In this package, you will find:
The author’s biography
A preview chapter from the book, Chapter no.3 "Creational Patterns"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Simon Timms is a developer who loves to write code. He writes in a variety of languages using a number of tools. For the most part, he develops web applications with .NET backends. He is very interested in visualizations and cloud computing. A background in build and system administration keeps him on the straight and narrow when it comes to DevOps.

He is the author of Social Data Visualization with HTML5 and JavaScript, Packt Publishing. He blogs at http://blog.simontimms.com/ and is also a frequent contributor to the Canadian Developer Connection, where his latest series explores evolving ASP.NET applications.

He is the President of the Calgary .NET User Group and a member of half a dozen others. He speaks on a variety of topics from DevOps to how the telephone system works.

He works as a web developer for Pacesetter Directional Drilling, the friendliest performance drilling company around.

I would like to thank my wonderful wife for all her support and my children who provided a welcome distraction from writing. I would also like to thank the Prime team at Pacesetter for their sense of humor and for putting up with me.

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
Mastering JavaScript Design Patterns

JavaScript is starting to become one of the most popular languages in the world. However, its history as a bit of a toy language, means that developers are tempted to ignore good design. Design patterns are a great tool to suggest some well-tried solutions.

What This Book Covers

This book is divided into two main parts, each of which contains a number of chapters. The first part of the book, which I'm calling Part 1, covers the classical design patterns, which are found in the GoF book. Part 2 looks at patterns, which are either not covered in the GoF book or the ones that are more specific to JavaScript.

Chapter 1, Designing for Fun and Profit, provides an introduction to what design patterns are and why we are interested in using design patterns. We will also talk about the history of JavaScript to give a historical context.

Chapter 2, Organizing Code, explains how to create the classical structures used to organize code: namespaces or modules and classes as JavaScript lack these constructs as first class citizens.

Chapter 3, Creational Patterns, covers the creational patterns outlined in the Gang of Four book. We'll discuss how these patterns apply to JavaScript, as opposed to the languages which were popular at the time when the Gang of Four wrote their book.

Chapter 4, Structural Patterns, examines the structural patterns from the Gang of Four book following on our look at creational patterns.

Chapter 5, Behavioral Patterns, covers the final set of patterns from the Gang of Four book that we'll examine. These patterns govern different ways to link classes together.

Chapter 6, Functional Programming, explains some of the patterns which can be found in functional programming languages. We'll look at how these patterns can be used in JavaScript to improve code.

Chapter 7, Model View Patterns, examines the confusing variety of different patterns to create single-page applications. We'll provide clarity and look at how to use libraries which use each of the existing patterns, as well as create their own lightweight framework.

Chapter 8, Web Patterns, covers a number of patterns which have specific applicability to web applications. We'll also look at some patterns around deploying code to remote runtimes such as the browser.

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
Chapter 9, *Messaging Patterns*, explains messaging which is a powerful technique to communicate inside, and even between, applications. We'll also look at some common structures around messaging and discuss why messaging is so useful.

Chapter 10, *Patterns for Testing*, focuses on some patterns which make for easier testing, giving you more confidence that your application is working as it should.

Chapter 11, *Advanced Patterns*, includes some patterns, such as aspect-oriented programming, that are rarely applied in JavaScript. We'll look at how these patterns can be applied in JavaScript and discuss if we should apply them.

Chapter 12, *ES6 Solutions Today*, discusses some of the tools available to allow you to use features from future versions of JavaScript today. We'll examine Microsoft's TypeScript as well as Traceur.

Appendix, *Conclusion*, covers what you have learned, in general, in the book, and you will be reminded of the goal of using patterns.

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
Creational Patterns

In the last chapter, we took a long look at how to fashion a class. In this chapter, we’ll look at how to create instances of classes. On the surface it seems like a simple concern, but how we create instances of a class can be of great importance.

We take great pains in creating our code such that it can be as decoupled as much as possible. Ensuring that classes have minimal dependence on other classes is the key to building a system that can change fluently with the changing needs of those using the software. Allowing classes to be too closely related means that changes ripple through them.

