Introduction and Prerequisites
Version Check

Introduction

So you've just been in an interview and someone asked you to describe a design pattern or maybe you were just describing a problem to a coworker and they said it sounds like you're reinventing the wheel. They might have even said to you that it sounds like you're describing a factory or a singleton. If this sounds like a situation that you've been in, then this course is for you. Our focus in this course is on presenting the design patterns described in the Gang of Four in an example-driven way using Java to do so. Hi. I'm Brian Hansen, and welcome to Creational Design Patterns using Java.

Why Learn Patterns?

You might ask yourself why design patterns are important. I first learned patterns as a means of communicating a problem to another developer. You may very well already know how to solve a particular problem and it might follow the structure of a pattern, but it's better to have a common vocabulary that you can explain to someone what that problem is. Patterns are an abstract topic, it isn't something that you look at the concept and then have it memorized and know how to apply it from there. Some patterns are more easily applied to particular problems than another, so I suggest revisiting pattern material as you develop more and more programming experience. Whether it be this course or a book or something else, it is amazing how your perception of a pattern will change after gaining more experience with it. You might come away with a different understanding of it after applying it through different coding practices. A lot of people have used a singleton and think they have used design patterns, but there is much more than just a singleton out there.

Pattern Classifications

The Gang of Four breaks patterns out into three groups. Those groups are creational, structural, and behavioral. This course is going to focus on patterns classified under the creational group. Creational parts are focused on, as you might guess, how objects are created. There's a lot more than just replacing the keyword new, though, when talking about creational patterns. The two remaining groups, structural and behavioral, are covered in two separate courses much like this one.

Which Patterns?

The creational patterns that we are going to cover discussed by the gang of four are as follows, the Singleton pattern, the Builder pattern, a Prototype pattern, the Factory, and then finally, the AbstractFactory. We will implement all of these patterns and compare and contrast them to one another and described when the best case to use each individual one is.

How Do We Learn Them?

Each module on this course is going to be structured to where we first give an overview of what the pattern is, then the concepts when choosing to use this pattern, what you need to consider is part of the design, we will cover a live example of how it's used in the Java API, then we're going to go through a demo where you end up coding your own design pattern. We'll discuss the pitfalls of this pattern, contrast it with another pattern, and then give a summary, an overview, of what we covered and how this pattern applies in day-to-day use.

Prerequisites

The prerequisites for this course are actually quite simple. I am using Java 7, but Java 7 or 8 will work just fine. And then I'm going to be using Spring STS, which is just a flavor of Eclipse. Honestly, though, this code will work with any IDE out there. If you prefer IntelliJ over Eclipse or Spring STS over Eclipse, any of them are going to run just fine. You could even run the code in a simple text editor and compile it on the command line as we don't have a very complex package structure and it should work just fine. I like Spring STS because it seems to work and be a little bit more stable than Eclipse, but it's really dealer's choice. You can choose to use whatever you want.

Next

So that covers the introduction and prerequisites for this course. Let's go ahead and get started by now looking at the singleton pattern.

Singleton Pattern

Introduction

Hi. This is Bryan Hansen, and in this module, we are going to look at the singleton design pattern. The singleton pattern is one of, if not the most, heavily used design patterns because of its simplicity to implement and the type of problem that it solves.

Concepts

The concepts when choosing a singleton are that it guarantees only one instance is going to be created. It also guarantees the control of a resource. Since this is a creational design pattern, the instantiation of it is all controlled through the implementation of the pattern. Although it doesn't have to be, it is usually lazily loaded. This ties in nicely with it being a creational pattern. Examples of this in the Java API or commonly used frameworks are the runtime environment, logger, but depending on the implementation this could be factory instead of singleton, and we will discuss this in more detail later, Spring Beans, if you've used the Spring framework at all, you will quickly learn that all Spring Beans are by default singletons. And a fourth example are graphics managers. Typically when you're using a

Graphics API of any kind, you're going to get an instance of your graphical environment, and we only want one of those instances at a time.

Design Considerations

The singleton is responsible for creating itself and managing its lifecycle. It is static in nature, although it is not implemented using a static class typically. The reason for not using a static class is that it needs to be thread safe, and static doesn't necessarily guarantee this for us. There is a private instance of a singleton, hence the minus sign or hyphen in the UML. There is also a private constructor that is marked the same way. This is because we want the singleton itself to call the constructor and nobody else. There are no parameters, and if you require parameters, that is typically a factory pattern and violates the rules of a singleton.

Example: Runtime

This is a code snippet of an everyday example using a singleton. We get an instance of the runtime environment and then print out the object address for it just so we can visually see what it is. We then get another instance of it and print out the address of that object as well. We can see that both objects are using the same object address by using the equals operator to compare them, and it will print out that they are the same object.

Demo: Runtime

In this example, you can see that we're going to go ahead and get an instance of the runtime environment, and then I went ahead and called garbage collect on here just so you could see it was the real runtime environment. Then we print out the object address using the System.out.println. Let me go ahead and grab another instance of it, print that address out, and you can see that they're the same object address. Then we'll use the equals operator just to again verify that they are the same object and print out the System.out.println. So if we go ahead and run this, you can see down below here that the object addresses are in fact the same and that it prints out that they are the same instance. So this is guaranteeing that we are in fact a singleton. This Runtime.getRuntime method is a singleton.

Exercise - Create Singleton

So let's go ahead and run an exercise to create our own singleton. In this demo, we're going to create a singleton, and then we're going to demonstrate that only one instance of our object is created. We're going to then go through and convert it to being lazily loaded and finally make it a thread-safe singleton.

Demo: Static Singleton

For our singleton exercise, we're going to go ahead and create a singleton to manage access to who can create connections to a database and where they can create them at. Let's right-click on our package, com.pluralsight.singleton, and say New, Class. I'm going to name this class DbSingleton, and click Finish. And since we're going to build upon this example, I want to start off by not making this lazily loaded or thread safe. So we'll create an instance of our class by saying private static DbSingleton instance = new DbSingleton. And this will create an instance that our application can hold on to, and we'll manage it so that it only returns one instance of our application. To do so, we're going to create a private constructor so that people can't create new instances of this class on their own, and we'll just say private DbSingleton, and this will manage how that gets created so that people can't new up those new instances. Lastly, to flush out our singleton and control how people will grab that instance back, we're going to do the customary method with a singleton of saying public static DbSingleton, so it's going to return an instance of DbSingleton with a getInstance method. Now this is just a standard naming convention, it doesn't have to be called this, but it's customary with a singleton to have a getInstance method, and this will return that instance that we created up above. So there is everything we need to create our singleton. We have our static private instance that will hold on to, we have a private constructor so that people can't use the keyword new to create their own instance of it, and then we'll return that instance to the calling class when they call the getInstance method. And again, it's not required that it be called getInstance, it's just a standard convention when implementing a singleton. To test this, let's go ahead and create a class called DbSingletonDemo that will execute this code. So just type in DbSingletonDemo here, and I'm going to check the public static void main create and click Finish. And inside of here I'm going to replace this TODO with a DbSingleton instance, and remember that was a static method, so we'll call DbSingleton.getInstance to return an instance of our singleton for us. And to verify that this is running, I'm just going to do a System.out.println and return the object address that is created from that instance. And I can run this by right-clicking on our main method and say Run As, Java Application. When this runs, you'll see that it prints out the object address in our console. We have for this time that I ran it a 56e88e24. Each time I run, it will return a different instance for me. But there's a couple of things to note here. Okay, we have a singleton. How do we verify that? Well, if we were trying to create a new instance of this since we did a private constructor, it won't allow us. So if I type in here DbSingleton, and we'll say testConst = new DbSingleton, it's not going to allow me to do this. See, in fact, it tries to suggest that I put the demo in here. If I take and close that off to the constructor, it tells me it won't allow me to do that. So it won't work that way, but let's test that we're only getting one instance of that object back. I can do that by recreating that line of the singleton again, saying Singleton anotherInstance = DbSingleton.getInstance. Now when I run this and I print out that object address again, it should print the same object address for me. So I'm going to copy and paste that down there and replace that with another instance. Save that, and right-click and Run As again, Java Application, and you'll see that the object addresses are the same down there. So, it's not newing up another instance, it's returning that same object regardless of how many times I asked for it through that .getInstance method. So we've now created our first singleton. Let's move on to making this lazily loaded, and then we're going to follow that up with being thread safe.

