ADVANCED JAVASCRIPT UNLEASHED

Master Advanced JavaScript Concepts like Prototypes, Symbols, Generators, and More

newline

YOUSAF KHAN

Unpacking Intermediate JavaScript - Confusing Parts

Yousaf Khan

This book is for sale at http://leanpub.com/advanced-javascript-unleashed

This version was published on 2024-04-10

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

Table of Contents

What is JavaScript

Course Overview

JavaScript is arguably the most widely used programming language on the planet, and there is a vast amount of content available for learning JavaScript. The problem is that not all the content on the internet does a good job of explaining the complex or confusing concepts of JavaScript.

This course aims to teach different concepts in JavaScript that are not easy to grasp, especially for beginners. Topics like closures, coercion, the asynchronous nature of JavaScript, etc., are examples of topics that most beginners struggle with because they are not easy to understand. The goal of this course is to provide in-depth, easy-to-understand explanations of such confusing topics.

Even those who have been working with JavaScript for a few years might need help understanding some of the concepts covered in this course or might have some gaps in their understanding. The goal of this course is to fill those gaps.

This course will not only provide easy-to-understand explanations of fundamental JavaScript topics like hoisting, coercion, event loop, etc., but will also cover advanced topics like promises and async-await syntax in a way that will be easy for the students to understand.

By the end of this course, students will have a deep understanding of the concepts covered in this course. They will become better at JavaScript by having a solid understanding of the topics that most JavaScript beginners struggle with. Students will be able to debug JavaScript code better and avoid common pitfalls by having a deep understanding of fundamental but confusing JavaScript topics.

Prerequisites

This course assumes that students have a basic understanding of JavaScript and programming in general. Understanding of topics like variables, loops, objects, arrays, functions, etc., is assumed.

Running code examples

You can download code examples from the same place where you purchased this book.

If you have any trouble finding or downloading the code examples, email us at us@fullstack.io.

Who is this course for?

The course is for those who:

- are struggling to have a deep understanding of JavaScript
- understand fundamental to advanced JavaScript concepts but want to fill gaps in their understanding of different JavaScript topics that are key to becoming a good JavaScript developer (topics like closures, coercion, event loop, asynchronous JavaScript, promises, etc.)
- want to understand JavaScript in-depth and get better at working with JavaScript
- want to get better at debugging JavaScript and preparing for job interviews, as this course will provide a deeper understanding of the JavaScript language
- want to solidify their understanding of JavaScript's fundamental topics
- want to avoid frustration due to a lack of understanding of JavaScript's confusing topics
- want to excel in working with JavaScript

This course is unlike online articles/blogs and many JavaScript video courses that either fail to provide in-depth, easy-to-understand explanations of JavaScript topics that are not easy for beginners to understand or do not cover all the necessary topics (fundamental to advanced) that a good JavaScript developer must understand.

This course combines fundamental but confusing JavaScript topics in a single course that aims to provide a solid understanding to the students and cover all the topics that are key to understanding JavaScript in depth.

JavaScript is currently the most widely used programming language. With the rise of modern front-end frameworks and platforms like Node.js, it seems that JavaScript will be the most used programming language for the foreseeable future.

It is the programming language of the web, most commonly used to add interactivity to web pages. With the help of technologies like Node.js, which allow us to write JavaScript outside the browser environment, software developers can now write full-stack applications by learning just one programming language.

So, how did this programming language that we know today come into existence? Let’s take a brief tour of its history.

History of JavaScript

In the early days of web browsers, web pages could only be static, without any dynamic behavior or interactivity. A server was needed for simple things like form input validations. With limited internet speeds in those days, requiring a roundtrip to the server just for input validation was highly undesirable.

There was a need for a client-side scripting language that allowed us to add interactivity to web pages. In 1995, a software developer at Netscape began working on such a language, initially called “mocha,” which was later changed to “LiveScript.”

This language was supposed to be part of the Netscape Navigator 2 browser, but just before the release of the browser, LiveScript was renamed to “JavaScript.”

When Netscape released its Netscape Navigator 3 browser, Microsoft came up with its web browser known as Internet Explorer 3, which included

Microsoft’s implementation of the JavaScript language.

This presented a problem: there were now two separate implementations of the JavaScript language. There was a need for a standards body that was responsible for the advancement of the JavaScript language.

Standardization of the JavaScript language

The problem of multiple versions of JavaScript led Netscape to submit the JavaScript language as a proposal to Ecma International, a standards organization.

JavaScript is standardized as “ECMAScript” and its standard is ECMA-262.

As mentioned in the previous lesson, the initial name for JavaScript was Mocha, which was later changed to LiveScript, and just before the release of the Netscape Navigator 2 browser, the name was changed to JavaScript.

So what’s the reason behind using the word “Java” inside the name “JavaScript” when there was already a popular programming language named “Java”?

This question of whether these two languages are related to each other arises in the mind of almost every person who is already familiar with one of these two languages and comes across the other language.

It was just a **marketing move**. Java was popular at that time, and Netscape wanted to cash in on the popularity of the Java language. That is why Netscape renamed the language from LiveScript to JavaScript.

Apart from some similarities in the syntax and, of course, in the name of both languages, the two languages are very different from each other.

Ecma International, TC39

JavaScript is standardized as **ECMAScript** by **Ecma International**, a standards organization.

Within Ecma International, people responsible for developing the JavaScript language are part of the **Technical Committee 39** (TC39). This committee consists of different stakeholders from the JavaScript community. Mostly people from big companies like Google, Microsoft, etc. are part of this group.

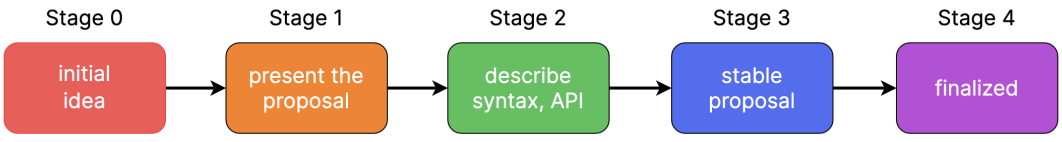
Other people can become part of TC39 as well and make their contribution in the development of the JavaScript language. Details of how anyone can either contribute or be part of TC39 meetings as a member can be found on the TC39 website.

TC39 also has a GitHub account which hosts multiple repositories related to their work. Repositories include notes from TC39 meetings, different proposals for the JavaScript language, details of how TC39 works, etc.

Proposals process

TC39 has a well-defined process through which proposals are passed before they can be added to the ECMAScript specification and ultimately be part of the JavaScript language.

Each proposal is passed through five stages which are mentioned below:



process stages

Stage 0

The first stage represents someone’s idea that they think is worthy of being part of the JavaScript language.

Not all proposals at this stage move to the next stage, but the ones that do are the ones that gain enough interest and have a TC39 committee member who is willing to work on the idea. Proposals at this stage that gain enough interest are discussed in the TC39 meeting. If the discussion goes well, the proposal is moved to the next stage.

Stage 1

Proposals at this stage are discussed and developed by the community (including non-committee members), and if the proposal still has enough interest and is thought to have potential benefit if added to the specification, initial specification language and API are developed and discussed in the TC39 meeting. Provided everything goes well, the proposal is moved to the next stage.

Stage 2

At this stage, the API, syntax, and so on are refined and described in greater detail using the formal specification language. Polyfills and Babel plugins may be developed at this stage for real-world experimentation with the solution of the proposal.

Once the proposal has been described in enough detail, it can be considered for the next stage.

Stage 3

Proposals at this stage are almost ready to be included in the ECMAScript specification.

For proposals to be moved to the final stage, they need to meet the following two criteria:

- a suite of tests
- two compatible implementations of the proposal that passed the tests

Once the conditions mentioned above are met, and the committee has a consensus that the proposal is ready to be part of the ECMAScript specification, it is moved to the next stage.

Stage 4

At this stage, the proposal is complete and is ready to be included in the ECMAScript specification and ultimately be added to the JavaScript language.

A pull request is generated to the ecma262 GitHub repository to include it in the specification. Once the pull request is approved, the proposal is part of the specification and ready to be implemented in the JavaScript engines that still need to implement it.

For further details of this process, refer to a process document on TC39's website, which explains the various stages through which proposals pass through.

Track upcoming features

Additions to the JavaScript language in recent years have completely transformed JavaScript into a programming language, and sometimes, it can be hard to stay updated with the new additions to the language.

Keeping ourselves updated with new changes is one thing, but how can we track changes that could potentially be part of JavaScript in the future?

The answer to this question lies in TC39's GitHub repository, which contains meeting notes and proposals that are at various stages of the proposal process described in the earlier lesson.

Notes and proposal repositories are a great place to keep track of upcoming changes, and in general, the TC39 GitHub repository is useful for tracking the work that the TC39 committee is doing.

How is ECMAScript versioned?

ECMAScript version 5, also known as “ES5”, came out in 2009, and until that point, the ECMAScript specification was referred to by its **edition** number, but with the introduction of ECMAScript’s 6th edition, also known as “ES6”, TC39 added the **year** number in the language’s name.

As the 6th edition came out in the year 2015, it was “ECMAScript 2015”, “ES2015” for short. The later editions were also named using the year number in which they came out, e.g., “ES2016”, etc.

You will also see these editions referred to as “ES6”, “ES7”, etc., and this naming convention is also fine to use. Which one you choose is up to your preference. Most of the time, you will see both types of names used interchangeably.

At the time of this writing, **ECMAScript 2023** is the latest edition of ECMAScript.

JavaScript is thought of as an “interpreted” language by many because of the nature of how it is executed, but calling JavaScript just an “interpreted” language is not entirely true.

In the case of compiled languages, compilers usually compile the source code and produce a binary executable file, which can then be distributed and executed.

On the other hand, in the case of interpreted languages, interpreters do not produce an executable output file; unlike compilers, which compile the source code ahead of time, interpreters read and execute code on the fly.

In the case of JavaScript, the JavaScript engines do not output an executable file, which is one of the reasons it is thought of as an interpreted language.

However, the JavaScript code is *compiled* into an intermediary form known as **byte code**, which is then executed by the virtual machine. The virtual machine **interprets** byte code, but modern JavaScript engines don't just interpret the byte code; they include what's known as the "**Just-in-time (JIT) compiler**" to compile the byte code into native machine code, which is executed at a faster speed than the byte code.

JIT Compiler

Just-in-time (JIT) compilation is a technique used by many modern JavaScript engines to increase the execution speed of the JavaScript code.

JavaScript code is converted into byte code, and the JavaScript engine then executes this byte code. However, modern JavaScript engines perform many optimizations to increase the performance of JavaScript code. These optimizations are performed based on the information collected by the engine while it is executing the code.

One way to optimize performance is to compile byte code into machine code, which executes faster than the byte code. The JavaScript engine identifies the "hot" parts of the code to do this - parts that are being executed frequently.

These "hot" parts of the code are then compiled into native machine code, and this machine code is then executed instead of the corresponding byte code.

So how is the JIT compiler different from a traditional compiler used by languages like C++? Unlike traditional compilers, which compile the code ahead of time, the JIT compiler compiles the code at runtime while the code is being executed.

While Javascript code is still distributed in source code format rather than executable format, it is compiled into byte code and possibly native machine code.

So, coming back to the question: is JavaScript a compiled or interpreted language? It is safe to say that it is both - **compiled as well as an interpreted language**.

We don't necessarily need to understand the nitty-gritty details of how exactly the JavaScript code that we write is executed, but to develop a good understanding of the language; it is important to have a basic understanding of how our JavaScript code gets transformed into something that a machine can understand and execute.

It is also important to understand what different things come into play while our code is executing; concepts like "**execution context**," "**call stack**," etc. are crucial to understanding the JavaScript language's runtime behavior and being able to work with it and debug it efficiently.

JavaScript Engine

To execute JavaScript code, we need another software known as a **JavaScript engine**. This engine contains all the necessary components to transform the code into something the machine can execute.

Different browser vendors typically create JavaScript engines; each major vendor has developed a JavaScript engine that executes the JavaScript code in their browser.

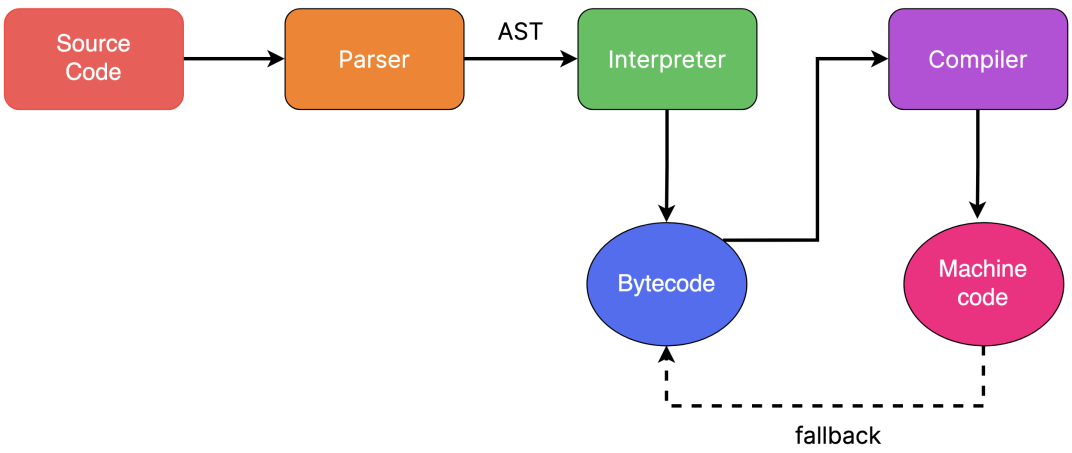
The following table shows some major browsers and their JavaScript engines.

Browser	Engine
Google Chrome	V8
Edge	Chakra
Mozilla Firefox	Spider Monkey
Safari	JavaScriptCore

While there are differences in the steps taken by each JavaScript engine to execute the JavaScript code, the major steps taken by each engine are more

or less the same and we will try to have a high-level overview of how our code gets transformed and executed by the JavaScript engines by understanding the Google Chrome's **V8** engine.

The following image shows the high-level overview of the execution pipeline of the V8 engine:



process stages

The JavaScript engine is complicated software that contains lots of steps and components that are used to transform and execute the JavaScript code, but the above image shows a simplified version of the execution pipeline of the V8 engine.

