Executive Summary:

In AngularJS, a child scope normally prototypically inherits from its parent scope. One exception to this rule is a directive that uses `scope: { ... }` -- this creates an "isolate" scope that does not prototypically inherit. (and directive with transclusion) This construct is often used when creating a "reusable component" directive. In directives, the parent scope is used directly by default, which means that whatever you change in your directive that comes from the parent scope will also change in the parent scope. If you set `scope:true` (instead of `scope: { ... }`), then prototypical inheritance will be used for that directive.

Scope inheritance is normally straightforward, and you often don't even need to know it is happening... until you try **2-way data binding** (i.e., form elements, ng-model) **to a primitive** (e.g., number, string, boolean) defined on the parent scope from inside the child scope. It doesn't work the way most people expect it should work. What happens is that the child scope gets its own property that hides/shadows the parent property of the same name. This is not something AngularJS is doing – this is how JavaScript prototypal inheritance works. New AngularJS developers often do not realize that ng-repeat, ng-switch, ng-view, ng-include and ng-if all create new child scopes, so the problem often shows up when these directives are involved. (See this example for a quick illustration of the problem.)

This issue with primitives can be easily avoided by following the "best practice" of always have a '.' in your ng-models – watch 3 minutes worth. Misko demonstrates the primitive binding issue with ng-switch.

Having a '.' in your models will ensure that prototypal inheritance is in play. So, use `<input type="text" ng-model="someObj.prop1">` rather than `<input type="text" ng-model="prop1">`.

If you really want/need to use a primitive, there are two workarounds:

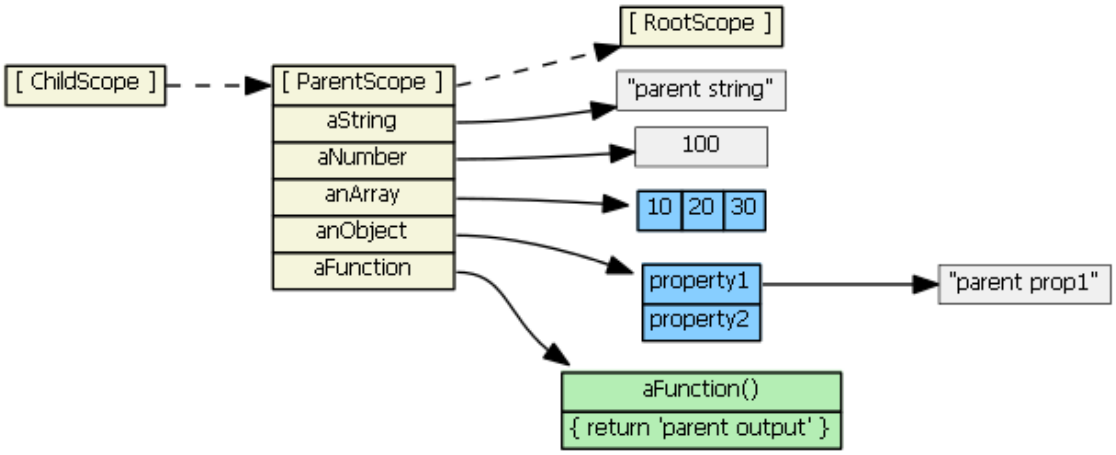
1. Use `$parent.parentScopeProperty` in the child scope. This will prevent the child scope from creating its own property.
2. Define a function on the parent scope, and call it from the child, passing the primitive value up to the parent (not always possible)

The Details:

JavaScript Prototypal Inheritance

It is important to first have a solid understanding of JavaScript prototypal inheritance, especially if you are coming from a server-side background and you are more familiar with classical inheritance. So let's review that first.

Suppose parentScope has properties aString, aNumber, anArray, anObject, and aFunction. If childScope prototypically inherits from parentScope, we have:



(Note that to save space, I show the `anArray` object as a single blue object with its three values, rather than an single blue object with three separate gray literals.)

