COEN 168/268

Mobile Web Application Development

Ember.js Object Model

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

The lecture contents is mostly from the Ember Guides available under the MIT license

Starting at: http://emberjs.com/guides/object-model/classes-and-instances/

Classes and Instances

Ember 'Emulates' Classes in JavaScript

Defining a new Ember *class*

Use `Ember.Object.extend()`:

```
1 App.Person = Ember.Object.extend({
2   say: function(thing) {
3     alert(thing);
4   }
5 });
```

This defines a new `App.Person` class with a `say()` method.

Subclassing Existing Classes

Use `extend()` on any Ember class definitions:

```
1 App.PersonView = Ember.View.extend({
2   tagName: 'li',
3   classNameBindings: ['isAdministrator']
4 });
```

When Subclassing, You Can Override Methods And Still Call Parent Class Using `_super()`

```
App.Person = Ember.Object.extend({
  say: function(thing) {
    var name = this.get('name');
    alert(name + " says: " + thing);
  }
});
App.Soldier = App.Person.extend({
  say: function(thing) {
    this._super(thing + ", sir!");
  }
});

var yehuda = App.Soldier.create({
  name: "Yehuda Katz"
});

yehuda.say("Yes"); // alerts "Yehuda Katz says: Yes, sir!"
```

Creating Instances

Use `create()` to create instance objects of Ember Classes:

```
1 var person = App.Person.create();
2 person.say("Hello"); // alerts " says: Hello"
```

Passing In Default Instance Properties To A Class Constructor

Pass an optional hash to the `create()` method:

```
 1 App.Person = Ember.Object.extend({
 2   helloWorld: function() {
 3     alert("Hi, my name is " + this.get('name'));
 4   }
 5 });
 6
 7 var tom = App.Person.create({
 8   name: "Tom Dale"
 9 });
10
11 tom.helloWorld(); // alerts "Hi, my name is Tom Dale"
```

Performance Reasons

- For performance reasons, note that you cannot redefine an instance's computed properties or methods when calling `create()`, nor can you define new ones.
- You should only set simple properties when calling `create()`. If you need to define or redefine methods or computed properties, create a new subclass and instantiate that.
- By convention, properties or variables that hold classes are PascalCased, while instances are not, like `App.Person`

Initializing Instances

- When a new instance is created, its `init` method is invoked.
- This is the ideal place to do setup required on new instances:

```
 1 App.Person = Ember.Object.extend({
 2   init: function() {
 3     var name = this.get('name');
 4     alert(name + ", reporting for duty!");
 5   }
 6 });
 7
 8 App.Person.create({
 9   name: "Stefan Penner"
10 });
11
12 // alerts "Stefan Penner, reporting for duty!"
```

Note About Subclassing and Overriding `init()`

- Make sure you call `this._super()`
- If you don't the parent class might not set up important things
- Will cause very strange behaviors that will be hard to debug

Use `get` and `set` when accessing properties

```
1 var person = App.Person.create();
2
3 var name = person.get('name');
4 person.set('name', "Tobias Fünke");
```

Make sure to use these accessor methods; otherwise, computed properties won't recalculate, observers won't fire, and templates won't update.

Computed Properties

What are Computed Properties?

- Computed properties let you declare functions as properties.
- You create one by defining a computed property as a function.
- Ember will automatically call this function when you ask for the property.
- You can then use it the same way you would any normal, static property.
- It's super handy for taking one or more normal properties and transforming or manipulating their data to create a new value.

Computed properties in action

```
App.Person = Ember.Object.extend({
  // these will be supplied by `create`
  firstName: null,
  lastName: null,

  fullName: function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});

var ironMan = App.Person.create({
  firstName: "Tony",
 lastName:  "Stark"
});

ironMan.get('fullName'); // "Tony Stark"
```

Computed properties in action, cont'd

Notice that the `fullName` function calls `property`. This declares the function to be a computed property, and the arguments tell Ember that it depends on the `firstName` and `lastName` attributes.

Whenever you access the `fullName` property, this function gets called, and it returns the value of the function, which simply calls `firstName + lastName`.

Alternate invocation

- At this point, you might be wondering how you are able to call the `.property` function on a function.
- This is possible because Ember extends the `function` prototype.
- Without the prototype override, you can do this:

```
1  fullName: Ember.computed('firstName', 'lastName', function() {
2    return this.get('firstName') + ' ' + this.get('lastName');
3  })
```

Chaining computed properties

- You can use computed properties as values to create new computed properties.
- Let's add a `description` computed property to the previous example, and use the existing `fullName` property and add in some other properties.

```
App.Person = Ember.Object.extend({
  firstName: null,
  lastName: null,
  age: null,
  country: null,

  fullName: function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName'),

  description: function() {
    return this.get('fullName') + '; Age: ' + this.get('age') + '; Country: ' + this.get('country');
  }.property('fullName', 'age', 'country')
});

var captainAmerica = App.Person.create({
  firstName: 'Steve',
  lastName: 'Rogers',
  age: 80,
 country: 'USA'
});

captainAmerica.get('description'); // "Steve Rogers; Age: 80; Country: USA"
```

Dynamic updating

- Computed properties, by default, observe any changes made to the properties they depend on and are dynamically updated when they're called.
- Let's use computed properties to dynamically update.

```
captainAmerica.set('firstName', 'William');

captainAmerica.get('description'); // "William Rogers; Age: 80; Country: USA"
```

So this change to `firstName` was observed by `fullName` computed property, which was itself observed by the `description` property.

