Directives are a way to teach HTML new tricks. During DOM compilation directives are matched against the HTML and executed. This allows directives to register behavior, or transform the DOM.

Angular comes with a built in set of directives which are useful for building web applications but can be extended such that HTML can be turned into a declarative domain specific language (DSL).

## Invoking directives from HTML

Directives have camel cased names such as `ngBind`. The directive can be invoked by translating the camel case name into snake case with these special characters `:` , `-`, or `_`. Optionally the directive can be prefixed with `x-`, or `data-` to make it HTML validator compliant. Here is a list of some of the possible directive names: `ng:bind`, `ng-bind`, `ng_bind`, `x-ng-bind` and `data-ng-bind`.

The directives can be placed in element names, attributes, class names, as well as comments. Here are some equivalent examples of invoking `myDir`. (However, most directives are restricted to attribute only.)

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;span my-dir=&quot;exp&quot;&gt;&lt;/span&gt;</td>
<td>Element name</td>
</tr>
<tr>
<td>&lt;span class=&quot;my-dir: exp;&quot;&gt;&lt;/span&gt;</td>
<td>Attribute values can be combined</td>
</tr>
<tr>
<td>&lt;my-dir&gt;&lt;/my-dir&gt;</td>
<td>Attribute names can be camel-cased</td>
</tr>
<tr>
<td>&lt;!-- directive: my-dir exp --&gt;</td>
<td>Comment format</td>
</tr>
</tbody>
</table>

Directives can be invoked in many different ways, but are equivalent in the end result as shown in the following example.

### Source

**index.html**

```html
<!doctype html>
<html ng-app>
<head>
  <script src="http://code.angularjs.org/1.0.4/angular.min.js"></script>
  <script src="script.js"></script>
</head>
<body>
  <div ng-controller="Ctrl1">
    Hello <input ng-model="name" />
    <hr/>
    &lt;span ng:bind="name"&gt; &lt;span ng:bind="name"&gt;</span> <br/>
    &lt;span ng_bind="name"&gt; &lt;span ng_bind="name"&gt;</span> <br/>
    &lt;span data-ng-bind="name"&gt; &lt;span data-ng-bind="name"&gt;</span> <br/>
    &lt;span x-ng-bind="name"&gt; &lt;span x-ng-bind="name"&gt;</span> <br/>
  </div>
</body>
</html>
```

**script.js**

```javascript
function Ctrl1($scope) {
```
Demo

Hello angular

String interpolation

During the compilation process the compiler matches text and attributes using the $interpolate service to see if they contain embedded expressions. These expressions are registered as watches and will update as part of normal digest cycle. An example of interpolation is shown here:

```html
<a href=img/{{username}}.jpg>Hello {{username}}!</a>
```

Compilation process, and directive matching

Compilation of HTML happens in three phases:

1. First the HTML is parsed into DOM using the standard browser API. This is important to realize because the templates must be parsable HTML. This is in contrast to most templating systems that operate on strings, rather than on DOM elements.

2. The compilation of the DOM is performed by the call to the $compile() method. The method traverses the DOM and matches the directives. If a match is found it is added to the list of directives associated with the given DOM element. Once all directives for a given DOM element have been identified they are sorted by priority and their compile() functions are executed. The directive compile function has a chance to modify the DOM structure and is responsible for producing a link() function explained next. The $compile() method returns a combined linking function, which is a collection of all of the linking functions returned from the individual directive compile functions.

3. Link the template with scope by calling the linking function returned from the previous step. This in turn will call the linking function of the individual directives allowing them to register any listeners on the elements and set up any watches with the scope. The result of this is a live binding between the scope and the DOM. A change in the scope is reflected in the DOM.

```javascript
var $compile = ...; // injected into your code
```
Reasons behind the compile/link separation

At this point you may wonder why the compile process is broken down to a compile and link phase. To understand this, let's look at a real world example with a repeater:

```html
  <ul>
    <li ng-repeat="action in user.actions">{{action.description}}</li>
  </ul>
```

The short answer is that compile and link separation is needed any time a change in model causes a change in DOM structure such as in repeaters.