:::note Please note that the team working on the V8 engine is continuously improving it; as a result, the simplified execution pipeline shown in the image above may change in the future. :::

Let's get a better understanding of how the execution pipeline shown above works by understanding what happens at each step of this pipeline.

Source Code

Before the JavaScript engine can begin its work, the source code needs to be downloaded from some source. This can either be from the network, a cache, or a service worker that pre-fetched the code.

The engine itself doesn't have the capability to download the code. The browser does it and then passes it to the engine, which can then begin transforming it and eventually execute it.

Parser

After downloading the source code, the next step is to transform it into **tokens**. Think of this step as identifying different parts of the code; for example, the word "function" is one token that is identified as a "keyword." Other tokens may include a string, an operator, etc. This process of dividing the code into tokens is done by a "scanner," and this process is known as "**tokenization**."

The following JavaScript code:

```
function add(num1, num2) {
  return num1 + num2;
}
```

It can be tokenized as shown below:

```
[
  { type: "keyword", value: "function" },
  { type: "identifier", value: "add" },
  { type: "openRoundParen", value: "(" },
  { type: "identifier", value: "num1" },
  { type: "identifier", value: "num2" },
  { type: "closeRoundParen", value: ")" },
  { type: "openCurlyParen", value: "{" },
  { type: "keyword", value: "return" },
  { type: "identifier", value: "num1" },
  { type: "addOperator", value: "+" },
  { type: "identifier", value: "num2" },
  { type: "closeCurlyParen", value: "}" }
];
```

Once the tokens have been generated, the parser uses them to generate an Abstract Syntax Tree (AST), a set of objects that represent the structure of the source code.

AST Explorer is a cool website that you can use to visualize the AST. Go ahead and paste the code above in the AST explorer and explore the generated AST.

Interpreter

The **Bytecode Generator** uses the AST produced by the parser to generate the bytecode. This generated bytecode is taken by the **Bytecode Interpreter**, which then interprets it.

V8 also passes the generated bytecode through some optimizers, which perform some optimizations to ensure efficient execution of the bytecode by the Bytecode Interpreter.

Compiler

While the bytecode is executed, the JavaScript engine collects information about the code being executed. The engine then uses this information to optimize the code further.

For example, the JavaScript engine can identify the parts of code that are being executed frequently, also known as the “hot” parts of the code. The “hot” parts of the code are then compiled into native machine code to ensure that these parts get executed as fast as possible.

However, the optimized native machine code sometimes has to be deoptimized back to the bytecode generated by the Bytecode Generator because of the way the code was written.

The need for falling back to the bytecode arises from the fact that JavaScript is a dynamically typed language. The dynamic nature means that we can call a particular JavaScript function with different kinds of values.

Consider the following code:

```
function print(obj) {
  console.log(obj);
}
```

The above function can be called with different “shapes” of objects.

```
print({ a: 1, b: 2, c: 3 });
print({ a: 1, c: 3 });
print({ b: 2 });
```

This means that if the `print` function is called multiple times with objects with the following shape:

```
{ a: 1, b: 2, c: 3 }
```

and if it is compiled to native machine code, but then if the same function is called with an object with a *different* shape, the JavaScript engine cannot use the optimized machine code and has to fall back to the bytecode.

The optimized native machine code is generated using the information collected during the execution of the JavaScript code. The native machine code requires certain checks to ensure that the assumptions made during the generation of the native machine code are not violated. If the checks fail, the JavaScript engine has to execute the bytecode instead of the native machine code. This process is called **deoptimization**.

References:

The following resources can be used to learn more about how the JavaScript code is executed:

- Understanding the V8 JavaScript Engine - (freeCodeCamp talk by Lydia Hallie)
- How JavaScript Works: Under the Hood of the V8 Engine - (freeCodeCamp blog)
- What does V8’s ignition really do? - (stackoverflow post)

- Ignition - an interpreter for V8 - (youtube video)
- Blazingly fast parsing, part 1: optimizing the scanner - (V8 blog)
- Overhead of Deoptimization Checks in the V8 JavaScript Engine - (paper by Dept. of Computer Engineering, University of California)

Whenever any JavaScript code is executed, it is executed inside an environment. By “environment”, I mean everything that is accessible by the code that aids in its execution. For example, the value of `this`, variables in the current scope, function arguments, etc. This environment is known as the “**Execution Context**”. Every time any JavaScript code is executed, an execution context is created before its execution.

:::note

Understanding the concept of execution context lays the foundation for understanding other JavaScript concepts like *hoisting*, *closures*, etc. These topics will be covered in later chapters.

:::

Types of execution contexts

The following are two main types of execution contexts that we will discuss:

- Global execution context
- Function execution context

:::info

There is a third type of execution context that is created for the execution of code inside the eval function. Still, as the use of the `eval` function is discouraged due to security concerns, we will only discuss the types of execution context mentioned above.

:::

Global execution context

The global execution context is the base context created whenever JavaScript code is loaded for execution. Global code, i.e., the code that is not inside a function, is executed inside a global execution context.

The global context contains the global variables, functions, etc. It also contains the value for `this` and a reference to the outer environment, which, in the case of a global execution context, is `null`.

Function execution context

Every time a JavaScript function is called, a new execution context is created for the execution of that function. Just like the global execution context, the function execution context contains:

- The variables and functions are declared inside the function.
- The value of `this` inside the function for the current function call.
- A reference to the outer environment.

The function execution context also contains the arguments passed to the function.

Execution context phases

Execution contexts have following two phases:

- Creation phase
- Execution phase

Creation phase

As the name suggests, the execution contexts (global and function) are created during the creation phase.

During this phase, the variable declarations and references to functions are saved as key-value pairs inside the execution context. The value of `this` and a reference to the outer environment are also set during this phase.

The values for variables are not assigned during the creation phase. However, variables that refer to functions do refer to functions during this phase. Variables declared using `var` are assigned `undefined` as their value during this phase, while variables declared using `let` or constants declared using `const` are left uninitialized.

:::info

In the case of a global context, there is no outer environment, so reference to the outer environment is set to `null`, but in the case of a function context, the value of `this` depends on how the function is called, so the value of `this` is set appropriately.

:::

Lexical and Variable environments

During the creation phase, the following two components are created:

- Lexical environment
- Variable environment

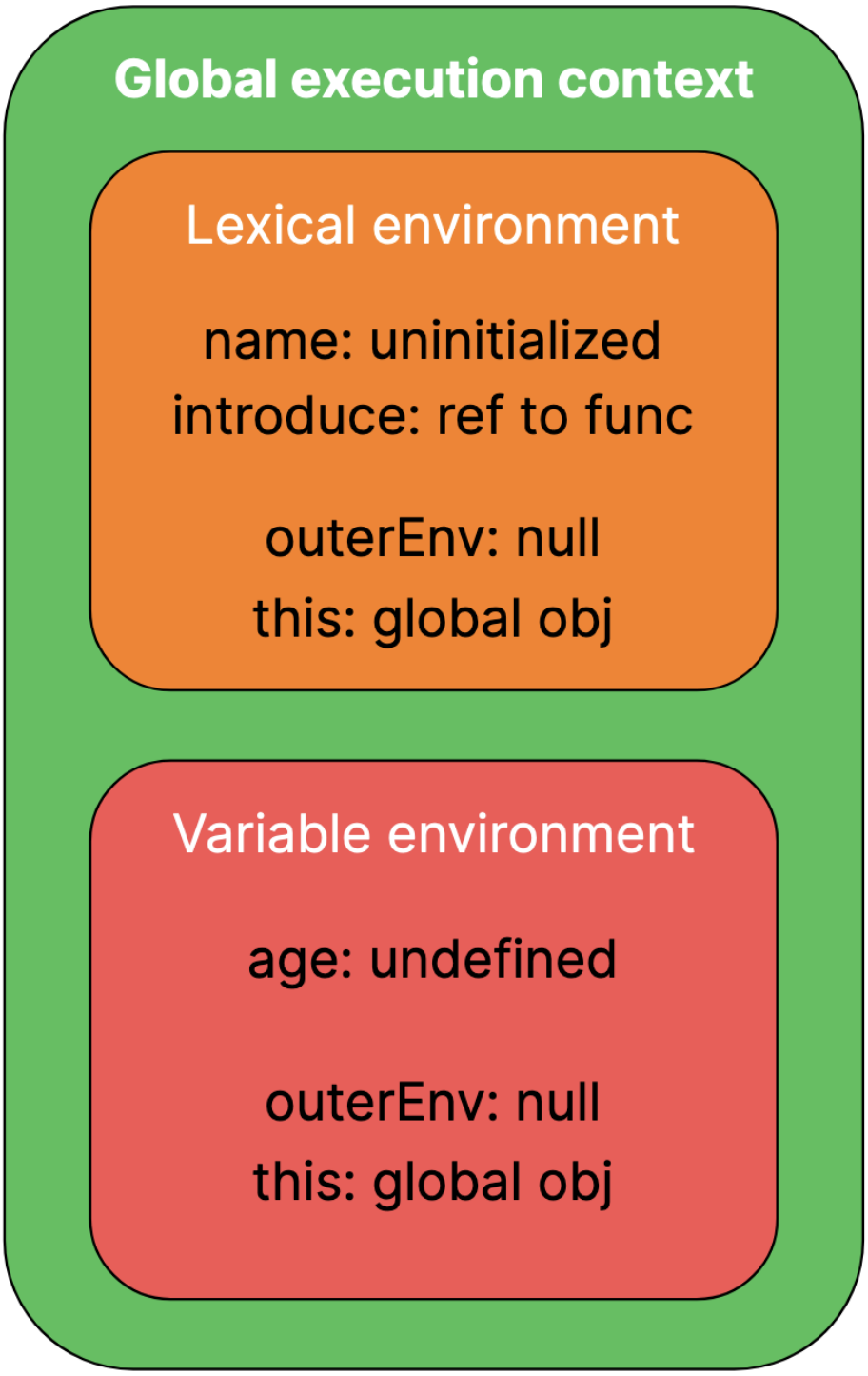
Lexical and Variable environments are structures that are used internally to hold key-value mappings of variables, functions, reference to the outer environment, and the value of `this`.

The difference between the lexical and variable environments is that the variable environment only holds the key-value mappings of variables declared with the `var` keyword, while function declarations and variables declared with `let` or constants declared using `const` are inside the lexical environment.

Consider the following code:

```
let name = "Jane Doe";
var age = 20;

function introduce(name, age) {
  console.log("Hello, I am " + name + " and I am " + age + " years old");
}
```

The execution context for the above code during the creation phase can be conceptually visualized as shown in the image below:



global execution context

Execution phase

As mentioned earlier, after the creation phase, different variables in the execution context are yet to be assigned their respective values. Assignments are done during the execution phase, and the code is finally executed.

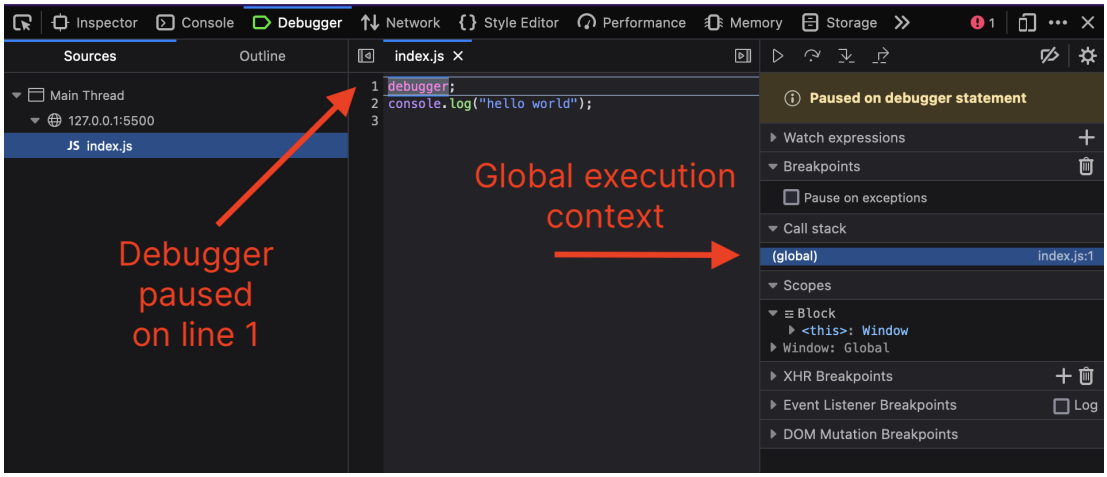
References

Following resources can be used to learn more about the execution context in JavaScript:

- Understanding Execution Context and Execution Stack in Javascript - (blog)
- JavaScript Execution Context – How JS Works Behind The Scenes - (freeCodeCamp blog)
- JavaScript Under The Hood - Execution Context - (youtube video)

A call stack is a structure that is used internally by the JavaScript engine to keep track of the piece of code that is currently executing. The call stack is simply a stack data structure that aids in the execution of the JavaScript code by keeping track of currently executing code. You can also think of a call stack in JavaScript as a collection of execution contexts.

Before executing any JavaScript code, a global execution context is created and pushed on the call stack. This can be easily visualized using the debugger in the browser developer tools.



global execution context pushed on the callstack

:::note

The `debugger` keyword simply creates a breakpoint, forcing the debugger to stop at the line containing the `debugger` keyword. We will cover debugging in a later chapter.

:::

Note that in the above image, before the execution of the `console.log` statement, there is something named “global” in the call stack. This is a global execution context that is created and pushed on the call stack before executing the code.

:::note

The label “global” is not important, and different browsers may use different labels to represent the global execution context in the call stack. For example, the debugger in the Google Chrome browser shows “(anonymous)” instead of “global.” The above screenshot was taken using the Firefox browser.

