Lesson 1: Iteration and the Todo Class

Practice Problems: Emulating Iteration Methods

Some interesting things we can do with **Array.prototype.reduce**:

- Convert an array to an object — this can be handy if you need to do lookups of some sort;
- We can transform short arrays into longer ones — this can be handy if you're reading data from a text file;
- Make two calculations in one traversal — we may need to make two calculations based on a single array;
- Combine mapping and filtering into one pass; and
- Run asynchronous functions in sequence as opposed to parallel — this can be handy if you have a rate limit on API requests or if you need to pass the result of each promise to the next one.

Build a TodoList Class: Getting Started

It's conventional to name methods with a leading underscore when they shouldn't be used from outside the class. The underscore suggests that the method is a private method (i.e., **_validate(index)**).

Build a TodoList Class: Add a **forEach** Method

The idea behind encapsulation is that we should hide implementation details from users of the class. We should neither encourage them to manipulate or use its internal state nor let them become dependent on its implementation. Instead, we want users to use the interface (i.e., the methods) that we provide for them.

For example, we should also prefer to use **TodoList.prototype.forEach** in favor of reaching into the **TodoList** object to access the **todos** array and then use the **Array.prototype.forEach** method. If we later decide to use something other than an array -- perhaps a database -- our users may not be able to use **list.todos.forEach** anymore. However, we can update our version of **forEach** to behave as though we had an array; we only have to determine a way to iterate over the todo objects. Our users won't see any change at all if they use **TodoList.prototype.forEach**.

It's possible to create private data in JS. That is, you can define data in an object that is not accessible from outside that object.

The concept of private data in JS isn't built-in to JS. It is something that you can only accomplish by using other features, such as "closure" and "immediately invoked function expressions".

Unless you employ one of the messy data-hiding techniques, all of your implementation details are public. You can't prevent other developers from using the private parts of your implementation.

Nevertheless, we'll treat methods as one way to hide implementaiton details. They don't hide things particularly well in JS but instead rely on trust rather than concealing and preventing access.

When a class has methods that provide the behaviours and actions you need, you should use those methods instead of accessing the properties directly.

The entire goal of creating a class and encapsulating logic in it is to hide implementation details and contain ripple effects when things change. Keep in mind that JS doesn't implement encapsulation in a way that directly supports private data and methods. Use the provided interface -- the methods -- instead whenever possible.

Lesson 2: Advanced Concepts

The **var** Statement

Thus far, we've used the **let** and **const** statements to declare variables. These statements are simple and relatively easy to understand and use; they declare and initialize variables and constants, respectively, and create variables with block scope, which is easy to understand.

The **var** statement provides no way to create constants, so that's one significant disadvantage to using **var**.

Using **var** at the top level of a program creates a property on the global object. This behaviour shows that **let** is safer than **var** when used at the program's top level. Placing properties on the global object may lead to conflicts and bugs; **let** alleviates that issue.

When you use **var** inside a function, the variable is **not** stored as a property of the global object.

Scope and **var**

A much more significant difference is that **let** is **block-scoped**, while **var** is **function-scoped**.

A block-scoped variable is only visible within the block where it is declared; in JS, a block is code delimited by curly braces. The term block usually refers to executable code between braces, including function bodies.

A function-scoped variable is visible within the function where it is declared.

When you run a JS program file with Node, Node "wraps" your code in a function that looks like this:

```
(function(exports, require, module, __filename, __dirname) {
	// your code
});
```

The wrapper function, or its absence, is the source of most problems involving differing behaviour in the REPL and a program file.

More About Scope

Declared Scope vs. Visibility Scope vs. Lexical Scope

One way to help keep things straight is to look at scope as having three subtly different but related meanings.

In one sense, scope refers to where a particular identifier -- a variable, function, or class name -- is available for use by your code. We can call this the **visibility scope**. If a variable is available throughout your code, then it has global scope. Otherwise, it has local scope.

In another sense, scope refers to how a particular identifier is declared. We'll call this the **declared scope**. For instance, we use the **let** keyword to declare variables with block scope, and use **var** to declare variables with function scope. Knowing the declared scope lets us determine where a variable is available.

Finally, scope can refer to the lexical structure of your code. We'll call this the **lexical scope**. The lexical scope distinguishes between variables that are declared inside a function or block and the variables that are declared outside of that function or block. Lexical scope is especially important with closure.

Though the visibility, declared, and lexical scopes have different meanings, there is considerable overlap.

Declared Scope

Declared scope concerns how a variable is declared: **let**, **const**, **class**, **var**, or **function**. The first three declare variables with **block scope** while the other two declare variables with **function scope**.

Even if the variable is declared outside of a function or block, it has either block or function scope:

```
let foo1 = 1; // declared scope is block scope
var bar1 = 2; // declared scope is function scope

if (true) {
	let foo2 = 3; // declared scope is block scope
	var bar2 = 4; // declared scope is function scope
}

function xyzzy() { // declared scope is function scope
	let foo3 = 5; // declared scope is block scope
	var bar3 = 6; // declared scope is function scope

	if (true) {
		let foo4 = 7; // declared scope is block scope
		var bar4 = 8; // declared scope is function scope
	}
}
```

Visibility Scope

Visibility scope concerns where a variable is visible. This can be either **global scope** or **local scope** (i.e., inside a function or block).

We will sometimes also talk about **local function scope** and **local block scope** when discussing the local visibility scope. However, we will often omit the word "local".

```
let foo1 = 1; // visibility scope is global
var bar1 = 2; // visibility scope is global

if (true) {
	let foo2 = 3; // visibility scope is local (local block)
	var bar2 = 4; // visibility scope is global
}

function xyzzy() { // visibility scope is global
	let foo3 = 5; // visibility scope is local (local function)
	var bar3 = 6; // visibility scope is local (local function)

	if (true) {
		let foo4 = 7; // visibility scope is local (local block)
		var bar4 = 8; // visibility scope is local (local function)
	}
}
```

The visibility scope is determined as a combination of how each variable is declared and on the lexical location of each declaration.

Lexical Scope

Lexical scope concerns how the structure of your code determines what variables are accessible or inaccessible at any point in the program. Lexical scope includes both **inner scope** and **outer scope**.

We'll start with some code that uses **let**:

```
let foo1 = 1; // outer scope of xyzzy, outer scope of if block

if (true) {
	let foo2 = 3; // inner scope of if block
}

function xyzzy() {
	let foo3 = 5; // inner scope of xyzzy, outer scope of if block inside the function

	if (true) {
		let foo4 = 7; // inner scope of if block inside the function
	}
}
```

Let's see what happens when we use **var**:

```
var bar1 = 1; // outer scope of xyzzy, outer scope of if block

if (true) {
	var bar2 = 3; // outer scope of xyzzy, outer scope of if block
}

function xyzzy() {
	var bar3 = 5; // inner scope of xyzzy, outer scope of if block inside the function

	if (true) {
		var bar4 = 7; // inner scope of xyzzy, outer scope of if block inside the function
	}
}
```

We usually talk about lexical scope when we want to talk about what variables can be accessed from any specific place in the program.

How to Talk About Scope

It's usually a matter of understanding your frame of reference:

- Use declared scope when you're talking about how an identifier is declared.
- Use visibility scope when you're talking about the visibility of a specific identifier.
- Use lexical scope when you want to talk about whether something is "in scope" -- that is, whether it is available for use.

Hoisting

The behaviour that arises from hoisting can be confusing, especially when **var** declarations are present.

What is Hoisting?

JS engines operate in two main phases: a **creation phase** and an **execution phase**. The creation phase is sometimes erroneously called the compilation phase.

The execution phase occurs when the program runs code line-by-line.

The creation phase does some preliminary work. One of those work items is to find all of the variable, function, and class **declarations**. When it encounters each of these identifiers, it records the name and designates its scope.

When the execution phase begins, JS knows what variables exist and where they are in scope. From the developer's perspective, the code acts like the declarations were moved to the top of their respective scope. In particular, function-scoped declarations are moved to the function's beginning, and block-scoped declarations are moved to the block's start. We call this process **hoisting**.

The effect of hoisting is that all the declarations get hoisted -- raised, lifted, moved -- to the top of their defined scope.

It's important to realize that **hoisting doesn't change the program**. It merely executes the program in a manner that makes it seem like the code was rearranged.

The Temporal Dead Zone

Variables declared with the **let**, **const**, and **var** statements are also hoisted.

When a **var** variable is hoisted, JS gives it an initial value of **undefined**. If you try to access the value assigned to a **var** variable before the original statement with the **var** declaration gets executed, JS returns **undefined**.