When the above example is compiled, the compiler visits every node and looks for directives. The `{{user}}` is an example of an interpolation directive. `ngRepeat` is another directive. But `ngRepeat` has a dilemma. It needs to be able to quickly stamp out new `li` for every `action in user.actions`. This means that it needs to save a clean copy of the `li` element for cloning purposes and as new `actions` are inserted, the template `li` element needs to be cloned and inserted into `ul`. But cloning the `li` element is not enough. It also needs to compile the `li` so that its directives such as `{{action.descriptions}}` evaluate against the right `scope`. A naive method would be to simply insert a copy of the `li` element and then compile it. But compiling on every `li` element clone would be slow, since the compilation requires that we traverse the DOM tree and look for directives and execute them. If we put the compilation inside a repeater which needs to unroll 100 items we would quickly run into performance problems.

The solution is to break the compilation process into two phases; the compile phase where all of the directives are identified and sorted by priority, and a linking phase where any work which links a specific instance of the `scope` and the specific instance of an `li` is performed.

`ngRepeat` works by preventing the compilation process from descending into the `li` element. Instead the `ngRepeat` directive compiles `li` separately. The result of the `li` element compilation is a linking function which contains all of the directives contained in the `li` element, ready to be attached to a specific clone of the `li` element. At runtime the `ngRepeat` watches the expression and as items are added to the array it clones the `li` element, creates a new `scope` for the cloned `li` element and calls the link function on the cloned `li`.

Summary:

- **compile function** - The compile function is relatively rare in directives, since most directives are concerned with working with a specific DOM element instance rather than transforming the template DOM element. Any operation which can be shared among the instance of directives should be moved to the compile function for performance reasons.

- **link function** - It is rare for the directive not to have a link function. A link function allows the directive to register
Writing directives (short version)

In this example we will build a directive that displays the current time.

Source

index.html

```html
<!doctype html>
<html ng-app="time">
  <head>
    <script src="http://code.angularjs.org/1.0.4/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl2">
      Date format: <input ng-model="format">
      <hr/>
      Current time is: <span my-current-time="format"></span>
    </div>
  </body>
</html>
```

script.js

```javascript
function Ctrl2($scope) {
  $scope.format = 'M/d/yy h:mm:ss a';
}

angular.module('time', [])
  .directive('myCurrentTime', function($timeout, dateFilter) {
    return function(scope, element, attrs) {
      var format, // date format
          timeoutId; // timeoutId, so that we can cancel the time updates

      // used to update the UI
      function updateTime() {
        element.text(dateFilter(new Date(), format));
      }

      // watch the expression, and update the UI on change.
      scope.$watch(attrs.myCurrentTime, function(value) {
        format = value;
        updateTime();
      });

      // schedule update in one second
      $timeout(setInterval(updateTime, 1000));
    }
  });
```
```javascript
function updateLater() {
  // save the timeoutId for canceling
  timeoutId = $timeout(function() {
    updateTime(); // update DOM
    updateLater(); // schedule another update
  }, 1000);
}

// listen on DOM destroy (removal) event, and cancel the next UI update
// to prevent updating time after the DOM element was removed.
element.bind('$destroy', function() {
  $timeout.cancel(timeoutId);
});

updateLater(); // kick off the UI update process.
}
```

**Demo**

Date format: **M/d/yy h:mm:ss a**

Current time is: 1/25/13 9:30:19 AM

**Writing directives (long version)**

An example skeleton of the directive is shown here, for the complete list list below.

```javascript
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  var directiveDefinitionObject = {
    priority: 0,
    template: '<div></div>',
    templateUrl: 'directive.html',
    replace: false,
    transclude: false,
    restrict: 'A',
    scope: false,
    compile: function compile(tElement, tAttrs, transclude) {
      return {
        pre: function preLink(scope, iElement, iAttrs, controller) { ... },
        post: function postLink(scope, iElement, iAttrs, controller) { ... }
      }
    },
    link: function postLink(scope, iElement, iAttrs) { ... }
  };
};
```
In most cases you will not need such fine control and so the above can be simplified. All of the different parts of this skeleton are explained in following sections. In this section we are interested only in some of this skeleton.

The first step in simplifying the code is to rely on the default values. Therefore the above can be simplified as:

```javascript
var myModule = angular.module(...);
myModule.directive('directiveName', function factory(injectables) {
    var directiveDefinitionObject = {
        compile: function compile(tElement, tAttrs) {
            return function postLink(scope, iElement, iAttrs) { ... }
        }
    }
    return directiveDefinitionObject;
});
```

Most directives concern themselves only with instances, not with template transformations, allowing further simplification:

```javascript
var myModule = angular.module(...);
myModule.directive('directiveName', function factory(injectables) {
    return function postLink(scope, iElement, iAttrs) { ... }
});
```

**Factory method**

The factory method is responsible for creating the directive. It is invoked only once, when the compiler matches the directive for the first time. You can perform any initialization work here. The method is invoked using the $injector.invoke which makes it injectable following all of the rules of injection annotation.

