Object Oriented Programming

- often used to model representation of objects in the real world
- encapsulation, polymorphism, inheritance

Encapsulation

- the inner workings are kept hidden inside the object & only the essential functionalities are exposed to the end user
- this involves keeping all the programming logic inside an object & making methods available to implement the functionality w/out the outside world needing to know how its done

Polymorphism

- the same process can be used for different objects
- various objects can share the same method, but also have the ability to override shared methods w/ a more specific implementation

Inheritance

- taking the features of one object then adding some new features
- take an object that already exists & inherit all its properties & methods
- improve on its functionality by adding new properties & methods

Many object-oriented languages, such as Java and Ruby, are known as class-based languages. This is because they use a class to define a blueprint for an object. Objects are then created as an instance of that class, and inherit all the properties and methods of the class

JavaScript didn't have classes before ES6, and used the concept of using actual objects as the blueprint for creating more objects. This is known as a prototype-based language

Constructor Functions

- an alternative way to create objects is to use a constructor function
- a function that defines the properties & methods of an object

When Parentheses Aren't Required

- parentheses are not required when instantiating a new object using a constructor function
- parentheses are required if any default arguments need to be provided

Built-In Constructor Functions

- Object
- Array
- Function

"The easiest way to create a new object is to use the literal syntax:
const literalObject = {};
<< {}

It is also possible to use the Object constructor function:
constructedObject = new Object();
<< {}

A literal is still considered to be an instance of the Object constructor:
literalObject instanceof Object;
<< true

Similarly, the easiest way to create an array is to use the literal syntax, like so:
const literalArray = [1,2,3];
<< [1, 2, 3]

But an alternative is to use the Array constructor function:
constructedArray = new Array(1,2,3);
<< [1, 2, 3]

Array constructor functions exhibit some strange behavior regarding the arguments supplied, however. If only one argument is given, it doesn't create an array with that argument as the first element, as you might expect. It sets the array's length property instead, and returns an array full of undefined!

"This results in an error being thrown if a floating point decimal number is provided as an argument, because the length of an array must be an integer"

"This behavior is another reason why it's recommended to always use literals to create arrays"

ES6 Class Declarations

"ES6 introduced the new class declaration syntax that does exactly the same thing as a constructor function, but looks much similar to writing a class in a class-based programming language"

Capitalizing Constructor Functions

"By convention, the names of constructor functions or class declarations are capitalized, which is the convention used for classes in class-based programming languages"

"The class declaration syntax works in exactly the same way as the constructor function syntax, because it's actually just syntactic sugar that is implemented in the same way in the background."

"The ES6 class declarations are preferable to the constructor function syntax because they are more succinct, easier to read and all code in a class definition is implicitly in strict mode, so doesn't need the 'use strict' statement. Using ES6 class declarations also avoids a number of pitfalls associated with constructor functions"

The Constructor Property

- All objects have a constructor property that returns the constructor function that created it
- when an object literal is used to create a new object, we can see that in the background, the Object constructor function is being used

"We can use the constructor property to instantiate a copy of an object, without having to reference the actual constructor function or class declaration directly"

Static Methods

"The static keyword can be used in class declarations to create a static method. These are sometimes called class methods in other programming languages. A static method is called by the class directly rather than by instances of the class"

- static methods are not available to instances of the class

Prototypal Inheritance

"JavaScript uses a prototypal inheritance model. This means that every class has a prototype property that is shared by every instance of the class. So any properties or methods of a class's prototype can be accessed by every object instantiated by that class"

The Prototype Property

"When creating a class, you would normally add any default properties and methods to the class declaration. But what if you want to augment the class with extra methods and properties after it has been created? It turns out that you can still do this using the prototype property of the class. This is particularly useful if you don't have access to the class declaration, but still want to add properties and methods to the class.

