Variable Declarations

Variable Declarations

`let` and `const` are two relatively new types of variable declarations in JavaScript. As we mentioned earlier, `let` is similar to `var` in some respects, but allows users to avoid some of the common "gotchas" that users run into in JavaScript. `const` is an augmentation of `let` in that it prevents re-assignment to a variable.

With TypeScript being a superset of JavaScript, the language naturally supports `let` and `const`. Here we'll elaborate more on these new declarations and why they're preferable to `var`.

If you've used JavaScript offhandedly, the next section might be a good way to refresh your memory. If you're intimately familiar with all the quirks of `var` declarations in JavaScript, you might find it easier to skip ahead.

`var` declarations

Declaring a variable in JavaScript has always traditionally been done with the `var` keyword.

```
var a = 10;
```

As you might've figured out, we just declared a variable named `a` with the value `10`.

We can also declare a variable inside of a function:

```
function f() {
    var message = "Hello, world!";

    return message;
}
```

and we can also access those same variables within other functions:

```
function f() {
    var a = 10;
    return function g() {
        var b = a + 1;
        return b;
    }
}

var g = f();
g(); // returns '11'
```

In this above example, `g` captured the variable `a` declared in `f`. At any point that `g` gets called, the value of `a` will be tied to the value of `a` in `f`. Even if `g` is called once `f` is done running, it will be able to access and modify `a`.

```
function f() {
    var a = 1;

    a = 2;
    var b = g();
    a = 3;

    return b;

    function g() {
        return a;
    }
}

f(); // returns '2'
```

Scoping rules

`var` declarations have some odd scoping rules for those used to other languages. Take the following example:

```
function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        var x = 10;
    }

    return x;
}

f(true);  // returns '10'
f(false); // returns 'undefined'
```

Some readers might do a double-take at this example. The variable `x` was declared *within the `if` block*, and yet we were able to access it from outside that block. That's because `var` declarations are accessible anywhere within their containing function, module, namespace, or global scope - all which we'll go over later on - regardless of the containing block. Some people call this *`var`-scoping* or *function-scoping*. Parameters are also function scoped.

These scoping rules can cause several types of mistakes. One problem they exacerbate is the fact that it is not an error to declare the same variable multiple times:

```
function sumMatrix(matrix: number[][]) {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

Maybe it was easy to spot out for some, but the inner `for`-loop will accidentally overwrite the variable `i` because `i` refers to the same function-scoped variable. As experienced developers know by now, similar sorts of bugs slip through code reviews and can be an endless source of frustration.

Variable capturing quirks

Take a quick second to guess what the output of the following snippet is:

```
for (var i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}
```

For those unfamiliar, `setTimeout` will try to execute a function after a certain number of milliseconds (though waiting for anything else to stop running).

Ready? Take a look:

```
10
10
10
10
10
10
10
10
10
10
```

Many JavaScript developers are intimately familiar with this behavior, but if you're surprised, you're certainly not alone. Most people expect the output to be

```
0
1
2
3
4
5
6
7
8
9
```

Remember what we mentioned earlier about variable capturing? Every function expression we pass to `setTimeout` actually refers to the same `i` from the same scope.

Let's take a minute to consider what that means. `setTimeout` will run a function after some number of milliseconds, *but only* after the `for` loop has stopped executing; By the time the `for` loop has stopped executing, the value of `i` is `10`. So each time the given function gets called, it will print out `10`!

A common work around is to use an IIFE - an Immediately Invoked Function Expression - to capture `i` at each iteration:

```
for (var i = 0; i < 10; i++) {
    // capture the current state of 'i'
    // by invoking a function with its current value
    (function(i) {
        setTimeout(function() { console.log(i); }, 100 * i);
    })(i);
}
```

This odd-looking pattern is actually pretty common. The `i` in the parameter list actually shadows the `i` declared in the `for` loop, but since we named them the same, we didn't have to modify the loop body too much.

`let` declarations

By now you've figured out that `var` has some problems, which is precisely why `let` statements were introduced. Apart from the keyword used, `let` statements are written the same way `var` statements are.

```
let hello = "Hello!";
```

The key difference is not in the syntax, but in the semantics, which we'll now dive into.

Block-scoping

When a variable is declared using `let`, it uses what some call *lexical-scoping* or *block-scoping*. Unlike variables declared with `var` whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block or `for`-loop.

```
function f(input: boolean) {
    let a = 100;

    if (input) {
        // Still okay to reference 'a'
        let b = a + 1;
        return b;
    }

    // Error: 'b' doesn't exist here
    return b;
}
```

Here, we have two local variables `a` and `b`. `a`'s scope is limited to the body of `f` while `b`'s scope is limited to the containing `if` statement's block.

Variables declared in a `catch` clause also have similar scoping rules.

