myOtherDirective: function() {
      return {template: '<div></div>'};
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive my-other-directive></div>');
    expect(function() {
      $compile(el);
    }).toThrow();
  });
});
```

The implementation for this is similar to what we did when we were checking duplicate inherited scopes. We'll introduce a variable with which we track any template directive that we have seen so far:

src/compile.js

```
function applyDirectivesToNode(directives, compileNode, attrs) {
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = [], postLinkFns = [], controllers = {};
  var newScopeDirective, newIsolateScopeDirective;
  var templateDirective;
  var controllerDirectives;

  // ...

}
```

Now that we have that variable, we can assign it when we encounter a template directive, and also check that we haven't already encountered one earlier:

src/compile.js

```
if (directive.template) {
  if (templateDirective) {
    throw 'Multiple directives asking for template';
  }
  templateDirective = directive;
  $compileNode.html(directive.template);
}
```

Template Functions

The value of the `template` attribute doesn't necessarily have to be a string. It can also be a *function* that returns a string. The function is called with two arguments: The DOM node the directive is being applied on, and that node's `Attributes` object. This gives you a chance to dynamically construct the template:

test/compile_spec.js

```
it('supports functions as template values', function() {
  var templateSpy = jasmine.createSpy()
    .and.returnValue('<div class="from-template"></div>');
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        template: templateSpy
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive></div>');
    $compile(el);
    expect(el.find('> .from-template').length).toBe(1);
    // Check that template function was called with element and attrs
    expect(templateSpy.calls.first().args[0][0]).toBe(el[0]);
    expect(templateSpy.calls.first().args[1].myDirective).toBeDefined();
  });
});
```

If the template is a function, we'll just invoke it instead of using it directly:

src/compile.js

```
if (directive.template) {
  if (templateDirective) {
    throw 'Multiple directives asking for template';
  }
  templateDirective = directive;
  $compileNode.html(_.isFunction(directive.template) ?
                    directive.template($compileNode, attrs) :
                    directive.template);
}
```


Isolate Scope Directives with Templates

When we implemented isolate scopes a couple of chapters ago, we discussed that an isolate scope is only ever used for the directive that asks for it - not for other directives on the element or its children.

There's an exception to this, and it is related to templates: When a directive defines both an isolate scope and a template, the directives used *inside the template* will receive the isolate scope (or one of its descendants). The template contents are considered to be part of the isolate. This makes sense when you think of this kind of directive as a component that "owns" its own template.

test/compile_spec.js

```
it('uses isolate scope for template contents', function() {
  var linkSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        scope: {
          isoValue: '=myDirective'
        },
        template: '<div my-other-directive></div>'
      };
    },
    myOtherDirective: function() {
      return {link: linkSpy};
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive="42"></div>');
    $compile(el)($rootScope);
    expect(linkSpy.calls.first().args[0]).not.toBe($rootScope);
    expect(linkSpy.calls.first().args[0].isoValue).toBe(42);
  });
});
```

In the node link function, as we call the child link function, we have thus far only been using the surrounding scope, never the isolate scope. We'll change this now, so that if we have an isolate scope directive, and that directive also has a template, we'll link the child nodes with the isolate scope. We can do this, because if this element has any children, they must have come from the isolate scope directive's template.

src/compile.js

```
_.forEach(preLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
```



```
    linkFn.require && getControllers(linkFn.require, $element)
  );
});
if (childLinkFn) {
  var scopeToChild = scope;
  if (newIsolateScopeDirective && newIsolateScopeDirective.template) {
    scopeToChild = isolateScope;
  }
  childLinkFn(scopeToChild, linkNode.childNodes);
}
_.forEachRight(postLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element)
  );
});
```

Asynchronous Templates: `templateUrl`

When you use the `template` attribute, you define the directive's template HTML inline inside a string in your JavaScript code. That's not a very convenient place for storing HTML, especially if there's a lot of it.

It is much more convenient to store HTML templates in separate `.html` files, and then load them into the application. For this purpose, Angular supports the `templateUrl` directive attribute. When that is defined, the template is loaded over HTTP from the specified URL.

Since loading things over HTTP is always asynchronous, this means that when we encounter a directive with a template URL, we need to *pause compilation* while the template is loading. We then need to resume compilation when the template arrives. The majority of the remainder of this chapter deals with issues caused by this pause-and-resume requirement.

The very first tests we'll add for this feature are for checking what the compiler *shouldn't* do when it encounters an asynchronous template directive.

First of all, any remaining directives that the element has should *not* be compiled at this point:

test/compile_spec.js

```
describe('templateUrl', function() {

  it('defers remaining directive compilation', function() {
    var otherCompileSpy = jasmine.createSpy();
    var injector = makeInjectorWithDirectives({
      myDirective: function() {
        return {templateUrl: '/my_directive.html'};
      },
```

```
      myOtherDirective: function() {
        return {compile: otherCompileSpy};
      }
    });
    injector.invoke(function($compile) {
      var el = $('<div my-directive my-other-directive></div>');
      $compile(el);
      expect(otherCompileSpy).not.toHaveBeenCalled();
    });
  });

});
```

For now, all we'll do is to just short-circuit the directive loop when we see a directive with a `templateUrl` attribute. We can do this by returning `false` from the loop, since LoDash `_.forEach` will end the loop for us when that happens:

src/compile.js

```
if (directive.template) {
  if (templateDirective) {
    throw 'Multiple directives asking for template';
  }
  templateDirective = directive;
  $compileNode.html(_.isFunction(directive.template) ?
                    directive.template($compileNode, attrs) :
                    directive.template);
}
if (directive.templateUrl) {
 return false;
}
```

Not only should the compiler short circuit the compilation of *other* directives, but it shouldn't even compile the *current* directive yet when it sees a template URL:

test/compile_spec.js

```
it('defers current directive compilation', function() {
  var compileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        templateUrl: '/my_directive.html',
        compile: compileSpy
      };
    }
  });
  injector.invoke(function($compile) {
```



```
    var el = $('<div my-directive></div>');
    $compile(el);
    expect(compileSpy).not.toHaveBeenCalled();
  });
});
```

To pass this requirement, we need to move things around a bit. The "`if (directive.compile) { }`" block should be moved so that it is in an `else if` branch after the `templateUrl` check. All the code inside the block remains unchanged. It is just moved so that it's not called when a directive has a `templateUrl`:

src/compile.js

```
_.forEach(directives, function(directive) {
  if (directive.$$start) {
    $compileNode = groupScan(compileNode, directive.$$start, directive.$$end);
  }

  if (directive.priority < terminalPriority) {
    return false;
  }

  if (directive.scope) {
    if (_.isObject(directive.scope)) {
      if (newIsolateScopeDirective || newScopeDirective) {
        throw 'Multiple directives asking for new/inherited scope';
      }
      newIsolateScopeDirective = directive;
    } else {
      if (newIsolateScopeDirective) {
        throw 'Multiple directives asking for new/inherited scope';
      }
      newScopeDirective = newScopeDirective || directive;
    }
  }
  if (directive.controller) {
    controllerDirectives = controllerDirectives || {};
    controllerDirectives[directive.name] = directive;
  }
  if (directive.template) {
    if (templateDirective) {
      throw 'Multiple directives asking for template';
    }
    templateDirective = directive;
    $compileNode.html(_.isFunction(directive.template) ?
                      directive.template($compileNode, attrs) :
                      directive.template);
  }
  if (directive.templateUrl) {
```



```
    return false;
  } else if (directive.compile) {
    var linkFn = directive.compile($compileNode, attrs);
    var isolateScope = (directive === newIsolateScopeDirective);
    var attrStart = directive.$$start;
    var attrEnd = directive.$$end;
    var require = directive.require;
    if (_.isFunction(linkFn)) {
      addLinkFns(null, linkFn, attrStart, attrEnd, isolateScope, require);
    } else if (linkFn) {
      addLinkFns(linkFn.pre, linkFn.post,
        attrStart, attrEnd, isolateScope, require);
    }
  }
  if (directive.terminal) {
    terminal = true;
    terminalPriority = directive.priority;
  }
});
```

Another thing that should happen at this point is that the current element's contents should be removed. They will eventually be replaced by the template's contents when it arrives, but we need to immediately remove any old contents, so that they won't get unnecessarily compiled:

test/compile_spec.js

```
it('immediately empties out the element', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {templateUrl: '/my_directive.html'};
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive>Hello</div>');
    $compile(el);
    expect(el.is(':empty')).toBe(true);
  });
});
```

For this purpose, we're going to introduce a new function called `compileTemplateUrl`, whose job will be to handle all the asynchronous work that goes into resolving the template:

src/compile.js

```
if (directive.templateUrl) {
  compileTemplateUrl($compileNode);
  return false;
}
```


This function (introduced *outside* of the `applyTemplatesToNode` function) will at this point do nothing but clear the node, making our current test suite pass:

src/compile.js

```
function compileTemplateUrl($compileNode) {
  $compileNode.empty();
}
```

What we have effectively implemented now is a suspension of the compilation process for this DOM subtree when a `templateUrl` is seen: The current directive or other directives on the current element won't be compiled, and the element's children have been removed.

Now we can start thinking about how to get to a point where we can *resume* the compilation. What we need to do is fetch the template specified in the URL. This we can do with the `$http` service that we implemented in the previous part of the book.

In order to test template fetching, we're going to need to install the fake XMLHttpRequest support from Sinon.js like we did in the `$http` chapter. Add the following setup code to the `describe('templateUrl')` test block:

test/compile_spec.js

```
describe('templateUrl', function() {

  var xhr, requests;

  beforeEach(function() {
    xhr = sinon.useFakeXMLHttpRequest();
    requests = [];
    xhr.onCreate = function(req) {
      requests.push(req);
    };
  });
  afterEach(function() {
    xhr.restore();
  });

  // ...

});
```

Now we can add the first test that deals with template loading. It checks that when a directive with a `templateUrl` is compiled, a GET request to that URL is made:

test/compile_spec.js

```
it('fetches the template', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {templateUrl: '/my_directive.html'};
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');

    $compile(el);
    $rootScope.$apply();

    expect(requests.length).toBe(1);
    expect(requests[0].method).toBe('GET');
    expect(requests[0].url).toBe('/my_directive.html');
  });
});
```

Note the `$apply` call after the `$compile` call. We need it to kick off the Promise chain within `$http`.

We're going to make the HTTP request from `compileTemplateUrl`. Before it can do that, it needs access to the directive object, so we should pass it in:

src/compile.js

```
if (directive.templateUrl) {
  compileTemplateUrl(directive, $compileNode);
  return false;
} else if (directive.compile) {
```

We can use the `get` method of the `$http` service to make the actual request:

src/compile.js

```
function compileTemplateUrl(directive, $compileNode) {
  $compileNode.empty();
  $http.get(directive.templateUrl);
}
```

We don't have `$http` in the `$compile` service yet, so we need to add an injection for it into `CompileProvider.$get`:

src/compile.js

```
this.$get = ['$injector', '$parse', '$controller', '$rootScope', '$http',
    function($injector, $parse, $controller, $rootScope, $http) {
```


This now satisfies our test.

What should happen when the template is eventually received? The most obvious effect is that the element's contents should be populated from the template:

test/compile_spec.js

```
it('populates element with template', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {templateUrl: '/my_directive.html'};
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');

    $compile(el);
    $rootScope.$apply();

    requests[0].respond(200, {}, '<div class="from-template"></div>');
    expect(el.find('> .from-template').length).toBe(1);
  });
});
```

This we can do by attaching a `success` handler to the Promise returned from the `$http` call. The first argument of the handler will be the response body - the template HTML:

src/compile.js

```
function compileTemplateUrl(directive, $compileNode) {
  $compileNode.empty();
  $http.get(directive.templateUrl).success(function(template) {
    $compileNode.html(template);
  });
}
```

Now we have the DOM in a state where we can resume directive compilation. This means that when a template response is received, we should also expect the current directive's `compile` function to finally get invoked:

test/compile_spec.js


```
it('compiles current directive when template received', function() {
  var compileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        templateUrl: '/my_directive.html',
        compile: compileSpy
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');

    $compile(el);
    $rootScope.$apply();

    requests[0].respond(200, {}, '<div class="from-template"></div>');
    expect(compileSpy).toHaveBeenCalled();
  });
});
```

The same is true for any remaining directives on the element - the ones we short-circuited in `applyDirectivesToNode`. We should now compile them as well:

test/compile_spec.js

```
it('resumes compilation when template received', function() {
  var otherCompileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {templateUrl: '/my_directive.html'};
    },
    myOtherDirective: function() {
      return {compile: otherCompileSpy};
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');

    $compile(el);
    $rootScope.$apply();

    requests[0].respond(200, {}, '<div class="from-template"></div>');
    expect(otherCompileSpy).toHaveBeenCalled();
  });
});
```

How should this work? What we would really like to do is somehow resume the logic of `applyDirectivesToNode` again when we have received the template, but to do it only for the directives that haven't been compiled yet. And that is exactly what we're going to do.

To begin with, `compileTemplateUrl` will need access not only to the current directive, but to all directives that haven't been applied yet. In other words, it needs a subarray of all the directives we have, starting from the index of the current directive. Additionally, we'll pass it the `Attributes` object of the current element, because we're going to need that too:

src/compile.js

```
if (directive.templateUrl) {
  compileTemplateUrl(_.drop(directives, i), $compileNode, attrs);
  return false;
```

The `i` variable here does not exist yet. We should introduce it as the second argument of the directive `_.forEach` loop. It refers to the current index in the loop:

src/compile.js

```
_.forEach(directives, function(directive, i) {

  // ...

});
```

Now, the first argument in `compileTemplateUrl` will be an array, and the directive with the `templateUrl` will be the first item of the array:

src/compile.js

```
function compileTemplateUrl(directives, $compileNode, attrs) {
  var origAsyncDirective = directives[0];
  $compileNode.empty();
  $http.get(origAsyncDirective.templateUrl).success(function(template) {
    $compileNode.html(template);
  });
}
```

Now we also have everything we need to call *back* to `applyDirectivesToNode`:

src/compile.js

```
function compileTemplateUrl(directives, $compileNode, attrs) {
  var origAsyncDirective = directives[0];
  $compileNode.empty();
  $http.get(origAsyncDirective.templateUrl).success(function(template) {
    $compileNode.html(template);
    applyDirectivesToNode(directives, $compileNode, attrs);
  });
}
```


There's still a problem with this though. We are resuming compilation from the directive that has the `templateUrl` attribute. That means that `applyDirectivesToNode` will immediately see the `templateUrl` and stop the compilation *again*. We're forever stuck fetching the template over and over.

We can fix this by first removing the asynchronous template directive from the directive array:

src/compile.js

```
function compileTemplateUrl(directives, $compileNode, attrs) {
  var origAsyncDirective = directives.shift();
  $compileNode.empty();
  $http.get(origAsyncDirective.templateUrl).success(function(template) {
    $compileNode.html(template);
    applyDirectivesToNode(directives, $compileNode, attrs);
  });
}
```

We'll then replace it with a *new* directive object that copies all the attributes of the original directive, but sets the `templateUrl` to `null`.

src/compile.js

```
function compileTemplateUrl(directives, $compileNode, attrs) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {templateUrl: null}
  );
  $compileNode.empty();
  $http.get(origAsyncDirective.templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    applyDirectivesToNode(directives, $compileNode, attrs);
  });
}
```

Another thing we're still missing from our resumed compilation process is the compilation of child nodes. The `applyDirectivesToNode` function does not do it.

test/compile_spec.js


```
it('resumes child compilation after template received', function() {
  var otherCompileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {templateUrl: '/my_directive.html'};
    },
    myOtherDirective: function() {
      return {compile: otherCompileSpy};
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');

    $compile(el);
    $rootScope.$apply();

    requests[0].respond(200, {}, '<div my-other-directive></div>');
    expect(otherCompileSpy).toHaveBeenCalled();
  });
});
```

All we need to do to make this work is to call `compileNodes` with the child nodes of the current node - which will at this point contain the child nodes that came from the template:

src/compile.js

```
function compileTemplateUrl(directives, $compileNode, attrs) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend({}, origAsyncDirective, {templateUrl: null});
  $compileNode.empty();
  $http.get(origAsyncDirective.templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    applyDirectivesToNode(directives, $compileNode, attrs);
    compileNodes($compileNode[0].childNodes);
  });
}
```

Template URL Functions

Just like inline templates can be defined as functions instead of strings, so can template URLs. The function signature is exactly the same for both. There are two arguments: The current node and its Attributes.

test/compile_spec.js

```
it('supports functions as values', function() {
  var templateUrlSpy = jasmine.createSpy()
    .and.returnValue('/my_directive.html');
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        templateUrl: templateUrlSpy
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');

    $compile(el);
    $rootScope.$apply();

    expect(requests[0].url).toBe('/my_directive.html');
    expect(templateUrlSpy.calls.first().args[0][0]).toBe(el[0]);
    expect(templateUrlSpy.calls.first().args[1].myDirective).toBeDefined();
  });
});
```

The implementation is also very similar. We simply see if the `teplateUrl` value is a function, and if it is we invoke it:

src/compile.js

```
function compileTemplateUrl(directives, $compileNode, attrs) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend({}, origAsyncDirective, {templateUrl: null});
  var templateUrl = _.isFunction(origAsyncDirective.templateUrl) ?
                      origAsyncDirective.templateUrl($compileNode, attrs) :
                      origAsyncDirective.templateUrl;
  $compileNode.empty();
  $http.get(templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    applyDirectivesToNode(directives, $compileNode, attrs);
    compileNodes($compileNode[0].childNodes);
  });
}
```

Disallowing More Than One Template URL Directive Per Element

Earlier in the chapter we added a check for making sure that no more than one directive is trying to apply a template to the same element. We should extend this check to

cover asynchronous template directives too. When a `template` directive has been seen, a `templateUrl` should not be allowed later:

test/compile_spec.js

```
it('does not allow templateUrl directive after template directive', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {template: '<div></div>'};
    },
    myOtherDirective: function() {
      return {templateUrl: '/my_other_directive.html'};
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div my-directive my-other-directive></div>');
    expect(function() {
      $compile(el);
    }).toThrow();
  });
});
```

We can cover this case by simply adding a check to the `templateUrl` branch of `applyDirectivesToNode`. If we've seen a template directive earlier, we throw an exception:

src/compile.js

```
if (directive.templateUrl) {
  if (templateDirective) {
    throw 'Multiple directives asking for template';
  }
  templateDirective = directive;
  compileTemplateUrl(_.drop(directives, i), $compileNode, attrs);
  return false;
```

The same check should also apply in the opposite order. When a `templateUrl` directive has been seen, a `template` directive should not be allowed later:

test/compile_spec.js

```
it('does not allow template directive after templateUrl directive', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {templateUrl: '/my_directive.html'};
    },
    myOtherDirective: function() {
      return {template: '<div></div>'};
    }
```



```
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');

    $compile(el);
    $rootScope.$apply();

    requests[0].respond(200, {}, '<div class="replacement"></div>');
    expect(el.find('> .replacement').length).toBe(1);
  });
});
```

Note that in this case we don't get an exception, because the check is done after the template URL has been asynchronously resolved, which happens in a separate execution context. Instead we simply check that the template from the second directive does not get applied.

This check is a bit more tricky, since by the time we're looking at the second directive, we're in the *second* invocation of `applyDirectivesToNode`, where all the local variables including `templateDirective` from the previous invocation have been cleared.

What we really need to do is *preserve the information* of any template directives we've seen between the two invocations. We essentially need to pass some state from one invocation of `applyDirectivesToNode` to another one, via the `compileTemplateUrl` invocation that happens in between.

For this purpose, we'll introduce an object that we'll call the *previous compile context*, into which we add the state that we should preserve. Its purpose is to "transport" some local state between two different invocations of the `applyDirectivesToNode` function. At this point the only thing we'll add to it is the template directive variable, but we'll add more later.

The previous compile context is given to `compileTemplateUrl` as the last argument:

src/compile.js

```
compileTemplateUrl(
  _.drop(directives, i),
  $compileNode,
  attrs,
  {templateDirective: templateDirective}
);
```

The `compileTemplateUrl` function does nothing with the previous compile context except pass it back to `applyDirectivesToNode` when it calls it for the second time:

src/compile.js

```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {templateUrl: null}
  );
  var templateUrl = _.isFunction(origAsyncDirective.templateUrl) ?
                      origAsyncDirective.templateUrl($compileNode, attrs) :
                      origAsyncDirective.templateUrl;
  $compileNode.empty();
  $http.get(templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    applyDirectivesToNode(
      directives, $compileNode, attrs, previousCompileContext);
    compileNodes($compileNode[0].childNodes);
  });
}
```

Back in `applyDirectivesToNode` we'll now receive the previous compile context, and initialize the local `templateDirective` variable from it:

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = [], postLinkFns = [], controllers = {};
  var newScopeDirective, newIsolateScopeDirective;
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives;
```

And now our test passes. In the process we have introduced a simple mechanism with which we can retain state between two calls to `applyDirectivesToNode`, when there's an asynchronous template fetch in between. We'll be adding more state to that object in the remainder of this chapter.

Linking Asynchronous Directives

We are now able to fully resume the compilation process after an async template fetch, but we're missing the ability to also *link* all the directives that get compiled asynchronously.

What we're currently doing is simply discarding their link functions, because when `applyDirectivesToNode` is called for the second time, the node link function it returns is thrown away. This means these directives will never get linked.

This is not how it should be. When the public link function is called, the directives that were compiled asynchronously should be linked like any other directives:

test/compile_spec.js

```
it('links the directive when public link function is invoked', function() {
  var linkSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        templateUrl: '/my_directive.html',
        link: linkSpy
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');

    var linkFunction = $compile(el);
    $rootScope.$apply();

    requests[0].respond(200, {}, '<div></div>');

    linkFunction($rootScope);
    expect(linkSpy).toHaveBeenCalled();
    expect(linkSpy.calls.first().args[0]).toBe($rootScope);
    expect(linkSpy.calls.first().args[1][0]).toBe(el[0]);
    expect(linkSpy.calls.first().args[2].myDirective).toBeDefined();
  });
});
```

The same applies to all directives in the child elements that come from the template. We should expect them to be linked too, but they currently aren't. That's because we also throw away the return value of the `compileNodes()` invocation we make from `compileTemplateUrl`.

test/compile_spec.js

```
it('links child elements when public link function is invoked', function() {
  var linkSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {templateUrl: '/my_directive.html'};
    },
    myOtherDirective: function() {
      return {link: linkSpy};
```



```
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');

    var linkFunction = $compile(el);
    $rootScope.$apply();

    requests[0].respond(200, {}, '<div my-other-directive></div>');

    linkFunction($rootScope);
    expect(linkSpy).toHaveBeenCalled();
    expect(linkSpy.calls.first().args[0]).toBe($rootScope);
    expect(linkSpy.calls.first().args[1][0]).toBe(el[0].firstChild);
    expect(linkSpy.calls.first().args[2].myOtherDirective).toBeDefined();
  });
});
```

The trick we're going to apply to enable asynchronous linking consists of several steps. Let's take them one at a time.