All classes and constructor functions have a prototype property that returns an object:

Turtle.prototype;
<< Turtle {}"

Finding Out The Prototype

- a # of ways to find the prototype of an object:
  - go via the constructor function's prototype property
  - use the Object.getPrototypeOf() method, which takes an object as a parameter
  - many JS engines also support non-standard __proto__ property (known as dunder proto, short for double underscore proto)
  - the __proto__ property can also be used to set the prototype of an object by assignment, but its use has been deprecated in favor of the setPrototypeOf() method
  - every object has a isPrototypeOf() method that returns a boolean to check if its the prototype of an instance

Own Properties & Prototype Properties

“Prototype properties are shared by every instance of the Turtle class. This means they’ll all have a weapon property, and it will always be the same value. If we create another instance of the Turtle class, we’ll see that it also inherits a weapon property that has the same value of 'Hands':

```
const don = new Turtle('Donatello');
<< Turtle { name: 'Donatello' }

don.weapon;
<< 'Hands'
```

Every time an instance of the Turtle class queries the weapon property, it will return 'Hands'. This value is the same for all the instances and only exists in one place — as a property of the prototype. This means that it only exists in memory in one place, which is more efficient than each instance having its own value. This is particularly useful for any properties that are the same.”

The Prototype is Live!

“The prototype object is live, so if a new property or method is added to the prototype, any instances of its class will inherit the new properties and methods automatically, even if that instance has already been created”

Overwriting A Prototype

- not possible to overwrite the prototype by assigning it to a new object literal if class declarations are used
- it is possible IF constructor functions are used

it can cause a lot of headaches if you accidentally redefine the prototype. This is because any instances that have already been created will retain the properties and methods of the old prototype, but will not receive any of the new properties and methods that are subsequently added to the redefined prototype.

↑another reason why its recommended to use class declarations instead of constructor functions

Overwriting Prototype Properties

- object instance can overwrite any properties or methods inherited from its prototype by simply assigning a new value to them

“When a property or method is called, the JavaScript engine will check to see if an object has its own property or method. If it does, it will use that one; otherwise, it will continue up the prototype chain until it finds a match or reaches the top of the chain”

What Should the Prototype Be Used For?

“The prototype can be used to add any new properties and methods after the class has been declared. It should be used to define any properties that will remain the same for every instance of the class. The weapon example was unsuitable because all the turtles use a different weapon (we just used it in the example above to demonstrate overwriting). They do, however, like the same food — pizza! This makes a good candidate for a prototype property, if it wasn't included in the original class declaration”

Use with care when Setting Default Values

"Be careful when using the prototype to set default values. They are shallow (there's more about shallow and deep copies later in the chapter). Any changes to an array or object made by an instance will be reflected in the prototype, and therefore shared between all instances.

A golden rule to remember is: Never use arrays or objects as a default value in prototype.

This is not a problem if arrays or objects are set as default values from within the constructor function in the class declaration."

"To summarize, the following points should be considered when using classes and prototypes to create instances:

- Create a class declaration that deals with any initialization, shared properties and methods.
- Any extra methods and properties that need to be augmented to the class declaration after it's been defined can be added to the prototype. These will be added to all instances, even those that have already been created.
- Add any properties or methods that are individual to a particular instance can be augmented using assignment to that object (a mixin could be used to add multiple properties at once, as we'll see later).
- Be careful when overwriting the prototype completely — the constructor class needs to be reset.

Public & Private Methods

"an object's methods are public in JavaScript. Methods and properties are said to be public because they can be queried directly and changed by assignment. The dynamic nature of the language means that an object's properties and methods can be changed after it has been created"

"giving users or external services too much access to properties and methods could be a recipe for disaster!"

"we can use the concept of variable scope to keep some properties and methods private inside of a class declaration. This will prevent them from being accessed or changed. Instead, we will provide a getter method to return the values of any private properties.

