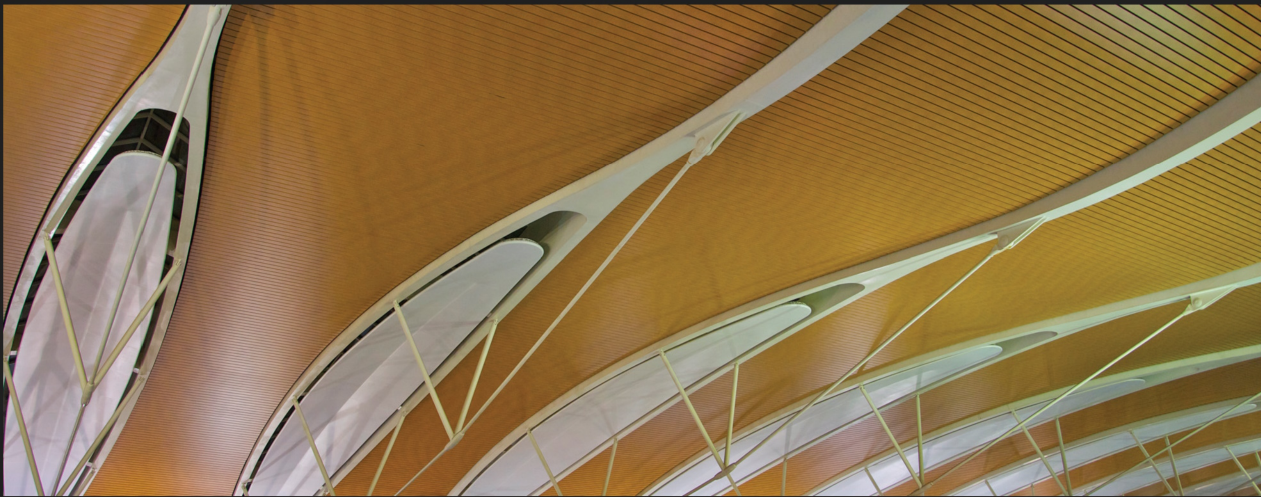

1ST EDITION

JavaScript Design Patterns

Deliver fast and efficient production-grade JavaScript applications at scale

HUGO DI FRANCESCO

JavaScript Design Patterns

Deliver fast and efficient production-grade JavaScript applications at scale

Hugo Di Francesco

packt

JavaScript Design Patterns

Group Product Manager: Rohit Rajkumar
Publishing Product Manager: Kushal Dave
Senior Content Development Editor: Feza Shaikh
Technical Editor: Simran Udasi
Copy Editor: Safis Editing
Project Coordinator: Aishwarya Mohan
Indexer: Subalakshmi Govindhan
Production Designer: Jyoti Kadam
Marketing Coordinators: Nivedita Pandey and Anamika Singh

First published: March 2024

Production reference: 1150224

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-80461-227-9

`www.packtpub.com`

To my wife, Amalia, for being my first supporter in all my endeavors.
To my daughter, Zoë, for making me want to show that the impossible sometimes is.

– Hugo Di Francesco

Contributors

About the author

Hugo Di Francesco is a software engineer who has worked extensively with JavaScript. He holds an MEng degree in mathematical computation from **University College London** (**UCL**). He has used JavaScript across the stack to create scalable and performant platforms at companies such as Canon and Elsevier, and in industries such as print on demand and mindfulness. He is currently tackling problems in the travel industry at Eurostar with Node.js, TypeScript, React, and Kubernetes, while running the eponymous Code with Hugo website. Outside of work, he is an international fencer, in the pursuit of which he trains and competes across the globe.

I want to thank all the people who have supported me in my life and writing journey, particularly my wife Amalia, and my family.

About the reviewers

Dr. Murugavel, a distinguished and versatile educator in the realms of computer science engineering and information technology. With over 13 years of enriching experience at renowned universities and an additional 8+ years dedicated to the dynamic field of data analytics, Dr. Murugavel stands as a beacon of expertise at the intersection of academia and technology.

His journey is marked by successive achievements, particularly in handling core subjects and programming languages, with a keen emphasis on practical knowledge. As a mentor and guide for major projects, Dr. Murugavel actively engages in groundbreaking research within his specialized field. His commitment to bridging theory and application has made him a valuable resource for students and researchers alike.