```
try {
    throw "oh no!";
}
catch (e) {
    console.log("Oh well.");
}

// Error: 'e' doesn't exist here
console.log(e);
```

Another property of block-scoped variables is that they can't be read or written to before they're actually declared. While these variables are "present" throughout their scope, all points up until their declaration are part of their *temporal dead zone*. This is just a sophisticated way of saying you can't access them before the `let` statement, and luckily TypeScript will let you know that.

```
a++; // illegal to use 'a' before it's declared;
let a;
```

Something to note is that you can still *capture* a block-scoped variable before it's declared. The only catch is that it's illegal to call that function before the declaration. If targeting ES2015, a modern runtime will throw an error; however, right now TypeScript is permissive and won't report this as an error.

```
function foo() {
    // okay to capture 'a'
    return a;
}

// illegal call 'foo' before 'a' is declared
// runtimes should throw an error here
foo();

let a;
```

For more information on temporal dead zones, see relevant content on the Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#Temporal_dead_zone_and_errors_with_let).

Re-declarations and Shadowing

With `var` declarations, we mentioned that it didn't matter how many times you declared your variables; you just got one.

```
function f(x) {
    var x;
    var x;

    if (true) {
        var x;
    }
}
```

In the above example, all declarations of `x` actually refer to the *same* `x`, and this is perfectly valid. This often ends up being a source of bugs. Thankfully, `let` declarations are not as forgiving.

```
let x = 10;
let x = 20; // error: can't re-declare 'x' in the same scope
```

The variables don't necessarily need to both be block-scoped for TypeScript to tell us that there's a problem.

```
function f(x) {
    let x = 100; // error: interferes with parameter declaration
}

function g() {
    let x = 100;
    var x = 100; // error: can't have both declarations of 'x'
}
```

That's not to say that block-scoped variable can never be declared with a function-scoped variable. The block-scoped variable just needs to be declared within a distinctly different block.

```
function f(condition, x) {
    if (condition) {
        let x = 100;
        return x;
    }

    return x;
}

f(false, 0); // returns '0'
f(true, 0);  // returns '100'
```

The act of introducing a new name in a more nested scope is called *shadowing*. It is a bit of a double-edged sword in that it can introduce certain bugs on its own in the event of accidental shadowing, while also preventing certain bugs. For instance, imagine we had written our earlier `sumMatrix` function using `let` variables.

```
function sumMatrix(matrix: number[][]) {
    let sum = 0;
    for (let i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (let i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

This version of the loop will actually perform the summation correctly because the inner loop's `i` shadows `i` from the outer loop.

Shadowing should *usually* be avoided in the interest of writing clearer code. While there are some scenarios where it may be fitting to take advantage of it, you should use your best judgement.

Block-scoped variable capturing

When we first touched on the idea of variable capturing with `var` declaration, we briefly went into how variables act once captured. To give a better intuition of this, each time a scope is run, it creates an "environment" of variables. That environment and its captured variables can exist even after everything within its scope has finished executing.

```
function theCityThatAlwaysSleeps() {
    let getCity;

    if (true) {
        let city = "Seattle";
        getCity = function() {
            return city;
        }
    }

    return getCity();
}
```

Because we've captured `city` from within its environment, we're still able to access it despite the fact that the `if` block finished executing.

Recall that with our earlier `setTimeout` example, we ended up needing to use an IIFE to capture the state of a variable for every iteration of the `for` loop. In effect, what we were doing was creating a new variable environment for our captured variables. That was a bit of a pain, but luckily, you'll never have to do that again in TypeScript.

`let` declarations have drastically different behavior when declared as part of a loop. Rather than just introducing a new environment to the loop itself, these declarations sort of create a new scope *per iteration*. Since this is what we were doing anyway with our IIFE, we can change our old `setTimeout` example to just use a `let` declaration.

```
for (let i = 0; i < 10 ; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}
```

and as expected, this will print out

```
0
1
2
3
4
5
6
7
8
9
```

`const` declarations

`const` declarations are another way of declaring variables.

```
const numLivesForCat = 9;
```

They are like `let` declarations but, as their name implies, their value cannot be changed once they are bound. In other words, they have the same scoping rules as `let`, but you can't re-assign to them.

This should not be confused with the idea that the values they refer to are *immutable*.

```
const numLivesForCat = 9;
const kitty = {
    name: "Aurora",
    numLives: numLivesForCat,
}

// Error
kitty = {
    name: "Danielle",
    numLives: numLivesForCat
};

// all "okay"
kitty.name = "Rory";
kitty.name = "Kitty";
kitty.name = "Cat";
kitty.numLives--;
```

Unless you take specific measures to avoid it, the internal state of a `const` variable is still modifiable. Fortunately, TypeScript allows you to specify that members of an object are `readonly`. The chapter on Interfaces (./interfaces.html) has the details.

`let` vs. `const`

Given that we have two types of declarations with similar scoping semantics, it's natural to find ourselves asking which one to use. Like most broad questions, the answer is: it depends.

