Object Oriented Programming

OOP, or Object Oriented Programming, is one of the major approaches to the software development process. In OOP, objects and classes organize code to describe things and what they can do.

In this course, you'll learn the basic principles of OOP in JavaScript, including the `this` keyword, prototype chains, constructors, and inheritance.

Create a Basic JavaScript Object

Think about things people see every day, like cars, shops, and birds. These are all *objects*: tangible things people can observe and interact with.

What are some qualities of these objects? A car has wheels. Shops sell items. Birds have wings.

These qualities, or *properties*, define what makes up an object. Note that similar objects share the same properties, but may have different values for those properties. For example, all cars have wheels, but not all cars have the same number of wheels.

Objects in JavaScript are used to model real-world objects, giving them properties and behavior just like their real-world counterparts. Here's an example using these concepts to create a `duck` object:

```
let duck = {
  name: "Aflac",
  numLegs: 2
};
```

This `duck` object has two property/value pairs: a `name` of `Aflac` and a `numLegs` of 2.

Create a `dog` object with `name` and `numLegs` properties, and set them to a string and a number, respectively.

Solution:

```
let dog = {
  name: "brownie",
  numLegs: 4
};
```

Use Dot Notation to Access the Properties of an Object

The last challenge created an object with various properties. Now you'll see how to access the values of those properties. Here's an example:

```
let duck = {
  name: "Aflac",
  numLegs: 2
};
console.log(duck.name);
```

Dot notation is used on the object name, `duck`, followed by the name of the property, `name`, to access the value of `Aflac`.

Print both properties of the `dog` object to your console.

Solution:

```
let dog = {
  name: "Spot",
  numLegs: 4
};
// Only change code below this line
```

```
console.log(dog.name);
console.log(dog.numLegs);
```

Create a Method on an Object

Objects can have a special type of property, called a *method*.

Methods are properties that are functions. This adds different behavior to an object. Here is the `duck` example with a method:

```
let duck = {
  name: "Aflac",
  numLegs: 2,
  sayName: function() {return "The name of this duck is " + duck.name + ".";}
};
duck.sayName();
```

The example adds the `sayName` method, which is a function that returns a sentence giving the name of the `duck`. Notice that the method accessed the `name` property in the return statement using `duck.name`. The next challenge will cover another way to do this.

Using the `dog` object, give it a method called `sayLegs`. The method should return the sentence `This dog has 4 legs.`

Solution:

```
let dog = {
  name: "Spot",
  numLegs: 4,
  sayLegs: function() {return "This dog has 4 legs.";}
```

```
};

dog.sayLegs();
```

Make Code More Reusable with the this Keyword

The last challenge introduced a method to the `duck` object. It used `duck.name` dot notation to access the value for the `name` property within the return statement:

```
sayName: function() {return "The name of this duck is " +
duck.name + ".";}
```

While this is a valid way to access the object's property, there is a pitfall here. If the variable name changes, any code referencing the original name would need to be updated as well. In a short object definition, it isn't a problem, but if an object has many references to its properties there is a greater chance for error.

A way to avoid these issues is with the `this` keyword:

```
let duck = {
  name: "Aflac",
  numLegs: 2,
  sayName: function() {return "The name of this duck is " +
this.name + ".";}
};
```

`this` is a deep topic, and the above example is only one way to use it. In the current context, `this` refers to the object that the method is associated with: `duck`. If the object's name is changed to `mallard`, it is not necessary to find all the references to `duck` in the code. It makes the code reusable and easier to read.

Modify the `dog.sayLegs` method to remove any references to `dog`. Use the `duck` example for guidance.

Solution:

```
let dog = {
  name: "Spot",
  numLegs: 4,
  sayLegs: function() {return "This dog has " + this.numLegs + " legs.";}
};

dog.sayLegs();
```

Define a Constructor Function

Constructors are functions that create new objects. They define properties and behaviors that will belong to the new object. Think of them as a blueprint for the creation of new objects.