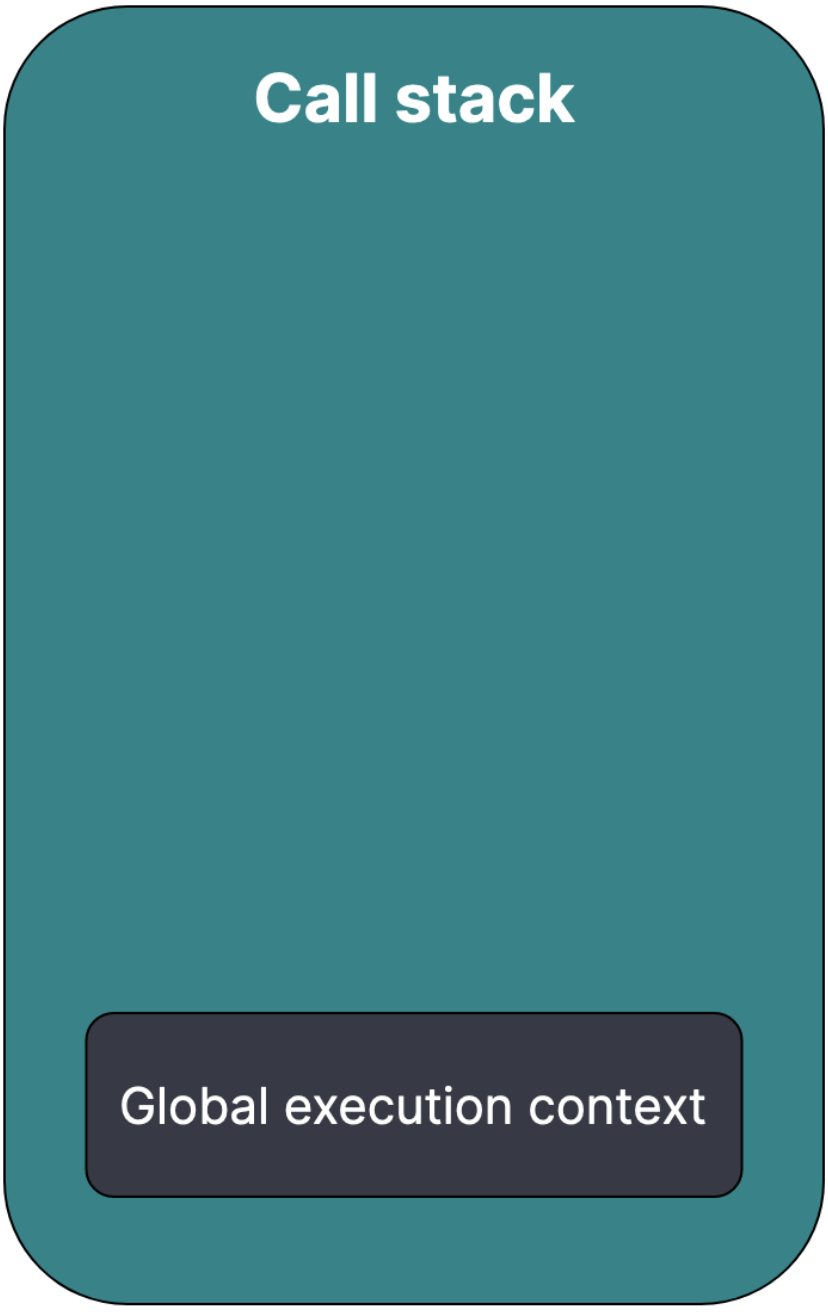
:::

After pushing the global execution context on the call stack, any function calls encountered during the execution of the code will lead to more entries in the call stack. For every function call, a new entry is added to the call stack before that function starts executing, and as soon as the function execution ends, that entry is popped off the stack. Consider the following code:

Consider the following code:

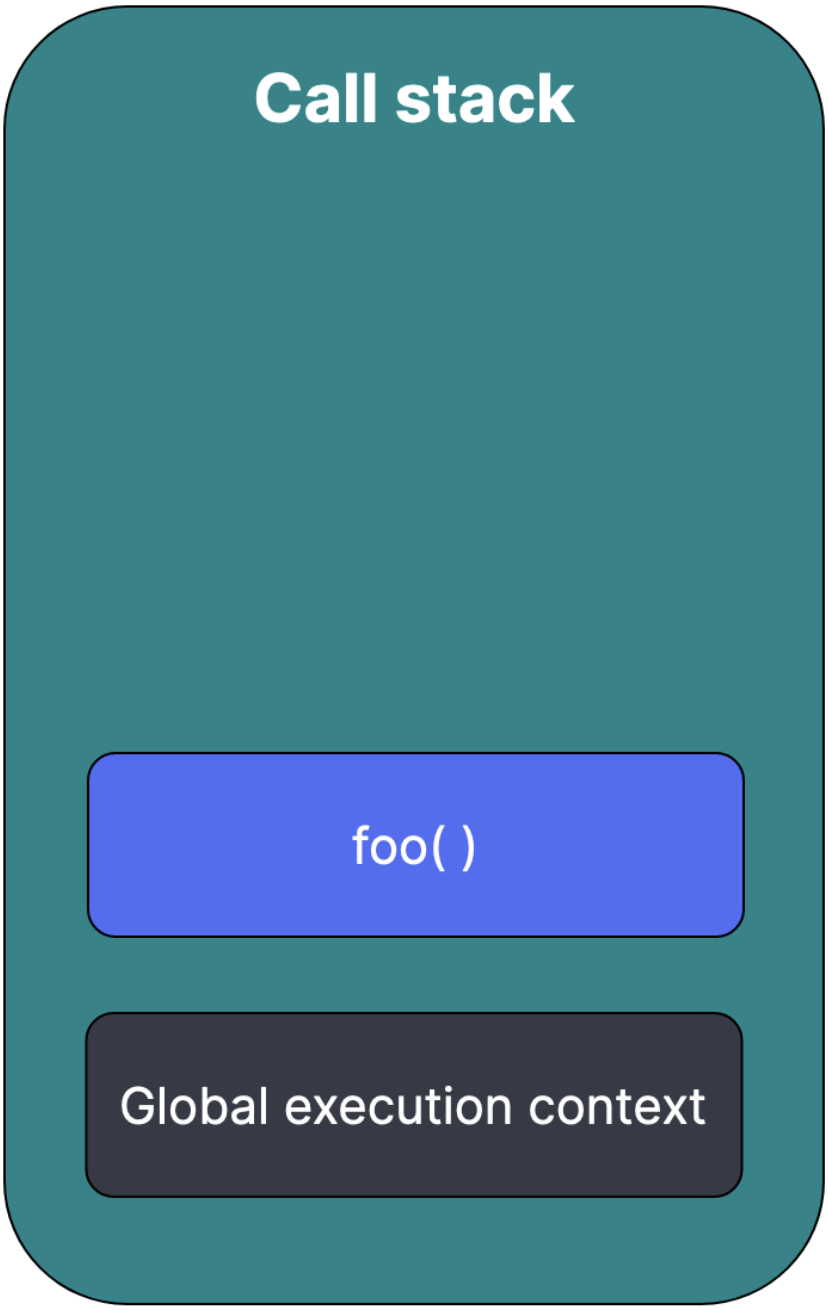
```
function bar() {
  console.log("hello world");
}

function baz() {
  bar();
}

function foo() {
  baz();
}

foo();
```

Before the execution of the above code, the global execution context is pushed on the call stack.



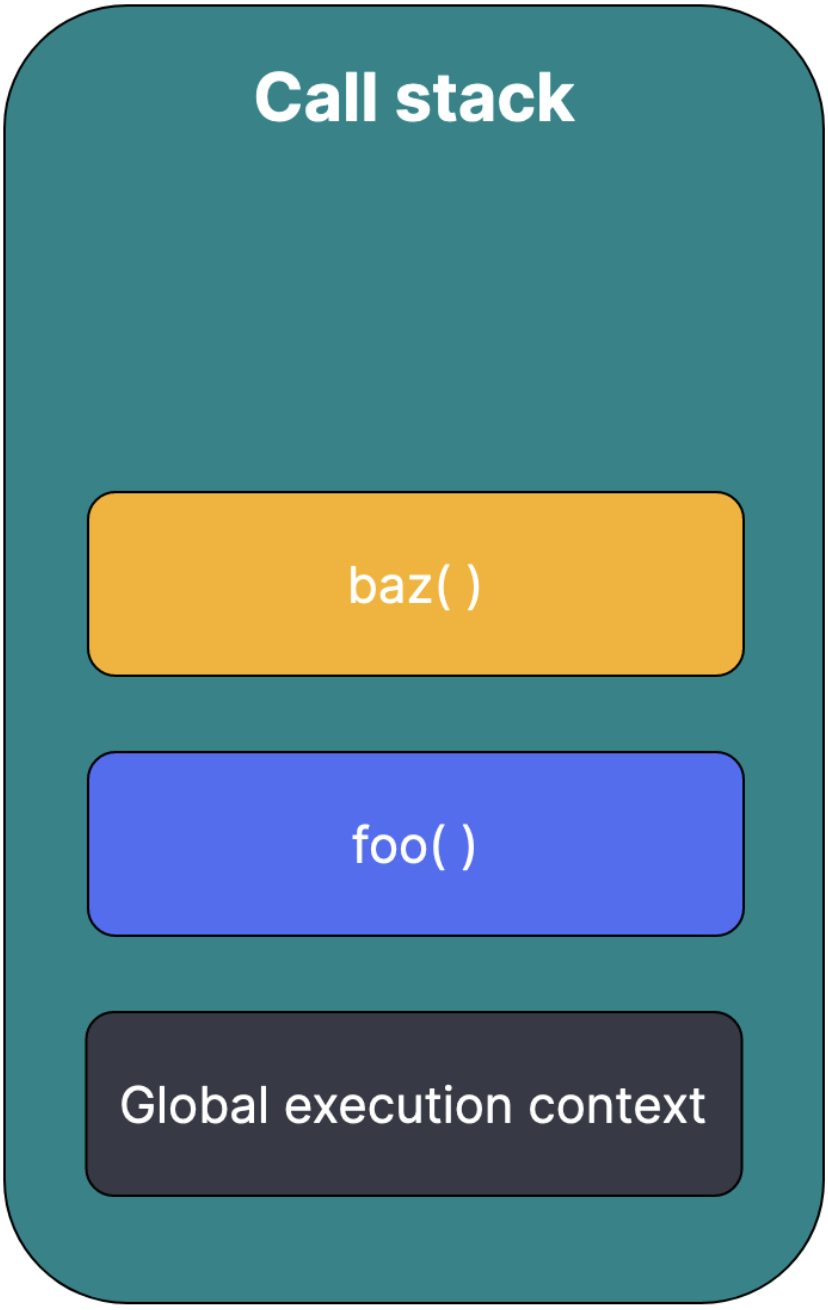
callstack

As soon as the `foo` function is called, a new entry is added to the call stack for the execution of the `foo` function.



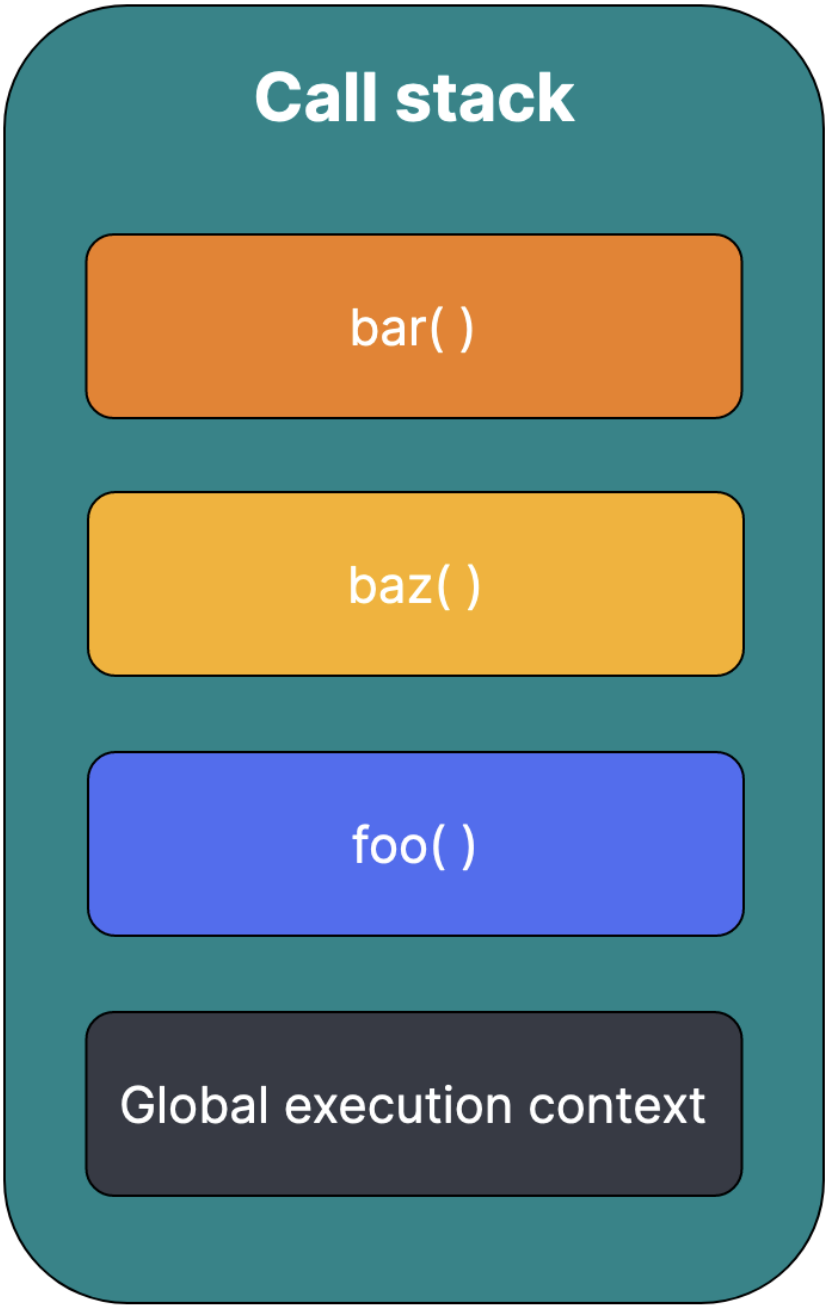
callstack

The `foo` function contains a call to the `baz` function. So, another entry is pushed on the call stack for the `baz` function call.



callstack

The `baz` function contains a call to the `bar` function. So, another entry is pushed on the call stack for the `bar` function call.



callstack

Note that the top element in the call stack represents the currently executing piece of code. As soon as the `bar` function execution ends, its entry in the call stack is removed. Ultimately, the code execution completes, and the call stack becomes empty.