Demo: Lazy Loading

To convert our singleton instance from being an eagerly loaded instance to being a lazy loaded instance is actually quite simple. Currently, it's eagerly loaded because on line 5 we're creating a new instance whether or not we use this. So all we need to do is grab this code, I'm going to copy it and change this to null, and then inside of our getInstance method, I'm going to do a simple null check and say if(instance) == null, then create a new instance of it. It's a very small change, and I will tell you right now that this isn't thread safe. We're going to talk about that in the next demo. But just changing this to being lazily loaded can be a substantial performance improvement for your application. I'm going to save this, switch back to our demo, and run this, and you'll see that it runs and executes exactly how we thought it would, and returns back our object addresses being the same across our original instance and our other instance. The difference here is just simply that we check to see that it's null and spool up that new instance. I've seen in large applications where we have a very slow startup because we're eagerly fetching all of those instances at the start up of our application. This will help it to where your app comes up quicker, and it's not such a memory hog when you're starting to spool up the application, it only uses what it needs. So, this can be a substantial improvement to your code and the performance of your code, which seems counterproductive where you're lazily loading it. Let's now look at making this thread safe.

Demo: Threadsafe

At the time of the original recording of this course, the keyword volatile was actually new of that version of Java, so we didn't record it originally with that concept in mind. So one of the things we'll do to make this thread safe is we'll go up to line 5 here and say private static volatile, and this will help us ensure that that instance will remain a singleton through any of the changes inside of the JVM. The other thing we're going to do is to ensure that nobody uses reflection on our code. So we're going to go to this private singleton that we've made and say if instance does not equal null, then we are going to throw a new RuntimeException, and inside of here we'll just say Use getInstance method to create. And we'll close this. Now, that will create our instance to where it's volatile and can't be reinstantiated through different things going on in the JVM. This will also protect us from having a Reflection class go ahead and create an instance of this. Now the next thing we're going to do is we're going to come down to our getInstance method. Now, we could do this a couple of ways. We're going to implement a double-checked locking mechanism and a synchronized check. Some people would originally on line 13 just make this whole method synchronized, and the problem with doing that is it's a performance hit. If we make this whole method synchronized every time we ask for an instance of it, we're going to actually synchronize that class and slow it completely down. Rather than doing that, let's go ahead and look and see if our instance is equal to null, and if it is, we're going to synchronize inside of here, so we'll say synchronized, and we're going to do it on the singleton, the DbSingleton class, and then we're actually going to check for null one more time. The idea behind this is that it's only going to actually happen if we're creating this one time, so it's a little bit of extra code, but it should only run if we're actually creating this for the very first time. So we'll do the exact same code again and we'll say if instance is equal to null, then create our instance of our DbSingleton. From there, we'll return back out our instance. So, we added our volatile to our instance, we protected this from being instantiated through reflection. Rather than synchronize on the whole method, we checked to

see if the instance is equal to null, then we synchronized on it. And the reason for doing this is it may be null, but if two threads are trying to go at it, once we've synchronized and checked for null again, if another class has a lock on that, it will then block our code and create the instance and return that synchronized lock to where our code would now go back in and say, if this instance is null it would go, oh, no, I'm already created and returned back out of this. You can see how that double checked with volatile instance inside of there is handling that functionality inside of our class. Now, let's double check this by going back to our demo and running it again, and you'll see that it works correctly. You're not going to notice a real performance hit inside of our application now because we're only doing this with two instances across our main method. But, it does add a little bit of overhead with that synchronization inside of there. But at the safety of our application not having multiple threads accessing this and our singleton really not truthfully being a singleton, we're now lazily loaded, thread safe, and we're using the latest version of volatile inside the JVM to make this a fully thread safe instance of a singleton.

Demo: Add Database

To make this an even more realistic example, I've gone ahead and added the Derby database JARs into our application to show you what this would look like to tie to a real database. Derby, if you're not familiar with it, is a very lightweight database that you can use for in-memory applications, and a lot of times it's utilized for testing, and I've even used it in a few production settings as well. So the first thing we're going to do is register a driver, and rather than have you watch me type all of this in, I'm going to just paste in these snippets, and I've replaced our constructor with this code that goes through and creates an instance of the driver manager, and this will go through and register our database driver using that driver manager and check to see if our connection is not equal to null for a reflection like we had done in the previous demo. So, you'll notice this connection statement is erring out. We're going to take and create an instance of that connection in here. Let me grab that snippet, and we'll copy this and paste it in there. We're going to do the same thing as we had done before by creating a volatile instance of this, and let me import Connection. And then the next thing that we need to do is create our getConnection method inside of here. And this looks very similar to what we did with our getInstance, but there is one key difference here. Our getConnection is not static, and this is by design. And we could set this up one of a couple of ways, but I chose to do it this way because it's very similar to how we would do this in production applications. I've gone ahead and made this to where we have to have an instance of our singleton and then use that instance to get our connection. And so we're going to utilize the Singleton to call an instance of this class first and then get the connection and return that. To test this, let's switch over to our DbSingleton demo, and inside of here, we're going to clean this up a little bit. We've got our code left over from the previous demo. Let's go ahead and cut all of this out except for that DbSingleton.getInstance, and we'll start by just grabbing a connection, and you'll see I've got some other code here. We're actually going to run that in a second. And grab that connection, and then we can go through and utilize that instance. So, what we have here is us getting an instance of our singleton and then just grabbing that connection. Then let's grab our statement, and I have a statement already built down here, and we'll execute this code inside of our demo instance as well. And this code goes through and creates an instance of a table using a prepared statement to do so and handles the exceptions. I'm going to import all

of those here, and we have a complete working example. So let's run this how it is, and we're going to add some performance metrics to this. Let's right-click and say Run As, Java Application. You'll see that it goes through and it says that we've created our table. It just dumps out that System.out.println on line 20 and prints out Table created. That's great. Let's do some more sophisticated things with this code, though. So, I'm going to switch back over here to my notes, and I've got some longs that we're using to grab the timeBefore and the timeAfter, and we're going to time our connection creation gap to see how long it takes us to create this, and then we'll call it again when we're done with this and see how quickly it will create that connection for us. So, I grab that chunk of code and replace this connection here with that, and you'll see that we're just grabbing the timeBefore and the timeAfter using a System.currentTimeMillis. I'm going to print that difference out. But before we execute that, let's do that same thing after. I'm going to go down here and grab another timeBefore and another timeAfter and print that out once again. And you'll see here that it will execute and require very little to no time to run that instance.getConnection again. So let's save this, and now when we run this, you'll notice that the first time it runs it took 503 ms to create it, it says that our table is created, and the next time that it runs it's been optimized to where it took 0 ms because it is returning us that connection that we already have created. So we have our singleton of our instance, and then our connections inside of that singleton that's returning that in 0 ms, so it's optimized that code to where we only have one connection that we're sharing across this. The creation of that connection has been optimized to where it returns in 0 ms. It's just returning us back the connection we have. You can see some real performance benefits for us doing our singleton this way.

Pitfalls

Some of the pitfalls of a singleton are that they are often overused. Once people discover the power and simplicity of this pattern, they have a tendency to make everything a singleton when it doesn't necessarily need to be. Although there aren't generally performance problems with singletons, if you make everything a singleton, it will slow your application down. Since singletons don't expose an interface and have private constructors, as well as private member variables, they can often be difficult to unit test. If you aren't careful when implementing it, they're not thread safe. Oftentimes, people start off with a singleton that's static, like we demonstrated, and it ends up morphing into something else and can oftentimes be confused for a factory. They start making the getInstance method take parameters. A rule of thumb is that as soon as it needs an argument in that method, it is not a singleton anymore, but rather a factory. Although not a pitfall, the java.util.Calendar is not a singleton, it is actually more of a prototype pattern because you are getting a new unique instance every time you call the getInstance method. People often confuse this, though, because it uses that identifier of getInstance, which is typically associated with a singleton.

Contrast to Other Patterns

To contrast the singleton with another design pattern that it's commonly confused with, the factory, let's go through the two side by side so you can see the differences. A singleton will return the same instance every time. There is a one constructor method with no arguments, and notice how I label that as constructor method. The constructor is always private so you

can't get access to it, so we have to access that through a construction method. And then there is typically no interface. Since this is a private constructor and a private instance inside of it, we don't expose an interface to help us adapt to different types of objects return. A factory, on the other hand, returns various instances, and, as the name implies, it returns multiple objects of various types. It also has multiple constructors because we're asking for those various types. There's different construction methods for us to get those instances back. It is usually interface driven. It's the opposite. We want to abstract out the back end and some of those things that we're returning so we usually do expose an interface with a factory, so it's a lot easier to unit testing and a lot easier to work with. It also has the ability to adapt to environments more easily than the singleton does. So, when you're looking at a singleton and it's not quite fitting, think about the factory and what the factory brings to the table, and some of these comparisons may help you in choosing a factory over a singleton, or a singleton over a factory.

Summary

To summarize the singleton pattern, we use this pattern when we want to guarantee that there's only one instance of an object inside of our application. It's very easy to implement, and it's even easy to make thread safe if you just spend a few minutes wrapping the construction of that object. It solves a very well defined problem where we only want to have one instance of that object inside of our application. But it can be abused quite easily. It's usually the most abused design pattern that we see out there in applications. Not everything needs to be a singleton, not everything needs to be guaranteed that there's one instance. So use it wisely and think about what you're needing to solve with this, and don't confuse it with a factory pattern as well.