Here is an example of a constructor:

```
function Bird() {
  this.name = "Albert";
  this.color = "blue";
  this.numLegs = 2;
}
```

This constructor defines a `Bird` object with properties `name`, `color`, and `numLegs` set to Albert, blue, and 2, respectively. Constructors follow a few conventions:

- Constructors are defined with a capitalized name to distinguish them from other functions that are not `constructors`.
- Constructors use the keyword `this` to set properties of the object they will create. Inside the constructor, `this` refers to the new object it will create.

- Constructors define properties and behaviors instead of returning a value as other functions might.

Create a constructor, `Dog`, with properties `name`, `color`, and `numLegs` that are set to a string, a string, and a number, respectively.

Solution:

```
function Dog() {
  this.name = "Brownie";
  this.color = "white";
  this.numLegs = 2;
}
```

Use the `Dog` constructor from the last lesson to create a new instance of `Dog`, assigning it to a variable `hound`.

Solution:

```
function Dog() {
  this.name = "Rupert";
  this.color = "brown";
  this.numLegs = 4;
}
// Only change code below this line
let hound = new Dog();
```

Extend Constructors to Receive Arguments

The `Bird` and `Dog` constructors from the last challenge worked well. However, notice that all `Birds` that are created with the `Bird` constructor are automatically named Albert, are blue in color, and have two legs. What if you want birds with different values for name and color? It's possible to change the properties of each bird manually but that would be a lot of work:

```
let swan = new Bird();
swan.name = "Carlos";
swan.color = "white";
```

Suppose you were writing a program to keep track of hundreds or even thousands of different birds in an aviary. It would take a lot of time to create all the birds, then change the properties to different values for every one. To more easily create different `Bird` objects, you can design your Bird constructor to accept parameters:

```
function Bird(name, color) {
  this.name = name;
  this.color = color;
  this.numLegs = 2;
}
```

Then pass in the values as arguments to define each unique bird into the `Bird` constructor: `let cardinal = new Bird("Bruce", "red");` This gives a new instance of `Bird` with `name` and `color` properties set to `Bruce` and `red`, respectively. The `numLegs` property is still set to 2. The `cardinal` has these properties:

```
cardinal.name
cardinal.color
cardinal.numLegs
```

The constructor is more flexible. It's now possible to define the properties for each `Bird` at the time it is created, which is one way that JavaScript constructors are so useful. They group objects together based on shared characteristics and behavior and define a blueprint that automates their creation.

Create another `Dog` constructor. This time, set it up to take the parameters `name` and `color`, and have the property `numLegs` fixed at 4. Then create a new `Dog` saved in a variable `terrier`. Pass it two strings as arguments for the `name` and `color` properties.

Solution:

```
function Dog(name, color) {
  this.name = name;
  this.color = color;
  this.numLegs = 4;
}

let terrier = new Dog("brownie", "blue");
```

Verify an Object's Constructor with instanceof

Anytime a constructor function creates a new object, that object is said to be an *instance* of its constructor. JavaScript gives a convenient way to verify this with the `instanceof` operator. `instanceof` allows you to compare an object to a constructor, returning `true` or `false` based on whether or not that object was created with the constructor. Here's an example:

```
let Bird = function(name, color) {
  this.name = name;
  this.color = color;
  this.numLegs = 2;
}

let crow = new Bird("Alexis", "black");

crow instanceof Bird;
```

This `instanceof` method would return `true`.

If an object is created without using a constructor, `instanceof` will verify that it is not an instance of that constructor:

```
let canary = {
  name: "Mildred",
  color: "Yellow",
  numLegs: 2
};

canary instanceof Bird;
```

This `instanceof` method would return `false`.

Create a new instance of the `House` constructor, calling it `myHouse` and passing a number of bedrooms. Then, use `instanceof` to verify that it is an instance of `House`.

