Tip 2: Reduce Scope Conflicts with let and const

In this tip, you'll learn that in cases where a value is going to change, let is the best choice.

We'll cover the following

You saw in the previous tip that when you're working with variables, you're better off avoiding reassignment. *But what do you do in situations where you really need to reassign a variable?* In those cases, you should use `let`.

Declaration using `let`

`let` is similar to `var` because it can be reassigned, but unlike `var`, which is *lexically scoped*, `let` is *block scoped*. You'll explore scope more in Tip 3, Isolate Information with Block Scoped Variables. For now, just know that **block-scoped** variables exist *only in blocks*, such as an `if` block or a `for` loop. Outside those blocks, they aren't accessible. As a rule, this means *the variable doesn't exist outside the curly braces in which it was declared*.

Example

To see how a block-scoped or a lexically scoped variable can change code, consider an example. This code looks for the *lowest price* for an item. To find the lowest price, it makes three simple checks:

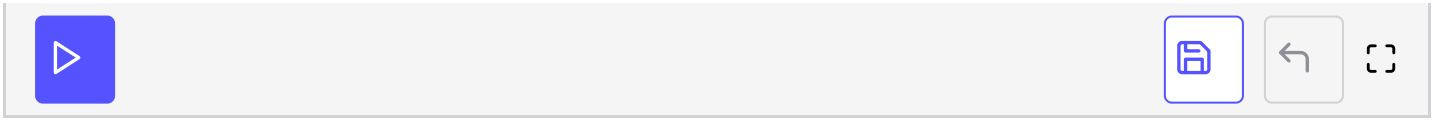
- If there is no inventory: **Return 0**.
- If there is a sale price and sale inventory: **Return sale price**.
- If there is no sale price or no sale inventory: **Return price**.

```
function getLowestPrice(item) {
  var count = item.inventory;
  var price = item.price;

  if (item.salePrice) {
    var count = item.saleInventory;
    if (count > 0) {
      price = item.salePrice;
    }
  }

  if (count) {
    return price;
  }

  return 0;
}
```



Take a moment and see if you can find the bug.

Look at each expected outcome and see what you can find.

Did you find it? *The problem is that you're reassigning a variable to the same variable name*.

If you have an `item` with no `inventory` and no sale price, the `item.salePrice` conditional will be skipped and you'll get `0`.

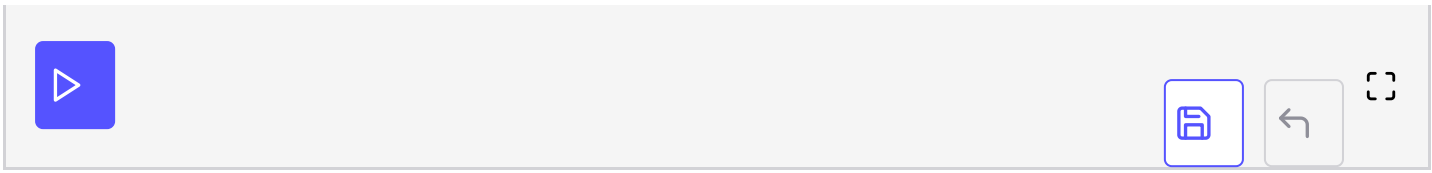
```
function getLowestPrice(item) {
  var count = item.inventory;
  var price = item.price;

  if (item.salePrice) {
    var count = item.saleInventory;
    if (count > 0) {
      price = item.salePrice;
    }
  }

  if (count) {
    return price;
  }

  return 0;
}

const item = {
  inventory: 0,
  price: 3,
  salePrice: 0,
  saleInventory: 0,
};
console.log(getLowestPrice(item));
```



Next, if you have a sale price and a sale inventory, you get the sale price. In this case, the returned value will be `2`.

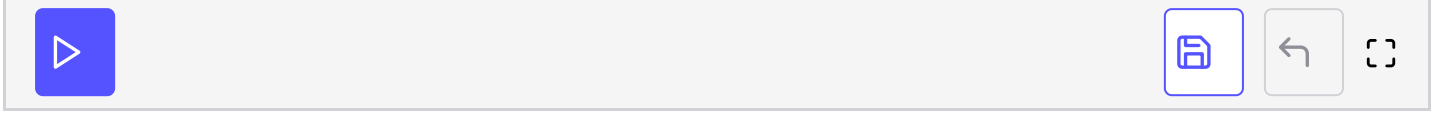
```
function getLowestPrice(item) {
  var count = item.inventory;
  var price = item.price;

  if (item.salePrice) {
    var count = item.saleInventory;
    if (count > 0) {
      price = item.salePrice;
    }
  }

  if (count) {
    return price;
  }

  return 0;
}

const item = {
  inventory: 3,
  price: 3,
  salePrice: 2,
  saleInventory: 1,
};
console.log(getLowestPrice(item));
```



Finally, if you have a sale price but no sale inventory, you expect to get the regular price, `3`. What you actually get is `0`.

