3.1.8 "Stay or Go" with Control Flow

Most round-based battle games give the player options to choose from throughout the game. These choices change the direction and often the outcome of the game. Our user can fight, but let's give them the option to skip the fight.

HIDE HINT

All of the new code we write will go into the `fight` function.

Before we begin coding, let's consider how we want to tackle this problem by answering these questions:

- What can we use to ask the user if they'd like to fight or skip the fight?
- When would we ask them?
- How can we tell the code pertaining to fighting to not run if a user chooses to skip?

We can drive ourselves crazy thinking about all of the "what if" scenarios for our users, so it's important to think about the main functionality first. Even if the answers we come up with won't 100% do the job, it gets us thinking critically about what we need to write into our program.

Here are some potential answers to the above questions:

- We can use a prompt to ask the user if they would like to fight or skip the fight.
- We should ask users towards the beginning of the `fight` function so we know what they want to do before they do it.
- We can use an `if` statement to check how the user responded and run the code to make the robots fight only if the user answers a certain way.

Let's start by adding a `window.prompt` to the `fight` function. Right below the first alert in the `fight` function, create a variable called `promptFight` and assign it to store the returning value of a prompt that asks the user if they want to fight or skip the battle.

The code should look something like this:

```
var promptFight = window.prompt("Would you like to FIGHT or SKIP this
```

Now when the `fight` function executes, the user will get an alert welcoming them to the game and then asking if they'd like to fight or skip the battle. We're giving them explicit instructions on how we expect them to answer so that we can use the value stored in `promptFight` later in an `if` statement condition.

PAUSE

How can we read back the response of `promptFight` to make sure it's working?

We can use `console.log(promptFight);` to log it to the Chrome DevTools Console by writing it after we create the `promptFight` variable.

Hide Answer

Now we can get the user's input and use it to determine whether we fight or skip the battle. Think about this in terms of "if this, then do that." If a user chooses "FIGHT", what should happen? If the user chooses "SKIP" instead, what happens?

These are the two main answers we're looking for, but how can we stop a user from entering whatever they want? The answer is we can't. So how do we handle a situation like this?

NERD NOTE

User interaction can lead to unpredictable input and actions. A thoughtful developer builds a program that works under ideal conditions and then adds protective code to handle **edge cases**, where things don't go as expected.

So now we have to make a decision. We could set up our program to explicitly check to see if our user chose to "FIGHT" and let anything other than that mean the user has chosen to skip the fight instead. This means that if a user were to respond with "tomato", the program would interpret that as the user wanting to skip the fight. This could be very confusing for the user.

At the very least, it should be our job to inform them they made an incorrect decision and that our program isn't designed to accept an answer like that. So instead we'll do the following:

- Check if the user picked the word "FIGHT" or "fight".
    - If yes (or "true"), then we'll continue with the battle and our robots will fight.
- If the user did not pick "FIGHT" or "fight", check if the user picked "SKIP" or "skip" instead.
    - If yes, penalize the player and end the `fight` function.
- If the user did not pick any of the above words, then let the user know that they need to pick a valid option.

IMPORTANT

> Computers do not read letters as actual letters, but as binary data, which is a combination of ones and zeros (e.g., the letter "a" is 01100001 in binary). This means that uppercase and lowercase versions of a letter are represented by different sets of binary code.
>
> The lowercase letter "a" is represented as 01100001 in binary code, but the uppercase letter "A" is 01000001. If a computer tries to compare uppercase "A" to lowercase "a", it will not find them to be a match.

Previously, we used control flow statements to take us in one of two possible directions. If the player had no health, let them know they lost. Otherwise, let them know how they're doing.

Now, we'll provide three possible directions. First we'll check to see if `promptFight` holds one value. If it doesn't, we'll check to see if it holds a different value. If it holds neither of the values, we'll simply take them in a third direction.

Let's update our `fight` function by wrapping all of the code we wrote that simulated the fighting between the player robot and the enemy robot into the following `if` statement for the fight option as follows:

```
// if player choses to fight, then fight
if (promptFight === "fight" || promptFight === "FIGHT") {
  // remove enemy's health by subtracting the amount set in the player
  enemyHealth = enemyHealth - playerAttack;
  console.log(
    playerName + " attacked " + enemyName + ". " + enemyName + " now h
  );

  // check enemy's health
  if (enemyHealth <= 0) {
    window.alert(enemyName + " has died!");
  } else {
    window.alert(enemyName + " still has " + enemyHealth + " health le
  }

  // remove player's health by subtracting the amount set in the enemy
  playerHealth = playerHealth - enemyAttack;
  console.log(
    enemyName + " attacked " + playerName + ". " + playerName + " now
  );

  // check player's health
  if (playerHealth <= 0) {
    window.alert(playerName + " has died!");
  } else {
    window.alert(playerName + " still has " + playerHealth + " health
  }
  // if player choses to skip
} else if (promptFight === "skip" || promptFight === "SKIP") {
  window.alert(playerName + " has chosen to skip the fight!");
} else {
  window.alert("You need to pick a valid option. Try again!");
}
```

Notice that at the bottom of the code block in the skip option, we send an alert notifying the user the skip option has been selected. The third option will occur if neither the fight or skip options are chosen.