The contract of `applyDirectivesToNode` is that it returns the node link function for that node. The node link function is defined inside `applyDirectivesToNode` with the statement `function nodeLinkFn(...)`.

When one of the directives on the node has an asynchronous template, we should *not* return this normal node link function, and instead return a special "delayed node link function". We'll set things up so that the delayed node link function will be returned by `compileTemplateUrl`. Assuming this will be the case, we can capture that return value and overwrite the local `nodeLinkFn` variable with it. The delayed node link functions becomes the return value of `applyDirectivesToNode`:

src/compile.js

```
if (directive.templateUrl) {
  if (templateDirective) {
    throw 'Multiple directives asking for template';
  }
  templateDirective = directive;
  nodeLinkFn = compileTemplateUrl(
    _.drop(directives, i),
    $compileNode,
    attrs,
    {templateDirective: templateDirective}
  );
  return false;
```

Inside `compileTemplateUrl` we should now introduce this delayed node link function. This function will be responsible for handling the whole linking process of this node - it has to

because we just replaced the regular node link function with it. The end result of all of this is that when the node link function for this node is called, what actually gets called is `delayedNodeLinkFn`:

src/compile.js

```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {templateUrl: null}
  );
  var templateUrl = _.isFunction(origAsyncDirective.templateUrl) ?
                      origAsyncDirective.templateUrl($compileNode, attrs) :
                      origAsyncDirective.templateUrl;
  $compileNode.empty();
  $http.get(templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    applyDirectivesToNode(
      directives, $compileNode, attrs, previousCompileContext);
    compileNodes($compileNode[0].childNodes);
  });

  return function delayedNodeLinkFn() {

  };
}
```

What exactly should we do inside the delayed node link function? One thing we should definitely do is link all the directives that we've compiled asynchronously - both from the current node and from the child nodes. We get the link functions for them by capturing the return values of the `applyDirectivesToNode` and `compileNodes` invocations that we were previously throwing away. We'll call them `afterTemplateNodeLinkFn` and `afterTemplateChildLinkFn`, since they are link functions for everything we linked *after* loading the template:

src/compile.js

```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {templateUrl: null}
  );
  var templateUrl = _.isFunction(origAsyncDirective.templateUrl) ?
```



```
                    origAsyncDirective.templateUrl($compileNode, attrs) :
                    origAsyncDirective.templateUrl;
  var afterTemplateNodeLinkFn, afterTemplateChildLinkFn;
  $compileNode.empty();
  $http.get(templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    afterTemplateNodeLinkFn = applyDirectivesToNode(
      directives, $compileNode, attrs, previousCompileContext);
    afterTemplateChildLinkFn = compileNodes($compileNode[0].childNodes);
  });

  return function delayedNodeLinkFn() {

  };
}
```

We should now call both of these functions from the delayed node link function. But first, let's think about the arguments the delayed node link function itself will receive. It will be called as a regular node link function, which means that it will get three arguments:

1. The child link function
2. The scope to link
3. The node being linked

The last two arguments are self-explanatory, but the first one - the child link function - is a bit more interesting: It will be the child link function from *before* we loaded the template. Since we cleared the node's children before we started loading the template, it will actually do nothing. We can safely ignore it, and instead just use `afterTemplateChildLinkFn` as we proceed with the linking:

src/compile.js

```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {templateUrl: null}
  );
  var templateUrl = _.isFunction(origAsyncDirective.templateUrl) ?
                    origAsyncDirective.templateUrl($compileNode, attrs) :
                    origAsyncDirective.templateUrl;
  var afterTemplateNodeLinkFn, afterTemplateChildLinkFn;
  $compileNode.empty();
  $http.get(templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
```

```
    afterTemplateNodeLinkFn = applyDirectivesToNode(
      directives, $compileNode, attrs, previousCompileContext);
    afterTemplateChildLinkFn = compileNodes($compileNode[0].childNodes);
  });

  return function delayedNodeLinkFn(_ignoreChildLinkFn, scope, linkNode) {
    afterTemplateNodeLinkFn(afterTemplateChildLinkFn, scope, linkNode);
  };
}
```

This satisfies our test case, but if you look at the order in which things are done in that test case, you may find it a bit strange: The test first waits for the template to be received, and *then* calls the public link function. Our current implementation actually necessitates this.

This is not a reasonable requirement. If we were to leave it at this, whoever calls the public link function would need to know when all asynchronous template fetches have been finished. In actual fact, as an Angular user, you don't have to think about this. It is indeed much more common to call the link function *immediately* after compilation finishes.

So we need to support the scenario where the public link function is called *before* the template has been received, and thus before we actually have the `afterTemplateNodeLinkFn` and `afterTemplateChildLinkFn` functions. What should happen in this case is that the linking occurs later when we finally do receive the template:

test/compile_spec.js

```
it('links when template arrives if node link fn was called', function() {
  var linkSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        templateUrl: '/my_directive.html',
        link: linkSpy
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');

    var linkFunction = $compile(el)($rootScope); // link first

    $rootScope.$apply();
    requests[0].respond(200, {}, '<div></div>'); // then receive template

    expect(linkSpy).toHaveBeenCalled();
    expect(linkSpy.calls.argsFor(0)[0]).toBe($rootScope);
    expect(linkSpy.calls.argsFor(0)[1][0]).toBe(el[0]);
    expect(linkSpy.calls.argsFor(0)[2].myDirective).toBeDefined();
  });
});
```


What this means that `delayedNodeLinkFn` may get called before we're ready to link. When this is the case, we should store the arguments we were given so that we can apply them when we're ready. Those arguments will go into a "link queue" that we store internally in `compileTemplateUrl`. We initialize it as an array first, and set it to `null` when we've received the template:

src/compile.js

```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {templateUrl: null}
  );
  var templateUrl = _.isFunction(origAsyncDirective.templateUrl) ?
                      origAsyncDirective.templateUrl($compileNode, attrs) :
                      origAsyncDirective.templateUrl;
  var afterTemplateNodeLinkFn, afterTemplateChildLinkFn;
  var linkQueue = [];
  $compileNode.empty();
  $http.get(templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    afterTemplateNodeLinkFn = applyDirectivesToNode(
      directives, $compileNode, attrs, previousCompileContext);
    afterTemplateChildLinkFn = compileNodes($compileNode[0].childNodes);
    linkQueue = null;
  });

  return function delayedNodeLinkFn(_ignoreChildLinkFn, scope, linkNode) {
    afterTemplateNodeLinkFn(afterTemplateChildLinkFn, scope, linkNode);
  };
}
```

In `delayedNodeLinkFn` we now have two choices: If there is a link queue, just put the arguments there because we're not ready yet. If there is no link queue (because it is `null`), just call the node link function right away as we did earlier:

src/compile.js

```
return function delayedNodeLinkFn(_ignoreChildLinkFn, scope, linkNode) {
  if (linkQueue) {
    linkQueue.push({scope: scope, linkNode: linkNode});
  } else {
    afterTemplateNodeLinkFn(afterTemplateChildLinkFn, scope, linkNode);
  }
};
```


Now, when we receive the template, if the link function has already been called, there will be one or more entries in the link queue. We'll apply those calls right away:

src/compile.js

```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {templateUrl: null}
  );
  var templateUrl = _.isFunction(origAsyncDirective.templateUrl) ?
                      origAsyncDirective.templateUrl($compileNode, attrs) :
                      origAsyncDirective.templateUrl;
  var afterTemplateNodeLinkFn, afterTemplateChildLinkFn;
  var linkQueue = [];
  $compileNode.empty();
  $http.get(templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    afterTemplateNodeLinkFn = applyDirectivesToNode(
      directives, $compileNode, attrs, previousCompileContext);
    afterTemplateChildLinkFn = compileNodes($compileNode[0].childNodes);
    _.forEach(linkQueue, function(linkCall) {
      afterTemplateNodeLinkFn(
        afterTemplateChildLinkFn, linkCall.scope, linkCall.linkNode);
    });
    linkQueue = null;
  });

  return function delayedNodeLinkFn(_ignoreChildLinkFn, scope, linkNode) {
    if (linkQueue) {
      linkQueue.push({scope: scope, linkNode: linkNode});
    } else {
      afterTemplateNodeLinkFn(afterTemplateChildLinkFn, scope, linkNode);
    }
  };
}
```

> Usually the link function is only called once, so there's really no need to have a *queue* of multiple link invocations. Since technically a link function can be called multiple times, this capability is preserved for asynchronous templates.

The link queue essentially time-shifts the linking process for this DOM subtree, so that it is only initiated when the asynchronous template fetch has completed.

Note that this also means that when you call a public link function in Angular, you can't always be sure that everything has been linked when the function returns. If there are asynchronous template directives in the DOM, the linking will only finish when they're done loading.

Linking Directives that Were Compiled Earlier

There are a few cases we still need to cover before we can say that our asynchronous compiler and linker is fully functional. One glaring omission is the linking of directives that were compiled *before* an asynchronous template was seen on the same element. We collected their link functions to the `preLinkFns` and `postLinkFns` collections, but then we simply threw them away as we replaced the node link function with the delayed node link function.

test/compile_spec.js

```
it('links directives that were compiled earlier', function() {
  var linkSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {link: linkSpy};
    },
    myOtherDirective: function() {
      return {templateUrl: '/my_other_directive.html'};
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');

    var linkFunction = $compile(el);
    $rootScope.$apply();

    linkFunction($rootScope);

    requests[0].respond(200, {}, '<div></div>');

    expect(linkSpy).toHaveBeenCalled();
    expect(linkSpy.calls.argsFor(0)[0]).toBe($rootScope);
    expect(linkSpy.calls.argsFor(0)[1][0]).toBe(el[0]);
    expect(linkSpy.calls.argsFor(0)[2].myDirective).toBeDefined();
  });
});
```

This is a situation where the "previous compile context" object we introduced earlier comes in handy. The pre-link functions and post-link function need to be preserved between the two calls to `applyDirectivesToNode`. We should add those collections to the context as we give it to `compileTemplateUrl`:

src/compile.js


```
nodeLinkFn = compileTemplateUrl(
  _.drop(directives, i),
  $compileNode,
  attrs,
  {
    templateDirective: templateDirective,
    preLinkFns: preLinkFns,
    postLinkFns: postLinkFns
  }
);
```

Then we also need to be ready to receive them when `applyDirectivesToNode` gets called for the second time:

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = previousCompileContext.preLinkFns || [];
  var postLinkFns = previousCompileContext.postLinkFns || [];
  var controllers = {};
  var newScopeDirective, newIsolateScopeDirective;
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives;

  // ...
}
```

And now we are keeping the same link function collections throughout asynchronous template loads. When we finally call the link functions, we call all of them, regardless of whether they were formed before or after a template was loaded.

Preserving The Isolate Scope Directive

Another thing that we currently "forget" as we go asynchronous is whether there was an isolate scope directive on the element. If there was one, its linking will fail:

test/compile_spec.js

```
it('retains isolate scope directives from earlier', function() {
  var linkSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        scope: {val: '=myDirective'},
        link: linkSpy
      };
    },
    myOtherDirective: function() {
      return {templateUrl: '/my_other_directive.html'};
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive="42" my-other-directive></div>');

    var linkFunction = $compile(el);
    $rootScope.$apply();

    linkFunction($rootScope);

    requests[0].respond(200, {}, '<div></div>');

    expect(linkSpy).toHaveBeenCalled();
    expect(linkSpy.calls.first().args[0]).toBeDefined();
    expect(linkSpy.calls.first().args[0]).not.toBe($rootScope);
    expect(linkSpy.calls.first().args[0].val).toBe(42);
  });
});
```

This is also something that should go on the previous compile context:

src/compile.js

```
nodeLinkFn = compileTemplateUrl(
  _.drop(directives, i),
  $compileNode,
  attrs,
  {
    templateDirective: templateDirective,
    newIsolateScopeDirective: newIsolateScopeDirective,
    preLinkFns: preLinkFns,
    postLinkFns: postLinkFns
  }
);
```

And correspondingly, we should unpack it when we come back:

src/compile.js


```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = previousCompileContext.preLinkFns || [];
  var postLinkFns = previousCompileContext.postLinkFns || [];
  var controllers = {};
  var newScopeDirective;
  var newIsolateScopeDirective = previousCompileContext.newIsolateScopeDirective;
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives;
```

Preserving Controller Directives

Finally, we should apply this trick one more time, for the controller directive mapping object. We currently forget all about the controller configurations that we saw before moving to delayed linking:

test/compile_spec.js

```
it('sets up controllers for all controller directives', function() {
  var myDirectiveControllerInstantiated, myOtherDirectiveControllerInstantiated;
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        controller: function MyDirectiveController() {
          myDirectiveControllerInstantiated = true;
        }
      };
    },
    myOtherDirective: function() {
      return {
        templateUrl: '/my_other_directive.html',
        controller: function MyOtherDirectiveController() {
          myOtherDirectiveControllerInstantiated = true;
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-other-directive></div>');

    $compile(el)($rootScope);
    $rootScope.$apply();
```



```
    requests[0].respond(200, {}, '<div></div>');

    expect(myDirectiveControllerInstantiated).toBe(true);
    expect(myOtherDirectiveControllerInstantiated).toBe(true);
  });
});
```

We should put `controllerDirectives` into the previous compile context:

src/compile.js

```
nodeLinkFn = compileTemplateUrl(
  _.drop(directives, i),
  $compileNode,
  attrs,
  {
    templateDirective: templateDirective,
    newIsolateScopeDirective: newIsolateScopeDirective,
    controllerDirectives: controllerDirectives,
    preLinkFns: preLinkFns,
    postLinkFns: postLinkFns
  }
);
```

And we should also get them back from the context:

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = previousCompileContext.preLinkFns || [];
  var postLinkFns = previousCompileContext.postLinkFns || [];
  var controllers = {};
  var newScopeDirective;
  var newIsolateScopeDirective = previousCompileContext.newIsolateScopeDirective;
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives = previousCompileContext.controllerDirectives;
```

Summary

Directive templates are not a complicated feature to understand or to implement. In fact, in the beginning of the chapter, we already had a functional implementation after just a couple of pages.

However, when we started to build support for the asynchronous loading of templates, things got a lot more complicated. We needed to build a full pause/resume mechanism for compilation and linking, and in the process ended up shifting a lot of state around between functions. It is sometimes surprising just how complicated it can be to implement seemingly simple features!

In this chapter you have learned:

- That when a directive has a `template` attribute, its contents are used to populate the inner HTML of the element.
- That when a template is used, it replaces any existing contents an element may have had.
- How a template's contents also get compiled and linked.
- That only one template directive may be used for each element.
- How you can also use a function as the value of `template` or `templateUrl`, to support dynamically forming the template or its URL.
- That when isolate scopes are used together with templates, the contents of the template are linked with the isolate scope.
- How the compilation process is paused and later resumed when a template is asynchronously loaded with `templateUrl`.
- How the linking process may be suspended and later resumed if the public link function is called while there are templates loading.

Chapter 20

Directive Transclusion

We've seen how to make directives that have their own templates, making them act a bit like self-contained "components". We've also seen how we can customize the behavior of these components by passing arguments to them, using scope attributes or isolate scopes and HTML attributes.

Sometimes it would be additionally useful to pass in whole HTML structures to a directive. A classic example of this is a "tab bar" component, that renders itself as a collection of tabs, but lets the user of the component provide the actual content that goes inside the tabs.

Something like this could conceivably already be done with our current directive implementation: You could cram the tab HTML content into attributes, to be passed into isolate scopes, for example. Or you could resort to a custom approach based on manual DOM manipulation. But these kinds of solutions are not optimal for components like our "tab bar" example. This is especially because with Angular you don't only care about DOM structures, but also the Scope hierarchies that data is passed around in. If there are directives used in the DOM, we want those directives to always be linked to the correct scopes.

This is where *transclusion* comes in: Transclusion allows passing a DOM structure to a directive, and lets the directive decide where and how to internally use it. Not only this, but transclusion also sets up a scope structure that makes this kind of DOM moving easier: Although the DOM you pass in will be used *inside* another directive's template, its scope will still be as if it was used where you wrote it: *Outside* the directive's template.

> The word `transclusion` is a bit esoteric, and often criticised for that. It wasn't invented for Angular though, and does have a history in computer science, which is interesting in itself. The word makes sense if you think of it as *inclusion* of content from once place *across* (trans) templates to another place.

In addition to managing scopes on the application developer's behalf, transclusion can also do *cloning* for the DOM elements being transcluded. This enables the same DOM elements to be *linked several times*, each time with different scope contents. You can, for example, transclude some piece of DOM for each item in a collection, a bit like `ngRepeat` does.

This cloning is so useful that sometimes you want to use it without actually moving anything from one place to another. For this, Angular provides the `transclude: 'element'` configuration option, that enables these cloning and multi-linking features without the actual transclusion feature. This is what some of Angular's core directives like `ng-repeat` are in fact built on.

These are all features we will be building throughout this chapter. The transclusion implementation is baked into the compilation and linking processes in `compile.js`. Throughout the chapter we'll be revisiting many parts of that file. Let's start with the most basic transclusion feature you can think of: Shifting a piece of DOM from one template to another.

Basic Transclusion

The most basic transclusion use case is this: When a transclusion directive is used on an element, take the child nodes of that element and move them to some location inside the directive's template.

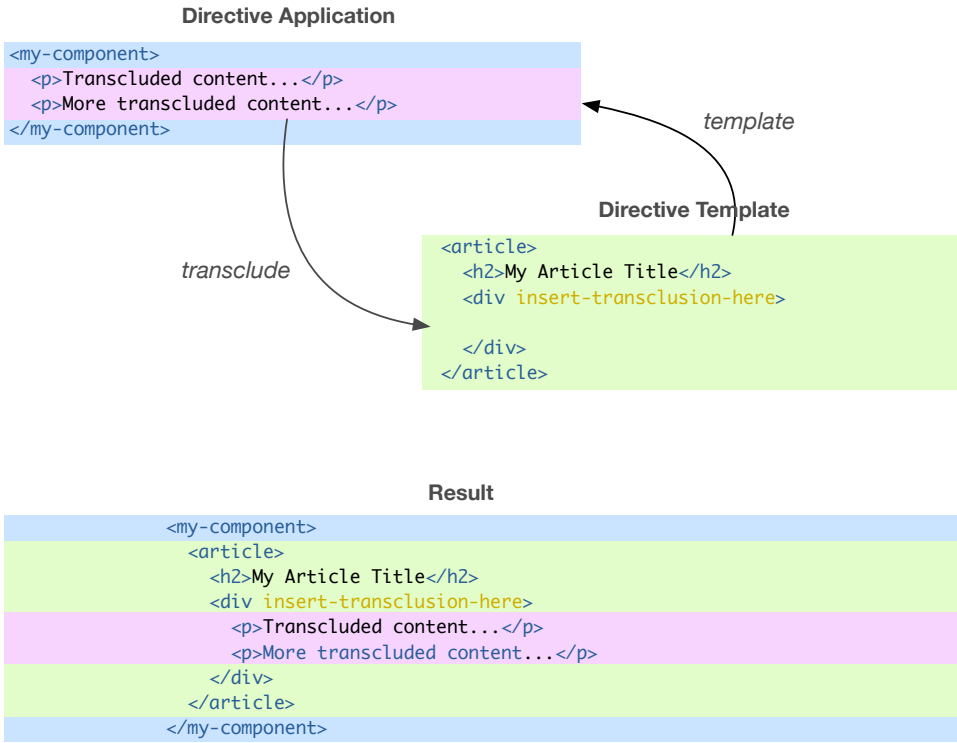

Figure 20.1: The shifting of DOM elements in transclusion

This feature is activated when a directive's definition object contains a `transclude: true`

entry. The very first visible effect of the feature is that the child nodes of the current element disappear from the DOM:

test/compile_spec.js

```
describe('transclude', function() {

  it('removes the children of the element from the DOM', function() {
    var injector = makeInjectorWithDirectives({
      myTranscluder: function() {
        return {transclude: true};
      }
    });
    injector.invoke(function($compile) {
      var el = $('<div my-transcluder><div>Must go</div></div>');

      $compile(el);

      expect(el.is(':empty')).toBe(true);
    });
  });

});
```

This we can achieve easily enough. While compiling directives in `applyDirectivesToNode`, we can check if one has a truthy `transclude` attribute and clear the node contents if so. We'll do this inside the directive loop, in between processing the `controller` and the `template` attributes of each directive:

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  // ...

  _.forEach(directives, function(directive, i) {
    // ...

    if (directive.controller) {
      controllerDirectives = controllerDirectives || {};
      controllerDirectives[directive.name] = directive;
    }
    if (directive.transclude) {
      $compileNode.empty();
    }
    if (directive.template) {
      // ...
    }
    // ..
  }};
  // ...
}
```


Here we simply get rid of the contents of the node when transclusion is configured. You may have guessed that that's not the end of the story. What should we really do with those child nodes?

One thing that should happen is those nodes should still be compiled. Currently they are not being compiled because we remove them, and that happens before `compileNodes` would traverse them. If we assert that they *are* compiled, the test fails:

test/compile_spec.js

```
it('compiles child elements', function() {
  var insideCompileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {transclude: true};
    },
    insideTranscluder: function() {
      return {compile: insideCompileSpy};
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div my-transcluder><div inside-transcluder></div></div>');

    $compile(el);

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

So we should compile the child nodes but still have them removed from the DOM tree we were originally compiling. We can achieve both by separately calling the `compile` service for those detached child nodes. Effectively, the transcluded content is compiled in a separate, independent compilation process:

src/compile.js

```
if (directive.transclude) {
  var $transcludedNodes = $compileNode.clone().contents();
  compile($transcludedNodes);
  $compileNode.empty();
}
```

Notice that we also clone the compile node before getting its contents. This is so that after we've emptied the node, we'll still have another clone that contains the transcluded content.

Now we're compiling the transcluded nodes, but we're still throwing them away after that. They're not used for anything and will never get attached to the page. What we want to do is enable *transcluding* those elements somewhere. But where, and how?

The question of *where* these elements get transcluded is one we cannot solve in the framework. The application developer should decide that. What we can do is *make those elements available* to the application developer, so that they can attach them where they want.