Solution:

```
function House(numBedrooms) {
  this.numBedrooms = numBedrooms;
}

let myHouse = new House(5);
myHouse instanceof House;
```

Understand Own Properties

In the following example, the `Bird` constructor defines two properties: `name` and `numLegs`:

```
function Bird(name) {
```

```
  this.name = name;
  this.numLegs = 2;
}

let duck = new Bird("Donald");
let canary = new Bird("Tweety");
```

`name` and `numLegs` are called *own properties*, because they are defined directly on the instance object. That means that `duck` and `canary` each has its own separate copy of these properties. In fact every instance of `Bird` will have its own copy of these properties. The following code adds all of the own properties of `duck` to the array `ownProps`:

```
let ownProps = [];

for (let property in duck) {
  if(duck.hasOwnProperty(property)) {
    ownProps.push(property);
  }
}

console.log(ownProps);
```

The console would display the value `["name", "numLegs"]`.

Add the own properties of `canary` to the array `ownProps`.

Solution:

```
function Bird(name) {
  this.name = name;
  this.numLegs = 2;
}
```

```
let canary = new Bird("Tweety");
let ownProps = [];

for (let property in canary) {
  if(canary.hasOwnProperty(property)) {
    ownProps.push(property);
  }
}
console.log(ownProps);

// Only change code below this line
```

Use Prototype Properties to Reduce Duplicate Code

Since `numLegs` will probably have the same value for all instances of `Bird`, you essentially have a duplicated variable `numLegs` inside each `Bird` instance.

This may not be an issue when there are only two instances, but imagine if there are millions of instances. That would be a lot of duplicated variables.

A better way is to use the `prototype` of `Bird`. Properties in the `prototype` are shared among ALL instances of `Bird`. Here's how to add `numLegs` to the `Bird prototype`:

```
Bird.prototype.numLegs = 2;
```

Now all instances of `Bird` have the `numLegs` property.

```
console.log(duck.numLegs);
console.log(canary.numLegs);
```

Since all instances automatically have the properties on the `prototype`, think of a `prototype` as a "recipe" for creating objects. Note that the `prototype` for `duck` and `canary` is part of the `Bird` constructor as `Bird.prototype`. Nearly every object in JavaScript has a `prototype` property which is part of the constructor function that created it.

Add a `numLegs` property to the `prototype` of `Dog`

Use Prototype Properties to Reduce Duplicate Code

Since `numLegs` will probably have the same value for all instances of `Bird`, you essentially have a duplicated variable `numLegs` inside each `Bird` instance.

This may not be an issue when there are only two instances, but imagine if there are millions of instances. That would be a lot of duplicated variables.

A better way is to use the `prototype` of `Bird`. Properties in the `prototype` are shared among ALL instances of `Bird`. Here's how to add `numLegs` to the `Bird` `prototype`:

```
Bird.prototype.numLegs = 2;
```

Now all instances of `Bird` have the `numLegs` property.

```
console.log(duck.numLegs);
console.log(canary.numLegs);
```

Since all instances automatically have the properties on the `prototype`, think of a `prototype` as a "recipe" for creating objects. Note that the `prototype` for `duck` and `canary` is part of the `Bird` constructor as `Bird.prototype`. Nearly every object in JavaScript has a `prototype` property which is part of the constructor function that created it.

Add a `numLegs` property to the `prototype` of `Dog`

Solution:

```
function Dog(name) {
  this.name = name;
}
Dog.prototype.numLegs = 4;
```

```
// Only change code above this line
let beagle = new Dog("Snoopy");
```

Add all of the own properties of `beagle` to the array `ownProps`. Add all of the `prototype` properties of `Dog` to the array `prototypeProps`.

Add all of the own properties of `beagle` to the array `ownProps`. Add all of the `prototype` properties of `Dog` to the array `prototypeProps`.