Setting any dependent property will propagate changes through any computed properties that depend on them, all the way down the chain of computed properties you've created.

Setting Computed Properties

- You can also define what Ember should do when setting a computed property.
- If you try to set a computed property, it will be invoked with the key (property name), the value you want to set it to, and the previous value.

```
App.Person = Ember.Object.extend({
  firstName: null,
  lastName: null,

  fullName: function(key, value, previousValue) {
    // setter
    if (arguments.length > 1) {
      var nameParts = value.split(/\s+/);
      this.set('firstName', nameParts[0]);
      this.set('lastName',  nameParts[1]);
    }

    // getter
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});


var captainAmerica = App.Person.create();
captainAmerica.set('fullName', "William Burnside");
captainAmerica.get('firstName'); // William
captainAmerica.get('lastName'); // Burnside
```

Computed Properties and Aggregate Data With @each

Computed Properties and Aggregate Data With @each

- Often, you may have a computed property that relies on all of the items in an array to determine its value.
- For example, you may want to count all of the todo items in a controller to determine how many of them are completed.

An @each example

```
App.TodosController = Ember.Controller.extend({
  todos: [
    Ember.Object.create({ isDone: true }),
    Ember.Object.create({ isDone: false }),
    Ember.Object.create({ isDone: true })
  ],

  remaining: function() {
    var todos = this.get('todos');
    return todos.filterBy('isDone', false).get('length');
  }.property('todos.@each.isDone')
});
```

An `@each` example, cont'd

- Note here that the dependent key (`todos.@each.isDone`) contains the special key `@each`.
- This instructs Ember.js to update bindings and fire observers for this computed property when one of the following four events occurs.

An `@each` example, cont'd

1. The `isDone` property of any of the objects in the `todos` array changes.
2. An item is added to the `todos` array.
3. An item is removed from the `todos` array.
4. The `todos` property of the controller is changed to a different array.

A @each example, cont'd

In the example above, the `remaining` count is `1`:

```
1 App.todosController = App.TodosController.create();
2 App.todosController.get('remaining');
3 // 1
```

A @each example, cont'd

If we change the todo's `isDone` property, the `remaining` property is updated automatically:

```
 1 var todos = App.todosController.get('todos');
 2 var todo = todos.objectAt(1);
 3 todo.set('isDone', true);
 4
 5 App.todosController.get('remaining');
 6 // 0
 7
 8 todo = Ember.Object.create({ isDone: false });
 9 todos.pushObject(todo);
10
11 App.todosController.get('remaining');
12 // 1
```

A `@each` example, cont'd

Note that `@each` only works one level deep. You cannot use nested forms like `todos.@each.owner.name` or `todos.@each.owner.@each.name`.

Observers

Observers

- Ember supports observing any property, including computed properties.
- You can set up an observer on an object by using the `observes` method on a function.

```
Person = Ember.Object.extend({
  // these will be supplied by `create`
  firstName: null,
  lastName: null,

  fullName: function() {
    var firstName = this.get('firstName');
    var lastName = this.get('lastName');

    return firstName + ' ' + lastName;
  }.property('firstName', 'lastName'),

  fullNameChanged: function() {
    // deal with the change
  }.observes('fullName').on('init')
});

var person = Person.create({
  firstName: 'Yehuda',
  lastName: 'Katz'
});

person.set('firstName', 'Brohuda'); // observer will fire
```

Observers and asynchrony

- Observers in Ember are currently synchronous.
- This means that they will fire as soon as one of the properties they observe changes.
- Because of this, it is easy to introduce bugs where properties are not yet synchronized.

A Synchronization Bug

```
1 Person.reopen({
2   lastNameChanged: function() {
3     // The observer depends on lastName and so does fullName. Because observers
4     // are synchronous, when this function is called the value of fullName is
5     // not updated yet so this will log the old value of fullName
6     console.log(this.get('fullName'));
7   }.observes('lastName')
8 });
```

A Synchronization Bug, Cont'd

This synchronous behaviour can also lead to observers being fired multiple times when observing multiple properties:

```
1 Person.reopen({
2   partOfNameChanged: function() {
3     // Because both firstName and lastName were set, this observer will fire twice.
4   }.observes('firstName', 'lastName')
5 });
6
7 person.set('firstName', 'John');
8 person.set('lastName', 'Smith');
```

Fix These Issues with `Ember.run.once`

Ensures that any processing you need to do only happens once, and happens in the next run loop once all bindings are synchronized.

```
Person.reopen({
  partOfNameChanged: function() {
    Ember.run.once(this, 'processFullName');
  }.observes('firstName', 'lastName'),

  processFullName: function() {
    // This will only fire once if you set two properties at the same time, and
    // will also happen in the next run loop once all properties are synchronized
    console.log(this.get('fullName'));
  }
});

person.set('firstName', 'John');
person.set('lastName', 'Smith');
```