Inheritance

The prototype chain

- peer further down the prototype chain, by calling the `Object.getPrototypeOf()` method recursively

The Object Constructor

- all objects ultimately inherit from the prototype of the `Object()` constructor function

"When an object calls a method, the JavaScript engine will check to see if the object has that method. If it doesn't, it will check if the object's prototype has the method. If not, it will check whether the prototype's prototype has it. This continues all the way up the prototype chain, until it reaches the prototype of the Object() constructor function, from which all objects in JavaScript inherit. If the prototype of Object() is without the method, an error will be returned saying the object doesn't exist"

"The prototype of the Object constructor function has a large number of methods that are inherited by all objects. The reason why the prototype appears as an empty object literal is because all of its methods are not enumerable"

Enumerable Properties

"Properties of objects in JavaScript are said to be enumerable or non-enumerable. If they aren't enumerable, this means they will not show up when a for-in loop is used to loop through an object's properties and methods.

There is a method called propertyIsEnumerable() that every object has (because it's a method of Object.prototype) that can be used to check if a property is enumerable. We can see in the following example that the eat() method we created earlier is enumerable (in fact, all properties and methods that are created by assignment are enumerable):

```
Turtle.prototype.propertyIsEnumerable('eat');
<< true"
```

- good practice is for all built-in methods to be non-enumerable, and any user-defined methods to be made enumerable
- This is so all the built-in methods don't keep showing up when looking at an Object's methods, but user-defined methods are easy to find

Inheritance using extends

- a class can inherit from another class using the extends keyword in a class declaration

Polymorphism

- different objects can have the same method, but implement it in different ways

"Object.prototype object has a toString() method that is shared by all objects. This means every object created in JavaScript will have a toString() method."

"Polymorphism means that objects are able to override this method with a more specific implementation. So although every object has a toString() method, the way it's implemented can vary between different objects"

Numbers, Strings, & Booleans

"The number, string, and boolean primitive types that we met way back in Chapter 2 have their own corresponding constructor functions: Number, String, and Boolean respectively."

"Primitives are actually without their own methods. The primitive wrapper objects Number, String, and Boolean are used in the background to provide primitive values with methods. When a method is called on a primitive value, JavaScript creates a wrapper object for the primitive, which converts it into an object and then calls the method on the object"

"The toString() method is used by a number of built-in functions in the background. It can be used without fear of causing an error because every object has the method, as it's inherited from Object.prototype."

"One example of a function that uses the toString() method is the console.log() method. If an object is given as an argument to this method that isn't a string, it will call toString() on that object in the background and display the return value in the console. For example, the code:

```
console.log([1,2,3]);
<< [ 1, 2, 3 ]".
```

"The toString() method is a good demonstration of polymorphism, since different objects have the same method but implement it differently. The advantage of this is that higher-level functions are able to call a single method, even though it may be implemented in various ways."

Adding Methods to Built-in Objects

"possible to add more methods to the prototype of JavaScript's built-in objects — such as Number, String, and Array — to add more functionality. This practice is known as monkey-patching, but it's mostly frowned upon in the JavaScript community, despite it being an incredibly powerful technique"

"Arrays are powerful objects, but seem to have some basic methods missing in JavaScript that are found in other languages. We can add a first() and last() methods that return the first and last items in the array"

"A useful example of monkey-patching is to add support for methods that are part of the specification, but not supported natively in some browsers. An example is the trim() method, which is a method of String.prototype, so all strings should inherit it. It removes all whitespace from the beginning and the end of strings, but unfortunately this method is not implemented in Internet Explorer version 8 or below. This can be rectified using this polyfill code that will use the built in String.prototype.trim if it exists, and if it doesn't, it monkey-patches the String prototype with the function provided"

"monkey-patching built-in objects can seem a good way to add extra or missing functionality, it can also add unexpected behavior. The current consensus in the JS community is that this shouldn't be done, so you should avoid monkey-patching any of the built-in object constructor prototypes, unless you have a very good reason. Further problems could occur if the method you've added is then implemented natively in the language"

"the suggested way is to check for built-in methods first then try to mimic the built-in functionality from the specification, like in the trim() polyfill shown above. This can still be problematic, though, if the specification changes and is different from your implementation. Remember also that you can never guarantee a method won't be implemented at some point in the future."