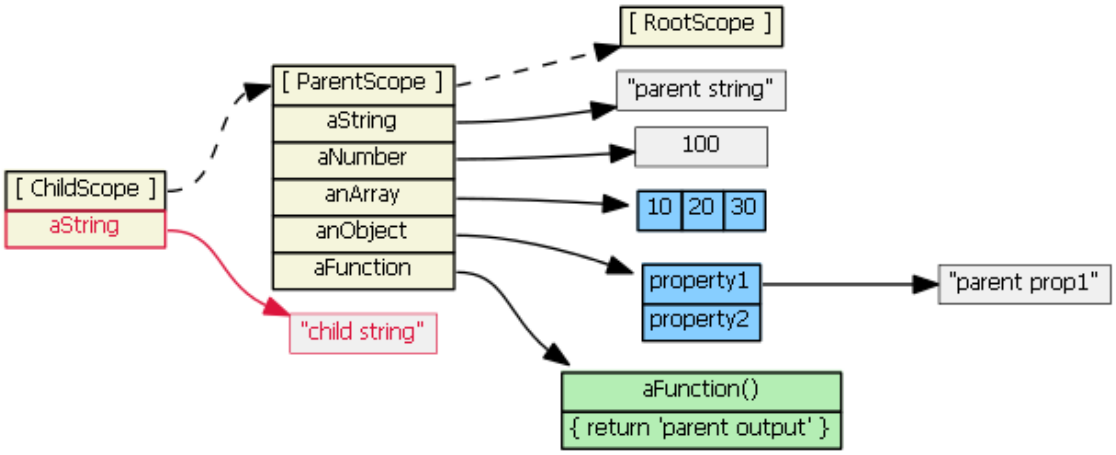
If we try to access a property defined on the parentScope from the child scope, JavaScript will first look in the child scope, not find the property, then look in the inherited scope, and find the property. (If it didn't find the property in the parentScope, it would continue up the prototype chain... all the way up to the root scope). So, these are all true:

```
childScope.aString === 'parent string'
childScope.anArray[1] === 20
childScope.anObject.property1 === 'parent prop1'
childScope.aFunction() === 'parent output'
```

Suppose we then do this:

```
childScope.aString = 'child string'
```

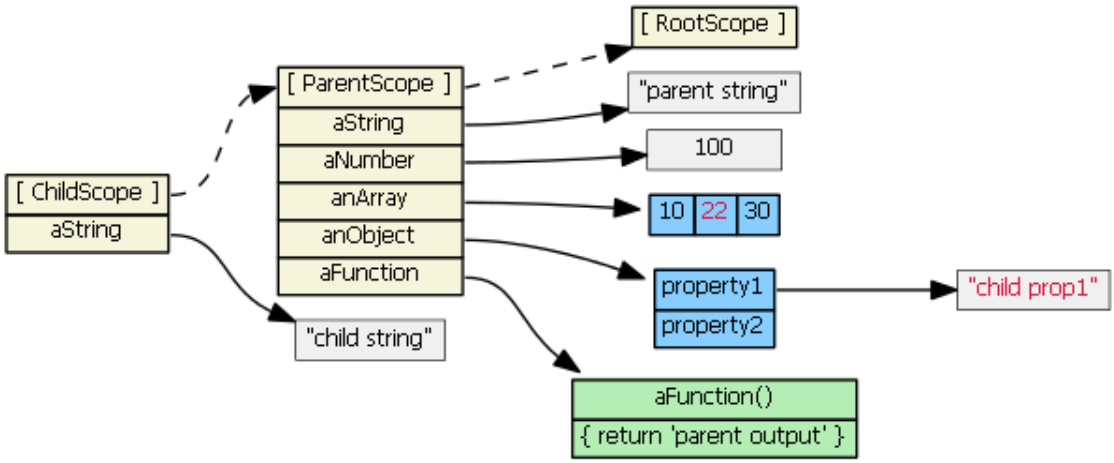
The prototype chain is not consulted, and a new aString property is added to the childScope. **This new property hides/shadows the parentScope property with the same name**. This will become very important when we discuss ng-repeat and ng-include below.