When **let** and **const** variables are hoisted, they are not given an initial value at all. Instead, they are left in an "unset" state; that is, they are "not defined". Such unset variables are said to be in the **Temporal Dead Zone**, or the **TDZ**. Such variables remain in the TDZ until the initialization code runs during the execution phases.

Hoisting for Function Declarations

JS also hoists function declarations to the top of the scope. In fact, it hoists the entire function declaration, including the function body.

Function declarations have function scope. That's another way of saying that hoisting also occurs with nested functions.

You shouldn't try to nest function declarations inside non-function blocks. If you must nest a function inside a block, use a function expression.

Hoisting for Function Expressions

Function expressions often involve assigning a function to a declared variable. Those variables obey the usual hoisting rules for variable declarations.

Hoisting for Classes

When JS encounters a class declaration, the class name gets hoisted, but the definition of the class does not. Much like **let** and **const** variables, **class** declarations live in the TDZ until their definition code is executed.

Hoisting for class expressions is similar: the variable name gets hoisted, but the definition doesn't get assigned to the name until the expression is evaluated.

Hoisting Variable and Function Declarations

What happens when a **var** variable and a function declaration have the same name? In that case, the function declaration gets hoisted to the top of the program and the variable declaration gets discarded.

Best Practice to Avoid Confusion

If you follow a few simple rules, you'll avoid many headaches:

- Whenever possible, use **let** and **const** instead of **var**: avoid the confusion and subtle behaviours that can occur with **var**.
- If you must use **var**, declare all of the variables at the top of the scope.
- If you can use **let** and **const**, delcare them as close to their first usage as possible.
- Declare functions before calling them.

Hoisting Isn't Real

Hoisting is really just a mental model that almost all JS developers use to explain how scope works. There is no actual hoisting process in JS.

There are edge cases for which hoisting doesn't provide a satisfactory explanation for how JS works.

Clearly, hoisting is something of an approximation for what really happens.

The behaviour that we try to explain with hoisting is merely a consequence of JS's two phases: the creation and execution phases. The creation phase finds all of the identifiers in your code and determines their scope at that time.

When the execution phase occurs, JS no longer cares about declarations. It does care about initialization and function/class definitions, but not the declarations themselves. The identifiers are already known, and their scope is already known. JS merely needs to look up the identifiers as required.

Recall that you can't have two declarations with the same name if one of those names is declared by **let**.

Syntax errors usually occur during the creation phase -- before "hoisting" affects the code.

Processing occurs from the top down during the creation phase.

Practice Problems: Hoisting and the var Statement

#3:

If we want to call a function before its body is defined, we need to use a function declaration.

#6:

Class hoisting:

let Image;

Image = class {};

We have to create a variable for the class name, then later assign it a class expression.

Strict Mode

Using strict mode will help you find and fix errors before they become mysterious bugs that plague your application for years.

Strict mode is an optional mode that modifies the semantics of JS and prevents certain kinds of errors and syntax.

What Does Strict Mode Do?

Strict mode makes three significant changes to JS semantics:

- Strict mode eliminates some **silent errors** that occur in sloppy mode by changing them to throw errors instead. Silent errors occur when a program does something that is unintended, but continues to run as though nothing is wrong. This can lead to incorrect results or errors much later in execution that are subsequently difficult to track down.

- Strict mode prevents some code that can inhibit JS's ability to optimize a program so that it runs faster.

- Strict mode prohibits using names and syntax that may conflict with future versions of JS.

These changes offer several benefits to JS developers:

- They prevent or mitigate bugs.
- They help make debugging easier.
- They help your code run faster.
- They help you avoid conflicts with future changes to the language.

Enabling Strict Mode

Strict mode is easy to turn on either at the global level of a program or at the individual function level. To enable it, add **"use strict";** to the beginning of the program file or function definition.

Nested functions inherit strict mode from the surrounding scope.

The **"use strict"** statement is an example of a **pragma**, a language construct that tells a compiler, interpreter, or other translator to process the code in a different way. Pragmas aren't part of the language, and often use odd syntax like **"use strict"** does.

You can't enable strict mode partway through a program or function.

You can not enable strict mode for a block. You can only enable it at the very beginning of a file or function.

Once you enable it, **you can't disable it later** in the same program or function.

JS enables strict mode automatically within the body of a **class**; there is no way to prevent that behaviour. The same thing happens with JS modules.

Strict mode is lexically scoped; that is, it only applies to the code that enables it.

Implicit Global Variables

No matter where your code initializes an undeclared variable, it becomes a global variable.

Strict mode disables this feature by not letting you create variables without explicitly declaring them.

This behaviour also helps identify misspelled names. If you declare a variable with one name, then later try to reassign it with a misspelled name, sloppy mode will create a new global variable.

Implicit Context In Functions

In strict mode, using function call syntax on a method sets **this** to **undefined**.

This change to JS's semantics may be the most significant change of all under strict mode. It probably won't break your code, but it should help you spot bugs caused by context loss much sooner.

Forgetting to Use **this**

Strict mode will help in situations where we forget to use **this** when performing operations on objects.

Leading Zeros

If you use a literal integer that begins with **0** but doesn't contain the digits **8** or **9**, sloppy mode JS interprets it as an octal number.

With strict mode, numbers that look like octal numbers raise an error.

Note that strict mode also prevents any number literal from beginning with **0** or **-0** except for **0** itself or **0** with a decimal component (i.e., **0.123**).

Other Strict Mode Differences

In addition to the changes described above, strict mode:

- prevents you from using function declarations in block;
- prevents declaring two properties with the same name in an object; } the first two are now allowed but are still considered bad practice
- prevents declaring two function parameters with the same name;
- prevents using some newer reserved keywords, such as **let** and **static**, as variable names;
- prevents you from using the **delete** operator on a variable name;
- forbids binding of **eval** and **arguments** in any way;
- disable access to some properties of the **arguments** object in functions; and
- disables the **with** statement, a statement whose use is not recommended even in sloppy mode.

When Should I Use Strict Mode?

Use strict mode in any new code that you write.

If you're adding new functions to an old codebase, it's safe to use function-level strict mode in the new functions, and you probably should do so.

However, if you're not creating a new function in that old database, you probably shouldn't try to use strict mode as this could easily break code that otherwise works well.

Summary

Recap of the semantic changes that occur when using strict mode:

- You cannot create global variables implicitly;
- Functions won't use the global object as their implicit execution context;
- Forgetting to use **this** in a method raises an error; and
- Leading zeros on numeric integers are illegal.

Strict mode makes other changes as well, but the above changes are the most important for most JS developers.

Closures

Closures let a function access a variable that was in lexical scope at the function's definition point even when that variable is no longer in scope.

You may not realize it, but you've been using closures every time you've defined a function that accesses a variable from its outer scope.

Closures

Closures and lexical scope are intimately related.

Closures use the lexical scope in effect at a function's definition point to determine what variables that function can access. What variables are in scope during a function's execution depend on the closure formed by the function's definition.

MDN defines **closure** as "the combination of a function and the lexical environment within which that function was defined".

You can think of closure as a function combined with any variables from its lexical scope that the function needs. In other words, if a function uses a variable that is not declared or initialized in that function, then that variable will be part of the closure (provided it exists).

Closures are created when you define a function or method. The closure essentially closes over its environment -- what's in lexical scope.

In effect, the function definition and all the identifiers in its lexical scope become a single entity called a closure. When the function is invoked, it can access any variables it needs from that environment. That is, the function can use variables from the lexical scope where the function was defined.

Even if those variables aren't in the lexical scope where you invoke the function, it can still access them.

Note that closures only close over the variables that the function needs.

Closure and scope are lexical concepts. Where you invoke a function is unimportant; where you define the function is.

A closure includes the variables it needs from the scope where you defined the function. Those variables may not be in scope when you invoke the function, but they're still available to the function.

A Helpful Mental Model

When you define a function, JS finds all of the variable names it needs from the lexical scope that contains the function definition. It then takes those names and places them inside a special "envelope" object that it attaches to the function object. Each name in the envelope is a pointer to the original variable, not the value it contains.

When a function encounters a variable name during execution, it first looks inside its local scope for that name. If it can't find the name, it peeks inside the envelope to see whether the variable is mentioned there. If it is, JS can follow the pointer and get the current value of the variable.

In fact, this is how scope works in JS: it first checks for local variables by a given name, then it looks to the closure if it can't find it. All that stuff about looking at outer scopes until you reach the global scope all happens when the closure is defined.

