Advanced JS: eComm Exercises 1-8

Common Tips

TIP 1: You will make your life easier if you have every class in these exercises in its own file.

TIP 2: Create a separate test.js file which imports your various classes, and update it after each exercise to creates a new instances of the classes, run the methods you've created and make sure they're working as expected.

TIP 3: Continue using the test.js class as your testing ground to create new instances of all the difference classes your about to make

TIP: You don't need any UI for this (yet) – in Step 8 you'll write a 'main' program which calls all the other classes and executes the code in such a way that allows a specific use case to occur.

Exercise 1: Basic Object with Getters and Setters

Objective: Create a class `Product` with properties `name`, `price`, and `description`, using getters and setters.

Tip: Start by defining the class with constructor arguments and implement getters and setters for each property.

Tip 2: Using test.js, create a new Product and Print out the products details, then update the Price and reprint the Product details after to make sure its all working correctly.

Exercise 2: Prototypal Inheritance

Objective: Create classes `TV` and `Shirt` that inherit from `Product`. Add specific properties like `screenSize` for `TV` and `size` for `Shirt`.

Tip: Use the `extends` keyword for inheritance and `super` to call the parent constructor.

Exercise 3: Using Constructor Functions

Objective: Create a new class `Book` using a constructor function, not ES6 classes, with properties and a prototype method to display its info.

Tip: Define properties inside the constructor function and methods on the prototype.

Exercise 4: Static Methods in Class

Objective: Add a static method 'Compare' to `Product` that compares two products based on price.

Tip: Define a static method inside the `Product` class that takes two `Product` instances as arguments.

Tip2: The Compare function should return a numberic value so that it can be used in a sorting routine later – see this site for example: https://freewebdesigntutorials.com/javaScriptTutorials/jsArrayObject/customSortFunction.php

Tip3: In your main.js file, create an Array of Products with different prices, then use the built in Array.sort() function, passing in the static method you just created as the custom sort function

Exercise 5: Implementing Validation Logic

Objective: In the `Product` class, add validation logic to ensure that the price is always a positive number.

Tip: Throw an error in the setter for `price` if a negative value is passed.

Tip: in your main.js, attempt to create a Product which breaks this validation and see what happens

Exercise 6: Private Members

Objective: Add a private field `stockCount` to `Product` and a method to update it safely.

Tip: Use the `#` syntax to define `stockCount` as a private field and implement a public setter method to modify it.

Exercise 7: Cart Class for Managing Products

Objective 7.1: Create a `Cart` class to manage a collection `Product` instances.

Tip: Store products in an array within the `Cart` class and use array methods for management.

Tip: Each item in the collection will be an object { product: Product, quantity: number }

Objective 7.2: Add a Sort method to the Cart, which sorts the Products by using the Compare method from Exercise 4

Exercise 8: Simulating Transactions and Handling Errors

Substep 8.1: Adding Products to Cart

Objective: Implement a method in the Cart class to add products.

Tip: Create an addItem method which takes a Product instance and quantity as arguments. Check if the product is already in the cart and update the quantity accordingly.

Tip: Each 'item' in the Cart should be an object with two properties: the Product & the Quantity

Adding a Product would like this:

items.push({ product, quantity });

Substep 8.2: Removing Products from Cart

Objective: Implement a method to remove a specific product from the cart.

Tip: Create a removeItem method that takes a product identifier and removes it from the cart's product array.

Substep 8.3: Updating Quantities in Cart

Objective: Allow updating the quantity of a specific item in the cart.

Tip: Add a method updateQuantity that takes a product identifier and a new quantity, then updates the item in the cart.

Substep 8.4: Add validation to the Cart

Objective: Add validation to the Cart so that:

1) Nothing except an object of type 'Product' can be added.
   Tip: using the built in 'instanceof' operator to check a parameter's type
2) The quantity of the Product being added must be greater than 0

Substep 8.5: Handle Sales

Objective: Implement functionality to apply sales to an instance of a product, BUT in a way where all **instances** of a particular **product type** (e.g., all TVs) will have the ***same*** sale price.

Tips:

Add a Boolean getter/setter isOnSale to the Product class.