Suppose we then do this:

```
childScope.anArray[1] = 22
childScope.anObject.property1 = 'child prop1'
```

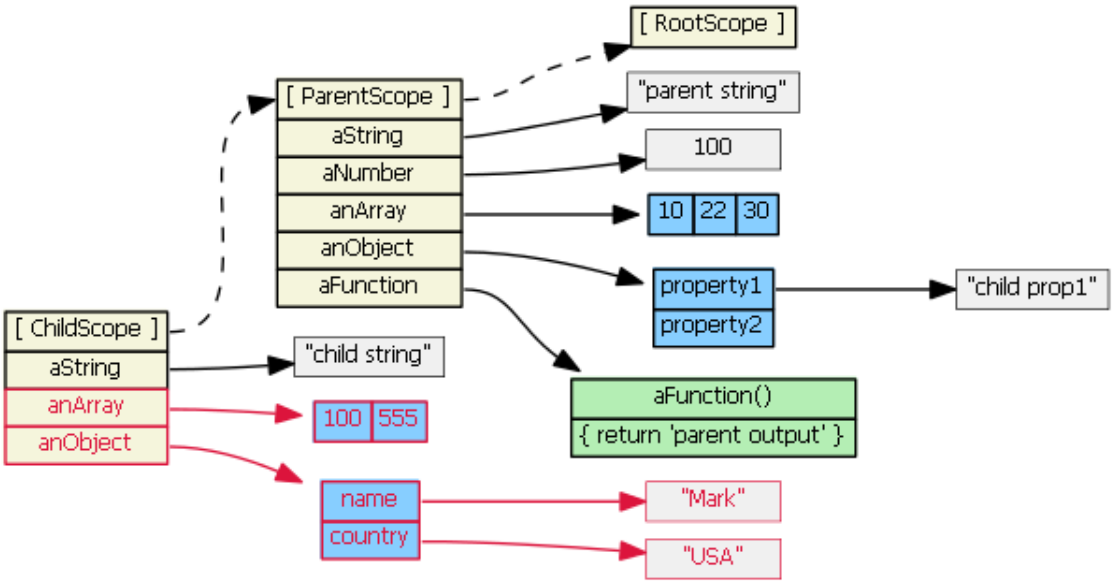
The prototype chain is consulted because the objects (anArray and anObject) are not found in the childScope. The objects are found in the parentScope, and the property values are updated on the original objects. No new properties are added to the childScope; no new objects are created. (Note that in JavaScript arrays and functions are also objects.)



Suppose we then do this:

```
childScope.anArray = [100, 555]
childScope.anObject = { name: 'Mark', country: 'USA' }
```

The prototype chain is not consulted, and child scope gets two new object properties that hide/shadow the parentScope object properties with the same names.