This we'll do by passing in a new, fifth argument to the transclusion directive's link function. That argument is the *transclusion function*. It's a function that gives the directive author access to the transcluded content. Here's our first test directive that actually does transclusion: It uses the transclusion function to get the transcluded content, and attaches it into its template.

test/compile_spec.js

```
it('makes contents available to directive link function', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        template: '<div in-template></div>',
        link: function(scope, element, attrs, ctrl, transclude) {
          element.find('[in-template]').append(transclude());
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div in-transcluder></div></div>');

    $compile(el)($rootScope);
    expect(el.find('> [in-template] > [in-transcluder]').length).toBe(1);
  });
});
```

So there should be a fifth argument to directive link functions, available for transclusion directives: The transclusion function. Let's see how to create that function and pass it in.

In `applyDirectivesToNode` we're going to introduce a new tracking variable called `childTranscludeFn`:

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = previousCompileContext.preLinkFns || [];
```



```
  var postLinkFns = previousCompileContext.postLinkFns || [];
  var controllers = {};
  var newScopeDirective;
  var newIsolateScopeDirective = previousCompileContext.newIsolateScopeDirective;
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives = previousCompileContext.controllerDirectives;
  var childTranscludeFn;

  // ...
}
```

In this variable we'll store the return value of the `compile` call we make for the transcluded content. That means it'll be the public link function for that content:

src/compile.js

```
if (directive.transclude) {
  var $transcludedNodes = $compileNode.clone().contents();
  childTranscludeFn = compile($transcludedNodes);
  $compileNode.empty();
}
```

The simplest thing we can now do to make our test pass is to modify the public link function, so that it *returns the node(s) that it linked*:

src/compile.js

```
return function publicLinkFn(scope) {
  $compileNodes.data('$scope', scope);
  compositeLinkFn(scope, $compileNodes);
  return $compileNodes;
};
```

We do this because now we can simply use the public link function of the transcluded content as the transclusion function given to directives:

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode) {

  // ...

  _.forEach(preLinkFns, function(linkFn) {
    linkFn(
      linkFn.isolateScope ? isolateScope : scope,
      $element,
      attrs,
```

```
      linkFn.require && getControllers(linkFn.require, $element),
      childTranscludeFn
    );
  });
  if (childLinkFn) {
    var scopeToChild = scope;
    if (newIsolateScopeDirective && newIsolateScopeDirective.template) {
      scopeToChild = isolateScope;
    }
    childLinkFn(scopeToChild, linkNode.childNodes);
  }
  _.forEachRight(postLinkFns, function(linkFn) {
    linkFn(
      linkFn.isolateScope ? isolateScope : scope,
      $element,
      attrs,
      linkFn.require && getControllers(linkFn.require, $element),
      childTranscludeFn
    );
  });
}
```

> Note that the transclusion function is passed to all the directives on the node - the one that had `transclude: true` and any other ones.

We've arrived at a key takeaway: At its core, *the transclusion function is really a link function*. Right now, it is the raw *public link function* for the transcluded content. Things aren't going to remain quite that simple though. We've completely ignored scope management for now, for example. But the core idea of transclusion functions really being link functions will remain.

Before we get into scope management, let's add one restriction to how transclusion can be used: Just like with templates, you can only do transclusion once per element. Doing it in two directives would make little sense, since the first one already clears out the node's contents, leaving nothing for the second one. So we explicitly throw an exception when two or more transclusion directives are used:

test/compile_spec.js

```
it('is only allowed once per element', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {transclude: true};
    },
    mySecondTranscluder: function() {
      return {transclude: true};
    }
  });
  injector.invoke(function($compile) {
```



```
    var el = $('<div my-transcluder my-second-transcluder></div>');

    expect(function() {
      $compile(el);
    }).toThrow();
  });
});
```

To track this we'll use another new variable in `applyDirectivesToNode`, which is simply a flag that tracks if we've seen a transclusion directive on this element already:

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = previousCompileContext.preLinkFns || [];
  var postLinkFns = previousCompileContext.postLinkFns || [];
  var controllers = {};
  var newScopeDirective;
  var newIsolateScopeDirective = previousCompileContext.newIsolateScopeDirective;
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives = previousCompileContext.controllerDirectives;
  var childTranscludeFn, hasTranscludeDirective;

  // ...

}
```

We then check this flag when a transclusion directive comes up:

src/compile.js

```
if (directive.transclude) {
  if (hasTranscludeDirective) {
    throw 'Multiple directives asking for transclude';
  }
  hasTranscludeDirective = true;
  var $transcludedNodes = $compileNode.clone().contents();
  childTranscludeFn = compile($transcludedNodes);
  $compileNode.empty();
}
```

Transclusion And Scopes

If and when there are any directives being used inside the transcluded content, they should be linked. We are already doing that, since we use the public link function as the transclusion function. But that linking is done with *no scope*, as we plainly see if we try to use a scope inside transcluded content:

test/compile_spec.js

```
it('makes scope available to link functions inside', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        link: function(scope, element, attrs, ctrl, transclude) {
          element.append(transclude());
        }
      };
    },
    myInnerDirective: function() {
      return {
        link: function(scope, element) {
          element.html(scope.anAttr);
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div my-inner-directive></div></div>');

    $rootScope.anAttr = 'Hello from root';
    $compile(el)($rootScope);
    expect(el.find('> [my-inner-directive]').html()).toBe('Hello from root');
  });
});
```

We can fix this by *pre-binding* the transclusion function to a scope. This is done by actually wrapping that function with another one that calls the original with a scope. This just means the directive author doesn't have to provide the scope - we do it from within the directive system.

src/compile.js

```
function boundTranscludeFn() {
  return childTranscludeFn(scope);
}

_.forEach(preLinkFns, function(linkFn) {
  linkFn(
```



```
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element),
    boundTranscludeFn
  );
});
if (childLinkFn) {
  var scopeToChild = scope;
  if (newIsolateScopeDirective && newIsolateScopeDirective.template) {
    scopeToChild = isolateScope;
  }
  childLinkFn(scopeToChild, linkNode.childNodes);
}
_.forEachRight(postLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element),
    boundTranscludeFn
  );
});
```

> Readers familiar with functional programming may recognize this as partial function application. We could simply use LoDash and do `var boundTranscludeFn = _.partial(childTranscludeFn, scope)`, but since we're going to do more work in the bound transclude function later, we use the manual form.

The scope we are binding the transclusion function to is not quite the correct one yet. The transcluded content should be linked to the scope in which it was defined, whereas right now it's linked to the scope in which it's *used*. For instance, if the transclusion directive produces an inherited scope, the transcluded content should know nothing about it. We see how this is a problem if the transclusion directive shadows an attribute from the parent scope. The transcluded content's scope should still see the parent's value but it doesn't:

test/compile_spec.js

```
it('does not use the inherited scope of the directive', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        scope: true,
        link: function(scope, element, attrs, ctrl, transclude) {
          scope.anAttr = 'Shadowed attribute';
          element.append(transclude());
        }
```

```
      };
    },
    myInnerDirective: function() {
      return {
        link: function(scope, element) {
          element.html(scope.anAttr);
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div my-inner-directive></div></div>');

    $rootScope.anAttr = 'Hello from root';
    $compile(el)($rootScope);
    expect(el.find('> [my-inner-directive]').html()).toBe('Hello from root');
  });
});
```

This poses a problem to our bound transclusion function, because it's created in the node link function where the inherited scope is the only scope we know about. In order to properly decide which scope to use for transclusion, we need to do it in the *composite link function* instead.

But first we need to let the composite link function know when a node has a transclusion directive. We can do this by attaching our two new tracking variables to the node link function as attributes:

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {

  // ...

  nodeLinkFn.terminal = terminal;
  nodeLinkFn.scope = newScopeDirective && newScopeDirective.scope;
  nodeLinkFn.transcludeOnThisElement = hasTranscludeDirective;
  nodeLinkFn.transclude = childTranscludeFn;

  return nodeLinkFn;
}
```

In the composite link function itself we can now set up the bound transclusion function. First, let's change the current code for making an inherited scope for a node so that it doesn't overwrite the parent scope variable but instead uses a separate variable:

src/compile.js

```
_.forEach(linkFns, function(linkFn) {
  var node = stableNodeList[linkFn.idx];
  if (linkFn.nodeLinkFn) {
    var childScope;
    if (linkFn.nodeLinkFn.scope) {
      childScope = scope.$new();
      $(node).data('$scope', childScope);
    } else {
      childScope = scope;
    }
    linkFn.nodeLinkFn(
      linkFn.childLinkFn,
      childScope,
      node,
      boundTranscludeFn
    );
  } else {
    linkFn.childLinkFn(
      scope,
      node.childNodes
    );
  }
});
```

And now, if there was a directive on the node that used transclusion (causing `transcludeOnThisElement` to become `true`), we make the bound transclusion function. It calls the original transclusion function (which is now in the node link function's `transclude` attribute) with the *surrounding* scope. We then pass the bound transclusion function to the node link function as an argument:

src/compile.js

```
_.forEach(linkFns, function(linkFn) {
  var node = stableNodeList[linkFn.idx];
  if (linkFn.nodeLinkFn) {
    var childScope;
    if (linkFn.nodeLinkFn.scope) {
      childScope = scope.$new();
      $(node).data('$scope', childScope);
    } else {
      childScope = scope;
    }

    var boundTranscludeFn;
    if (linkFn.nodeLinkFn.transcludeOnThisElement) {
      boundTranscludeFn = function() {
        return linkFn.nodeLinkFn.transclude(scope);
      };
    }
```



```
    linkFn.nodeLinkFn(
      linkFn.childLinkFn,
      childScope,
      node,
      boundTranscludeFn
    );
  } else {
    linkFn.childLinkFn(
      scope,
      node.childNodes
    );
  }
});
```

We now have a new argument we need the node link function to receive. While adding it, *also remove the `function boundTranscludeFn` statement from the node link function* - we no longer want it since we use the one given as argument instead.

src/compile.js

```
function nodeLinkFn(childLinkFn, scope, linkNode, boundTranscludeFn) {

  // ...

}
```

What we ended up with here is a version of the bound transclusion function that does the binding to the scope *outside* of the transclusion directive. The transcluded content now has access to the scope contents that it needs.

This scope is still not *quite* the one we need though. While it does have exactly the data - and the prototypal inheritance hierarchy - that we want, there's a problem that has to do with scope lifecycle: When the scope of the transclusion directive is destroyed, we'd like all the watches and event listeners from inside the transclusion to be destroyed as well. This does not currently happen, because we use the *surrounding* scope for the transclusion, and that may continue to exist a long time after the transclusion directive is gone.

test/compile_spec.js

```
it('stops watching when transcluding directive is destroyed', function() {
  var watchSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        scope: true,
        link: function(scope, element, attrs, ctrl, transclude) {
```



```
          element.append(transclude());
          scope.$on('destroyNow', function() {
            scope.$destroy();
          });
        }
      };
    },
    myInnerDirective: function() {
      return {
        link: function(scope) {
          scope.$watch(watchSpy);
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div my-inner-directive></div></div>');
    $compile(el)($rootScope);

    $rootScope.$apply();
    expect(watchSpy.calls.count()).toBe(2);

    $rootScope.$apply();
    expect(watchSpy.calls.count()).toBe(3);

    $rootScope.$broadcast('destroyNow');
    $rootScope.$apply();
    expect(watchSpy.calls.count()).toBe(3);
  });
});
```

Here we have a watch expression registered inside the transclusion. We'd expect it to stop being active when the transclusion directive's scope is destroyed, but it doesn't.

This is the point where we see how transclusion scopes are different from other scopes: The parent scope they get their data from should actually be different from the parent scope that determines when they're destroyed. They need *two* parents.

All the way back in Chapter 2 we implemented a feature that enables something like this with Scope objects. When you call `scope.$new()`, you can give an optional second argument: A Scope object that becomes the `$parent` of the new Scope. That Scope will determine when the new Scope gets destroyed, while the JavaScript prototype (and hence all the data) is still set to the Scope you call `$new` on.

Now we can make use of this feature: We should create a special *transclusion scope* that prototypally inherits from the surrounding scope, but whose `$parent` is set to the Scope of the transclusion directive.

That latter Scope is supplied by the node link function, which now creates a *second* layer of binding wrappers to the transclusion function - the *scope-bound transclusion function*:

src/compile.js

```
function scopeBoundTranscludeFn() {
  return boundTranscludeFn(scope);
}

_.forEach(preLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element),
    scopeBoundTranscludeFn
  );
});
if (childLinkFn) {
  var scopeToChild = scope;
  if (newIsolateScopeDirective && newIsolateScopeDirective.template) {
    scopeToChild = isolateScope;
  }
  childLinkFn(scopeToChild, linkNode.childNodes);
}
_.forEachRight(postLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element),
    scopeBoundTranscludeFn
  );
});
```

When you receive a transclusion function in your directive, *this* is actually what you receive: The raw link function of the transcluded content wrapped in two separate binding functions.

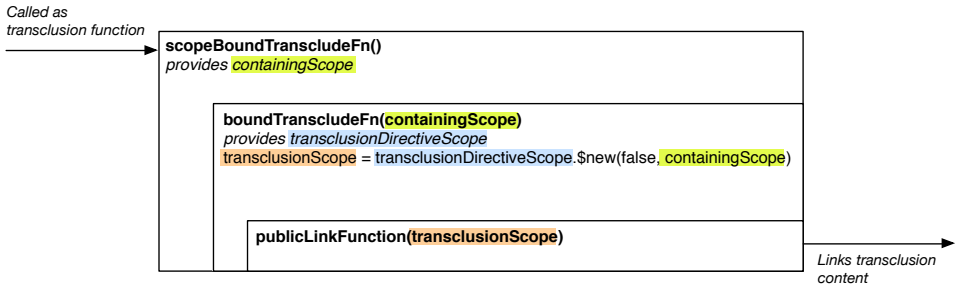

Figure 20.2: The binding wrappers used in transclusion

In the earlier, "inner" bound transclude function, we should now receive this "containing scope":

src/compile.js


```
var boundTranscludeFn;
if (linkFn.nodeLinkFn.transcludeOnThisElement) {
  boundTranscludeFn = function(containingScope) {
    return linkFn.nodeLinkFn.transclude(scope);
  };
}
```

This gives us everything we needs to construct the transclusion scope, which is what we actually link to:

src/compile.js

```
var boundTranscludeFn;
if (linkFn.nodeLinkFn.transcludeOnThisElement) {
  boundTranscludeFn = function(containingScope) {
    var transcludedScope = scope.$new(false, containingScope);
    return linkFn.nodeLinkFn.transclude(transcludedScope);
  };
}
```

So, the prototypal parent of the transcluded scope will be the outer `scope`, while `$parent` will be the inner `containingScope`.

> If the transclusion directive doesn't use scope inheritance or an isolate scope, these two scopes will actually be the same. That's because the composite link function just passes the surrounding scope to the node link function, which will then "provide it back" with the scope-bound transclusion function.

There's one more thing about transclusion scopes before we move on to the next topic: As the directive user, you can actually bypass the transclusion scope creation we just implemented, and just pass your own scope from the directive. You can do it by giving it to the transclusion function when you call it:

test/compile_spec.js

```
it('allows passing another scope to transclusion function', function() {
  var otherLinkSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        scope: {},
        template: '<div></div>',
        link: function(scope, element, attrs, ctrl, transclude) {
          var mySpecialScope = scope.$new(true);
          mySpecialScope.specialAttr = 42;
```



```
          transclude(mySpecialScope);
        }
      };
    },
    myOtherDirective: function() {
      return {link: otherLinkSpy};
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div my-other-directive></div></div>');

    $compile(el)($rootScope);

    var transcludedScope = otherLinkSpy.calls.first().args[0];
    expect(transcludedScope.specialAttr).toBe(42);
  });
});
```

This means that `scopeBoundTranscludeFn` takes an optional argument: The scope to use for transclusion. It just passes it to the inner bound transclusion function, as the first argument:

src/compile.js

```
function scopeBoundTranscludeFn(transcludedScope) {
  return boundTranscludeFn(transcludedScope, scope);
}
```

The inner bound transclude function receives this transclusion scope, or creates it if one wasn't given:

src/compile.js

```
var boundTranscludeFn;
if (linkFn.nodeLinkFn.transcludeOnThisElement) {
  boundTranscludeFn = function(transcludedScope, containingScope) {
    if (!transcludedScope) {
      transcludedScope = scope.$new(false, containingScope);
    }
    return linkFn.nodeLinkFn.transclude(transcludedScope);
  };
}
```

Transclusion from Descendant Nodes

As we have seen, when you have a directive with `transclude: true`, it will get a transclusion function as the fifth argument of its link function(s), which gives it access to the transcluded content. In fact, that fifth argument is available to *all* directives on that element, because we pass it to all pre- and postlink function that we have.

The transclusion function is actually available to even more directives than that: The fifth argument is given whenever there's a transclusion on the current element or any ancestor element. This means that you can actually do the attachment of the transcluded content in some directive *inside* the template of the transclusion directive, as we do here:

test/compile_spec.js

```
it('makes contents available to child elements', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        template: '<div in-template></div>'
      };
    },
    inTemplate: function() {
      return {
        link: function(scope, element, attrs, ctrl, transcludeFn) {
          element.append(transcludeFn());
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div in-transclude></div></div>');

    $compile(el)($rootScope);

    expect(el.find('> [in-template] > [in-transclude]').length).toBe(1);
  });
});
```

This test is currently failing because the `inTemplate` directive is not actually receiving a transclude function, so it's trying to call `undefined`.

In the node link function, as we call the *child* link function, we should pass in the bound transclusion function to make it available to child nodes:

src/compile.js

```
_.forEach(preLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
```

```
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element),
    scopeBoundTranscludeFn
  );
});
if (childLinkFn) {
  var scopeToChild = scope;
  if (newIsolateScopeDirective && newIsolateScopeDirective.template) {
    scopeToChild = isolateScope;
  }
  childLinkFn(scopeToChild, linkNode.childNodes, boundTranscludeFn);
}
_.forEachRight(postLinkFns, function(linkFn) {
  linkFn(
    linkFn.isolateScope ? isolateScope : scope,
    $element,
    attrs,
    linkFn.require && getControllers(linkFn.require, $element),
    scopeBoundTranscludeFn
  );
});
```

Note that we do *not* pass in the scope-bound transclusion function, just the inner bound transclusion function. Child nodes will eventually construct their own scope-bound transclusion functions.

The child link function being called is the composite link function of the child nodes. That function is not yet ready to receive this third argument. Let's add it, and call it the `parentBoundTranscludeFn`, as it is a bound transclusion function from a parent node:

src/compile.js

```
function compositeLinkFn(scope, linkNodes, parentBoundTranscludeFn) { _
  // ...
}
```

Now, inside the link function loop in this function, we'll use this parent-bound transclude function as the bound transclude function - but only if the child node doesn't do any transclusion of its own:

src/compile.js

```
var boundTranscludeFn;
if (linkFn.nodeLinkFn.transcludeOnThisElement) {
  boundTranscludeFn = function(transcludedScope, containingScope) {
    if (!transcludedScope) {
      transcludedScope = scope.$new(false, containingScope);
```

```
    }
    return linkFn.nodeLinkFn.transclude(transcludedScope);
  };
} else if (parentBoundTranscludeFn) {
  boundTranscludeFn = parentBoundTranscludeFn;
}
```

That passes our test: We can now attach transcluded content coming from a transclusion directive on some ancestor node. The lifecycle of the eventual transclusion scope is still based on where the transclusion is actually done.

Not every element in the DOM is going to have directives, and we should be able to pass the transclusion function across them as well. In this test, there's a plain `div` in the transcluding directive's template, causing the `in-template` directive to not receive the transclude function:

test/compile_spec.js

```
it('makes contents available to indirect child elements', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        template: '<div><div in-template></div></div>'
      };
    },
    inTemplate: function() {
      return {
        link: function(scope, element, attrs, ctrl, transcludeFn) {
          element.append(transcludeFn());
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div in-transclude></div></div>');

    $compile(el)($rootScope);

    expect(el.find('> div > [in-template] > [in-transclude]').length).toBe(1);
  });
});
```

We again have the same problem where the directive is trying to invoke `undefined` as the transclusion function.

This we can fix by passing the parent-bound transclusion function onward in the composite link function, in the case where there is no node link function for the current node:

src/compile.js


```
if (linkFn.nodeLinkFn) {
  // ...
} else {
  linkFn.childLinkFn(
    scope,
    node.childNodes,
    parentBoundTranscludeFn
  );
}
```

One additional way of "passing on" the transclusion function is useful in cases where you do something complex in your directive that requires you to run your own, manual compilation and/or linking for child nodes. A "lazily compiling" directive in the vein of `ng-if` would be one example of such a use case.

When you have a directive like that, and you use it in the middle of a transclusion, things may break since the transclusion function doesn't find its way from the parent to the children if you're linking them separately.

You can support transclusion in these kinds of directives too, by passing an additional argument to the public link function of your custom-compiled nodes. That argument is an `options` object, and one of its supported keys is `parentBoundTranscludeFn`. With that you can pass the transclusion function you received across to the other linking process:

test/compile_spec.js

```
it('supports passing transclusion function to public link function', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function($compile) {
      return {
        transclude: true,
        link: function(scope, element, attrs, ctrl, transclude) {
          var customTemplate = $('<div in-custom-template></div>');
          element.append(customTemplate);
          $compile(customTemplate)(scope, {
            parentBoundTranscludeFn: transclude
          });
        }
      };
    },
    inCustomTemplate: function() {
      return {
        link: function(scope, element, attrs, ctrl, transclude) {
          element.append(transclude());
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
```



```
    var el = $('<div my-transcluder><div in-transclude></div></div>');

    $compile(el)($rootScope);

    expect(el.find('> [in-custom-template] > [in-transclude]').length).toBe(1);
  });
});
```

The public link function should take this optional `options` argument, and grab the `parentBoundTranscludeFn` attribute from it when available. It can then pass it to the composite link function, which is already able to take the parent-bound transclude function as its third argument:

src/compile.js

```
return function publicLinkFn(scope, options) {
  options = options || {};
  var parentBoundTranscludeFn = options.parentBoundTranscludeFn;
  $compileNodes.data('$scope', scope);
  compositeLinkFn(scope, $compileNodes, parentBoundTranscludeFn);
  return $compileNodes;
};
```

> `options` is actually the *third* argument to the public link function, not the second. The second argument is something we haven't added yet but will do later in this chapter.

