LEARN BASIC OOP BY BUILDING A SHOPPING CART

Introduction:

OOP, or Object Oriented Programming, is one of the major approaches to the software development process. In OOP, developers use objects and classes to structure their code.

In this shopping cart project, you'll learn how to define classes and use them. You'll create class instances and implement methods for data manipulation.

This project will cover concepts like the ternary operator, the spread operator, the `this` keyword, and more.

Step 1:

You will be building a shopping cart application. The HTML and CSS are already provided, but you will need to build the JavaScript to make the page interactive.

To start, you will need to get some of your elements from the DOM. Start by using `document.getElementById()` to get the`#cart-container`, `#products-container`, and `#dessert-card-container` elements. Store them in variables named `cartContainer`, `productsContainer`, and `dessertCards`, respectively.

Since these will not change, remember to use `const` to declare them.

Step 2:

Now you need to get your two buttons. Continuing the pattern, get the `#cart-btn` and `#clear-cart-btn` elements. Store them in variables named `cartBtn` and `clearCartBtn`, respectively.

Step 3:

Next is to get your totals. Get the `#total-items`, `#subtotal`, `#taxes`, and `#total` elements. Store them in variables named `totalNumberOfItems`, `cartSubTotal`, `cartTaxes`, and `cartTotal`, respectively.

Step 4:

The last element to get is the `#show-hide-cart` element. Store it in a variable named `showHideCartSpan`.

Then, use `let` to declare a variable named `isCartShowing` and set it to `false`.

Step 5:

A shopping cart does not serve much purpose without products. Declare a `products` variable and set it to an empty array. Using an array will allow you to store multiple products.

Step 6:

You now need to start adding products. Before you do that, you need to consider the structure of your product data. A product will need a unique identifier to distinguish it from other products, a price so people know how much it costs, and a name so people know what they are buying. You should also add a category to each product.

Add an object to your `products` array. Give this object an `id` property set to the number `1`, a `name` property set to the string `"Vanilla Cupcakes (6 Pack)"`, a `price` property set to the number `12.99`, and a `category` property set to the string `"Cupcake"`.

Step 7:

Following that same data structure, add the products from this table (in order) to your `products` array. Increment the `id` for each product, counting up.

name	price	category
French Macaron	3.99	Macaron
Pumpkin Cupcake	3.99	Cupcake
Chocolate Cupcake	5.99	Cupcake
Chocolate Pretzels (4 Pack)	10.99	Pretzel
Strawberry Ice Cream	2.99	Ice Cream
Chocolate Macarons (4 Pack)	9.99	Macaron
Strawberry Pretzel	4.99	Pretzel
Butter Pecan Ice Cream	2.99	Ice Cream
Rocky Road Ice Cream	2.99	Ice Cream

```
Vanilla Macarons (5 Pack)     11.99     Macaron

Lemon Cupcakes (4 Pack)       12.99     Cupcake
```

Step 8:

Now that you have your list of products, you can use JavaScript to insert them into the HTML. With this approach, if you decide to add more products, the HTML will automatically reflect that.

Start by calling the `.forEach` method of your `products` array. Use arrow syntax to create an empty callback function.

Step 9:

Remember that you can use destructuring to extract multiple values from an array or object in a single statement.

For the first parameter of your callback function, destructure the `name`, `id`, `price`, and `category` properties from the object passed in.

Step 10:

You need to display the available products in your HTML. Start by using the addition assignment operator to add an empty template literal string to the `innerHTML` property of the `dessertCards` variable.

Step 11:

In your template literal, create a `div` element with a class of `dessert-card`. In that `div`, create an `h2` element and give it the text of the `name` variable.

Step 12:

After your `h2` element, create two `p` elements. Give the first a `class` of `dessert-price`, and set the text to a dollar sign `"$"` followed by the value of the `price` variable. Give the second a `class` of `product-category`, and the text `"Category: "` followed by the value of the `category` variable.