Builder Pattern

Introduction

Hi. This is Bryan Hansen, and in this module, we're going to look at the builder pattern. The builder pattern is a pattern that people often use, but rarely create their own. It is a great pattern for handling the construction of objects that may contain a lot of parameters, and we want to make the object immutable once we're done constructing it.

Concepts

When considering a builder, you want to focus on whether or not the construction of an object is complex. By complex, we are specifically talking about lots of arguments for a constructor, or lots of setters, and then guaranteeing a contract of how that object gets built. Another key concept that is often overlooked is that you can force immutability on an object once the construction is finished, which you can't necessarily do with just a bunch of setters. This just helps further enforce our contract. Examples of this in the Java API are the StringBuilder, probably one of the most common ones out there, which is honestly one of the

best examples of a builder pattern; the DocumentBuilder, which I feel like illustrates how this pattern can be used on a complex object creation; and then a Locale.Builder, which is just another good example of this pattern and how it takes various parameters as part of its construction.

Design Considerations

The builder pattern solves a very common problem in object oriented programming and that is determining what constructor to use. Oftentimes, people create multiple constructors, and it can become difficult to manage. The creation of multiple constructors with each parameter variation is called a telescoping constructor. The builder pattern helped us overcome this by handling that construction with an object and rather than by parameters. The builder is typically written with a static inner class. The reason for this is that it returns an instance of the object that it is building. It doesn't negate the need for constructors and rather works in unison with those to call the appropriate constructor based off of its state. The builder pattern can negate the need for the job of being anti-pattern of exposing setters for every parameter that we could pass in. Since Java 1.5, we can take advantage of generics where it warrants to utilize a builder to expose various types of objects. But this is often not a necessary feature.

Example: StringBuilder

The StringBuilder class was introduced in Java 1.5. It fixed a lot of wrong doings that people were using just the string class for, or trying to do things better by using the string buffer class. The StringBuilder is a great example of a builder class, although oftentimes they are implemented as an internal static class, but it still shows the usefulness of this pattern by building it retroactively for the string object.

Demo: StringBuilder

Here is a simple demo of the StringBuilder object. You can see that we're going ahead and getting a new instance of the StringBuilder and then appending these individual strings to it as we go, and eventually we come down here and append an integer to it. There is just about any type of object that you want available through the corresponding append method to the builder, which we finally end up calling the builder.toString, which for this particular builder, since we are building a string, they just took advantage of the toString method to make that be our builder. There isn't a .build method, we call builder.toString like this right here. When we run this, you can see that it will go ahead and generate our string. This is an example of the builder pattern, and it has methods to append almost anything we want to a string and the number 42. The good thing about this is it is really performant and it gives us a nicer way to build strings rather than using the plus sign or the concat operator inside the string object. It is also a lot more performant than the StringBuffer object. The StringBuffer object does some locking much like the old vector object did compared to an array list, and this will result in faster performance for our application, as well as just simple ease of use, and it really is a great example of the builder over a buffer.

Exercise - Create Builder

Now that we've seen an example of the string builder in action, let's go ahead and build our own implementation of the builder pattern. First we're gonna look at demonstrating exposed setters out of a bean. Then we'll show some of the drawbacks of telescoping constructors. Then we'll finally create our own builder and finish building out the rest of the example.

Demo: JavaBean Setters

For the first part of our builder example, we're just looking at a straight bean here, and this is to create a lunch order. This was actually a problem I ran into for a company that I was doing some work for while building out this class. They had orders for lunch for a simple application that they were doing and they wanted to be able to build out each individual order, and the first stab that they took at this was a simple lunch order bean like this, and I've simplified this example down, but you can see we have bread, some condiments, whatever dressing we want on there, and then finally a meat that may or may not be there. The problem with this example is first, we're using the default no-args constructor, if we were to build this out, you could see that we just have a public LunchOrderBean here. So we're using this default no-arg constructor here, and then we have a getter and setter exposed for every individual property that we have there. The problem with this is for one, it's not immutable, meaning that after we create it, we can go through and change it because we have all these setters, and it's really unclear what the contract is of what they must have or must not have to signify what their lunch order is. Now to run this, I've got a little simple demo here, we just create an instance of the LunchOrderBean, and then just set each individual property that you want. We set the bread, set the condiments, set the dressing, and then set whatever meat we want; and that works fine, and it will facilitate what we're trying to do. If we run this example, we can see that it has our wheat, lettuce, mustard, and ham, which is exactly what we set up here, but there are kind of some bigger problems with it. Like I said, it's not immutable, it has no contract as to what actually signifies being a valid lunch order. We could go ahead and comment out all of these, and it would still run, and we don't have a real kind of idea of what our order should be. So, it'll work, it'll get the job done, but there's quite a few problems with it. In the next step, we're going to go ahead and look at telescoping constructors to see if it won't solve this problem.

Demo: Telescoping

Now this class is very similar to the LunchOrder bean, except it's using telescoping constructors. So if we look at the constructors in this class, I've got a LunchOrderTele here, and if I set these two side by side, you can see that they're very similar in nature to one another, except that we have a bean model, which is a no arguments constructor with setters and getters versus a constructor model, which is building upon the constructor for each individual case that we want to work with. Now, let's move this back over here and look at this in a little bit greater detail. You can see that our constructors build upon one another, so we have one that takes the bread, one that takes the bread and condiments, one that takes bread, condiments and dressing, then one that takes bread, condiments, dressing, and meat; and I've telescoped one way with this, I've gone up, but you could go down. So what I mean by that is we have this bread, condiments, and dressing, which is calling this

constructor, which in turn calls this constructor, which in turn calls this constructor. So they loop or recurse through each one. The bad part of this is well, we could also go the opposite way, which can be a little confusing. I could call this and pass in null to the other ones, or some default value, and work our way down all four of these constructors. But what happens if I want to build a sandwich that I don't want bread, or I don't want condiments? I just want bread and meat. Or maybe I'm on the Paleo diet or Atkins diet and I want to do condiments, I just want lettuce wrapped around meat and I don't want any dressing, I don't want anything else. Well, this example won't do this, so I have to add another constructor just for that. So I could do a constructor that is bread, or condiments and meat, or just condiments, or condiments, meat, or straight meat, which it doesn't lend itself very well to that. Now the bean demo does do that a little bit better, but then we have getters and setters for everything and it's immutable after the case. So this example is immutable, so I could have it one way or the other. I can have an immutable example using the telescopic constructors, or I can have one that will work for all the configurations that I have, which is the bean model. So you can see the drawback to both of them, and just to show you how this runs, I've got a constructor here that takes wheat, lettuce, mustard, and ham, if I go ahead and run this example, you'll see that it prints out what we're expecting here. But I've left in these setters to show you what we would have to do to get the configurability we want from the bean example, and we can't do that if I uncomment this, it's going to tell us that that's not available because that setter doesn't exist. So this is one of the problems with the telescoping constructor method versus the bean. Now the next example is going to be using the builder, which is the best of both worlds in this case, and you can see how it's a little bit more configurable doing these types of things.

Demo: Builder

I've got a basic class here for our LunchOrder that is very similar to what we had for our LunchOrderBean and our LunchOrderTelescoping constructor model, except there's a few little changes in here. To start with, I've got this String bread, condiments, dressing, and meat. And then we've got the skeleton of a public static class called Builder inside of here. Now inside of this, we're going to actually implement this builder and show you how this works. It's a little different, so I didn't want to use any copy and paste code in here. First we want to go through and this seems a little redundant, but we're going to create the same fields that we have inside of our LunchOrder class. So we're going to do private String bread. We're going to do private String condiments. And then we're going to do dressing and meat. The purpose for this is that the builder is its own container until we tell it to finally make what our LunchOrder is. So now that we have our bread, condiments, dressing, and meat in here, we start off first with a basic no args constructor because we don't have any qualifications about what we might want to have inside of our sandwich. Now the beauty of this is that we can mandate that this constructor take arguments for things that are required. So if we required everybody to have a bread and a meat associated, we could make this constructor force that. But we don't care about that. So from here, let's go ahead and we're going to create a bunch of methods that are going to look like constructors, but they're not. And this is the key to the builder pattern. Let's go ahead and make this full-screen while we're doing the rest of this example. So we're going to do public Builder. It's going to return an instance of itself. And this one's going to be for bread. I'm going to say String bread. Now the beauty of this is that we could also use typesafe enums for this as well. So I'll say this.bread is equal to bread. Now the