Observers and object initialization

- Observers never fire until after the initialization of an object is complete.
- If you need an observer to fire as part of the initialization process, you cannot rely on the side effect of set.
- Instead, specify that the observer should also run after init by using `.on('init')`:

Observers and object initialization, cont'd

```
1 App.Person = Ember.Object.extend({
2   init: function() {
3     this.set('salutation', "Mr/Ms");
4   },
5
6   salutationDidChange: function() {
7     // some side effect of salutation changing
8   }.observes('salutation').on('init')
9 });
```

Unconsumed Computed Properties Do Not Trigger Observers

- If you never `get` a computed property, its observers will not fire even if its dependent keys change.
- You can think of the value changing from one unknown value to another.
- This doesn't usually affect application code because computed properties are almost always observed at the same time as they are fetched.

Unconsumed Computed Properties Do Not Trigger Observers

- For example, you get the value of a computed property, put it in DOM (or draw it with D3), and then observe it so you can update the DOM once the property changes.
- If you need to observe a computed property but aren't currently retrieving it, just get it in your init method.

Without prototype extensions

You can define inline observers by using the `Ember.observer` method if you are using Ember without prototype extensions:

```
1 Person.reopen({
2   fullNameChanged: Ember.observer('fullName', function() {
3     // deal with the change
4   })
5 });
```

Outside of class definitions

You can also add observers to an object outside of a class definition using `addObserver`:

```
1 person.addObserver('fullName', function() {
2   // deal with the change
3 });
```

Bindings

Bindings

- Creates a link between two properties such that when one changes, the other one is updated to the new value automatically.
- Can connect properties on the same object, or across two different objects.
- Unlike most other frameworks that include some sort of binding implementation, bindings in Ember.js can be used with any object, not just between views and models.

Creating a Two-way binding

The easiest way to create a two-way binding is to use a computed alias, that specifies the path to another object.

```
wife = Ember.Object.create({
  householdIncome: 80000
});

husband = Ember.Object.create({
  wife: wife,
  householdIncome: Ember.computed.alias('wife.householdIncome')
});

husband.get('householdIncome'); // 80000

// Someone gets raise.
husband.set('householdIncome', 90000);
wife.get('householdIncome'); // 90000
```

Bindings Do Not Update Immediately

- Ember waits until all of your application code has finished running before synchronizing changes.
- This is so you can change a bound property as many times as you'd like without worrying about the overhead of syncing bindings when values are transient.

One-Way Bindings

- A one-way binding only propagates changes in one direction.
- Often, one-way bindings are just a performance optimization.
- Sometimes one-way bindings are useful to achieve specific behaviour such as a default that is the same as another property but can be overriden
- (e.g. a shipping address that starts the same as a billing address but can later be changed)

```
user = Ember.Object.create({
  fullName: "Kara Gates"
});

userView = Ember.View.create({
  user: user,
  userName: Ember.computed.oneWay('user.fullName')
});

// Changing the name of the user object changes
// the value on the view.
user.set('fullName', "Krang Gates");
// userView.userName will become "Krang Gates"

// ...but changes to the view don't make it back to
// the object.
userView.set('userName', "Truckasaurus Gates");
user.get('fullName'); // "Krang Gates"
```

Reopening Classes and Instances

Reopening Classes and Instances

- You don't need to define a class all at once.
- You can reopen a class and define new properties using the `reopen` method.

```
1 Person.reopen({
2   isPerson: true
3 });
4
5 Person.create().get('isPerson') // true
```

You can also override existing methods using `reopen`

When using `reopen`, you can also override existing methods and call `this._super`.

```
1 Person.reopen({
2   // override `say` to add an ! at the end
3   say: function(thing) {
4     this._super(thing + "!");
5   }
6 });
```

You can also add instance methods and properties

- `reopen` is used to add instance methods and properties that are shared across all instances of a class.
- It does not add methods and properties to a particular instance of a class as in vanilla JavaScript (without using prototype).
- But when you need to create class methods or add properties to the class itself you can use `reopenClass`.

```
Person.reopenClass({
  createMan: function() {
    return Person.create({isMan: true})
  }
});

Person.createMan().get('isMan') // true
```

Bindings, Observers, Computed Properties: What Do I Use When?

Computed Properties

- Use *computed properties* to build a new property by synthesizing other properties.
- Computed properties should not contain application behavior, and should generally not cause any side-effects when called.
- Except in rare cases, multiple calls to the same computed property should always return the same value (unless the properties it depends on have changed, of course.)

Observers

- *Observers* should contain behavior that reacts to changes in another property.
- Observers are especially useful when you need to perform some behavior after a binding has finished synchronizing.

Bindings

- *Bindings* are most often used to ensure objects in two different layers are always in sync.
- For example, you bind your views to your controller using Handlebars.

The lecture contents is mostly from the Ember Guides available under the MIT license

Starting at: http://emberjs.com/guides/object-model/classes-and-instances/

COEN 168/268

Mobile Web Application Development

Ember.js Object Model

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University