Solution:

```
function Dog(name) {
  this.name = name;
}
Dog.prototype.numLegs = 4;

let beagle = new Dog("Snoopy");

let ownProps = [];
let prototypeProps = [];

for (let property in beagle) {
  if (beagle.hasOwnProperty(property)) {
    ownProps.push(property);
  } else {
    prototypeProps.push(property);
  }
}
// Only change code below this line
```

```
function Dog(name) {
  this.name = name;
}
Dog.prototype.numLegs = 4;

let beagle = new Dog("Snoopy");

let ownProps = [];
let prototypeProps = [];

for (let property in beagle) {
  if (beagle.hasOwnProperty(property)) {
    ownProps.push(property);
  } else {
    prototypeProps.push(property);
  }
}
// Only change code below this line
```

Understand the Constructor Property

There is a special `constructor` property located on the object instances `duck` and `beagle` that were created in the previous challenges:

```
let duck = new Bird();
let beagle = new Dog();


console.log(duck.constructor === Bird);
console.log(beagle.constructor === Dog);
```

Both of these `console.log` calls would display `true` in the console.

Note that the `constructor` property is a reference to the constructor function that created the instance. The advantage of the `constructor` property is that it's possible to check for this property to find out what kind of object it is. Here's an example of how this could be used:

```
function joinBirdFraternity(candidate) {
  if (candidate.constructor === Bird) {
    return true;
  } else {
    return false;
  }
}
```

Note: Since the `constructor` property can be overwritten (which will be covered in the next two challenges) it's generally better to use the `instanceof` method to check the type of an object.

Write a `joinDogFraternity` function that takes a `candidate` parameter and, using the `constructor` property, return `true` if the candidate is a `Dog`, otherwise return `false`.

Understand the Constructor Property

There is a special `constructor` property located on the object instances `duck` and `beagle` that were created in the previous challenges:

```
let duck = new Bird();
let beagle = new Dog();


console.log(duck.constructor === Bird);
console.log(beagle.constructor === Dog);
```

Both of these `console.log` calls would display `true` in the console.

Note that the `constructor` property is a reference to the constructor function that created the instance. The advantage of the `constructor` property is that it's possible to check for this property to find out what kind of object it is. Here's an example of how this could be used:

```
function joinBirdFraternity(candidate) {
  if (candidate.constructor === Bird) {
    return true;
  } else {
    return false;
  }
}
```

Note: Since the `constructor` property can be overwritten (which will be covered in the next two challenges) it's generally better to use the `instanceof` method to check the type of an object.

Write a `joinDogFraternity` function that takes a `candidate` parameter and, using the `constructor` property, return `true` if the candidate is a `Dog`, otherwise return `false`.

Solution:

```
function Dog(name) {
  this.name = name;
}

// Only change code below this line
function joinDogFraternity(candidate) {
  if (candidate.constructor === Dog) {
    return true;
  } else {
    return false;
  }
}
```

Change the Prototype to a New Object

Up until now you have been adding properties to the `prototype` individually:

```
Bird.prototype.numLegs = 2;
```

This becomes tedious after more than a few properties.

```
Bird.prototype.eat = function() {
  console.log("nom nom nom");
}

Bird.prototype.describe = function() {
  console.log("My name is " + this.name);
}
```

A more efficient way is to set the `prototype` to a new object that already contains the properties. This way, the properties are added all at once:

```
Bird.prototype = {
  numLegs: 2,
  eat: function() {
    console.log("nom nom nom");
  },
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Add the property `numLegs` and the two methods `eat()` and `describe()` to the `prototype` of `Dog` by setting the `prototype` to a new object.

Solution:

```
function Dog(name) {
  this.name = name;
}

Dog.prototype = {
  // Only change code below this line
  numLegs: 4,
  eat() {
    console.log("nom nom nom");;
  },
  describe() {
    console.log("My name is" + thisname);
  }
};
```

Remember to Set the Constructor Property when Changing the Prototype

There is one crucial side effect of manually setting the prototype to a new object. It erases the `constructor` property! This property can be used to check which constructor function created the instance, but since the property has been overwritten, it now gives false results:

```
duck.constructor === Bird;
duck.constructor === Object;
duck instanceof Bird;
```

In order, these expressions would evaluate to `false`, `true`, and `true`.

To fix this, whenever a prototype is manually set to a new object, remember to define the `constructor` property:

```
Bird.prototype = {
  constructor: Bird,
  numLegs: 2,
  eat: function() {
    console.log("nom nom nom");
  },
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Define the `constructor` property on the `Dog` `prototype`.