catch here is that we return an instance of this. And by this, we mean the builder object, this object that we're creating. Now that seems a little weird. It's so you can do a little approach or technique here that I'm going to demonstrate in a minute about using the return to build out your object. So let's finish this out for the others. We're going to do the same for our condiments. And rather than type everything out, I'll just copy this and paste it in so you can see it. So we're going to paste in our condiments, dressing, and meat. So now all of these are the same. We have an instance of a method for bread that returns an instance of Builder, an instance of itself. And then we're going to do one last key part, and that's implement the actual builder. Now to do that, we're going to do a method called, I'm going to create a method called public LunchOrder that returns, excuse me public LunchOrder, and it has a method in here called Build. Now this is a little different than the others because this is going to return a new LunchOrder using this. Okay, so this has a--- Inside of our LunchOrder object, we have a constructor that takes a builder, and then that copies over the bread, condiments, dressing, or meat that we have created. And we can enforce the contract inside of our builder. But also notice that there's only getters down here. I've minimized them so it doesn't take up as much space. There's only getters, so there's no setters. Now the beauty of this is we have the flexibility of the bean approach with the contract nature of the constructors. So if we look at our BuildLunchOrder demo, our BuilderLunchOrderDemo, you can see that what we did is created our new instance using the static inner class here. So we say LunchOrder. Builder returns an instance of LunchOrder.Builder, and we have our builder object. Now from here, we say builder.bread Wheat .condiments.dressing.meat. That's why we return an instance of ourselves so that we can just tack on these methods as we go and build out our object. Now why I like this and why this solved the particular problem that I was looking at here with this LunchOrder problem is that I was parsing these values out of a CSV file, out of a comma-separated value file as I was going. So I, instead of trying to gather all those at once and all the different conditions, I could grab it. And if that value existed, I could then append it on. And if it didn't, I'd go to the next one. So if somebody had bread, no condiments, and mayo and turkey, great. If somebody just had bread and meat, great. I could keep going through. I don't have to have this in here. So I can cut one of these out, and it will work just fine. So if we run this, you can then see that the next thing we do after we have all of our builder condiments and everything else added on there, we call build. If you wanted to, you could name this method MakeMeASandwich. We call build, and it returns an instance of our LunchOrder. But we also have that immutability that nobody can edit it. So now if we run this, we can see there is our wheat, lettuce, mayo, and turkey. But we can also chop out one of these and it'll work just fine, and we don't have to have an edge case constructor for it. We don't have to worry about the bean method, immutability, or whether or not. Now notice this value is null. We could also set a default type in there, the empty string or whatever we want to do. So we get a lot more flexibility about the creation of our object and how to handle these types of things. And like I said just to reiterate, if you did want to force them to have certain things in there, we could make it to where they have to use one instance of the builder with whatever those values are. So we can get that benefit of the telescoping constructors without the limitations or the problems of maintaining those individual constructors.

Pitfalls

The builder pattern really doesn't have a lot of negative things about it, so there really aren't that big of pitfalls, but maybe just some things to consider when choosing to implement

it. Objects created with the builder are typically designed to be immutable. The pattern itself is also typically implemented with a static inner class, again, not a big issue, and as we demonstrated with the StringBuilder API, there are ways around that. Unlike the prototype pattern, it isn't something that is usually refactored in after the fact, it does add a little bit more complexity to our implementing class over what could have been done with just a constructor, but without some of the nice features of the builder patterns. Adding to the complexity is that people are typically not used to an object returning itself for each subsequent call.

Contrast to Other Patterns

To contrast the builder pattern, let's compare it with the prototype pattern. The builder pattern is designed to handle complex constructors. There's typically no need for an interface, but you can implement the builder with one if you want. It can be implemented with a separate class, even though it is typically implemented in the class that it is building. And since it is implemented in a separate class, it can easily integrate with legacy code without needing change it. The prototype, on the other hand, is implemented around a clone method and trying to avoid the need to call a complex or costly constructor. Since the clone method is focused around member variables and constructors, it is implemented inside the class that it is trying to clone. This can make it difficult to implement in legacy code. Both of these patterns, though, are focused on complex constructors within one class. Their approach, though, is quite a bit different to solving that problem. So the builder tries to work with complex constructors where the prototype tries to avoid having to call them again.

Summary

Let's just recap what we covered with the builder pattern. It is a creative way to deal with complexity surrounding constructors in the creation of objects. It is fairly easy to implement, I would say almost as easy as the singleton to implement if done correctly. There are very few drawbacks to it, in fact, you kind of have to go out of your way to find some drawbacks with the builder pattern. And lastly, you can refactor it in with a separate class. Although it's typically implemented within itself, there's no reason that static inner class has to be a static inner class. It can be an external class that we go ahead and create what we want with our builder and then just call the constructor or whatever we want regarding setters or whatnot on the class that we are the builder for.

Prototype Pattern

Introduction

Hi. This is Bryan Hansen, and in this module, we are going to look at the prototype design pattern. The prototype pattern is used when the type of object to create is determined by a prototypical instance, which is cloned to produce a new instance. Oftentimes, the prototype pattern is used to get a unique instance of the same object.

Concepts

The concepts when choosing a prototype are when you are trying to avoid costly creation. Choosing a prototype is sometimes not as cut and dry as other patterns. I personally feel that it is often a refractory pattern and not a pattern that people usually think of upfront. Unlike the singleton, where you start off knowing that you want only one instance, you usually don't think of it in terms of something being expensive to create. Prototypes also avoid subclassing. Usually you create an instance of the actual prototype that you are trying to work with. The next concept can be a little confusing, too, but they typically don't use the keyword new. The first instance created might use the keyword new, but after that they are cloned. Although it can be implemented without it, there are good reasons to create an interface for your prototype instance. Prototypes are also usually implemented with some sort of registry. The original object is created and then kept in our registry. When another object is needed, we create a clone of that object from the registry. An example of the prototype pattern from the Java API is the java.lang.Object clone method. Although it is not all of a prototype pattern, it is definitely the basis for what we will design our prototype patterns around.

Design Considerations

The prototype pattern is an interesting pattern in that it is just changing the way that we call the keyword new. If an object is expensive to create, but we can get what we need by copying the member variables, then the prototype is a great fit. The prototype typically implements the Clone/Cloneable method and interface. This enables us to avoid using the keyword new. Typically, if our creation is expensive, it is going to be when we call the keyword new. Although we are using the clone method and essentially just making a copy, each instance is still unique. Different from patterns like the builder, costly construction is not handled by the client. In fact, I would say that the builder is the opposite of the prototype pattern. Different from the singleton, you can utilize parameters in the clone if you need to, but typically you don't. As the architect, you can choose whether you want to do a shallow versus deep copy. A shallow copy just copies the immediate properties, whereas a deep copy will copy any of its object references as well.

Example: Statement

Here is some code of an everyday example that you might run into. Creating database statements can be expensive, especially if we just want to swap out the parameters that we are passing in so that we can run the query again. In this example, you can see that the constructor takes a lot of information, but the clone method itself is very simple and takes advantage of the values passed in previously to get another instance. Let's run this sample code in a demo and dive into it a little deeper.

Demo: Statement

For this everyday example, we have created a statement object. The more functionality we tie into this object, the more expensive it would become to create it. For our example, you can

see that we have a constructor that takes a string, an array list, and a custom object that we have created called Record. Not to stray very far from this demo, but if you are familiar with prepared statements in the Java API, you will know that they are compiled against the database that we're using. This example is to help illustrate this same functionality without duplicating the prepared statement object. Notice that this object also implements the cloneable interface, which forces us to then implement the clone method. From here, you can see that we just call the parent clone method, super.clone. And this is what returns our instance that we then cast to our statement object. The record object itself is just an empty object, but I have included it here to show what happens with object references that are included in our clone. So we have a clone object inside of here and we implement the cloneable, it then, in turn, creates an instance of this record object. So let's see what happens. I've created a demo over here where we build a simple string with some SQL in it and then pass in a list of parameters. In our parameters we've added a movie of Star Wars, and then we have a record object, and right now our record object is empty. And since it's passed by reference, we could shove some values into this and then return that object to see what happens. So let's go ahead and we create a statement and then run this statement and see what happens. First, we're going to go through and see what returns for our SQL, our parameters, and our record. And then we're going to call clone on that, and our second statement is going to return some values that will also look like the same thing, our SQL, our parameters. But I want you to pay particular attention to what happens with our record object here. So let's run this, just right-click and say Run As, Java Application. And you'll see that it dumps out our SQL, so it says our select * from movies where title = whatever, and that's what are are our SQL is that's injected there, and then returned here is well, and then it returns our Star Wars instance or basically our array list that has those parameters added. But it also returns the same object, our same record object. So even though it went through and it did our clone inside of statement, it just returned the references to the array lists that were passed in and the references to the record objects that were passed in. This is an example of a shallow copy. Now, if we were doing a deep copy, it would return a new array list with those parameters passed into it, and a new record object with whatever values the record object had stored in here, which currently is nothing. But this is an example of a shallow clone because those objects are just getting returned the same as what we had passed in. So it's not necessarily a true copy. In fact, it's a little bit of a dangerous copy because we could go through and change the parameters in this array list, and it would reflect in both objects.