Only variables that are in scope when you define the function are available to the function.

Example of Closure

Higher-order function example:

```
function foo() {
    let name = 'Pete';
    return function() {
        console.log(name);
    };
}

let printPete = foo();
printPete(); // Pete
```

At a minimum, the closure formed by the returned function's definition contains a pointer to **name** in its envelope. That pointer means that **name**'s value won't get discarded when **foo** is done.

Though **name** is out of scope when **foo** finishes, the returned function has an envelope that contains a pointer to **name**. Thus, the function can still follow the pointer to the original variable, and find its current value, and that lets **printPete()** print 'Pete'.

Functions that return functions are perhaps the most powerful feature of closure in JS.

A closure is not a snapshot of the program state. Thus, if a variable's value changes, the closure ensures that the function sees the new value, not the old one.

Another example:

```
function makeCounter() {
    let counter = 0;

    return function() {
        counter += 1;
        return counter;
    };
}

let incrementCounter = makeCounter();
console.log(incrementCounter()); // 1
console.log(incrementCounter()); // 2
```

Note that **counter** is now a private variable in the sense that we can not access it directly. This form of data protection is a big reason why returning a function from another function is so powerful.

It's important to remember that closure definitions are purely lexical. Closures are based on your program's structure, not by what happens when you execute it. Even if you never call a particular function, that function forms a closure with its surrounding scope.

Partial Function Application

Consider the following code:

```
function add(first, second) {
	return first + second;
}

function makeAdder(firstNumber) {
	return function(secondNumber) {
		return add(firstNumber, secondNumber);
	};
}

let addFive = makeAdder(5);
let addTen = makeAdder(10);
```

A function such as **makeAdder** is said to use **partial function application**. It applies some of the function's arguments (the **add** function's **first** argument here) when called, and applies the remaining arguments when you call the returned function.

Partial function application refers to the creation of a function that can call a second function with fewer arguments than the second function expects. The created function applies the remaining arguments.

Partial function application is most useful when you need to pass a function to another function that won't call the passed function with enough arguments. It lets you create a function that fills in the gaps by applying the missing elements.

For instance, suppose you have a function that downloads an arbitrary file from the Internet. The download may fail, so the function also expects a callback function that it can call when an error occurs:

```
function download(locationOfFile, errorHandler) {
	// try to download the file
	if (gotError) {
		errorHandler(reasonCode);
	}
}

function errorDetected(url, reason) {
	// handle the error
}

download("https://example.com/foo.txt", …);
```

Our error handling function, **errorDetected**, takes two arguments, but **download** only passes one argument to the error handler. Suppose the **download** function is part of a 3rd party library that you can't modify. You can turn to partial function application to get around the single-argument limitation:

```
function makeErrorHandlerFor(locationOfFile) {
	return function(reason) {
		errorDetected(locationOfFile, reason);
	};
}

let url = "https://example.com/foo.txt";
download(url, makeErrorHandlerFor(url));
```

The **download** method now calls the partially applied function returned by **makeErrorHandlerFor**, and **errorDetected** gets both argument it needs.

Rather than creating a **makeErrorHandlerFor** function, you can use **bind** to perform partial function application. In most cases, **bind** is all you need (i.e., **download(url, errorDetected.bind(null, url))**).

Partial function application requires a reduction in the **number of arguments** you have to provide when you call a function. If the number of arguments isn't reduced, it isn't partial function application.

What are Closures Good For?

Here are some other things made possible by closures:

- Currying (a special form of partial function application);
- Emulating private methods;
- Creating functions that can only be executed once;
- Memoization (avoiding repetitive resource-intensive operations);
- Iterators and generators;
- The module pattern (putting code and data into modules); and
- Asynchronous operations.

Closures and Private Data

We'll focus on how we can use closures to define private data in JS objects.

Private Data

Functions combine with the environment at their definition point to form closures.

A closure lets a function access its definition environment regardless of when and where the program invokes the function.

Why Do We Need Private Data?

Using closures to restrict data access is an excellent way to force other developers to use the intended interface. By keeping the collections private, we enforce access via the provided methods.

Restricting access helps protect data integrity since developers must use the interface.

Private data also helps prevent the user of an object from becoming dependent on the implementation. Other developers shouldn't care that a todo-list object stores todos in an array, and your code shouldn't encourage them to depend on it.

Instead, your object should supply an API that lets other developers manipulate the todo-list regardless of its implementation. If you later change the implementation, your API should remain the same.

Be careful though. Even when you restrict access, it's easy to expose data by returning references to that data.

You shouldn't rely on private data to keep sensitive information hidden. Encryption is the only reasonable safe way to protect such data.

Modern debuggers, such as the ones built-in to your browser, let you step through a program and inspect data at every step, and that makes even private data visible. Program errors can also expose private data, quite by accident.

Immediately Invoked Function Expressions

These expressions let you define and execute a function in a single step. They let you use variable and function names that won't conflict with other names in your code.

What Are IIFEs and How Do You Use Them?

An **IIFE** or **immediately invoked function expression** is a function that you define and invoke simultaneously. In JS, we use a special syntax to create and execute an IIFE:

```
(function() {
	console.log('hello');
})();
```

You should always use the surrounding parentheses to more clearly show that you're running an IIFE.

Parentheses

Parentheses act as a grouping control; they control the order of evaluation in an expression.

With IIFEs, the parentheses around the function definition tell JS that we first want to evaluate the function as an expression. We then take the return value of the expression and use function call parentheses to invoke the function.

All functions, including IIFEs, can take arguments and return values:

```
(function(number) {
	return number + 1;
})(2);
```

IIFE Style Issues

You may sometimes see a slightly different style for IIFEs:

```
(function() {
	console.log('hello');
}());
```

Here, the argument list is inside the outer set of parentheses. As with the original style, they let JS distinguish between an ordinary function declaration and an IIFE.

JS handles this style the same as with the earlier approach. However, the style makes what is happening less apparent. You should use the style that we showed ealier.

We can omit the parentheses around an IIFE when the function definition is an expression that doesn't occur at the beginning of a line.

```
let foo = function() {
	return function() {
		return 10;
	}() + 5;
}();

console.log(foo); // => 15
```

We should use parentheses here as well:

```
let foo = (function() {
	return (function() {
		return 10;
	})() + 5;
})();
```

Using IIFEs With Arrow Functions

IIFEs work with all kinds of functions, including arrow functions:

```
((first, second) => first * second)(5, 6); // => 30
```

Creating Private Scopes With IIFEs

IIFEs are a great way to add some code to a program without having to worry about conflicting variable names and other potential issues.

Using Blocks For Private Scope

In ES6 JS and later, you can use blocks to create private scopes. Although blocks are a more straightforward way to create private scopes, existing code often relies on IIFEs to achieve that result.

Using IIFEs to Define Private Data

IIFEs also let us create functions with private data. Of course, we can already do that without IIFEs. In some cases, though, IIFEs lead to simpler code that is more convenient to use.

When we talk about private scope, we're talking about how you can use scope to prevent some code from making accidental changes to variables in its outer scope.

When we discuss private data, we're talking about encapsulation: making local data inaccessible from the code's outer scope.

Practice Problems: IIFE

#1:

A function declaration must be converted to a function expression before you can invoke it with immediate invocation syntax.

#5:

The named function inside an IIFE isn't visible outside of the scope created by the IIFE.

#7:

Bear in mind that named functions created as IIFEs can be referenced by those same functions. That is, you can call a function's name in the body of the IIFE.

Shorthand Notation

ES6 introduced several shorthand notations that are very handy when working with objects and arrays.

Concise Property Initializers

```
function xyzzy(foo, bar, qux) {
	return {
		foo,
		bar,
		qux,
	};
}
```

Concise Methods

```
let obj = {
	foo() {},
	bar(arg1, arg2) {},
};
```

Object Destructuring

One of the most useful new features in ES6 is **destructuring**, a shorthand syntax that lets you perform multiple assignments in a single expression.

```
let obj = {
	foo: ‘foo’,
	bar: ‘bar’,
	qux: 42,
};

let { qux, foo, bar } = obj;
```

The result from this code is: **qux** is assigned **obj.qux**, **foo** is assigned **obj.foo**, and **bar** is assigned **obj.bar**.

We can even use different names for the result (i.e., **let { qux: myQux, foo, bar } = obj**). This example creates a **myQux** variable that receives the value of **obj.qux**.

Destructuring also works with function parameters:

```
function xyzzy({ foo, bar, qux }) {}

xyzzy(obj);
```

In this code, we pass an object to the function. The function's definition uses destructuring to pull out the needed properties and store them in local variables.