**Directive Definition Object**

The directive definition object provides instructions to the compiler. The attributes are:

- **name** - Name of the current scope. Optional and defaults to the name at registration.
- **priority** - When there are multiple directives defined on a single DOM element, sometimes it is necessary to specify the order in which the directives are applied. The priority is used to sort the directives before their compile functions get called. Higher priority goes first. The order of directives within the same priority is undefined.
- **terminal** - If set to true then the current priority will be the last set of directives which will execute (any directives at the current priority will still execute as the order of execution on same priority is undefined).
- **scope** - If set to:
  - true - then a new scope will be created for this directive. If multiple directives on the same element request a new scope, only one new scope is created. The new scope rule does not apply for the root of the template since the root of the template always gets a new scope.
  - {} (object hash) - then a new 'isolate' scope is created. The 'isolate' scope differs from normal scope in that it does not prototypically inherit from the parent scope. This is useful when creating reusable components, which should not accidentally read or modify data in the parent scope.
The 'isolate' scope takes an object hash which defines a set of local scope properties derived from the parent scope. These local properties are useful for aliasing values for templates. Locals definition is a hash of local scope property to its source:

- @ or @attr - bind a local scope property to the value of DOM attribute. The result is always a string since DOM attributes are strings. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given `<widget my-attr="hello {{name}}">` and widget definition of scope: `{ localName: '@myAttr' }`, then widget scope property localName will reflect the interpolated value of hello {{name}}. As the name attribute changes so will the localName property on the widget scope. The name is read from the parent scope (not component scope).

- = or =attr - set up bi-directional binding between a local scope property and the parent scope property of name defined via the value of the attr attribute. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given `<widget my-attr="parentModel">` and widget definition of scope: `{ localModel: '='myAttr' }`, then widget scope property localModel will reflect the value of parentModel on the parent scope. Any changes to parentModel will be reflected in localModel and any changes in localModel will reflect in parentModel.

- & or &attr - provides a way to execute an expression in the context of the parent scope. If no attr name is specified then the attribute name is assumed to be the same as the local name. Given `<widget my-attr="count = count + value">` and widget definition of scope: `{ localFn: '&myAttr' }`, then isolate scope property localFn will point to a function wrapper for the count = count + value expression. Often it's desirable to pass data from the isolated scope via an expression and to the parent scope, this can be done by passing a map of local variable names and values into the expression wrapper fn. For example, if the expression is increment(amount) then we can specify the amount value by calling the localFn as localFn({amount: 22}).

- controller - Controller constructor function. The controller is instantiated before the pre-linking phase and it is shared with other directives if they request it by name (see require attribute). This allows the directives to communicate with each other and augment each other's behavior. The controller is injectable with the following locals:

  - $scope - Current scope associated with the element
  - $element - Current element
  - $attrs - Current attributes object for the element
  - $transclude - A transclude linking function pre-bound to the correct transclusion scope: function(cloneLinkingFn).

- require - Require another controller be passed into current directive linking function. The require takes a name of the directive controller to pass in. If no such controller can be found an error is raised. The name can be prefixed with:

  - ? - Don't raise an error. This makes the require dependency optional.
  - ^ - Look for the controller on parent elements as well.

- restrict - String of subset of EACM which restricts the directive to a specific directive declaration style. If omitted directives are allowed on attributes only.

  - E - Element name: `<my-directive></my-directive>
  - A - Attribute: `<div my-directive="exp">` `<div>
  - C - Class: `<div class="my-directive: exp;">` `<div>
  - M - Comment: `<!-- directive: my-directive exp -->`

- template - replace the current element with the contents of the HTML. The replacement process migrates all of the attributes / classes from the old element to the new one. See Creating Widgets section below for more information.

- templateUrl - Same as template but the template is loaded from the specified URL. Because the template loading is asynchronous the compilation/linking is suspended until the template is loaded.

- replace - if set to true then the template will replace the current element, rather than append the template to the
element.