Exercise - Create Prototype

Now that we've seen an example of one, let's go ahead and create our own prototype in this exercise. We're going to create a prototype pattern, demonstrate the shallow copy of it, and then create an object with a Registry.

Demo: Prototype

For this example, I've set up a couple of base classes, and objects, just to implement, to prove out our prototype pattern. Let's start out with this Item class, and the Item class has just some very basic things in here, a string, a price, that's a title, a double that's a price, and

a string that's a URL; and what we're going to implement here is just some basic object look ups. Think of it in terms of a company like Amazon, that if they were just to display an object for every item on their page, it would become very expensive to create all those objects, especially if we're filling all of this type of information in. The sub classes of Item are Book, which just extends Item, and then has a number of pages just to have an additional filled in here to offset it from a basic item, and then a Movie, which we have that has a runtime associated with it. The two other classes that we have associated with this are also a Registry. We're going to go through and create this registry and fill it the rest of the way out. Now this is probably a lot of the meat of the prototype and shows actually how the prototype works, and I've purposely left this createItem method empty. There is a loadItems method down below that just shoves some basic information in there, but this is really where the heart and soul of the prototype is going to take place. Lastly, there's a demo class that we're going to run, and you're going to see how this executes what we're doing inside of our object. So, let's go ahead and get started. If we open up our Item object, this Item.java, you can see that it's an abstract class, meaning that we're going to want to have the Book and Movie objects implement the final functionality of this or determine what that final functionality is. We want to go ahead and implement the clone method in here. So we're going to say that this implements Cloneable, and the Cloneable method, the Cloneable interface forces us to implement this method, which is the clone method, and you'll see it throws a CloneNotSupportedException, that's in case we have a subclass that doesn't do what we want it to do, and then it will also return this object of clone. So let's go ahead and just save this right now, this object from the super object that we're cloning. Now, if we have nothing unique in our sub objects, this will go ahead and do the clone for us, and that's actually going to work for what we're doing right now. So our basic clone method inside of our Item class gets extended by implementing the Cloneable interface. I don't have to do anything to Book, and I don't have to do anything to Movie. Now, let's go ahead and go over to our registry now, and look at our createItem method. So this is just a method that we've created, and it's our own basic registry. There's many types of registry that you could implement out there, but we're going to go ahead and just put a little try catch here, and inside of our catch, we're going to catch that CloneNotSupportedException, and just for sake of debugging, we'll do an e.printStackTrace, where you'd want to do something a little bit more official inside of our class if we were going to use it for production code. Now let's go ahead and say item =, and we'll want to cast this because it returns an object, (items.get(type)).clone(). Now you'll notice that we have to cast this interface right here, or cast our item being returned to our item, and that's because of our interface inside of here that implements Cloneable, forces it to return an object. Unfortunately, the Cloneable interface was created with Java 1.0, which means it doesn't have any knowledge of generic, so we can't tell it that we're going to pass in this object and return this type of object on our clone. I actually think this is one problem with using just the basic Cloneable interface from the Java API, this is why I recommend oftentimes implementing your own interface and creating your own clone functionality that way. Not a big deal, but one of the shortcomings, people will get a little leery when they start having to typecast things like this. So now that we have our createItem method in our registry, when we run this, what it's going to do is it's going to go through, and the first time we run this, our registry has a loadItems method in there that goes through and creates some basic items and puts them in our registry. So, we're going to create a movie, put that in our registry, using this items.put, and then we're going to create a book, also put that in our registry using this items.put down here. Now, what this does is when we go through and want to get an instance of it, we say, give us back an

instance of a movie, and give us back an instance of a book, and then we can change those values from there, and we get a unique instance every time. Now, one last thing to know is that our object types are being used by a key to look this up, they're not necessarily a bad thing, but, I would maybe consider also implementing this with an enum. So let's look at our run method inside of our PrototypeDemo, I've got a basic main method set up where I create an instance of the registry, then I go ahead and create a movie, and this is basically executing our clone method for us, and then I have my instance and can do whatever I want with it, override whatever values I want to, and manipulate whatever fields I want. So it gives us a way to set up some defaults that we want for each object that's going to be returned without having that heavyweight expensive create an object every time. Think if you had 10,000 objects on the page, which, believe it or not, is a common problem if you're implementing like a hibernate, where it creates a lot of objects and returns those, it can be very object or labor intensive to return that information. So now let's go ahead and run this, it's going to go through, create the registry, create an instance of a movie, print it out, just so we can see what the address is, and all the various information with it, then it's going to create another one, and show us that it's also a unique instance, and go ahead and give us our values back. So let's right-click and say, Run As Java Application, and you'll see down below here that we have returned an object of type Movie, with this address, and it says what its runtime is, and we didn't override that runtime, that's why it's still stuck at two hours, and then we have the Creational Patterns in Java, and then we created another instance which is moving down here with its own unique entrance, and its unique object address, and then it has a runtime of two hours as well, and it's looking at the Gang of Four. URL we didn't set in either of them, so it's being returned as null in both instances. So you can see how it gives us a great way to set some default information up, which was all defined right here, and then override those without having to call new each time. So the only time we ever call new inside this application is right here on this line and right here on this line, every time we want to get an instance of it, we get an instance back of a movie, and we don't have to use the keyword new anymore. So it's a lot lighter weight object instantiation, and a lot faster, but we're getting a unique instance each time, and that's the definition of that prototype, is getting a unique instance every time we ask for this object back.

Pitfalls

What are some of the pitfalls of a prototype? Well, prototypes are often not used. This is a difference from the singleton pattern, where people usually overuse it. Another pitfall would be that you typically have to use it with another pattern. A loose definition of a framework versus a pattern is that if a pattern contains other patterns, it is a framework. This isn't always true, but it makes you sometimes question the use of a prototype because we typically have to implement that with some sort of registry. Lastly, a lot of times you want a deep copy and the clone interface only does a shallow copy. You can, of course, implement the functionality of a deep copy yourself, but that requires more coding yourself, and people start to second guess the validity of the pattern and whether it's solving anything for them.

Contrast to Other Patterns

Since it's easy to always just pick on the singleton pattern, let's compare the factory with the prototype pattern. A prototype is focused on lightweight construction, either through a copy constructor or using the clone method, like we demonstrated in our two examples. You can choose to do a shallow versus deep copy, but really in the end, you're looking at just creating a copy of yourself, even though in our one example we did an abstract class with a sub item, we're looking at creating a copy of whatever item instance we are. So we were a movie and we got copies of a movie, we weren't looking for different objects based off of what we were trying to do. The factory, on the other hand, is focused on dealing with flexible objects based upon your request. you can utilize multiple constructors instead of just the clone method, and it also utilizes creating a concrete instance of an object, and it is a fresh instance since we're utilizing the keyword new, so there aren't any programmatic defaults by nature. You remember back in our demo, we had set a default URL on a price and a run time, things like that; typically, that's not a feature that we program into a factory pattern.

Summary

To summarize the prototype pattern, we are using this pattern to guarantee a unique instance every time we ask for it. A drawback or a side note on it is that it's often something that gets refactored in later, and that's because we're usually looking for it to help us with some performance issues inside of our application. So if we have an application, it's creating a lot of objects, we want to go ahead and implement this pattern so that it can help us obtain these objects faster without the heavyweight or overbearing nature of creating then using the keyword new every time we want an object. And lastly, I would note, don't always just jump to a factory. Look at a prototype to see if it will solve your problem with your current situation because a factory can often lead to other things that a prototype is nicely suited for.

Factory Method Pattern

Introduction

Hi. This is Bryan Hansen, and in this module, we are going to look at the Factory Method designed pattern. The factory method pattern is in some ways the opposite of the singleton pattern and it probably is the second most used creational design pattern.

Concepts

The concepts when choosing a factory are that it doesn't expose instantiation logic. The client knows next to nothing about even the type of object that is being created. It is able to do this by deferring the instantiation or a creation logic to the subclass. All the client typically knows about is a common interface that the factory exposes. Factories are oftentimes implemented by an architecture or a framework and implemented by the user of that framework. This establishes a contract for how things will be implemented within the framework, but allowing flexibility for the end user to define how it can be implemented. Examples of this in the Java

API are the Calendar, ResourceBundles, and a NumberFormat. Oftentimes people think that Calendar is a singleton because it has no arguments or a no arguments constructor, and factory methods can have arguments. The difference is that a calendar can return different subclasses of the calendar and the client is unaware, whereas with a singleton you are just getting a single instance of that implementing class.