This does introduce a problem related to scope lifecycle: What we are passing over is the *scope-bound transclusion function*, because that's what we have in the link function. The `$parent` of the transclusion scope will incorrectly be bound to the current scope, even though we're not linking the transclusion here. This is plain when we destroy the custom-linked content and expect watches inside the transcluded content to stop firing:

test/compile_spec.js

```
it('destroys scope passed through public link fn at the right time', function() {
  var watchSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myTranscluder: function($compile) {
      return {
        transclude: true,
        link: function(scope, element, attrs, ctrl, transclude) {
          var customTemplate = $('<div in-custom-template></div>');
          element.append(customTemplate);
          $compile(customTemplate)(scope, {
            parentBoundTranscludeFn: transclude
          });
```



```
        }
      };
    },
    inCustomTemplate: function() {
      return {
        scope: true,
        link: function(scope, element, attrs, ctrl, transclude) {
          element.append(transclude());
          scope.$on('destroyNow', function() {
            scope.$destroy();
          });
        }
      };
    },
    inTransclude: function() {
      return {
        link: function(scope) {
          scope.$watch(watchSpy);
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div in-transclude></div></div>');

    $compile(el)($rootScope);

    $rootScope.$apply();
    expect(watchSpy.calls.count()).toBe(2);

    $rootScope.$apply();
    expect(watchSpy.calls.count()).toBe(3);

    $rootScope.$broadcast('destroyNow');
    $rootScope.$apply();
    expect(watchSpy.calls.count()).toBe(3);
  });
});
```

We need to figure out how to "unbind" the scope-bound transclude function in this case. The trick is to attach an attribute to that function, which points to the function being wrapped:

src/compile.js

```
function scopeBoundTranscludeFn(transcludedScope) {
  return boundTranscludeFn(transcludedScope, scope);
}
scopeBoundTranscludeFn.$$boundTransclude = boundTranscludeFn;
```


In the public link function we can use this attribute to unwrap when appropriate. This causes everything to line up again:

src/compile.js

```
return function publicLinkFn(scope, options) {
  options = options || {};
  var parentBoundTranscludeFn = options.parentBoundTranscludeFn;
  if (parentBoundTranscludeFn && parentBoundTranscludeFn.$$boundTransclude) {
    parentBoundTranscludeFn = parentBoundTranscludeFn.$$boundTransclude;
  }
  $compileNodes.data('$scope', scope);
  compositeLinkFn(scope, $compileNodes, parentBoundTranscludeFn);
  return $compileNodes;
};
```

Transclusion in Controllers

You can also choose to call the transclusion function from within a directive's controller, as an alternative to doing it from the link function. The transclusion function is available in the controller as the injected `$transclude` argument:

test/compile_spec.js

```
it('makes contents available to controller', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        template: '<div in-template></div>',
        controller: function($element, $transclude) {
          $element.find('[in-template]').append($transclude());
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div in-transclude></div></div>');
    $compile(el)($rootScope);

    expect(el.find('> [in-template] > [in-transclude]').length).toBe(1);
  });
});
```


We can simply add the scope-bound transclude function to the controller's locals object before constructing the controller:

src/compile.js

```
if (controllerDirectives) {
  _.forEach(controllerDirectives, function(directive, directiveName) {
    var locals = {
      $scope: directive === newIsolateScopeDirective ? isolateScope : scope,
      $element: $element,
      $transclude: scopeBoundTranscludeFn,
      $attrs: attrs
    };
    var controllerName = directive.controller;
    if (controllerName === '@') {
      controllerName = attrs[directive.name];
    }
    var controller =
      $controller(controllerName, locals, true, directive.controllerAs);
    controllers[directive.name] = controller;
    $element.data('$' + directive.name + 'Controller', controller.instance);
  });
}
```

This means the fifth argument to a directive's link function and the `$transclude` argument to a directive's controller both give you exactly the same thing: The scope-bound transclude function.

The Clone Attach Function

We've seen how the transclusion function you get in your directive's link function (or controller) generally works: You call it, and it returns you a reference to the transcluded DOM, which you can then attach somewhere. Internally it links the transcluded DOM to a transclusion scope, whose parents are based on where the transclusion was defined and where you're calling the transclusion function from. You can also optionally give the transclusion function a scope as an argument, in which case it'll use that scope instead of making a transclusion scope.

There's one more central aspect of the transclusion function that we need to cover. That is supplying it with a *clone attach function*.

A clone attach function is a function you can supply whenever you are linking a piece of DOM. When you supply one, Angular will not actually link the original DOM that was compiled. Instead it will first *make a clone* of that DOM and then link that clone. It will also *call* your clone attach function during the compilation, giving it the cloned DOM as well as the scope used for linking. You're expected to attach the clone somewhere at that point - hence the name "clone attach function".

So there are really two, related but separate, purposes for the clone attach function:

1. As a side effect, using one causes a clone of the compiled DOM to be created and linked.
2. It acts as a callback to the linking process - it gets invoked at a specific point in time after the DOM has been cloned.

We'll soon see how both of these can be useful.

At its core, the clone attach function doesn't necessarily have anything to do with transclusion at all. It is part of the public linking API and can be used without transclusion. We just cover it in this chapter since the two so often go together.

When you give a clone attach function to the public link function, it will get called during linking:

test/compile_spec.js

```
describe('clone attach function', function() {

  it('can be passed to public link fn', function() {
    var injector = makeInjectorWithDirectives({});
    injector.invoke(function($compile, $rootScope) {
      var el = $('<div>Hello</div>');
      var myScope = $rootScope.$new();
      var gotEl, gotScope;

      $compile(el)(myScope, function cloneAttachFn(el, scope) {
        gotEl = el;
        gotScope = scope;
      });

      expect(gotEl[0].isEqualNode(el[0])).toBe(true);
      expect(gotScope).toBe(myScope);
    });
  });

});
```

We can make this initial test pass by just calling the supplied clone attach function from the public link function. We do this just before actually linking the DOM:

src/compile.js

```
return function publicLinkFn(scope, cloneAttachFn, options) {
  options = options || {};
  var parentBoundTranscludeFn = options.parentBoundTranscludeFn;
  if (parentBoundTranscludeFn && parentBoundTranscludeFn.$$boundTransclude) {
    parentBoundTranscludeFn = parentBoundTranscludeFn.$$boundTransclude;
  }
  $compileNodes.data('$scope', scope);
  if (cloneAttachFn) {
    cloneAttachFn($compileNodes, scope);
```



```
  }
  compositeLinkFn(scope, $compileNodes, parentBoundTranscludeFn);
  return $compileNodes;
};
```

Since we added the clone attach function as the *second* argument, that pushes *options* to the third position. We need to modify the earlier test cases `"supports passing transclusion function to public link function"` and `"destroys scope passed through public link fn at the right time"` to support this. There, we can just pass `undefined` as the clone attach function:

```
$compile(customTemplate)(scope, undefined, {
  parentBoundTranscludeFn: transclude
});
```

As discussed, this is no ordinary callback function though, since it actually causes the DOM to be a *clone* of the original:

test/compile_spec.js

```
it('causes compiled elements to be cloned', function() {
  var injector = makeInjectorWithDirectives({});
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div>Hello</div>');
    var myScope = $rootScope.$new();
    var gotClonedEl;

    $compile(el)(myScope, function(clonedEl) {
      gotClonedEl = clonedEl;
    });

    expect(gotClonedEl[0].isEqualNode(el[0])).toBe(true);
    expect(gotClonedEl[0]).not.toBe(el[0]);
  });
});
```

This cloned DOM isn't just used for the clone attach function, but it is also the version of the DOM that gets linked. The original DOM, on the other hand, will *not* get linked. So the element received by a directive's link function will in this case be different from the one received by its compile function:

test/compile_spec.js

```
it('causes cloned DOM to be linked', function() {
  var gotCompileEl, gotLinkEl;
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
```



```
        compile: function(compileEl) {
          gotCompileEl = compileEl;
          return function link(scope, linkEl) {
            gotLinkEl = linkEl;
          };
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    var myScope = $rootScope.$new();

    $compile(el)(myScope, function() {});

    expect(gotCompileEl[0]).not.toBe(gotLinkEl[0]);
  });
});
```

So, if a clone attach function is given, we should make a clone the compiled nodes, and then give it to the clone attach function as well as to the composite link function, instead of the original compiled nodes.

src/compile.js

```
return function publicLinkFn(scope, cloneAttachFn, options) {
  options = options || {};
  var parentBoundTranscludeFn = options.parentBoundTranscludeFn;
  if (parentBoundTranscludeFn && parentBoundTranscludeFn.$$boundTransclude) {
    parentBoundTranscludeFn = parentBoundTranscludeFn.$$boundTransclude;
  }
  var $linkNodes;
  if (cloneAttachFn) {
    $linkNodes = $compileNodes.clone();
    cloneAttachFn($linkNodes, scope);
  } else {
    $linkNodes = $compileNodes;
  }
  $linkNodes.data('$scope', scope);
  compositeLinkFn(scope, $linkNodes, parentBoundTranscludeFn);
  return $linkNodes;
};
```

This makes our test suite pass. Do note that we also change the `$scope` jQuery data attachment to happen on the cloned nodes, because those are the ones that are actually linked to the scope

Passing a clone attach function causes nodes to be cloned before linking, but this doesn't yet explain why we need a *function* for it though. Wouldn't a "clone" boolean flag do? For

the current implementation, it would, but using a function starts to make more sense when we begin bringing this discussion back to transclusion.

First of all, you can also pass a clone attach function to the transclusion function. When you do that, it gives you an alternative way to obtain the transcluded nodes: They are not only given as the return value of the transclusion function, but also as the first argument to the clone attach function. That means you can do something like this:

test/compile_spec.js

```
it('allows connecting transcluded content', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        template: '<div in-template></div>',
        link: function(scope, element, attrs, ctrl, transcludeFn) {
          var myScope = scope.$new();
          transcludeFn(myScope, function(transclNode) {
            element.find('[in-template]').append(transclNode);
          });
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div in-transclude></div></div>');

    $compile(el)($rootScope);

    expect(el.find('> [in-template] > [in-transclude]').length).toBe(1);
  });
});
```

What the directive is calling is the scope-bound transclusion function, so it should know how to reive the clone attach function. It can just pass it on to the inner bound transclusion function - also as the second argument:

src/compile.js

```
function scopeBoundTranscludeFn(transcludedScope, cloneAttachFn) {
  return boundTranscludeFn(transcludedScope, cloneAttachFn, scope);
}
scopeBoundTranscludeFn.$$boundTransclude = boundTranscludeFn;
```

The inner bound transclude function receives this new second argument, and passes it on to the actual transclude function - which is the public link function that already supports clone attach functions:

src/compile.js


```
var boundTranscludeFn;
if (linkFn.nodeLinkFn.transcludeOnThisElement) {
  boundTranscludeFn = function(transcludedScope, cloneAttachFn, containingScope) {
    if (!transcludedScope) {
      transcludedScope = scope.$new(false, containingScope);
    }
    return linkFn.nodeLinkFn.transclude(transcludedScope, cloneAttachFn);
  };
} else if (parentBoundTranscludeFn) {
  boundTranscludeFn = parentBoundTranscludeFn;
}
```

This still doesn't explain why we need a *function* though. We could have just passed in a boolean flag for this all to work.

There is one reason for the function form already in our implementation, which is related to timing: When you get the return value of the transclusion function, the DOM will already have been linked. But the clone attach function is called *before* linking. This gives you a chance to manipulate the fresh clone of the DOM before it gets linked: Just make your changes from inside the clone attach function.

But the bigger reason for the existence of the clone attach function has to do with scopes, and we will arrive at it in the next couple of pages.

For one thing, you don't *have* to supply your own transclusion scope when you use a clone attach function, like we did in our last test. You can omit it, and just pass the clone attach function as the only argument. (In that case, the default logic for creating the transclusion scope gets used.)

test/compile_spec.js

```
it('can be used as the only transclusion function argument', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        template: '<div in-template></div>',
        link: function(scope, element, attrs, ctrl, transcludeFn) {
          transcludeFn(function(transclNode) {
            element.find('[in-template]').append(transclNode);
          });
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div in-transclusion></div></div>');

    $compile(el)($rootScope);
```

```
    expect(el.find('> [in-template] > [in-transclusion]').length).toBe(1);
  });
});
```

This if the first time we see the transclusion function being used in the way it most often gets used in the wild: Passing the clone attach function as the only argument.

What this means is the transclusion function can actually be called in three different ways:

1. With a transclusion scope and a clone attach function
2. With just a transclusion scope
3. With just a clone attach function

This means that we need to check the "type" of the first argument passed to that function. If it doesn't look like a scope object, we assume it'll be the clone attach function (or just `undefined`):

src/compile.js

```
function scopeBoundTranscludeFn(transcludedScope, cloneAttachFn) {
  if (!transcludedScope || !transcludedScope.$watch ||
      !transcludedScope.$evalAsync) {
    cloneAttachFn = transcludedScope;
    transcludedScope = undefined;
  }
  return boundTranscludeFn(transcludedScope, cloneAttachFn, scope);
}
scopeBoundTranscludeFn.$$boundTransclude = boundTranscludeFn;
```

Here's the biggest reason the clone attach function is a function. When you *don't* supply your own scope, and a default transclusion scope is used, the clone attach function is the *only* way you can gain access to that scope from your transclusion directive. And you do often need access to it: If you remove the transcluded DOM before your transclusion directive itself gets removed, as the documentation states, it is your responsibility to destroy the transclusion scope, and you can only do that when you have access to it.

Furthermore, the clone attach function lets you put data on the transclusion scope *before* the transcluded content gets linked. So you can essentially pass additional data to the transcluded content via the scope, as the following test case shows. It passes right away as we've already implemented everything it needs, but we include it for the purpose of illustrating the point:

test/compile_spec.js

```
it('allows passing data to transclusion', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        template: '<div in-template></div>',
        link: function(scope, element, attrs, ctrl, transcludeFn) {
          transcludeFn(function(transclNode, transclScope) {
            transclScope.dataFromTranscluder = 'Hello from transcluder';
            element.find('[in-template]').append(transclNode);
          });
        }
      };
    },
    myOtherDirective: function() {
      return {
        link: function(scope, element) {
          element.html(scope.dataFromTranscluder);
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div my-other-directive></div></div>');

    $compile(el)($rootScope);

    expect(el.find('> [in-template] > [my-other-directive]').html())
      .toEqual('Hello from transcluder');
  });
});
```

Transclusion with Template URLs

In the previous chapter we put a whole lot of effort into the pause-resume mechanism that makes the asynchronous loading of templates possible. How does this work with transclusion?

The answer is that currently it doesn't. The bound transclusion function that node link functions now receive isn't supported by the delayed node link function that takes control when `templateUrl`s are used. We should fix that, since using `templateUrl`s should have no bearing on whether transclusion works or not.

Let's first think about the case where a `templateUrl` is used but the template still happens to arrive before linking occurs. Add the following test block in the "templateUrl" section of `compile_spec.js`:

test/compile_spec.js

```
describe('with transclusion', function() {

  it('works when template arrives first', function() {
    var injector = makeInjectorWithDirectives({
      myTranscluder: function() {
        return {
          transclude: true,
          templateUrl: 'my_template.html',
          link: function(scope, element, attrs, ctrl, transclude) {
            element.find('[in-template]').append(transclude());
          }
        };
      }
    });
    injector.invoke(function($compile, $rootScope) {
      var el = $('<div my-transcluder><div in-transclude></div></div>');

      var linkFunction = $compile(el);
      $rootScope.$apply();
      requests[0].respond(200, {}, '<div in-template></div>'); // respond first
      linkFunction($rootScope); // then link

      expect(el.find('> [in-template] > [in-transclude]').length).toBe(1);
    });
  });

});
```

The test fails, as we expected. What we need to do is have the delayed node link function accept the bound transclusion function, which is being given to it. If the template has alreary arrived and the `linkQueue` is no longer there, we can just pass it on to the regular node link function:

src/compile.js

```
return function delayedNodeLinkFn(
    _ignoreChildLinkFn, scope, linkNode, boundTranscludeFn) {
  if (linkQueue) {
    linkQueue.push({scope: scope, linkNode: linkNode});
  } else {
    afterTemplateNodeLinkFn(
      afterTemplateChildLinkFn, scope, linkNode, boundTranscludeFn);
  }
};
```

This still doesn't fix the test though. What's going on?

The problem is that because this directive has the `transclude` option on it, once we populate its contents with the template and call `applyDirectivesToNode` from `compileTemplateUrl`

again, it will *remove* all the nodes that just came from the template. We don't want to be doing all that again after the template arrives, because we set up the transclusion already in the previous `applyDirectivesToNode` call. So we can just set the `transclude` flag to `null` in our derived synchronous directive so that the transclusion logic doesn't activate for a second time:

src/compile.js

```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {
      templateUrl: null,
      transclude: null
    }
  );
  // ...
}
```

The other, arguably more common order of events with template URLs is when the public link function gets invoked before our template has arrived from the server. We are still not properly handling transclusion in that case:

test/compile_spec.js

```
it('works when template arrives after', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: true,
        templateUrl: 'my_template.html',
        link: function(scope, element, attrs, ctrl, transclude) {
          element.find('[in-template]').append(transclude());
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder><div in-transclude></div></div>');

    var linkFunction = $compile(el);
    $rootScope.$apply();
    linkFunction($rootScope); // link first
    requests[0].respond(200, {}, '<div in-template></div>'); // then respond

    expect(el.find('> [in-template] > [in-transclude]').length).toBe(1);
  });
});
```


This is the case where the link queue is used to store pending linkings. The bound transclude function needs to be added to the link queue so that we don't just drop it on the floor:

src/compile.js

```
return function delayedNodeLinkFn(
    _ignoreChildLinkFn, scope, linkNode, boundTranscludeFn) {
  if (linkQueue) {
    linkQueue.push(
      {scope: scope, linkNode: linkNode, boundTranscludeFn: boundTranscludeFn});
  } else {
    afterTemplateNodeLinkFn(
      afterTemplateChildLinkFn, scope, linkNode, boundTranscludeFn);
  }
};
```

Once the template then arrives, we can grab the stored bound transclude function and hand it over to the node link function:

src/compile.js

```
$http.get(templateUrl).success(function(template) {
  directives.unshift(derivedSyncDirective);
  $compileNode.html(template);
  afterTemplateNodeLinkFn = applyDirectivesToNode(directives, $compileNode, attrs, p
  afterTemplateChildLinkFn = compileNodes($compileNode[0].childNodes);
  _.forEach(linkQueue, function(linkCall) {
    afterTemplateNodeLinkFn(
      afterTemplateChildLinkFn,
      linkCall.scope,
      linkCall.linkNode,
      linkCall.boundTranscludeFn
    );
  });
  linkQueue = null;
});
```

The final point about handling transclusion with asynchronous directives is controlling that transclusion isn't used in two directives on the same element. We already have this check for regular synchronous directives but it doesn't work when `templateUrl` is used. The second directive here should not be compiled:

test/compile_spec.js


```
it('is only allowed once', function() {
  var otherCompileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        priority: 1,
        transclude: true,
        templateUrl: 'my_template.html'
      };
    },
    mySecondTranscluder: function() {
      return {
        priority: 0,
        transclude: true,
        compile: otherCompileSpy
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder my-second-transcluder></div>');

    $compile(el);
    $rootScope.$apply();
    requests[0].respond(200, {}, '<div in-template></div>');

    expect(otherCompileSpy).not.toHaveBeenCalled();
  });
});
```

The `hasTranscludeDirective` tracking variable is another one of those things we should pass on through the *previous compile context* object. We should put it in when we construct the context:

src/compile.js

```
nodeLinkFn = compileTemplateUrl(
  _.drop(directives, i),
  $compileNode,
  attrs,
  {
    templateDirective: templateDirective,
    newIsolateScopeDirective: newIsolateScopeDirective,
    controllerDirectives: controllerDirectives,
    hasTranscludeDirective: hasTranscludeDirective,
    preLinkFns: preLinkFns,
    postLinkFns: postLinkFns
  }
);
```

And then we should unpack it when arriving back at `applyDirectivesToNode`:

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = previousCompileContext.preLinkFns || [];
  var postLinkFns = previousCompileContext.postLinkFns || [];
  var controllers = {};
  var newScopeDirective;
  var newIsolateScopeDirective = previousCompileContext.newIsolateScopeDirective;
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives = previousCompileContext.controllerDirectives;
  var childTranscludeFn;
  var hasTranscludeDirective = previousCompileContext.hasTranscludeDirective;

  // ...
}
```

Transclusion with Multi-Element Directives

A second special case that needs extra care when combined with transclusion is multi-element directives. You could argue it makes little sense to use these two features together: Since multi-element directives start and end in different sibling nodes, exactly what node's children should become the transcluded contents?

The answer isn't obvious, and as it happens, Angular contains no special logic for this - it just does whatever the jQuery/jqLite DOM manipulation functions do by default when applied to multiple elements. Angular does, however, have rudimentary support for using `transclude` and `multiElement` together. We should add a test for it in the `describe('transclude')` block of `compile_spec.js`:

test/compile_spec.js

```
it('can be used with multi-element directives', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function($compile) {
      return {
        transclude: true,
        multiElement: true,
        template: '<div in-template></div>',
        link: function(scope, element, attrs, ctrl, transclude) {
          element.find('[in-template]').append(transclude());
        }
      };
```



```
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $(
      '<div><div my-transcluder-start><div in-transclude></div></div>'+
      '<div my-transcluder-end></div></div>'
    );
    $compile(el)($rootScope);
    expect(el.find('[my-transcluder-start] [in-template] [in-transclude]').length)
      .toBe(1);
  });
});
```

All we really need to do to make this work is ensure the link function wrapper used for grouped elements passes the transclusion function through :

src/compile.js

```
function groupElementsLinkFnWrapper(linkFn, attrStart, attrEnd) {
  return function(scope, element, attrs, ctrl, transclude) {
    var group = groupScan(element[0], attrStart, attrEnd);
    return linkFn(scope, group, attrs, ctrl, transclude);
  };
}
```

The ngTransclude Directive

When you first learned transclusion, it is likely that it was introduced with a reference to the `ng-transclude` directive, which you can use inside the template of your transclusion directive, to mark where the transcluded content should go. Using `ng-transclude` you can forego having to call the transclusion function yourself, or even knowing that one exists. It's kind of a more declarative way to implement common transclusion scenarios.

As it turns out, this directive can be fully implemented using the features we've introduced in this chapter, and we don't actually need much code to do it. But let's begin by pinning down the behavior of this directive with a test suite. The very first thing we need is some setup code into a new file `test/directives/ng_transclude_spec.js`. We'll add the usual Angular setup as well as a helper function that creates a transclusion directive for a given template.

test/directives/ng_transclude_spec.js

```
describe('ngTransclude', function() {

  beforeEach(function() {
    delete window.angular;
    publishExternalAPI();
```



```
  });

  function createInjectorWithTranscluderTemplate(template) {
    return createInjector(['ng', function($compileProvider) {
      $compileProvider.directive('myTranscluder', function() {
        return {
          transclude: true,
          template: template
        };
      });
    }]);
  }

});
```

That lets us get on with the test definitions. Firstly, a transclusion directive whose template contains the `ng-transclude` attribute on some element will attach the transcluded content inside that element:

test/directives/ng_transclude_spec.js

```
it('transcludes the parent directive transclusion', function() {
  var injector = createInjectorWithTranscluderTemplate(
    '<div ng-transclude></div>'
  );
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder>Hello</div>');
    $compile(el)($rootScope);
    expect(el.find('> [ng-transclude]').html()).toEqual('Hello');
  });
});
```

Secondly, any existing contents the element with the `ng-transclude` attribute may have had will be removed:

test/directives/ng_transclude_spec.js

```
it('empties existing contents', function() {
  var injector = createInjectorWithTranscluderTemplate(
    '<div ng-transclude>Existing contents</div>'
  );
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder>Hello</div>');
    $compile(el)($rootScope);
    expect(el.find('> [ng-transclude]').html()).toEqual('Hello');
  });
});
```

In addition to an attribute, you can make `ng-transclude` an element instead:

test/directives/ng_transclude_spec.js

```
it('may be used as element', function() {
  var injector = createInjectorWithTranscluderTemplate(
    '<ng-transclude>Existing contents</ng-transclude>'
  );
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder>Hello</div>');
    $compile(el)($rootScope);
    expect(el.find('> ng-transclude').html()).toEqual('Hello');
  });
});
```

Finally, you can also make `ng-transclude` a CSS class:

test/directives/ng_transclude_spec.js

```
it('may be used as class', function() {
  var injector = createInjectorWithTranscluderTemplate(
    '<div class="ng-transclude">Existing contents</div>'
  );
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-transcluder>Hello</div>');
    $compile(el)($rootScope);
    expect(el.find('> .ng-transclude').html()).toEqual('Hello');
  });
});
```

There's our test suite for `ng-transclude`. Let's go ahead and implement the code to make it pass. We need a directive factory, which we'll put in `src/directives/ng_transclude.js`:

src/directives/ng_transclude.js

```
var ngTranscludeDirective = function() {
  'use strict';

  return {
  };

};
```

We also need to register this directive into the `ng` module so that the compiler finds it when it's applied:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
  ngModule.provider('$httpBackend', $HttpBackendProvider);
  ngModule.provider('$http', $HttpProvider);
  ngModule.provider('$httpParamSerializer', $HttpParamSerializerProvider);
  ngModule.provider('$httpParamSerializerJQLike',
    $HttpParamSerializerJQLikeProvider);
  ngModule.provider('$compile', $CompileProvider);
  ngModule.provider('$controller', $ControllerProvider);
  ngModule.directive('ngController', ngControllerDirective);
  ngModule.directive('ngTransclude', ngTranscludeDirective);
}
```

And finally, let's fill in the details. Literally all this directive needs to do is call the transclusion function, and append the received DOM element to the current element, while also clearing out any existing contents:

src/directives/ng_transclude.js

```
var ngTranscludeDirective = function() {
  'use strict';

  return {
    restrict: 'EAC',
    link: function(scope, element, attrs, ctrl, transclude) {
      transclude(function(clone) {
        element.empty();
        element.append(clone);
      });
    }
  };

};
```

As it turns out, `ng-transclude`, just like `ng-controller`, is actually a very simple directive, although it's a "major" framework feature. Both directives simply make some of the core features of the directive compiler more easily accessible.