If you need to use destructuring elsewhere -- an assignment, for instance -- you can do so. However, you may need to enclose the expression in parentheses (i.e., **({ foo, bar, qux } = obj)**).

Array Destructuring

```
let foo = [1,2,3];
let [ first, second, third ] = foo;
```

If you don't need all of the elements, you can skip them:

```
let bar = [1,2,3,4,5,6,7];
let [ first, , , fourth, fifth, , seventh ] = bar;
```

Finally, you can use rest syntax in array destructuring to assign a variable to the rest of an array:

```
let foo = [1,2,3,4];
let [ bar, ...qux ] = foo;
console.log(bar); // 1
console.log(qux); // [2,3,4]
```

Spread Syntax

The **spread syntax** uses **...** to "spread" the elements of an array or object into separate items.

Here are common use-cases for the spread syntax:

- Create a clone of an array

  ```
  let foo = [1,2,3];
  let bar = [...foo];
  console.log(bar); // [1,2,3]
  console.log(foo === bar); // false -- bar is a new array
  ```

- Concatenate arrays

  ```
  let foo = [1,2,3];
  let bar = [4,5,6];
  let qux = [...foo, ...bar];
  qux; // => [1,2,3,4,5,6]
  ```

The spread syntax can be used for expanding an array into a number of arguments.

- Insert an array into another array

  ```
  let foo = [1,2,3];
  let bar = [...foo, 4, 5, 6, ...foo];
  bar; // => [1,2,3,4,5,6,1,2,3]
  ```

Spread syntax also works with objects:

- Create a clone of an object

```
let foo = { qux: 1, baz: 2 };
let bar = { ...foo };
console.log(bar); // { qux: 1, baz: 2 }
console.log(foo === bar); // false -- bar is a new object
```

- Merge objects

```
let foo = { qux: 1, baz: 2 };
let xyz = { baz: 3, sup: 4 };
let obj = { ...foo, ...xyz };
obj; // => { qux: 1, baz: 3, sup: 4 }
```

Using the spread syntax can help eliminate some messy looking code, and can also eliminate the need to use **apply**. The result is easier to type and read.

Note that spread syntax with objects only returns the properties that **Object.keys** would return. That is, it only returns enumerable "own" properties.

That means, in part, that it's not the right choice when you need to duplicate objects that inherit from some other object. It also means that you lose the object prototype.

Rest Syntax

In some ways, you can think of **rest syntax** as the opposite of spread syntax. Instead of spreading an array or object out into separate items, it instead collects multiple items into an array or object.

We saw an example of this in the section on destructuring arrays.

You can also use rest syntax with objects.

Note that the rest element must be the last item in any expression that uses rest syntax.

Rest syntax is used most often when working with functions that take an arbitrary number of parameters.

```
function maxItem(first, ...moreArgs) {
    let maximum = first;
    moreArgs.forEach(value => {
        if (value > maximum) {
            maximum = value;
        }
    });
    return maximum;
}

console.log(maxItem(2,6,10,4,-3));
```

The rest parameter syntax allows us to capture an indefinite number of arguments in an array.

Modules

We'll focus on one of the two module systems: CommonJS modules, also knwon as Node modules. We'll also touch on JS modules, also known as "ES modules" and "ECMAScript modules".

Benefits of Modules

As programs grow in size and complexity, they become harder to understand. If you could somehow split the program up into self-contained pieces, it becomes much easier to understand each component.

Large single-file programs also tend to lose cohesion. That is, the different parts of the program tend to become entangled as you make more and more changes to the code.

If you make a change someplace, there's a ripple effect that often occurs with older, larger programs.

Another issue that arises with large programs is encapsulation. We've seen several ways to define private data in objects and functions. However, the additional work needed to encapsulate your data and methods is often too much bother. It can also lead to messy code.

The solution to all these problems is to split up a program into multiple files, commonly called **modules**.

Each module can focus on a particular part of the problem, and a team of developers can work on separate parts without fear of conflicts.

With well-defined modules, you probably don't need to chase down all the function calls.

Furthermore, modules help keep a program's concerns separated; the different parts don't become entangled in the face of many enhancements and bug fixes.

Modules can also make it easier to work with private data, which helps maintain encapsulation. You must explicitly export the items you want to make available; everything else is private to the module.

Best of all, you can easily use a module elsewhere without first having to disentangle it from the program.

CommonJS Modules

From its earliest days, Node has supported modules. It's this capability that lets us **require** (or **import**) some code into a program.

To use a CommonJS module, all you have to do is import it with the **require** function. You may have to install the module first with the Node Package Manager, NPM, but once it's installed, you only need to **require** it.

Synchronous loading makes CommonJS modules unsuitable for the browser environment -- it takes too long to load them. They're suitable for Node applications where everything resides on the same machine, but not the browser.

Browsers don't natively support CommonJS modules, but you can use a transpiler like **Babel** to transpile code that uses CommonJS modules into a format that can be used by browsers.

Creating CommonJS Modules

First, create a file with the code that you want to modularize, then add some additional code to export the items that you want other modules to use:

```
function logIt(string) {
        console.log(string);
}

module.exports = logIt;
```

Once you have the module file, you can import it into your main program (or another module). The names that it exports then become available to your program:

```
const logIt = require("./logit");
logIt("You rock!");
```

When importing modules that weren't installed by NPM, you need to specify the path to the file that contains the module.

Typically, such paths start with **./**, which tells Node to look inside your project folder. If you installed a package with NPM, you can usually require it without the **./** or any other qualification.

You can export any values you want from a module: functions, constants, variables, and classes are all eligible for export.

You can export multiple items at once:

```
let prefix = “>> ”;

function logIt(string) {
        console.log(`${prefix}${string}`);
}

function setPrefix(newPrefix) {
        prefix = newPrefix;
}

module.exports = {
        logIt,
        setPrefix,
};

const { logIt, setPrefix } = require(“./logit”);
```

CommonJS Variables

In Node, all code is part of a CommonJS module, including your main progam. Each module provides several variables that you can use in your code:

- **module**: an object that represents the current module;
- **exports**: the name(s) exported by the module (same as **module.exports**);
- **require(moduleName)**: the function that loads a module;
- **__dirname**: the absolute pathname of the directory that contains the module; and
- **__filename**: the absolute pathname of the file that contains the module.

JS Modules

Built-in support for JS modules in your browser is a relatively new feature.

With ES6, JS now supports modules natively. It adds the **export** and **import** keywords to the language, and most modern browsers now support them.

If you must support older browsers, you can use tools like **Babel** and **Webpack**.

Babel transpiles (converts) ES6 code to ES5 code. Webpack consolidates all of the modules you need into a single file.

Using JS Modules

If your browser supports JS modules, using them is not difficult. For instance, your module file might look like this:

- **foo.js**

```
import { bar } from “./bar”;

let xyz = 1;

export function foo() {
        console.log(xyz);
        xyz += 1;
        bar();
}
```

- **bar.js**

```
export let items = [];
export let counter = 0;
export function bar() {
        counter += 1;
        items.push(`item ${counter}`);
}

export function getCounter() {
        return counter;
}
```

- **main.js**

```
import { foo } from "./foo";
import { bar, getCounter, items, counter } from "./bar";
```

Notice that exporting is as simple as preceding each declaration with the word **export**.

Anything that you don't export explicitly is local to the module. Thus, the **xyz** variable in the **foo** modules is local to **foo**.

Note that, by default, Node does not support ES6 modules directly. However, recent versions of Node do provide ways to use ES modules by either renaming the modules or by updating **package.json**.

If you're using an older version of Node, you can use something like Babel to transpile ES6 modules into a form that Node understands.

Exceptions

ReferenceError, **TypeError**, and **SyntaxError** are the names of **exceptions**. They are objects that inherit from **Error**.

What Are Exceptions

When JS encounters an error that it cannot recover from, it issues an error message and usually terminates the program. Developers talk about such errors by saying that the program **threw an error** or **raised an exception**.

By default, exceptions terminate the program, but a program can catch and handle exceptions with an **exception handler**.

JS implements exception handlers in the form of **try/catch** statements. The **try** block runs the code that might raise an exception, while the **catch** block tries to do something in response.

Not all errors are exceptions. For instance, your program may display an error message when the user enters some invalid information. However, it probably won't throw an exception when it does. Instead, it will validate the inputs to see whether there is a problem. If there is, it will display an error message and ask the user to try again.