Design Considerations

I view the factory as almost being the opposite of the singleton. The factory is responsible for creating instances and managing the lifecycle, at least the creation part of the lifecycle. Objects created are referenced through a common interface. Factories will also reference multiple concrete classes or implementations, but the client is unaware since they are referenced through the common interface. The other key design principle is that the method to request an object is typically parameterized. These parameters are what are used to determine the concrete type. The UML diagram on the left shows the Factory, which is an implementation of the factory pattern. It has a factoryMethod, which will call and then return an interface to whatever object type we are attempting to build. The Factory itself refers to a concrete implementation that does the actual instantiation of our ConcreteObject. So we have our Factory class or our Factory abstract class that has a static factoryMethod that we are going to call, and based off of those parameters that we pass in, we're going to call a concrete instance that's going to return the object type for us.

Example: Calendar

Here's an everyday example we use in Java all the time, and that is the actual Calendar class. So in this snippet of code, we go ahead and do Calendar.getInstance, which returns an instance of the Calendar class. And then on the next line I call System.out.println. Now this is to show you the actual object type that's going to be returned. This particular class will show you the implementing class it's underneath. So the concrete implementation rather the in the abstract base class that we're using. And then finally, just to show an example of how the code is being ran, I used the calendar.get and pass in a parameter of Calendar.DAY_OF_MONTH just to show that the object is working, how we think it should be.

Demo: Calendar

Here is that same example code that we were just looking at. I want to point out a couple of things. First when we run our code, it's going to go ahead and call our Calendar.getInstance, but we're not going to know exactly what type of calendar's going to be returned, and that's why I use this System.out.println here. So if we go ahead and run that, you'll see that the java.util.GregorianCalendar down here gets dumped out from that System.out.println, so the Gregorian calendar is the actual concrete implementation that's being called underneath this factory instance. Now another thing that I wanted to point out is that we can use different types of calendars, and that's what makes this different than some of the other design patterns because it is parameterized. So we can do Calendar cal = Calendar.getInstance, and you can see that there are different time zones available, locales, different things like that that we can call to get various types of

calendars. So we could use a time zone from the East Coast, the West Coast, Pacific/Mountain, whatever, and get our calendar with that. So instead of like the singleton where I'm going to just get whatever object it says is available for me, I can get different types of calendars based just simply off of time zone or locale, so that's a nicer feature of the Calendar API, and really that's a feature of the factory method over singletons or some of the other creational design patterns that are out there.

Exercise - Create Factory

Now that we've seen an everyday example, let's look at what it would be like to write our own factory and go ahead and implement an exercise doing so. First, we're going to start off by creating some basic pages. So this example is going to be a canned tool for generating websites, much like if you went to wordpress.org or a site like that that will generate websites for you. The next piece that we're going to do is actually create the website from the pages, and this is just going to be a templated tool. We're not going to actually implement a website. The next step will be to create the concrete classes that actually implement those features of creating the website. And then finally, we're going to create our factory. We're going to polish that last bit of our factory off by creating an enum that switches the logic in the factory based off of that enum.

Demo: Factory

One of the drawbacks to the factory pattern is that it takes a lot of moving pieces to really demonstrate what we're trying to do with it. I've gone ahead and set up some scaffolding, some skeleton pages to help facilitate this, and so we're not just watching me type forever. We're going to create an application that's the equivalent of a website generator. If you went to something like wordpress.org and requested a new website, you're going to get a bunch of canned pages to begin with. Now, we're not going to create a website, but you'll get the idea of what these canned pages are and how this works. So I've gone ahead and created a page abstract class, it's just a base class, and it's just to facilitate showing what you would do in an actual application that you're trying to implement. So let's go ahead and add one more page in here. Just so you can see how I'm creating these. I'm saying New, Class, and I'm going to do a CartPage and this will be for a shopping cart in our basic application here. We're going to try and we're going to create two types of website, a blog versus a shopping or a, you know, Amazon style e-commerce type application. So we have our CartPage. Again, there's our AboutPage, our ContactPage our PostPage. Nothing too extravagant going on there. Now, the next thing I'm going to do is I'm going to go through and create the website abstract class. Now, this is going to be the start of our factory pattern. So I want to go out here and right-click and say New, Class, and we're going to say that this is a website. It is abstract. And let's go ahead and just click Finish on this. Now inside of here, we're going to add a few things. We're going to add all of our pages that were going to create, so we're going to say protected ListPage, pages, = new ArrayList, and save this. So now what we've got is just some holders for the pages are going to create. Now, this point is really critical, and I didn't want to just copy and paste this in because this is kind of some of the key points of what's going on with the factory pattern here. The next thing we're going to do is we're going to just put a default no-args constructor in here where we say public Website

and we call this.createWebsite. Now we haven't created this method yet, so it's going to complain that this doesn't exist yet. Now, while we're at it, we might as well generate a getter for these, Source, Generate Getters and Setters, and we just want a getter for the get pages. So we say OK. And now we've got our basic structure there. Now the last thing we want to do is create our public abstract void createWebsite method. Now, this is the crux of the the factory method pattern here. So all of the base classes, excuse me, the concrete classes, that implement this base class, are going to go through and override this method to generate those. So now that we've got our website, let's go through and start off by making our first type of website which is going to be a blog. So we'll say New, Class, and it is a blog, and it's super class is going to be Website, and click Save that. Now, you'll see it automatically overrides that method for us because we haven't declared it as abstract, which is what we want. So we're going to say pages.add, and we're going to add a new PostPage and we'll add a pages.add, and let's do a new AboutPage. And you can see how we're putting pages in here that are specific to the blog. So say new CommentPage, and just for good measure, let's add one more for the ContactPage. So we'll say pages.add and will do new ContactPage. So you can see how the base class doesn't have anything to do with the creation, but rather the concrete implementation does. So the blog class, which is implementing that factory method, is what's concerned about creating the implementation that we're looking at. Now that we have our blog, let's go ahead and do the same thing and create our shop website. I'm going to go over here and say New, Class, and we want to call this Shop, and it's super class is going to be Website, click Finish. And we're going to do basically the same thing. For the shop, we want to do pages.ad, and we're going to do a new CartPage, and then we want to do pages. add, and you see how the content here is different for the website, for the shop website versus the blog website. And the concrete implementation is what's concerned with how that gets implemented. But we're still going to return the website interface or contracts. We are using an abstract class here, which is giving us the same thing as an interface. We could implement a website interface and achieve the same thing. By interface, we're really talking about the contract and not the keyword of interface inside of our application. So just for good measure here we'll throw a SearchPage and save that. So now we have our shop and our blog. Now we can move on to our actual factory. Now, I have created a class out here called WebsiteFactory with nothing in it yet, and just to show you that typically when we have a class, the factory for it is just, it's more of a rule of thumb, but is usually named whatever that class is with the factory name. So website, we have a WebsiteFactory, pretty common sense there. So we're going to say public static Website getWebsite, and this is where the factory method really steps into play here because we have those concrete implementations. We want those to be concerned with how they're created, not this base class, but we don't want people to be able to access those concrete classes on their own. So we're going to put a sightType inside of here, and we can do a basic switch statement here, and as of Java 7, we can do a switch on a string. So we can say sightType and pass into here some basic cases. So we'll say case and we'll do blog and we can say return new blog. And now let's shift this in here, and we'll say case shop and do the same thing here, pass in for our arguments, not really our arguments, but our meat of our method here, as a new Shop. And then we can always, which is a nice feature of this, add a default case where we just return null. So now we have our basic factory method and where we're getting our method from, we have our factory that calls based off these types, but our true method is occurring, our factory method pattern, is occurring in this createWebsite method that's overridden from our website itself. And that's why I was saying this is a little bit more complicated of a pattern because there are a couple of moving pieces to it. So our factory is going to call the new no args

constructor for a blog, and it could have arguments too, it doesn't have to be a no arguments, which is then overridden from Website that calls our constructor at this.createWebsites. And this is where our factory method is coming the case, it's called a factory method, a lot of people will shorten that to just factory. It is a factory method. Our method is what's concerned about our instantiation of a createWebsite, which in our blog or our shop, then goes through and actually builds what our application is going to do here. So now to demo this, we can go over to our Factory Demo and we can say WebFactory.get, and I have this basic code already set up for us to where I just call our WebsiteFactory.getWebsite and we can pass in a blog and get that string back and call system.println with us, and it will show what pages we have built in that. And then we call WebsiteFactory.getWebsite for our shop, and it will also print out what pages we have available to us there. So let's go ahead now and run this. Save that, and you'll see that the pages are different down here based off of what type of site we have. So for our blog, we have a PostPage, an AboutPage, ContactPage, CommentPages, those types of things. Where for our shop, we have our ItemPage and our SearchPage, and those types of things, so.