Full Element Transclusion

For the remainder of this chapter we'll focus our attention on a slightly different use case for the transclusion features we have built. It will also require us to extend those transclusion features a bit.

Usually, when transclusion is discussed, we're talking about taking the contents of an element and including them inside another element in another template. That's how the feature was introduced in this chapter, and that's what people usually mean when they talk about transclusion.

There is a second kind of "transclusion" supported by the `transclude` configuration option, called full element transclusion. It is enabled by setting the value of `transclude` not to `true` but to the string `'element'`.

Superficially, there is just a minor difference between this kind of transclusion and the regular kind: Full element transclusion takes *the whole element with the transclusion directive itself* as the transclusion content, whereas regular transclusion *only* takes its children.

That is not where the differences end, however. It turns out that `transclude: 'element'` is actually designed for a completely different use case from regular transclusion. It just happens to be enabled by the same configuration option and shares a lot of the same implementation.

The difference between the use cases is this: `transclude: true` is meant to include part of one template in another template. `transclude: element` is meant to keep the element in place, but to provide more control over what happens to it: The element could be added only when some condition holds true, or it could be added at a later time, or it could even be added multiple times.

This is a major building block for directives like `ngIf` and `ngRepeat`, and directives like that are in fact what full element transclusion is designed to support. It provides these features largely by building on the cloning and scope management features of transclusion that we have already implemented.

Let's begin exploring what element transclusion means exactly, by adding our first test for it. When you use a directive with `transclude: 'element'`, the element with that directive actually disappears from the DOM during compilation:

test/compile_spec.js

```
describe('element transclusion', function() {

  it('removes the element from the DOM', function() {
    var injector = makeInjectorWithDirectives({
      myTranscluder: function() {
        return {
          transclude: 'element'
        };
      }
    });
    injector.invoke(function($compile) {
      var el = $('<div><div my-transcluder></div></div>');
```



```
      $compile(el);

      expect(el.is(':empty')).toBe(true);
    });
  });

});
```

That's a slightly strange way to begin, but it is something that really should happen.

Let's introduce element transclusion inside `applyDirectivesToNode`, where we can add an `if-else` construct to separate it from regular transclusion:

src/compile.js

```
if (directive.transclude) {
  if (hasTranscludeDirective) {
    throw 'Multiple directives asking for transclude';
  }
  hasTranscludeDirective = true;
  if (directive.transclude === 'element') {
    $compileNode.remove();
  } else {
    var $transcludedNodes = $compileNode.clone().contents();
    childTranscludeFn = compile($transcludedNodes);
    $compileNode.empty();
  }
}
```

This element does not *just* disappear, however. Instead, an HTML comment gets introduced in its place. The comment contains the directive name, followed by a colon character, followed by *two* space characters:

test/compile_spec.js

```
it('replaces the element with a comment', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: 'element'
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div><div my-transcluder></div></div>');

    $compile(el);

    expect(el.html()).toEqual('<!-- myTranscluder:  -->');
  });
});
```


Notice that we wrap the element with the directive into an outer `<div>` so that we can more easily inspect what happened.

We can create this comment using the standard DOM `document.createComment()` function. We can then *replace* the current node with this new comment node, using the jQuery/jqLite `replaceWith()` function. This function manipulates the DOM so that the comment goes where the original element used to be.

Note that this does *not* replace the internal contents of the `$compileNode` variable itself, though based on the API it may look like that. `$compileNode` will keep holding the original element.

src/compile.js

```
if (directive.transclude) {
  if (hasTranscludeDirective) {
    throw 'Multiple directives asking for transclude';
  }
  hasTranscludeDirective = true;
  if (directive.transclude === 'element') {
    $compileNode.replaceWith(
      $(document.createComment(' ' + directive.name + ':  ')));
  } else {
    var $transcludedNodes = $compileNode.clone().contents();
    childTranscludeFn = compile($transcludedNodes);
    $compileNode.empty();
  }
}
```

One additional piece of information we should add to the generated HTML comment is the directive attribute's value, if there was one on the original element. The comment we generate essentially becomes one that looks like the original directive had been applied as a comment directive:

test/compile_spec.js

```
it('includes directive attribute value in comment', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {transclude: 'element'};
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div><div my-transcluder=42></div></div>');

    $compile(el);

    expect(el.html()).toEqual('<!-- myTranscluder: 42 -->');
  });
});
```


We can just grab the corresponding attribute from the current Attributes object to make this test pass:

src/compile.js

```
if (directive.transclude === 'element') {
  $compileNode.replaceWith($(document.createComment(
    ' ' + directive.name + ': ' +attrs[directive.name] + ' '
  )));
} else {
  // ...
}
```

If we replace the element with a comment node in the DOM, what exactly should we pass to the directive's compile and link functions? We actually pass in the comment node. So when you have a directive with `transclude: 'element'`, peculiarly enough it always gets compiled and linked with a *comment node*:

test/compile_spec.js

```
it('calls directive compile and link with comment', function() {
  var gotCompiledEl, gotLinkedEl;
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: 'element',
        compile: function(compiledEl) {
          gotCompiledEl = compiledEl;
          return function(scope, linkedEl) {
            gotLinkedEl = linkedEl;
          };
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div><div my-transcluder></div></div>');

    $compile(el)($rootScope);

    expect(gotCompiledEl[0].nodeType).toBe(Node.COMMENT_NODE);
    expect(gotLinkedEl[0].nodeType).toBe(Node.COMMENT_NODE);
  });
});
```


The trick we need to apply here is to replace the `$compileNode` variable with one that holds the new HTML comment, since that's what will eventually be given to compile and link functions. So let's reinitialize `$compileNode`, but before that, store its original value (the original element) in another variable:

src/compile.js

```
if (directive.transclude === 'element') {
  var $originalCompileNode = $compileNode;
  $compileNode = $(document.createComment(
    ' ' + directive.name + ': ' + attrs[directive.name] + ' '
  ));
  $originalCompileNode.replaceWith($compileNode);
} else {
  // ...
}
```

So at this point `$originalCompileNode` will contain the original element that had the directive, and `$compileNode` will contain the genrated comment. The original element isn't attached anywhere, and the comment is attached wherever the original element originally was.

Interestingly though, if there are lower priority directives on the same element, they should still get compiled too, even though we've removed the element from the DOM:

test/compile_spec.js

```
it('calls lower priority compile with original', function() {
  var gotCompiledEl;
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        priority: 2,
        transclude: 'element'
      };
    },
    myOtherDirective: function() {
      return {
        priority: 1,
        compile: function(compiledEl) {
          gotCompiledEl = compiledEl;
        }
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $('<div><div my-transcluder my-other-directive></div></div>');

    $compile(el);

    expect(gotCompiledEl[0].nodeType).toBe(Node.ELEMENT_NODE);
  });
});
```


While these directives do get compiled in the current implementation, it happens with the comment node, which is a bit strange since they are not even present on that comment node. The test above fails because it expects the compilation to occur on an element node, not a comment node.

In addition to this, any directives on child elements of the original element should also be compiled:

test/compile_spec.js

```
it('calls compile on child element directives', function() {
  var compileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: 'element'
      };
    },
    myOtherDirective: function() {
      return {
        compile: compileSpy
      };
    }
  });
  injector.invoke(function($compile) {
    var el = $(
      '<div><div my-transcluder><div my-other-directive></div></div></div>');

    $compile(el);

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

How do we fulfill these requirements? What we need to do is split this compilation into two: We need to stop the current compilation on the current directive, so that it is the last thing that will get compiled. Then we'll launch *another* compilation for the element we just replaced.

To stop the current compilation, we can use the terminal priority support we already have, by setting the terminal priority to the current directive's priority:

src/compile.js

```
if (directive.transclude === 'element') {
  var $originalCompileNode = $compileNode;
  $compileNode = $(document.createComment(
    ' ' + directive.name + ': ' + attrs[directive.name] + ' '
```



```
  ));
  $originalCompileNode.replaceWith($compileNode);
  terminalPriority = directive.priority;
} else {
  // ...
}
```

Now the lower-priority directives on the original element won't get compiled, nor will anything in the original element's child nodes. Our latest test cases are still failing, but now failing a bit differently.

Now we'll launch the second compilation on the element we just replaced:

src/compile.js

```
if (directive.transclude === 'element') {
  var $originalCompileNode = $compileNode;
  $compileNode = $(document.createComment(
    ' ' + directive.name + ': ' + attrs[directive.name] + ' '
  ));
  $originalCompileNode.replaceWith($compileNode);
  terminalPriority = directive.priority;
  compile($originalCompileNode);
} else {
  // ...
}
```

Here we have introduced a big problem though, which you'll notice when running the test suite after this change. When we call `compile` on the original node, it'll find our element transclusion directive again and redo this same logic for a second time. It will keep doing this until it blows the stack. The unfortunate end result is that the test runner crashes or at least slows to a crawl.

We need to somehow let that second compilation know that it should only compile directives with *lower priority* than what the transclusion directive had. The higher priority ones were already compiled on the first run.

The first step to doing that is to pass the current directive's priority into the recursive `compile` invocation:

src/compile.js

```
if (directive.transclude === 'element') {
  var $originalCompileNode = $compileNode;
  $compileNode = $(document.createComment(
    ' ' + directive.name + ': ' + attrs[directive.name] + ' '
  ));
  $originalCompileNode.replaceWith($compileNode);
  terminalPriority = directive.priority;
  compile($originalCompileNode, terminalPriority);
} else {
  // ...
}
```


The `compile` function receives this argument as an argument called `maxPriority`. It passes it on to `compileNodes`:

src/compile.js

```
function compile($compileNodes, maxPriority) {
  var compositeLinkFn = compileNodes($compileNodes, maxPriority);

  // ...

}
```

In `compileNodes` we pass the argument on to `collectDirectives`:

src/compile.js

```
function compileNodes($compileNodes, maxPriority) {
  var linkFns = [];
  _.forEach($compileNodes, function(node, i) {
    var attrs = new Attributes($(node));
    var directives = collectDirectives(node, attrs, maxPriority);
    // ...
  });

  // ...
}
```

Note that we do *not* pass the max priority to the recursive `compileNodes` call for the element's children. When used, `maxPriority` only applies to the root element of the compilation, whereas in child elements all directives are compiled regardless of priority.

In `collectDirectives` we pass the argument onward one more time, to each invocation of the `addDirective` function:

src/compile.js

```
function collectDirectives(node, attrs, maxPriority) {
  var directives = [];
  var match;
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName, 'E', maxPriority);
    _.forEach(node.attributes, function(attr) {
      var attrStartName, attrEndName;
      var name = attr.name;
      var normalizedAttrName = directiveNormalize(name.toLowerCase());
      var isNgAttr = /^ngAttr[A-Z]/.test(normalizedAttrName);
```



```
      if (isNgAttr) {
        name = _.kebabCase(
          normalizedAttrName[6].toLowerCase() +
          normalizedAttrName.substring(7)
        );
        normalizedAttrName = directiveNormalize(name.toLowerCase());
      }

      attrs.$attr[normalizedAttrName] = name;

      var directiveNName = normalizedAttrName.replace(/(Start|End)$/, '');
      if (directiveIsMultiElement(directiveNName)) {
        if (/Start$/.test(normalizedAttrName)) {
          attrStartName = name;
          attrEndName = name.substring(0, name.length - 5) + 'end';
          name = name.substring(0, name.length - 6);
        }
      }
      normalizedAttrName = directiveNormalize(name.toLowerCase());
      addDirective(
        directives, normalizedAttrName, 'A', maxPriority,
        attrStartName, attrEndName);
      if (isNgAttr || !attrs.hasOwnProperty(normalizedAttrName)) {
        attrs[normalizedAttrName] = attr.value.trim();
        if (isBooleanAttribute(node, normalizedAttrName)) {
          attrs[normalizedAttrName] = true;
        }
      }

    });
    var className = node.className;
    if (_.isString(className) && !_.isEmpty(className)) {
      while ((match = /([\d\w\-_]+)(?:\:([^;]+))?;?/.exec(className))) {
        var normalizedClassName = directiveNormalize(match[1]);
        if (addDirective(directives, normalizedClassName, 'C', maxPriority)) {
          attrs[normalizedClassName] = match[2] ? match[2].trim() : undefined;
        }
        className = className.substr(match.index + match[0].length);
      }
    }
  } else if (node.nodeType === Node.COMMENT_NODE) {
    match = /^\s*directive\:\s*([\d\w\-_]+)\s*(.*)$/.exec(node.nodeValue);
    if (match) {
      var normalizedName = directiveNormalize(match[1]);
      if (addDirective(directives, normalizedName, 'M', maxPriority)) {
        attrs[normalizedName] = match[2] ? match[2].trim() : undefined;
      }
    }
  }
  directives.sort(byPriority);
  return directives;
```



```
}
```

Finally, in `addDirective` we actually do something about this argument. If given, it means we're only interested in directives with a numerically lower priority:

src/compile.js

```
function addDirective(
    directives, name, mode, maxPriority, attrStartName, attrEndName) {
  var match;
  if (hasDirectives.hasOwnProperty(name)) {
    var foundDirectives = $injector.get(name + 'Directive');
    var applicableDirectives = _.filter(foundDirectives, function(dir) {
      return (maxPriority === undefined || maxPriority > dir.priority) &&
             dir.restrict.indexOf(mode) !== -1;
    });
    // ...
  }
  return match;
}
```

In our current use case, this causes the directive with the element transclusion (and any higher-priority directives), to be ignored in the recursive transclusion compilation.

> Since `maxPriority` is an argument of the public API of the compile service, you can also pass one yourself when you're doing manual compilation, if you ever find reason for it.

So as we've seen, in the case of element transclusion, directives in child nodes get compiled as part of the second, recursive compilation. But there's something fishy going on, which becomes apparent when we check *how many times* the compile function of a child element directive gets called. We would expect it to be one:

test/compile_spec.js

```
it('compiles original element contents once', function() {
  var compileSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {transclude: 'element'};
    },
    myOtherDirective: function() {
      return {
        compile: compileSpy
      };
    }
  });
```



```
  injector.invoke(function($compile) {
    var el = $(
      '<div><div my-transcluder><div my-other-directive></div></div></div>');

    $compile(el);

    expect(compileSpy.calls.count()).toBe(1);
  });
});
```

It is in fact not one but two. Why is that?

This is caused by a subtle bug in the `compileNodes` function, where we loop over each node in the node collection. We're using the loop variable `node` to hold the node we're currently at. When and if that node gets replaced by a comment in `applyDirectivesToNode`, the local variable `node` in the `compileNodes` loop does *not* get replaced, and the children of the original node still get compiled, even though that node is now gone from the DOM.

We can fix this by not using a separate loop variable for the node, but by instead always referring to the node through its index in the node collection. That way, when the node gets replaced, we also start automatically using the replacement node. So we should change the loop from a `_.forEach` loop to a `_.times` loop and use the array index to refer to the current node:

src/compile.js

```
function compileNodes($compileNodes, maxPriority) {
  var linkFns = [];
  _.times($compileNodes.length, function(i) {
    var attrs = new Attributes($($compileNodes[i]));
    var directives = collectDirectives($compileNodes[i], attrs, maxPriority);
    var nodeLinkFn;
    if (directives.length) {
      nodeLinkFn = applyDirectivesToNode(directives, $compileNodes[i], attrs);
    }
    var childLinkFn;
    if ((!nodeLinkFn || !nodeLinkFn.terminal) &&
        $compileNodes[i].childNodes && $compileNodes[i].childNodes.length) {
      childLinkFn = compileNodes($compileNodes[i].childNodes);
    }
    if (nodeLinkFn && nodeLinkFn.scope) {
      attrs.$$element.addClass('ng-scope');
    }
    if (nodeLinkFn || childLinkFn) {
      linkFns.push({
        nodeLinkFn: nodeLinkFn,
        childLinkFn: childLinkFn,
        idx: i
      });
    }
```

```
  });

  // ...
}
```

And now we can finally discuss how all of this makes full element transclusion possible.

The original element that was replaced by the comment should be made available through the transclusion function to the transclusion directive, so that it can link clones of the original when it needs to, and as many times as it needs to. For example, here's a test directive that links and attaches two copies of the original content:

test/compile_spec.js

```
it('makes original element available for transclusion', function() {
  var injector = makeInjectorWithDirectives({
    myDouble: function() {
      return {
        transclude: 'element',
        link: function(scope, el, attrs, ctrl, transclude) {
          transclude(function(clone) {
            el.after(clone);
          });
          transclude(function(clone) {
            el.after(clone);
          });
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div><div my-double>Hello</div>');

    $compile(el)($rootScope);

    expect(el.find('[my-double]').length).toBe(2);
  });
});
```

This is essentially a directive that duplicates an element - complete with all the directives in it. It is easy to see how this could be extended to something like an `ngRepeat` directive, which is precisely what element transclusion is designed for!

There's just one simple piece of the puzzle we're missing before this will work. The return value of the recursive `compile` call should become the transclusion function:

src/compile.js

```
if (directive.transclude === 'element') {
  var $originalCompileNode = $compileNode;
  $compileNode = $(document.createComment(
    ' ' + directive.name + ': ' + attrs[directive.name] + ' '
  ));
  $originalCompileNode.replaceWith($compileNode);
  terminalPriority = directive.priority;
  childTranscludeFn = compile($originalCompileNode, terminalPriority);
} else {
  var $transcludedNodes = $compileNode.clone().contents();
  childTranscludeFn = compile($transcludedNodes);
  $compileNode.empty();
}
```

And that connects the dots!

To recap, the implementation of full element transclusion is pretty much the same as that of regular transclusion:

- In regular transclusion the *child nodes* become the transclusion content, and are compiled separately and made available through the transclusion function.
- In full element transclusion the *element itself* becomes the transclusion content. All *lower priority* directives on the current element as well as all child elements are compiled separately and made available through the transclusion function.

Two different use cases with very similar implementations.

There's one final inconsistency we need to iron out before our element transclusion implementation is complete: While the compile and link functions of the transclusion directive now receive the comment node as the element, they are still able to manipulate the attributes of the *original element* through the Attributes object. That should not be possible, since the original element is now in the domain of the transclusion:

test/compile_spec.js

```
it('sets directive attributes element to comment', function() {
  var injector = makeInjectorWithDirectives({
    myTranscluder: function() {
      return {
        transclude: 'element',
        link: function(scope, element, attrs, ctrl, transclude) {
          attrs.$set('testing', '42');
          element.after(transclude());
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div><div my-transcluder></div></div>');
```



```
    $compile(el)($rootScope);

    expect(el.find('[my-transcluder]').attr('testing')).toBeUndefined();
  });
});
```

We can make things consistent by replacing not only `$compileNode` but the element inside the Attributes object:

src/compile.js

```
if (directive.transclude === 'element') {
  var $originalCompileNode = $compileNode;
  $compileNode = attrs.$$element = $(document.createComment(
    ' ' + directive.name + ': ' + attrs[directive.name] + ' '
  ));
  $originalCompileNode.replaceWith($compileNode);
  terminalPriority = directive.priority;
  childTranscludeFn = compile($originalCompileNode, terminalPriority);
} else {
  // ...
}
```

Requiring Controllers from Transcluded Directives

In the chapter about controllers we also implemented the `require` configuration option, using which you can access some other directive's controller from the current element or an ancestor element. How does this fit in with transclusion, where an element may be shifted from one place to another, causing its DOM ancestry to also change?

With regular transclusion, everything is just fine, as ancestor elements can still be found by walking the DOM. But with full element transclusion, we have an issue. If there's a controller on some directive on the same element where we have an element transclusion directive, and we try to require that controller from inside the transclusion, it does not work:

test/compile_spec.js

```
it('supports requiring controllers', function() {
  var MyController = function() { };
  var gotCtrl;
  var injector = makeInjectorWithDirectives({
    myCtrlDirective: function() {
      return {controller: MyController};
    },
    myTranscluder: function() {
      return {
```