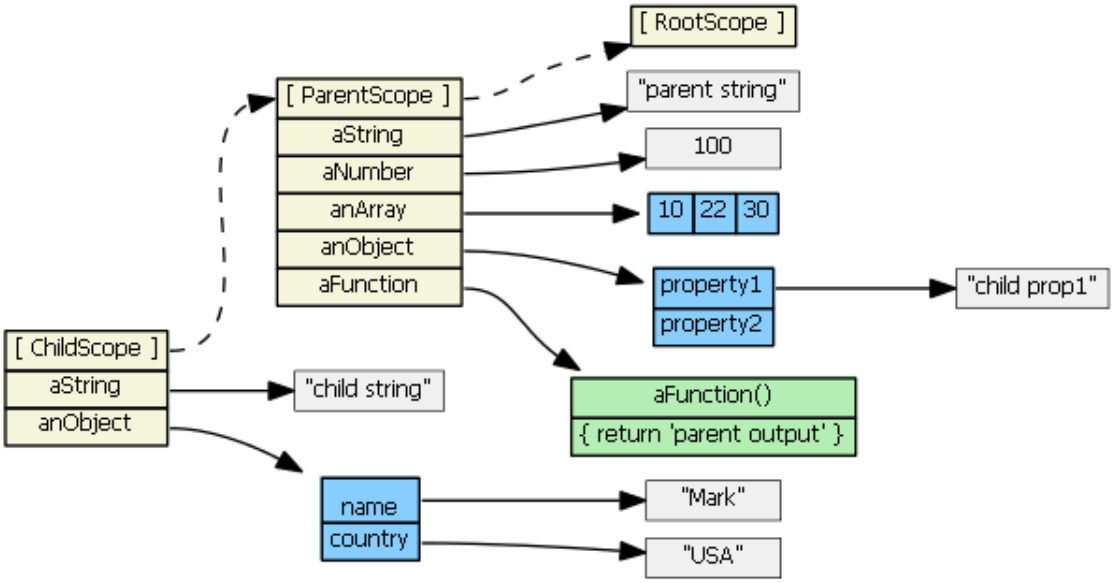
Takeaways:

- If we read childScope.propertyX, and childScope has propertyX, then the prototype chain is not consulted.
- If we set childScope.propertyX, the prototype chain is not consulted.

One last scenario:

```
delete childScope.anArray
childScope.anArray[1] === 22  // true
```

We deleted the childScope property first, then when we try to access the property again, the prototype chain is consulted.



Here is a jsfiddle where you can see the above javascript prototypical inheritance examples being modified and their result (open up your browser's console to see the output. The console output can be viewed as what the 'RootScope' would see).

Angular Scope Inheritance

The contenders:

- The following create new scopes, and inherit prototypically: ng-repeat, ng-include, ng-switch, ng-view, ng-controller, directive with `scope: true`, directive with `transclude: true`.
- The following creates a new scope which does not inherit prototypically: directive with `scope: { ... }`. This creates an "isolate" scope instead.

Note, by default, directives do not create new scope -- i.e., the default is `scope: false`.

ng-include

Suppose we have in our controller:

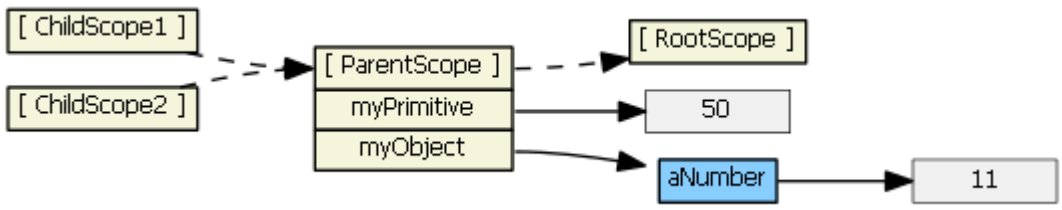
```
$scope.myPrimitive = 50;
$scope.myObject    = {aNumber: 11};
```

And in our HTML:

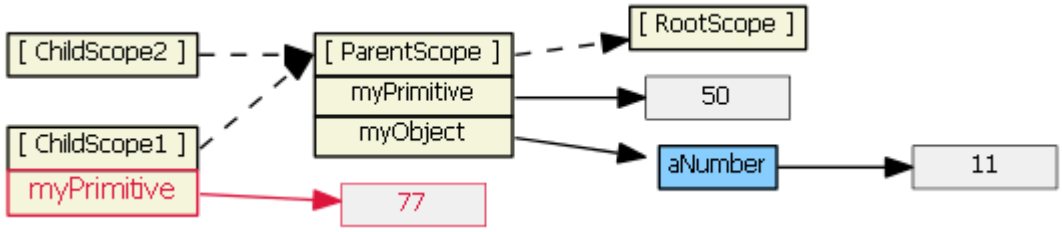
```
<script type="text/ng-template" id="/tpl1.html">
    <input ng-model="myPrimitive">
</script>
<div ng-include src="'/tpl1.html'"></div>

<script type="text/ng-template" id="/tpl2.html">
    <input ng-model="myObject.aNumber">
</script>
<div ng-include src="'/tpl2.html'"></div>
```