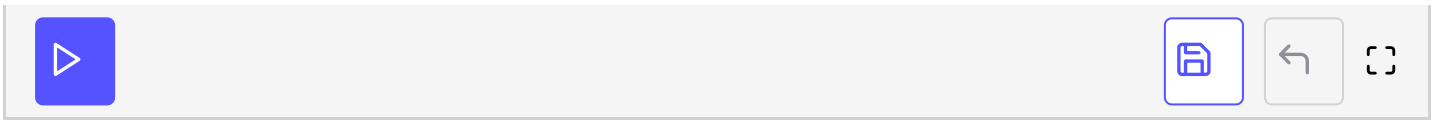
```
function getLowestPrice(item) {
  var count = item.inventory;
  var price = item.price;

  if (item.salePrice) {
    var count = item.saleInventory;
    if (count > 0) {
      price = item.salePrice;
    }
  }

  if (count) {
    return price;
  }
```

```
  return 0;
}

const item = {
  inventory: 3,
  price: 3,
  salePrice: 2,
  saleInventory: 0,
};
console.log(getLowestPrice(item));
```



If you're still a little confused, that's okay. It's a tricky bug. The problem is that you declare the variable `count` on **lines 2 to 3**. There's a sale price, so you go into the next `if` block. At this point, you redeclare the variable `count` on **line 6**. Now the problem is that this is set to `0` because there's no more sale inventory. By the time you get to the next `if` block on **line 12**, the inventory is wrong. It looks like there's no sale inventory and no regular priced inventory.

Even though you have a regular inventory, you're accidentally checking the sale inventory and returning the wrong value.

You might want to dismiss this problem as trivial. But bugs like this are subtle and hard to catch if they make it into production.

Avoiding variable redeclaration using `let`

You can avoid the issue above using `let`. In fact, `let` helps you avoid this issue in two ways.

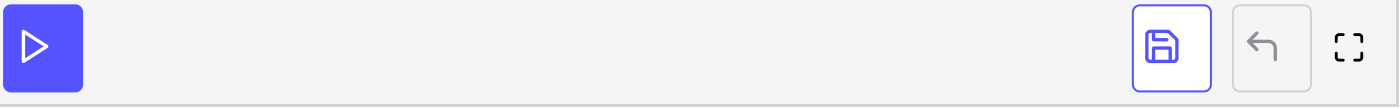
Method 1

`let` is block scoped, which again means any *variable declared inside a block doesn't exist outside the block*.

```
function getLowestPrice(item) {
    let count = item.inventory;
    let price = item.price;
    if (item.salePrice) {
        let count = item.saleInventory;
        if (count > 0) {
            price = item.salePrice;
        }
    }
    if (count) {
        return price;
    }
```

```
    return 0;
}

const item = {
  inventory: 3,
  price: 3,
  salePrice: 2,
  saleInventory: 0,
};
console.log(getLowestPrice(item));
```



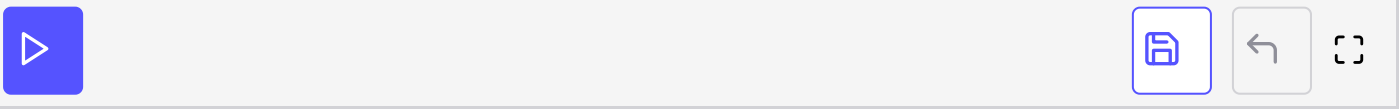
In this case, using `let` to declare the `count` variable in the `if` block isn't going to conflict with the `count` variable declared at the start of the function.

Method 2

Of course, `let` isn't the only variable declaration that's block scoped. `const` is also block scoped. Because you're never reassigning `count`, you can use `const` instead and keep things even more clear, although you'll need to continue to use `let` to declare `price` because that may update. Honestly, you should just use different names to keep things clear. The final code would be this:

```
function getLowestPrice(item) {
    const count = item.inventory;
    let price = item.price;
    if (item.salePrice) {
        const saleCount = item.saleInventory;
        if (saleCount > 0) {
            price = item.salePrice;
        }
    }
    if (count) {
        return price;
    }
    return 0;
}

const item = {
  inventory: 3,
  price: 3,
  salePrice: 2,
  saleInventory: 0,
};
console.log(getLowestPrice(item));
```



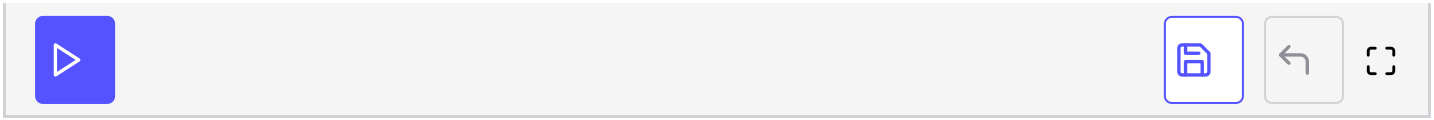
As an added bonus, `let` and `const` have another protection. You can't redeclare a

variable of the same name. With `var`, you can redeclare a variable of the same name in the same scope. In other words, you can say `var price = 1` at, say **line 10** and `var price = 5` at **line 25** with no conflict. This can be a huge problem if you unintentionally reuse a variable name. With `let`, you can't make this mistake.

This code would generate a `TypeError`.

```
function getLowestPriceDeclaration(item) {
    const count = item.inventory;
    let price = item.price;
    if (!count) {
        return 0;
    }
    // ...
    let price = item.saleInventory ? item.salePrice : item.wholesalePrice;
    return price;
}

const item = {
  inventory: 3,
  price: 3,
  salePrice: 2,
  saleInventory: 0,
};
console.log(getLowestPriceDeclaration(item));
```



This issue won't come up often, but it's a nice way to catch a potential bug early in the process.

1

What will be the output of the code below?

```
let value=55;
let value=47;
console.log(value);
```

2

What will be the output of the code below?

```
1- let value1 = 5;
2- function func() {
3-     let value2 = 6;
4-
5-     if (value2 > value1) {
6-         let value3 = 9;
7-         value2++;
8-     }
9-
10-     console.log(value1);
11-     console.log(value2);
12-     console.log(value3);
13- }
14-
15- func();
```

Retake Quiz

In the next tip, you'll take a deeper look into scope and how let solves one of the most common and perplexing scope conflicts in JavaScript.