- alternative way to avoid causing problems is to use extends to subclass a built class & create your own class
- an obvious problem w/ this is that you would have to use this more unwieldy syntax instead of array literals
  - has the advantage of not interfering w/ the built-in array class @ all

Property Attributes & Descriptors

- each property has a # of attributes that provide info about the property
- these attributes are stored in a property descriptor, which is an object that contains values of each attribute

"All object properties have the following attributes stored in a property descriptor:

- value — This is the value of the property and is undefined by default
- writable — This boolean value shows whether a property can be changed or not, and is false by default
- enumerable — this boolean value shows whether a property will show up when the object is displayed in a for in loop, and is false by default
- configurable — this boolean value shows whether you can delete a property or change any of its attributes, and is false by default."

- possible to set each of the property attributes by using a property descriptor

"The property descriptor for the name property might look like this:

{ value: 'DAZ', writable: true, enumerable: true, configurable: true }

- disadvantage w/ this is that it can only be used to set the value attribute of the property
- not possible to set the writable, enumerable, configurable attributes in this manner
- these will be set as true when an assignment is made

Getting & Setting Property Descriptors

"The Object() constructor function has a number of methods for getting and defining property descriptors. We can see these values using the Object.getOwnPropertyDescriptor() method"

"Instead of using assignment, we can add properties to an object using the Object.defineProperty() method. This provides more fine-grained control when adding new properties, as it allows each attribute to be set. The first argument is the object to which you want to add the property, followed by a property descriptor containing the attributes you want to set. Any attributes left out will take the default values"

- Object is returned w/ the new property added

Getters & Setters

"object property descriptor can have get() and set() methods instead of a value attribute. All objects must have one or the other, they can't have both. The get() and set() methods can be used to control how a property is set using assignment and the value that is returned when a property is queried"

- particular useful if a property relies on a value of another property
- allow for much more fine-grained control over how assignments work
- means we can change the way assignment works & use the get() method to return anything we like, regardless of what value was set using assignment
- get & set property descriptors are particularly useful for controlling the getting & setting of a properties in classes

"getter and setter methods give you much more power in controlling the way property assignment works. However, they should be used sparingly and with care, as changing the expected behavior of an assignment has the potential to cause a lot of confusion"

Creating Objects from Other Objects

"It's possible to avoid using classes altogether, and create new objects based on another object that acts as a 'blueprint' or prototype instead"

"The Object() constructor function has a method called create that can be used to create a new object that is an exact copy of the object that is provided as an argument. The object that is provided as the argument acts as the prototype for the new object"

"An alternative way is to add a second argument to the Object.create() method containing properties that are to be added to the new object:

const jimmy = Object.create(Human, { name: { value: 'Jimmy Olsen', enumerable: true }, job: { value: 'Photographer', enumerable: true } });

This method is a little unwieldy as the properties have to be added using property descriptors, making the syntax awkward and overly verbose. It's often easier to create the object, then add each new property one by one. This can be made quicker using the mixin() function that is covered later."

The Human Object is a Prototype

- for any objects created using it as an argument & remember that prototypes are live
  - meaning any changes made to the Human object will be reflected in all the objects created this way

Object-Based Inheritance

"The Human object can also act like a 'super-class', and become the prototype of another object called Superhuman. This will have all the properties and methods that the Human object has, but with some extra methods"

- a new object can be created & initialized by a single line by adding the call to the init() method @ the end of the line that creates the object.
- ↑ this is an example of chaining

Object Prototype Chain

- creating objects from objects will create a prototype chain

"Every time a new object is created using the Object.create() method, the new object inherits all the properties and methods from the parent object, which becomes the new object's prototype"