One ripple isn’t a huge problem but as you throw more and more changes into the mix, the ripples add up and create interference patterns. Soon the once placid surface is an unrecognizable mess of additive and destructive nodes. This problem also occurs in our applications: the changes magnify and interact in unexpected ways. One situation in which we tend to forget about coupling is when creating objects as can be seen in the following code:

```javascript
var Westeros;
(function (Westeros) {
    var Ruler = (function () {
        function Ruler() {
            this.house = new Westeros.Houses.Targaryen();
        }
        return Ruler;
    })();
    Westeros.Ruler = Ruler;
})(Westeros || (Westeros = {}));
```

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
You can see in this class that the `Ruler` variable's house is strongly coupled to the `Targaryen` class. If this were ever to change, then this tight coupling would have to change in a great number of places. This chapter discusses a number of patterns, which were originally presented in the Gang of Four book, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley. The goal of these patterns is to improve the degree of coupling in applications and increase the opportunities for code reuse.

The patterns are as follows:

- Abstract Factory
- Builder
- Factory Method
- Singleton
- Prototype

Of course not all of these are applicable to JavaScript, but we'll see all about that as we work through the creational patterns.

## Abstract Factory

The first pattern presented here is a method to create kits of objects without knowing the concrete types of the objects. Let's continue on with the system presented earlier for ruling a kingdom.

In the kingdom in question, the ruling house changes with some degree of frequency. In all likelihood, there is a degree of battling and fighting during the change of house but we'll ignore that for the moment. Each house will rule the kingdom differently. Some value peace and tranquility, and rule as benevolent leaders, while others rule with an iron fist. The rule of a kingdom is too large for a single individual, so the king defers some of his decisions to a second-in-command known as the **hand of the king**. The king is also advised on matters by a council, which consists of some of the more savvy lords and ladies of the land.
A diagram of the classes in our description is as follows:

Unified Modeling Language (UML) is a standardized language developed by the Object Management Group (OMG), which describes computer systems. There is vocabulary in the language to create user interaction diagrams, sequence diagrams, and state machines, among others. For the purposes of this book, we're most interested in class diagrams, which describe the relationship between a set of classes.

The entire UML class diagram vocabulary is extensive and is beyond the scope of this book. However, the Wikipedia article at https://en.wikipedia.org/wiki/Class_diagram acts as a great introduction, as does Derek Banas's excellent video tutorial on class diagrams at https://www.youtube.com/watch?v=3cmzqZzwNDM.

The issue is that with the ruling family, and even the member of the ruling family on the throne changing so frequently, coupling to a concrete family such as Targaryen or Lannister makes our application brittle. Brittle applications do not fare well in an ever-changing world.
An approach to fixing this is to make use of the **Abstract Factory** pattern. The Abstract Factory pattern declares an interface to create each of the various classes related to the ruling family, as shown in the following diagram:

The Abstract Factory class may have multiple implementations for each of the various ruling families. These are known as **concrete factories** and each of them will implement the interface provided by the Abstract Factory. The concrete factories, in return, will return concrete implementations of the various ruling classes. These concrete classes are known as **products**.

Let's start by looking at the code for the interface for the Abstract Factory.

No code? Well, actually that is exactly the case. The lack of classes in JavaScript precludes the need for interfaces to describe classes. Instead of having interfaces, we'll move directly to creating the classes, as shown in the following diagram:
Instead of interfaces, JavaScript trusts that the class you provide implements all the appropriate methods. At runtime, the interpreter will attempt to call the method you request, and call it, if it is found. The interpreter simply assumes that if your class implements the method, then it is that class. This is known as **duck typing**.

For More Information:  
www.packtpub.com/web-development/mastering-javascript-design-patterns
Duck typing

The name duck typing originates a post made by Alex Martelli in the year 2000 to the news group comp.lang.python, where he wrote:

In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.

I enjoy the possibility that Martelli took the term from the witch hunt sketch from Monty Python's search for the Holy Grail. Although I can find no evidence of that, I find it quite likely as the Python programming language takes its name from Monty Python.

