3.3.6 Add the Shop Function

The shop feature will add some much-needed strategy to the game, as players must decide if they're willing to lower their score for additional perks.

Like the endgame logic, we'll put all the shop logic in a function. Again, we do this to keep the code organized and because we're likely to call `shop()` in more than one place.

Let's create a new function that, for now, simply logs a message. Put it after the `endGame()` function definition and before the `startGame()` function call at the end:

```
var shop = function() {
  console.log("entered the shop");
};
```

Before we get carried away writing the rest of the logic, let's make sure this function can be reached by defining the condition that calls it. Remember, the player should have the option to shop after they skip or defeat an enemy but only if there are still more enemies to fight. How will we know if there are more enemies? We'll know because there will still be items in the array.

In the `startGame()` function, add an `if` statement directly after we call the `fight()` function:

```
fight(pickedEnemyName);

// if we're not at the last enemy in the array
if (i < enemyNames.length - 1) {
  shop();
}
```

This will ensure that `shop()` is called after every fight but only if the loop iterator, `i`, still has room to increment.

REWIND

If an array has 10 items in it, the length of the array is 10. The indexes start at zero, however, so the last item in the array would be at index 9. Therefore, `nameOfArray.length - 1` would give us the last index, no matter how long the array is.

There's one more condition we should probably add to this `if` statement, however. A player can't shop if they've been defeated, so we'll need to check their health again.

Update the `if` statement to look like this:

```
// if player is still alive and we're not at the last enemy in the arr
if (playerHealth > 0 && i < enemyNames.length - 1) {
  shop();
}
```

This would be a good time to test the game in the browser. Verify that the console message `"entered the shop"` appears after a successful fight or skip.

HIDE PRO TIP

A good developer tests their code often! It's important to build and test in small steps. That way, if something goes wrong, there isn't too much code you have to backtrack through to discover the problem.

While testing, you may have noticed that the player is never asked if they want to shop; it goes directly into the `shop()` function. That could confuse the user. Let's fix that.

Add a `confirm()` before calling the `shop()` function:

```
if (playerHealth > 0 && i < enemyNames.length - 1) {
  // ask if user wants to use the store before next round
  var storeConfirm = window.confirm("The fight is over, visit the stor

  // if yes, take them to the store() function
  if (storeConfirm) {
    shop();
  }
}
```

Now that we've established how and when `shop()` gets called, we can focus our efforts solely on completing the shop functionality.

Replace the `console.log()` in `shop()` with a `prompt()`:

```
var shop = function() {
  // ask player what they'd like to do
  var shopOptionPrompt = window.prompt(
    "Would you like to REFILL your health, UPGRADE your attack, or LEA
  );
};
```

Whatever the user types in the prompt window will become the value of the variable `shopOptionPrompt`. There are four possibilities we need to account for: REFILL, UPGRADE, LEAVE, and anything else. If your instinct is to use a series of `if` statements, that's great thinking! In programming, however, there's always more than one way to solve a problem. We'll use this prompt as a chance to explore `switch` statements as an alternative to `if`.

A basic example of a `switch` statement looks like this:

```
var num = 5;

switch(num) {
  case 1:
    console.log("the variable was 1");
    break;
  case 2:
    console.log("the variable was 2");
    break;
  case 3:
    console.log("the variable was 3");
    break;
  default:
    console.log("the variable was something else");
    break;
}
```

Use `switch` statements when checking a single value against multiple possibilities, or **cases**. In this example, we're defining what should happen when the variable `num` equals 1, 2, 3, or something else (the `default` case). Each case ends with a `break` to specify that nothing more should happen. In the previous example, `"the variable was something else"` will print because `num` was 5.

We could have also written this using `if` statements:

```
if (num === 1) {
  console.log("the variable was 1");
}
else if (num === 2) {
  console.log("the variable was 2");
}
else if (num === 3) {
  console.log("the variable was 3");
}
else {
  console.log("the variable was something else");
}
```

Both examples accomplish the same thing, so it's ultimately a matter of preference. Using `switch` statements simply cuts down on how often you have to write `x === y`, `x === z`, etc. Because we only have one variable (`shopOptionPrompt`) that can be multiple values, a `switch` makes sense.

Add the following `switch` statement to the bottom of the `shop()` function:

```
// use switch to carry out action
switch (shopOptionPrompt) {
  case "refill":
    window.alert("Refilling player's health by 20 for 7 dollars.");

    // increase health and decrease money
    playerHealth = playerHealth + 20;
    playerMoney = playerMoney - 7;
    break;
  case "upgrade":
    window.alert("Upgrading player's attack by 6 for 7 dollars.");

    // increase attack and decrease money
    playerAttack = playerAttack + 6;
    playerMoney = playerMoney - 7;
    break;
  case "leave":
    window.alert("Leaving the store.");

    // do nothing, so function will end
    break;
  default:
    window.alert("You did not pick a valid option. Try again.");

    // call shop() again to force player to pick a valid option
    shop();
    break;
}
```