You can run the code above in the Replit below to see the call stack in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/Lesson-9-callstack" />

Stack overflow

The term "stack overflow" is a familiar term for every software developer, either because of every software developer's favorite website stackoverflow or because of the stack overflow error due to infinite recursion.

We will be discussing the term "stack overflow" in the context of the call stack. The call stack has a fixed size and can contain a limited number of entries.

So what happens if we create a function that just calls itself?

```
function foo() {
  foo();
}
```

Remember what happens when a function is called? A new entry is added to the call stack. In the case of the above function that just calls itself and never finishes executing, we are just adding the new entries in the call stack without ever removing any entries. This is infinite recursion, and this leads to an error known as **stack overflow**. This error is thrown when the call stack gets filled up to its limit and can no longer hold more entries.

There are many resources on the internet that claim that primitives are allocated on the stack and objects are allocated on the heap, but the reality is not that simple.

The official ECMAScript specification doesn't state anything about how JavaScript engines should allocate memory for different types of values or how they should free up memory once it is no longer needed by the program. As a result, different JavaScript implementations are free to choose how they want to handle memory management in JavaScript.

So instead of simply believing that the primitive values are stored on the stack, and the objects are stored on the heap, we should understand that memory allocation in JavaScript is an implementation detail, and different JavaScript engines might handle memory differently because the language specification doesn't mandate how memory should be handled in JavaScript.

In the V8 engine, for example, almost everything is stored on the heap. The following quote from the official V8 blog invalidates the common misconception regarding memory allocation in JavaScript.

> JavaScript values in V8 are represented as objects and allocated on the V8 heap, no matter if they are objects, arrays, numbers, or strings. This allows us to represent any value as a pointer to an object.

This doesn't mean that we should assume that everything is allocated on the heap. JavaScript engines may allocate *most* values on the heap but could use the stack for optimization and store temporary values that might not last longer than a function call.

JavaScript engines are complicated softwares that are heavily optimized. It is unreasonable to assume that they all just follow the simple rule of *primitives go on the stack and objects on the heap*.

The most important point you should take away from this lesson is that different JavaScript engines may handle memory differently, and "primitives in javaScript simply go on the stack" is a **misconception**.

Further reading

- JavaScript memory model demystified - (blog)
- What are JavaScript variables made of - (blog)
- Where does Javascript allocate memory for the result of a function call? Stack or heap? - (stackoverflow post)

Automatic garbage collection

Unlike other languages like C, where the programmer is responsible for freeing up the memory when it is no longer needed, JavaScript makes the job of the programmer easier by automatically handling the memory.

Memory that is no longer needed is automatically freed by the JavaScript engine, and this process is known as **garbage collection**. Modern JavaScript engines include a garbage collector that is responsible for determining which parts of the memory are no longer needed and can be freed. Once such blocks of memory have been determined, those blocks are freed by the garbage collector.

Different algorithms are used to determine which blocks of memory are no longer needed and are eligible for garbage collection. Currently, modern JavaScript engines use a Mark-and-sweep algorithm.

This algorithm determines which blocks of memory are “unreachable”; such blocks of memory are considered eligible for garbage collection. This algorithm is an improvement over the reference counting algorithm, which has its limitations.

The Java language also has the mechanism of automatic garbage collection, but in Java, programmers can manually trigger the garbage collection process, whereas JavaScript programmers don’t have this level of control over garbage collection. Some might see this as a limitation, but there is no doubt that automatic garbage collection is really helpful for programmers to avoid memory leaks that are often encountered in languages that don’t handle this automatically.

Hoisting

In JavaScript, variables and functions can be accessed *before* their actual declaration, and the term used to describe this in JavaScript is "**Hoisting**."

"var" declarations

The term "hoisting" is mostly associated with function declarations and variables declared with the "var" keyword. Let's take a look at how *hoisting* is associated with the "var" variables.

Consider the following code example:

```
1 console.log(result); // undefined
2 var result = 5 + 10;
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/hoisting-example1" />

The output of the above code shows *hoisting* in action. The first line in the above code outputs `undefined` on the console, but how is this possible? How are we able to access the `result` variable *before* it is actually declared on the second line?

This is made possible because of the **parsing** step before the code is executed. The preprocessing of the JavaScript code before its execution allows the JavaScript engine to detect some errors early before any code is executed. The following code example shows this in action:

```
1 function print(obj) {
2   console.log(obj;    // error
3 }
4
5 console.log("hello world");
```

Here’s the Replit to run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/hoisting-example2” />

Without any preprocessing of the code, the JavaScript engine cannot detect the syntax error inside the `print` function unless the function is invoked or called, and the above code should log “hello world” on the console without throwing any errors because the function containing the syntax error hasn’t been called. But the above code throws an error instead of logging “hello world” on the console. Why is that? The answer is the processing step before the code execution.

The JavaScript engine scans the code before executing it, allowing it to detect some errors before any code is executed. This also enables the engine to handle variable declarations by registering the variables declared in the current scope. Before any scope starts, all the variables declared in that scope are registered for that scope. In other words, all the variables declared in a scope are reserved for that scope before the code in that scope is executed. This preprocessing of the code before its execution is what enables *hoisting*. This allows us to refer to “var” variables *before* they are actually declared in the code.

Let us revisit the code example given above that logs `undefined` to the console.

```
1 console.log(result); // undefined
2 var result = 5 + 10;
```

If the “var” variables are *hoisted*, and we can refer to them *before* their declaration, then why is `undefined` logged on the console instead of the actual value of the `result` variable, which should be 15?

The thing with the hoisting of the “var” variables is that only their **declaration** is hoisted, not their values. These variables are assigned the value of `undefined`, and the actual value, 15 in the above code example, is

assigned when their declaration is executed during the step-by-step execution of the code.

Function declarations

Function declarations, just like variables declared using the “var” keyword, are also hoisted. The following code example shows the hoisting of a function declaration in action:

```
startCar(); // starting the car...

function startCar() {
  console.log("starting the car...");
}
```

Here’s the Replit to run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/hoisting-example4” />

If “var” variables are assigned a value of `undefined` before their declaration is executed, how are we able to call the function *before* its declaration? How is the hoisting of function declarations different from the hoisting of variables declared using the `var` keyword? The difference here is that in the case of function declarations, the function’s name is registered as a variable in the scope containing the function declaration, and it is initialized with the function itself.

In the above code example, the `startCar` is registered as a variable in the global scope, and it is assigned the function. Unlike the “var” variables, there is no initialization with the `undefined` value in the case of function declarations.

It is hard to see how hoisting can be a useful feature for a programming language until we see the hoisting of function declarations. To be able to call a function before or after the function declaration is really useful and frees the developer from arranging the code in such a way that every

function declaration comes *before* it is called. This helps in code organization, as we can declare functions together either at the top or at the bottom of the code file and call them from anywhere we want in that file.

Function declarations inside blocks

For a long time, function declarations inside blocks weren't part of the ECMAScript specification, but that changed with the introduction of ES2015. Since the function declarations weren't part of the specification before 2015 and were allowed in the language by the JavaScript engines, they were handled differently by different engines.

In ES2015, the ECMAScript specification defined **standard** and **legacy** rules for handling function declarations.

Standard rules

According to the standard rules, the function declarations inside blocks are *hoisted* to the top of the block, converted into a function expression, and assigned to a variable declared with the `let` keyword.

The function hoisted inside the block is limited to the containing block and cannot be accessed by code outside the block containing the function.

:::note

It is important to note that the standard rules only come into effect in strict mode.

:::

The following code example will help you better understand the standard rules for function declarations inside blocks:

```
"use strict";

function process() {
```

```
  if (true) {
    console.log("processing...");

    function fnInsideBlock() {
      console.log("I am inside a block");
    }
  }
}
```

According to the standard rules, the function inside the `if` block should be treated as shown below:

```
"use strict";

function process() {
  if (true) {
    let fnInsideBlock = function fnInsideBlock() {
      console.log("I am inside a block");
    };

    console.log("processing...");
  }
}
```

The `fnInsideBlock` function in the above code can only be called from within the `if` block.

Legacy rules

The legacy rules are applied to the non-strict code in web browsers. According to the legacy rules, apart from a `let` variable for a function declaration inside a block, there is also a `var` variable in the containing function scope.

Let us take the code example given in the above section and add the function calls:

```
"use strict";

function process() {
  if (true) {
    console.log("processing...");
```

```
6
7     function fnInsideBlock() {
8       console.log("I am inside a block");
9     }
10  }
11
12  fnInsideBlock();
13 }
14
15 process(); // error
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/hoisting-example7” />

As the above code is executed in strict mode, the standard rules apply; as a result, the above code throws an error because, as explained in the above section, the function inside the block is hoisted to the top of the containing scope and is only accessible inside that block. As a result, the function call outside the `if` block throws an error.

If the `"use strict"` directive is removed from the above code, it will execute without any error. Why is that? This is because, in non-strict mode, the legacy rules for function declarations in a block apply, and according to the legacy rules, the hoisted function inside the block is assigned to a `var` variable that is declared in the containing function scope.

The above code, when executed in non-strict mode, is treated by the JavaScript engines as shown below:

```
1 function process() {
2   var fnInsideBlockVar;
3
4   if (true) {
5     let fnInsideBlock = function fnInsideBlock() {
6       console.log("I am inside a block");
7     };
8
9     console.log("processing...");
10
11    fnInsideBlockVar = fnInsideBlock;
```

```
  }

  fnInsideBlockVar();
}

process();
```

Here’s a Replit demonstrating the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/hoisting-example8” />

:::note

In the transformed code above, `fnInsideBlockVar` and `fnInsideBlock` are the same variables handled behind the scenes by the JavaScript engines. The names are shown to be different just for the sake of explaining how the function declarations in blocks are handled according to the legacy rules.

:::

Now, it should be clear why removing the “use strict” directive allows the code to be executed without any error. The hoisted function inside the block is assigned to a var variable defined in the function scope. As a result, the function inside the block is accessible outside the containing if block.

:::tip

The legacy rules are complicated and confusing; as a result, we shouldn’t rely on code that depends on the legacy rules for function declarations inside blocks. We should write code in strict mode to keep confusing complexities from entering our code.

:::

Further reading

- What are the precise semantics of block-level functions in ES6? - (stackoverflow post)
- Why does block assigned value change global variable? - (stackoverflow post)
- Function declaration in block moving temporary value outside of block? - (stackoverflow post)

Class declarations

Like function declarations, class declarations are also hoisted, but they are hoisted *differently* compared to the function declarations.

While we can access a function declaration *before* its declaration, we cannot do the same in the case of class declarations. Doesn't that mean that the class declarations aren't hoisted? No, they are hoisted, but *differently*.

Let us first verify that class declarations are indeed hoisted with the help of the following code example:

```
let Car = "Honda";

if (true) {
  console.log(typeof Car); // error

  class Car {}
}
```

Here's a Replit demonstrating the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/hoisting-example9" />

The above code throws an error that proves that the class declarations are indeed hoisted.

If the class declarations weren't hoisted, then the `console.log` function call should have logged "Honda" to the console, but that isn't the case, and that is because the class declaration inside the `if` block is hoisted and any code,

before or after the `Car` declaration inside the block, that accesses `Car` will access the class declaration and not the `Car` variable declared above the `if` statement. The fact that the identifier `Car` inside the `if` block refers to the class declaration and not the `Car` variable declared before the `if` statement proves that the class declarations are indeed hoisted.

So, if class declarations are hoisted, then why can't we access them *before* their declaration? The answer to this question is the "**Temporal Dead Zone (TDZ)**".

Temporal Dead Zone (TDZ)

Temporal Dead Zone (TDZ) refers to the time during which the block-scoped variables (`let`, `const`) or class declarations cannot be accessed. The time starts from the start of the scope till the declaration is executed. The following code example will help us visualize TDZ:

```
{
  // TDZ start
  console.log("hello");
  console.log("world");
  let count = 5; // TDZ end

  // can access "count" after TDZ ends
}
```

TDZ is the reason class declarations cannot be accessed before their declaration is executed during the step-by-step execution of the code.

let and const

As TDZ also applies to the `let` and `const`, are the variables declared using `let` or constants using `const` also hoisted? Yes, they are also hoisted, but, like the class declarations, they are hoisted *differently* because of the TDZ.

:::info

It is a common misconception that block-scoped variables and constants are not hoisted, but as we discussed above, they are hoisted; it's just that their hoisting is *different* as compared to the hoisting of variables declared using the `var` keyword.

:::

```
var count = 5;

{
  console.log(count); // hoisted but cannot access due to TDZ
  let count = 10;
}
```

You can see the above code example in action in the following Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/hoisting-example11" />

Function and class expressions

The function and class expression are not hoisted. Consider the following code example:

```
console.log(typeof Car); // undefined
console.log(typeof Person); // undefined

var Car = class {
  constructor(model) {
    this.model = model;
  }
};

var Person = function (name) {
  this.name = name;
};
```

Here's the above code example in action in Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/hoisting-example12" />

In the above code example, as the variables `Car` and `Person` have been declared using the `var` keyword, we can access them *before* their declaration, and their value are `undefined`. This is because only the declarations are hoisted, not their values. In the above code example, the values are expressions, and they are not hoisted.

If we try to create an instance of the `Car` class or call the `Person` constructor, we will get an error. It is worth noting that the error we get in this case is not a reference error (an error that is thrown in the case of undeclared identifiers) but a type error, and this is because `Car` and `Person` are hoisted and initialized with the value of `undefined`, and we cannot call an `undefined` as a constructor function. Recall that only the declarations are hoisted in this case, not their values.

Hoisting can be a confusing concept to wrap your head around, especially for beginners, because of the differences in which hoisting is associated with `var` variables, function declarations, and ES2015 changes, i.e., block-scoped variables/constants and ES2015 classes.

Common misconception about hoisting

Many JavaScript beginners have a misconception about the concept of hoisting, and that is that the JavaScript engine *moves* the hoisted declarations to the top of the file. Although this makes it easy to understand the concept of hoisting, that is not the reality.

JavaScript engines don’t move the hoisted declarations to the top of the file. Instead, they simply process the declarations before the step-by-step execution of the code. In the case of `var` variables, they are assigned the value of `undefined` until their declaration is executed. In the case of block-scoped variables, they are marked as “uninitialized”.

Scope

The scope is a general concept in the field of computer science that refers to the parts of the program where a particular variable, function, etc., can be accessed. In other words, the scope of an identifier (variable, function, etc.) is the part of a program where it is visible or can be referenced.