Duck typing is a powerful tool in dynamic languages allowing, for much less overhead in implementing a class hierarchy. It does, however, introduce some uncertainty. If two classes implement an identically named method that have radically different meanings, then there is no way to know whether the one being called is the correct one. For example, consider the following code:

```javascript
class Boxer{
    function punch(){}
}
class TicketMachine{
    function punch(){}
}
```

Both classes have a `punch()` method but they clearly have different meanings. The JavaScript interpreter has no idea that they are different classes and will happily call `punch` on either class, even when one doesn't make sense.

In some dynamic languages, there is support for the generic method, which is called whenever an undefined method is called. Ruby, for instance, has `missing_method`, which has proven to be very useful in a number of scenarios. As of this writing, there is no support for `missing_method` in JavaScript. However, it may be possible to implement such a feature in ECMAScript 6.

For More Information:

www.packtpub.com/web-development/mastering-javascript-design-patterns
Implementation

To demonstrate an implementation of the Abstract Factory pattern, the first thing we'll need is an implementation of the King class. The following code provides that implementation:

```javascript
var KingJoffery = (function () {
    function KingJoffery() {
    }
    KingJoffery.prototype.makeDecision = function () {
        ...
    };
    KingJoffery.prototype.marry = function () {
        ...
    };
    return KingJoffery;
})();
```

This code does not include the module structure suggested in Chapter 2, Organizing Code. Including the boiler-plate module code in every example is tedious and you are all smart cookies, so you know to put this in modules if you're going to actually use it. The fully modularized code is available in the distributed source code.

This is just a regular concrete class and could really contain any implementation details. Similarly, we'll need an implementation of the HandOfTheKing class that is equally unexciting:

```javascript
var LordTywin = (function () {
    function LordTywin() {
    }
    LordTywin.prototype.makeDecision = function () {
    };
    return LordTywin;
})();
```

The concrete factory method looks like this:

```javascript
var LannisterFactory = (function () {
    function LannisterFactory() {
    }
    LannisterFactory.prototype.getKing = function () {
        return new KingJoffery();
    }
})();
```

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
The preceding code simply instantiates new instances of each of the required classes and returns them. An alternative implementation for a different ruling family would follow the same general form and might look like the following code:

```javascript
var TargaryenFactory = (function () {
    function TargaryenFactory() {
    }
    TargaryenFactory.prototype.getKing = function () {
        return new KingAerys();
    };,
    TargaryenFactory.prototype.getHandOfTheKing = function () {
        return new LordConnington();
    };,
    return TargaryenFactory;
})();
```

The implementation of the Abstract Factory pattern in JavaScript is much easier than in other languages. However, the penalty for this is that you lose the compiler checks, which force a full implementation of either the factory or the products. As we proceed through the rest of the patterns, you’ll notice that this is a common theme. Patterns that have a great deal of plumbing in statically typed languages are far simpler but create a greater risk of runtime failure. Appropriate unit tests or a JavaScript compiler can ameliorate this situation.

To make use of the Abstract Factory pattern, we’ll first need a class that requires the use of some ruling family. The following is the code for this class:

```javascript
var CourtSession = (function () {
    function CourtSession(abstractFactory) {
        this.abstractFactory = abstractFactory;
        this.COMPLAINT_THRESHOLD = 10;
    }
    CourtSession.prototype.complaintPresented = function (complaint) {
        if (complaint.severity < this.COMPLAINT_THRESHOLD) {
```
We can now call the `CourtSession` class and inject different functionality depending on which factory we pass in:

```javascript
var courtSession1 = new CourtSession(new TargaryenFactory());
courtSession1.complaintPresented({ severity: 8 });
courtSession1.complaintPresented({ severity: 12 });

var courtSession2 = new CourtSession(new LannisterFactory());
courtSession2.complaintPresented({ severity: 8 });
courtSession2.complaintPresented({ severity: 12 });
```

Despite the differences between a static language and JavaScript, this pattern remains applicable and useful in JavaScript applications. Creating a kit of objects, that work together is useful in a number of situations: any time when a group of objects needs to collaborate to provide functionality but may need to be replaced wholesale. It may also be a useful pattern when attempting to ensure that a set of objects be used together without substitutions.