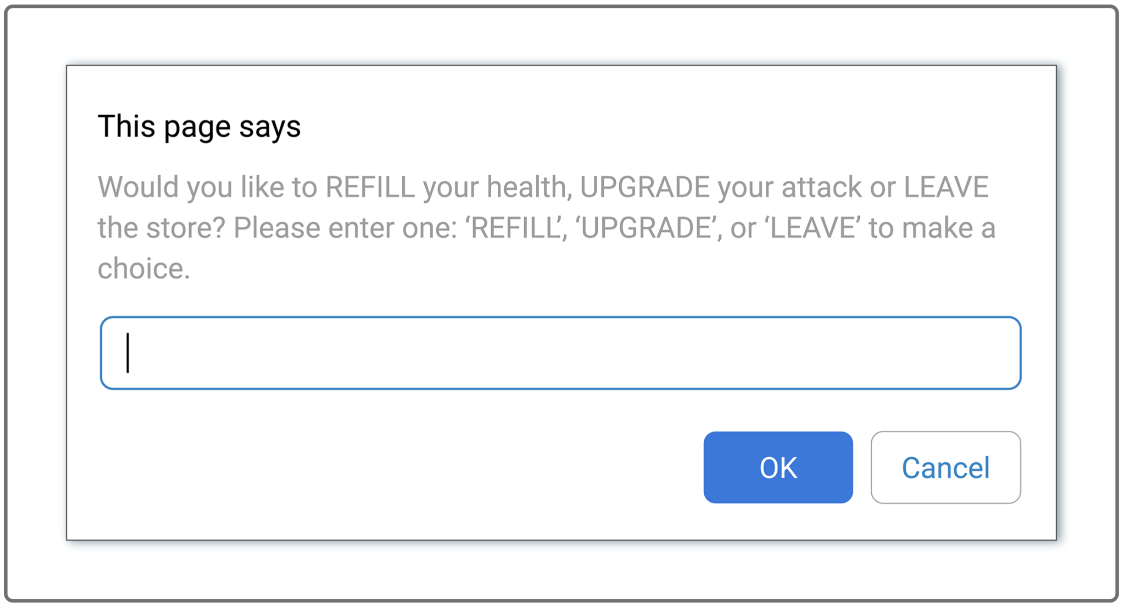
If the values `20`, `7`, and `6` seem too generous or meager, you can always adjust them. It can take many playthroughs of a game to pinpoint the right values that maximize the fun and challenge for you.

In any case, test the game to see if the shop options are working. One problem with the shop is that players can refill or upgrade even if they don't have enough money. We can put `if` statements inside the `switch` cases to catch that.

Update the "refill" and "upgrade" cases as follows:

```
case "refill":
  if (playerMoney >= 7) {
    window.alert("Refilling player's health by 20 for 7 dollars.");

    // increase health and decrease money
    playerHealth = playerHealth + 20;
    playerMoney = playerMoney - 7;
  }
  else {
    window.alert("You don't have enough money!");
  }

  break;
case "upgrade":
  if (playerMoney >= 7) {
    window.alert("Upgrading player's attack by 6 for 7 dollars.");

   // increase attack and decrease money
    playerAttack = playerAttack + 6;
    playerMoney = playerMoney - 7;
  }
  else {
    window.alert("You don't have enough money!");
  }

  break;
```

The last thing we should be mindful of is that the instructions in the prompt window capitalize the commands (e.g., REFILL, UPGRADE, LEAVE):


This page says

Would you like to REFILL your health, UPGRADE your attack or LEAVE the store? Please enter one: 'REFILL', 'UPGRADE', or 'LEAVE' to make a choice.

There's a good chance that players will try to capitalize their input as well, but what if they don't? Whether the user types "refill" or "REFILL," the same thing should happen. How do we account for that?

With `if` statements, we could use a `||` operator:

```
if (shopOptionPrompt === "refill" || shopOptionPrompt === "REFILL") {

}
```

The `switch` statements don't support `||` operators, but we can simply write additional cases for these repeated options.

Add a few extra cases to your `switch` so that the entire thing looks like this:

```
switch (shopOptionPrompt) {
  case "REFILL": // new case
  case "refill":
    if (playerMoney >= 7) {
      window.alert("Refilling player's health by 20 for 7 dollars.");

      playerHealth = playerHealth + 20;
      playerMoney = playerMoney - 7;
    }
    else {
      window.alert("You don't have enough money!");
    }

    break;
  case "UPGRADE": // new case
  case "upgrade":
    if (playerMoney >= 7) {
      window.alert("Upgrading player's attack by 6 for 7 dollars.");

      playerAttack = playerAttack + 6;
      playerMoney = playerMoney - 7;
    }
    else {
      window.alert("You don't have enough money!");
    }

    break;
  case "LEAVE": // new case
  case "leave":
    window.alert("Leaving the store.");
    break;
  default:
    window.alert("You did not pick a valid option. Try again.");
    shop();
    break;
}
```

Notice how `case "REFILL":` doesn't include any other code and is immediately followed by `case "refill":`. Because the first case doesn't `break`, it will **fall through** to the next one. This fall-through can continue until the code reaches another `break`.

DEEP DIVE

DEEP DIVE

Check out other examples of fall-through in the [MDN web docs on switch statements (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch#Methods_for_multi-criteria_case)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch#Methods_for_multi-criteria_case).

Play the game a few times in the browser to make sure the shop functionality is working correctly. If not, check the DevTools console for errors. For example, the error `Uncaught ReferenceError: refill is not defined` might mean we wrote `case refill:` instead of `case "refill":` with quotation marks.