In JS, a typical exception is an object of the **Error** type or an object that inherits from the **Error** type.

Throwing Exceptions

You can also raise exceptions yourself. For instance, you may have noticed that JS doesn't treat division by zero as an error. We can do the following:

```
function div(first, second) {
        if (second === 0) {
                throw new Error('Divide by zero!');
        }
        return first / second;
}

let result = div(1,0); // Error: Divide by zero!
console.log(result); // not run
```

Most JS code throws a **new Error** to raise an exception rather than one of the 7 built-in types that inherit from **Error**. However, you can throw any type you want, including the built-in error types, primitive values, and other objects.

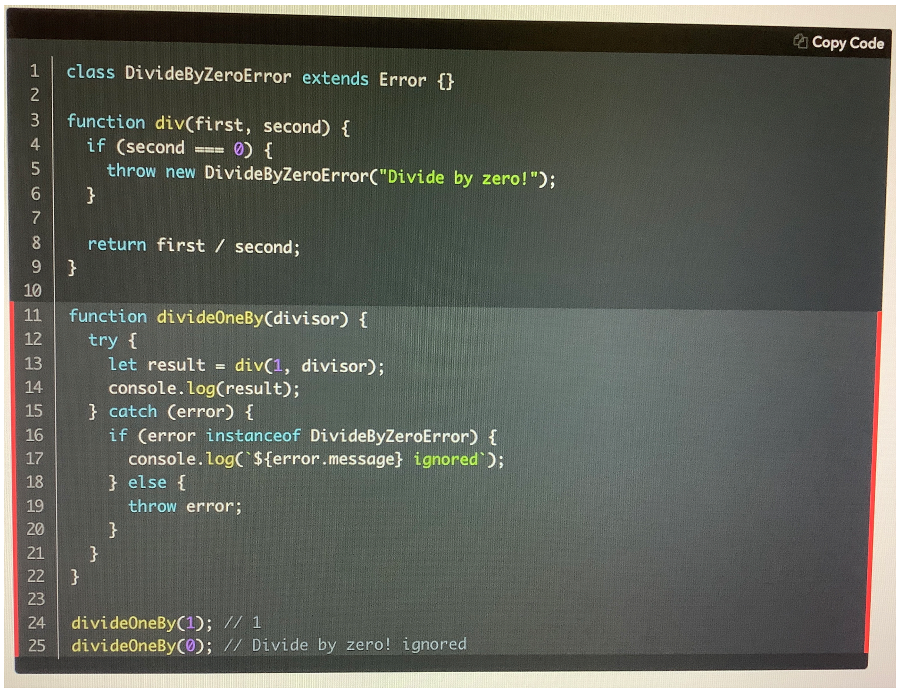
If you want, you can even define a custom error type that inherits from **Error**:

```
class DivideByZeroError extends Error {}

function div(first, second) {
        if (second === 0) {
                throw new DivideByZeroError('Divide by zero!');
        }
        return first / second;
}

let result = div(1,0); // DivideByZeroError: Divide by zero!
console.log(result); // not run
```

This can be useful when you need to catch an exception.

Catching Exceptions

It's possible to **catch an exception**; that is, you can detect when an exception occurs and attempt to recover from the error.



Copy Code

```
class DivideByZeroError extends Error {}

function div(first, second) {
  if (second === 0) {
    throw new DivideByZeroError("Divide by zero!");
  }

  return first / second;
}

function divideOneBy(divisor) {
  try {
    let result = div(1, divisor);
    console.log(result);
  } catch (error) {
    if (error instanceof DivideByZeroError) {
      console.log(`${error.message} ignored`);
    } else {
      throw error;
    }
  }
}

divideOneBy(1); // 1
divideOneBy(0); // Divide by zero! ignored
```

When an exception is raised within the **try** block, it causes the **try** block to terminate. However, since we have a **catch** block, it does not terminate the program. Instead, it begins executing the **catch** block.

The **error** variable is set to reference the exception object thrown in the **try** block.

Note that the **catch** block doesn't know whether any other errors can occur in the **try** block. Thus, it's good practice to check for the specific exceptions that you want to handle. After all, you don't want to catch and ignore an **InternalError**, for instance. There's no telling what might happen if you do.

What Happens When A Program Throws An Exception

When a JS program throws an exception, the exception bubbles up through the program looking for a **try** block that contains the code that eventually led to the exception.

If it never finds a **try** block, JS issues an appropriate error message, then terminates the program.

If the exception does encounter a **try** block, it then exits from the **try** block without executing any remaining code and executes the **catch** block. If the **catch** block re-throws the exception, the bubbling process resumes as before, again looking for another **try** block.

If a re-throw does not occur in the **catch** block and the **catch** block does not throw an exception of its own, the program discards the exception object and resumes execution with the code after the **try/catch** statement.

If the **catch** block does raise a new exception, the program still discards the original exception object. However, the new exception object begins to bubble up through the program.

Every exception terminates the program unless a **catch** block handles it without re-throwing it.

When Should I Use Exceptions?

Exceptions should be used to handle exceptional conditions, not normal and expected conditions. Another way to say that is that you should not use exceptions for flow control.

If there's a better way to handle the error, use it instead of raising an exception.

You should throw an exception when an event occurs that should not be ignored or when the condition is truly anomalous or exceptional.

You should not throw an exception if you can handle the problem in your local code, or if it's a normal or expected part of the control flow.

Exceptions are sometimes likened to the "ask forgiveness" approach to error handling. Rather than asking whether we can do something, we just go ahead and try it. If an exception does occur, then we have to ask forgiveness -- handle the exception or just accept the cost of an error.

In contrast to the "ask forgiveness" approach, the "get permission" approach involves determining beforehand whether something is going to work. For instance, if we expect a user to input a number, we can test wheter they really did enter a number before we try to use it as a number. This is often harder to do than allowing an exception to be raised, but once permission is granted, we know things will work.

In general, an exception handler in a **catch** block should do as little as possible. For instance, you can:

- Ignore the exception;
- Return an error value (i.e., **undefined**, **null**) to the calling function;
- Set a flag that the program can test after the handler finishes running;
- Log a simple error message; and
- Throw another exception with an explicit **throw** statement.

The final item on that list implies that you should not do anything in your **catch** block that might unexpectedly raise an exception. You can use the **throw** statement to throw an explicit exception, but be careful about calling some other function that can raise an exception.

Potentially having to deal with a secondary exception is unwise since we're still trying to handle the original exception. If we haven't finished recovering from it, we never will if another exception occurs.

Garbage Collection

Every value in JS requires a certain amount of memory. The precise amount varies based on its type and the specific data stored in each value.

As developers, we don't usually have to worry about the wasted space allocated to unused values. Instead, JS uses a process called **garbage collection** (**GC**) to free up the memory. Garbage collection happens automatically and is usually completely invisible to the developer.

Note that we don't need code to claim and release memory; the JS runtime handles that for us. It gets memory from the system when we create new values and releases it when the program no longer needs them.

How Does Garbage Collection Work?

How does JS know when a given value is no longer needed? The details are implementation-specific.

One of the simplest and naive implementation uses a ‘reference count’ to track how many references to the value exist in the program. When the reference gets removed, it decrements the reference count. When the reference count drops to 0, the value becomes eligible for GC.

The Stack and Heap

Most programming languages divide memory into two principal regions: the stack and the heap.

JS stores most primitive values as well as references on the stack, and everything else on the heap. You can think of references as pointers to the actual value of an object, array, or string that lives in the heap.

The stack doesn’t participate in garbage collection. That means that **primitve values don’t get involved in garbage collection when they are stored on the stack**.

When a function or block begins executing in a JS program, JS allocates memory on the stack for the variables defined in that block or function. Since each item has fixed size, JS can calculate the amount of memory it needs during the creation phase of execution without knowing the specific values. That means it can determine how much stack space it needs when hoisting occurs.

When the block or function is done running, the allocated stack memory gets returned to the system automatically. This process is somewhat similar to garbage collection, but it is considered distinct.

Some primitive values can’t be stored on the stack. Stack values typically must have a fixed size. Values that don’t fit in that size must be stored elsewhere, typically the heap. Strings and BigInts, for instance, usually can’t be stored in 64 bits, so they get placed in the heap or somewhere else.

The heap is much trickier to deal with since each value has a different size that can’t be determined ahead of time. Instead, new values must be added to the heap when they get created. It needs to rely on garbage collection to detect when a value’s reference count reaches 0.

Garbage collection can occur at any time; it often occurs at periodic intervals during a program’s lifetime. In particular, the programmer usually has no control over when GC occurs.