Solution:

```
function Dog(name) {
  this.name = name;
}

// Only change code below this line
Dog.prototype = {
  constructor: Dog,
  numLegs: 4,
  eat: function() {
    console.log("nom nom nom");
  },
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Understand Where an Object's Prototype Comes From

Just like people inherit genes from their parents, an object inherits its `prototype` directly from the constructor function that created it. For example, here the `Bird` constructor creates the `duck` object:

```
function Bird(name) {
  this.name = name;
}

let duck = new Bird("Donald");
```

`duck` inherits its `prototype` from the `Bird` constructor function. You can show this relationship with the `isPrototypeOf` method:

```
Bird.prototype.isPrototypeOf(duck);
```

This would return `true`.

Use `isPrototypeOf` to check the `prototype` of `beagle`.

Solution:

```
function Dog(name) {
  this.name = name;
}

let beagle = new Dog("Snoopy");

Dog.prototype.isPrototypeOf(beagle);
// Only change code below this line
function Dog(name) {
  this.name = name;
```

```
}

let beagle = new Dog("Snoopy");

Dog.prototype.isPrototypeOf(beagle);
// Only change code below this line
```

Modify the code to show the correct prototype chain.

```
Run the Tests (Ctrl + Enter)Reset All Code

Get Help
```

Solution:

```
function Dog(name) {
  this.name = name;
}

let beagle = new Dog("Snoopy");

Dog.prototype.isPrototypeOf(beagle);  // yields true

// Fix the code below so that it evaluates to true
Object.prototype.isPrototypeOf(Dog.prototype);
```

Use Inheritance So You Don't Repeat Yourself

There's a principle in programming called *Don't Repeat Yourself (DRY)*. The reason repeated code is a problem is because any change requires fixing code

in multiple places. This usually means more work for programmers and more room for errors.

Notice in the example below that the `describe` method is shared by `Bird` and `Dog`:

```
Bird.prototype = {
  constructor: Bird,
  describe: function() {
    console.log("My name is " + this.name);
  }
};

Dog.prototype = {
  constructor: Dog,
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

The `describe` method is repeated in two places. The code can be edited to follow the DRY principle by creating a `supertype` (or parent) called `Animal`:

```
function Animal() { };

Animal.prototype = {
  constructor: Animal,
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Since `Animal` includes the `describe` method, you can remove it from `Bird` and `Dog`:

```
Bird.prototype = {
  constructor: Bird
};

Dog.prototype = {
  constructor: Dog
};
```

The `eat` method is repeated in both `Cat` and `Bear`. Edit the code in the spirit of DRY by moving the `eat` method to the `Animal supertype`.

Solution:

```
function Cat(name) {
  this.name = name;
}

Cat.prototype = {
  constructor: Cat,
};

function Bear(name) {
  this.name = name;
}

Bear.prototype = {
  constructor: Bear,
};

function Animal() { }

Animal.prototype = {
  constructor: Animal,
    eat: function() {
```

```
    console.log("nom nom nom");
  }
};
```

Inherit Behaviors from a Supertype

In the previous challenge, you created a `supertype` called `Animal` that defined behaviors shared by all animals:

```
function Animal() { }
Animal.prototype.eat = function() {
  console.log("nom nom nom");
};
```

This and the next challenge will cover how to reuse the methods of `Animal` inside `Bird` and `Dog` without defining them again. It uses a technique called inheritance. This challenge covers the first step: make an instance of the `supertype` (or parent). You already know one way to create an instance of `Animal` using the `new` operator:

```
let animal = new Animal();
```

There are some disadvantages when using this syntax for inheritance, which are too complex for the scope of this challenge. Instead, here's an alternative approach without those disadvantages:

```
let animal = Object.create(Animal.prototype);
```

`Object.create(obj)` creates a new object, and sets `obj` as the new object's `prototype`. Recall that the `prototype` is like the "recipe" for creating an object. By setting the `prototype` of `animal` to be the `prototype` of `Animal`, you are effectively giving the `animal` instance the same "recipe" as any other instance of `Animal`.

