220

Creating Objects and Managing Dependencies

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley #doc220

In this lecture we look at patterns for creating objects. Object-oriented languages all necessarily give us ways to create objects, but there are various patterns that we can apply to improve clarity, reliability or flexibility.

Making New Objects

```
public class Thing {

  private final String msg;

  public Thing(String msg) {
    this.msg = msg ;
  }

  public void doSomething() {
    System.out.println(msg);
  }

}

new Thing("Hello World!").doSomething();
```

#doc220

Creating objects in Java should be familiar to you. We use a class to define a type, and when we use the new operator we stamp out a new object based on the template of the class. We use a constructor function if we want to perform any initialisation, say of fields, and we may pass in parameters to the constructor to help with this.

Making New Objects

```
class Thing

  def initialize(msg)
    @msg = "Hello World!"
  end

  def do_something
    puts @msg
  end

end

Thing.new("Hello World!").do_something
```

#doc220

Ruby is very similar. Again we define a class. We have a constructor in the form of the initalize method, which is invoked when we use the new operator. Note that we write Thing.new rather than new Thing. Inside the constructor we can *create* a field (or *instance variable*) by assigning to the field, denoted by an @.

Making New Objects


```
var Thing = function(msg) {
  this.msg = msg;
  this.doSomething = function() {
    console.log(this.msg);
  };
};

new Thing("Hello World!").doSomething();
```

#doc220

JavaScript is a little bit different. JavaScript is based on objects - almost everything is an object - but there are no classes. In order to create an object we define a constructor function, which by convention is named with a capital letter. When we use the new operator, JavaScript creates an empty object, runs the constructor function in the context of that object (creating fields etc) and then returns the object.

Making New Objects


```
var Thing = function(msg) {
  this.doSomething = function() {
    console.log(msg);
  };
};

new Thing("Hello World!").doSomething();
```

more idiomatically, use a closure to hold msg

#doc220

We can assign parameters passed to the constructor to fields of the new object, but in fact it is not necessary or idiomatic to do so in JavaScript. In the example, we use the value of msg in a newly created function, and when we create this, JavaScript allocates some storage for any values that were in scope when the function was created, which can be used whenever this function is called later. This is called *closure*.

Factory

Define an interface for creating objects. We may also use factories to defer choice of which class to instantiate until runtime.

Factory methods are Item #1 in Josh Bloch's book *Effective Java*

#doc220

Sometimes it is not clear from a call to a constructor function what we will get back, or how the parameters that we pass will be interpreted. It would be helpful of we could increase the clarity of intention by giving the constructor function a name

Factory

Factory methods can have names, unlike constructors

```
class TimeZone {
  static TimeZone forId(String id) {...}
  static TimeZone forOffsetHours(int offset)
   { return new TimeZone(3); }
}

TimeZone.forId("Europe/London");
TimeZone.forOffsetHours(3);
```

instead of:

this version has a lot less **clarity of intention**

```
new TimeZone(3);
```

http://joda-time.sourceforge.net/apidocs/org/joda/time/DateTimeZone.html

#doc220

The can do this by applying the *factory* pattern. We can add a factory method to a class to use instead of a constructor. This is a named static method on a Java class that internally calls the constructor and returns the object. We may have many factory methods with different names. We can prevent use of the new operator with this class by making the constructors themselves private.

Factory

```
class TimeZone

  def initialize(offset)
    @offset = offset
  end

  def TimeZone.for_offset_hours(hours)
    TimeZone.new(hours)
  end

  def to_s
    "GMT + #{@offset}"
  end

end
```

#doc220

A similar pattern is possible in Ruby. We create the equivalent of a static method - a *class method* by using the name of the class as a prefix for the method name in the definition.

Factory


```
var TimeZone = function(offset) {
  this.offset = offset;
  this.toString = function() {
    return "GMT +" + this.offset;
  };
}

TimeZone.forOffsetHours = function(offset) {
  return new TimeZone(offset);
};
```

#doc220

In JavaScript we do not have a class to put the factory method on to, so instead we can attach it to the closest thing we have - the constructor function. In JavaScript functions are themselves objects, and so can have fields and other functions defined on them. Here we define a function as a property of the constructor function to use as a factory method.