Step 13:

Finally, after your `p` elements, create a `button` element. Give it an `id` set to the value of the `id` variable, a `class` of `btn add-to-cart-btn`, and use `"Add to cart"` for the text.

Step 14:

You are already familiar with an HTML `class`, but JavaScript also has a class. In JavaScript, a class is like a blueprint for creating objects. It allows you to define a set of properties and methods, and instantiate (or create) new objects with those properties and methods.

The `class` keyword is used to declare a class. Here is an example of declaring a `Computer` class:

Example Code:

```
class Computer {};
```

Declare a `ShoppingCart` class.

Step 15:

Classes have a special `constructor` method, which is called when a new instance of the class is created. The `constructor` method is a great place to initialize properties of the class. Here is an example of a class with a `constructor` method:

Example Code:

```
class Computer {

  constructor() {

  }

}
```

Add an empty `constructor` method to the `ShoppingCart` class.

Step 16:

The `this` keyword in JavaScript is used to refer to the current object. Depending on where `this` is used, what it references changes. In the case of a class, it refers to the instance of the object being constructed. You can use the `this` keyword to set the properties of the object being instantiated. Here is an example:

Example Code:

```
class Computer {

  constructor() {

    this.ram = 16;

  }
```

```
}
```

In your constructor, use the `this` keyword to set the `items` property to an empty array. Also, set the `total` property to `0`, and the `taxRate` property to `8.25`.

Step 17:

Your `ShoppingCart` class needs the ability to add items. Create an empty `addItem` method, which takes two parameters: `id` and `products`. Creating a method might look like this:

Example Code:

```
class Computer {

  constructor() {

    this.ram = 16;

  }



  addRam(amount) {

    this.ram += amount;

  }

}
```

The first parameter, `id`, is the `id` of the product the user has added to their cart. The second parameter, `products`, is an array of product objects. By using a parameter instead of directly referencing your existing `products` array, this method will be more flexible if you wanted to add additional product lists in the future.

Step 18:

You need to find the product that the user is adding to the cart. Remember that arrays have a `.find()` method. In your `addItem` function, declare a `product` variable, and assign it the value of calling the `.find()` method on the `products` array.

For the callback to `.find()`, pass a function that takes a single parameter `item`, and returns whether the `id` property of `item` is strictly equal to the `id` parameter passed to `addItem`.

Step 19:

Use `const` and destructuring to extract `name` and `price` variables from `product`.

Step 20:

Now you need to push the `product` into the cart's `items` array. Remember to use the `this` keyword.

Step 21:

You now need a total count of each product that the user has in the cart. Declare a `totalCountPerProduct` variable, and assign it an empty object.

Step 22:

Use the `.forEach()` method to loop through the `items` array. Pass an empty callback function that takes a single parameter `dessert`.

Step 23:

In your `forEach` callback, you need to update the `totalCountPerProduct` object. Using the `id` of the current `dessert` as your property, update the value of the property to be the current value plus one. Do not use the addition assignment operator for this.

Step 24:

You now have a small bug. When you try to access a property of an object and the property doesn't exist, you get `undefined`. This means if the dessert isn't already present in the `totalCountPerProduct` object, you end up trying to add `1` to `undefined`, which results in `NaN`.

To fix this, you can use the `||` operator to set the value to `0` if it doesn't exist. Wrap your right-hand `totalCountPerProduct[dessert.id]` in parentheses, and add `|| 0` to the end of the expression.

Step 25:

Now you need to get prepared to update the display with the new product the user added. Declare a `currentProductCount` variable, and assign it the value of the `totalCountPerProduct` object's property matching the `id` of `product`.

Step 26:

You haven't written the code to generate the HTML yet, but if a product has already been added to the user's cart then there will be a matching element which you'll need.

Use `.getElementById()` to get the matching element - you'll be setting the `id` value to `product-count-for-id${product.id}`, so use a template literal to query that value.