```
        transclude: 'element',
        link: function(scope, el, attrs, ctrl, transclude) {
          el.after(transclude());
        }
      };
    },
    myOtherDirective: function() {
      return {
        require: '^myCtrlDirective',
        link: function(scope, el, attrs, ctrl, transclude) {
          gotCtrl = ctrl;
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $(
      '<div><div my-ctrl-directive my-transcluder><div my-other-directive></div></di

    $compile(el)($rootScope);

    expect(gotCtrl).toBeDefined();
    expect(gotCtrl instanceof MyController).toBe(true);
  });
});
```

The reason for this is that the element with the controller now becomes an HTML comment, which does not support jQuery data, and is also not in the transcluded content's ancestry (they're siblings instead). So our normal require implementation does not cut it.

What we need to do is explicitly give references to the controllers to the element that eventually gets transcluded. We also need to do this in JavaScript, since jQuery data isn't available to us.

Firstly, let's add a variable to `applyDirectivesToNode` that marks if an element transclusion directive is present on the current node.

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = previousCompileContext.preLinkFns || [];
  var postLinkFns = previousCompileContext.postLinkFns || [];
  var controllers = {};
  var newScopeDirective;
  var newIsolateScopeDirective = previousCompileContext.newIsolateScopeDirective;
```



```
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives = previousCompileContext.controllerDirectives;
  var childTranscludeFn;
  var hasTranscludeDirective = previousCompileContext.hasTranscludeDirective;
  var hasElementTranscludeDirective;

  // ...
}
```

Let's also set this flag when we see an element transclusion directive:

src/compile.js

```
if (directive.transclude === 'element') {
  hasElementTranscludeDirective = true;
  var $originalCompileNode = $compileNode;
  $compileNode = attrs.$$element = $(document.createComment(
    ' ' + directive.name + ': ' + attrs[directive.name] + ' '
  ));
  $originalCompileNode.replaceWith($compileNode);
  terminalPriority = directive.priority;
  childTranscludeFn = compile($originalCompileNode, terminalPriority);
} else {
  // ...
}
```

In the scope-bound transclusion function, we'll pass a new argument to the inner bound transclusion function (but only if this is a full element transclusion). The argument will be the object of all the controllers on this element:

src/compile.js

```
function scopeBoundTranscludeFn(transcludedScope, cloneAttachFn) {
  var transcludeControllers;
  if (!transcludedScope || !transcludedScope.$watch ||
      !transcludedScope.$evalAsync) {
    cloneAttachFn = transcludedScope;
    transcludedScope = undefined;
  }
  if (hasElementTranscludeDirective) {
    transcludeControllers = controllers;
  }
  return boundTranscludeFn(
    transcludedScope, cloneAttachFn, transcludeControllers, scope);
}
scopeBoundTranscludeFn.$$boundTransclude = boundTranscludeFn;
```

In the bound transclusion function, we should now receive this argument and pass it on to the original transclusion function (the public link function of the transcluded content), as part of the `options` object:

src/compile.js

```
var boundTranscludeFn;
if (linkFn.nodeLinkFn.transcludeOnThisElement) {
  boundTranscludeFn = function(
      transcludedScope, cloneAttachFn, transcludeControllers, containingScope) {
    if (!transcludedScope) {
      transcludedScope = scope.$new(false, containingScope);
    }
    return linkFn.nodeLinkFn.transclude(transcludedScope, cloneAttachFn, {
      transcludeControllers: transcludeControllers
    });
  };
} else if (parentBoundTranscludeFn) {
  boundTranscludeFn = parentBoundTranscludeFn;
}
```

Finally, in the public link function we'll grab any transcluded controllers from the options object, and attach them as jQuery data to the transcluded node. This means that references to the controllers are made discoverable by directives inside this transcluded clone, using the normal `require` discovery mechanism:

src/compile.js

```
return function publicLinkFn(scope, cloneAttachFn, options) {
  options = options || {};
  var parentBoundTranscludeFn = options.parentBoundTranscludeFn;
  var transcludeControllers = options.transcludeControllers;
  if (parentBoundTranscludeFn && parentBoundTranscludeFn.$$boundTransclude) {
    parentBoundTranscludeFn = parentBoundTranscludeFn.$$boundTransclude;
  }
  var $linkNodes;
  if (cloneAttachFn) {
    $linkNodes = $compileNodes.clone();
    cloneAttachFn($linkNodes, scope);
  } else {
    $linkNodes = $compileNodes;
  }
  _.forEach(transcludeControllers, function(controller, name) {
    $linkNodes.data('$' + name + 'Controller', controller.instance);
  });
  $linkNodes.data('$scope', scope);
  compositeLinkFn(scope, $linkNodes, parentBoundTranscludeFn);
  return $linkNodes;
};
```


Recall that the controllers object contains *partially constructed* controller functions, which means that to attach the actual controller object to the DOM, we need to use the `instance` attribute.

Summary

Transclusion is a big topic, both in terms of the complexity of the implemention, and because of how it is baked into the $compile service: We've touched a great number of different parts of the `compile.js` code while adding this feature.

What we've managed to do while doing that is introduce two important features to the DOM compiler: The ability to include parts of one template in another (regular transclusion) and the ability to easily control when and how many copies of a given element are linked and attached (full element transclusion).

In this chapter you have learned:

- How transclusion causes the contents of an element to be removed from the DOM, separately compiled, and made available to link and attach later using the transclusion function.
- That the transclusion function is really just a public link function, wrapped inside a couple of other functions.
- That the transclusion function is available to all directives on the element, not just the one that asked for transclusion. Even when there is no transclusion directive on an element, the transclusion function of the nearest transcluding parent is received.
- That only one transclusion directive is allowed per element.
- That by default, transcluded contents are linked using a special transclusion scope, that prototypally inherits from the surrounding scope so that it has the correct data, but whose `$parent` is based on where the transclusion function is called, so that the lifecycle is correct.
- That when a directive author calls the transclusion function, they can choose to provide their own scope in which case the default transclusion scope is not constructed.
- That the public link function has an "options" argument, through which arguments like `parentBoundTranscludeFn` and `transcludeControllers` can be passed. While they are mostly used internally by the transclusion system, they can be used when calling the public link function in other contexts.
- That the transclusion function is available to directive controllers through the `$transclude` injection, and that it is the exact same function given to directive link functions as the fifth argument.
- That the public link function can receive a "clone attach function" that causes a clone of the original DOM to be linked instead of the original DOM itself. This is useful when, for example, you want to link several copies of a DOM that was previously compiled once.
- That you can also pass the clone attach function to the transclude function.
- That when you pass a clone attach function, it will be called with the transcluded DOM and scope. Also, it will be called *before linking occurs* so that you can manipulate the DOM and the scope just before they are linked.

- How transclusion is supported by the asynchronous template loading triggered by templateUrl.
- How transclusion is (kind of) supported by multi-element directives.
- How the `ngTransclude` directive works - by simply making the core transclusion features available through a simple declarative interface.
- How full element transclusion has a similar implementation as regular transclusion, but that it is meant for a completely different purpose.
- That full element transclusion causes the original element to be replaced with a comment, and the original element with its children to be available through the transclusion function.
- That the public compile function takes a `maxPriority` argument, used by full element transclusion, but available for other potential use cases as well.
- How controller requiring is enabled for element transclusion

Chapter 21

Interpolation

Almost every Angular tutorial out there begins with an introduction to data binding, with examples of JavaScript values being bound to HTML using markup like `{{this}}`. While we've fully covered change detection and expressions during the course of this book, we haven't yet talked about this particular manifestation of those features. This type of markup is called *interpolation*, and using it is familiar to every Angular developer.

An *interpolation expression* consists of double curly braces `{{` and `}}` with an Angular expression inside. The expression itself is one of those things we implemented in Part 2 of the book. What interpolation adds to it is a way to easily attach the value of the expression into the DOM, and to automatically update the DOM when the expression's value changes over time.

> The word "interpolation" comes from a feature called string interpolation that exists in many programming languages. It refers to the process of replacing placeholders in strings with actual values. This is essentially what happens in Angular too.

Interpolation builds on the foundations that we have implemented earlier in the book: Watches, expressions, and directives. In this chapter we bring all of these things together to provide this higher-level feature.

The `$interpolate` service

Much of the functionality required for interpolation is driven by a specialized Angular service called `$interpolate`. It also provides a good place for us to begin exploring this feature.

This service is rarely used by application developers directly, because of the way it is integrated with `$compile`. We will see how that integration works in this chapter, but first let's look at some of those lower level things you can do with `$interpolate` alone.

What we need to do first is to bootstrap the `$interpolate` service, with a pattern that's familiar by now. We'll need a test suite, so let's start by adding that. The very first test for `$interpolate` checks that the service does in fact exist:

test/interpolate_spec.js

```
describe('$interpolate', function() {

  beforeEach(function() {
    delete window.angular;
    publishExternalAPI();
  });

  it('exists', function() {
    var injector = createInjector(['ng']);
    expect(injector.has('$interpolate')).toBe(true);
  });

});
```

Like all our other core services, `$interpolate` is created by a Provider. Just like with `$compile`, what the Provider returns is actually just a function. For now, let's return an empty one:

src/interpolate.js

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

function $InterpolateProvider() {

  this.$get = function() {

    function $interpolate() {
    }

    return $interpolate;
  };

}
```

Before the service actually springs to life, we still need to include it into the `ng` module:

src/angular_public.js

```
function publishExternalAPI() {
  'use strict';

  setupModuleLoader(window);

  var ngModule = angular.module('ng', []);
  ngModule.provider('$filter', $FilterProvider);
```



```
  ngModule.provider('$parse', $ParseProvider);
  ngModule.provider('$rootScope', $RootScopeProvider);
  ngModule.provider('$q', $QProvider);
  ngModule.provider('$$q', $$QProvider);
  ngModule.provider('$httpBackend', $HttpBackendProvider);
  ngModule.provider('$http', $HttpProvider);
  ngModule.provider('$httpParamSerializer', $HttpParamSerializerProvider);
  ngModule.provider('$httpParamSerializerJQLike',
    $HttpParamSerializerJQLikeProvider);
  ngModule.provider('$compile', $CompileProvider);
  ngModule.provider('$controller', $ControllerProvider);
  ngModule.provider('$interpolate', $InterpolateProvider);
  ngModule.directive('ngController', ngControllerDirective);
  ngModule.directive('ngTransclude', ngTranscludeDirective);
}
```

Interpolating Strings

With the infrastructure in place, we can start talking about what this `$interpolate` function actually does.

The general contract of this function is that it takes a string, that may or may not include expressions within curly braces `{{ }}`. It processes that string and returns a function. That function can later be called to *evaluate* all the expressions found in the string and to produce the interpolated result. The function is given a *context object* (often a Scope), in which the expressions will be evaluated:

```
var interpolateFn = $interpolate('{{a}} and {{b}}');
interpolateFn({a: 1, b: 2}); // => '1 and 2'
```

It is notable that this is very similar to how `$parse` works: Both `$interpolate` and `$parse` take strings that are first "parsed". The results are functions that can be evaluated in the context of some Scope. This similarity is not surprising, since `$interpolate` is actually a layer on top of `$parse`: Each of the expressions inside curly braces is individually processed with `$parse`.

Before we get there though, the very simplest kind of interpolation for us to start with is one that doesn't actually include any expressions. When given a simple string, `$interpolate` should return a function that produces that same string:

test/interpolate_spec.js

```
it('produces an identity function for static content', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('hello');
  expect(interp instanceof Function).toBe(true);
  expect(interp()).toEqual('hello');
});
```


The simplest way to get this test passing is to return a function that returns the original argument:

src/interpolate.js

```
this.$get = function() {

  function $interpolate(text) {

    return function interpolationFn() {
      return text;
    };

  }

  return $interpolate;
};
```

Now we can start making things more interesting by actually including an interpolated expression. We should get back a function that evaluates it:

test/interpolate_spec.js

```
it('evaluates a single expression', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('{{anAttr}}');
  expect(interp({anAttr: '42'})).toEqual('42');
});
```

Inside `$interpolate` we can now check if the given text contains the markers `{{` and `}}`. If it does, we grab the substring in between into a variable called `exp`:

src/interpolate.js

```
function $interpolate(text) {
  var startIndex = text.indexOf('{{');
  var endIndex = text.indexOf('}}');
  var exp;
  if (startIndex !== -1 && endIndex !== -1) {
    exp = text.substring(startIndex + 2, endIndex);
  }

  return function interpolationFn() {
    return text;
  };

}
```


What should we do with that substring? Well, it's going to be an Angular expression, so we should parse it! For that we need the `$parse` service, which we inject and use to obtain an expression function:

src/interpolate.js

```
this.$get = ['$parse', function($parse) {

  function $interpolate(text) {
    var startIndex = text.indexOf('{{');
    var endIndex = text.indexOf('}}');
    var exp, expFn;
    if (startIndex !== -1 && endIndex !== -1) {
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
    }

    return function interpolationFn() {
      return text;
    };

  }

  return $interpolate;
}];
```

At evaluation time we can now check whether we have an expression function or not. If we do, we should evaluate it, within the context that we were given. If there was no expression in the text, we still just return the text itself:

src/interpolate.js

```
return function interpolationFn(context) {
  if (expFn) {
    return expFn(context);
  } else {
    return text;
  }
};
```

Now we're getting somewhere. But our implementation is still highly limited, as it can only understand a single expression or static text. Interpolation strings may contain several expressions with static parts in between, so we should support that.

test/interpolate_spec.js


```
it('evaluates many expressions', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('First {{anAttr}}, then {{anotherAttr}}!');
  expect(interp({anAttr: '42', anotherAttr: '43'})).toEqual('First 42, then 43!');
});
```

This means we need to loop over the text and gather any expressions we find while doing it. We can make a `while` loop that keeps going until we reach the end of the string. At each turn of the loop, we find the next expression starting from the current index, or break the loop if there are no more expressions to be found:

src/interpolate.js

```
function $interpolate(text) {
  var index = 0;
  var startIndex, endIndex, exp, expFn;
  while (index < text.length) {
    startIndex = text.indexOf('{{', index);
    endIndex = text.indexOf('}}', index);
    if (startIndex !== -1 && endIndex !== -1) {
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
      index = endIndex + 2;
    } else {
      break;
    }
  }

  return function interpolationFn(context) {
    if (expFn) {
      return expFn(context);
    } else {
      return text;
    }
  };

}
```

This loop finds all the expressions, but we're not collecting them yet. For that we can add an array to put all the parts in - called `parts`. At each turn of the loop, we append both the static text preceding the expression we found, and then the expression function. If no expression is found, we just append all the remaining text:

src/interpolate.js

```
function $interpolate(text) {
  var index = 0;
  var parts = [];
  var startIndex, endIndex, exp, expFn;
  while (index < text.length) {
    startIndex = text.indexOf('{{', index);
    endIndex = text.indexOf('}}', index);
    if (startIndex !== -1 && endIndex !== -1) {
      if (startIndex !== index) {
        parts.push(text.substring(index, startIndex));
      }
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
      parts.push(expFn);
      index = endIndex + 2;
    } else {
      parts.push(text.substring(index));
      break;
    }
  }

  return function interpolationFn(context) {
    if (expFn) {
      return expFn(context);
    } else {
      return text;
    }
  };

}
```

This gives us an array of all the parts of the interpolation string. Some of the items will be static strings, and some will be expression functions. What we can then do is go over the array at evaluation time to produce the result.

We'll reduce the array into a string, at each step checking if the current part is a function or a string. Functions are evaluated in the given context, while strings are just concatenated as-is:

src/interpolate.js

```
function $interpolate(text) {
  var index = 0;
  var parts = [];
  var startIndex, endIndex, exp, expFn;
  while (index < text.length) {
    startIndex = text.indexOf('{{', index);
    endIndex = text.indexOf('}}', index);
    if (startIndex !== -1 && endIndex !== -1) {
      if (startIndex !== index) {
```

```
        parts.push(text.substring(index, startIndex));
      }
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
      parts.push(expFn);
      index = endIndex + 2;
    } else {
      parts.push(text.substring(index));
      break;
    }
  }

  return function interpolationFn(context) {
    return _.reduce(parts, function(result, part) {
      if (_.isFunction(part)) {
        return result + part(context);
      } else {
        return result + part;
      }
    }, '');
  };

}
```

There we have a simple interpolation evaluator!

There's still one problem though. If you specify an ill-defined expression where the end marker precedes the start marker, things don't end up as you'd expect:

test/interpolate_spec.js

```
it('passes through ill-defined interpolations', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('why u no }}work{{');
  expect(interp({})).toEqual('why u no }}work{{');
});
```

We can fix this by always finding the end index of the next expression so that it must come *after* the start expression. Otherwise we shouldn't handle it as an expression at all:

src/interpolate.js

```
while (index < text.length) {
  startIndex = text.indexOf('{{', index);
  if (startIndex !== -1) {
    endIndex = text.indexOf('}}', startIndex + 2);
```



```
  }
  if (startIndex !== -1 && endIndex !== -1) {
    if (startIndex !== index) {
      parts.push(text.substring(index, startIndex));
    }
    exp = text.substring(startIndex + 2, endIndex);
    expFn = $parse(exp);
    parts.push(expFn);
    index = endIndex + 2;
  } else {
    parts.push(text.substring(index));
    break;
  }
}
```

Value Stringification

The results of interpolation are always strings. The same is not true for Angular expressions - they can return anything. This means that we may need to do something about non-string expression values during interpolation, to coerce them into strings.

For instance, values that are `null` or `undefined` should just become empty strings. You never see a `"null"` or `"undefined"` string as a result of a curly brace expression:

test/interpolate_spec.js

```
it('turns nulls into empty strings', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('{{aNull}}');
  expect(interp({aNull: null})).toEqual('');
});

it('turns undefineds into empty strings', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('{{anUndefined}}');
  expect(interp({})).toEqual('');
})
```

To handle these cases, we're going to introduce a function called `stringify` that takes a value and coerces it into a string. We'll call it for each expression value we add to the interpolation result:

src/interpolate.js


```
return function interpolationFn(context) {
  return _.reduce(parts, function(result, part) {
    if (_.isFunction(part)) {
      return result + stringify(part(context));
    } else {
      return result + part;
    }
  }, '');
};
```

The first version of this function returns an empty string for `null` and `undefined`, as discussed. Everything else is just coerced into a string using concatenation:

src/interpolate.js

```
function stringify(value) {
  if (_.isNull(value) || _.isUndefined(value)) {
    return '';
  } else {
    return '' + value;
  }
}
```

Numbers and booleans should also be coerced into strings:

test/interpolate_spec.js

```
it('turns numbers into strings', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('{{aNumber}}');
  expect(interp({aNumber: 42})).toEqual('42');
});

it('turns booleans into strings', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('{{aBoolean}}');
  expect(interp({aBoolean: true})).toEqual('true');
});
```

These test cases are already passing, and that's because we're concatenating them into strings and when that happens JavaScript does what we want for numbers and booleans. We'll keep the test cases anyway, to make sure we don't break things later.

Compound values - arrays and objects - are more challenging. If a compound value is interpolated, we'd like to be able to see its contents. Arrays and objects don't have a useful string representation when just coerced, so what we want to do is turn them into *JSON* strings instead:

test/interpolate_spec.js


```
it('turns arrays into JSON strings', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('{{anArray}}');
  expect(interp({anArray: [1, 2, [3]]})).toEqual('[1,2,[3]]');
});

it('turns objects into JSON strings', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('{{anObject}}');
  expect(interp({anObject: {a: 1, b: '2'}})).toEqual('{"a":1,"b":"2"}');
});
```

This requires us to check if the value is of type `object` (which both objects and arrays are), and to use `JSON.stringify` when that is the case:

src/interpolate.js

```
function stringify(value) {
  if (_.isNull(value) || _.isUndefined(value)) {
    return '';
  } else if (_.isObject(value)) {
    return JSON.stringify(value);
  } else {
    return '' + value;
  }
}
```

Rendering objects and arrays in the UI like this is rarely needed for production applications, but it can be useful during development. You can just interpolate a data structure to the DOM and see what's in it.

Supporting Escaped Interpolation Symbols

If you ever want to output the characters `{{` or `}}` themselves into the UI, without having them intepreted as interpolation markers, you can do so by escaping each character with a backslash: `\{\{` or `\}\}`. What we should do in `$interpolate` when we see these characters is *unescape* them, so that what ends up on the screen is `{{` or `}}`.

> The Angular documentation actually recommends that user-supplied data returned from the server always has these characters escaped. This is essentially an Angular-specific extension to OWASP cross-site scripting prevention guidelines.

Here is this requirement as a test case. Note that to represent a single backslash character in the interpolation string, we need to use two in the test JavaScript code:

test/interpolate_spec.js

```
it('unescapes escaped sequences', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('\\{\\{expr\\}\\} {{expr}} \\{\\{expr\\}\\}');
  expect(interp({expr: 'value'})).toEqual('{{expr}} value {{expr}}');
});
```

Each time we insert static content in the `parts` array, we're going to run it through a helper function called `unescapeText` that processes these escape sequences:

src/interpolate.js

```
while (index < text.length) {
  startIndex = text.indexOf('{{', index);
  if (startIndex !== -1) {
    endIndex = text.indexOf('}}', startIndex + 2);
  }
  if (startIndex !== -1 && endIndex !== -1) {
    if (startIndex !== index) {
      parts.push(unescapeText(text.substring(index, startIndex)));
    }
    exp = text.substring(startIndex + 2, endIndex);
    expFn = $parse(exp);
    parts.push(expFn);
    index = endIndex + 2;
  } else {
    parts.push(unescapeText(text.substring(index)));
    break;
  }
}
```

This helper function can use a couple of regular expression replacements to turn the escape sequences into their unescaped counterparts:

src/interpolate.js

```
function unescapeText(text) {
  return text.replace(/\\{\\{/g, '{{')
             .replace(/\\}\\}/g, '}}');
}
```


Skipping Interpolation When There Are No Expressions

The way the `$interpolate` function currently works is that it always returns a function, whether there's actually something to interpolate or not. This makes a nice and consistent API.

For performance reasons though, it can be useful to *not* make an interpolation function if there's nothing to interpolate. Interpolation functions may end up being called a *lot* since they often end up in watchers, so it makes sense to think about these kinds of optimizations.

To enable this optimization, you can pass a boolean flag that we'll call `mustHaveExpressions` as the second argument to `$interpolate`. When it's set to `true`, we'll only return a function *if* there were expressions in the string.

test/interpolate_spec.js

```
it('does not return function when flagged and no expressions', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('static content only', true);
  expect(interp).toBeFalsy();
});
```

We should probably still make sure a function *is* returned when there *are* expressions in the string, even when the flag is set:

test/interpolate_spec.js