His technical proficiency extends across a spectrum of disciplines. He is well-versed in full stack web development, SQL, data analytics, Python, and BI tools, showcasing theoretical knowledge and a hands-on understanding of these technologies. His extensive portfolio includes the development of numerous applications using JSP, ASP, and ASP.NET, reflecting his prowess in both frontend and backend development.

In the realm of databases, he demonstrates versatility across MS-SQL Server, MySQL, MongoDB, Django, MS Access, Oracle, and FoxPro. His proficiency in various **Integrated Development Environments** (**IDEs**) and tools such as Anaconda, Visual Studio, GitHub, JBuilder, JCreator, MATLAB, Sublime 3, and Adobe Dreamweaver further solidifies his standing in the technological landscape.

In the realm of data science and **Business Intelligence** (**BI**) tools, his skills are extensive, encompassing PowerBI, DAX, VBA Macros for Excel, SSAS, and SSIS. His ability to harness these tools illuminates the path to insightful data analysis and visualization.

Shubham Thakur, a dynamic senior software engineer (A3 grade) at EPAM, specializes in technologies such as JavaScript, Angular, Next.js, Node, MySQL, MongoDB, AWS Cloud, and IoT. His expertise in these domains has significantly contributed to his project successes. He expresses deep gratitude to Priya for her unwavering love and to his brother, Yash, for his constant support. Shubham also acknowledges the profound impact of his mentors, Avnish Aggarwal, Yogesh Dhandekar, and Amit Jain, whose guidance has been instrumental in shaping his professional journey. Their mentorship has not only honed his technical skills but also enriched his approach to complex problem-solving in the tech industry.

Table of Contents

3

Leveraging Behavioral Design Patterns 53

Part 2: Architecture and UI Patterns

4

Exploring Reactive View Library Patterns 83

Part 3: Performance and Security Patterns

7

Asynchronous Programming Performance Patterns 183

8

Event-Driven Programming Patterns 209

9

Maximizing Performance – Lazy Loading and Code Splitting 233

Preface

Welcome! JavaScript design patterns are techniques that allow us to write more robust, scalable, and extensible applications in JavaScript. JavaScript is the main programming language available in web browsers and is one of the most popular programming languages with support beyond browsers.

Design patterns are solutions to common problems that can be reused. The most-written-about design patterns come from the world of object-oriented programming.

JavaScript's attributes as a lightweight, multi-paradigm, dynamic, single-threaded language give it different strengths and weaknesses to other mainstream programming languages. It's common for software engineers to use JavaScript in addition to being well versed in a different programming language. JavaScript's different gearing means that implementing design patterns verbatim can lead to non-idiomatic and under-performing JavaScript applications.

There are many resources on JavaScript and design patterns, but this book provides a cohesive and comprehensive view of design patterns in modern (ECMAScript 6+) JavaScript with real-world examples of how to deploy them in a professional setting. In addition to this complete library of patterns to apply to projects, this book also provides an overview of how to structure different parts of an application to deliver high performance at scale.

In this book, you will be provided with up-to-date guidance through the world of modern JavaScript patterns based on nine years of experience building and deploying JavaScript and React applications at scale at companies such as Elsevier, Canon, and Eurostar, delivering multiple system evolutions, performance projects, and a next-generation frontend application architecture.

Who this book is for

This book is for developers and software architects who want to leverage JavaScript and the web platform to increase productivity, software quality, and the performance of their applications.

Familiarity with software design patterns would be a plus but is not required.

The three main challenges faced by developers and architects who are the target audience of this content are as follows:

- They are familiar with programming concepts but not how to effectively implement them in JavaScript
- They want to structure JavaScript code and applications in a way that is maintainable and extensible
- They want to deliver more performance to the users of their JavaScript applications

What this book covers

Chapter 1, Working with Creational Design Patterns, covers creational design patterns, which help to organize object creation. We'll look at implementing the prototype, singleton, and factory patterns in JavaScript.

Chapter 2, Implementing Structural Design Patterns, looks at structural design patterns, which help to organize relationships between entities. We'll implement the proxy, decorator, flyweight, and adapter patterns in JavaScript.

Chapter 3, Leveraging Behavioral Design Patterns, delves into behavioral design patterns, which help to organize communication between objects. We'll learn about the observer, state, strategy, and visitor patterns in JavaScript.

Chapter 4, Exploring Reactive View Library Patterns, explores reactive view libraries, such as React, which have taken over the JavaScript application landscape. With these libraries come new patterns to explore, implement, and contrast.

Chapter 5, Rendering Strategies and Page Hydration, takes a look at optimizing page performance, which is a key concern nowadays. It's a concern both for improving the on-page conversion of customers and search engine optimization, since search engines such as Google take core web vitals into account.

