3.4.6 Convert Data to Custom Objects

We just spent a fair amount of time working with the `Math` object. Objects can have both properties (`Math.PI`) and methods/functions (`Math.random()`). There are many other built-in JavaScript objects that we'll discover over time, and we can even make our own! Here's an example of a simple custom object:

```
var food = {
  name: "Banana",
  type: "fruit",
  calories: 105
};
```

Objects are created using curly brackets, and object properties are defined within using `property: value` syntax and separated by a comma. Accessing these properties works just like it did for the `Math` object:

```
console.log(food.name); // "Banana"
console.log(food.type); // "fruit"
console.log(food.calories); // 105
```

We can use this same syntax to create a new player object. This would help keep all of our player data coupled together, something that would become even more important if we were to have multiple players later on with hundreds of different properties each.

At the top of `game.js`, delete the four player variables (`playerName`, `playerHealth`, `playerAttack`, `playerMoney`) and replace them with an object:

```
var playerInfo = {
  name: window.prompt("What is your robot's name?"),
  health: 100,
  attack: 10,
  money: 10
};
```

That will momentarily break the game because we now have references to undefined variables all over the place. We'll need to update these variable references to point to the object:

- Replace all instances of `playerName` with `playerInfo.name`
- Replace all instances of `playerHealth` with `playerInfo.health`
- Replace all instances of `playerAttack` with `playerInfo.attack`
- Replace all instances of `playerMoney` with `playerInfo.money`

HIDE PRO TIP

In VS Code, press Ctrl+F on Windows or Command+F on Mac to open the Find and Replace menu.

Save and test the game to make sure you didn't miss any variables. Note that switching to a player object didn't change the game at all, but it did consolidate a lot of important data. Accessing this data also makes for more readable code, because `playerInfo.health` establishes a direct link between the health property and its owner.

HIDE PRO TIP

Another way to access object properties is with bracket notation: `playerInfo["health"]`. This is useful in situations where the property you're looking up is based on a variable. For instance:

```
var userInput = "money";

// will equate to playerInfo["money"], which is the same as
console.log(playerInfo[userInput]);
```

While we're at it, delete the `enemyNames`, `enemyHealth`, and `enemyAttack` variables. Under the player object, create a new array of enemy objects:

```
var enemyInfo = [
  {
    name: "Roborto",
    attack: 12
  },
  {
    name: "Amy Android",
    attack: 13
  },
  {
    name: "Robo Trumble",
    attack: 14
```

```
  }
];
```

Even though the data in the array looks much different, it's still an array with numerical indexes. That means the first robot object can be accessed as `enemyInfo[0]`, and getting that robot's name is as simple as `enemyInfo[0].name`.

Note that we didn't include `health` as a property of the array objects. Once an object has been defined, properties can still be added after the fact. For example, we could add an extra property to the first robot only: `enemyInfo[0].special = true;`. We'll postpone defining the health to better demonstrate this idea.

In the `startGame()` function, adjust the following lines of code to reference the enemy array:

- In the `for` loop, replace both mentions of `enemyNames.length` with `enemyInfo.length`
- Replace `var pickedEnemyName = enemyNames[i];` with `var pickedEnemyObj = enemyInfo[i];`
- Change `enemyHealth = randomNumber(40, 60);` to `pickedEnemyObj.health = randomNumber(40, 60);`
- Change `fight(pickedEnemyName);` to `fight(pickedEnemyObj);`

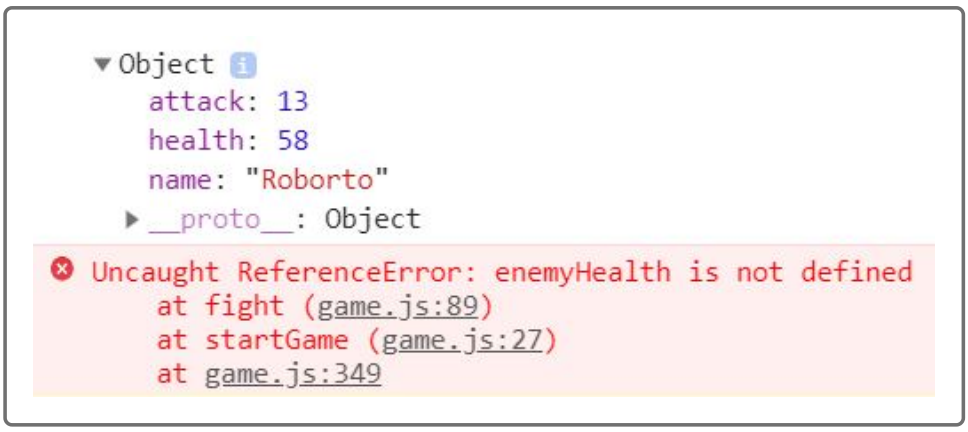
Whoa, hold on. We were originally passing a string (`pickedEnemyName`) into the `fight()` function, but now we're passing an entire object. We're allowed to do that, of course, but that means the code inside the function must change to reflect the object.