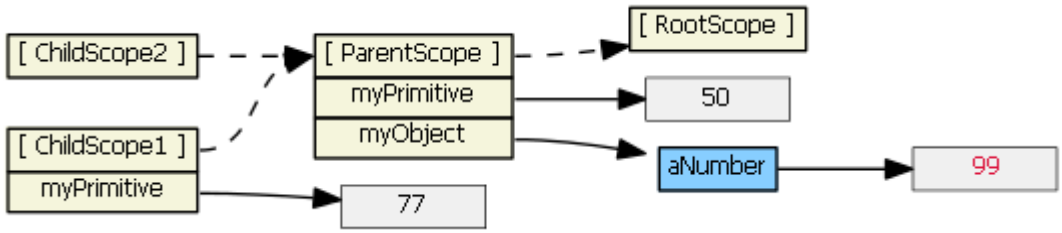
Each ng-include generates a new child scope, which prototypically inherits from the parent scope.



Typing (say, "77") into the first input textbox causes the child scope to get a new myPrimitive scope property that hides/shadows the parent scope property of the same name. This is probably not what you want/expect.



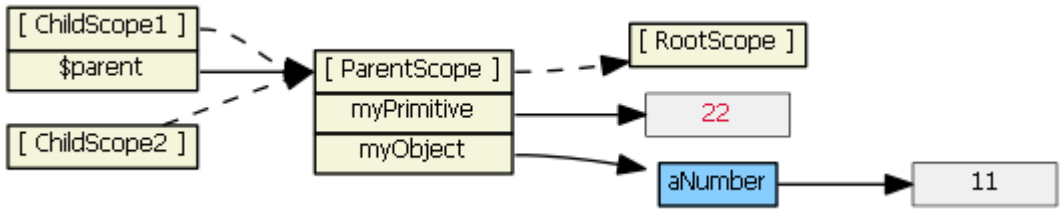
Typing (say, "99") into the second input textbox does not result in a new child property. Because tpl2.html binds the model to an object property, prototypal inheritance kicks in when the ngModel looks for object myObject -- it finds it in the parent scope.



We can rewrite the first template to use $parent, if we don't want to change our model from a primitive to an object:

```
<input ng-model="$parent.myPrimitive">
```

Typing (say, "22") into this input textbox does not result in a new child property. The model is now bound to a property of the parent scope (because $parent is a child scope property that references the parent scope).



For all scopes (prototypal or not), Angular always tracks a parent-child relationship (i.e., a hierarchy), via scope properties $parent, $$childHead and $$childTail. I normally don't show these scope properties in the diagrams.

For scenarios where form elements are not involved, another solution is to define a function on the parent scope to modify the primitive. Then ensure the child always calls this function, which will be available to the child scope due to prototypal inheritance. E.g.,

```
// in the parent scope
$scope.setMyPrimitive = function(value) {
    $scope.myPrimitive = value;
}
```

Here is a sample fiddle that uses this "parent function" approach. (This was part of a Stack Overflow post.)

See also http://stackoverflow.com/a/13782671/215945 and https://github.com/angular/angular.js/issues/1267.

ng-switch

ng-switch scope inheritance works just like ng-include. So if you need 2-way data binding to a primitive in the parent scope, use $parent, or change the model to be an object and then bind to a property of that object. This will avoid child scope hiding/shadowing of parent scope properties.