Factory

more idiomatically, use a closure to hold offset

```
var TimeZone = function(offset) {
   this.toString = function() {
    return "GMT +" + offset;
  };
}

TimeZone.forOffsetHours = function(offset) {
  return new TimeZone(offset);
};
```

#doc220

Again, a more idiomatic approach to this in JavaScript is to use a closure to store the parameter passed in implicitly, rather than assigning it to a field of the new object.

Factory

We can decide which type to return at run-time

can return any sub-type of Logo, polymorphically

```
class LogoFactory {
  static Logo createLogo() {
    if (config.country().equals(Country.UK) {
      return new FlagLogo("Union Jack");
    }
    if (config.country().equals(Country.USA) {
      return new FlagLogo("Stars and Stripes");
    }
    return new DefaultLogo();
  }

  class FlagLogo implements Logo {...}
  class DefaultLogo implements Logo {...}
}
```

#doc220

Another use for a factory is to defer the decision about the exact type of the object that is to be created until runtime. We can use some property only known or calculated at runtime (such as here the country in which the application is running) to determine which subtype of the required type to create and return for the client to use polymorphically.

Builder

Separate the construction of a complex object from its representation.

Consider a builder when faced with many constructor parameters

(from Joshua Bloch's Effective Java)

#doc220

An alternative to the factory is a *builder*. A builder is an object separate from the one being created that takes responsibility for gathering configuration parameters, and then, once all of that information has been collected, producing a fully formed object in a valid state.

Builder

```
public class BananaBuilder {

    private double ripeness = 0.0;
    private double curve = 0.5;

    private BananaBuilder() {}

    public static BananaBuilder aBanana() {
        return new BananaBuilder();
    }

    public Banana build() {
        Banana banana = new Banana(curve);
        banana.ripen(ripeness);
        return banana;
    }

    public BananaBuilder withRipeness(double ripeness){
        this.ripeness = ripeness;
        return this;
    }

    public BananaBuilder withCurve(double curve) {
        this.curve = curve;
        return this;
    }
}
```

private constructor enforces that builder itself is created using factory method

configuration methods return 'this', giving a **fluent interface** which allows method chaining

Example from http://www.natpryce.com/articles/000769.html
http://martinfowler.com/bliki/FluentInterface.html

#doc220

We call the *with* methods to set the values that we want for the properties, although they may have default values, and then call the build() method at the end to create the object. This helps us by always giving us an object in a fully constructed valid state. It gives an alternative to having many constructor parameters, many of which the client may not care about in a different scenario. In this example we have also used a factory method to create the builder.

Builder

statically importing factory method from the builder class

call the build() method at the end

```
import static BananaBuilder.*;

public class FruitBasket {

  private Collection<Fruit> basket = new ArrayList<Fruit>();

  public FruitBasket() {

    Banana banana = aBanana().withRipeness(2.0).withCurve(0.9).build();

    basket.add(banana);

    ...

  }
}
```

more expressive code style supported by the **fluent interface**

Example from http://www.natpryce.com/articles/000769.html
http://martinfowler.com/bliki/FluentInterface.html

#doc220

In the usage example we see how we can chain the calls to the *with* methods together, as each of them returns the receiver this. This is known as a fluent interface. At the end of the chain we call build() to create the object.

Singleton

Ensure a class only has one instance and provide a global point of access to it.

Careful: This is the most over-used and abused pattern from the GoF.

#doc220

The aim of the singleton pattern is to ensure that we only have one instance of a particular object in our system. In practice this is a much overused pattern, and we should be careful where we use it.

Singleton

Most of the time you don't **require** that you only have one of a certain type of object, you just happen to have only one.