```
it('returns function when flagged and has expressions', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');

  var interp = $interpolate('has an {{expr}}', true);
  expect(interp).not.toBeFalsy();
});
```

So, `$interpolate` should take this `mustHaveExpressions` flag as an argument. We'll also introduce an internal flag for marking whether any expressions were found in the text. Then we only return the interpolation function if there were expressions or if none were required:

src/interpolate.js


```
function $interpolate(text, mustHaveExpressions) {
  var index = 0;
  var parts = [];
  var hasExpressions = false;
  var startIndex, endIndex, exp, expFn;
  while (index < text.length) {
    startIndex = text.indexOf('{{', index);
    if (startIndex !== -1) {
      endIndex = text.indexOf('}}', startIndex + 2);
    }
    if (startIndex !== -1 && endIndex !== -1) {
      if (startIndex !== index) {
        parts.push(unescapeText(text.substring(index, startIndex)));
      }
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
      parts.push(expFn);
      hasExpressions = true;
      index = endIndex + 2;
    } else {
      parts.push(unescapeText(text.substring(index)));
      break;
    }
  }

  if (hasExpressions || !mustHaveExpressions) {
    return function interpolationFn(context) {
      return _.reduce(parts, function(result, part) {
        if (_.isFunction(part)) {
          return result + stringify(part(context));
        } else {
          return result + part;
        }
      }, '');
    };
  }

}
```

We'll return to the `$interpolate` service later in the chapter to add a few more features, but this is all we need for the time being.

Text Node Interpolation

Now that we have a usable implementation of the interpolation service, we can start talking about how it's integrated to the rest of the Angular framework.

The most important of the use cases for interpolation is embedding expressions in the DOM, either on your host HTML page or in template files. The natural place to process these kinds of interpolations is in `$compile`, because it has to walk over the DOM for the purpose of processing directives anyway. That means we're going to add interpolation support to `compile.js`.

There are two places in HTML where interpolation expressions can be used. You can have expressions embedded as text in the HTML:

```
<div>Your username is {{user.username}}</div>
```

This is called *text node interpolation*, as these expressions will end up in DOM text nodes, as opposed to element or comment nodes.

There is also interpolation inside element attributes:

```
<img src="picture.png" alt="{{altText}}">
```

This is called - unsurprisingly - *attribute interpolation*.

We'll start with text node interpolation, since it is easier to implement.

When there is an interpolated expression in some text node, it is replaced by the expression's value, as evaluated on the scope surrounding the text node. Furthermore, that expression is *watched* so that when its value changes, the text node's content also updates. Here's our first test case for this (in `compile_test.js`):

test/compile_spec.js

```
describe('interpolation', function() {

  it('is done for text nodes', function() {
    var injector = makeInjectorWithDirectives({});
    injector.invoke(function($compile, $rootScope) {
      var el = $('<div>My expression: {{myExpr}}</div>');
      $compile(el)($rootScope);

      $rootScope.$apply();
      expect(el.html()).toEqual('My expression: ');

      $rootScope.myExpr = 'Hello';
      $rootScope.$apply();
      expect(el.html()).toEqual('My expression: Hello');
    });
  });

});
```

The way we're going to do this is that whenever we encounter a text node during compilation, we're going to *generate a directive on the fly* and add it to the directives of that text node. In the generated directive we can then execute our interpolation logic.

We haven't done anything with text nodes before, since the `collectDirectives` function is only interested in the node types `Node.ELEMENT_NODE` and `Node.COMMENT_NODE`. We aren't currently applying any directives to any other node types. Now this is going to change, as we're adding a new branch to that function for text nodes:

src/compile.js

```
function collectDirectives(node, attrs, maxPriority) {
  var directives = [];
  var match;
  if (node.nodeType === Node.ELEMENT_NODE) {
    // ...
  } else if (node.nodeType === Node.COMMENT_NODE) {
    // ...
  } else if (node.nodeType === Node.TEXT_NODE) {

  }
  directives.sort(byPriority);
  return directives;
}
```

In this branch we're going to invoke a new function called `addTextInterpolateDirective` that will generate and attach the new directive for us. It takes the directive collection and the node's text value as arguments:

src/compile.js

```
} else if (node.nodeType === Node.TEXT_NODE) {
  addTextInterpolateDirective(directives, node.nodeValue);
}
```

The first thing this new function (which you can add just below the `addDirective` function) does is call the `$interpolate` service to generate the interpolation function. The input for interpolation is just the node's text value. We'll enable the `mustHaveExpressions` flag so that if there is nothing to interpolate, no function will be returned:

src/compile.js

```
function addTextInterpolateDirective(directives, text) {
  var interpolateFn = $interpolate(text, true);
}
```

This requires us to inject the `$interpolate` service as a new dependency to `$compile`:

src/compile.js


```
this.$get = ['$injector', '$parse', '$controller', '$rootScope',
             '$http', '$interpolate',
    function($injector, $parse, $controller, $rootScope, $http, $interpolate) {
```

If `$interpolate` does return an interpolation function, we're going to generate the directive mentioned earlier, and add it to the `directives` collection:

src/compile.js

```
function addTextInterpolateDirective(directives, text) {
  var interpolateFn = $interpolate(text, true);
  if (interpolateFn) {
    directives.push({
      priority: 0,
      compile: function() {
        return function link(scope, element) {

        };
      }
    });
  }
}
```

Since we're creating this directive internally, it doesn't go through normal directive registration where default values for `priority` and `compile` would be populated. This means we have to define them here. The priority isn't really significant since there cannot be any other directives on the text node, but we add it for consistency anyway.

Now that we've added this directive, it will get compiled by `applyDirectivesToNode` and linked by the node link function, just like any other directive.

In the link function we'll start watching the interpolated value on the current scope. Conveniently, we can just use the interpolation function as the watch function directly, since it fulfills the watch contract: It takes a Scope as an argument and returns the interpolated value. It is exactly that value that we want to detect changes on. In the listener function we then replace the text node's value with the interpolation result.

src/compile.js

```
function addTextInterpolateDirective(directives, text) {
  var interpolateFn = $interpolate(text, true);
  if (interpolateFn) {
    directives.push({
      priority: 0,
      compile: function() {
        return function link(scope, element) {
          scope.$watch(interpolateFn, function(newValue) {
            element[0].nodeValue = newValue;
          });
```



```
        };
      }
    });
  }
}
```

And that's all we need to make `{{expressions}}` work in text nodes!

Angular does do a couple of additional things to text nodes during this process. They are mostly done to aid development and introspection tools like Batarang, to make information about interpolations available through the DOM. For instance, whenever there is an interpolation in a text node, an `ng-binding` CSS class is added to its parent element:

test/compile_spec.js

```
it('adds binding class to text node parents', function() {
  var injector = makeInjectorWithDirectives({});
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div>My expression: {{myExpr}}</div>');
    $compile(el)($rootScope);

    expect(el.hasClass('ng-binding')).toBe(true);
  });
});
```

Our generated text interpolation directive can simply add this class to the current element's parent during linking:

src/compile.js

```
return function link(scope, element) {
  element.parent().addClass('ng-binding');
  scope.$watch(interpolateFn, function(newValue) {
    element[0].nodeValue = newValue;
  });
};
```

Also added to the parent element is a list of all the actual expressions that are being interpolated. They are attached to a jQuery data attribute called `$binding`. Its value is an array of all expressions in the text node children of the element:

test/compile_spec.js

```
it('adds binding data to text node parents', function() {
  var injector = makeInjectorWithDirectives({});
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div>{{myExpr}} and {{myOtherExpr}}</div>');
    $compile(el)($rootScope);

    expect(el.data('$binding')).toEqual(['myExpr', 'myOtherExpr']);
  });
});
```


We're going to assume that the expressions used in a given interpolation function will be attached to an attribute called `expressions` on the function:

src/compile.js

```
return function link(scope, element) {
  element.parent().addClass('ng-binding')
    .data('$binding', interpolateFn.expressions);

  scope.$watch(interpolateFn, function(newValue) {
    element[0].nodeValue = newValue;
  });
};
```

Since that attribute isn't there yet, we need to add it. Let's first change things in the `$interpolate` function so that the `hasExpressions` boolean flag is replaced by an actual array in which we collect all the expressions:

src/interpolate.js

```
function $interpolate(text, mustHaveExpressions) {
  var index = 0;
  var parts = [];
  var expressions = [];
  var startIndex, endIndex, exp, expFn;
  while (index < text.length) {
    startIndex = text.indexOf('{{', index);
    if (startIndex !== -1) {
      endIndex = text.indexOf('}}', startIndex + 2);
    }
    if (startIndex !== -1 && endIndex !== -1) {
      if (startIndex !== index) {
        parts.push(unescapeText(text.substring(index, startIndex)));
      }
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
      parts.push(expFn);
      expressions.push(exp);
      index = endIndex + 2;
    } else {
      parts.push(unescapeText(text.substring(index)));
      break;
    }
  }

  if (expressions.length || !mustHaveExpressions) {
    return function interpolationFn(context) {
```



```
      return _.reduce(parts, function(result, part) {
        if (_.isFunction(part)) {
          return result + stringify(part(context));
        } else {
          return result + part;
        }
      }, '');
    };
  }

}
```

Now to fulfill the new requirement we have in `$compile`, we can just attach this `expressions` array to the interpolation function that we return. We'll use lodash _.extend to attach it:

src/interpolate.js

```
return _.extend(function interpolationFn(context) {
  return _.reduce(parts, function(result, part) {
    if (_.isFunction(part)) {
      return result + stringify(part(context));
    } else {
      return result + part;
    }
  }, '');
}, {
  expressions: expressions
});
```

We've still got a slight problem with the `$binding` data attribute, which we can reveal by using two text nodes separated by an element node under the same parent:

test/compile_spec.js

```
it('adds binding data to parent from multiple text nodes', function() {
  var injector = makeInjectorWithDirectives({});
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div>{{myExpr}} <span>and</span> {{myOtherExpr}}</div>');
    $compile(el)($rootScope);

    expect(el.data('$binding')).toEqual(['myExpr', 'myOtherExpr']);
  });
});
```

We have two separate text nodes here, both of which have the same parent. We need the parent node's `$binding` data to include all expressions from all text node children, but as the test shows, it currently only inludes them from the *last* one. The reason is that we're replacing the `$binding` data each time we link one of the text interpolation directives. We have to assume the data attribute may already exist and keep the previous contents:

src/compile.js


```
return function link(scope, element) {
  var bindings = element.parent().data('$binding') || [];
  bindings = bindings.concat(interpolateFn.expressions);
  element.parent().data('$binding', bindings);
  element.parent().addClass('ng-binding');

  scope.$watch(interpolateFn, function(newValue) {
    element[0].nodeValue = newValue;
  });
};
```

> Just like the `ng-scope` class and `$scope` data attribute we implemented earlier, in the original AngularJS the `ng-binding` class and `$binding` data are only attached when the `debugInfoEnabled` flag hasn't been set to `false`

Attribute Interpolation

The second kind of interpolation that can be done in the DOM is inside attributes. While it is basically very similar to interpolation in text nodes, there are a few complications involved since attributes may interact with other directives that are present on the element. We'll need to make sure our implementation interoperates well with those other directives.

The basic behavior of attribute interpolation is still completely the same as with text nodes. An interpolation expression in an attribute is replaced during linking and watched for changes:

test/compile_spec.js

```
it('is done for attributes', function() {
  var injector = makeInjectorWithDirectives({});
  injector.invoke(function($compile, $rootScope) {
    var el = $('<img alt="{{myAltText}}">');
    $compile(el)($rootScope);

    $rootScope.$apply();
    expect(el.attr('alt')).toEqual('');

    $rootScope.myAltText = 'My favourite photo';
    $rootScope.$apply();
    expect(el.attr('alt')).toEqual('My favourite photo');
  });
});
```


The implementation is also very similar to text node interpolation: We generate a directive on the fly. This is done in a function called `addAttrInterpolateDirective`, which we call for every attribute on every element in the DOM. We give it the `directives` collection, as well as the attribute's value and name:

src/compile.js

```
function collectDirectives(node, attrs, maxPriority) {
  var directives = [];
  var match;
  if (node.nodeType === Node.ELEMENT_NODE) {
    var normalizedNodeName = directiveNormalize(nodeName(node).toLowerCase());
    addDirective(directives, normalizedNodeName, 'E', maxPriority);
    _.forEach(node.attributes, function(attr) {

      // ...

      addAttrInterpolateDirective(directives, attr.value, normalizedAttrName);
      addDirective(directives, normalizedAttrName, 'A', maxPriority,
        attrStartName, attrEndName);

      // ...

    });

    // ...
  }

  // ...
}
```

This function's basic structure is familiar: In tries to make an interpolation function for the attribute value. If one is found, it generates a directive that starts watching that interpolation function and sets the attribute value in the listener function.

src/compile.js

```
function addAttrInterpolateDirective(directives, value, name) {
  var interpolateFn = $interpolate(value, true);
  if (interpolateFn) {
    directives.push({
      priority: 100,
      compile: function() {
        return function link(scope, element) {
          scope.$watch(interpolateFn, function(newValue) {
            element.attr(name, newValue);
          });
        };
      }
    });
  }
}
```


Note that this directive has its priority set to `100`, causing it to be compiled before most directives written by application developers.

That's the basic behavior of attribute interpolation taken care of. Now we can start talking about the several special cases caused by the fact that there may be other directives present on the element.

First of all, other directives may be *observing* attributes on this element through `Attributes.$observe`. That may be because they have `@` bindings on their isolate scope, or because they register observers explicitly. When an attribute value changes because of interpolation, we want those observers to be fired:

test/compile_spec.js

```
it('fires observers on attribute expression changes', function() {
  var observerSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        link: function(scope, element, attrs) {
          attrs.$observe('alt', observerSpy);
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<img alt="{{myAltText}}" my-directive>');
    $compile(el)($rootScope);

    $rootScope.myAltText = 'My favourite photo';
    $rootScope.$apply();
    expect(observerSpy.calls.mostRecent().args[0])
      .toEqual('My favourite photo');
  });
});
```

The test fails because the observer was never called with the attribute value that got interpolated. Instead it was just called with the original text before interpolation.

We can fix this by changing the way we set the attribute. Currently we're just doing it with jQuery's `attr` function. What we should use instead is the `$set` method of the Attributes object, because it is aware of observers and will call them in addition to actually setting the attribute in the DOM:

src/compile.js

```
return function link(scope, element, attrs) {
  scope.$watch(interpolateFn, function(newValue) {
    attrs.$set(name, newValue);
  });
};
```


Another problem is one we saw already when the previous test was failing: The observer was getting called *before interpolation* with the string `'{{myAltText}}'`. Getting a value like that pretty much never makes sense from an application developer's point of view. We want that observer to *only* fire after interpolation has already been applied:

test/compile_spec.js

```
it('fires observers just once upon registration', function() {
  var observerSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        link: function(scope, element, attrs) {
          attrs.$observe('alt', observerSpy);
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<img alt="{{myAltText}}" my-directive>');
    $compile(el)($rootScope);
    $rootScope.$apply();

    expect(observerSpy.calls.count()).toBe(1);
  });
});
```

In the Attributes chapter we implemented a feature where observers always get called during the next digest after registration. We now want to *skip* this feature when observing interpolated attributes. Observers for them will get called on the next digest anyway, because the interpolation watcher will set the attribute's value.

Let's assume that the observers array for a given attribute will contain a marker attribute `$$inter` if the attribute is an interpolated one. We'll skip the initial call of the observer if that marker is set:

src/compile.js

```
Attributes.prototype.$observe = function(key, fn) {
  var self = this;
  this.$$observers = this.$$observers || Object.create(null);
  this.$$observers[key] = this.$$observers[key] || [];
  this.$$observers[key].push(fn);
  $rootScope.$evalAsync(function() {
    if (!self.$$observers[key].$$inter) {
      fn(self[key]);
    }
  });
```



```
  return function() {
    var index = self.$$observers[key].indexOf(fn);
    if (index >= 0) {
      self.$$observers[key].splice(index, 1);
    }
  };
};
```

While linking the attribute interpolation directive, we can set this flag. We have the Attributes object, and we can just add the `$$inter` flag to the observers in it. We do need to make sure the `$$observers` data structure exists before doing that, because it is created lazily:

src/compile.js

```
return function link(scope, element, attrs) {
  attrs.$$observers = attrs.$$observers || {};
  attrs.$$observers[name] = attrs.$$observers[name] || [];
  attrs.$$observers[name].$$inter = true;
  scope.$watch(interpolateFn, function(newValue) {
    attrs.$set(name, newValue);
  });
};
```

Another concern we have regarding other directives on the same element is that we should take care of the interpolation *before* they are linked. Usually when you access attributes from a directive, you don't want to have to think about whether they've already been interpolated or not. The framework should just take care that they are. Currently we're not doing that properly.

test/compile_spec.js

```
it('is done for attributes by the time other directive is linked', function() {
  var gotMyAttr;
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        link: function(scope, element, attrs) {
          gotMyAttr = attrs.myAttr;
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-attr="{{myExpr}}"></div>');
    $rootScope.myExpr = 'Hello';
    $compile(el)($rootScope);

    expect(gotMyAttr).toEqual('Hello');
  });
});
```


We'd like that attribute to be `Hello` when accessed from `myDirective`, but it is still `{{myExpr}}`. That needs to be fixed.

There are actually a couple of reasons why this happens. One is that the interpolation is only applied on the *first digest after linking*, because it is done by the listener function of the watcher. We should actually do it once already before setting up the watcher, so that we have interpolated before any digests have actually taken place.

src/compile.js

```
compile: function() {
  return function link(scope, element, attrs) {
    attrs.$$observers = attrs.$$observers || {};
    attrs.$$observers[name] = attrs.$$observers[name] || [];
    attrs.$$observers[name].$$inter = true;
    attrs[name] = interpolateFn(scope);
    scope.$watch(interpolateFn, function(newValue) {
      attrs.$set(name, newValue);
    });
  };
}
```

This still doesn't fix the test though. The other reason for the failure lies in the application order of the directives. The attribute interpolation directive has a priority of `100`, which means it's compiled before our "normal" directive, which has a default priority. However, we set up the interpolation in the *post-link* function of the directive, and if you recall from the linking chapter, post-link functions are invoked in *reverse priority order*. Our normal directive is thus linked before the interpolation directive.

If we change the link function into a *pre-link function*, things will start working like we want them to:

src/compile.js

```
compile: function() {
  return {
    pre: function link(scope, element, attrs) {
      attrs.$$observers = attrs.$$observers || {};
      attrs.$$observers[name] = attrs.$$observers[name] || [];
      attrs.$$observers[name].$$inter = true;
      attrs[name] = interpolateFn(scope);
      scope.$watch(interpolateFn, function(newValue) {
        attrs.$set(name, newValue);
      });
    }
  };
}
```


Now, what about isolate scope bindings for attributes? We already discussed that the `@` attribute bindings are implemented using observers, and we've taken care of observers, haven't we?

In general, the bindings do work, but there is still an issue with the initial values of those bindings. During the linking of an isolate scope directive, its attribute bindings are currently pointing to attribute values *before interpolation*, which is a similar problem as we had with regular attribute access earlier:

test/compile_spec.js

```
it('is done for attributes by the time bound to iso scope', function() {
  var gotMyAttr;
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        scope: {myAttr: '@'},
        link: function(scope, element, attrs) {
          gotMyAttr = scope.myAttr;
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-attr="{{myExpr}}"></div>');
    $rootScope.myExpr = 'Hello';
    $compile(el)($rootScope);

    expect(gotMyAttr).toEqual('Hello');
  });
});
```

The reason for this is in the isolate binding setup for these attributes. We're setting up an initial value to the binding destination in `initializeDirectiveBindings`. That's before *any* directives, including the attribute interpolation one, have been linked:

src/compile.js

```
case '@':
  attrs.$observe(attrName, function(newAttrValue) {
    destination[scopeName] = newAttrValue;
  });
  if (attrs[attrName]) {
    destination[scopeName] = attrs[attrName];
  }
  break;
```

If we change this initial value setup so that it goes through interpolation, the issue will be fixed:

src/compile.js


```
case '@':
  attrs.$observe(attrName, function(newAttrValue) {
    destination[scopeName] = newAttrValue;
  });
  if (attrs[attrName]) {
    destination[scopeName] = $interpolate(attrs[attrName])(scope);
  }
  break;
```

Here we use the form of `$interpolate` *without* the `mustHaveExpressions` flag so we can just call the interpolation function whether there actually is anything to interpolate or not.

Another thing that may occur when there are directives on an element is that those directives manipulate the element's attributes during compilation. This test case illustrates a situation where a directive replaces the value of an attribute in its `compile` function. The problem is that our attribute interpolation function doesn't pick this up. It still ends up using the attribute value *before the replacement*, which is definitely something we don't want:

test/compile_spec.js

```
it('is done for attributes so that compile-time changes apply', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        compile: function(element, attrs) {
          attrs.$set('myAttr', '{{myDifferentExpr}}');
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-attr="{{myExpr}}"></div>');
    $rootScope.myExpr = 'Hello';
    $rootScope.myDifferentExpr = 'Other Hello';
    $compile(el)($rootScope);
    $rootScope.$apply();

    expect(el.attr('my-attr')).toEqual('Other Hello');
  });
});
```

The problem here is that the attribute interpolation function is generated during compilation, and this directive replaces the attribute *later* during the compilation.

What we should do is add a check to the link function of the attribute interpolation directive, which regenerates the interpolation function if the attribute value has changed since compilation:

src/compile.js


```
pre: function link(scope, element, attrs) {
  var newValue = attrs[name];
  if (newValue !== value) {
    interpolateFn = $interpolate(newValue, true);
  }

  attrs.$$observers = attrs.$$observers || {};
  attrs.$$observers[name] = attrs.$$observers[name] || [];
  attrs.$$observers[name].$$inter = true;

  attrs[name] = interpolateFn(scope);
  scope.$watch(interpolateFn, function(newValue) {
    attrs.$set(name, newValue);
  });
}
```

What also might happen is a directive may completely *remove* an attribute during compilation. If it had interpolation going on before, it shouldn't anymore:

test/compile_spec.js

```
it('is done for attributes so that compile-time removals apply', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        compile: function(element, attrs) {
          attrs.$set('myAttr', null);
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-attr="{{myExpr}}"></div>');
    $rootScope.myExpr = 'Hello';
    $compile(el)($rootScope);
    $rootScope.$apply();

    expect(el.attr('my-attr')).toBeFalsy();
  });
});
```

Not only is the interpolation not working correctly, it is now actually throwing an exception during linking. That's definitely not something we want to happen.

We should guard the re-generation of the interpolation function so that we don't try to do it if there is no new value at all. We should also exit early from the link function before setting up the watcher.

src/compile.js