Mixins

A mixin is a way of adding properties and methods of some objects to another object without using inheritance. It allows more complex objects to be created by 'mixing' basic objects together"

"Basic mixin functionality is provided by the Object.assign() method. This will assign to the object provided as the first argument all of the properties from any objects provided as further arguments:"

"There is a problem with this method, however. If any of the properties being mixed in are arrays or nested objects, only a shallow copy is made, which can cause a variety of issues"

Copying By Reference

"Arrays and functions are objects, so whenever they're copied by assignment they will just point to the same object. And when one changes, they all change. This is known as making a shallow copy of an object. A deep or hard copy will create a completely new object that has all the same properties as the old object. The difference is that when a hard copy is changed, the original remains the same. But when a shallow copy is changed, the original changes too."

"To avoid only a shallow copy, we're going to create our own mixin() function that will assign all properties of an object to another object as a deep copy.

This means that every object will inherit this method and be able to use it to augment itself with the properties and methods from other objects"

Using Mixin to Add Properties

"One use for the mixin() function is to add a large number of properties to an object all at once"

Using Mixins to Create a copy() Function

"Another use of the mixin() function is to create a copy() method that can be used to make an exact, deep copy of an object"

"The mixin() function is then used to add all the properties and methods of the object to this new object, effectively making an exact copy of itself"

Factory Functions

- Our copy() function can now be used to create a factory function for superheros.
- A factory function is a function that can be used to return an object

Using the Mixin Function to Add Modular Functionality

"The mixin() function lets us encapsulate properties and methods in an object, then add them to other objects without the overhead of an inheritance chain being created"

"One way to think about the difference between prototypal inheritance and inheritance from mixin objects is to consider whether an object is something or whether it has something."

"For example, a tank is a vehicle, so it might inherit from a Vehicle prototype. The tank also has a gun, so this functionality could be added using a gun mixin object. This gives us extra flexibility, since other objects might also use a gun, but not be a vehicle, such as a soldier object, for example. The soldier object might inherit from a Human prototype and also have the gun mixin"

Returning this

- each of the mixins above has a return value of this (you'll see later)

Chaining Functions

"If a method returns this, its methods can be chained together to form a sequence of method calls that are called one after the other. For example, the superman object can call all three of the superpower methods at once"

"technique that is commonly used by a number of JavaScript libraries, most notably jQuery. It helps to make code more concise by keeping multiple method calls on the same line, and with some clever method naming it can make the calls read almost like a sentence; the Jest testing library that we used in Chapter 10 makes use of this"

"big drawback with this technique is that it can make code more difficult to debug. If an error is reported as occurring on a particular line, there is no way of knowing which method caused the error, since there are multiple method calls on that line.

It's worth keeping in mind that if a method lacks a meaningful return value, it might as well return this so that chaining is possible"

Binding this

"We saw earlier that the value of this points to the object calling a method. It allows us to create generalized methods that refer to properties specific to a particular object. Be aware of a certain problem when a function is nested inside another function, which can often happen when using methods in objects, especially ones that accept callback functions. The problem is that the value of this loses its scope, and points to the global object inside a nested function, as can be seen in this example:

```
superman.friends = [batman,wonderWoman,aquaman]

superman.findFriends = function(){
this.friends.forEach(function(friend) {
   console.log(`${friend.name} is friends with ${this.name}`);
}
);
}

superman.findFriends()
<<  Batman is friends with undefined
   Wonder Woman is friends with undefined
   Aquaman is friends with undefined
```

The findFriends() method fails to produce the expected output because this.name is actually referencing the name property of the global window object, which has the value of undefined.

There are a couple of solutions to this problem."