**Builder**

In our fictional world, we sometimes have some rather complicated classes that need to be constructed. The classes contain different implementations of an interface depending on how they are constructed. In order to simplify the building of these classes and encapsulate the knowledge of building the class away from the consumers, a *builder* may be used. Multiple concrete builders reduce the complexity of the constructor in the implementation. When new builders are required, a constructor does not need to be added, a new builder just needs to be plugged in.

Tournaments are an example of a complicated class. Each tournament has a complicated setup involving the events, the attendees, and the prizes. Much of the setup for these tournaments is similar: each one has a joust, archery, and a melee. Creating a tournament from multiple places in the code means that the responsibility of knowing how to construct a tournament is distributed. If there is a need to change the initiation code, then it must be done in a lot of different places.

For More Information:  
Creational Patterns

Employing a Builder pattern avoids this issue by centralizing the logic necessary to build the object. Different concrete builders can be plugged into the builder to construct different complicated objects, as shown in the following diagram:

![Diagram of Creational Patterns]

**Implementation**

Let’s drop in and look at some of the code. To start with, we’ll create a number of utility classes, which will represent the parts of a tournament. We can see this in the following code:

```javascript
var Event = (function () {
    function Event(name) {
        this.name = name;
    }
    return Event;

})();
Westeros.Event = Event;

var Prize = (function () {
    function Prize(name) {
        this.name = name;
    }
    return Prize;

})();
Westeros.Prize = Prize;

var Attendee = (function () {
    function Attendee(name) {
        this.name = name;
    }
    return Attendee;

})();
Westeros.Attendee = Attendee;
```

For More Information:
The tournament itself is a very simple class as we don't need to assign any of the public properties explicitly, as shown in the following code:

```javascript
var Tournament = (function () {
    this.Events = [];
    function Tournament() {
    }
    return Tournament;
})();

Westeros.Tournament = Tournament;
```

We'll implement two builders that create different tournaments. This can be seen in the following code:

```javascript
var LannisterTournamentBuilder = (function () {
    function LannisterTournamentBuilder() {
    }
    LannisterTournamentBuilder.prototype.build = function () {
        var tournament = new Tournament();
        tournament.events.push(new Event("Joust"));
        tournament.events.push(new Event("Melee"));
        tournament.attendees.push(new Attendee("Jamie"));
        tournament.prizes.push(new Prize("Gold"));
        tournament.prizes.push(new Prize("More Gold"));
        return tournament;
    }
    return LannisterTournamentBuilder;
})();

Westeros.LannisterTournamentBuilder = LannisterTournamentBuilder;
```

```javascript
var BaratheonTournamentBuilder = (function () {
    function BaratheonTournamentBuilder() {
    }
    BaratheonTournamentBuilder.prototype.build = function () {
        var tournament = new Tournament();
        tournament.events.push(new Event("Joust"));
    }
    return BaratheonTournamentBuilder;
})();
```

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
Creational Patterns

tournament.events.push(new Event("Melee"));

tournament.attendees.push(new Attendee("Stannis"));
tournament.attendees.push(new Attendee("Robert"));

return tournament;

return BaratheonTournamentBuilder;
});
Westeros.BaratheonTournamentBuilder = BaratheonTournamentBuilder;

Finally, the director, or as we're calling it TournamentBuilder, simply takes a builder and executes it:

var TournamentBuilder = (function () {
    function TournamentBuilder() {
    }
    TournamentBuilder.prototype.build = function (builder) {
        return builder.build();
    }
    return TournamentBuilder;
})(function () {
    Westeros.TournamentBuilder = TournamentBuilder;
});

Again, you'll see that the JavaScript implementation is far simpler than the traditional implementation as there is no need for interfaces.

Builders need not return a fully realized object. This means that you can create a builder that partially hydrates an object, then allows the object to be passed onto another builder for it to finish. This approach allows us to divide the work of building an object amongst several classes with limited responsibility. In our preceding example, we could have a builder that is responsible for populating the events and another that is responsible for populating the attendees.