Demo: Enum

One thing we can do to make our factory a little bit better is to get rid of these string liberals in here and convert this over to enum. Now, this is definitely not a requirement of the factory pattern, but it really is good coding practices. And since patterns are an architectural type principle, really, there is a better way to do this and we want to show you that. So what we're going to do is we're going to go up here and right-click and say New, Enum, and we're going to call this the WebsiteType enum. And inside of here, we're going to create two of them. We're going to create BLOG and SHOP, and just store those two. Now, if we come over to our factory, we can get rid of this string here by saying this is a WebsiteType. And the beauty of this is when we switch on an enum, we don't even have to preface it with whatever that type is. I can just say BLOG and SHOP, and it already knows what I'm trying to do. So I don't even have to say WebsiteType.BLOG and WebsiteType.SHOP, it already knows since we're switching on an enum to just go ahead and do that. I really like this style of coding with enums. As we go over here to our demo and now pass in WebsiteType.BLOG and WebsiteType.SHOP, and we've gotten rid of all of those string literals now. If we go ahead and run our code again, just to make sure that's working correctly, you see it works just fine and we've gotten rid of those strings which could break and become a little bit more brittle in our code down the road.

Pitfalls

The pitfalls of a factory method pattern are namely complexity. You will notice compared to the other creational design patterns the factory pattern is almost double the amount of code to demonstrate it. The part that most people often get wrong when implementing the factory is that creation doesn't take place in the factory itself, but rather in the subclasses of the type of factory method we are creating. The factory method pattern is also the pattern that's generally not refactored into. You need to design from the beginning that it's going to be a factory and then plan accordingly.

Contrast to Other Patterns

To contrast the factory pattern with another pattern, we're actually going to do the same comparison we did with the singleton. The singleton and the factory are almost the exact opposite of one another. The singleton returns the same instance. It has one constructor method with no arguments and no interface and no subclasses typically, whereas the factory returns various instances and has multiple constructors or arguments that we can add to a constructor type method. It is very interface driven, and when I say interface driven, I don't always mean that we're implementing an interface, but it can be an abstract class or a contract, it's very contract driven. There are always subclasses involved. You can't have a factory pattern without having some fashion of a subclass involved. And it's easily adaptable to your environment, so a lot of frameworks are written using the factory pattern where you implement them per environment as you want to use them.

Summary

Just to recap the factory pattern and how it differs from some of the other patterns, it is parameter driven, in fact, this is one of the few creational patterns that is parameter driven. It solves complex creation in a different fashion than all the other patterns. So we had the builder that was involved with enforcing a contract with our constructor and multiple parameters, but it didn't support parameter driven construction meaning that if we wanted to choose a type at run time, the factory is really the only one that deals with that. One of the drawbacks of the factory is that it can be a little bit complex and it really is the opposite of a singleton. So if you're looking at a singleton, and it doesn't seem like it's the right fit for it, you probably need to be looking at a factory. Most of these patterns stand on their own, so unlike traditional courses we do here where we try to talk about the next module that's going to come up, all of these patterns will stand on their own two feet, except for the factory has kind of a cousin, which is the AbstractFactory, and that is the next module we're going to cover in this series is the AbstractFactory and how that applies to the factory method pattern.

AbstractFactory Pattern

Introduction

Hi. This is Bryan Hansen, and in this module, we're going to look at the abstract factory design pattern. The abstract factory is very similar to the factory method pattern, and in fact, is typically implemented as a factory of factory patterns.

Concepts

The concepts when choosing an AbstractFactory are that it is a factory of factories. Although it can be implemented without using the factory method pattern, more often than not it is. It is typically summarized as a factory of related objects. It also takes the concept of a common interface a step further. The common interface is implemented throughout the AbstractFactory and its underlying factories, and just like the factory method pattern is

deferring the instantiation or creation logic to subclasses as well. Examples in the Java API are the DocumentBuilder from the XML APIs. There aren't a lot of other examples because it is often implemented in frameworks and not just in the standard Java API.

Design Considerations

The design principles when implementing an AbstractFactory are that you want to group a collection of factories together. The factory is still responsible for the lifecycle itself, and it has a common interface that is carried throughout the AbstractFactory, as identified on the UML on the left here, down through the ConcreteFactory, and finally to the implementing class below. Just like the factory, there are concrete classes that are finally returned from the underlying factory. The AbstractFactory also has parameterized create methods just like the factory method pattern does as well. One key point is that the AbstractFactory is typically built using composition where that is not the case with the factory method pattern, so one very key distinction there. The AbstractFactory itself implements a very good use of objects for development through composition.

Example: DocumentBuilderFactory

One of the few examples in the Java API is the DocumentBuilderFactory from the Java XML APIs. The DocumentBuilderFactory itself is an AbstractFactory, the DocumentBuilder is a factory, and then the document itself is the concrete class that is created from those factories. The document is actually an interface, and the factory chooses an implementation of that concrete class to return to the end user or the client. Let's look at a live example of this code.

Demo: DocumentBuilderFactory

Just a quick walkthrough of this code. We start off by creating a simple XML document that we've stored into a string, and then to utilize that in the DocumentBuilderFactory and the DocumentBuilder, we have to convert that into a ByteArrayInputStream like we've done here. The AbstractFactory in this example is the DocumentBuilderFactory. Now the reason it's an AbstractFactory is because we don't know what the underlying implementation of this is, nor do we know what the underlying implementation of the factory is when we get it. All we know is that we get our document and can eventually run and use that document. Once we have our document, we can get an element and work with it and see what we have there. I've added a couple of System.out.println's at the bottom of this just for some rudimentary debug purposes to show you what the actual implementation of the AbstractFactory is and the factory is, so you can see we call a getClass on them down below here. Now if we go ahead and run this, you'll see that we have a Root element of document that gets displayed there, but we also get our actual implementation class of that DocumentBuilderFactory and that DocumentBuilder. You can see that our DocumentBuilderFactory is com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl, and then we have that same thing for the DocumentBuilder itself, so we're using the xerces implementation. Now if you don't know much about this factory, and it's not really an

important part of the demo, but you can swap out this implementation by switching up a command line parameter that tells it to use a different library. Most people are happy with what the default implementation of it is, which is the apache xerces implementation, but that's why this is an AbstractFactory. We know nothing about what's going on from a client perspective of which implementation is chosen, and we don't care what factory's returning as well. All we care is that we get our document and it's usable. So all of those hideous details of creating that are hidden from us, but we can swap them out if we need to. Now, as I've mentioned a couple of times now, the xerces and XML APIs are more of a framework for parsing XML documents, and that's why this is dumping out that code as part of a framework and it's chosen for us under the hood. So, you can see the variables are named appropriately for an AbstractFactory versus factory in this example just to help illustrate which pieces of this are in fact the AbstractFactory versus the factory.

Exercise - Create AbstractFactory

Now that we've seen a demo of an AbstractFactory, let's go ahead and build our own. We will first walk through some code that I've set us as a template because it is quite a lengthy example. Then we're going to build the actual AbstractFactory itself. Since it calls a factory, we will build out that piece, and then once we're all finished with that we'll finally produce a final product and run that demo.

Demo: AbstractFactory

To not make this demo take so long, I've gone ahead and set up some templated scaffold code that we're going to walk through really quick. This implements a factory for creating credit cards, so basically think of it as you're submitting an application for a credit card. This application is going to choose for you and create the right credit card based off of the criteria you put in. It dabbles a little bit on some strategy or some behavioral patterns too, and I'm not going to go into too much detail there, but this is a great example and one that I've had to do inside of applications before of how you can use this in your day-to-day code. The AbstractFactory, in this case, is the CreditCardFactory, and we're going to go ahead and implement this out, but we'll walk through the other code first. There are concrete factories in here, or factory methods, such as this VisaFactory. Now it's given me some red errors here because I've deleted some of the AbstractFactory code, so it's saying hey, you're not overriding anything in this. You can see that as we go onto this factory, and if you've gone through the Factory Method Design Pattern module, that it's coming in here and switching on a type and saying oh, you want a gold card? Well I'm going to return the VisaGoldCreditCard and defer to the factory below to handle creation, and I'm just using the default no-args constructor in the factory below because I really don't care about that implementation of this detail. I'm more concerned about the AbstractFactory. Now there is some validator code in here for validating the credit card information you're putting in, and we're just going to show how you can plug those pieces in. I'm not going to take the example much deeper than that. So there are a bunch of different things in here. There are some CardTypes, so we have GOLD, and PLATINUM. We have a basic CreditCard class that goes through and has a cardNumberLength and a cscNumber. It also has some Validator code, so I put in a PlatinumCreditCard validator, a PlatinumCreditCard and a PlatinumValidator, and