Modern JavaScript has four main types of scopes that are mentioned below:

- Global scope
- Function scope
- Block scope
- Module scope

Before discussing the above-mentioned types of scopes in JavaScript, we need to understand what “lexical” scope is.

Lexical scope

In JavaScript, the scope of different identifiers (variables, functions, etc.) is determined at **compile time**. During compilation, the JavaScript engines determine the scope of different identifiers declared in the code by analyzing the code structure. This means that before the step-by-step execution of the JavaScript code starts, JavaScript engines determine the scopes of different declarations in the code.

Scopes can be nested within other scopes, with each nested scope having access to the outer or parent scope.

```
1 const myName = "John doe";
2
3 function hello() {
4   const greeting = "hello " + myName;
5   console.log(greeting);
6 }
```

In the above code example, there are three different declarations:

- `myName` variable declaration
- `hello` function declaration
    - `greeting` variable declaration

The scope of the above declarations depends on where they are written in the code structure above. The `myName` variable and `hello` function are both in global scope, so they are available globally in the above code. The `greeting` variable declaration is inside the `hello` function, so its scope is local to the `hello` function.

This type of scope, which is determined at compile time by analyzing the code structure, is known as lexical scope. JavaScript is not the only language that has a lexical scope. Other languages, like Java, also have a lexical scope.

:::info

Lexical scope is also known as “static” scope. An alternative type of scope is Dynamic scope.

:::

The global scope is generally the outermost scope that contains all other scopes nested inside it. Each nested scope has access to the global scope. In JavaScript, the global scope is the browser window or, more accurately, a browser window tab. The global scope is exposed to the JavaScript code using the `window` object.

Variables created using the `var` keyword or function declarations declared in the global scope are added as properties on the `window` object. The following code example verifies this claim:

```
var todoList = ["grocery", "exercise", "meeting"];

function emptyTodoList() {
  todoList = [];
}

```

```
console.log(window.hasOwnProperty("todoList")); // true
console.log(window.hasOwnProperty("emptyTodoList")); // true
```

If the `todoList` was declared with `let` or `const`, it wouldn't have been added as a property on the `window` object, but it would still be a global variable. Similarly, if the `emptyTodoList` was a function expression instead of a function declaration, and if the identifier referring to this function expression was declared using `let` or `const`, it also wouldn't be part of the `window` object as its property. Instead, it would just be a global function expression.

```
const todoList = ["grocery", "exercise", "meeting"];

let emptyTodoList = function () {
  todoList = [];
};

console.log(window.hasOwnProperty("todoList")); // false
console.log(window.hasOwnProperty("emptyTodoList")); // false
```

Avoid polluting the global scope

"Avoid polluting the global scope" - As a JavaScript developer, either you have already heard this advice, or sooner or later, you will hear it from someone. But what does "polluting" the global scope even mean? Why should it be avoided?

Declaring everything (unnecessarily) in the global scope is what's considered as "polluting" the global scope. Too many declarations in the global scope can lead to unwanted problems.

The global scope is the parent scope of all other scopes; as a result, the declarations inside the global scope are visible to all other scopes. This can cause problems like variable name clashes, shadowing, etc. Another thing about the global scope is that it isn't destroyed until the application is closed, so if we are not careful, declarations in the global scope can remain in memory regardless of whether they are needed or not until the application is closed.

Having said all that, declarations in the global scope are typically unavoidable. So, the best we can do is avoid the global scope as much as possible. Keep the global declarations to a minimum. If a variable is only used inside a function, there is no point in declaring it in the global scope.

So, if you hadn't heard it before, you are hearing it now: *"Avoid polluting the global scope."*

Implicit globals

JavaScript as a language has many quirks. One of those is the implicit creation of global variables. Whenever there is an assignment to an *undeclared* variable, JavaScript will declare that undeclared variable as a **global** variable. This is most likely a mistake by the programmer, and instead of throwing an error, javaScript hides this by automatically declaring a global variable by the same name.

```
function printSquare(num) {
  result = num * num;
  console.log(result); // 64
}

printSquare(8);

console.log("implicit global: " + result); // WHAT??!!
```

Here's the above code in action in Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/global-scope-example3" />

Notice that the `result` variable is not declared. It is assigned the result of multiplication as though it has already been declared. One would normally expect an error in this case, but JavaScript will declare the `result` as a global variable for you.

It is important to mention that this weird behavior only happens in non-strict mode. In strict mode, as expected, JavaScript throws an error,

informing the programmer about the assignment to an undeclared variable.

This is also one of the reasons always to write JavaScript code in strict mode. Using the strict mode keeps such confusing behaviors of JavaScript away from our code.

HTML attributes

Apart from the assignment to undeclared variables, there is another way we get *implicit* global variables. The value of the `id` attribute or the `name` attribute of HTML elements also gets added as a variable in the global scope of JavaScript.

```
1 <h1 id="mainHeading">Hello World!</h1>
```

The `id` of the `h1` element above gets added to the global scope as a variable. We can access the `h1` element using the `mainHeading` as a variable in JavaScript code. This feature is referred to as Named access on the Window object.

This was first added by the Internet Explorer browser and was gradually implemented in other browsers as well, simply for compatibility reasons. There are sites out there that rely on this behavior, so for the sake of backward compatibility, browsers have no choice but to implement this behavior.

Although this is supported by most browsers, this feature shouldn't be relied upon, and we should always use standard mechanisms to target HTML elements. Functions like `getElementById` or `querySelector` should be used instead of relying on this behavior.

Writing code that relies on this feature is a bad idea because it can result in code that is hard to read and maintain. Imagine seeing an identifier in the code and not being able to identify where it is declared. Such code is vulnerable to name clashes with other variables in our code. Also, keep in mind that writing code that depends on the HTML markup can break easily

if the HTML markup is changed. In short, avoid relying on this feature and use better alternatives for targeting HTML elements in your JavaScript code.

Further reading

- Global Variables are Bad - (article)
- Do DOM tree elements with IDs become global properties? - (stackoverflow post)

The function scope refers to the area of the code within the function body. The function scope starts from the opening curly parenthesis of the function body and ends before the closing curly parenthesis. Declarations inside the function scope are limited to that function's scope and cannot be directly accessed by the code outside that function.

Shadowing declarations

Declarations inside a nested scope can "shadow" the declarations with the *same* name in the outer scope. This is referred to as "shadowing declaration" or simply "shadowing."

Consider the following code example:

```
let hobby = "reading";

function printHobbies() {
  const hobby = "traveling";
  console.log(hobby); // traveling
}

printHobbies();
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/function-scope-example1">

The variable `hobby` inside the function is shadowing the `hobby` variable declared in the global scope.

It is generally not ideal to shadow other declarations because that can reduce the readability of the code. It could also make it impossible for code inside the nested scope to access the shadowed declarations. Consider the following code example:

```
let prefix = ">";

function log(logLevel, msg) {
  let prefix = ":::";
  console.log(`${prefix} ${logLevel} : ${msg}`);
}

log("debug", "error caught"); // ::: debug : error caught
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/function-scope-example2">

In the code example above, we have shadowed the `prefix` variable declared in the global scope, and the code inside the `log` function is unable to access the global `prefix` variable.

:::note

Recall that the `var` declarations in the global scope are added as properties on the `window` object. So, if the global `prefix` variable was declared using the `var` keyword, we could have used `window.prefix` to access the shadowed variable.

:::

Function parameter scope

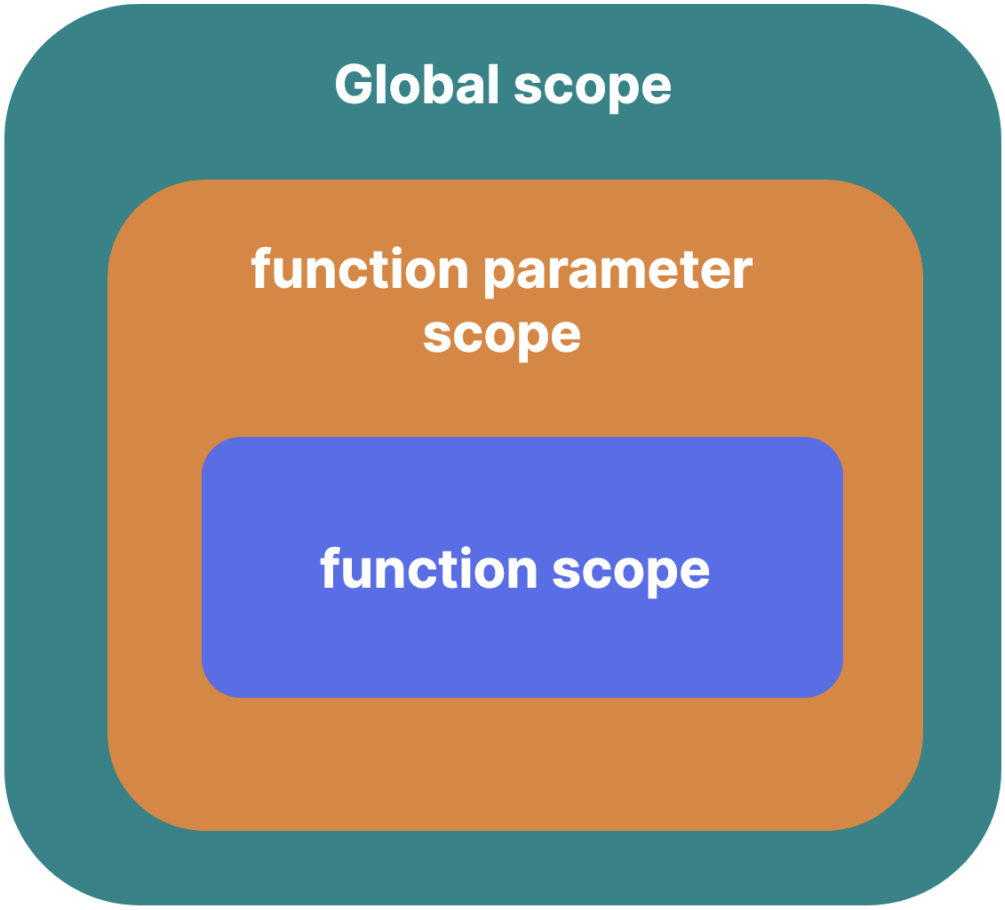
It is a common misconception that the function parameters are defined in the function's local scope or that the parameters behave as if they are defined in the function's local scope, but that is not always true.

First, we need to differentiate between "simple" and "non-simple" parameter lists. If the function parameters are defined in such a way that they use ES2015+ features like Default parameters, Destructuring, or Rest parameters, such parameters are considered to be **non-simple** parameters; if the parameters don't use any of these features, they are considered to be **simple** parameters.

```
function simpleParameters(val1, val2) {
  // code...
}

function nonSimpleParameters(val1 = 12, ...restParams) {
  // code...
}
```

If the parameters are simple, they behave like they are declared in the function's local scope, but if the parameters are non-simple, they are declared in their own scope. Non-simple parameter scope can be thought of as between the function scope and the scope containing the function.

If the function with non-simple parameters is defined in the global scope, its parameter scope can be conceptually visualized as shown in the image below:



function parameter scope

The following code example proves that the non-simple parameters indeed exist in a different scope than the function's local scope:

```
function paramScope(arr = ["initial array"], buff = () => arr) {
  var arr = [1, 2, 3];
  console.log(arr); // [1, 2, 3]
  console.log(buff()); // ["initial array"]
}

paramScope();
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/function-scope-example4">

The `paramScope` function in the code example above has a non-simple parameter list; the first parameter, `arr` has an array as a default value, whereas the second parameter, `buff` has a function as its default value.

Inside the function, there is a `var` declaration with the same name as the first parameter of the `paramScope` function. The two `console.log` calls inside the function log two different values to the console; why is that? Why does the `buff` parameter return the default value of the `arr` parameter and not the value of the `arr` inside the local scope of the function?

The answer is that the `arr` parameter and the `arr` variable inside the function are two *different* variables that exist in two *different* scopes. The `arr` inside the function shadows the `arr` parameter, but calling the `buff` function returns the parameter `arr`. If the parameter and the local `arr` were the same variable, the `buff` function would return `[1, 2, 3]` instead of the default value of the `arr` parameter.

Remove the `var` keyword inside the function to show the different output:

```
function paramScope(arr = ["initial array"], buff = () => arr) {
  arr = [1, 2, 3];
  console.log(arr); // [1, 2, 3]
  console.log(buff()); // [1, 2, 3]
}

paramScope();
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/function-scope-example5">

The `arr` inside the function now refers to the `arr` parameter, so any assignment to `arr` inside the function is reflected when the `buff` function is called.

Function expression name scope

The function expressions are mostly written as an anonymous function expression, as shown below:

```
let fn = function () {
  // code ...
};
```

This is an anonymous function expression that is assigned to the `fn` variable.