- transclude - compile the content of the element and make it available to the directive. Typically used with ngTransclude. The advantage of transclusion is that the linking function receives a transclusion function which is pre-bound to the correct scope. In a typical setup the widget creates an isolate scope, but the transclusion is not a child, but a sibling of the isolate scope. This makes it possible for the widget to have private state, and the transclusion to be bound to the parent (pre-isolate) scope.

  - true - transclude the content of the directive.
  - 'element' - transclude the whole element including any directives defined at lower priority.

- compile: This is the compile function described in the section below.

- link: This is the link function described in the section below. This property is used only if the compile property is not defined.

### Compile function

```javascript
1. function compile(tElement, tAttrs, transclude) { ... }
```

The compile function deals with transforming the template DOM. Since most directives do not do template transformation, it is not used often. Examples that require compile functions are directives that transform template DOM, such as ngRepeat, or load the contents asynchronously, such as ngView. The compile function takes the following arguments.

- tElement - template element - The element where the directive has been declared. It is safe to do template transformation on the element and child elements only.
- tAttrs - template attributes - Normalized list of attributes declared on this element shared between all directive compile functions. See Attributes.
- transclude - A transclude linking function: function(scope, cloneLinkingFn).

NOTE: The template instance and the link instance may not be the same objects if the template has been cloned. For this reason it is not safe in the compile function to do anything other than DOM transformation that applies to all DOM clones. Specifically, DOM listener registration should be done in a linking function rather than in a compile function.

A compile function can have a return value which can be either a function or an object.

- returning a function - is equivalent to registering the linking function via the link property of the config object when the compile function is empty.
- returning an object with function(s) registered via pre and post properties - allows you to control when a linking function should be called during the linking phase. See info about pre-linking and post-linking functions below.

### Linking function

```javascript
1. function link(scope, iElement, iAttrs, controller) { ... }
```

The link function is responsible for registering DOM listeners as well as updating the DOM. It is executed after the template has been cloned. This is where most of the directive logic will be put.

- scope - Scope - The scope to be used by the directive for registering watches.
- iElement - instance element - The element where the directive is to be used. It is safe to manipulate the children of the element only in postLink function since the children have already been linked.
- iAttrs - instance attributes - Normalized list of attributes declared on this element shared between all directive linking functions. See Attributes.
controller - a controller instance - A controller instance if at least one directive on the element defines a controller. The controller is shared among all the directives, which allows the directives to use the controllers as a communication channel.

Pre-linking function
Executed before the child elements are linked. Not safe to do DOM transformation since the compiler linking function will fail to locate the correct elements for linking.

Post-linking function
Executed after the child elements are linked. It is safe to do DOM transformation in the post-linking function.

Attributes
The Attributes object - passed as a parameter in the link() or compile() functions - is a way of accessing:

- normalized attribute names: Since a directive such as 'ngBind' can be expressed in many ways such as 'ng:bind', or 'x-ng-bind', the attributes object allows for normalized accessed to the attributes.

- directive inter-communication: All directives share the same instance of the attributes object which allows the directives to use the attributes object as inter directive communication.

- supports interpolation: Interpolation attributes are assigned to the attribute object allowing other directives to read the interpolated value.

- observing interpolated attributes: Use $observe to observe the value changes of attributes that contain interpolation (e.g. src="{{bar}}"). Not only is this very efficient but it’s also the only way to easily get the actual value because during the linking phase the interpolation hasn't been evaluated yet and so the value is at this time set to undefined.

```
1. function linkingFn(scope, elm, attrs, ctrl) {
2.   // get the attribute value
3.   console.log(attrs.ngModel);
4.   
5.   // change the attribute
6.   attrs.$set('ngModel', 'new value');
7.   
8.   // observe changes to interpolated attribute
9.   attrs.$observe('ngModel', function(value) {
10.      console.log('ngModel has changed value to ' + value);
11.    });
12. }
```

Understanding Transclusion and Scopes
It is often desirable to have reusable components. Below is a pseudo code showing how a simplified dialog component may work.
Clicking on the "show" button will open the dialog. The dialog will have a title, which is data bound to `username`, and it will also have a body which we would like to transclude into the dialog.