First, rename the function parameter to something more appropriate:

```
var fight = function(enemy) {
  console.log(enemy);

  // other logic...
};
```

If you console log `enemy`, you'll see that it's an object with three properties, including the newly added `health` property:



Unfortunately, the console displays an error immediately afterwards, because we still have leftover variables like `enemyHealth` that need to be swapped out.

Delete the `console.log(enemy);` statement, and then make the following changes in the `fight()` function:

- Replace all instances of `enemyHealth` with `enemy.health`
- Replace all instances of `enemyName` with `enemy.name`
- Replace all instances of `enemyAttack` with `enemy.attack`

That should take care of the remaining errors. Test the game again to make sure you didn't forget any.

DEEP DIVE

DEEP DIVE

Passing objects into a function presents an interesting gotcha in JavaScript. In the previous example, where we defined `enemy` as a parameter, that does not create a brand new object called `enemy`. Instead, it creates a reference to the original object. Therefore, updating a property on `enemy` also updates the original object.

Here's a smaller example to demonstrate **passing by reference**:

```
var oldObj = {
 name: "test",
 count: 1
};

var addOne = function(newObj) {
 // increment count property of newObj by one
 newObj.count = newObj.count + 1;
};

// pass oldObj into the function
addOne(oldObj);

console.log(oldObj.count); // prints 2
```

Updating `newObj` in the function also updated `oldObj`. Sometimes this behavior can work to our advantage, like with our enemy objects. But if you're not aware that JavaScript does this, it can feel like something's broken!

Note that passing by reference applies to objects and arrays.

Now that we have an array of enemy objects, we can easily define different attack values for each. In fact, why not make these attack values random using our handy `randomNumber()` function?

Update the objects in the `enemyInfo` array as such:

```
var enemyInfo = [
  {
    name: "Roborto",
    attack: randomNumber(10, 14)
  },
  {
    name: "Amy Android",
    attack: randomNumber(10, 14)
  },
  {
    name: "Robo Trumble",
    attack: randomNumber(10, 14)
  }
];
```

If you test the game, though, you'll get the following error: `Uncaught TypeError: randomNumber is not a function`.

PAUSE

Why does the browser think `randomNumber` is not a function?

The `enemyInfo` array is being defined before `randomNumber`.

Hide Answer

```
playerInfo.money = 10;

// in shop()
playerInfo.health = playerInfo.health + 20;
playerInfo.money = playerInfo.money - 7;
```

We could consolidate these updates into methods like `playerInfo.reset()`. This would be helpful for a few reasons:

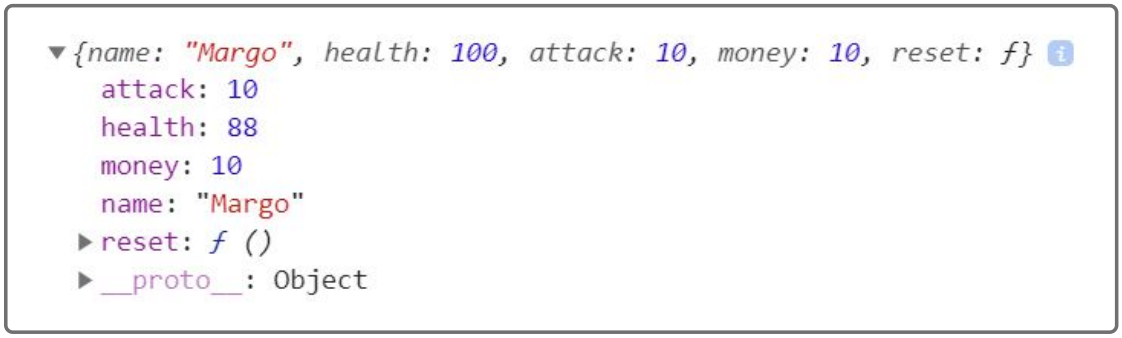
- The player object becomes an even more valuable "source of truth" for all things related to player data.
- It declutters the main game logic, which can already be somewhat difficult to follow.
- The intentions are clearer (e.g., `playerInfo.reset()` is self-explanatory).