```
animal.eat();
```

```
animal instanceof Animal;
```

The `instanceof` method here would return `true`.

Use `Object.create` to make two instances of `Animal` named `duck` and `beagle`.

Solution:

```
function Animal() { }

Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};

// Only change code below this line

let duck = Object.create(Animal.prototype); // Change thi
s line
let beagle = Object.create(Animal.prototype); // Change t
his line
```

Set the Child's Prototype to an Instance of the Parent

In the previous challenge you saw the first step for inheriting behavior from the supertype (or parent) `Animal`: making a new instance of `Animal`.

This challenge covers the next step: set the `prototype` of the subtype (or child)—in this case, `Bird`—to be an instance of `Animal`.

```
Bird.prototype = Object.create(Animal.prototype);
```

Remember that the `prototype` is like the "recipe" for creating an object. In a way, the recipe for `Bird` now includes all the key "ingredients" from `Animal`.

```
let duck = new Bird("Donald");
duck.eat();
```

`duck` inherits all of `Animal`'s properties, including the `eat` method.

Modify the code so that instances of `Dog` inherit from `Animal`.

Solution:

```
function Animal() { }

Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};

function Dog() { }

// Only change code below this line

Dog.prototype = Object.create(Animal.prototype);

let beagle = new Dog();
```

Reset an Inherited Constructor Property

When an object inherits its `prototype` from another object, it also inherits the supertype's constructor property.

Here's an example:

```
function Bird() { }
Bird.prototype = Object.create(Animal.prototype);
let duck = new Bird();
duck.constructor
```

But `duck` and all instances of `Bird` should show that they were constructed by `Bird` and not `Animal`. To do so, you can manually set the constructor property of `Bird` to the `Bird` object:

```
Bird.prototype.constructor = Bird;
duck.constructor
```

Fix the code so `duck.constructor` and `beagle.constructor` return their respective constructors.

Solution:

```
function Animal() { }
function Bird() { }
function Dog() { }

Bird.prototype = Object.create(Animal.prototype);
Dog.prototype = Object.create(Animal.prototype);

// Only change code below this line
Bird.prototype.constructor = Bird;

Dog.prototype.constructor = Dog;


let duck = new Bird();
let beagle = new Dog();
```

Add Methods After Inheritance

A constructor function that inherits its `prototype` object from a supertype constructor function can still have its own methods in addition to inherited methods.

For example, `Bird` is a constructor that inherits its `prototype` from `Animal`:

```
function Animal() { }
Animal.prototype.eat = function() {
  console.log("nom nom nom");
};
function Bird() { }
Bird.prototype = Object.create(Animal.prototype);
Bird.prototype.constructor = Bird;
```

In addition to what is inherited from `Animal`, you want to add behavior that is unique to `Bird` objects. Here, `Bird` will get a `fly()` function. Functions are added to `Bird's prototype` the same way as any constructor function:

```
Bird.prototype.fly = function() {
  console.log("I'm flying!");
};
```

Now instances of `Bird` will have both `eat()` and `fly()` methods:

```
let duck = new Bird();
duck.eat();
duck.fly();
```

`duck.eat()` would display the string `nom nom nom` in the console, and `duck.fly()` would display the string `I'm flying!`.

Add all necessary code so the `Dog` object inherits from `Animal` and the `Dog`'s `prototype` constructor is set to `Dog`. Then add a `bark()` method to the `Dog` object so that `beagle` can both `eat()` and `bark()`. The `bark()` method should print `Woof!` to the console.

Solution:

```
function Animal() { }
Animal.prototype.eat = function() { console.log("nom nom nom"); };

function Dog() { }
// Only change code below this line
Dog.prototype = Object.create(Animal.prototype)
Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {
  console.log("Woof!");
}



