3.1.7 Use Conditions to Make Decisions

State, a popular programming term, refers to the data in our application at a specific point in time. It not only conveys what's happening now but can also help us decide what will happen next.

For example, in the real world, after exercising, we may want to take a shower or drink a glass of water. If it's late at night and we feel tired, we decide to go to sleep. In both cases, we perform a mental check of our current state, and based on that information, we determine what to do next.

In JavaScript, we can perform similar checks on the status of our code and help the program determine what to do. These checks, called **conditions**, are another crucial set of tools.

Here's what a condition looks like in JavaScript:

```
var playerHealth = 100;

// check to see if the value of the playerHealth variable is greater t
if (playerHealth > 0) {
  console.log("Your player is still alive!");
}
```

Let's examine that block of code. Using the `if` keyword built into the JavaScript language, we're telling the browser that we're going to check for a condition to be true or false and execute a block of code based on the result.

The condition in question here appears between the parentheses: `playerHealth > 0`. In this case, we're checking whether the value of the variable `playerHealth` is greater than zero. If it is, then the condition is considered "true," and the code between the curly braces executes.

If we set the `playerHealth` variable to a value of zero, then the result would be "false," because zero isn't greater than zero. Because of this, the code between the braces wouldn't run at all.

In that very basic example, we gave the program one condition to check and a block of code to execute if that condition was true. We only gave our code one direction based on that condition. Now let's give it two possible directions.

Take a look at the following code block:

```
var playerHealth = 10;

if (playerHealth === 0) {
  console.log("This will not run.");
}
else {
  console.log("This will run instead.");
}
```

We just used a **control flow** statement. We used the `if` keyword to check if the value of `playerHealth` was zero. (In JavaScript, three equal signs allows us to check if two values are directly equal to one another.) These two values didn't equal one another, so the program skipped the first code block and ran the second instead. The JavaScript `else` keyword denoted the second code block.

We wrote our code to take a certain path depending on a condition. The condition used here would always result in running the `else` code block, because 10 will never equal 0—but consider what happens when the condition we're checking is the current value of our robot's health. That value will change throughout our application, so checking it from time to time enables us to change the game's direction if it gets down to zero.

DEEP DIVE

DEEP DIVE

For more information, see the **MDN web docs on conditionals (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals)**.

Check Robot Health

In our `fight` function, we currently have the two robots attacking each other but no way to determine if either can still fight again. This means that if we wanted to, we could execute the `fight` function over and over until both of the robots' health values fall into the negatives. Because health should stop decreasing when it gets to zero, we'll need to check whether it gets there.

In the `fight` function, add the following code after we `console.log()` that our enemy has been attacked by the player:

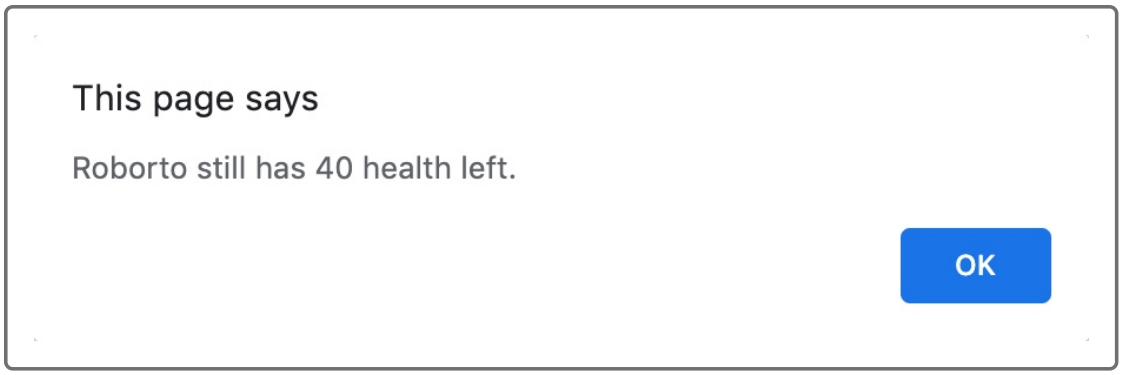
```
// put new code under this
console.log(
  playerName + " attacked " + enemyName + ". " + enemyName + " now has
```

```
);

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

When we save this file and refresh our browser, we should receive two alerts from the window. The first alert will display "Welcome to Robot Gladiators!". Which alert message will occur next?

It will be this one:



Notice in the alert message, the enemy robot health points have diminished by 10 points due to the battle.

To see the other alert message, we must temporarily lower the initial `enemyHealth` value to under 10 or temporarily increase the `playerAttack` value to 50 or more to test the condition. This will ensure that after one round of battle, the enemy will no longer have any health points to trigger the `else` condition in our control flow statement.

Make one (or both) of these changes, and reload the page to see what happens. This will only be momentary for testing purposes since later there will be multiple rounds of fighting that will incur more damage.

Notice the operator in the condition. The `<=` operator evaluates anything less than or equal to zero to be the true condition and will execute the following alert message:

Once you have triggered the "Roboto has died!" alert, restore the original variable values.

HIDE HINT

When writing conditions based on values, try running the code with different values to see how it works at those different values. Just don't forget to set it back to what you want it to start at when done testing!

Now when our `fight` function executes, it performs the following tasks in this order:

1. We let the user know the fight has begun.

2. We have our robot attack the enemy robot by subtracting our robot's `playerAttack` value by the enemy's `enemyHealth` value, resulting in our enemy's new `enemyHealth` value.

3. We then check to see if that last attack destroyed our robot and got its health down to zero or below by using a conditional statement.
    - If `enemyHealth` is zero or below, the enemy robot has lost.
    - Else `enemyhealth` is above 0 and the enemy robot can still fight.
4. We have the enemy robot attack the player. (Later in the project, we'll make sure that dead robots can't fight!)

There is one more piece we need to add to our `fight` function for it to complete a full round of attacks. If we're checking to see if the enemy robot has died, then we also need to check if our robot has died as well!

To do that, we'll use the same conditional statements that we used for the enemy robot. Add the following to the `fight` function after the enemy robot attacks our player:

```
// put new code under this
console.log(enemyName + " attacked " + playerName + ". " + playerName

// check player's health
if (playerHealth <= 0) {
  window.alert(playerName + " has died!");
}
else {
  window.alert(playerName + " still has " + playerHealth + " health le
}
```

Now when we save our `game.js` file and refresh the browser, we'll see three alert dialog windows pop up. The third one will be about our robot's status. If our robot's health (represented by the `playerHealth` variable) is zero or below, we'll receive an alert that our robot has died. If it's above zero, then we'll receive an alert about how much health our robot has left.

See if you can temporarily change the `enemyAttack` or `playerHealth` values to trigger the alert announcing the death of your robot. (Don't worry;

when you restore the original values your robot will be back on their feet!)

Get That MVP!

Our Robot Gladiators game is starting to take shape! The application can run one round of the game, pitting two robots against each other and checking if either robot has lost. The app can also interact with the user through browser alerts and prompts.

However, the user's interaction doesn't actually affect the outcome of the game, so it's not as much of a game as it is a simulation of a battle. Thus, it hasn't reached MVP status yet. This is a game jam after all, so we need to add meaningful user interaction to it. We'll do that by giving them a choice of skipping the fight for a fee or going ahead with the round of fighting.

Let's commit our code and get that MVP!