In each child class of Product, add a static property 'salePrice' which is private

Add a method getSalePrice that returns a standard SalePrice for that type of product.

Create a CalculateTotal method in the Cart class. It should check if each Product is on sale (isOnSale is true). If so, use the SalePrice from the corresponding child class; otherwise, use the standard Price.

Substep 8.6: Handle Discounts

Objective: Implement functionality to apply discount codes to the Cart i.e. all Products in the Cart need to have the Discount Code applied.

Tips:

In the Product class, add a method applyDiscount which takes a discount percentage and calculates the discounted price.

In the Cart class, add a private collection of predefined DiscountCodes. Each code object should include a code string (e.g., "Hot32"), a Boolean flag indicating if the code is applied (e.g., isApplied), and a DiscountPercentage (e.g., 10%).

Add a method applyDiscountCode in the Cart class. This method should take a code and check against the predefined discount codes. If it matches, set the isApplied flag to true for that discount code.

Modify the CalculateTotal method in the Cart class. It should check which discount codes are applied and calculate the total price considering these discounts. If a discount code is applied, apply its discount percentage to every product in the cart.

Substep 8.7: Create a Main Program use case

Objective: Write a ‘main’ program which uses all the classes above to simulate the following scenario:

User adds 3 TVs, and 1 Tshirt to the Cart

Display the Cart total (print it contents and total to the console).

User clicks ‘Sort’ on the Cart

Display the Cart total (print it contents and total to the console).

The user then enters a valid discount code, which results in the whole order being 15% discounted.

Display the Cart total (print it contents and total to the console).

The user then adds 4 books to their Cart. The books are on sale.

NOTE: You will most likely get a error here because we can’t add anything to Cart unless it’s a Product. See **TIP#1** below to fix the error

Display the Cart total (print it contents and total to the console).

The user removes 2 TVs from their Cart.

Display the Cart total (print it contents and total to the console).

User clicks ‘Sort’ on the Cart again to see if that changes the order of the products…

Display the Cart total (print it contents and total to the console).

Make sure the total prints out what you expect each time.

TIP#1: The error you encountered is because we're trying to use ES6 class-based inheritance (Product) with the older function-based approach (Book). To resolve this, we should also convert Book into an ES6 class, which aligns with the ES6 class structure of Product i.e. use ‘extends’ in book.js like we’ve done with tv.js and shirt.js – including giving the book a sale price and a static getSalePrice method!
Also – be careful about the order of parameters in your constructor functions.

Solution to 8.7

(including this here as a guide)

```
const Product = require('./Product');
const TV = require('./TV');
const Shirt = require('./Shirt');
const Book = require('./Book');
const Cart = require('./Cart');

// Initialize Cart and Products
const cart = new Cart();
const tv = new TV("Super HD TV", 1500, "75-inch 4K TV", 75);
const tshirt = new Shirt("Casual Shirt", 29.99, "Cotton shirt", 'L');
const book = new Book("1984", 17, "A great book", "George Orwell");

// User adds 3 TVs and 1 T-shirt to the Cart
cart.addItem(tv, 3);
cart.addItem(tshirt, 1);
console.log("Cart after adding 3 TVs and 1 T-shirt:");
cart.displayCart();

// User clicks ‘Sort’ on the Cart
cart.sort();
console.log("\nCart after sorting:");
cart.displayCart();

// User enters a valid discount code
try {
    cart.applyDiscountCode("Hot32"); // Assuming this code gives a 15% dis-
count
} catch (error) {
    console.error(error.message);
}
console.log("\nCart after applying discount code:");
cart.displayCart();

// User adds 4 books to their Cart (books are on sale)
book.isOnSale = true;
cart.addItem(book, 4);
console.log("\nCart after adding 4 books on sale:");
cart.displayCart();

// User removes 2 TVs from their Cart
cart.updateQuantity(tv, 1); // Updating the quantity of TVs to 1
console.log("\nCart after removing 2 TVs:");
cart.displayCart();

// User clicks ‘Sort’ on the Cart again
cart.sort();
console.log("\nCart after sorting again:");
cart.displayCart();
```