Here is an example of what the template definition for the `dialog` widget may look like:

```html
<ng-show visible>...
</ng-show>

This will not render properly, unless we do some scope magic.

The first issue we have to solve is that the dialog box template expects `title` to be defined, but the place of instantiation would like to bind to `username`. Furthermore the buttons expect the `onOk` and `onCancel` functions to be present in the scope. This limits the usefulness of the widget. To solve the mapping issue we use the `locals` to create local variables which the template expects as follows:

```
scope: {
  title: '@',      // the title uses the data-binding from the parent scope
  onOk: '&',       // create a delegate onOk function
  onCancel: '&',   // create a delegate onCancel function
  visible: '='     // set up visible to accept data-binding
}
```

Creating local properties on widget scope creates two problems:

1. isolation - if the user forgets to set `title` attribute of the dialog widget the dialog template will bind to parent scope property. This is unpredictable and undesirable.

2. transclusion - the transcluded DOM can see the widget locals, which may overwrite the properties which the transclusion needs for data-binding. In our example the `title` property of the widget clobbers the `title` property of the transclusion.

To solve the issue of lack of isolation, the directive declares a new `isolated` scope. An isolated scope does not prototypically inherit from the child scope, and therefore we don't have to worry about accidentally clobbering any properties.

However `isolated` scope creates a new problem: if a transcluded DOM is a child of the widget isolated scope then it will not be able to bind to anything. For this reason the transcluded scope is a child of the original scope, before the widget.
created an isolated scope for its local variables. This makes the transcluded and widget isolated scope siblings. This may seem to be unexpected complexity, but it gives the widget user and developer the least surprise.

Therefore the final directive definition looks something like this:

```javascript
transclude: true,
scope: {
  title: '@title', // the title uses the data-binding from the parent scope
  onOk: '&onOk', // create a delegate onOk function
  onCancel: '&onCancel', // create a delegate onCancel function
  visible: '='visible ' // set up visible to accept data-binding
},
restrict: 'E',
replace: true
```

## Creating Components

It is often desirable to replace a single directive with a more complex DOM structure. This allows the directives to become a short hand for reusable components from which applications can be built.

Following is an example of building a reusable widget.

### Source

**index.html**

```html
<!doctype html>
<html ng-app="zippyModule">
  <head>
    <script src="http://code.angularjs.org/1.0.4/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl3">
      Title: <input ng-model="title" /> <br>
      Text: <textarea ng-model="text"></textarea>
    </div>
    <div class="zippy" zippy-title="Details: {{title}}...">{{text}}</div>
  </body>
</html>
```

**style.css**

```css
.zippy {
  border: 1px solid black;
  display: inline-block;
  width: 250px;
}
.zippy.opened > .title:before { content: '▼'; }
.zippy.opened > .body { display: block; }
```
.zippy.closed > .title:before { content: '► '; }
.zippy.closed > .body { display: none; }
.zippy > .title {
  background-color: black;
  color: white;
  padding: .1em .3em;
  cursor: pointer;
}
.zippy > .body {
  padding: .1em .3em;
}
function Ctrl3($scope) {
  $scope.title = 'Lorem Ipsum';
  $scope.text = 'Neque porro quisquam est qui dolorem ipsum quia dolor...';
}

angular.module('zippyModule', []).
directive('zippy', function(){
  return {
    restrict: 'C',
    // This HTML will replace the zippy directive.
    replace: true,
    transclude: true,
    scope: { title:'@zippyTitle' },
    template: '<div>
    <div class="title">{{title}}</div>
    <div class="body" ng-transclude></div>
    </div>',
    // The linking function will add behavior to the template
    link: function(scope, element, attrs) {
      // Title element
      var title = angular.element(element.children()[0]),
      // Opened / closed state
      opened = true;

      // Clicking on title should open/close the zippy
      title.bind('click', toggle);

      // Toggle the closed/opened state
      function toggle() {
        opened = !opened;
        element.removeClass(opened ? 'closed' : 'opened');
        element.addClass(opened ? 'opened' : 'closed');
      }

      // initialize the zippy
      toggle();
    }
  }
});

End to end test :
it('should bind and open / close', function() {
  input('title').enter('TITLE');
  input('text').enter('TEXT');
  expect(element('.title').text()).toEqual('Details: TITLE...');
  expect(binding('text')).toEqual('TEXT');
  expect(element('.zippy').prop('className')).toMatch(/closed/);
  element('.zippy > .title').click();
  expect(element('.zippy').prop('className')).toMatch(/opened/);
});
<table>
<thead>
<tr>
<th>Title</th>
<th>Lorem Ipsum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Neque porro quisquam est qui dolorem ipsum quia dolor...</td>
</tr>
</tbody>
</table>

▶ Details: Lorem Ipsum...