PROGRAMMING USER INTERFACES

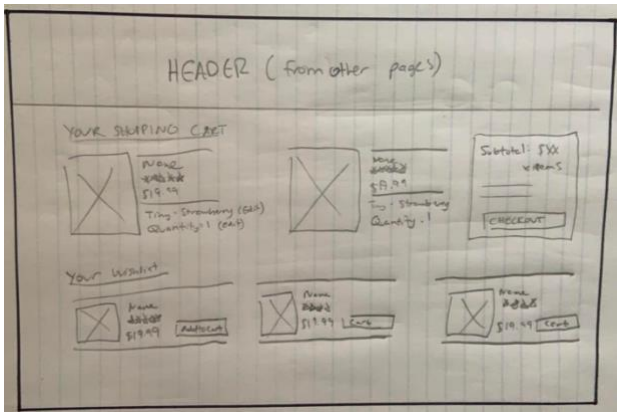
Assignment 6A – Creating a Shopping Cart Page and Adding Products to Cart

[Link to GitHub Repository]

[Link to Live GitHub Hosted Site]

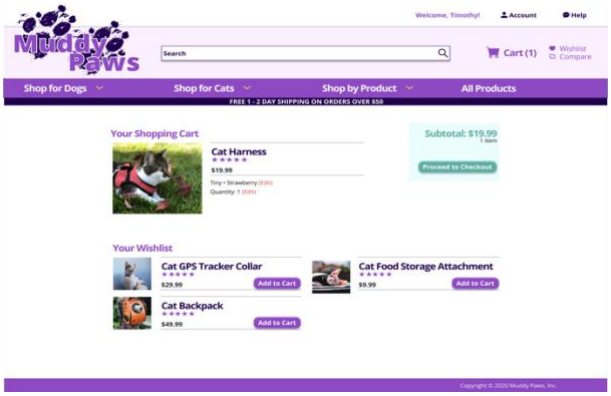
Low-Fidelity Prototype

In the low fidelity prototype, I tried to make the items that are in the cart salient by putting them at the top of the page and making them large.



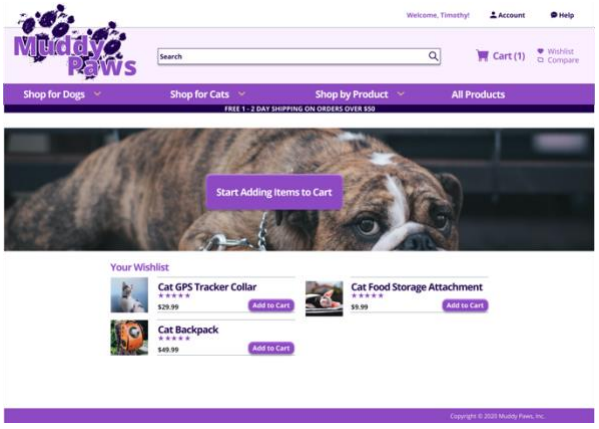
I also wanted to let the user know at a glance what their cart subtotal would be (this is a driving decision in deciding whether to go to checkout or remove items) and so encased the subtotal and number of items in a box on the right-hand side, which viewers can see after verifying the individual items in the cart. The proceed to checkout button is in this same box, as I imagine that users would move to checkout after they found the individual cart items and overall subtotal accurate.

At this point, I didn't want to remove the header and replace it with a navigation bar because I think people might navigate to this page just to see what is in their cart, even if they have no intention of moving to checkout. The wishlist below the cart summary should also provide users with some suggestions on items to add to cart, and this connection is why I included these items on this page.



High-Fidelity Prototype

I made some slight tweaks here. I stopped showing two cart items per page because it ended up being too busy for the mockup. Instead, new cart items would show up below the existing cart, creating a list of sorts and making it easier to see which items have been added recently.



I also added a new page for if nothing is in the cart because the page looks barren and unappealing otherwise. The simple sad dog image and inviting button should both let people know their cart is empty, and motivate them to add items to their cart.

PROGRAMMING USER INTERFACES

Assignment 6B – Creating a Shopping Cart Page and Adding Products to Cart

[Link to GitHub Repository]
[Link to Live GitHub Hosted Site]

Reflection

Bug #1: Issues with storage of information when opening new session
I used sessionStorage initially to get and set arrays of information across pages, and this led to issues and breakdowns when I would close my tab and then reopen it, as all the information would be wiped clean.