Assign your query to a `currentProductCountSpan` variable.

Step 27:

The behaviour of the `addItem` method needs to change if the product is already in the cart or not. Create a ternary that checks if the current product is already in the cart. Use `undefined` for both the truthy and falsy expressions to avoid a syntax error.

Step 28:

For your truthy expression, removing the `undefined`, you need to update the `textContent` of the `currentProductCountSpan` to be the `currentProductCount` followed by an `x`. Use a template literal to do so.

Step 29:

For your falsy expression, you'll need to add new HTML to your `productsContainer`. Start by removing the `undefined`, then use the addition assignment operator and template literal syntax to add a `div` with the `class` set to `product` and the `id` set to `dessert${id}` to the `innerHTML` property of the `productsContainer`.

Step 30:

Inside your `div`, add two `p` elements. Set the text of the second `p` element to be the value of the `price` variable.

Step 31:

In your first `p` element, add a `span` element. Give the `span` a class of `product-count` and an `id` of `product-count-for-id${id}`. Then, after the span, give your `p` element the text of the `name` variable.

Step 32:

There is still more functionality that your `ShoppingCart` class needs, but first you need to be able to test the code you have currently written. You'll need to instantiate a new `ShoppingCart` object and assign it to a variable. Here is an example of instantiating the `Computer` class from earlier examples:

Example Code:

```
const myComputer = new Computer();
```

Declare a `cart` variable, and assign it a new `ShoppingCart` object. Note the use of the `new` keyword when instantiating the object.

Step 33:

You need to get all of the `Add to cart` buttons that you added to the DOM earlier. Declare an `addToCartBtns` variable, and assign it the value of calling the `getElementsByClassName()` method on the `document` object, passing in the string `"add-to-cart-btn"`.

Step 34:

You need to iterate through the buttons in your `addToCartBtns` variable. However, `.getElementsByClassName()` returns a Collection, which does not have a `forEach` method.

Use the spread operator on the `addToCartBtns` variable to convert it into an array. Then, use the `forEach` method to iterate through the array. Do not pass a callback function yet.

Step 35:

Add your callback function to the `forEach` method. It should take a `btn` parameter. Then, in the callback, add an event listener to the `btn`. The event listener should listen for a `click` event, and should take another callback with an `event` parameter. The second callback should be empty.

Step 36:

In your event listener callback, call the `.addItem()` method of your `cart` object, and pass in the `event.target.id`. Remember that the `id` here will be a string, so you need to convert it to a number.

Pass your `products` array as the second argument.

Step 37:

Your cart currently isn't visible on the webpage. To make it visible, start by adding an event listener to the `cartBtn` variable, listening for the click event. The callback does not need any parameters.

Step 38:

Start by inverting the value of `isCartShowing`. Remember that you can use the logical not operator `!` to invert the value of a boolean. Assign the inverted value to `isCartShowing`.

Step 39:

Now assign the `textContent` of the `showHideCartSpan` variable the result of a ternary expression which checks if `isCartShowing` is true. If it is, set the `textContent` to `"Hide"`, otherwise set it to `"Show"`.

Step 40:

Finally, update the `display` property of the `style` object of the `cartContainer` variable to another ternary which checks if `isCartShowing` is true. If it is, set the `display` property to `"block"`, otherwise set it to `"none"`.

Now you should be able to see your cart and add items to it.

Step 41:

You need a way to access the total number of items in the cart. The best way to do this is to add another method to your `ShoppingCart` class, rather than accessing the `items` array directly.

Add a `getCounts` method to the `ShoppingCart` class. It should return the number of items in the `items` array.

Step 42:

Now you can update the total number of items on the webpage. Assign the value of your `cart` object's `.getCounts()` method to the `textContent` of the `totalNumberOfItems` variable.

Step 43:

You also need to update the total price of the cart when the user adds an item. Create a `calculateTotal` method in the `ShoppingCart` class.