See also AngularJS, bind scope of a switch-case?

ng-repeat

Ng-repeat works a little differently. Suppose we have in our controller:

```
$scope.myArrayOfPrimitives = [ 11, 22 ];
$scope.myArrayOfObjects    = [{num: 101}, {num: 202}]
```

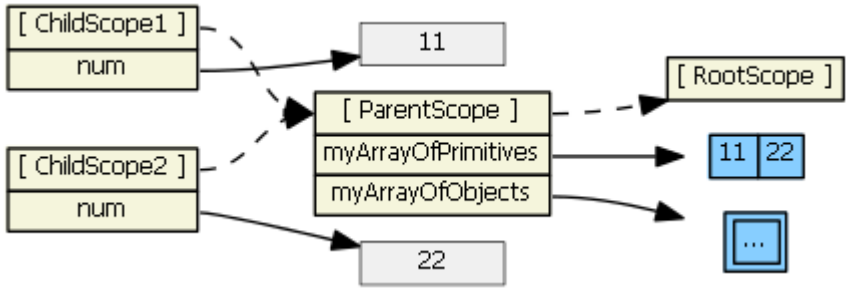
And in our HTML:

```
<ul><li ng-repeat="num in myArrayOfPrimitives">
       <input ng-model="num"></input>
    </li>
</ul>
<ul><li ng-repeat="obj in myArrayOfObjects">
       <input ng-model="obj.num"></input>
    </li>
</ul>
```

For each item/iteration, ng-repeat creates a new scope, which prototypically inherits from the parent scope, **but it also assigns the item's value to a new property on the new child scope**. (The name of the new property is the loop variable's name.) Here's what the Angular source code for ng-repeat actually is:

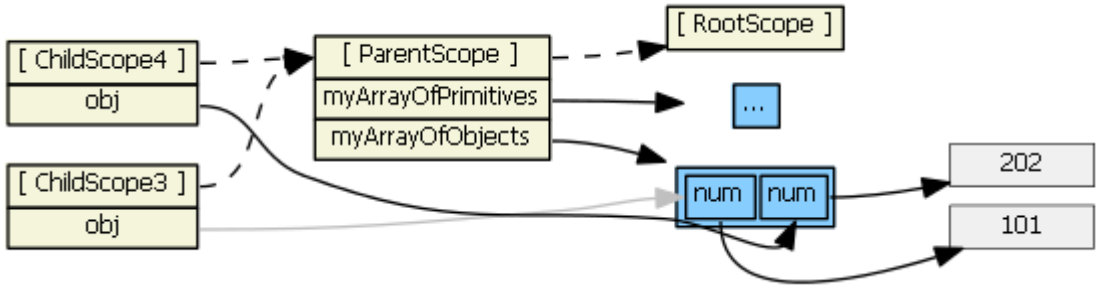
```
childScope = scope.$new(); // child scope prototypically inherits from parent scope
...
childScope[valueIdent] = value; // creates a new childScope property
```

If item is a primitive (as in myArrayOfPrimitives), essentially a copy of the value is assigned to the new child scope property. Changing the child scope property's value (i.e., using ng-model, hence child scope property `num`) does **not** change the array the parent scope references. So in the first ng-repeat above, each child scope gets a `num` property that is independent of the myArrayOfPrimitives array:



This ng-repeat will not work (like you want/expect it to). In Angular 1.0.2 or earlier, typing into the textboxes changes the values in the gray boxes, which are only visible in the child scopes. In Angular 1.0.3+, typing into the text boxes has no effect. (See Artem's explanation as to why on StackOverflow.) What we want is for the inputs to affect the myArrayOfPrimitives array, not a child scope primitive property. To accomplish this, we need to change the model to be an array of objects.

So, if item is an object, a reference to the original object (not a copy) is assigned to the new child scope property. Changing the child scope property's value (i.e., using ng-model, hence `obj.num`) **does** change the object the parent scope references. So in the second ng-repeat above, we have:



(I colored one line gray just so that it is clear where it is going.)

