Chapter 1
Scopes and Dirty-Checking

Download the code for the starting point of this chapter.

Scope Objects

Scopes can be created by applying the `new` operator to the `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

```
'use strict';

var Scope = require('../src/scope');

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 require `Scope`, which we are expecting to find under the `src` directory. The test itself creates a `Scope`, assigns an arbitrary property on it and checks that it was indeed assigned.

If you have Karma 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

```
'use strict';

function Scope() {

}
```

```
module.exports = Scope;
```

In the test case we're assigning a property (called `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 `$watch` you can attach something called a *watcher* to a scope. A watcher is something that is notified when a change occurs on the scope. You can create a watcher by calling `$watch` with two arguments, both of which should be functions:

- 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();
  });
};
```

This function is using the `forEach` function from LoDash, so we need to require LoDash at the top of the file:

src/scope.js

```
'use strict';

var _ = require('lodash');

// ...
```

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 more convenient, we can pass it as an argument to watch functions. The watch functions may then easily grab and return something from the scope:

```
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 the 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`. The listener should be invoked in this case as well, but it doesn't because our current implementation doesn't consider an initial `undefined` value as a "change":

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, including `undefined`.

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`.

Keeping The Digest Going While It Stays 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 Karma 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.

We don't yet have LoDash available in `scope_spec.js`, so we need to require it in in order to use the `range` and `times` functions:

test/scope_spec.js

```
'use strict';

var _ = require('lodash');
var Scope = require('../src/scope');
```

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 in this book. 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 we'll instead just use 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.

The equality characteristics of NaN really come from the IEEE standard for floating point numbers, and apply to many languages other than JavaScript too. There's an interesting discussion about this on Stack Overflow.

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.

Handling Exceptions

Our dirty checking 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, 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 during 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 such a service is not implemented in this book, we'll simply log the exceptions to the console.

In watches there are two points where 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;
};
```

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 `unshift` instead of `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:

test/scope_spec.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 one watch removes *several watches*:

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:

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

Summary

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

- The two-sided process underlying Angular's dirty-checking: `$watch` and `$digest`.
- The dirty-checking loop and the TTL mechanism for short-circuiting it.
- The difference between reference-based and value-based comparison.
- Exception handling in the Angular digest.
- Destroying watches so they won't get executed again.