Use that = this

"common solution is to set the variable that to equal this before the nested function, and refer to that in the nested function instead of this"

Use bind(this)

"method for all functions and is used to set the value of this in the function. If this is provided as an argument to bind() while it's still in scope, any reference to this inside the nested function will be bound to the object calling the original method"

Use for-of Instead of forEach()

"ES6 introduced the for-of syntax for arrays and this does not require a nested function to be used, so this remains bound to the superman object"

Use Arrow Functions

- introduced in ES6
- advantage → they don't have their own this context, so this remains bound to the original object making the function call
- should be used when anonymous functions are required in callbacks

Borrowing Methods from Prototypes

- possible to borrow methods from objects w/out having to inherit all their properties & methods
- done by making a reference to the function that you want to borrow

Borrowing Array Methods

"One of the most common uses of borrowing methods was to borrow methods from arrays in ES5. There are many array-like objects in JavaScript, such as the arguments object that's available in functions, and the node lists that many of the DOM methods return. These act like arrays but are missing a lot of the methods arrays have — often it would be convenient if they had them"

Composition over Inheritance

"There are a number of benefits to object-oriented programming, but there are also some problems that come with inheritance"

"The 'Gorilla Banana' problem occurs when you need a method from an object, so you inherit from that object. The name comes from a quote by Joe Armstrong, the creator of the Erlang programming language:

You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

The problem he describes is that if an object requires a banana() method that belongs to the Gorilla class, you have to inherit the whole class to gain access to that method. But as well as the method you wanted, the object also inherits a lot of other properties and methods that are not needed, causing it to become unnecessarily bloated."

If you do decide to use classes, it's recommended to make them 'skinny' — meaning they don't have too many properties and methods. Another good practice when creating classes is to keep inheritance chains short. If you have long lines of inheritance, the objects at the end of these chains will usually end up being bloated with properties and methods they don't need. It also causes problems if any of the objects in the chain need to change, as these changes will also affect other objects in the chain. A good rule of thumb is to only inherit once, keeping the inheritance chain to just two objects makes unpicking any issues far easier."

"If you want to use a particular method from a class, but it has lots of properties and methods you don't need, then it would be preferable to just 'borrow' the method instead, as we saw in the last section. So, borrow the banana method from the Gorilla class instead of inheriting the whole Gorilla!

banana = Gorilla.prototype.banana;

An even better approach would be to move the banana() method into a separate object then add it as a mixin to the Gorilla class, and any other objects that required it"

Chapter Summary

- Object-oriented programming (OOP) is a way of programming that uses objects that encapsulate their own properties and methods.
- The main concepts of OOP are encapsulation, polymorphism and inheritance.
- Constructor functions can be used to create instances of objects.
- ES6 introduced class declarations that use the class keyword. These can be used in place of constructor functions.
- Inside a constructor function or class declaration, the keyword this refers to the object returned by the function.
- All instances of a class or constructor function inherit all the properties and methods of its prototype.
- The prototype is live, so new properties and methods can be added to existing instances.
- The prototype chain is used to find an available method. If an object lacks a method, JavaScript will check whether its prototype has the method. If not, it will check that function's prototype until it finds the method or reaches the Object constructor function.
- Private properties and methods can be created by defining variables using const and defining a function inside a constructor function. These can be made public using getter and setter functions.
- Monkey-patching is the process of adding methods to built-in objects by augmenting their prototypes. This should"be done with caution as it can cause unexpected behavior in the way built-in objects work.
- A mixin method can be used to add properties and methods from other objects without creating an inheritance chain.
- Methods can be chained together and called in sequence if they return a reference to this.

- Polymorphism allows objects to override shared methods with a more specific implementation.
- The value of this is not retained inside nested functions, which can cause errors. This can be worked around by using that = this, using the bind(this) method and using arrow functions.
- Methods can be borrowed from other objects.
- Composition over inheritance is a design pattern where objects are composed from 'building-block' objects, rather than inheriting all their properties and methods from a parent class.

Questions:

1. Why is it called monkey-patching?

Fun Thing I Noted:

There are 2 monkey references made in the chapter:
1. Gorilla
2. Monkey