Chapter 6, Micro Frontends, Zones, and Islands Architectures, explores micro frontends. Akin to the microservices movement in the service tier, micro frontends are designed to split a large surface area into smaller chunks that can be worked on and delivered at higher velocity.

Chapter 7, Asynchronous Programming Performance Patterns, looks at how JavaScript's single-threaded event-loop-based concurrency model is one of its greatest strengths but is often misunderstood or under-leveraged in performance-sensitive situations. Writing asynchronous-handling code in JavaScript in a performant and extensible manner is key to delivering a smooth user experience at scale.

Chapter 8, Event-Driven Programming Patterns, explores how event-driven programming in JavaScript is of paramount importance in security-sensitive applications as it is a way to pass information from and to different web contexts. Event-driven applications can often be optimized to enable better performance and scalability.

Chapter 9, Maximizing Performance – Lazy Loading and Code Splitting, deals with how, in order to maximize the performance of a JavaScript application, reducing the amount of unused JavaScript being loaded and interpreted is key. The techniques that can be brought to bear on this problem are called lazy loading and code splitting.

Chapter 10, Asset-Loading Strategies and Executing Code off the Main Thread, looks at how there are situations in the lifecycle of an application where loading more JavaScript or assets is inevitable. You will learn about asset-loading optimizations in the specific case of JavaScript, as well as other web resources, and finally how to execute JavaScript off the main browser thread.

To get the most out of this book

You will need to have prior experience with JavaScript and developing for the web. Some of the more advanced topics in the book will be of interest to developers with intermediate experience in building for the web with JavaScript.

Software/hardware covered in the book	Operating system requirements
Node.js 20+	Windows, macOS, or Linux
NPM v8+	Windows, macOS, or Linux
ECMAScript 6+	Windows, macOS, or Linux
React v16+	Windows, macOS, or Linux
Next.js	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/JavaScript-Design-Patterns`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: " In order to make the code easier to follow, we'll switch on the lowercased version of `tagName`."

A block of code is set as follows:

```
<script>
  // handle receiving messages from iframe -> parent
  const allowedMessageOrigins = ['http://127.0.0.1:8000'];
  window.addEventListener('message', (event) => {
    if (!allowedMessageOrigins.includes(event.origin)) {
      console.warn(
        `Dropping message due to non-allowlisted origin ${event.
origin}`,
        event,
      );
      return;
    }
    // no change to the rest of the message handler
  });
</script>
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "When opening the select, things seem to work ok, we're seeing the **Fruit:** prefix for all the options."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *JavaScript Design Patterns*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-80461-227-9`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Design Patterns

In this part, you will get an overview of design patterns and how they can be implemented effectively in modern JavaScript. You will learn how and when to implement creational, structural, and behavioral design patterns in the "classical" object-oriented way and how modern JavaScript features can be used to make this implementation more idiomatic to the language. Finally, you'll see real-world examples of design patterns being applied in the JavaScript ecosystem, thereby learning how to recognize them.

This part has the following chapters:

- *Chapter 1*, *Working with Creational Design Patterns*
- *Chapter 2*, *Implementing Structural Design Patterns*
- *Chapter 3*, *Leveraging Behavioral Design Patterns*

1

Working with Creational Design Patterns

JavaScript design patterns are techniques that allow us to write more robust, scalable, and extensible applications in JavaScript. JavaScript is a very popular programming language, in part due to its place as a way to deliver interactive functionality on web pages. The other reason for its popularity is JavaScript's lightweight, dynamic, multi-paradigm nature, which means that design patterns from other ecosystems can be adapted to take advantage of JavaScript's strengths. JavaScript's specific strengths and weaknesses can also inform new patterns specific to the language and the contexts in which it's used.

Creational design patterns give structure to object creation, which enables the development of systems and applications where different modules, classes, and objects don't need to know how to create instances of each other. The design patterns most relevant to JavaScript – the prototype, singleton, and factory patterns – will be explored, as well as situations where they're helpful and how to implement them in an idiomatic fashion.

We'll cover the following topics in this chapter:

- A comprehensive definition of creational design patterns and definitions for the prototype, singleton, and factory patterns
- Multiple implementations of the prototype pattern and its use cases
- An implementation of the singleton design pattern, eager and lazy initialization, use cases for singleton, and what a singleton pattern in modern JavaScript looks like
- How to implement the factory pattern using classes, a modern JavaScript alternative, and use cases

By the end of this chapter, you'll be able to identify when a creational design pattern is useful and make an informed decision on which of its multiple implementations to use, ranging from a more idiomatic JavaScript form to a classical form.

What are creational design patterns?

Creational design patterns handle object creation. They allow a consumer to create object instances without knowing the details of how to instantiate the object. Since, in object-oriented languages, instantiation of objects is limited to a class's constructor, allowing object instances to be created without calling the constructor is useful to reduce noise and tight coupling between the consumer and the class being instantiated.

In JavaScript, there's ambiguity when we discuss "object creation," since JavaScript's multi-paradigm nature means we can create objects without a class or a constructor. For example, in JavaScript this is an object creation using an object literal – `const config = { forceUpdate: true }`. In fact, modern idiomatic JavaScript tends to lean more toward procedural and function paradigms than object orientation. This means that creational design patterns may have to be adapted to be fully useful in JavaScript.

In summary, creational design patterns are useful in object-oriented JavaScript, since they hide instantiation details from consumers, which keeps coupling low, thereby allowing better module separation.

In the next section, we'll encounter our first creational design pattern – the prototype design pattern.

Implementing the prototype pattern in JavaScript

Let's start with a definition of the prototype pattern first.

The prototype design pattern allows us to create an instance based on another existing instance (our prototype).

In more formal terms, a `prototype` class exposes a `clone()` method. Consuming code, instead of calling `new SomeClass`, will call `new SomeClassPrototype(someClassInstance).clone()`. This method call will return a `new SomeClass` instance with all the values copied from `someClassInstance`.

Implementation

Let's imagine a scenario where we're building a chessboard. There are two key types of squares – white and black. In addition to this information, each square contains information such as its row, file, and which piece sits atop it.

A `BoardSquare` class constructor might look like the following:

```
class BoardSquare {
  constructor(color, row, file, startingPiece) {
    this.color = color;
    this.row = row;
```

```
    this.file = file;
  }
}
```

A set of useful methods on `BoardSquare` might be `occupySquare` and `clearSquare`, as follows:

```
class BoardSquare {
  // no change to the rest of the class
  occupySquare(piece) {
    this.piece = piece;
  }
  clearSquare() {
    this.piece = null;
  }
}
```

Instantiating `BoardSquare` is quite cumbersome, due to all its properties:

```
const whiteSquare = new BoardSquare('white');
const whiteSquareTwo = new BoardSquare('white');
// ...
const whiteSquareLast = new BoardSquare('white');
```

Note the repetition of arguments being passed to `new BoardSquare`, which will cause issues if we want to change all board squares to black. We would need to change the parameter passed to each call of `BoardSquare` is one by one for each `new BoardSquare` call. This can be quite error-prone; all it takes is one hard-to-find mistake in the `color` value to cause a bug:

```
const blackSquare = new BoardSquare('black');
const blackSquareTwo = new BoardSquare('black');
// ...
const blackSquareLast = new BoardSquare('black');
```

Implementing our instantiation logic using a classical prototype looks as follows. We need a `BoardSquarePrototype` class; its constructor takes a `prototype` property, which it stores on the instance. `BoardSquarePrototype` exposes a `clone()` method that takes no arguments and returns a `BoardSquare` instance, with all the properties of `prototype` copied onto it:

```
class BoardSquarePrototype {
  constructor(prototype) {
    this.prototype = prototype;
  }
  clone() {
    const boardSquare = new BoardSquare();
    boardSquare.color = this.prototype.color;
```

```
    boardSquare.row = this.prototype.row;
    boardSquare.file = this.prototype.file;
    return boardSquare;
  }
}
```

Using `BoardSquarePrototype` requires the following steps:

1. First, we want an instance of `BoardSquare` to initialize – in this case, with `'white'`. It will then be passed as the `prototype` property during the `BoardSquarePrototype` constructor call:

```
const whiteSquare = new BoardSquare('white');
const whiteSquarePrototype = new BoardSquarePrototype
  (whiteSquare);
```

2. We can then use `whiteSquarePrototype` with `.clone()` to create our copies of `whiteSquare`. Note that `color` is copied over but each call to `clone()` returns a new instance.