Let's save our `game.js` file and run our game several times to test each option.

Each selection should yield a different result based on the user response. If we chose "FIGHT" or "fight", we ran the battle code that we wrote previously. If we chose "SKIP" or "skip", we left the fight; and if we chose neither, we were informed that we made an incorrect choice.

We used `if`, `else if`, and `else` keywords to have our code pick one of three directions to go based on checking to see what the value of a variable was.

Let's dissect some of this syntax, starting with our initial condition:

```
if (promptFight === "FIGHT" || promptFight === "fight")
```

Keep in mind that anything we put between the parentheses in an `if` statement will result in a true or false value, so when we put `promptFight === "FIGHT"` in there, we're checking to see if the value of `promptFight` is the word "FIGHT". Anything other than that value, different casing included, would result in the condition being false.

Since we cannot always count on a user using all uppercase letters when typing the word, we also want to check to see if they entered the word "fight" in all lowercase letters instead. So now, within one condition check, we're seeing if *either* of these checks result in true. If at least one of them comes back true, then the whole condition is considered true.

We made this happen by using the `||` syntax, which is called an "OR" operator. When we use this in an `if` statement, we're telling the program to execute that code as long as at least one of the condition checks results in true. If both are false, then move on.

DEEP DIVE

DEEP DIVE

The OR `||` operator isn't the only logical operator available in JavaScript. To learn more, see the **MDN web docs on logical operators (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators)**.

So what happens if the user doesn't enter "FIGHT" or "fight"? What if they type "Fight" instead? That would be considered false because we don't explicitly check for that permutation of the word. We'll explore other methods of fixing issues like this later in the game, but for now let's focus on what happens if the first `if` statement is false and the code doesn't run.

After the block (enclosed in curly braces `{ }`) holding the first `if` statement's code ends, we can see something new:

```
else if ([Condition])
```

This means that if the first `if` statement fails, then try another one. If the first `if` statement is true, then we'll never get to checking the `else if` statement. This is used as a fall-back condition check.

In our case, we're using `else if` to do a similar check as our `if` statement, but now we're checking to see if the user wrote in "SKIP" or "skip" instead. But what happens if the user doesn't pick that either?

If the user doesn't choose one of the values we've explicitly listed, then our code reaches the `else` block of code. Think of the `else` code as a default action if all else fails.

Leave the Fight, at a Cost

We have all of the important pieces in place to make this game interactive. They can pick their robot's name and decide if they want to go through with the battle, which will play heavily into when our robot battles multiple enemies soon. There is one issue with our logic, however. What is there to stop a player from skipping all of the fights and finishing with full health?

This is a hard question to answer, as there are a lot of different answers we can provide. We could penalize a player by subtracting health points upon leaving a fight, but that doesn't make too much sense, as the whole point of leaving the fight is to retain health for the next possible battle. So rather than gauging a player's success based on how much health they have at the end, let's gauge it based on how much money they have at the end instead.

> **NERD NOTE**
>
> Think about how old arcade games used to display "High Score" lists to rank players. Most games are not about getting to the end, but rather how high of a score they have when the game is over for them.

We'll finish off this lesson by adding in functionality to penalize our player if they choose to skip a fight, so let's start by adding a variable to keep track of how much money they have.

At the top of `game.js` where we declared all our other player-based variables, create another one called `playerMoney` with an initial value of 10:

```
var playerMoney = 10;
```

We now have the ability to keep track of how much money our player has. This will come into play later in the game when we add the ability to purchase items, but for now we'll use it to penalize a player when they choose to skip a fight.

Let's go ahead and add this penalty to our `fight` function. If it should only happen when a user chooses to "SKIP" or "skip" the fight, where should we place it? It should go in the `else if` statement that checks if the user chose one of those values.

In the `else if` statement, we'll add the following steps:

- Ask the user to confirm that they want to quit.
- If they answer "yes," subtract 2 from the `playerMoney` variable and create an alert that lets the user know they're leaving the game.
- If they say "no," execute the `fight` function to start the fight over again. This will give them the choice to fight or skip, so they can choose "fight" and keep playing.