Does the builder pattern still make sense in view of JavaScript's prototype extension model? I believe so. There are still cases where a complicated object needs to be created according to different approaches.

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
Factory Method

We've already looked at the Abstract Factory and Builder patterns. The Abstract Factory pattern builds a family of related classes and the builder creates complicated objects using different strategies. The Factory Method pattern allows a class to request a new instance of an interface without the class making decisions about which implementation of the interface to use. The factory may use some strategy to select which implementation to return. This is shown in the following diagram:

```
+-----------------------------+    +-----------------------------+
| Client                      |    | ProductFactory              |
|    +-------------------------+    |    +-------------------------+    |
|    | Requests an instance from |    | CreateInstance              |
|    | Client Uses               |    |                            |
|    +-------------------------+    |    +-------------------------+    |
|   +-------------------------+    |    +-------------------------+    |
|   | <Interface>               |    | Product                     |
|   | Product                   |    |                            |
|   +-------------------------+    |    +-------------------------+    |
|       +---------------------+    |    +-------------------------+    |
|       | Implements              |    | Creates an instance of      |
|       +---------------------+    |    +-------------------------+    |
|       |                        |    |                             |
```

Sometimes, this strategy is simply to take a string parameter or to examine some global setting to act as a switch.

Implementation

In our example world of Westeros, there are plenty of times when we would like to defer the choice of implementation to a factory. Just like the real world, Westeros has a vibrant religious culture with dozens of competing religions worshiping a wide variety of gods. When praying in each religion, different rules must be followed. Some religions demand sacrifices while others demand only that a gift be given. The prayer class doesn't want to know about all the different religions and how to construct them.

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
Let's start with creating a number of different gods to which prayers can be offered. The following code creates three gods, including a default god to whom prayers fall if no other god is specified:

```javascript
var WateryGod = (function () {
    function WateryGod() {
    }
    WateryGod.prototype.prayTo = function () {
    return WateryGod;
    })();
Religion.WateryGod = WateryGod;

var AncientGods = (function () {
    function AncientGods() {
    }
    AncientGods.prototype.prayTo = function () {
    return AncientGods;
    })();
Religion.AncientGods = AncientGods;

var DefaultGod = (function () {
    function DefaultGod() {
    }
    DefaultGod.prototype.prayTo = function () {
    return DefaultGod;
    })();
Religion.DefaultGod = DefaultGod;
```

I've avoided any sort of implementation details for each god. You may imagine whatever traditions you want to populate the `prayTo` methods. There is also no need to ensure that each of the gods implements an `IGod` interface. Next, we'll need a factory that is responsible for constructing each of the different gods, as shown in the following code:

```javascript
var GodFactory = (function () {
    function GodFactory() {
    }
    GodFactory.Build = function (godName) {
        if (godName === "watery")
```
You can see that, in this example, we're taking in a simple string to decide how to create a god. It could be done via a global or via a more complicated object. In some polytheistic religions in Westeros, gods have defined roles as gods of courage, beauty, or some other aspect. The god to which one must pray is determined by not just the religion but the purpose of the prayer. We can represent this with a GodDeterminant class, as shown in the following code:

```javascript
var GodDeterminant = (function () {
    function GodDeterminant(religionName, prayerPurpose) {
        this.religionName = religionName;
        this.prayerPurpose = prayerPurpose;
    }
    return GodDeterminant;
})();
```

The factory would be updated to take this class instead of the simple string.

Finally, the last step is to see how this factory would be used. It is quite simple; we just need to pass in a string that denotes which religion we wish to observe and the factory will construct the correct god and return it. The following code demonstrates how to call the factory:

```javascript
var Prayer = (function () {
    function Prayer() {
    }
    Prayer.prototype.pray = function (godName) {
        GodFactory.Build(godName).prayTo();
    }
    return Prayer;
})();
```

Once again, there is certainly need for a pattern such as this in JavaScript. There are plenty of times when separating the instantiation from the use is useful. Testing the instantiation is also very simple thanks to the separation of concerns, and the ability to inject a fake factory to allow testing of the Prayer class is also easy.
Continuing the trend of creating simpler patterns without interfaces, we can ignore the interface portion of the pattern and work directly with the types, thanks to duck typing.