Think about the Highlander - "there can be only one"

http://code.google.com/p/google-singleton-detector/wiki/WhySingletonsAreControversial
Matt Stephens: Perils of the Singleton http://www.softwarereality.com/design/singleton.jsp
Kevlin Henney: Parameterise from Above: Deglobalisation: http://accu.org/index.php/journals/1470
Mark Radford: Singleton - the anti-pattern! http://accu.org/index.php/journals/337
John Vlissides "To Kill a Singleton" - http://www.research.ibm.com/designpatterns/pubs/ph-jun96.txt

http://niclasnilsson.se/articles/2006/04/14/the-highlander-pattern/

#doc220

A singleton, in the way it is described in the GoF book, is effectively a global variable. In general we don't like global variables as they are hard to reason about, and introduce dependencies into our system that span widely across the object graph. They make it difficult to reuse parts of our code, and also often make it hard to test. There are many articles, some referenced here, that explain the problems that a singleton can cause.

Singleton

if you really need to ensure that everyone is using the same object

```
public class BankAccountStore {

    private static BankAccountStore instance = new BankAccountStore();

    private Collection<BankAccount> accounts;

    private BankAccountStore() {
      // initialise accounts
      // set up data etc
    }

    public static BankAcountStore getInstance() {
        return instance;
    }

    public BankAccount lookupAccountById(int id) {
        ...
    }

    ...
}
```

static initialisation runs when class is loaded

private constructor enforces that no-one else can call `new BankAcountStore()`

clients call `getInstance()` method to gain access to the global instance

#doc220

If you do need to use a singleton, you can implement it in Java by having a class that guards the single instance, creates it once on initialisation, and allows other objects to get a reference to it through a static method typically named getInstance(). This static method can be called anywhere in your codebase, which is what makes this similar to a global variable. We make the constructor private to prevent clients using the new operator to create extra instances.

Singleton

... and if you care about lazy initialisation (and thread safety)

```
public class BankAccountStore {

    private static BankAccountStore instance;

    private Collection<BankAccount> accounts;

    private BankAccountStore() {
      // initialise accounts
      // set up big expensive data stores etc etc
    }

    public static synchronized BankAcountStore getInstance() {
        if (instance == null) {
            instance = new BankAccountStore();
        }
        return instance;
    }

    public BankAccount lookupAccountById(int id) {
        ...
    }

    ...
}
```

don't initialise the singleton on class load

make this method synchronized to avoid race condition: multiple threads creating multiple instances

create the instance first time anyone calls `getInstance()` i.e. as late as possible

#doc220

Sometimes we want to initialise the singleton lazily, perhaps because doing so it very expensive and we don't want to do it until we need it. In that case we can use this other design, but be careful here about what happens if two threads enter the getInstance() method at the same time. To prevent the race condition where multiple instances are created, we have to make this method synchronized.

Singleton

```
class BankAccountStore

  def initialize
    puts "setting up all the accounts"
  end

  @@instance = BankAccountStore.new

  def BankAccountStore.instance
    return @@instance
  end

  def lookup_account_by_id(id)
    # ....
  end

  private_class_method :new
end
```

class variable initialisation runs when class is loaded

clients call BankAccountStore.instance method to gain access to the instance

use this to stop anyone calling the constructor

#doc220

We can follow a similar pattern in Ruby, initialising the single instance when the class is defined, and assigning it to a class variable (double @). In this case we can make the constructor private using `private_class_method :new` after we have called it. No one else can call *new* on our class after this.

Singleton

```
require 'singleton'

class BankAccountStore
  include Singleton

  def initialize
    puts "set up all the accounts"
  end

end
```

use mixin singleton module to do this for us

clients still call BankAccountStore.instance method to gain access to the instance

class variable initialisation runs on first call of BankAccountStore.instance

#doc220

More usual if you want to use a singleton in Ruby is not to write this functionality yourself but to reuse it from a library. Here we include the singleton *mixin* which adds the single instance and the access method to our class. The singleton mixin is actually initialised lazily and takes care of thread safety.

Dependencies

```
class PushSwitch
  def initialize
    @light = Light.new
    @on = false
  end

  def press
    @on = ! @on
    if (@on)
      @light.on
    else
      @light.off
    end
  end
end

switch = PushSwitch.new
switch.press
switch.press
```

```
class Light
  def on
    puts "Shining"
  end

  def off
    puts "Dark"
  end
end
```

#doc220

One of the things we need to be careful of in our code is the degree of coupling that is present. If we have close coupling, we can make it more difficult to re-use our classes in different situations.