// Only change code above this line

let beagle = new Dog();
```

Override Inherited Methods

In previous lessons, you learned that an object can inherit its behavior (methods) from another object by referencing its `prototype` object:

```
ChildObject.prototype = Object.create(ParentObject.prototype);
```

Then the `ChildObject` received its own methods by chaining them onto its `prototype`:

```
ChildObject.prototype.methodName = function() {...};
```

It's possible to override an inherited method. It's done the same way - by adding a method to `ChildObject.prototype` using the same method name as the one to override. Here's an example of `Bird` overriding the `eat()` method inherited from `Animal`:

```
function Animal() { }
Animal.prototype.eat = function() {
  return "nom nom nom";
};
function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.eat = function() {
  return "peck peck peck";
};
```

If you have an instance `let duck = new Bird();` and you call `duck.eat()`, this is how JavaScript looks for the method on the `prototype` chain of `duck`:

1. `duck` => Is `eat()` defined here? No.
2. `Bird` => Is `eat()` defined here? => Yes. Execute it and stop searching.
3. `Animal` => `eat()` is also defined, but JavaScript stopped searching before reaching this level.
4. Object => JavaScript stopped searching before reaching this level.

Override the `fly()` method for `Penguin` so that it returns the string `Alas, this is a flightless bird.`

Override Inherited Methods

In previous lessons, you learned that an object can inherit its behavior (methods) from another object by referencing its `prototype` object:

```
ChildObject.prototype = Object.create(ParentObject.prototype);
```

Then the `ChildObject` received its own methods by chaining them onto its `prototype`:

```
ChildObject.prototype.methodName = function() {...};
```

It's possible to override an inherited method. It's done the same way - by adding a method to `ChildObject.prototype` using the same method name as the one to override. Here's an example of `Bird` overriding the `eat()` method inherited from `Animal`:

```
function Animal() { }
Animal.prototype.eat = function() {
  return "nom nom nom";
};
function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.eat = function() {
  return "peck peck peck";
};
```

If you have an instance `let duck = new Bird();` and you call `duck.eat()`, this is how JavaScript looks for the method on the `prototype` chain of `duck`:

1. `duck` => Is `eat()` defined here? No.
2. `Bird` => Is `eat()` defined here? => Yes. Execute it and stop searching.
3. `Animal` => `eat()` is also defined, but JavaScript stopped searching before reaching this level.
4. Object => JavaScript stopped searching before reaching this level.

Override the `fly()` method for `Penguin` so that it returns the string `Alas, this is a flightless bird.`

Solution:

```
function Bird() { }
```

```
Bird.prototype.fly = function() { return "I am flying!";
};

function Penguin() { }
Penguin.prototype = Object.create(Bird.prototype);
Penguin.prototype.constructor = Penguin;

// Only change code below this line
Penguin.prototype.fly = function() {
  return "Alas, this is a flightless bird.";
}


// Only change code above this line

let penguin = new Penguin();
console.log(penguin.fly());
```

Use a Mixin to Add Common Behavior Between Unrelated Objects

As you have seen, behavior is shared through inheritance. However, there are cases when inheritance is not the best solution. Inheritance does not work well for unrelated objects like `Bird` and `Airplane`. They can both fly, but a `Bird` is not a type of `Airplane` and vice versa.

For unrelated objects, it's better to use *mixins*. A mixin allows other objects to use a collection of functions.

```
let flyMixin = function(obj) {
  obj.fly = function() {
    console.log("Flying, wooosh!");
  }
```

```
};
```

The `flyMixin` takes any object and gives it the `fly` method.