Factory Method is a very useful pattern; it allows classes to defer the selection of the implementation of an instantiation to another class. This pattern is very useful when there are multiple similar implementations such as the strategy pattern (see Chapter 5, Behavioral Patterns), and is commonly used in conjunction with the Abstract Factory pattern. The Factory Method pattern is used to build the concrete objects within a concrete implementation of the Abstract Factory. An Abstract Factory may contain a number of factory methods. Factory Method is certainly a pattern that remains applicable in the field of JavaScript.

**Singleton**

The Singleton pattern is perhaps the most overused pattern. It is also a pattern that has fallen out of favor in recent years. To see why people are starting to advise against using Singleton, let’s take a look at how the pattern works.

Singleton is used when a global variable is desirable, but Singleton provides protection against accidentally creating multiple copies of a complex object. It also allows for the deferral of object instantiation until the first use.

The UML diagram for Singleton is as follows:

![Singleton UML Diagram]

It is clearly a very simple pattern. The Singleton pattern acts as a wrapper around an instance of the class and the Singleton itself lives as a global variable. When accessing the instance, we simply ask Singleton for the current instance of the wrapped class. If the class does not yet exist within the Singleton, it is common to create a new instance at that time.

For More Information:

www.packtpub.com/web-development/mastering-javascript-design-patterns
Implementation

Within our ongoing example in the world of Westeros, we need to find a case where there can only ever be one of an item. Unfortunately, it is a land with frequent conflicts and rivalries, and so my first idea of using the king as the Singleton pattern is simply not going to fly. This split also means that we cannot make use of any of the other obvious candidates (capital city, queen, general...) as there may be many instances of each of those too. However, in the far north of Westeros, there is a giant wall constructed to keep an ancient enemy at bay. There is only one of these walls and it should pose no issue having it in the global scope.

Let's go ahead and create a Singleton class in JavaScript:

```javascript
var Westeros;
(function (Westeros) {
    var Wall = (function () {
        function Wall() {
            this.height = 0;
            if (Wall._instance)
                return Wall._instance;
            Wall._instance = this;
        }
        Wall.prototype.setHeight = function (height) {
            this.height = height;
        };
        Wall.prototype.getStatus = function () {
            console.log("Wall is " + this.height + " meters tall");
        };
        Wall.getInstance = function () {
            if (!Wall._instance) {
                Wall._instance = new Wall();
            }
            return Wall._instance;
        };
        Wall._instance = null;
        return Wall;
    })();
    Westeros.Wall = Wall;
})(Westeros || (Westeros = {}));
```

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
Creational Patterns

The code creates a lightweight representation of the Wall class. The Singleton pattern is demonstrated in the two highlighted sections. In a language like C# or Java, we would normally just set the constructor to be private so that it could only be called by the static getInstance method. However, we don't have that ability in JavaScript: constructors cannot be private. Thus, we do the best we can and return the current instance from the constructor. This may appear strange, but in the way we've constructed our classes the constructor is no different from any other method, so it is possible to return something from it.

In the second highlighted section, we set a static variable, _instance, to be a new instance of Wall when one is not already there. In the case that _instance already exists, we return that. In C# and Java, there will be a need for some complicated locking logic in this function to avoid race conditions as two different threads attempt to access the instance at the same time. Fortunately, there is no need to worry about this in JavaScript, where the multithreading story is different.

Disadvantages

Singletons have gained a somewhat bad reputation in the last few years. They are, in effect, glorified global variables. As we've discussed, global variables are ill conceived and the potential cause of numerous bugs. They are also difficult to test with unit tests, as the creation of the instance cannot easily be overridden. The single largest concern I have with them is that singletons have too much responsibility. They control not just themselves but also their instantiation. This is a clear violation of the single responsibility principle. Almost every problem that can be solved by using a Singleton pattern is better solved using some other mechanism.