This works as expected. Typing into the textboxes changes the values in the gray boxes, which are visible to both the child and parent scopes.

See also Difficulty with ng-model, ng-repeat, and inputs and
ng-repeat and databinding

ng-view

TBD, but I think it acts just like ng-include.

ng-controller

Nesting controllers using ng-controller results in normal prototypal inheritance, just like ng-include and ng-switch, so the same techniques apply. However, "it is considered bad form for two controllers to share information via $scope inheritance" -- http://onehungrymind.com/angularjs-sticky-notes-pt-1-architecture/ A service should be used to share data between controllers instead.

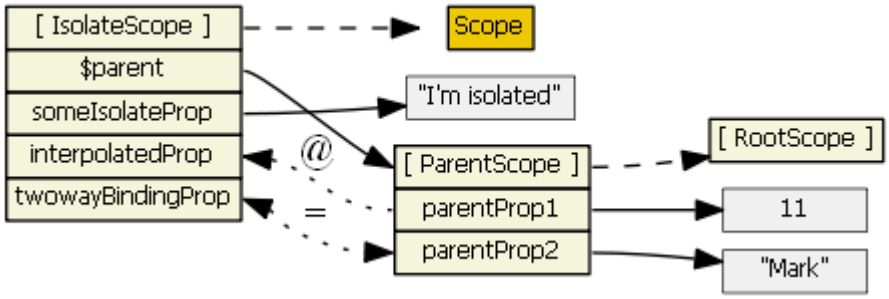
(If you really want to share data via controllers scope inheritance, there is nothing you need to do. The child scope will have access to all of the parent scope properties. See also Controller load order differs when loading or navigating)

directives

1. default (`scope: false`) - the directive does not create a new scope, so there is no inheritance here. This is easy, but also dangerous because, e.g., a directive might think it is creating a new property on the scope, when in fact it is clobbering an existing property. This is not a good choice for writing directives that are intended as reusable components.
2. `scope: true` - the directive creates a new child scope that prototypically inherits from the parent scope. If more than one directive (on the same DOM element) requests a new scope, only one new child scope is created. Since we have "normal" prototypal inheritance, this is like ng-include and ng-switch, so be wary of

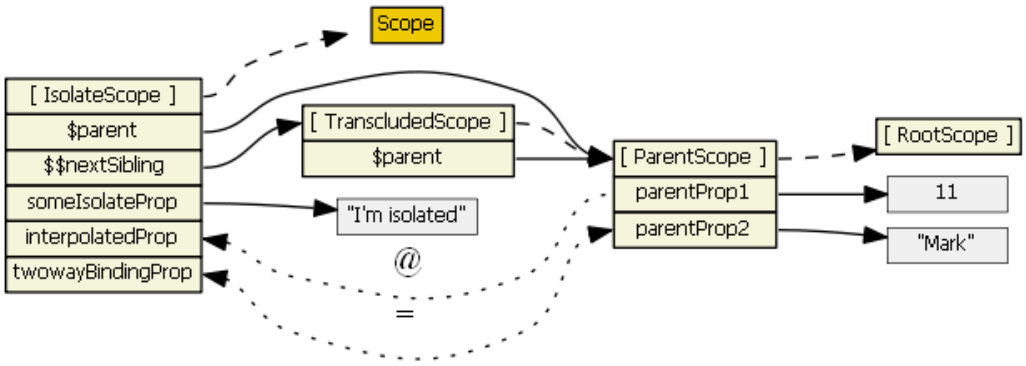
2-way data binding to parent scope primitives, and child scope hiding/shadowing of parent scope properties.

