Angular Expressions

Angular expressions are JavaScript-like code snippets that are usually placed in bindings such as `{{ expression }}`.

For example, these are valid expressions in Angular:

- `1+2`
- `a+b`
- `user.name`
- `items[index]`

Angular Expressions vs. JavaScript Expressions

Angular expressions are like JavaScript expressions with the following differences:

- **Context:** JavaScript expressions are evaluated against the global `window`. In Angular, expressions are evaluated against a `scope`object.
- **Forgiving:** In JavaScript, trying to evaluate undefined properties generates `ReferenceError` or `TypeError`. In Angular, expression evaluation is forgiving to `undefined` and `null`.
- **No Control Flow Statements:** you cannot use the following in an Angular expression: conditionals, loops, or exceptions.
- **Filters:** You can use filters within expressions to format data before displaying it.

Example.

```
1+2={{1+2}}
```

Context

Angular does not use JavaScript's `eval()` to evaluate expressions. Instead Angular's $parse service processes these expressions.

Unlike JavaScript, where names default to global `window` properties, Angular expressions must use `$window` explicitly to refer to the global `window` object. For example, if you want to call `alert()` in an expression you must use `$window.alert()`. This restriction is intentional. It prevents accidental access to the global state – a common source of subtle bugs.

```
function Cntl1($window, $scope){
  $scope.name = 'World';


  $scope.greet = function() {
    $window.alert('Hello ' + $scope.name);
  };
}
-------------------------------------------------------------------------
<div class="example2" ng-controller="Cntl1">
  Name: <input ng-model="name" type="text"/>
  <button ng-click="greet()">Greet</button>
</div>
```

No Control Flow Statements

Apart from the ternary operator (`a ? b : c`), you cannot write a control flow statement in an expression. The reason behind this is core to the Angular philosophy that application logic should be in controllers, not the views. If you need a real conditional, loop, or to throw from a view expression, delegate to a JavaScript method instead.

One-time binding

An expression that starts with `::` is considered a one-time expression. One-time expressions will stop recalculating once they are stable, which happens after the first digest if the expression result is a non-undefined value

```
<div ng-controller="EventController">
  <button ng-click="clickMe($event)">Click Me</button>
  <p id="one-time-binding-example">One time binding:
{{::name}}</p>
  <p id="normal-binding-example">Normal binding: {{name}}</p>
</div>
```

```
angular.module('oneTimeBidingExampleApp', []).
  controller('EventController', ['$scope', function($scope) {
    var counter = 0;
    var names = ['Igor', 'Misko', 'Chirayu', 'Lucas'];
    /*
     * expose the event object to the scope
     */
    $scope.clickMe = function(clickEvent) {
      $scope.name = names[counter % names.length];
      counter++;
    };
  }]);
```

Why this feature

The main purpose of one-time binding expression is to provide a way to create a binding that gets deregistered and frees up resources once the binding is stabilized. Reducing the number of expressions being watched makes the digest loop faster and allows more information to be displayed at the same time.

How to benefit from one-time binding

When interpolating text or attributes. If the expression, once set, will not change then it is a candidate for one-time expression.

```
  <div name="attr: {{::color}}">text: {{::name}}</div>
```

When using a directive with bidirectional binding and the parameters will not change

```
someModule.directive('someDirective', function() {
  return {
```

```
    scope: {
      name: '=',
      color: '@'
    },
    template: '{{name}}: {{color}}'
  };
});
  <div some-directive name="::myName" color="My color is
{{::myColor}}"></div>
```

When using a directive that takes an expression

```
<ul>
  <li ng-repeat="item in ::items">{{item.name}};</li>
</ul>
```