Object-Oriented Design

How to use object-oriented concepts in JavaScript to make more re-usable code.

Video: Object Types

\*

Challenge: Double Rainbow

Create a Rainbow object type that can describe the many rainbows in the world.

Make a Rainbow object

This program describes a rainbow in an object literal and then draws it. To make it easy for you to make lots of rainbows with the same kinds of properties, create a Rainbow object that stores an x, y, and radius.

rainbow

bigRainbow

drawRainbow

Initialize a new Rainbow

Now that you've defined the object constructor, use it to make a rainbow and pass that rainbow to the draw function.

Now that youâ€™ve created a new Rainbow, pass that Rainbow into the call to drawRainbow().

You should be able to delete the code for the original rainbow, now that you have the new fancy way of making Rainbows. Remember to delete both the code that creates the variable bigRainbow, and the code that draws the bigRainbow. But don't delete drawRainbow()!

Double Rainbow!

Now it's easy to make new Rainbows! Now make another rainbow with different properties.

Now that youâ€™ve created a second rainbow, make sure to draw it by calling drawRainbow with your second rainbow as the argument.

Video: Object Methods

\*

Challenge: SmileyFace

Create a SmileyFace object with its own methods.

Create a SmileyFace object

We have a lovely function for drawing smiley faces, based on object literals that we pass to it. However, we'd like to use object oriented design instead, so that we can create instances and add methods. Start off by creating a SmileyFace constructor that stores centerX and centerY properties.

faceObject

drawSmiley

Initialize a new SmileyFace

Now that you've defined the object constructor, use it to make a face and pass that face to the draw function. Once you've done that, remove the original face drawing code.

Add a draw method

In this step, you'll turn the drawSmiley function into a draw method for the SmileyFace object prototype, so that you can call it on any SmileyFace instance.

You'll also have to modify the the code inside this draw method, so that it takes no arguments and gets all the properties using this instead of faceObj.

As best practice, you should define everything about your object type first - the constructor and all its methods - and initialize instances of the object type after. Can you move your draw method definition to be right after the constructor, with nothing in between?

Add a speak method

Now, add a speak method that will allow you to make the SmileyFace talk. The speak method should accept a parameter - a string - and display that next to the smiley using the text command. Then call it to make sure it works!

If you don't remember what a parameter is, it's a good time to rewatch the talk-through about them in the Functions section.

speak

More smileys! MORE!

Now, make more smileys using your SmileyFace object, at different locations with different messages.

Video: Object Inheritance

\*

Challenge: Flower Grower

Create a Flower object type that Sunflower and Tulip can inherit from.

Add more flowers

This program creates two flowers, tulip (an instance of the Tulip type) and sunflower (an instance of the Sunflower type). The drawScene function draws these flowers by calling their draw methods. The mouseClicked function makes these flowers change height, when you click the mouse, by calling their growBy methods.

Play around with the program to see how it works. When you are done playing around:

- add one more flower of the Tulip type

- add one more flower of the Sunflower type

- modify the drawScene() function, so that it also draws your new flowers.

Where you place the code to create a new object instance, like a new flower, is important. A new object instance should be created somewhere below its constructor, but above any code that uses one of its methods or properties.

OBJECT TYPES

Flower Object Type

Tulip Object Type

petals

create object instances

Make the new flowers grow

In the previous step you added two new flowers and drew them. Now you need to make your new flowers grow.

Recall that, the mouseClicked function makes flowers change height, when you click the mouse, by calling their growBy methods.

Modify the mouseClicked() function, so that it also changes the height of your new flowers.

Make a Flower object

The Tulip and the Sunflower object types share a lot in common - the constructors are the same, and the growBy methods are the same. They're also conceptually similar - both are types of flowers! Now, let's use inheritance to make both these object types extend from a Flower object type.

Make a constructor for the Flower object type.

Where you place the code for a new object type, like Flower, is important. It should be placed somewhere above all of the object types that want to inherit from it.

Define the growBy method

The Tulip and Sunflower object types have identical code in their growBy methods. If we wanted to create a new type of flower, we would want the new flower type to use the same code too.

Add a growBy method to the Flower prototype, so that all flowers will be able to inherit it. The code should be the same as the growBy methods on the Tulip and Sunflower prototypes.

When your code makes changes to an object type's prototype, like adding the method growBy to the Flower prototype, it is good practice to place the code just below the object type's constructor.

Inherit from Flower!

Now let's actually put your Flower object type to work. Change the Tulip constructor so that it calls the Flower constructor with its arguments, and base the Tulip prototype on the Flower prototype so that it inherits the growBy method.

To avoid errors and prevent overwriting the methods on the Tulip prototype:

- the Tulip constructor should be placed above all of the methods on the Tulip prototype

- the statement for Tulip to inherit from the Flower prototype, should be placed immediately after the Tulip constructor

You may need to press 'Restart' for the environment to recognize your changes to the Tulip object.

Inherit from Flower - AGAIN!

Now let's do the same thing for Sunflower, so that both of your object types are inheriting from the Flower object type.

Change the Sunflower constructor so that it calls the Flower constructor with its arguments, and base the Sunflower prototype on the Flower prototype so that it inherits the growBy method.