3. `scope: { ... }` - the directive creates a new isolate/isolated scope. It does not prototypically inherit. This is usually your best choice when creating reusable components, since the directive cannot accidentally read or modify the parent scope. However, such directives often need access to a few parent scope properties. The object hash is used to set up two-way binding (using '=') or one-way binding (using '@') between the parent scope and the isolate scope. There is also '&' to bind to parent scope expressions. So, these all create local scope properties that are derived from the parent scope. Note that attributes are used to help set up the binding -- you can't just reference parent scope property names in the object hash, you have to use an attribute. E.g., this won't work if you want to bind to parent property `parentProp` in the isolated scope: `<div my-directive>` and `scope: { localProp: '@parentProp' }`. An attribute must be used to specify each parent property that the directive wants to bind to: `<div my-directive the-Parent-Prop=parentProp>` and `scope: { localProp: '@theParentProp' }`.
   Isolate scope's `__proto__` references a Scope object. Isolate scope's $parent references the parent scope, so although it is isolated and doesn't inherit prototypically from the parent scope, it is still a child scope.
   For the picture below we have
   `<my-directive interpolated="{{parentProp1}}" twowayBinding="parentProp2">` and `scope: { interpolatedProp: '@interpolated', twowayBindingProp: '=twowayBinding' }`
   Also, assume the directive does this in its linking function: `scope.someIsolateProp = "I'm isolated"`



   One final note: use `attrs.$observe('attr_name', function(value) { ... })` in the linking function to get the interpolated value of isolate scope properties that use the '@' notation. E.g., if we have this in the linking function -- `attrs.$observe('interpolated', function(value) { ... })` -- `value` would be set to 11. (`scope.interpolatedProp` is undefined in the linking function. In contrast, `scope.twowayBindingProp` is defined in the linking function, since it uses the '=' notation.)
   For more information on isolate scopes see http://onehungrymind.com/angularjs-sticky-notes-pt-2-isolated-scope/
4. `transclude: true` - the directive creates a new "transcluded" child scope, which prototypically inherits from the parent scope. ~~So if your transcluded content (i.e., the stuff that ng-transclude will be replaced with) requires 2-way data binding to a primitive in the parent scope, use $parent, or change the model to be an object and then bind to a property of that object. This will avoid child scope hiding/shadowing of parent scope properties.~~
   ~~The transcluded and the isolated scope (if any) are siblings -- the $parent property of each scope references the same parent scope. When a transcluded and an isolate scope both exist, isolate scope property $$nextSibling will reference the transcluded scope.~~
   ~~For more information on transcluded scopes, see AngularJS two way binding not working in directive with transcluded scope~~

~~For the picture below, assume the same directive as above with this addition: `transclude: true`~~



. The former information is outdated since v1.3. Isolated scope is the $parent of transcluded scope now. See Why ng-transclude's scope is not a child of its directive's scope - if the directive has an isolated scope? for more detail.

This fiddle has a `showScope()` function that can be used to examine an isolate scope and its associated transcluded scope. See the instructions in the comments in the fiddle.

Summary

There are four types of scopes:

1. normal prototypal scope inheritance -- ng-include, ng-switch, ng-controller, directive with `scope: true`
2. normal prototypal scope inheritance with a copy/assignment -- ng-repeat. Each iteration of ng-repeat creates a new child scope, and that new child scope always gets a new property.
3. isolate scope -- directive with `scope: {...}`. This one is not prototypal, but '=', '@', and '&' provide a mechanism to access parent scope properties, via attributes.
4. transcluded scope -- directive with `transclude: true`. This one is also normal prototypal scope inheritance, but it is also a sibling of any isolate scope.

For all scopes (prototypal or not), Angular always tracks a parent-child relationship (i.e., a hierarchy), via properties $parent and $$childHead and $$childTail.

Diagrams were generated with GraphViz "*.dot" files, which are on github. Tim Caswell's "Learning JavaScript with Object Graphs" was the inspiration for using GraphViz for the diagrams.

The above was originally posted on StackOverflow.