```
const whiteSquareTwo = whiteSquarePrototype.clone();
// ...
const whiteSquareLast = whiteSquarePrototype.clone();

console.assert(
  whiteSquare.color === whiteSquareTwo.color &&
    whiteSquareTwo.color === whiteSquareLast.color,
  'Prototype.clone()-ed instances have the same color
     as the prototype'
);
console.assert(
  whiteSquare !== whiteSquareTwo &&
    whiteSquare !== whiteSquareLast &&
    whiteSquareTwo !== whiteSquareLast,
  'each Prototype.clone() call outputs a different
     instances'
);
```

Per the assertions in the code, the cloned instances contain the same value for `color` but are different instances of the `Square` object.

A use case

To illustrate what it would take to change from a white square to a black square, let's look at some sample code where `'white'` is not referenced in the variable names:

```
const boardSquare = new BoardSquare('white');
const boardSquarePrototype = new BoardSquarePrototype(boardSquare);

const boardSquareTwo = boardSquarePrototype.clone();
// ...
const boardSquareLast = boardSquarePrototype.clone();

console.assert(
  boardSquareTwo.color === 'white' &&
    boardSquare.color === boardSquareTwo.color &&
    boardSquareTwo.color === boardSquareLast.color,
  'Prototype.clone()-ed instances have the same color as
     the prototype'
);
console.assert(
  boardSquare !== boardSquareTwo &&
    boardSquare !== boardSquareLast &&
    boardSquareTwo !== boardSquareLast,
  'each Prototype.clone() call outputs a different
    instances'
);
```

In this scenario, we would only have to change the `color` value passed to `BoardSquare` to change the color of all the instances cloned from the prototype:

```
const boardSquare = new BoardSquare('black');
// rest of the code stays the same
console.assert(
  boardSquareTwo.color === 'black' &&
    boardSquare.color === boardSquareTwo.color &&
    boardSquareTwo.color === boardSquareLast.color,
  'Prototype.clone()-ed instances have the same color as
     the prototype'
);
console.assert(
  boardSquare !== boardSquareTwo &&
    boardSquare !== boardSquareLast &&
    boardSquareTwo !== boardSquareLast,
  'each Prototype.clone() call outputs a different
```

```
    instances'
);
```

The prototype pattern is useful in situations where a "template" for the object instances is useful. It's a good pattern to create a "default object" but with custom values. It allows faster and easier changes, since they are implemented once on the template object but are applied to all `clone()`-ed instances.

Increasing robustness to change in the prototype's instance variables with modern JavaScript

There are improvements we can make to our prototype implementation in JavaScript.

The first is in the `clone()` method. To make our prototype class robust to changes in the prototype's constructor/instance variables, we should avoid copying the properties one by one.

For example, if we add a new `startingPiece` parameter that the `BoardSquare` constructor takes and sets the `piece` instance variable to, our current implementation of `BoardSquarePrototype` will fail to copy it, since it only copies `color`, `row`, and `file`:

```
class BoardSquare {
  constructor(color, row, file, startingPiece) {
    this.color = color;
    this.row = row;
    this.file = file;
    this.piece = startingPiece;
  }
  // same rest of the class
}

const boardSquare = new BoardSquare('white', 1, 'A',
  'king');
const boardSquarePrototype = new BoardSquarePrototype
  (boardSquare);
const otherBoardSquare = boardSquarePrototype.clone();

console.assert(
  otherBoardSquare.piece === undefined,
  'prototype.piece was not copied over'
);
```

> **Note**
>
> Reference for `Object.assign`: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign`.

If we amend our `BoardSquarePrototype` class to use `Object.assign(new BoardSquare(), this.prototype)`, it will copy all the enumerable properties of `prototype`:

```
class BoardSquarePrototype {
  constructor(prototype) {
    this.prototype = prototype;
  }
  clone() {
    return Object.assign(new BoardSquare(), this.prototype);
  }
}

const boardSquare = new BoardSquare('white', 1, 'A',
  'king');
const boardSquarePrototype = new BoardSquarePrototype
  (boardSquare);
const otherBoardSquare = boardSquarePrototype.clone();

console.assert(
  otherBoardSquare.piece === 'king' &&
    otherBoardSquare.piece === boardSquare.piece,
  'prototype.piece was copied over'
);
```

The prototype pattern without classes in JavaScript

For historical reasons, JavaScript has a prototype concept deeply embedded into the language. In fact, classes were introduced much later into the ECMAScript standard, with ECMAScript 6, which was released in 2015 (for reference, ECMAScript 1 was published in 1997).

This is why a lot of JavaScript completely forgoes the use of classes. The JavaScript "object prototype" can be used to make objects inherit methods and variables from each other.

One way to clone objects is by using the `Object.create` to clone objects with their methods. This relies on the JavaScript prototype system:

```
const square = {
  color: 'white',
  occupySquare(piece) {
    this.piece = piece;
  },
  clearSquare() {
    this.piece = null;
  },
```

```
};
const otherSquare = Object.create(square);
```

One subtlety here is that `Object.create` does not actually copy anything; it simply creates a new object and sets its prototype to `square`. This means that if properties are not found on `otherSquare`, they're accessed on `square`:

```
console.assert(otherSquare.__proto__ === square, 'uses JS
  prototype');

console.assert(
  otherSquare.occupySquare === square.occupySquare &&
    otherSquare.clearSquare === square.clearSquare,
  "methods are not copied, they're 'inherited' using the
     prototype"
);

delete otherSquare.color;
console.assert(
  otherSquare.color === 'white' && otherSquare.color ===
    square.color,
  'data fields are also inherited'
);
```

A further note on the JavaScript prototype, and its existence before classes were part of JavaScript, is that subclassing in JavaScript is another syntax for setting an object's prototype. Have a look at the following `extends` example. `BlackSquare extends Square` sets the `prototype.__proto__` property of `BlackSquare` to `Square.prototype`:

```
class Square {
  constructor() {}
  occupySquare(piece) {
    this.piece = piece;
  }
  clearSquare() {
    this.piece = null;
  }
}

class BlackSquare extends Square {
  constructor() {
    super();
    this.color = 'black';
  }
```

```
}

console.assert(
  BlackSquare.prototype.__proto__ === Square.prototype,
  'subclass prototype has prototype of superclass'
);
```

In this section, we learned how to implement the prototype pattern with a prototype class that exposes a `clone()` method, which code situations the prototype patterns can help with, and how to further improve our prototype implementation with modern JavaScript features. We also covered the JavaScript "prototype," why it exists, and its relationship with the prototype design pattern.

In the next part of the chapter, we'll look at another creational design pattern, the singleton design pattern, with some implementation approaches in JavaScript and its use cases.

The singleton pattern with eager and lazy initialization in JavaScript

To begin, let's define the singleton design pattern.

The singleton pattern allows an object to be instantiated only once, exposes this single instance to consumers, and controls the instantiation of the single instance.

The singleton is another way of getting access to an object instance without using a constructor, although it's necessary for the object to be designed as a singleton.

Implementation

A classic example of a singleton is a logger. It's rarely necessary (and often, it's a problem) to instantiate multiple loggers in an application. Having a singleton means the initialization site is controlled, and the logger configuration will be consistent across the application – for example, the log level won't change depending on where in the application we call the logger from.

A simple logger looks something as follows, with a constructor taking `logLevel` and `transport`, and an `isLevelEnabled` private method, which allows us to drop logs that the logger is not configured to keep (for example, when the level is `warn` we drop `info` messages). The logger finally implements the `info`, `warn`, and `error` methods, which behave as previously described; they only call the relevant `transport` method if the level is "enabled" (i.e., "above" what the configured log level is).

The possible `logLevel` values that power `isLevelEnabled` are stored as a static field on `Logger`:

```
class Logger {
  static logLevels = ['info', 'warn', 'error'];
  constructor(logLevel = 'info', transport = console) {
```

```
    if (Logger.#loggerInstance) {
      throw new TypeError(
        'Logger is not constructable, use getInstance()
           instead'
      );
    }
    this.logLevel = logLevel;
    this.transport = transport;
  }
  isLevelEnabled(targetLevel) {
    return (
      Logger.logLevels.indexOf(targetLevel) >=
      Logger.logLevels.indexOf(this.logLevel)
    );
  }
  info(message) {
    if (this.isLevelEnabled('info')) {
      return this.transport.info(message);
    }
  }
  warn(message) {
    if (this.isLevelEnabled('warn')) {
      this.transport.warn(message);
    }
  }
  error(message) {
    if (this.isLevelEnabled('error')) {
      this.transport.error(message);
    }
  }
}
```

In order to make `Logger` a singleton, we need to implement a `getInstance` static method that returns a cached instance. In order to do, this we'll use a static `loggerInstance` on `Logger`. `getInstance` will check whether `Logger.loggerInstance` exists and return it if it does; otherwise, it will create a new `Logger` instance, set that as `loggerInstance`, and return it:

```
class Logger {
  static loggerInstance = null;
  // rest of the class
  static getInstance() {
    if (!Logger.loggerInstance) {
      Logger.loggerInstance = new Logger('warn', console);
    }
```