All the garbage collector must do is look for and deallocate values that are eligible for garbage collection. If it uses a reference counting system, it needs to look for values with a reference count of 0.

Note that GC doesn’t happen when a variable goes out of scope. That’s a common misconception. A variable can go out of scope, but there can be many other references. Closures, arrays, and objects are a significant source of persistent references.

Why Do You Need To Know About Garbage Collection

Modern JS engines use what is called a mark and sweep algorithm to do GC.

Some Light, Optional Reading

Regardless of the programming language, the memory life cycle is pretty much always the same:

- Allocate the memory you need;
- Use the allocated memory (read, write); and
- Release the allocated memory when it is not needed anymore.

The second part is explicit in all languages. The first and last parts are explicit in low-level languages but are mostly implicit in high-level languages like JS.

The purpose of a garbage collector is to monitor memory allocation and determine when a block of allocated memory is no longer needed and reclaim it. This automatic process is an approximation since the general problem of determining whether or not a specific piece of memory is still needed is undecidable.

Side Effects and Pure Functions

A function call that performs any of the following actions is said to have side effects:

- It reassigns any non-local variable.
- It mutates the value of any object referenced by a non-local variable.
- It reads from or writes to any data entity (files, network connections, etc.) that is non-local to your program.
- It raises an exception.
- It calls another function that has any side effects that are **not** confined to the current function. For instance, if you call a function that mutates an argument, but that argument is local to the calling function, then it isn't a side effect.

It's more correct to talk about whether a specific function **call** has any side effects. A function might have no side effects when given certain arguments, but that same function might have side effects when called with other arguments.

In practice, functions that have side effects have them regardless of what arguments are passed in.

Functions that have unexpected side effects are a major source of bugs. Be mindful of all side effects that your functions can cause.

If the function reassigns any variable that is not declared inside the function, the function has a side effect.

A more subtle side effect occurs when you do any kind of input or output (I/O) operation from within a function. Some everyday actions in this category include:

- Reading from a file on the system's disk
- Writing to a file on the system's disk
- Reading input from the keyboard
- Writing to the console
- Accessing a database
- Updating the display on a web page
- Reading data from a form on a web page
- Sending data to a remote web site
- Receiving data from a remote web site
- Accessing system hardware such as:
    - The mouse, trackpad, or other pointing devices
    - The clock
    - The random number generator
    - The audio speakers
    - The camera

The list goes on. Using any of these things are considered side effects.

Anything that causes JS to acquire data from or send data outside the actual program is a side effect.

If a function can raise an exception and doesn't catch and handle it, it has a side effect.

Suppose a function invokes another function, and that invoked function has a side effect that is visible outside of the calling function. In that case, the calling function also has a side effect.

One thing to note is that this type of side effect is only important when the invoked function has side effects that aren't local to the calling function. If the side effects can only be seen inside the calling function, then that side effect has no effect on whether the calling function has side effects.

Mixing Side Effects and Return Values

Most functions should return a useful value or they should have a side effect, but not both. By useful value, we mean that the function returns a value that has meaning to the calling code.

A function that returns an arbitrary value or that always returns the same value is not returning a useful value.

There are exceptions to the rule about mixing side effects and return values. For instance, if you read something from a database, you almost certainly have to return a value.

Pure Functions

Pure functions are functions that:

- Have no side effects; and
- Given the same set of arguments, the function always returns the same value during the function's lifetime. This rule implies that the return value of a pure function depends solely on its arguments.

The consistent return value is possibly the most important feature of pure functions. The fact that the return value is dependent solely on the arguments implies that **nothing else in the program can influence the function during the function's lifetime**.

A function's **lifetime** begins when the function is created. It ends when the function is destroyed.

A big benefit of pure functions is that the consistent return value and lack of side effects make them easy to test.

It's more correct to talk about whether a specific function **call** is pure or impure. A function that is pure with one set of arguments could be impure with another.

If the function is always pure when used as intended, then we say the function itself is pure. In practice, functions that are pure are always pure regardless of what arguments are passed in.

Pure functions are essential in functional programming, a programming paradigm that relies heavily on pure functions, declarative code, and no mutations.

Quiz:

#1:

Hoisting ensures that the names of variables, functions, and classes are known at the top of their scope.

Hoisting changes the definition point for variables, but does not initialize them to the initializer's value. It doesn't change the initialization point. It doesn't initialize the variables until JS encounters the initialization during execution.

#2:

Class names are hoisted, not the class definitions. The name of a class gets put into the TDZ, but the definition itself does not get hoisted.

Function expressions aren't hoisted, even when they're named.

Hoisted class declarations can't be accessed before the class declaration is executed.

#7:

Closures are lexical. They are created based on the structure of a program, not on anything that happens at execution time.

Closures provide access to the variables that are in the lexical scope where a function is defined. Variables that are in scope at the function's invocation point are only accessible if those variables are also in scope at the function's definition point.

#8:

Partial function application should create a new function.

#10:

Private data helps protect data integrity since developers must use the interface.

Private data helps prevent an object's user from becoming dependent on its implementation.

#18:

You can throw any value, either an object or a primitive, as an exception (i.e., **throw 'This is an error!'**).

#20:

A side-effect is anything that occurs within a function that modifies something not directly under the control of the function.

Lesson 3: Introduction to Testing

Introduction

As a beginner, you should write tests to prevent **regression**.

In software development, you can think of regression as an event that causes previously working code to stop working after a change to your code or environment.

In effect, your program rergresses to an earlier development state.

We write tests so that we can change things without having to verify manually that everything still works.

If you must give a name to what we're going to cover, you can call it learning unit testing. All advanced testing tools and methodologies build upon this knowledge.

Setting Up Jest

Note that **Jest** requires the **.test.js** component of the file name. It usually looks for files with a **.test.js** or **.spec.js** extension. These names also help in other ways; they help you identify and differentiate test files from other JS files.

To run a test file use the following:

jest temp.test.js

This command may raise an error since Jest requires a configuration file. You can place the configuration file in the same directory as the test file, or in any of its parent directories. We can use a **jest.config.js** file.

Lecture: Using Jest

A few terms you should know:

- A **Test Suite** is the entire set of tests that accompanies your program or application. You can think of it as all of the tests for a project.
- A **Test** is a specific situation or context that you're attempting to test. For instance, a test may attempt to verify that you get an error message when you try to log in with the wrong password. Each test can contain multiple assertions. You may sometimes see tests referred to as **specs**.
- An **Assertion** is the verification step that confirms that your program did what it should. In particular, many assertions test whether the return value of a function or method matches the expected results. You can make multiple assertions within a test. Assertions are also called **expectations**.

You don't have to provide a file name to the **jest** command. If you run it without arguments, it runs all of the **.test.js** files and outputs the results to the terminal.

When we run the **jest** CLI, it makes all the necessary Jest library methods available to the test file.

The **describe** method groups your tests. Groups are there to help you structure your tests into logical sections. It takes two arguments: the first one is a string that describes the group of tests and the second one is a callback. We write our tests inside the callback.

Note that **describe** is optional: you don't have to use it when testing. However, if you have more than a few tests, the grouping and description text will help identify which tests are failing.

Each invocation of **test** defines a new test. Like the **describe** method, it takes a description string and callback. The string argument describes the test.

Within each test, we need to make one or more assertions. These assertions confirm the behaviour that we're trying to verify. Before we make any assertions, however, we must set up any data that we need in the test.

Each expectation in Jest begins with a call to the **expect** method. The argument passed to **expect** is the value that we want to assert; it's often called the **actual value**.

The **expect** method, in turn, returns an object that includes a variety of **matcher** methods. Matchers compare the actual value passed to **expect** with the expected value, but don't return a meaningful value (i.e., they don't return a boolean value). Instead, they simply inform Jest of the results, and Jest takes care of treating that result as a success or failure.

There are many matchers available in Jest. For now, we'll stick with **toBe**; it merely checks for equality. Thus, if the value passed to **expect** is the same as the value passed to **toBe**, the assertion succeeds.

For example:

```
describe('The Car class', () => {
        test('has four wheels', () => {
                let car = new Car();
                expect(car.wheels).toBe(4);
        });
});
```

Jest has many options, including options that format the output in different ways.

The check mark in front of the test denotes success; a cross mark denotes failure. The description and test text may not appear when running multiple test files.

You may sometimes want to skip some tests. Jest lets you skip tests with the **test.skip** method. All you have to do is call **test.skip** instead of **test**. When you run a test file with skipped tests, the output reports that to you.