JavaScript makes the problem even worse. It isn't possible to create a clean implementation of the Singleton pattern due to the restrictions on the constructor. This, coupled with the general problems around the Singleton pattern, lead me to suggest that this pattern should be avoided in JavaScript.

Prototype

The final creational pattern in this chapter is the Prototype pattern. Perhaps this name sounds familiar. It certainly should: it is the mechanism through which JavaScript inheritance is supported.
We looked at prototypes for inheritance but the applicability of prototypes need not be limited to inheritance. Copying existing objects can be a very useful pattern. There are numerous cases when being able to duplicate a constructed object is handy. For instance, maintaining a history of the state of an object is easily done by saving previous instances created by leveraging some sort of cloning.

**Implementation**

In Westeros, we find that members of a family are frequently very similar: as the adage goes, "like father, like son." As each generation is born, it is easier to create the new generation by copying and modifying an existing family member than to build one from scratch.

In Chapter 2, *Organizing Code*, we looked at how to copy existing objects and presented a very simple piece of code for cloning:

```javascript
function clone(source, destination) {
    for(var attr in source.prototype) {
        destination.prototype[attr] = source.prototype[attr];
    }
}
```

The following code can easily be altered to be used inside a class to return a copy of itself:

```javascript
var Westeros;
(function (Westeros) {
    (function (Families) {
        var Lannister = (function () {
            function Lannister() {
            }
            Lannister.prototype.clone = function () {
                var clone = new Lannister();
                for (var attr in this) {
                    clone[attr] = this[attr];
                }
                return clone;
            }
        }());
        return Lannister;
    }());
    Families.Lannister = Lannister;
})(Westeros.Families || (Westeros.Families = {}));
```

For More Information:

Creational Patterns

The highlighted section of the preceding code is the modified `clone` method. It can be used as follows:

```javascript
var janie = new Westeros.Families.Lannister();
janie.swordSkills = 9;
janie.charm = 6;
janie.wealth = 10;

var tyrion = janie.clone();

// tyrion.charm == 6
// tyrion.swordSkill == 9
```

The Prototype pattern allows for a complex object to be constructed only once and is then cloned into any number of objects that vary only slightly. If the source object is not complicated, there is little to be gained from taking a cloning approach. Care must be taken when using the prototype approach to think about dependent objects. Should the clone be a deep one?

Prototype is obviously a useful pattern and one that forms an integral part of JavaScript from the get go. As such, it is certainly a pattern that will see some use in any JavaScript application of appreciable size.

Hints and tips

Creational patterns allow for specialized behavior in creating objects. In many cases, such as the factory, they provide extension points into which crosscutting logic can be placed. This means the logic applies to a number of different types of objects. If you're looking for a way to inject, say, logging throughout your application, then being able to hook into a factory is of great utility.

Despite the utility of these creational patterns, they should not be used very frequently. The vast majority of your object instantiations should still be just the normal method of newing up objects. Although it is tempting to treat everything as a nail when you've got a new hammer, the truth is that each situation needs to have a specific strategy. All these patterns are more complicated than simply using `new`, and complicated code is more liable to have bugs than simple code. Use `new` whenever possible.

For More Information:
www.packtpub.com/web-development/mastering-javascript-design-patterns
Summary

This chapter presented a number of different strategies for creating objects. These methods provide abstractions over the typical methods for creating objects. The Abstract Factory pattern provides for a method to build interchangeable kits or collections of related objects. The Builder pattern provides for a solution to the telescoping parameters issue. It makes the construction of large complicated objects easier. The Factory Method pattern, which is a useful complement to Abstract Factory, allows for different implementations to be created through a static factory. Singleton is a pattern to provide a single copy of a class that is available to the entire solution. It is the only pattern we’ve seen so far that has presented some questions around applicability in modern software. The Prototype pattern is a commonly used pattern in JavaScript to build objects based on other existing objects.

We’ll continue our examination of classical design patterns in the next chapter by looking at structural patterns.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.