```
let bird = {
  name: "Donald",
  numLegs: 2
};

let plane = {
  model: "777",
  numPassengers: 524
};

flyMixin(bird);
flyMixin(plane);
```

Here `bird` and `plane` are passed into `flyMixin`, which then assigns the `fly` function to each object. Now `bird` and `plane` can both fly:

```
bird.fly();
plane.fly();
```

The console would display the string `Flying, wooosh!` twice, once for each `.fly()` call.

Note how the mixin allows for the same `fly` method to be reused by unrelated objects `bird` and `plane`.

Create a mixin named `glideMixin` that defines a method named `glide`. Then use the `glideMixin` to give both `bird` and `boat` the ability to glide.

Solution:

```
let bird = {
```

```
  name: "Donald",
  numLegs: 2
};

let boat = {
  name: "Warrior",
  type: "race-boat"
};

let glideMixin = function(obj) {
  obj.glide = function() {

  }
}
glideMixin(bird);
glideMixin(boat);

// Only change code below this line
```

Use Closure to Protect Properties Within an Object from Being Modified Externally

In the previous challenge, `bird` had a public property `name`. It is considered public because it can be accessed and changed outside of `bird`'s definition.

```
bird.name = "Duffy";
```

Therefore, any part of your code can easily change the name of `bird` to any value. Think about things like passwords and bank accounts being easily changeable by any part of your codebase. That could cause a lot of issues.

The simplest way to make this public property private is by creating a variable within the constructor function. This changes the scope of that variable to be within the constructor function versus available globally. This way, the variable

can only be accessed and changed by methods also within the constructor function.

```
function Bird() {
  let hatchedEgg = 10;

  this.getHatchedEggCount = function() {
    return hatchedEgg;
  };
}
let ducky = new Bird();
ducky.getHatchedEggCount();
```

Here `getHatchedEggCount` is a privileged method, because it has access to the private variable `hatchedEgg`. This is possible because `hatchedEgg` is declared in the same context as `getHatchedEggCount`. In JavaScript, a function always has access to the context in which it was created. This is called `closure`.

Change how `weight` is declared in the `Bird` function so it is a private variable. Then, create a method `getWeight` that returns the value of `weight` 15.

Solution:

```
function Bird() {
  let weight = 15;
  this.getWeight = function() {
    return weight;
  }

}
```

Understand the Immediately Invoked Function Expression (IIFE)

A common pattern in JavaScript is to execute a function as soon as it is declared:

```
(function () {
  console.log("Chirp, chirp!");
})();
```

This is an anonymous function expression that executes right away, and outputs `Chirp, chirp!` immediately.

Note that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an *immediately invoked function expression* or *IIFE*.

Rewrite the function `makeNest` and remove its call so instead it's an anonymous immediately invoked function expression (IIFE).

solution:

```
(function () {
  console.log("A cozy nest is ready");
})();
```

Use an IIFE to Create a Module

An immediately invoked function expression (IIFE) is often used to group related functionality into a single object or *module*. For example, an earlier challenge defined two mixins:

```
function glideMixin(obj) {
  obj.glide = function() {
    console.log("Gliding on the water");
  };
}
function flyMixin(obj) {
  obj.fly = function() {
    console.log("Flying, wooosh!");
  };
}
```

We can group these mixins into a module as follows:

```
let motionModule = (function () {
  return {
    glideMixin: function(obj) {
      obj.glide = function() {
        console.log("Gliding on the water");
      };
    },
    flyMixin: function(obj) {
      obj.fly = function() {
        console.log("Flying, wooosh!");
      };
    }
  }
})();
```

Note that you have an immediately invoked function expression (IIFE) that returns an object `motionModule`. This returned object contains all of the mixin behaviors as properties of the object. The advantage of the module pattern is

that all of the motion behaviors can be packaged into a single object that can then be used by other parts of your code. Here is an example using it:

```
motionModule.glideMixin(duck);
duck.glide();
```

Create a module named `funModule` to wrap the two mixins `isCuteMixin` and `singMixin`. `funModule` should return an object.

Solution:

```
let funModule = (function () {
  return {
    isCuteMixin(obj) {
  obj.isCute = function() {
    return true;
  };
},

singMixin(obj) {
  obj.sing = function() {
    console.log("Singing to an awesome tune");
  };
}
  }
})();
```