a GoldCreditCard and a GoldValidator, which are just stubs for you actually putting your own code into. The real catch here is that we have our AbstractFactory, which is our CreditCardFactory, and then our concrete factories, which are an AmexFactory here, which is pretty empty, and then a VisaFactory here, which this one is implemented more complete. Well let's go ahead and get started. So we're going to open up the CreditCardFactory, and since this is our AbstractFactory, this comes in here and determines, based off of what we're doing, which card the person should get. So we're going to say if(creditScore is greater than 650, then we're going to give them an American Express. This doesn't mean you have to have a great credit score to have an American Express or if you have a bad credit score you're going to get a Visa. It's just a simple example based off of conditions. So we're going to say AmexFactory. Now we're going to save this, else we'll return them a Visa, return new VisaFactory. Now the way an AbstractFactory works is there are also some interfaces that we're going to pass through to our concrete classes and concrete factories down below. So we're going to say okay, we want a public abstract CreditCard meaning that every factory that we return is going to implement this method. So we're going to get the credit card type that we have, so we say getCreditCard, and this is going to take in an enum that we've created called CardType, and I showed that earlier, it was the CardType of platinum versus gold, so we'll say cardType, and then we're also going to just, because a factory is a group of similar factories, we're going to put a validator in here as well. We'll say public abstract, and we'll do Validator getValidator, and we'll also pass in a CardType here as well. And basically the logic behind this is that you're going to apply for a card, we're going to give you back the card you qualify for, and then we're going to validate that you can actually have the card that you say you have access to. So we have our Validator and our CreditCard. Now let's go ahead and open up our ConcreteFactory, which is our AmexFactory that we've created right here. So the AbstractFactory just says I'll return you whichever factory you need to fulfill these two methods down here. So we're going to open up our AmexFactory, and inside of here we want to go ahead and build a switch based off the card type. Now this is one of the things people don't like about the AbstractFactory, so we're going to say switch, I'm going to do cardType, let's actually implement out our switch statement first, and we'll say cardType. Now I've said this in past examples, but this is another case that I just get to bring this up. I really like using switches for this because I can, as I'm passing in an enum in this switch, I have the values here already available as my cases, so I can say case GOLD, and I can return, return new AmexGoldCreditCard. And if we really wanted to get into the finite details of the factory method pattern here, we don't have to rely on just the default no-args constructor, we could do something more with the template method pattern. I'll say PLATINUM, and we'll do AmexPlatinumCreditCard and save that. So now we have our factory in place that's going to go ahead and defer to this instance to create what is being returned. So whatever the credit card is, whatever the creation of that credit card is, our parent class has no knowledge of what's going on here or how those were created though. So our AbstractFactory, it barely even knows which factory we've chosen let alone what type of credit card's going to get returned, it doesn't know if it's a platinum versus a gold card or whatever other card we have implemented there. It also doesn't know how those objects were created down there. So we have our AmexFactory. The other thing we need to do is pass in our validator code. Now I do have that copied here just to save on typing out this information. I'm going to go ahead and paste it in and save this. So now we have our AbstractFactory, which is our CreditCardFactory, that determines which factory should be chosen. Then we defer to our AmexFactory, and we are calling the no-args constructor here. If we wanted to do the template method pattern, we could do a template method, a factory

template method that would go ahead and call the appropriate instance down below. We can also choose some things based off a constructor. Go ahead and look at the factory pattern though. We don't need to belabor it in this example. And then from here, we can go down and do some more sophisticated things in our credit cards down below. Right now I just have the default empty class with some fields that are provided to us by way of our CreditCard. I have a protected int cardNumberLength and a cscNumber, the security code number that just goes ahead and returns that information for us. Now we can go ahead and open up our demo, so we have our AbstractFactoryDemo, and inside of here you can see a couple of things. I have gone ahead and created an instance of the CreditCardFactory, and like our previous example, this is named abstract just to show you which that factory is. The only thing it knows and passes into it is saying hey, I want to get a CreditCardFactory, and my credit score is 775. That's all we know is we're just going to pass in that my credit score is 775, and this is great. I'll get you the right one based off of your credit score. Then it says okay, now get me the credit card, and I think I want a platinum credit card. So it will come back and say great, I'm going to go ahead and give you a platinum credit card based off that factory. So no knowledge of how that was created or what was done is passed back to the client. The same thing, I go ahead and get another instance of the factory down here based off of 600 and get the alternate type of credit card of a gold and pass that back. So let's go ahead and run this, and you'll see that we get, for our first example, an AmexPlatinumCreditCard back, and for our second example, the VisaGoldCreditCard back. So our American Express Platinum gets returned for this set of code here, and the Visa Gold gets returned for this set of code here, which is exactly what we'd expect to do because as we go through our AbstractFactory, which is our CreditCardFactory, it just determines which factory to use. Again, it's not telling us how to create the card, it's not telling us what should be returned based off that, it just says hey, this is the factory you need based off the parameters you gave me. And there are no variables stored here, there are no values stored here, it's just a collection of multiple factories here. Notice our AmexFactory knows nothing of our VisaFactory, but they both tie in the common interface of our CreditCardFactory, and then we have a CreditCard and a Validator. Again, that common interface that's going to get passed back throughout all of our factories. Our CreditCard is actually an abstract class, not an interface, but just to mix it up I created a Validator interface that is, in fact, an interface. So to show you how both of those tie together from our CreditCardFactory, we have our common interface of the CreditCardFactory, as well as our common interface of our CreditCard and our Validator that gets returned. Then we go into our ConcreteFactory, and from the ConcreteFactory, we actually go ahead and create our instance and handle our validation. Now I've left the implementation of the validators and even the credit cards pretty simple, but it's because it has nothing really to do with the relevancy of the pattern. This is what we're trying to do with the pattern, showing that the creation logic is not known by the client, in fact, it's not even known by the AbstractFactory, it just knows to defer that to the right factory and then it handles it from there. So our individual factories know about their creation logic, and they've got grouping of similar features, so they both have validators, they both have credit cards. Another great example where I saw this used and where I've used it in the past is for different database types. Most people nowadays are using Hibernate or some ORM mapping tool like Hibernate, but I've done the same thing where I've had a database and I know I've had the same set of queries I want to run against tables, but I didn't know if I was running it against Oracle or SQL Server or MySQL. Same exact example. I'm going to have the same queries return. I just need to know what type of database I'm going against. I'm going to have the same type of credit card returned, I just need you to get the right factory back for me and

give me those queries from there. So this is a little example of how we want to use that factory and how it ties everything together.

Pitfalls

Some of the pitfalls with the AbstractFactory are its complexity. The AbstractFactory is the most complex of the creational design patterns and is definitely more difficult to implement. At some point in development, there is a runtime switch. The client has some influence what we do with the switch, and this makes some people afraid that the client knows too much about what is going on. It isn't a big issue, it can just be a hang up for some people architecturally. Although other patterns sometimes do this, it is definitely a pattern that contains other patterns. It's also very problem specific. Other patterns solve a broader problem, say for instance like the singleton, it just limits it to a single instance where the AbstractFactory is a grouping of factories. Another pitfall is that it usually starts off as a factory and then is refactored to an AbstractFactory. Most people don't often visualize using the AbstractFactory to begin with.

Contrast to Other Patterns

Rather than contrast the AbstractFactory with another type of creational pattern, I'm going to compare it with the factory pattern. A factory returns various instances and allows multiple constructors. It is interface-driven and it is adaptable to each environment more easily. All of these hold true for the AbstractFactory as well. It is implemented with a factory, it hides the underlying ConcreteFactory, and the AbstractFactory adds one more layer of abstraction to our environment. Another key difference between these two is that the AbstractFactory is typically built through composition. So, all of these features of the factory apply to the AbstractFactory with these additional, nice added bonuses of the AbstractFactory. It hides what factory we're using. It also abstracts our environment built with composition. I should note, though, it doesn't have to be implemented with the factory, it just usually is 99% of the time.

Summary

To recap, the AbstractFactory is a group of similar factories. It is quite complex a lot more so than the other creational design patterns that we've gone through. It is heavily abstracted. We utilize interfaces, subclasses, composition, and just general software contracts to develop this pattern and achieve various levels of abstraction. We typically don't do this with other design patterns. And it's typically written as a framework pattern meaning that this is part of a larger framework. Where the other patterns may solve a particular problem and do it in a generic way that we can use it anywhere in our code, the AbstractFactory is typically built as part of a larger framework inside of our code. The AbstractFactory is the last of the creational patterns that we have gone through in this Creational Design Pattern Series if you've been following these in order. I urge you to look at this in relation to the factory, even though it can be implemented with something else, this is usually written with the factory pattern in mind.

What Next?

What Next?

Well thank you for completing this design patterns course covering creational design patterns. If you're wondering what to look at next, I might recommend that you keep an eye out for the Design Patterns in Java covering structural design patterns, as well as the course covering behavioral patterns. I'm working on both of those at the time of this being published and it'll be out shortly. There are also all of my other courses on Pluralsight, Maven Fundamentals, Spring Fundamentals, Spring MVC, Spring MVC4, Spring JPA Hibernate, and Spring Security just to mention a few. Please explore those other courses if you've enjoyed this, and keep an eye out for those two design patterns courses coming within the next couple of months. Thank you.