In the code above, `PushSwitch` creates an instance of `Light`. We mention the class `Light` by name inside the sourcecode of `PushSwitch`. This creates a dependency from `PushSwitch` to `Light`. We can't use a `PushSwitch` without a `Light`. But we might quite legitimately want to use our `PushSwitch` code in a different situation, to switch something different on and off. With this code, we can't do that.

Managing Dependencies

```ruby
class PushSwitch
  def initialize(light)
    @light = light
    @on = false
  end

  def press
    @on = ! @on
    if (@on)
      @light.on
    else
      @light.off
    end

  end
end

switch = PushSwitch.new(Light.new)
switch.press
```

```ruby
class Light
  def on
    puts "Shining"
  end

  def off
    puts "Dark"
  end
end
```

#doc220

With a small change, we can reduce the coupling and make the `PushSwitch` re-usable.

Instead of creating an instance of `Light` inside `PushSwitch`, we accept the light as a parameter to the constructor (the `initialize` method). We now no longer mention the class `Light` by name inside `PushSwitch`. This small change is powerful as it removes the dependency from `PushSwitch` to `Light`.

Duck Typing

```ruby
class PushSwitch
  def initialize(powered)
    @powered = powered
    @on = false
  end

  def press
    @on = ! @on
    if (@on)
      @powered.on
    else
      @powered.off
    end

  end
end

switch = PushSwitch.new(Fan.new)
switch.press
```

```ruby
class Fan
  def on
    puts "Rotating"
  end

  def off
    puts "Still"
  end
end
```

#doc220

Now, all that `PushSwitch` depends on is something that responds to the messages `on` and `off`. We can pass in any object we like to constructor of `PushSwitch` as long as it is capable of handling these messages. In the above code example we've changed from using a `Light` to using a `Fan`. The code works fine. All we did in `PushSwitch` was to rename the variables to something more general than `light`.

This code is in Ruby, which is dynamically typed. Types aren't checked at compile time, so we can pass whatever object of whatever type we like to the constructor of `PushSwitch`. It just needs to support the relevant messages. As far as the `PushSwitch` is concerned, anything that supports `on` and `off` is fine. This is known as *duck typing*.

Interfaces as Roles

```ruby
class PushSwitch {

  private final Switchable powered;
  private boolean on = false;

  PushSwitch(Switchable powered) {
    this.powered = powered;
  }

  void press() {
    on = ! on;
    if (on) {
      powered.on();
    } else {
      powered.off();
    }
  }

}
```

```ruby
interface Switchable {
  public void on();
  public void off();
}
```

```ruby
class Fan implements Switchable {
  public void on() {
    System.out.println("Rotating");
  }
  public void off() {
    System.out.println("Still");
  }
}
```

#doc220

In Java we would like to have the same absence of coupling, but we don't have quite the flexibility in terms of typing - Java is statically typed.

We have identified that what a `PushSwitch` needs is something that responds to the messages `on` and `off`. We can use this knowledge to create an interface that expresses this role. Here we've called this interface `Switchable`. Now we can pass in any object we like to constructor of `PushSwitch` as long as it implements the `Switchable` interface.

Again we've reduced the coupling and improved the possibility of re-use for `PushSwitch`. We've used the an interface to define a role that needs to be played by `PushSwitch`'s collaborators.

Singleton creates Dependency

```
class BankTransfer {

  private Money amount;
  private AccountNumber fromAccNum;
  private AccountNumber toAccNum;

  public void execute() {

    BankAccountStore accountStore = BankAccountStore.getInstance();
    Account from = accountStore.accountWithNumber(fromAccNum);
    Account to = accountStore.accountWithNumber(toAccNum);

    // ...
  }

}
```

#doc220

If we revisit the singleton example, we see why this can cause problems. We access the singleton directly in the middle of our code, mentioning it by its class name, and creating tight coupling between the two classes. Using a singleton in this way can make it very difficult to re-use the calling code, or to swap out the implementation of the collaborators.