In that method, declare a `subTotal` variable and use the `reduce` method on the `items` array to calculate the sum of the `price` property of each item in the array. Use `total` and `item` as the parameters for your callback.

Remember to set your initial value in the `reduce` method.

Step 44:

Part of the total cost will include the tax, so you need to calculate that as well. Create a `calculateTaxes` method in the `ShoppingCart` class. This method should take an `amount` parameter.

Step 45:

Your `calculateTaxes` method should return the value of the `taxRate` (divided by 100, to convert it to a percent) multiplied by the `amount` parameter.

For clarity, wrap the `taxRate / 100` calculation in parentheses.

Step 46:

Because of the way computers store and work with numbers, calculations involving decimal numbers can result in some strange behavior. For example, `0.1 + 0.2` is not equal to `0.3`. This is because computers store decimal numbers as binary fractions, and some binary fractions cannot be represented exactly as decimal fractions.

We want to clean up the number result from your calculation. Wrap your calculation in parentheses (don't include the `return` statement!) and call the `.toFixed()` method on it. Pass the `.toFixed()` method the number `2` as an argument. This will round the number to two decimal places and return a string.

Step 47:

The issue with `.toFixed()` returning a string is that you want to be able to perform calculations with the tax rate. To fix this, you can pass the `.toFixed()` call (including the calculation) to the `parseFloat()` function. This will convert the fixed string back into a number, preserving the existing decimal places.

Pass your `.toFixed()` call to `parseFloat()`.

Step 48:

Declare a variable `tax` and assign it the value of calling your new `.calculateTaxes()` method, passing `subTotal` as the argument.

Step 49:

Update the `total` value to be the sum of the `subTotal` and `tax` values.

Step 50:

You're going to update the HTML in this method as well. Set the `textContent` of the `cartSubTotal` to be the value of `subTotal` to a fixed 2 decimal places. Use template literal syntax to add the dollar sign to the beginning of the value.

Step 51:

Following the same pattern as your `cartSubTotal`, update the `cartTaxes` to display the `tax` value, and your `cartTotal` to display the `total` property.

Step 52:

Finally, return the value of the `total` property. Remember your `this` keyword.

Step 53:

Now call your `.calculateTotal()` method inside your `forEach` loop.

Step 54:

Your last feature is to allow users to clear their cart. Add a `clearCart()` method to your `ShoppingCart` class.

Step 55:

The first thing you should do is check if the `items` array is empty. If it is, display an `alert` to the user with the text `Your shopping cart is already empty`, then return from the function.

Remember that `0` is a falsy value, so you can use the `!` operator to check if the array is empty.

After displaying the alert, return from the function to stop execution.

Step 56:

Browsers have a built-in `confirm()` function which displays a confirmation prompt to the user. `confirm()` accepts a string, which is the message displayed to the user. It returns `true` if the user confirms, and `false` if the user cancels.

Declare a variable `isCartCleared` and assign it the value of calling `confirm()` with the string `"Are you sure you want to clear all items from your shopping cart?"`.

Step 57:

You only want to clear the cart if the user confirms the prompt. Create an `if` statement that checks if the user confirmed the prompt.

In the `if` statement, set the `items` property back to an empty array, then set the `total` property to `0`.

Step 58:

You also need to start clearing the HTML. Set the `innerHTML` property of the `productsContainer` back to an empty string.

Step 59:

Set the `textContent` of the `totalNumberOfItems`, `cartSubTotal`, `cartTaxes`, and `cartTotal` elements all to the number `0`.

Step 60:

Your final step is to make your clear button functional. Add a `click` event listener to the `clearCartBtn`. For the callback, you can pass in `cart.clearCart` directly.

However, doing so will not work, because the context of `this` will be the `clearCartBtn` element. You need to bind the `clearCart` method to the `cart` object.

You can do this by passing `cart.clearCart.bind(cart)` as the callback.

And with that, your shopping cart project is complete!