Revisit the `playerInfo` object and add another property, this time in the form of a method/function:

```
var playerInfo = {
  name: window.prompt("What is your robot's name?"),
  health: 100,
  attack: 10,
  money: 10,
  reset: function() {
    this.health = 100;
    this.money = 10;
    this.attack = 10;
  }
};
```

This does introduce a new keyword, though: `this`. Take a moment to think about what `this` might mean. Because `reset()` is a method that belongs to the `playerInfo` object, we need a way for the method to self-

reference its owner. If you were to console log `this` inside the `reset()` method, you would see that it is, in fact, the entire original object:



Using `this`, we not only have access to all of the object's properties but its methods too! You can think of it as, "`this` refers to THIS object." So if we update a property on `this` (e.g., `this.health = 100`), it will update the original object.

Now that we have a `reset()` method, update the beginning of the `startGame()` function to call the method instead of writing `playerInfo.health = 100`, `playerInfo.attack = 10`, etc:

```
var startGame = function() {
  // reset player stats
  playerInfo.reset();

  // other game logic...
};
```

Test the game again to verify that nothing broke in the process. If you see an error like `Uncaught SyntaxError: Unexpected identifier`, it usually means we forgot to type a character that JavaScript needed. Remember that object properties and methods are separated by commas. Here's an example of a common syntactical error:

```
var food = {
  name: "Banana",
```

Move the `enemyInfo` array and `playerInfo` object closer to the bottom of the `game.js` file, directly above the call to `startGame()`. Organizing the code this way ensures that all functions are defined before other objects or methods try to use them.

At this point, we've worked with objects quite a bit, and there may still be some uneasiness in our understanding of all the different pieces such as properties and methods. This video will help give us a little clarity before we move on:

We've made great strides in optimizing our code with objects, but we've only scratched the surface of what's possible. Remember, objects can also have methods, where methods are functions that belong to an object. What methods would be useful to have on our `playerInfo` object? We have a few places in the code where multiple player values are being updated at once:

```
// in startGame()
playerInfo.health = 100;
playerInfo.attack = 10;
```

```
  type: "fruit"
  // JavaScript expected a comma on the previous line but didn't see c
  calories: 105
};
```

Now that you have a `reset()` method, add two more methods to the `playerInfo` object to update the health and attack properties:

```
var playerInfo = {
  name: window.prompt("What is your robot's name?"),
  health: 100,
  attack: 10,
  money: 10,
  reset: function() {
    this.health = 100;
    this.money = 10;
    this.attack = 10;
  }, // comma!
  refillHealth: function() {
    this.health += 20;
    this.money -= 7;
  }, // comma!
  upgradeAttack: function() {
    this.attack += 6;
    this.money -= 7;
  }
};
```


REWIND

Remember the `+=` and `-=` syntax? Those operators simplify coding an addition or a substraction from a variable. So, in the code above, `this.health += 20` is shorthand for `this.health = this.health + 20`, and `this.money -= 7` is shorthand for `this.money = this.money - 7`.

This is a common programming trick for writing concise code, similar to writing `i++` instead of `i = i + 1`.

There's nothing stopping us from also writing conditional logic in these methods. Expand the current `refillHealth()` and `upgradeAttack()` code to include `if` statements and `alert()` calls:

```
refillHealth: function() {
  if (this.money >= 7) {
    window.alert("Refilling player's health by 20 for 7 dollars.");
    this.health += 20;
    this.money -= 7;
  }
  else {
    window.alert("You don't have enough money!");
  }
},
upgradeAttack: function() {
  if (this.money >= 7) {
    window.alert("Upgrading player's attack by 6 for 7 dollars.");
    this.attack += 6;
    this.money -= 7;
  }
  else {
    window.alert("You don't have enough money!");
  }
}
```

These `if` statements look a lot like the ones we originally wrote in the `shop()` function. In fact, they're identical! Let's follow the DRY principle and delete most of that old code.

In the `shop()` function's `switch` statement, update the following cases as such:

```
case "REFILL":
case "refill":
  playerInfo.refillHealth();
  break;
case "UPGRADE":
case "upgrade":
  playerInfo.upgradeAttack();
  break;
```

The `switch` statement looks much more readable now! Objects can greatly help in cleaning up an application's main logic. They're also useful for keeping like data coupled together (e.g., player stats) and making apps easier to scale up. For instance, what if, in a later version of the game, we wanted to add a shield object to every enemy? Without objects, it could be a huge hassle to track all of those different variables. With objects, it's simply a matter of adding another property:

```
var enemy = {
  name: "Roberto",
  attack: randomNumber(10, 14),
  shield: {
    type: "wood",
    strength: 10
  }
};
```

In future applications, continue thinking about where and how objects can be used. They're valuable tools, right up there with functions and arrays.