Update the `else if` statement so it looks like this:

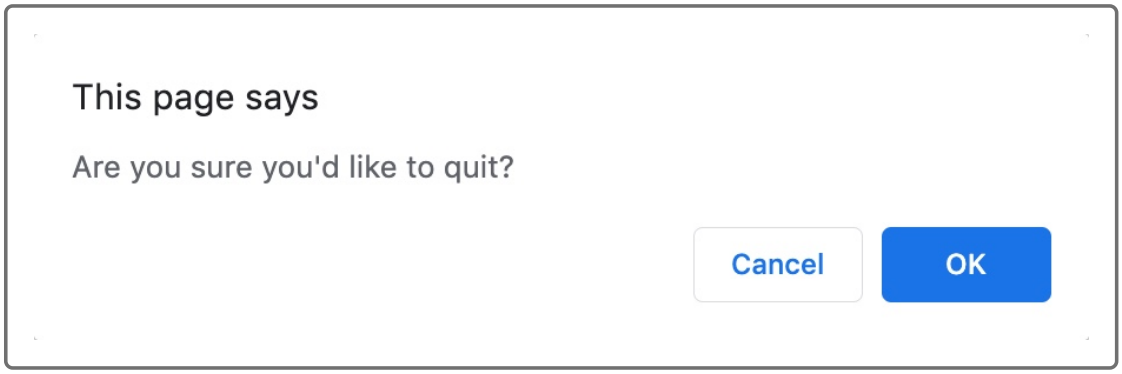
```
} else if (promptFight === "skip" || promptFight === "SKIP") {
  // confirm user wants to skip
  var confirmSkip = window.confirm("Are you sure you'd like to quit?")

  // if yes (true), leave fight
  if (confirmSkip) {
    window.alert(playerName + " has decided to skip this fight. Goodby
    // subtract money from playerMoney for skipping
    playerMoney = playerMoney - 2;
  }
  // if no (false), ask question again by running fight() again
  else {
    fight();
  }
}
```

There are two things to notice about this code:

1. What is this `window.confirm()` thing?

2. Did we just use an `if / else` statement inside another conditional statement?

Both of these should be somewhat familiar you by now. You probably guessed that `window.confirm()` is a built-in browser function. Like `window.prompt()`, it asks the user for input and stores their response in a variable. But instead of letting the user enter whatever they want, it requires a simple "yes" or "no" answer, as shown in the following image:



If the user clicks OK, the Boolean value `true` is returned and stored in the `confirmSkip` variable. If the user clicks Cancel, the `false` value is returned instead.

Think about how we can use this for our game. If the user chooses to "skip" the fight, we should ask them to confirm that because it will make them leave the game. If they click OK, then the `confirmSkip` value is true and the game will end. Otherwise, the `fight` function will restart.

This is where the next part comes in, where we check a condition inside another condition. This is a very common practice in programming, and we can nest conditional statements as much as we need to—within reason, of course.

Remember that with conditional statements, we're checking to see if something is true or false. So what is the value of `confirmSkip` if we choose to quit? The value is `true`.

Consider these two examples:

```
var confirmSkip = true;

if (confirmSkip === true) {
  // do something
}

if (confirmSkip) {
  // do something
}
```

In the first example, we're directly checking to see if the value of `confirmSkip` is the Boolean value `true`. In the second, we're asking "Is `true` true?" They achieve the same results, but the second one uses less code and can also be used in other ways (which we'll see later).

If we now run our program and choose to skip, we'll be asked if we really want to quit. If we choose to quit, then we'll have 2 subtracted from our `playerMoney` variable's value and then, because there's nothing else to do, the program will end.

If the user chooses skip and then decides that they don't want to quit (i.e., they click Cancel), then the value of `confirmSkip` is false. This conditional logic asks "Is `false` true?". Because the user clicked Cancel, `false` does not equal true.

DEEP DIVE

DEEP DIVE

We can use JavaScript conditional logic in a number of ways. Because JavaScript tries to define logic in simple `true` or `false` fashion, our conditions might not always fit that bill so easily. Luckily, JavaScript knows this and can do a little magic to make certain conditions and values result in `true` or `false`.

These are called **truthy** and **falsy** values. For more information, see the **MDN web docs on truthy values (https://developer.mozilla.org/en-US/docs/Glossary/Truthy)** and the **MDN web docs on falsy values (https://developer.mozilla.org/en-US/docs/Glossary/Falsy)** .

Excellent work! You now have one round of the Robot Gladiators game running and can move on to adding more features into it. But first, don't forget to add, commit, and push your code up to the GitHub feature branches!

Now that we have reached the end of our lesson, let's do a quick knowledge check:

In the following code, what would be logged to the console?

```
if (5 === 5) {
  console.log("Hi!");
}
else {
  console.log("Hello!");
}
```

- ◯ "Hi!"
- ◯ "Hello!"

Check Answer

In the following code, what would be logged to the console?

```
var name = "lernantino";

if (name === "Lernantino") {
  console.log("dogs!");
}
else {
  console.log("cats!");
}
```

- ◯ "dogs!"
- ◯ "cats!"

Check Answer

In the following code, what would be logged to the console?

```
var name = "lernantino";

if (name) {
  console.log("sheep!");
}
else {
  console.log("goats!");
}
```

- ◯ "sheep!"
- ◯ "goats!"

Check Answer

Which of these values is NOT considered false?

- ◯ 0
- ◯ "0"
- ◯ null

- ○ Null
- ○ ""

Check Answer

Finish ▸