We can also write a *named* function expression as shown below:

```
let fn = function namedFn() {
  // code ...
};
```

In the code example above, the name of the function expression `namedFn` is only accessible inside the function body. As a result, some might incorrectly believe that the name of a named function expression is declared inside the function body, but that is not correct; the name is declared in a *different* scope. The following code proves this claim:

```
let fn = function namedFn() {
  let namedFn = 123;
  console.log(namedFn);
};
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/function-scope-example8”>

The `let` doesn’t allow the re-declaration of a variable. So if the `nameFn` was declared inside the function scope, then the code example above should have thrown an error; instead, there is no error, and this is valid code. The `nameFn` inside the function body is actually shadowing the name of the function expression.

Named function expression's name scope is nested between the scope containing the function expression and the function expression's local scope, similar to the scope of the non-simple parameter list discussed above.

Further reading

- Where are arguments positioned in the lexical environment? - (stackoverflow post)
- Implied scopes
- Named function expression - (MDN)

Block Scope

A block scope in JavaScript refers to the scope that exists between blocks of code, such as `if` blocks or loops.

Prior to the addition of block-scoped `let` and `const` in the JavaScript language, variables defined with the `var` keyword in a block were accessible outside that block. This is because the variables declared with the `var` keyword have function scope. However, the variables declared using `let`, or the constants declared using `const` are scoped to the block in which they are defined unless they are declared in the global scope, which makes them global variables.

The block-scoped `let` and `const` solve problems like unnecessary exposure of variables outside blocks, closure inside loops, etc. We will discuss the closure inside loops problem in a module related to the topic of closure.

Another thing to know about the block scope is how function declarations inside blocks are handled. This was discussed in a module related to hoisting.

Module Scope

Modules are also among the features that were added in recent years to JavaScript. Modules solve many problems related to code management that existed in code bases that involved multiple JavaScript files. Modules allow us to split our code into manageable chunks.

The code inside an ES module exists in the module scope. In other words, the declarations inside a module are scoped to the module and aren't exposed outside of the module, except the code that is *explicitly* exported from the module. Declarations at the top level of a module are limited to the module and aren't part of the global scope.

Further reading

- JavaScript Modules - (MDN)
- ES modules: A cartoon deep-dive - (MDN)

Scope Chain

Different scopes can be nested inside other scopes, creating a chain of scopes. This is known as a **scope chain**.

Every time a new scope is created, it is linked to its enclosing scope. This linkage between different scopes creates a chain of scopes that can then be used for lookups for identifiers that cannot be found in the current scope.

When an identifier is encountered in a particular scope, the JavaScript engine will look for the declaration of that variable in the parent scope of the current scope. If the declaration is not found in the parent scope, then the JavaScript engine will look for that variable declaration in the outer (parent) scope of the parent scope. This process of traversing the scope chain will continue until the global scope is reached and there are no other scopes to look into for the declaration.

Consider the following code example:

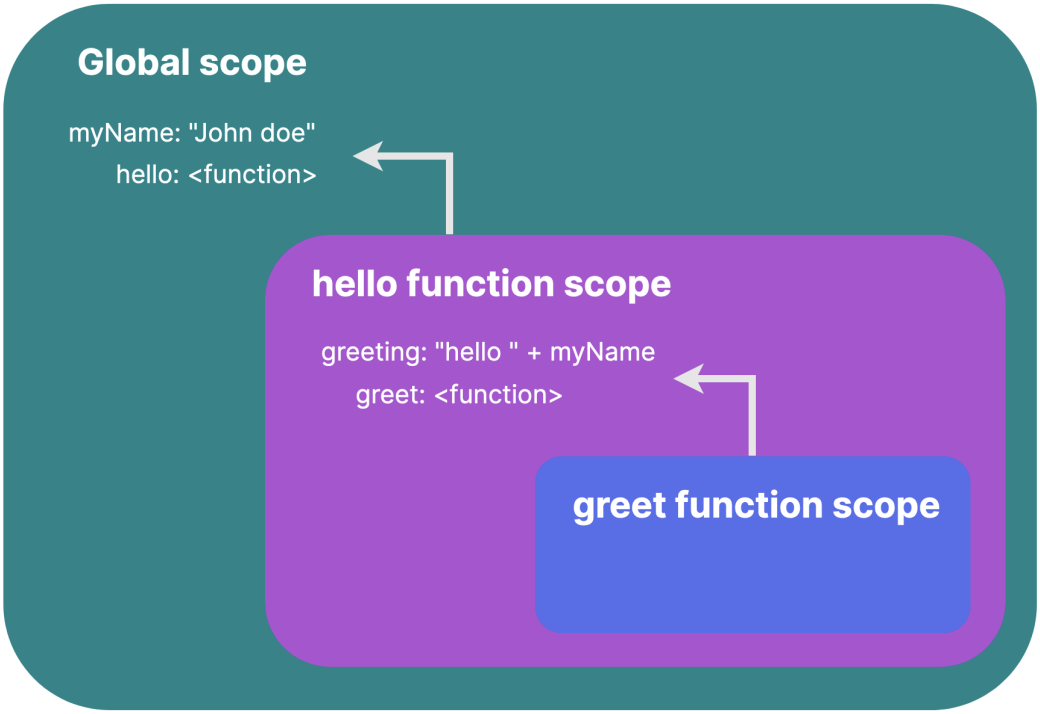
```
1 const myName = "John doe";
2
```

```
function hello() {
  const greeting = "hello " + myName;

  function greet() {
    console.log(greeting);
  }

  greet();
}

hello();
```

You can preview the code above in this Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/scope-chain-example1">

The scope chain created as a result of the `hello` function call can be conceptually visualized as shown below:



scope chain

The scope chain enables the lookup of identifiers that are not declared in the current scope.

Optimizing scope chain lookup

Modern JavaScript apps contain many lines of JavaScript code that must be compiled and executed in the shortest amount of time possible to ensure that app users do not experience poor app performance. As a result, modern JavaScript engines heavily optimize the JavaScript code we write to allow it to execute as efficiently as possible. Different optimizations are performed on the code while it is being transformed before it is executed. Optimizations are not only limited to the steps *before* the code execution but code is also optimized during its execution (recall the JIT compilation).

As the scope in JavaScript is determined at compile time, in most cases, this information allows the JavaScript engine to avoid having to traverse the scope chain during the code execution. If the JavaScript already knows what scope a particular variable is defined in, it is highly inefficient to traverse the scope chain every time a particular variable declaration cannot be found in the current scope.

So unless the scope of a variable cannot be determined at compile time, JavaScript engines don’t need to traverse the scope chain during runtime. However, there might be cases where the scope of a particular identifier cannot be determined at compile time; in these cases, the JavaScript engine has no choice but to traverse the scope chain at runtime to determine what scope that identifier is declared in.

Further reading

- The Scope Chain

Coercion

Coercion in JavaScript is the conversion of one type of value into another type of value.

MDN defines coercion as:

> Type coercion is the automatic or implicit conversion of values from one data type to another (such as strings to numbers).

According to MDN's definition, if the conversion of values is *implicit*, then it is *coercion*, whereas *type conversion* can either be *implicit* or *explicit*.

So, if the developer expresses the intention to convert one type of value into another, it is just type conversion. The following code example is an example of an *explicit* type conversion:

```
1 const age = Number(ageStr);
```

And if the type conversion is implicit, where the developer expressed no intention to convert any value into another value, then it is implicit type conversion or coercion. The following code is an example of coercion:

```
1 const result = "50" - 20; // 30
```

Whenever JavaScript sees a value of one type in a context that expects a value of a different type, it tries to coerce or convert the value into the expected type. In the above code example, `"50"` is the unexpected value type because the operation is subtraction. The subtraction is between numbers, not between a string and a number. So `"50"`, being an unexpected value, gets converted into a number, i.e. `50`.

However, some might argue that any type of conversion in a dynamically typed language can be considered coercion. Moreover, the difference

between *implicit* and *explicit* type conversion depends on how one views *implicit* and *explicit* type conversion.

Having said that, what's important is that we understand the process of type conversion in JavaScript. The goal of this module is to help you understand how JavaScript converts one type of value into another and make type conversion (implicit or explicit) less scary.

Coercion is one of those topics that many JavaScript developers generally misunderstand, and this is because most online resources advise staying away from coercion and presenting it as a topic that should be avoided instead of taking the time to understand it and take advantage of it where possible.

What makes coercion scary for many JavaScript developers, especially beginners, is the need for more understanding of this topic. Coercion is presented as a feature of JavaScript that is better avoided than understood.

Coercion is one of the core topics of JavaScript, and its understanding is key to understanding JavaScript in depth. No matter how many online resources tell you to avoid this topic, it is unavoidable if you work with JavaScript. Instead of avoiding it, why not make an effort to understand it? With this in mind, in this module, we will take a deeper look at this topic, and hopefully, by the end of this module, you will have a solid understanding of type coercion.

Coercion is one of the core topics of JavaScript, and its understanding is key to understanding JavaScript in depth. No matter how many online resources tell you to avoid this topic, it is unavoidable if you work with JavaScript. So, instead of avoiding it, why not try to understand it? With this in mind, in this module, we will take a deeper look at this topic, and hopefully, by the end of this module, you will have a solid understanding of type coercion.

To understand coercion in-depth, we need to understand how JavaScript goes about converting one value into another type of value. We need to

understand what algorithms or steps JavaScript takes to perform type conversion.

To deep dive into the world of coercion, let us understand the following:

- Abstract operations
- Abstract equality operator (==)
- Addition operation (+)
- Relational operators (<, >, <=, >=)

Understanding the above-mentioned topics will help us lay the foundation for understanding the process of type conversion in JavaScript. So, without further ado, let us begin by understanding the first item on our list, i.e., abstract operations.

The ECMAScript specification has documented several mechanisms that are used by the JavaScript language to convert one type of value into another type of value. These mechanisms are known as "**abstract operations**"; abstract in the sense that these are not some real functions that can be referred to or called by the JavaScript code; instead, they are just algorithmic steps internally used by the language to perform type conversion.

These abstract operations are written in the specification as though they were actual functions. For example, `operationName(arg1, arg2, ...)`, but the specification clarifies that the abstract operations are algorithms rather than actual functions that can be invoked.

There are many abstract operations mentioned in the ECMAScript specification, but some of the common ones that come into play when dealing with coercion are mentioned below:

- ToPrimitive
- ToNumber
- ToString
- ToBoolean

Although the names of the above-mentioned abstract operations are self-descriptive, let us understand how exactly they aid in type conversion.

ToPrimitive

The ToPrimitive abstract operation is used to convert an object to a primitive value. This operation takes two arguments:

- `input`: an object that should be converted into a primitive value
- `preferredType`: an optional second argument that specifies the type that should be favored when converting an object into a primitive value

OrdinaryToPrimitive

This operation invokes another abstract operation known as OrdinaryToPrimitive to do the actual conversion, and it also takes two arguments:

- `O`: an object that should be converted into a primitive value
- `hint`: a type that should be favored when converting an object to a primitive value

`ToPrimitive` abstract operation invokes the `OrdinaryToPrimitive` abstract operation, passing in the object, that is to be converted into a primitive value, as the first argument, and the second argument hint is set based on the value of `preferredType` argument as described below:

- If the `preferredType` is “string”, set `hint` to string
- If the `preferredType` is a “number”, set `hint` to the number
- If `preferredType` is not specified, set `hint` to the number

Each object in JavaScript inherits the following two methods from the object that sits at the top of the inheritance hierarchy, i.e., the Object.prototype object:

- `toString()`
- `valueOf()`

toString( )

The `toString` method is used to convert an object into its string representation. The default behavior of the `toString` method is to convert objects in the following (not so-useful) form:

```
1 const obj = { a: 123 };
2 obj.toString(); // [object Object]
```

Here's the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/abstract-operations-example1" />

As the default implementation of the `toString` method is not useful at all, different objects override this method to make its output more useful. The built-in `Date` object, for example, when converted to a string, outputs a human-readable string representation of the date:

```
1 new Date().toString(); // Sat Feb 04 2023 20:44:23 GMT+0500
```

Here's the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/abstract-operations-example2" />

valueOf( )

The `valueOf` method is used to convert an object into a primitive value. The default implementation of this method, like the `toString` method, is useless, as it just returns the object on which this method is called.

```
1 const arr = [];
2 arr.valueOf() === arr; // true
```

Here’s the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example3” />

It is meant to be overridden by other objects. Many built-in objects override this method. For example, for the `Date` objects, this method returns the number of milliseconds since midnight 01 January 1, 1970 UTC.

```
// number of milliseconds will be different for you,
// depending on when you execute the code below
new Date().valueOf(); // 1675526929129
```

Here’s the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example4” />

The `OrdinaryToPrimitive` abstract operation invokes the `toString` and the `valueOf` methods to convert an object into a primitive value. Among these two methods, in some cases, only one of them is called; in other cases, both of them are called.

The `hint` argument received by the `OrdinaryToPrimitive` abstract operation determines which of these two methods is called first.

Prefer string

If the `hint` argument is “string”, then the `OrdinaryToPrimitive` abstract operation first invokes the `toString` method on the object. If the `toString` method returns a primitive value, *even if that primitive value is not of the string type*, then that primitive value is used as a primitive representation of the object.

If the `toString` method doesn’t exist or doesn’t return a primitive value, then the `valueOf` method is invoked. If the `valueOf` method returns a

primitive value, then that value is used; otherwise, a `TypeError` is thrown, indicating that the object couldn't be converted to a primitive value.

```
const obj = {
  toString() {
    console.log("toString invoked");
    return "hello world";
  },
  valueOf() {
    console.log("valueOf invoked");
    return 123;
  }
};

console.log(`${obj}`);
// toString invoked
// hello world
```

Here's the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/abstract-operations-example5" />

In the above code example, we have an object containing overridden implementations of the `toString` and the `valueOf` method. At the end of the code example, we are trying to log `obj`, embedded in a template literal, to the console. In this case, the `obj` will be converted into a string.

As discussed above, when the `hint` argument of the `OrdinaryToPrimitive` abstract operation is "string", the `toString` method is invoked to convert an object into a primitive value, preferably into a value of string type.

As the `toString` implementation of the `obj` object is returning a string primitive value, the `valueOf` method is not invoked, and the object-to-primitive conversion process for the `obj` object is complete at this point. The primitive value returned by the `toString` method of the `obj` object is used by the template literal.

But it was mentioned above that the value returned by the `toString` method can be of non-string primitive type. The following code example verifies

this claim:

```
const obj = {
  toString() {
    console.log("toString invoked");
    return true;
  },
  valueOf() {
    console.log("valueOf invoked");
    return 123;
  }
};

console.log(`${obj}`);
// toString invoked
// true
```

Here's the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/abstract-operations-example6" />

The `toString` method in the above code example returns a boolean (non-string) primitive value. Instead of invoking the `valueOf` method or converting the non-string return value of the `toString` method into a string value, the boolean value is accepted as the primitive representation of the `obj` object.

The next case we need to verify is what happens if the `toString` method doesn't return a primitive value. The following code example demonstrates this case:

```
const obj = {
  toString() {
    console.log("toString invoked");
    return [];
  },
  valueOf() {
    console.log("valueOf invoked");
    return 123;
  }
};

```

```
console.log(`${obj}`);
// toString invoked
// valueOf invoked
// 123
```

Here’s the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example7” />

As explained earlier, if the `toString` method doesn’t return a primitive value, the `valueOf` method will be invoked to get a primitive representation of the object. In the code example above, the `toString` method returns an empty array of an object type; as a result, the `valueOf` method is invoked.

The `valueOf` method is invoked even if the `toString` is not defined for an object. The following code example shows this behavior:

```
const obj = {
  toString: undefined,
  valueOf() {
    console.log("valueOf invoked");
    return 123;
  }
};

console.log(`${obj}`);
// valueOf invoked
// 123
```

Here’s the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example8” />

The last case we need to verify is what happens when JavaScript can’t get a primitive value, even after invoking the `toString` and the `valueOf` method.