```
pre: function link(scope, element, attrs) {
  var newValue = attrs[name];
  if (newValue !== value) {
    interpolateFn = newValue && $interpolate(newValue, true);
  }
  if (!interpolateFn) {
    return;
  }

  attrs.$$observers = attrs.$$observers || {};
  attrs.$$observers[name] = attrs.$$observers[name] || [];
  attrs.$$observers[name].$$inter = true;

  attrs[name] = interpolateFn(scope);
  scope.$watch(interpolateFn, function(newValue) {
    attrs.$set(name, newValue);
  });
}
```

And finally, an additional precaution we should have is to prevent users from doing interpolation in event handlers like `onclick`. If a user was to do that, it wouldn't work as expected, because the regular event handler would be triggered outside of an Angular digest and without access to any scope. Because this is not always clear, especially to beginners, Angular explicitly won't let you do it and throws an exception if you try. You should use the `ng-*` event handler directives instead.

test/compile_spec.js

```
it('cannot be done for event handler attributes', function() {
  var injector = makeInjectorWithDirectives({});
  injector.invoke(function($compile, $rootScope) {
    $rootScope.myFunction = function() { };
    var el = $('<button onclick="{{myFunction()}}"></button>');
    expect(function() {
      $compile(el)($rootScope);
    }).toThrow();
  });
});
```

We'll guard the pre-link function of the attribute interpolation directive with a check that sees if the attribute name begins with `on`, or equals `formaction`. This covers all current, and likely all future standard event attributes:

src/compile.js


```
pre: function link(scope, element, attrs) {
  if (/^(on[a-z]+|formaction)$/.test(name)) {
    throw 'Interpolations for HTML DOM event attributes not allowed';
  }

  // ...
}
```

Optimizing Interpolation Watches With A Watch Delegate

Interpolation functions end up being executed very often, because of the watchers that `$compile` sets up for them. Also, a typical application contains a relatively large number of interpolations, because they're convenient and we end up peppering them all over our views.

For these reasons, it makes sense to spend some time trying to make interpolations as fast as possible. There's one particular optimization Angular applies here, and that is to try to minimize the amount of times the interpolation result is constructed. It does this by using watch delegates - a feature we introduced in Part 2 of the book.

What currently happens with interpolation is we are watching the value of the resulting string after all the expressions and static parts of the text are combined. This is conceptually exactly what we want: We want to update the DOM when the text that goes in the DOM changes.

But when can that text actually change? It can only change when at least one of the values of the *expressions* in the interpolation changes. We are currently constructing a new string each time the watch runs, whether the inputs changed or not, causing us to do a lot of unnecessary work and to add a lot of garbage collection pressure because of all those strings. If we could change this so that we wouldn't bother constructing a new string if none of the expression values had changed, we could go faster. Here's where watch delegates become useful.

As a reminder, a watch delegate is a special method you can attach to a watch function as the `$$watchDelegate` attribute. When you do that, the watcher implementation in Scopes will use that delegate to detect changes instead of the return value of the watch function itself.

In our case, we can construct a watch delegate that checks for changes in the interpolated expressions without having to form the resulting string.

To begin with, we can test that the interpolation functions returned by `$interpolate` do in fact have watch delegates on them:

test/interpolate_spec.js

```
it('uses a watch delegate', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');
```



```
  var interp = $interpolate('has an {{expr}}');
  expect(interp.$$watchDelegate).toBeDefined();
});
```

We can attach one by adding it to the object extension we already have for the interpolation function. Recall that a watch delegate takes the Scope where watching should be done, and the listener function to call when changes are detected:

src/interpolate.js

```
return _.extend(function interpolationFn(context) {
  return _.reduce(parts, function(result, part) {
    if (_.isFunction(part)) {
      return result + stringify(part(context));
    } else {
      return result + part;
    }
  }, '');
}, {
  expressions: expressions,
  $$watchDelegate: function(scope, listener) {

  }
});
```

This fixes our newest test but breaks a bunch of other ones. That's because Scope is now picking up our `$$watchDelegate` to use, but it doesn't do anything yet. We'll use our existing tests as a guide to tell us when we have a working watch delegate: When they're back to green, we're done.

In the watch delegate we should detect any changes to any of the expressions we found in the interpolated text. We don't yet have a collection of the parsed expression functions, so let's add one first:

src/interpolate.js

```
function $interpolate(text, mustHaveExpressions) {
  var index = 0;
  var parts = [];
  var expressions = [];
  var expressionFns = [];
  var startIndex, endIndex, exp, expFn;
  while (index < text.length) {
    startIndex = text.indexOf('{{', index);
    if (startIndex !== -1) {
      endIndex = text.indexOf('}}', startIndex + 2);
    }
    if (startIndex !== -1 && endIndex !== -1) {
```

```
      if (startIndex !== index) {
        parts.push(unescapeText(text.substring(index, startIndex)));
      }
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
      parts.push(expFn);
      expressions.push(exp);
      expressionFns.push(expFn);
      index = endIndex + 2;
    } else {
      parts.push(unescapeText(text.substring(index)));
      break;
    }
  }

  // ...

}
```

Now we can watch these expression function. We can actually watch them all at once using the `$watchGroup` feature of the Scope:

src/interpolate.js

```
$$watchDelegate: function(scope, listener) {
  return scope.$watchGroup(expressionFns, function() {

  });
}
```

What we should do inside the watch group's listener function is to actually construct the result string and give it to the listener function. We need to repeat pretty much the same reduction code that we have in the interpolation function itself:

src/interpolate.js

```
$$watchDelegate: function(scope, listener) {
  return scope.$watchGroup(expressionFns, function() {
    listener(_.reduce(parts, function(result, part) {
      if (_.isFunction(part)) {
        return result + stringify(part(scope));
      } else {
        return result + part;
      }
    }, ''));
  });
}
```

Because of this repetition, we can introduce a function called `compute` that wraps the reduction logic, given a context object:

src/interpolate.js

```
function compute(context) {
  return _.reduce(parts, function(result, part) {
    if (_.isFunction(part)) {
      return result + stringify(part(context));
    } else {
      return result + part;
    }
  }, '');
}
```

Then we can just use `compute` in both the main interpolation function and the watch delegate, so that we don't have so much duplicated logic:

src/interpolate.js

```
return _.extend(function interpolationFn(context) {
  return compute(context);
}, {
  expressions: expressions,
  $$watchDelegate: function(scope, listener) {
    return scope.$watchGroup(expressionFns, function() {
      listener(compute(scope));
    });
  }
});
```

The watch delegate is still a bit haphazard, since it doesn't actually fulfill the contract of listener function calls: Listeners should be called with both the new *and old values* (as well as the scope object). We're calling our listener function with just the new values, which isn't enough:

test/interpolate_spec.js

```
it('correctly returns new and old value when watched', function() {
  var injector = createInjector(['ng']);
  var $interpolate = injector.get('$interpolate');
  var $rootScope = injector.get('$rootScope');

  var interp = $interpolate('{{expr}}');
  var listenerSpy = jasmine.createSpy();

  $rootScope.$watch(interp, listenerSpy);
  $rootScope.expr = 42;
```



```
  $rootScope.$apply();
  expect(listenerSpy.calls.mostRecent().args[0]).toEqual('42');
  expect(listenerSpy.calls.mostRecent().args[1]).toEqual('42');

  $rootScope.expr++;
  $rootScope.$apply();
  expect(listenerSpy.calls.mostRecent().args[0]).toEqual('43');
  expect(listenerSpy.calls.mostRecent().args[1]).toEqual('42');
});
```

In the watch delegate we can track the last value we computed, and then give it as the old value each time. We'll also pass in the scope to the listener to fulfill the contract:

src/interpolate.js

```
$$watchDelegate: function(scope, listener) {
  var lastValue;
  return scope.$watchGroup(expressionFns, function() {
    var newValue = compute(scope);
    listener(newValue, lastValue, scope);
    lastValue = newValue;
  });
}
```

The listener contract also says that on the *first* watch run both the old and new values should be the same. We should cover that as well. If the watch group gives the same new and old values, we give the same new and old value to our listener:

src/interpolate.js

```
$$watchDelegate: function(scope, listener) {
  var lastValue;
  return scope.$watchGroup(expressionFns, function(newValues, oldValues) {
    var newValue = compute(scope);
    listener(
      newValue,
      (newValues === oldValues ? newValue : lastValue),
      scope
    );
    lastValue = newValue;
  });
}
```

There's an additional optimization trick we could still apply here. We are getting the `newValues` array by our watch group, and that array contains the newest computed values of all the expressions from the interpolation. But we don't really use them for anything, and instead in our listener function we end up evaluating those expressions *again* in order to construct the interpolated string. We should find a way to use those values given to us to eliminate the double calculation. This requires a bit of refactoring.

What we currently have in the `parts` array is a mixed array of static strings and expression functions. What we want to do in our watch delegate is to use the pre-supplied expression values in `newValues` instead of calling the expression functions in `parts`.

Before we can do that, we need to know the positions in which to fill in the pre-supplied values. Let's collect an array of the position in which each expression resides inside the `parts` array:

src/interpolate.js

```
function $interpolate(text, mustHaveExpressions) {
  var index = 0;
  var parts = [];
  var expressions = [];
  var expressionFns = [];
  var expressionPositions = [];
  var startIndex, endIndex, exp, expFn;
  while (index < text.length) {
    startIndex = text.indexOf('{{', index);
    if (startIndex !== -1) {
      endIndex = text.indexOf('}}', startIndex + 2);
    }
    if (startIndex !== -1 && endIndex !== -1) {
      if (startIndex !== index) {
        parts.push(unescapeText(text.substring(index, startIndex)));
      }
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
      expressions.push(exp);
      expressionFns.push(expFn);
      expressionPositions.push(parts.length);
      parts.push(expFn);
      index = endIndex + 2;
    } else {
      parts.push(unescapeText(text.substring(index)));
      break;
    }
  }

  // ...

}
```


Notice that we need to push to `expressionPositions` before we push to `parts` to get the indexes to line up correctly.

Now we can rewrite the `compute` function so that instead of evaluating expression functions, it receives the array of precomputed values. It'll walk over those values and *replace them in the `parts` array* in the locations determined by `expressionPositions`. The result is an array of strings and strings only - some of them static and some the latest results of expressions. The interpolated string can now be formed by just joining the parts together:

src/interpolate.js

```
function compute(values) {
  _.forEach(values, function(value, i) {
    parts[expressionPositions[i]] = stringify(value);
  });
  return parts.join('');
}
```

The contract of `compute` has now changed. In the interpolation function we need to evaluate all expression functions to get the values that we can give to compute:

src/interpolate.js

```
return _.extend(function interpolationFn(context) {
  var values = _.map(expressionFns, function(expressionFn) {
    return expressionFn(context);
  });
  return compute(values);
}, {
  // ..
});
```

In the watch delegate - and here's the payoff of this refactoring - we can just pass in the `newValues` array we get from the watch group. No need to evaluate any more expressions:

src/interpolate.js

```
return _.extend(function interpolationFn(context) {
  var values = _.map(expressionFns, function(expressionFn) {
    return expressionFn(context);
  });
  return compute(values);
}, {
  expressions: expressions,
  $$watchDelegate: function(scope, listener) {
    var lastValue;
    return scope.$watchGroup(expressionFns, function(newValues, oldValues) {
      var newValue = compute(newValues);
      listener(
```



```
        newValue,
        (newValues === oldValues ? newValue : lastValue),
        scope
      );
      lastValue = newValue;
    });
  }
});
```

That's the final, optimized version of the watch delegate!

Making Interpolation Symbols Configurable

There's one more thing we should add to our implementation of the `$interpolate` service before it's complete. That is the ability to configure the start and end markers - or *symbols* - of interpolation expressions. You see, Angular lets you change these from the defaults `{{` and `}}` to anything you want.

The value of using this feature is questionable, mostly because it makes your views look different from all other Angular applications in the world. But the feature is there, and it's good to know how it works.

The start and end symbols can be changed at configuration time by calling the methods `startSymbol` and `endSymbol` on the `$interpolateProvider`. At run time you can read (but no longer change) them from the `startSymbol` and `endSymbol` methods of the `$interpolate` service itself:

test/interpolate_spec.js

```
it('allows configuring start and end symbols', function() {
  var injector = createInjector(['ng', function($interpolateProvider) {
    $interpolateProvider.startSymbol('FOO').endSymbol('OOF');
  }]);
  var $interpolate = injector.get('$interpolate');
  expect($interpolate.startSymbol()).toEqual('FOO');
  expect($interpolate.endSymbol()).toEqual('OOF');
});
```

In the provider we'll make the setters, that also double as getters when called with no arguments. The symbols are stored in variables inside the provider:

src/interpolate.js

```
function $InterpolateProvider() {
  var startSymbol = '{{';
  var endSymbol = '}}';
```



```
  this.startSymbol = function(value) {
    if (value) {
      startSymbol = value;
      return this;
    } else {
      return startSymbol;
    }
  };

  this.endSymbol = function(value) {
    if (value) {
      endSymbol = value;
      return this;
    } else {
      return endSymbol;
    }
  };

  // ...

}
```

The runtime getters we can attach to the `$interpolate` function when we have it. The symbols will be constant at runtime so we can just use _.constant for them:

src/interpolate.js

```
function $InterpolateProvider() {
  var startSymbol = '{{';
  var endSymbol = '}}';

  // ...

  this.$get = ['$parse', function($parse) {

    function $interpolate(text, mustHaveExpressions) {

      // ...

    }

    $interpolate.startSymbol = _.constant(startSymbol);
    $interpolate.endSymbol   = _.constant(endSymbol);

    return $interpolate;
  }];

}
```

Now we have configurable symbols, so let's actually make use of them. This test configures the symbols to `FOO` and `OOF` and then checks that they do in fact act as expression markers in interpolation strings:

test/interpolate_spec.js

```
it('works with start and end symbols that differ from default', function() {
  var injector = createInjector(['ng', function($interpolateProvider) {
    $interpolateProvider.startSymbol('FOO').endSymbol('OOF');
  }]);
  var $interpolate = injector.get('$interpolate');
  var interpFn = $interpolate('FOOmyExprOOF');
  expect(interpFn({myExpr: 42})).toEqual('42');
});
```

Furthermore, we should check that the default start and end symbols do *not* work when they have been changed to something else. They are interpreted as static text:

test/interpolate_spec.js

```
it('does not work with default symbols when reconfigured', function() {
  var injector = createInjector(['ng', function($interpolateProvider) {
    $interpolateProvider.startSymbol('FOO').endSymbol('OOF');
  }]);
  var $interpolate = injector.get('$interpolate');
  var interpFn = $interpolate('{{myExpr}}');
  expect(interpFn({myExpr: 42})).toEqual('{{myExpr}}');
});
```

The support for all of this resides in the `while` loop in `$interpolate`. Instead of using the hardcoded `{{` and `}}` strings it should use the `startSymbol` and `endSymbol` variables. Furthermore, it can no longer assume that the length of the symbols will be two, so it has to use the lengths of the variables to calculate the different points in which to split the string:

src/interpolate.js

```
while (index < text.length) {
  startIndex = text.indexOf(startSymbol, index);
  if (startIndex !== -1) {
    endIndex = text.indexOf(endSymbol, startIndex + startSymbol.length);
  }
  if (startIndex !== -1 && endIndex !== -1) {
    if (startIndex !== index) {
      parts.push(unescapeText(text.substring(index, startIndex)));
    }
    exp = text.substring(startIndex + startSymbol.length, endIndex);
    expFn = $parse(exp);
```

```
    expressions.push(exp);
    expressionFns.push(expFn);
    expressionPositions.push(parts.length);
    parts.push(expFn);
    index = endIndex + endSymbol.length;
  } else {
    parts.push(unescapeText(text.substring(index)));
    break;
  }
}
```

These custom start and end symbols should also support the unescaping mechanism we have. If a user configures the start symbol as `FOO`, the string `\F\O\O` in their template should be interpolated to `FOO`:

test/interpolate_spec.js

```
it('supports unescaping for reconfigured symbols', function() {
  var injector = createInjector(['ng', function($interpolateProvider) {
    $interpolateProvider.startSymbol('FOO').endSymbol('OOF');
  }]);
  var $interpolate = injector.get('$interpolate');
  var interpFn = $interpolate('\\F\\O\\OmyExpr\\O\\O\\F');
  expect(interpFn({})).toEqual('FOOmyExprOOF');
});
```

This is a bit trickier. We'll no longer be able to use those hardcoded regular expressions for unescaping. We need to form regular expressions at runtime based on what has been configured as the start and end symbols. For both symbols, we'll make a regular expression using the RegExp constructor. The pattern of the regex is based on running each character in the configured symbol through the `escapeChar` function (which we'll introduce in a moment):

src/interpolate.js

```
this.$get = ['$parse', function($parse) {
  var escapedStartMatcher =
    new RegExp(startSymbol.replace(/./g, escapeChar), 'g');
  var escapedEndMatcher   =
    new RegExp(endSymbol.replace(/./g, escapeChar), 'g');

  // ...

}];
```

The `escapeChar` function puts not one, nor two, but *three* backslashes before the character. We additionally need to escape each of them in the string, so we end up with *six* backslashes in total:

src/interpolate.js


```
function escapeChar(char) {
  return '\\\\\\' + char;
}
```

The first two backslashes end up in the regular expression as the matcher for the backslash character itself (`/\\/`). The third one is to escape the original character as well. Our default start and end symbols have curly braces which would have a special meaning in regular expressions unless we escaped them.

This code will dynamically form the following kind of regexp dynamically for the default start symbol:

```
/\\\{\\\{/g
```

And this for the custom start symbol used in our test:

```
/\\\F\\\O\\\O/g
```

Now we can change `unescapeText` to use these generated regexps instead of the hardcoded ones:

src/interpolate.js

```
this.$get = ['$parse', function($parse) {
  var escapedStartMatcher =
    new RegExp(startSymbol.replace(/./g, escapeChar), 'g');
  var escapedEndMatcher   =
    new RegExp(endSymbol.replace(/./g, escapeChar), 'g');

  function unescapeText(text) {
    return text.replace(escapedStartMatcher, startSymbol)
               .replace(escapedEndMatcher, endSymbol);
  }

  // ...
}];
```

That's how start and end symbols can be customized. But what happens if you do do that, and also happen to use third party code from other projects or open source libraries? Do things immediately break because you decided to use interpolation symbols that differ from what the third party code assumes?

The answer is that things won't break and you can still use third party code. That's because in `$compile` we are going to *denormalize* all directive templates, which means we'll replace the default `{{` and `}}` symbols in them with your custom ones. Any templates that don't follow your convention will be processed so that they do.

Here's an example: An application where the start and end symbols have been reconfigured to `[[` and `]]`. It uses a directive whose template still uses `{{` and `}}`. Interpolation *should still work* inside the template:

test/compile_spec.js


```
it('denormalizes directive templates', function() {
  var injector = createInjector(['ng',
      function($interpolateProvider, $compileProvider) {
    $interpolateProvider.startSymbol('[[').endSymbol(']]');
    $compileProvider.directive('myDirective', function() {
      return {
        template: 'Value is {{myExpr}}'
      };
    });
  }]);
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $rootScope.myExpr = 42;
    $compile(el)($rootScope);
    $rootScope.$apply();

    expect(el.html()).toEqual('Value is 42');
  });
});
```

What we'll do is run all directive templates through a function called `denormalizeTemplate`. That happens just after we've obtained those templates. For `template` attributes that happens in `applyDirectivesToNode`:

src/compile.js

```
if (directive.template) {
  if (templateDirective) {
    throw 'Multiple directives asking for template';
  }
  templateDirective = directive;
  var template = _.isFunction(directive.template) ?
                    directive.template($compileNode, attrs) :
                    directive.template;
  template = denormalizeTemplate(template);
  $compileNode.html(template);
}
```

For `templateUrl` attributes that happens in `compileTemplateUrl` after we've received the template from `$http`:

src/compile.js


```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {

  // ...

  $http.get(templateUrl).success(function(template) {
    template = denormalizeTemplate(template);

    // ...

  });

  // ...
}
```

The `denormalizeTemplate` function is introduced right inside the `$get` method of the `$compile` provider.

We first get the start and end symbols from the `$interpolate` service and compare them to the default ones. If they haven't been customized, we'll initialize `denormalizeTemplate` to a function that does nothing. If they have been customized, we'll make a function that uses regular expression substitution to replace the default symbols with the custom ones:

src/compile.js

```
this.$get = ['$injector', '$parse', '$controller', '$rootScope',
             '$http', '$interpolate',
function($injector, $parse, $controller, $rootScope, $http, $interpolate) {

  var startSymbol = $interpolate.startSymbol();
  var endSymbol   = $interpolate.endSymbol();
  var denormalizeTemplate = (startSymbol === '{{' && endSymbol === '}}') ?
    _.identity :
    function(template) {
      return template.replace(/\{\{/g, startSymbol)
        .replace(/\}\}/g, endSymbol);
    };

  // ...

}];
```

Summary

Expression interpolation is a hugely important feature in Angular. It's pretty much the first thing taught to new Angular developers, and no Angular application can get very far with it.

The implementation of the feature itself isn't hugely complicated, since it builds on the expression parsing in `$parse`, the watchers on Scopes, as well as the directive implementation provided by `$compile`. For us, it serves as a nice demonstration to how these low-level features can be combined to produce something very useful.

We also learned some tricks on how watch delegates and watch groups can be used to eliminate work that would otherwise have to be done during change detection. Similar tricks could be applied in application code as well.

In this chapter you have learned:

- That there is an `$interpolate` service with which you can turn any string into an interpolation function.
- How the service parses the expressions contained in the string, and evaluates them when the interpolation function is called.
- How expression results are stringified before they are concatenated into the resulting string.
- That you can escape the start and end symbols if you want to include them literally in your UI: `\{\{` and `\}\}`.
- That Angular documentation actually recommends doing this escaping for all server-provided dynamic content, as an additional cross-site scripting forgery prevention measure.
- How you can instruct `$interpolate` to skip creating an interpolation function if there are no dynamic expressions in the string, using the `mustHaveExpressions` flag.
- How interpolation integrates into text nodes: Directives are generated for text nodes on the fly. The directives watch the interpolation expressions and update the node contents.
- That the `hg-binding` classes and `$binding` data is added to the parents of text nodes to aid in tooling and introspection.
- How interpolation integrates into element attributes: Directives are generated on the fly. The directives watch the interpolation expressions and update the attribute values.
- How attribute interpolation integrates with attribute observers.
- That Angular does extra work to make sure interpolation is done before other directives for the element are linked.
- How Angular prevents you from doing interpolation in standard DOM event listeners.
- How interpolation watching is optimized with watch delegates to try to minimise the work needed to check for changes.
- How you can configure your own custom interpolation start and end symbols to replace `{{` and `}}`.
- That Angular denormalizes directive templates to enable code reuse even when you reconfigure your interpolation symbols.

Chapter 22

Bootstrapping Angular