Fix #1: localStorage vs. sessionStorage
In a PUI Lab, our TA used localStorage instead of sessionStorage when getting and setting items, and it became clear to me that my issue was in using sessionStorage for my work (the arrays were only stored as long as the session was running, and stopped as soon as I closed the browser. I replaced all my instances of sessionStorage with localStorage and fixed the issue. I also learned not to incorporate JS code from online unless I really understood its purpose – the reason I had gotten into this issue in the first place was because I had been using sessionStorage without knowing its precise purpose and differentiation from other storage methods

Bug #2: Not knowing whether JavaScript functions were running or not
A huge issue I ran into was when I would modify my JavaScript code and I would have absolutely no idea if it was executing properly or not unless I ran through all the possible configurations of the code and saw things break. In many instances (e.g., when it came to adding a cart object instance to an array, the code would fail silently and I'd have no idea of the issue until much later in the process without knowing where in the code the error was.

Fix #2: Console.log will light the way
In searching online, I found that using console.log to print text to the console at various points of the code execution was the best solution, as it left a breadcrumb trail of sorts to the source of code effort. Developer tools were also incredibly helpful, as they would help pinpoint the exact part where code failed, although console.log was still essential because it would help me understand if my array was reflective of the information I needed even if the code didn't explicitly fail (by printing the array's contents at various points in the process). I learned just how different troubleshooting code in HTML / CSS is vs. JS, as you can usually trust what you see with errors in the former, but must really be vigilant for errors in the latter.

Bug #3: Locally storing a blank array
A final major error I had was in storing blank arrays of information to local storage, then parsing them (I did this to update a shopping cart total to 0 once I removed all items from the cart). However, when the array was empty and I used the

“localStorage.getItem()” function then tried to iterate through the array, I would get errors everywhere, despite the fact that you should be able to iterate through an empty array.

Fix #3: Operators for understandability
The issue was that JSON had issues with parsing empty arrays and wouldn’t return them as an empty array but would instead throw an error. To fix this, I was able to create a simple check:

```
let getNum = localStorage.getItem("cartCollection")
  cartCollection = getNum ? JSON.parse(getNum) : [];
```

This checked to see if the array was recognized or not (i.e., if it had one or more instance, and then returned the array. If not, it would return an empty array instead of the error that was being returned now. I learned from this how important it is to fully understand the behavior of code I use, as typically the code I had worked with until then had no problem with empty arrays, but that didn’t mean JSON would behave the same and it was up to me to identify the differences.

Programming Concepts

Concept #1: Cloning Feature

Overview

In order to create cart item cards dynamically based off the number of items in the cart, I had to be able copy a specific card template to reflect the total number of cart items. I learned to do this using the “cloneNode” property, where I was able to iterate through the number of cart objects in my array of cart objects and clone the card template for each object. I also was able to dynamically update the id of these newly cloned cards so that each one had a distinct id reflective of its order in the cart array, making removal of
these cards later much easier.

Application

```
function duplicateCard(arr) {
  let cartItems = document.getElementById("cart-
items");
  let cartItemCard =
document.getElementsByClassName("cart-item-
card")[0];
  for (i=0; i < arr.length-1; i++) {
    let clone = cartItemCard.cloneNode(true);
    clone.id = `order${i+1}`;
    cartItems.appendChild(clone);
  }
}
```

Concept #2: Navigating Other Elements Relative to a Selected Element

Overview

Another element I struggled with initially was understanding how to change elements that could only be identified by their relative positioning to an element that I defined in JavaScript (e.g., parent, children, sibling elements). Initially, I tried to give all these elements unique ids, but realized that was going against the philosophy of id creation. It also would not have been helpful for my specific need – I was trying to use the position of a "remove item from cart" button as a starting point to remove the cart item card that was the parent of that button.

Instead, I learned about parentNode and childNode. Using these properties allowed me to identify the element that a "remove item from cart" button was in, then iterate through to the parent cart card and remove the entire card instead of just the button.