```
const obj = {
  toString() {
    console.log("toString invoked");
```

```
    return [];
  },
  valueOf() {
    console.log("valueOf invoked");
    return [];
  }
};

console.log(`${obj}`);
// toString invoked
// valueOf invoked
// TypeError ...
```

Here’s the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example9” />

When JavaScript can’t get a primitive value after invoking both methods, a `TypeError` is thrown, indicating that the object couldn’t be converted into a primitive value. So, it is important to remember when overriding these methods that at least one of them should return a primitive value.

We have discussed what happens when the preferred type is a string in the object-to-primitive conversion process. Next, let’s discuss what happens when the preferred type is a number.

Prefer number

If the hint argument is “number”, then the `OrdinaryToPrimitive` abstract operation first invokes the `valueOf` method and then the `toString` method, if needed.

This is similar to the “prefer string” case discussed above, except that the order in which the `valueOf` and the `toString` methods are invoked is the opposite.

```
const obj = {
  toString() {
    console.log("toString invoked");
```

```
    return "hello";
  },
  valueOf() {
    console.log("valueOf invoked");
    return 123;
  }
};

console.log(obj + 1);
// valueOf invoked
// 124
```

Here’s the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example10” />

This code example above is the same as the one shown in the section above, except that instead of embedding `obj` in a template literal and logging it to the console, we are adding 1 to `obj`. We are using `obj` as if it were a number.

So when JavaScript gets an object in a context where it expects a number, it tries to coerce the object into a primitive type, preferably into a number type.

In this case, the `hint` argument passed to the `OrdinaryToPrimitive` abstract operation is “number”; as a result, the `valueOf` method is invoked first. Since it returned a primitive value, there is no need to invoke the `toString` method.

The rest of the cases are the same as discussed in the “prefer string” section. The only difference is that the `valueOf` method is invoked first when the preferred type is “number”.

What will happen if the `valueOf` method returns a boolean value? It is a primitive value. It is not a number but still a primitive value. So JavaScript should accept it as a primitive representation of the `obj` object, right?

Consider the following code example:

```
const obj = {
  valueOf() {
    console.log("valueOf invoked");
    return true;
  }
};

console.log(obj + 1);
// valueOf invoked
// 2
```

Here's the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/abstract-operations-example11" />

Why did we get two as an output? The answer is that `true` is accepted as a primitive representation of the `obj` object, but we cannot add `true` and `1`. JavaScript expects a number in this context. So, it tries to coerce `true` into the expected type of value, which in this case is `1`. If the `valueOf` method had returned `false`, it would have been coerced to `0`.

No preference

When the `ToPrimitive` abstract operation is called without the preferred type or hint, or if the hint is set to "default", then this operation generally behaves as if the hint were "number". So, by default, the `ToPrimitive` prefers the conversion to number type.

However, the objects can override this default `ToPrimitive` behavior by implementing the Symbol.toPrimitive function. This function is passed a preferred type as an argument, and it returns the primitive representation of the object.

```
const obj = {
  [Symbol.toPrimitive](hint) {
    if (hint === "string") {
```

```
      return "hello";
    } else {
      return 123;
    }
  }
};

console.log(`${obj}`); // hello
console.log(obj + 1); // 124
```

Here’s the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example12” />

Out of the built-in objects, only the `Date` and `Symbol` objects override the default behavior of the `ToPrimitive` abstract operation. The `Date` objects implement the default behavior as if the preferred type or hint is “string.”

```
new Date()[Symbol.toPrimitive]("default"); // Tue Feb 07 2023 23:47:42 GMT+0500
```

Here’s the above code in action:

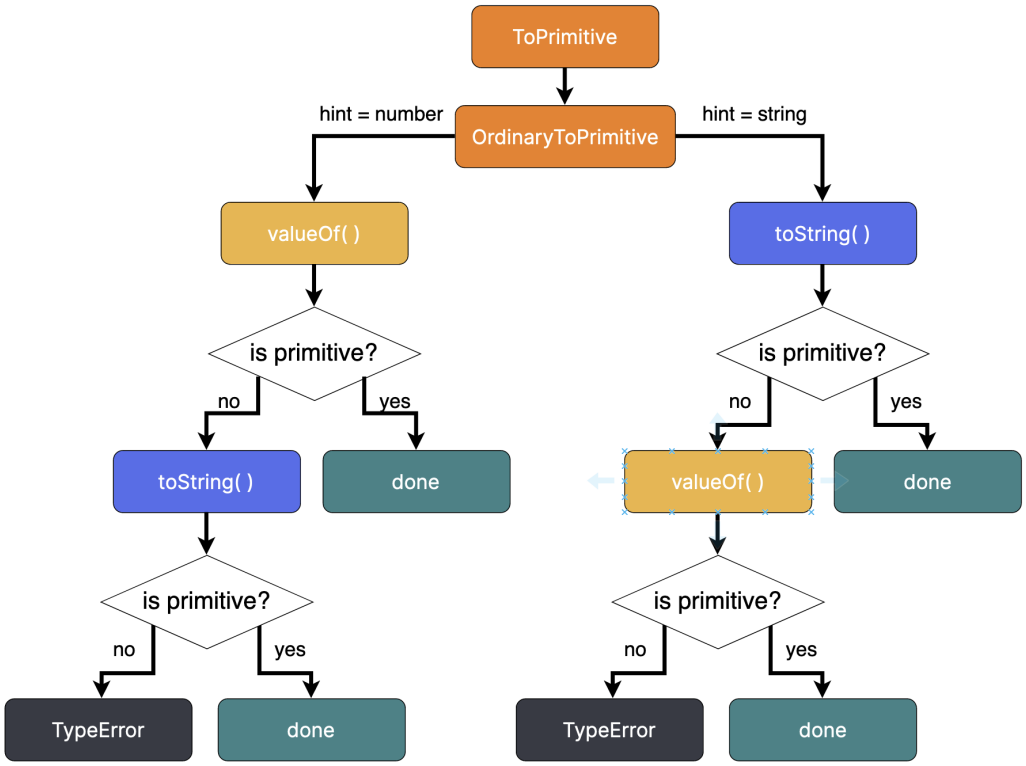
<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example13” />

:::note

We rarely need to call the `Symbol.toPrimitive` function explicitly. JavaScript calls this function automatically when it needs to convert an object into a primitive value.

:::

The `ToPrimitive` abstract operation is summarized in the image below:



Object to primitive conversion summarized

ToNumber

The ToNumber abstract operation is used whenever JavaScript needs to convert any non-number value into a number.

The following table shows the results of this abstract operation applied to some non-number values:

Value	ToNumber(value)
“”	0
“0”	0
“-0”	-0
” 123 “	123
“45”	45
“abc”	NaN
false	0
true	1
undefined	NaN
null	0

As far as objects are concerned, the `ToNumber` abstract operation first converts the object into a primitive value using the `ToPrimitive` abstract operation with “number” as the preferred type, and then the resulting value is converted into a number.

The BigInt values allow *explicit* conversion into a number type, but the *implicit* conversion is not allowed; implicit conversion throws a `TypeError`.

```
console.log(Number(10n)); // 10

console.log(+10n); // TypeError...
```

Here’s the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/abstract-operations-example14” />

ToString

The ToString abstract operation is used to convert any value into a string.

The following table shows the results of this abstract operation applied to some non-string values:

Value	ToNumber(value)
null	“null”
undefined	“undefined”
0	“0”
-0	“0”
true	“true”
false	“false”
123	“123”
NaN	“NaN”

In the case of objects, the `ToString` abstract operation first converts the object into a primitive value using the `ToPrimitive` abstract operation with “string” as the preferred type, and then the resulting value is converted into a string.

ToBoolean

The ToBoolean abstract operation is used to convert a value into a boolean value.

Unlike the above-mentioned abstract operations, this operation is simply a lookup of whether a value is a falsy value. If it is, we get `false` as a return value; for all other values, this operation returns `true`.

The following is a list of falsy values:

- false
- 0, -0, 0n
- undefined
- null
- NaN
- ””

As mentioned earlier, there are many abstract operations mentioned in the ECMAScript specification that are used for type conversion; in this lesson,

we have only discussed the common ones.

Further reading

- Number coercion - (MDN)
- String coercion - (MDN)
- Boolean coercion - (MDN)
- Coercing values - (You Don't Know JS Yet)

Let's discuss the infamous "double equal" operator that is used for *loosely* comparing two values. It is also known as the "abstract equality" operator.

This operator is infamous because many resources online, and JavaScript developers, in general, discourage its use because of its coercive behavior.Instead of blindly ignoring the double equality operator, we should try to understand how it behaves, and then we can decide for ourselves whether we want to not use it at all in our code or use it where it is safe to use.

Despite what you hear about his operator, it behaves according to some predefined algorithmic steps, and if we understand how it works, this operator won't scare us, and we might even prefer this operator in some cases over its cousin, the strict equality operator (===).

When two values are compared using the double equals operator, the steps taken by JavaScript to compare the two values are described by an abstract operation known as IsLooselyEqual.

Summary of abstract equality operator

The working of the double equals operator is roughly summarized in the steps below:

- If the values being compared are of the same type, then perform the strict equality comparison.

- If one value is `undefined` or `null` and the other value is also `undefined` or `null`, return true.
- If one or both values are objects, they are converted into primitive types, preferring the number type.
- If both values are primitives but are of different types, convert the types until they match, preferring the number type for coercing values.

Delegating to strict equality comparison

Think about the first point mentioned above. If the types of values being compared using this operator are the same, under the hood, the two values get compared using the triple equals or the strict equality operator. So, if we know that only the same type of values will get compared in some piece of code, it doesn't make any difference whether we use the double equals or the triple equals operator; in this case, we will always have the strict equality operator.

null vs undefined

The second point is also worth pondering over. Unlike the strict equality operator, the abstract or loose equality operator considers `null == undefined` comparison to be true.

```
1 console.log(null === null); // true
2 console.log(undefined === undefined); // true
3 console.log(null === undefined); // false
4
5 console.log(null == null); // true
6 console.log(undefined == undefined); // true
7 console.log(null == undefined); // true
```

You can see the code above in action below:

<ReplitEmbed src="https://replit.com/@newlineauthors/abstract-equality-operator-example1" />

The fact that `null` and `undefined` are equal to each other according to the abstract equality operator makes for an interesting use case of the abstract equality operator. When writing JavaScript code, it is common to check if a value is neither `null` nor `undefined`.

With the strict equality operator, we will have a check that looks something like the following:

```
if (someVal !== null && someVal !== undefined) {
  // code
}
```

Whereas with the abstract equality operator, we can shorten this check to just one condition:

```
if (someVal != null) {
  // code
}

// or

if (someVal != undefined) {
  // code
}
```

Considering how often we need to guard against `null` and `undefined` in our JavaScript code, I feel the abstract equality operator is ideal for this case. Having said that, you won't be wrong if you use the strict equality operator in this case.

"if" conditions

Although the coercive behavior of the abstract equality operator is predictable, as explained above, people often fall into a trap because of how they use this operator in the `if` statement conditions. Consider the following code example:

```
const someVal = {};

if (someVal == true) {
```

```
  console.log("if");
} else {
  console.log("else");
}
```

You can see the code above in action below:

<ReplitEmbed src="https://replit.com/@newlineauthors/abstract-equality-operator-example4" />

We know from an earlier lesson that objects are truthy values. So, in the above code example, it seems reasonable to assume that the `if` condition would be evaluated as `true`, leading to the execution of the `if` block. But, if you execute the above code, you might be surprised to know that instead of the `if` block, the `else` would execute because the `someVal == true` check would evaluate to `false`.

If either operand of the abstract equality operator is a boolean value, it is first converted to a number - `false` into `0` and `true` into `1`. So the `if` condition in the above code example would be evaluated as follows:

1. One operand is an object, and the other one is a boolean, so according to **step 10** of the isLooselyEqual abstract operation, if one operand is a boolean value, convert it into a number using the `ToNumber` abstract operation. So our condition would become:

   ```
   someVal == 1;
   ```

2. After coercing a boolean value into a number, we have a comparison between an object and a number. According to **step 12** of the `isLooselyEqual` abstract operation, the object `someVal` would be converted into a primitive value using the `ToPrimitive` abstract operation, passing "number" as the preferred type. The default primitive representation of object literals is `"[object Object]"`, so our condition after coercion would become: `js "[object Object]" == 1;`

3. Now, we have a comparison between a string and a number. According to **step 6** of the `isLooselyEqual` abstract operation, if one operand is a string and the other one is a number, convert a string into a number. Converting `"[object Object]"` into a number will give us NaN. So our condition would become:

```
1 NaN == 1;
```

4. After coercing three times, we have a comparison between a `NaN` value and a number. They are not equal to each other. *(`NaN` value is not equal to any other value, including itself.)*. So our condition fails to evaluate as `true`.

The purpose of the above discussion is to understand that checking if a value is `true` or `false` using the abstract equality operator doesn't always work as one might expect. It is easy to blame the abstract equality operator in such cases. Still, the fact is that those who write such code need help understanding or remembering how the abstract equality operator works.

In such cases where we want to check if a value is truthy or falsy, instead of using the abstract equality operator, it is enough to take advantage of the coercive behavior of the `if` statement. What I mean is that the `if` condition in the above code should be written as:

```
1 if (someVal) {
2   // code
3 }
```

With the above condition, we will get the expected result because `someVal` will be checked if it is a truthy value; if it is, the condition will evaluate to true, leading to the execution of the `if` block.

So, as a piece of advice, avoid checks such as `someVal == true`, where one operand is a boolean value. In such cases, take advantage of the implicit coercive nature of the `if` statement, which will check if the value is a valid value or not.

Further reading

- Boolean gotcha - You Don’t Know JS Yet
- Runtime semantics of “if” statement (ECMAScript specification)

Addition Operator

The addition operator can be used to perform the addition of two numbers, or it can be used to join two strings, also known as string concatenation.

The working of this operator is based on the ApplyStringOrNumericBinaryOperator. The way this works is that if any or both operands are non-primitive values, they are converted into primitive values using the `ToPrimitive` abstract operation, and no preferred type is specified. As a result, the “number” is the preferred type in this case because that is the default behavior of the `ToPrimitive` abstract operation.

After checking for non-primitive operands and coercing them, if any, into primitive values, the next step is to check if either or both operands are strings. If that’s the case, then the non-string operands, if any, are coerced into strings, and string concatenation is performed.

If neither operand is a string, then the addition is performed after coercing non-number operands into numbers.

Relational operators

The relational operators (<, >, <=, >=) are used to compare both strings and numbers. The abstract operation invoked in the case of relational operators is IsLessThan abstract operation. Despite its name, this operation handles “<=”, “>”, and “>=” comparisons as well.

If either operand is an object, it is converted into a primitive value with “number” as the preferred type. If both operands are strings, then they are compared using their Unicode code points. If not strings, then the operands are generally converted into numbers and then compared.

We can also compare date objects using relational operators. Recall that the `Date` objects are converted into strings when converted into primitive values using the `ToPrimitive` operation with no preferred type, but in the case of relational operators, `Date` objects, when converted into primitives, result in a number representation of the `Date` objects because in the case of relational operators, `ToPrimitive` abstract operation is passed "number" as the preferred type.