To avoid errors and prevent overwriting the methods on the Sunflower prototype:

- the Sunflower constructor should be placed above all of the methods on the Sunflower prototype

- the statement for Sunflower to inherit from the Flower prototype, should be placed immediately after the Sunflower constructor

You may need to press 'Restart' for the environment to recognize your changes to the Sunflower object.

Change the Flowers

The great thing about object-oriented design and inheritance is that we can now change things on Flower, and everything that inherits from it should change too.

Test out the inheritance by changing the growBy method, so that it causes flowers to grow by twice the inputted amount.

You may need to press 'Restart' for the environment to recognize your changes to the Flower object.

\*

Review: Object-Oriented Design

This is a review of what we covered in this tutorial on object-oriented design.

When we create programs, we often find that we want to create many different objects that all share similar properties - like many cats, that have slightly different fur color and size, or many buttons, with different labels and positions. We want to be able to say "this is generally what a cat is like" and then say "let's make this specific cat, and this other cat, and they'll be similar in some ways and different in a few ways as well." In that case, we want to use object-oriented design to define object types and create new instances of those objects.

To define an object type in JavaScript, we first have to define a "constructor function". This is the function that we'll use whenever we want to create a new instance of that object type. Here's a constructor function for a Book object type:

Book

title

author

numPages

currentPage

The function takes in arguments for the aspects that will be different about each book - the title, author, and number of pages. It then sets the initial properties of the object based on those arguments, using the this keyword. When we use this in an object, we are referring to the current instance of an object, referring to itself. We need to store the properties on this to make sure we can remember them later.

To create an instance of a Book object, we declare a new variable to store it, then use the new keyword, followed by the constructor function name, and pass in the arguments that the constructor expects:

Robot Dreams

Isaac Asimov

We can then access any properties that we stored in the object using dot notation:

I loved reading

I loved reading Robot Dreams

is my fav author

Isaac Asimov is my fav author

Let's contrast this for a minute, and show what would have happened if we didn't set up our constructor function properly:

Little Brother

Cory Doctorow

I loved reading undefined

undefined is my favorite author

If we pass the arguments into the constructor function but do not explicitly store them on this, then we will \*not\* be able to access them later! The object will have long forgotten about them.

When we define object types, we often want to associate both properties \*and behavior\* with them - like all of our cat objects should be able to meow() and eat(). So we need to be able to attach functions to our object type definitions, and we can do that by defining them on what's called the object prototype:

readItAll

You read

pages

It's like how we would define a function normally, except that we hang it off the Book's prototype instead of just defining it globally. That's how JavaScript knows that this is a function that can be called on any Book object, and that this function should have access to the this of the book that it's called on.

We can then call the function (which we call a method, since it's attached to an object), like so:

Animal Farm

George Orwell

You read 112 pages!

Remember, the whole point of object-oriented design is that it makes it easy for us to make multiple related objects (object instances). Let's see that in code:

Pirate Cinema

Cory Doctorow

The Giver

Lois Lowry

Tuck Everlasting

Natalie Babbit

You read 384 pages!

You read 179 pages!

You read 144 pages!

That code gives us three books that are similar - they all have the same types of properties and behavior, but also different. Sweet!

Now, if you think about the world, cats and dogs are different types of objects, so you'd probably create different object types for them if you were programming a Cat and a Dog. A cat would meow(), a dog would bark(). But they're also similar- both a cat and dog would eat(), they both have an age, and a birth, and a death. They're both mammals, and that means they share a lot in common, even if they're also different.

In that case, we want to use the idea of object inheritance. An object type could inherit properties and behavior from a parent object type, but then also have its own unique things about it. All the Cats and Dogs could inherit from Mammal, so that they wouldn't have to invent eat()ing from scratch. How would we do that in JavaScript?

Let's go back to our book example, and say that Book is the "parent" object type, and we want to make two object types that inherit from it - Paperback and EBook.

A paperback is like a book, but it has one main thing different, at least for our program: it has a cover image. So, our constructor needs to take four arguments, to take in that extra info:

PaperBack

cover

Now, we don't want to have to do all the work that we already did in the Book constructor to remember those first three arguments - we want to take advantage of the fact that the code for that would be the same. So we can actually call the Book constructor from the PaperBack constructor, and pass in those arguments:

We still need to store the cover property in the object though, so we need one more line to take care of that:

Now, we have a constructor for our PaperBack, which helps it share the same properties as Books, but we also want our PaperBack to inherit its methods. Here's how we do that, by telling the program that the PaperBack prototype should be based on the Book prototype:

We might also want to attach paperback-specific behavior, like being able to burn it, and we can do that by defining functions on the prototype, after that line above:

Omg, you burnt all \_\_ pages

And now we can create a new paperback, read it all, and burn it!

calvin

The Essential Calvin & Hobbes

Bill Watterson

You read 256 pages!

Omg, you burnt all 256 pages!

(Well, we're not really going to burn it, because that's an amazing book, but perhaps if we were stuck in a glacial desert and desperate for warmth and about to die.)

And now you can see how we can use object-oriented design principles in JavaScript to create more complex data for your programs and model your program worlds better.