Application

```
function removeCard(x) {
  let newId = 0;
  let card =
x.parentNode.parentNode.parentNode.parentNode
  let id = card.id;
  id = parseInt(id.substring(id.length-1));
  newId = id + 1;


  console.log(id);
  console.log(cartCollection);


card.parentNode.removeChild(x.parentNode.parentNod
e.parentNode.parentNode)
```

Concept #3: Constructing and Keeping Track of Class Instances

Overview

Whenever I added an item to cart, I needed to ensure that I had a systematic way of tracking all the various features of that product. I found that the class constructor feature was best-suited for this, as I could easily create an instance that tied together the salient features of the product like its name, price, color, size, and quantity.

However, in order to easily access the full list of instances I created for when I populated the cart page, I needed to take an additional step, as there was no way that object instances could be iterated

Application

```
class cartAddition {
  constructor(product, price, color, size, quantity) {
    this.product = product;
    this.price = price;
    this.color = color;
    this.size = size;
    this.quantity = quantity;
    cartCollection.push(this);
  }
}
let cartCollection = [];
```

through on their own. To accomplish this, I learned to add these items to a cart array (called cartCollection) by pushing the instance to the array automatically upon creation. This made the cart page far easier to populate, as I could easily iterate through products added to cart.

```
if (cartFull === true) {
    new cartAddition(productName, price, colorChoice,
sizeChoice, quantityChoice);
    cartCount = 0;
    for (i=0; i < cartCollection.length; i++) {
      cartCount = cartCount + cartCollection[i].quantity
    }
```

Concept #4: Revealing and Hiding Elements

Overview

One thing I struggled with initially was understanding how to use JavaScript to easily make elements appear and disappear, perfectly formatted, initially in the context of creating collapsible menus on the product details page. Initially, I tried to create the contents of the collapsible menu, format them in JavaScript, and then append them to the collapsible menu header. However, this became taxing and the code became unruly, so I looked into a way to pre-create these detailed views in HTML and CSS, then use JS to show and hide them.

For this, I found the content.style.display === “none” // “block” feature, which allowed me to toggle between making sections of code visible or invisible. It was far easier to use, loaded faster, and created the perfect collapsing / expanding effect I was looking for.

Application

```
function collapsibleInteraction(x) {
    let coll =
document.getElementsByClassName("collapsible");
        let arrow = document.getElementById('arrow'+x);
        let content = coll[x].nextElementSibling;
        if (content.style.display === "block") {
          content.style.display = "none";
          arrow.style.transform ='rotate(0deg)';
        } else {
          content.style.display = "block";
          arrow.style.transform ='rotate(180deg)';
        };
      };
```

Concept #5: Storing Information Across Pages

Overview

Once I added an item to my wishlist, I wanted the header at the top of the page to display the total number of items in the wishlist. This was easy enough to do on

Application

```
function addToWishlist(name, price){
  new wishListAddition(name, price);
  wishlistCount = 0;
```

the product details page initially, but I realized that once I navigated between pages, the wishlist count reverted back to 0.

In order to keep the page updated, I learned about JSON and local storage in the PUI Lab, which was a handy way to store the wishListCollection array temporarily through “getItem” then reassign the variable to an array using setItem on the new page (almost like conducting a handoff between pages). Once the handoff was conducted, I triggered a function that started on the load of the page to update the wishlist total tally in the header and keep information consistent between pages.

```
    let wishlistDisplay =
document.getElementsByClassName("shopping-
features")[0].children[0];
    wishlistDisplay.innerHTML = `<img class="icon"
src="Images/Heart-Filled.svg" alt="wishlist
icon">Wishlist (${wishlistCollection.length})`;

localStorage.setItem("wishlistCollection",JSON.stringify(
wishlistCollection))
}

function checkWishList() {
  let getWish =
localStorage.getItem("wishlistCollection")
  wishlistCollection = getWish ? JSON.parse(getWish) :
[];

  wishlistCount = 0;
  let wishlistDisplay =
document.getElementsByClassName("shopping-
features")[0].children[0];
  wishlistDisplay.innerHTML = `<img class="icon"
src="Images/Heart-Filled.svg" alt="wishlist
icon">Wishlist (${wishlistCollection.length})`;
}
```