You can replace **skip.test** with **xtest** (i.e., they are identical so it's just an alias). Many developers prefer **xtest** since it's nearly effortless to enable and disable specific tests by just inserting or removing the letter **x**.

More On Matchers

List of matchers:

- **toBe** - Fails unless actual value === expected value

- **toEqual** - Same as **toBe** but can also test for object equality (i.e., **{a: 1}** is equal to **{a: 1}**). If two objects have the same number of properties with the same values, they're considered equal.

- **toBeUndefined** - Fails unless the actual value is undefined. Same as **toBe(undefined)**

- **toThrow** - Fails unless the expression passed in to **expect** raises/throws an error

- **toBeNull** - Fails unless the actual value is **null**. Same as **toBe(null)**

- **toBeTruthy** - Fails unless the actual value is truthy

- **toContain** - Fails unless the given array includes a value. Also finds substrings in strings.

A test that passes with **toBe** will also pass with **toEqual** but not necessarily the other way around.

toContain also works with strings and some other “iterable” types like sets. When working with strings, it matches substrings.

You may sometimes want to assert the opposite of a matcher. To do that, use the **not** property on the object returned by **expect** (i.e., **expect(car.wheels).not.toBeUndefined()**).

SEAT Approach

In larger projects there are usually 4 steps to writing a test:

- **S**et up the necessary objects.
- **E**xecute the code against the object we’re testing.
- **A**ssert the results of execution.
- **T**ear down and clean up any lingering artifacts.

These steps comprise the **SEAT** approach.

We can use the **beforeEach** function to run code that is duplicated in the tests that we want to perform.

The **beforeEach** callback is called before running each test.

Likewise, the **afterEach** callback, if present is called after running each test. In some cases, we may need a teardown to clean up files, log some information, or close a database connection.

It’s better to create a new object for each test so that you have one with a known state.

In the simplest cases, we don’t need either setup or teardown. However, keep in mind that there are 4 steps to running a test: SEAT. At the minimum, you need EA, even when the E is just a simple instantiation.

Code Coverage

When writing tests, we want to get an idea of code coverage, or how much of our program code was tested by a test suite.

Most testing coverage programs can determine coverage based on the percentage of functions or methods called by your tests or by the percentage of lines of code that executed as a result of your tests.

It doesn’t tell us whether our code works correctly, only that they seem to work with the test cases we used. It certainly doesn’t mean that we’ve covered all of the edge cases. While not foolproof, code coverage is one metric that you can use to gauge code quality and reliability.

There are many code coverage tools. We can supply the **--coverage** option to the **jest** command to get a table that shows information about the percentage of statements, branches, functions, and lines that we’ve tested:

```
jest --coverage todolist.test.js
```

The “% Lines” column shows the most useful data: this is the percentage of program lines that got executed at least once during testing.

It’s not always necessary to achieve 100% coverage, but the percentage should depend on the project. The more fault-tolerant your code must be, the more thorough your tests must be.

You can find a more detailed report by looking at the file **coverage/lcov-report/index.html** with your browser. This page shows a table that summarizes the same information that is shown in the terminal. However, the file names are clickable. If you click on one of the files, you can see a color-coded image of your code that shows how many times you executed each line (the green highlights to the right of the line numbers) and which lines were not tested (shown in red).

Note that the “Uncovered Line #s” column shows you which lines of your code haven’t been tested.

Summary

Don’t forget the following:

- Jest is the most popular testing library for JS.
- A test suite contains many tests. A test can contain many assertions.
- Use the **toBe** and **toEqual** matchers liberally, but don’t be afraid to look up other assertions when needed.
- Use the SEAT approach to write tests.
- Use code coverage metrics to gauge test quality but remember that it’s not the only metric.

Quiz:

#1:

Regression tests check for bugs that occur in formerly working code after you make changes somewhere in the codebase. Using tests to identify these bugs means we don’t have to verify that everything works manually after each change.

#5:

When instantiating objects in a **beforeEach** method, we should assign them to local variables and delcare those variables outside the **beforeEach** callback.

Including **Set Up** and **Tear Down** steps reduces redundancy in the Test Suite code.

#10:

You can’t test for an error with **toBe**.

Lesson 4: Packaging Code

Setting Up the Project Directory

The first step when creating a new JS project is to set up your working directory. You should restrict the name to just lowercase letters, digits, underscores, and hyphens.

Strange bugs arise when directory names contain spaces.

A project is simply a collection of one or more files used to develop, test, build, and distribute software. The software may be an executable program, a library module, or a combination of programs and library files.

Most Node-based projects follow a standard layout. In particular:

- Some specific files and directories must be present.
- Some kinds of data must be in specific locations.
- Some data must use well-defined formats.

The most common standard for Node projects is the npm standard.

Node projects typically have a strict organization. Specifically, developers expect to find test code in a **test** directory and code files in the **lib** directory.

Web-based programs generally require “assets” like images, JavaScript, and CSS (stylesheets) -- these often reside in an **assets** directory with a subdirectory for each file type: **images**, **javascript**, and **stylesheets**.

Depending on the type and scale of the project, you may also decide to house all browser-related code in a folder of its own, typically named **client**.

Using npm

A standard Node installation includes a vast library of code that is always available to your programs.

Developers often package the reusable code and make it available to the broader developer community.

The **npm database** hosts hundreds of thousands of such free code packages. You can download and use these packages in your projects.

Node Packages

Node packages, as the name suggests, are packages of code that you can download, install, and use in your code and system. You can import some packages into your programs, and use others from the terminal command line. You can even use some packages in both ways.

The **npm** command, which comes bundles with node, manages your packages.

There are many node packages:

- **eslint**: This package checks for violations of JS style conventions and other potential issues in your code.
- **jest**: The testing library makes the **jest** executable available as a command in your terminal.
- **readline-sync**: This package lets you get input from the user in a terminal program in a simple, synchronous manner.

When we create modules for the exclusive use of our project, we typically use a relative path when we import the file. However, with npm packages, we don't need or use a relative path.

Instead, we can omit the relative path, and **require** looks inside the **node_modules** directory for a folder with the same name as the argument.

Local vs. Global Packages

You can install Node packages in two ways: locally and globally.

Packages needed by an application or project are commonly installed locally inside the project directory.

Typically, these packages are often the kind that we import into our program, but they can also provide executable programs that we need.

To install a package locally, use the **npm install** command without the **--global** option.

For example:

npm install lodash --save

This command downloads and installs the **lodash** package in the **node_modules** directory. The **npm** command looks for an existing directory with the name **node_modules** in the current folder. If it doesn't find one, it looks in the parent directory. If it doesn't find it there, it keeps searching up the directory hierarchy until it finds one. If it doesn't find one, it creates one in the directory where you ran the **npm install** command.

This directory search is why you should never nest your project directory inside a directory that already contains a **node_modules** directory. If you do, npm will store your local packages in the existing **node_modules** directory, not a new one in your project directory as intended. You want to install all of your dependencies inside your project directory, not above it.

An npm package is simply a node project with files and sub-directories inside it.

Importing a local package into your code uses the **require** function.

The **lodash** module object is, by convention, referenced by an underscore variable. The library has a ton of convenience functions for manipulating and querying collections.

Importing an entire module merely to use one function is unnecessary. We can only import the function we need.

With packages that have multiple independent files like **lodash** where each file exports its public names with **module.exports**, we can use:

const chunk = require('lodash/chunk');

The benefit of this approach is more efficient loading; Node doesn't have to import the entire module, just the part that you need. It's also less burdensome on system memory.

Not all node packages have that type of structure. When a package doesn't provide separate independent files, we must use a different technique:

const chunk = require('lodash').chunk;

This code doesn't solve the processing issue; Node still needs to read the entire module. However, thanks to garbage collection, it doesn't put much strain on memory resources; the imported but unneeded names are immediately eligible for garbage collection.

The **package.json** and **package-lock.json**

Typically, your project should have a **package.json** file that lists all the packages that your project needs, together with the versions of each package. The file also provides a convenient place to store some other configuration settings.

To initialize (i.e., create and populate) a **package.json** file, we'll run **npm init**.

The **package.json** file is, in effect, a configuration file written in JSON format.

Npm's main strength is its ability to manage a project's dependencies. Thus, it's natural that a project's **package.json** mostly concerns itself with specifying the project's dependencies.

To add a dependency to our **package.json** file, we'll add a **dependencies** key to the file and add some dependencies as the key's values. We can then save the file and run **npm install** without arguments or oprtions.