Applying the principle of least privilege (https://en.wikipedia.org/wiki/Principle_of_least_privilege), all declarations other than those you plan to modify should use `const`. The rationale is that if a variable didn't need to get written to, others working on the same codebase shouldn't automatically be able to write to the object, and will need to consider whether they really need to reassign to the variable. Using `const` also makes code more predictable when reasoning about flow of data.

Use your best judgement, and if applicable, consult the matter with the rest of your team.

The majority of this handbook uses `let` declarations.

Destructuring

Another ECMAScript 2015 feature that TypeScript has is destructuring. For a complete reference, see the article on the Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment). In this section, we'll give a short overview.

Array destructuring

The simplest form of destructuring is array destructuring assignment:

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

This creates two new variables named `first` and `second`. This is equivalent to using indexing, but is much more convenient:

```
first = input[0];
second = input[1];
```

Destructuring works with already-declared variables as well:

```
// swap variables
[first, second] = [second, first];
```

And with parameters to a function:

```
function f([first, second]: [number, number]) {
    console.log(first);
    console.log(second);
}
f([1, 2]);
```

You can create a variable for the remaining items in a list using the syntax `...`:

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

Of course, since this is JavaScript, you can just ignore trailing elements you don't care about:

```
let [first] = [1, 2, 3, 4];
console.log(first); // outputs 1
```

Or other elements:

```
let [, second, , fourth] = [1, 2, 3, 4];
```

Object destructuring

You can also destructure objects:

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
};
let { a, b } = o;
```

This creates new variables `a` and `b` from `o.a` and `o.b`. Notice that you can skip `c` if you don't need it.

Like array destructuring, you can have assignment without declaration:

```
({ a, b } = { a: "baz", b: 101 });
```

Notice that we had to surround this statement with parentheses. JavaScript normally parses a `{` as the start of block.

You can create a variable for the remaining items in an object using the syntax `...`:

```
let { a, ...passthrough } = o;
let total = passthrough.b + passthrough.c.length;
```

Property renaming

You can also give different names to properties:

```
let { a: newName1, b: newName2 } = o;
```

Here the syntax starts to get confusing. You can read `a: newName1` as " `a as newName1` ". The direction is left-to-right, as if you had written:

```
let newName1 = o.a;
let newName2 = o.b;
```

Confusingly, the colon here does *not* indicate the type. The type, if you specify it, still needs to be written after the entire destructuring:

```
let { a, b }: { a: string, b: number } = o;
```

Default values

Default values let you specify a default value in case a property is undefined:

```
function keepWholeObject(wholeObject: { a: string, b?: number }) {
    let { a, b = 1001 } = wholeObject;
}
```

`keepWholeObject` now has a variable for `wholeObject` as well as the properties `a` and `b`, even if `b` is undefined.

Function declarations

Destructuring also works in function declarations. For simple cases this is straightforward:

```
type C = { a: string, b?: number }
function f({ a, b }: C): void {
    // ...
}
```

But specifying defaults is more common for parameters, and getting defaults right with destructuring can be tricky. First of all, you need to remember to put the pattern before the default value.

```
function f({ a="", b=0 } = {}): void {
    // ...
}
f();
```

> The snippet above is an example of type inference, explained later in the handbook.

Then, you need to remember to give a default for optional properties on the destructured property instead of the main initializer. Remember that `C` was defined with `b` optional:

```
function f({ a, b = 0 } = { a: "" }): void {
    // ...
}
f({ a: "yes" }); // ok, default b = 0
f(); // ok, default to { a: "" }, which then defaults b = 0
f({}); // error, 'a' is required if you supply an argument
```

Use destructuring with care. As the previous example demonstrates, anything but the simplest destructuring expression is confusing. This is especially true with deeply nested destructuring, which gets *really* hard to understand even without piling on renaming, default values, and type annotations. Try to keep destructuring expressions small and simple. You can always write the assignments that destructuring would generate yourself.

Spread

The spread operator is the opposite of destructuring. It allows you to spread an array into another array, or an object into another object. For example:

```
let first = [1, 2];
let second = [3, 4];
let bothPlus = [0, ...first, ...second, 5];
```

This gives bothPlus the value `[0, 1, 2, 3, 4, 5]` . Spreading creates a shallow copy of `first` and `second` . They are not changed by the spread.

You can also spread objects:

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { ...defaults, food: "rich" };
```

Now `search` is `{ food: "rich", price: "$$", ambiance: "noisy" }` . Object spreading is more complex than array spreading. Like array spreading, it proceeds from left-to-right, but the result is still an object. This means that properties that come later in the spread object overwrite properties that come earlier. So if we modify the previous example to spread at the end:

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { food: "rich", ...defaults };
```

Then the `food` property in `defaults` overwrites `food: "rich"` , which is not what we want in this case.

Object spread also has a couple of other surprising limits. First, it only includes an objects' own, enumerable properties (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties). Basically, that means you lose methods when you spread instances of an object:

```
class C {
  p = 12;
  m() {
  }
}
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // error!
```

Second, the Typescript compiler doesn't allow spreads of type parameters from generic functions. That feature is expected in future versions of the language.