This command installs the dependencies that we added to the **package.json** directory, then builds a new file named **package-lock.json**.

The **package-lock.json** file shows the precise versions of the packages that npm installed. It also shows the dependencies of each package and the version of each dependency.

Npm and the **package.json** file follow semantic versioning. For example, in **package.json**, the versions that we've specified are the major versions. Node then chooses a specific minor and patch version that is compatible with the rest of the dependencies and adds that information to **package-lock.json**.

The next time we run **npm install** it'll look at **package-lock.json** and install the specific versions specified there. That is why you must add **package-lock.json** to your git repo. You want all contributors and users of your package to install the correct versions and avoid versioning problems.

You can add a new dependency to your project and **package.json** in one of two ways:

- Directly add the dependency to **package.json**; and
- Use **npm install**.

You don't have to edit **package.json** directly. Instead, you just run **npm install** with the package name and the **--save** option to install the package and save it to both **package.json** and **package-lock.json**.

The **save** option tells npm to save the package to the dependencies list in **package.json**.

Sometimes, your project only needs a package during development. Ideally, you should only install such tools in the development environment, not in production.

Fortunately, **package.json** lets you identify such development dependencies by adding a **devDependencies** property. The easiest way to do that is to use **npm install** with the **--save-dev** option.

The reason you install a package locally, though, is that you want to use that specific version of the package inside your project. Your globally installed version may not be compatible with your project for some reason.

There are several ways to run a local npm executable package, but the simplest way is to precede the executable's name by **npx**:

```
npx eslint lib/todolist.js
```

npx merely checks for a local installation first. If it can't find the package locally or globally, it downloads and uses a temporary version of the named package.

To delete a dependency, use the **npm uninstall** command. This command removes the package from **node_modules**, but it doesn't remove it from the **package.json** dependencies.

To delete it from the dependencies as well, use the **--save** option. Likewise, **--save-dev** removes development dependencies.

You can also use **npm prune** to remove dependencies. This command is useful when you manually remove some dependencies from **package.json** and want to remove the packages from **node_modules**.

As a general rule, you should install almost all of your packages locally in your project's **node_modules** directory. It ensures that projects that need specific versions of packages have them locally available. Since each package is local to the project, each project is free to use the version it needs.

Global Packages

Not all node packages require local installation. To install a package globally, use the **-g** flag with **npm install**.

If you're considering a global install for a package, be sure to check the documentation for that package; don't install the package globally unless they recommend doing so or explicitly state that global installations are allowed.

Transpilation

Transpilation is the process of converting source code written in one language into another language with a similar level of abstraction to the original code.

Most often, that means converting code that uses the latest language features to an older version of the language. For instance, we may want to transpile a program written with ES6 features like **let**, **class**, and arrow functions into a form that lets it run in browsers that don't recognize these features.

The JS standard is evolving rapidly; new features are added to the language often. It takes time to implement new features.

Thus, the JS runtime environments, such as Node.js and the browsers, often lag behind the standard. Transpilers, such as Babel, let us use the newer features in our code without worrying about whether it works in the intended runtime environment. Since users often don't update their browsers, transpilers are most useful when working with browser-based applications.

Babel

There are many tools available for transpiling ES6+ JS code into ES5 code. Babel is, by far, the best and most widely used.

To use Babel from the command line we must install **@babel/core** and **@babel/cli** as local packages:

```
npm install --save-dev @babel/core @babel/cli
```

We can now transpile the files in our **lib** folder:

```
npx babel lib --out-dir dist
```

This command tells Babel to transpile all JS files in the **lib** directory and to output the resulting code to files with the same names in the **dist** directory.

We need to install the Babel **env** preset to provide smart transpilation. A preset is a plug-in that has all the information needed to compile one version of JS to another.

The **env** transpiles code to ES5 and automatically detects what it needs to do to support different environments. We can install **env** preset with the following command:

```
npm install --save-dev @babel/preset-env
```

We can now tell Babel what presets it should use:

```
npx babel lib --out-dir dist --presets=@babel/preset-env
```

Automating Tasks with npm Scripts

Scripts automate repetitive tasks such as building your project, minifying files, and deleting temporary files and folders.

npm scripts provide a simple and versatile way to automate repetitive tasks.

The Script Object in **package.json**

To use npm scripts, you first need to define them in the **package.json** file with the **“scripts”** object.

The **“scripts”** object takes a series of key/value pairs in which each key is the name of the script, and the value is the script you want to run.

npm scripts are merely a way to run terminal commands. Npm scripts let us name our terminal command(s). That can be very useful, especially when a task involves multiple commands or when it is lengthy or has a bunch of arguments.

One useful feature of npm scripts is that **npm** knows how to find command-line executables, even those that are part of a local package. It uses commands from local packages in preference to those stored in more traditional locations, such as the directories specified by the **PATH** environment variable. Thus, you don’t need to use the **npx** command.

One significant difference with omitting **npx** from a script’s definition is that **npx** can search for and install packages for one-time execution. Without **npx**, you can only use pre-installed packages.

Packaging the TodoList

Most JS projects use npm packages as the distribution mechanism.

When you’re ready to prepare your project for distribution follow these steps:

- Create a **package.json** file.
- Provide values for the **name**, **version**, and **main** fields:
    - **name** is the name of your package.
    - **version** is the initial module version.
    - **main** is the name of the file that Node will load when someone imports your package.
- Publish your node package.

The final step you need to take to publish your package is to execute the following command:

```
npm publish --access public
```

Note that for the **npm publish** command to be successful, you’ll need to have setup an **npm account** and logged in using the **npm adduser** or **npm login** command.

Summary

npm provides a library of code that you can download and run, or use directly inside your JS programs. You use the **npm** command to manage the packages you need.

Babel is a JS transpiler that compiles code written with newer syntax into older code. Transpilation is useful when we want to use the latest JS features, but also want our code to run in environments that don't support those features.

The **package.json** file provides a way to list all of your project's dependencies and their versions in a single file. **npm** uses this file to resolve the interdependencies between all the packages and installs the appropriate versions.

npm also provides the mechanisms you need to publish your own modules. Those modules can be packages of code that you require into your JS programs or independent command-line programs.

Lesson 5: Asynchronous Programming

Asynchronous functions, as opposed to synchronous functions, are functions that don't block execution for the rest of the program while they execute.

Said differently, asynchronous functions run concurrently with other operations so that the caller doesn't have to wait for the task to finish running.

Asynchronous Execution with setTimeout

Sequential code is when each line runs in sequence. Another term for such code is **synchronous code**.

It's possible to write code that runs partly now, then pauses and continues to run later after a delay of milliseconds, minutes, hours, or even days. We call such code **asynchronous code**; it doesn't run continuously or even when the runtime encounters it.

setTimeout() is one of the simplest ways to run code asynchronously. It takes two arguments: a callback function and the time to delay execution of the callback, in milliseconds (1/1000th of a second or 1 ms). It sets a timer that waits until the specified delay elapses, then invokes the callback.

For now, just realize that code being run by **setTimeout()** only runs when JS isn't doing anything else.

That means that your program must stop running before code executed with **setTimeout()** will even begin to run.

Even if your program runs for days at a time, any code executed by **setTimeout()** will not run until your program has no more code to run. No matter how small the delay period, nothing will happen.

Ultimately, working with asynchronous code means you must reason about both what it does and when it does it.

Repeating Execution with setInterval

Instead of invoking the callback once, **setInterval** invokes it repeatedly at intervals until told to stop.

Like **setTimeout**, **setInterval** is not part of the JS specification. However, most environments make it available.

We can write **let id = setInterval(function() { console.log('hello"); }, 2000)**.

In this example, **setInterval** returns an identifier that we can use to cancel the repetitive execution. To do so, we pass the identifier to the **clearInterval()** function (i.e., **clearInterval(id)**).

setInterval is useful when you must run some code at regular intervals. For instance, perhaps you need to auto-save a user's work in a large web form:

```
function save() {
        // send the form values to the server for safekeeping
}

// Call save() every 10 seconds
let id = setInterval(save, 10000);

// Later, perhaps after the user submits the form
clearInterval(id);
```

More Async Functions

Promises are a new feature of JS that have been adopted by many new browser API's and nearly all frameworks and libraries.

They are another solution to the problem solved by callbacks, and a way to register behaviour that needs to run after an event.

Unlike callbacks, though, promises are objects that represent an eventual value. Having an object within a program that represents an eventual value allows for a variety of powerful abstractions to be built.

But promises also have other benefits, in that they provide behaviour that is cumbersome to create with event listeners.