```
const d1 = new Date("2022-11-03");
const d2 = new Date("2023-05-10");

console.log(d1 < d2); // true
```

Here's a Replit of the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/relational-operators-example1" />

Now that we have had a deeper look at coercion and how some of the common abstract operations work to make coercion work, let us test what we learned in this module. This following exercise will allow us to consolidate our understanding of the topic of coercion.

Following are some expressions that involve coercion. Try to guess the output based on the knowledge you gained in this module. Don't worry if you don't understand all of them. Their explanation is also given below. You can obviously refer to the specification and the earlier lessons in this module to understand and guess the output of the expressions below.

```
0 == false

"" == false

0 == []

[123] == 123

[1] < [2]

[] == ![]
```

```
12
13 !!"true" == !!"false"
14
15 [1, 2, 3] + [4, 5, 6]
16
17 [undefined] == 0
18
19 [[]] == ''
20
21 [] + {}
```

Below is an explanation of the output of each of the above-mentioned expressions.

`0 == false`

Let's start with an easy one, and most people will probably get it right, even without reading this module. Now that we know how the abstract equality operator works let us understand the steps taken to evaluate this expression as `true`.

1. As the types are not equal and one of the operands is a boolean, the boolean operand is converted into a number using the ToNumber abstract operation. So, the first coercive step is to convert `false` into a number, i.e., `0`. The expression becomes:

   ```
   1 0 == 0
   ```

2. Now the types are equal, so the strict equality comparison is performed, i.e., `0 === 0`, giving us `true` as an output.

`"" == false`

1. As the types are not equal and one of the operands is a boolean, the boolean operand is converted into a number using the `ToNumber` abstract operation. So, the first coercive step is to convert `false` into a number, i.e., `0`. The expression becomes:

```
1 "" == 0
```

2. Now, we have a string and a number. Recall that the abstract equality operator prefers number comparison, so the string operand is converted into a number using the `ToNumber` abstract operation. An empty string, when converted into a number, outputs `0`. So the expression becomes:

```
1 0 == 0
```

3. The types are equal, so the strict equality comparison is performed, i.e., `0 === 0`, giving us `true` as an output.

`0 == []`

1. The array is converted into a primitive value using the `ToPrimitive` abstract operation. As the abstract equality operator prefers number comparison, the array is converted into a primitive value with a number as the preferred type. An empty array, when converted into a primitive value, outputs an empty string. So the expression becomes:

```
1 0 == ""
```

2. Next, the string will be converted into a number. An empty string converted into a number outputs `0`. So the expression becomes:

```
1 0 == 0
```

3. The types are equal, so the strict equality comparison is performed, i.e., `0 === 0`, giving us `true` as an output.

`[123] == 123`

1. We have a comparison between an array and a number. So, the array is converted into a primitive value using the `ToPrimitive` abstract operation, with number as the preferred type. The `valueOf` method will

be invoked first, as the preferred type is a number. But we know that the default implementation of the `valueOf` method simply returns the object on which it is called. So, the `toString` is invoked next. For Arrays, the `toString` method returns an empty string for empty arrays; for an array like `[1, 2, 3]`, it returns the contents of the array as a string, joined by commas, i.e., `"1,2,3"`. Each array element is coerced into a string and then joined by comma(s).

In our case, we have a single element in an array, i.e., `[123]`, so it will be coerced into `"123"`. So the expression becomes:

```
1 "123" == 123
```

2. Next, the string will be converted into a number. So the expression becomes:

```
1 123 == 123
```

3. The types are equal, so the strict equality comparison is performed, i.e., `123 === 123`, giving us `true` as an output.

:::info Weird fact about array conversion into a primitive value: an array containing `null` or `undefined` is coerced into an empty string, i.e., `[null]` —> `""` and `[undefined]` —> `""`. Similarly, an array containing both of them is coerced into a string containing a single comma, i.e., `[null, undefined]` —> `","`. Why don't we get `"null"`, `"undefined"`, and `"null,undefined"` for such arrays, respectively? This is just one of the corner cases of coercion. :::

`[1] < [2]`

1. This is a comparison between two objects. Both arrays are converted into primitive values using the `ToPrimitive` abstract operation, with number as the preferred type. As explained in the previous example, the `toString` will eventually be called to convert both arrays into

primitive values, giving us `"1"` and `"2"` as output, respectively. So the expression becomes:

```
"1" < "2"
```

2. Now, we have two strings. The types are equal, so the strict equality comparison is performed, i.e., `"1" < "2"`, giving us `true` as an output because the strings are compared using their Unicode code points.

`[] == ![]`

1. In this comparison, we have two operators: the abstract equality operator and the Not (!) operator. The Not operator has a higher precedence than the equality operator, so the sub-expression `![]` is evaluated first.

   The Not operator converts true into false, and vice versa. But here it is used with a non-boolean value. So what happens when JavaScript sees a value of one type in a context where it expects a value of a different type? Coercion! So `[]` will be coerced into a boolean value, as boolean is the expected type, using the `ToBoolean` abstract operation. As `[]` is a truthy value, it is coerced into `true` and then the Not operator negates it, converting `true` into `false`. So the expression becomes:

   ```
   [] == false
   ```

2. Next, the boolean operand, i.e., `false` is converted into a number, i.e., `0`. The expression is now:

   ```
   [] == 0
   ```

3. Now we have a comparison between an object and a number. Recalling how the abstract equality operator works, the object will be converted into a primitive value, preferring the number type. As mentioned in one of the earlier examples, an empty array is converted into an empty string, so the expression becomes:

```
1 "" == 0
```

4. Next, the empty string is converted into a number, i.e., `0`, using the `ToNumber` abstract operation.

   ```
   1 0 == 0
   ```

5. The types are equal, so the strict equality comparison is performed, i.e., `0 === 0`, giving us `true` as an output.

:::note If you are wondering how I know which operand, either left or right, is coerced first and what coercion is performed, I am simply referring to the steps mentioned in the ECMAScript specification. For example, for an expression involving a comparison using the abstract equality operator, I am referring to the steps of the IsLooselyEqual abstract operation.

This is what you should do as well. There is no need to memorize every step. Just understand the basics of how coercion is performed, which abstract operations are involved, and just refer to the specification. :::

`!!"true" == !!"false"`

1. Again, we have two operators in an expression. As mentioned before, the precedence of the logical Not operator is higher, so the sub-expressions `!!"true"` and `!!"false"` will be evaluated first.

   The string `"true"` in the expression `!!"true"` is a truthy value, so it will be coerced to the boolean value `true`. So the expression will become `!!true`. Next, we have two occurrences of the Not operator. Applying it twice to `true` will first convert it to `false` and then back to `true`.

   The second sub-expression `!!"false"` will also evaluate to `true` because the string `"false"` is a truthy value, so same as the first sub-expression, the expression will become `!!true` and then applying the

Not operator twice will give us `true`. So after the sub-expressions have been coerced and evaluated, our expression will become:

```
1 true == true
```

2. The types are equal, so the strict equality comparison is performed, i.e., `true === true`, giving us `true` as an output.

`[1, 2, 3] + [4, 5, 6]`

1. Recall how the addition operator works. The abstract operation involved here is ApplyStringOrNumericBinaryOperator.

   As both of the operands are objects, they are first converted into primitive values with no preferred type specified for the `ToPrimitive` abstract operation. So, by default, "number" is the preferred type. Arrays, when coerced into primitive values, are converted into primitive values using the `toString` method. For the array that we have in our expression, we will get `"1,2,3"` and `"4,5,6"` respectively. So the expression becomes:

   ```
   1 "1,2,3" + "4,5,6"
   ```

2. As both operands, after coercion, are strings, instead of addition, concatenation is performed, joining both strings, giving us `"1,2,34,5,6"` as output.

`[undefined] == 0`

1. As we have seen many times in this lesson when there is a comparison between an object and a number using the abstract equality operator, the object is first converted into a primitive value. Recall from a note earlier in this lesson that `[undefined]`, when converted into a primitive value, outputs an empty string. So the expression becomes:

   ```
   1 "" == 0
   ```

2. An empty string, when converted into a number, gives us `0`. So the expression becomes:

```
0 == 0
```

3. The types are equal, so the strict equality comparison is performed, i.e., `0 === 0`, giving us `true` as an output.

`[[]] == ''`

1. In this expression, we have an array containing an empty array and an empty string. The array operand is first converted into a primitive value. Recall how arrays are converted into primitive values. Apart from some corner cases mentioned earlier, arrays are converted into primitives by coercing their elements into strings and then joining them using commas. So, the nested empty array will be converted into a primitive value. What do we get when an empty array is converted into a primitive value? Yes, an empty string. So, the outer array is also converted into an empty string. The expression after coercion becomes:

```
"" == ""
```

2. An empty string is equal to an empty string, so the output is `true`.

`[] + {}`

1. As the operator is an addition operator and both operands are objects, they are both converted into primitives with no preferred type. So, by default, the preferred type is set to “number”.

   An empty array is converted into an empty string, and the default primitive representation of object literals is the string `"[object Object]"`. So the expression becomes:

```
"" + "[object Object]"
```

2. Both operands are now strings that are concatenated, giving us `“[object Object]” as an output.

References

Some of the expressions above were taken from the following github repo: wtfjs

Closures

Closure is one of the fundamental topics in JavaScript and can be tricky to understand for beginners. It is a powerful feature, and developing a good understanding of this topic is essential. In this module, we will take a deeper look at what closures are and how they work.

What is a closure?

The closure is a combination of the following two things:

- A Function
- A reference to the environment/scope in which that function is created

In other words, whenever we define a function in JavaScript, that function saves a reference to the environment in which it was created. This is what's referred to as a closure: a function along with a reference to the environment in which it is created.

Closures allow a nested function to access the declarations inside the containing function, even after the execution of the containing function has ended.

```
function outerFn() {
  const outerVar = 123;

  return function inner() {
    console.log(outerVar);
  };
}

const innerFn = outerFn();
innerFn();
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/Example1">

How does the `innerFn`, function returned by `outerFn`, have access to the `outerVar` variable declared in `outerFn` even after the “outerFn” execution is complete?

The above code works because JavaScript functions always create closures when they are created. In some programming languages, a function’s locally defined variables only exist for the duration of that function’s execution; when a function’s execution ends, variables defined in its local scope are destroyed. But that’s not the case in JavaScript, as is evident from the code example above. So, how do closures work?

How do closures work?

To understand how closures work, we need to understand how JavaScript resolves the scope of any identifier.

Although we had a detailed discussion on scope in an earlier module dedicated to the topic of scope, let us once again briefly go over how the lexical scope is resolved. Consider the following code example:

```
let isReading = true;

function learnJavaScript() {
  console.log(isReading);
}

learnJavaScript();
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/Example2”>

When the `learnJavaScript` function is invoked to log the value of the `isReading` variable, the JavaScript first needs to identify where it is defined. The first place where JavaScript will search is the local scope of the `learnJavaScript` function because this is where a reference to the `isReading` variable was found.

As the `isReading` variable is not defined in the local scope of the `learnJavaScript` function, JavaScript will search for this variable in the outer scope, which in this case is the global scope. JavaScript will find the declaration of the `isReading` variable in the global scope, so it will get its value and pass it to the `console.log` function so that it can be logged to the console.

Let us take a look at another code example that involves a nested function:

```
let isReading = true;

function learnJavaScript() {
  function stepsToLearnJavaScript() {
    console.log(isReading);
  }

  stepsToLearnJavaScript();
}

learnJavaScript();
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/Example3”>

In this code example, we have a nested function, `stepsToLearnJavaScript`, that logs the value of a variable defined in the global scope. If JavaScript hasn’t already determined the scope of the `isReading` variable, how will it find its declaration? Recall the scope chain!

JavaScript will need to traverse the scope chain to find the declaration of the `isReading` variable. As in the earlier example, the declaration is first searched in the current scope, which in this case is the local scope of the `stepsToLearnJavaScript` function. If the current scope doesn’t contain the declaration JavaScript is looking for, the search is sequentially expanded to the outer scopes. In our example, the `stepsToLearnJavaScript` function doesn’t contain the declaration of the `isReading` variable, so the search is moved to the outer scope, which in this case is the local scope of the `learnJavaScript` function. The declaration doesn’t exist in this scope

either; JavaScript traverses to the outer scope of the `learnJavaScript` function scope. The outer scope now is the global scope.

In the code example above, the variable declaration is found in the global scope, so the search for the variable declaration will be stopped when the search reaches the global scope. But what will happen if we remove the variable declaration from the above code example?

```
function learnJavaScript() {
  function stepsToLearnJavaScript() {
    console.log(isReading);
  }

  stepsToLearnJavaScript();
}

learnJavaScript();
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/Example4">

Now, the `stepsToLearnJavaScript` function is logging an undeclared variable. Now, what will happen when the search for the variable declaration reaches the global scope? At this point, javascript will do one of the following two things:

- throw an error if the code is in strict mode
- declare a global variable for you in non-strict mode (recall "implicit globals" in the scope module)

How are different scopes linked?

At this point, we know how the scope is resolved, and the different scopes are linked together, forming a chain that is known as the "scope chain." But have you wondered how different scopes are linked? It can't be magic; some mechanism must link different scopes together. How does JavaScript determine the outer scope of the current scope?

Let us revisit one of the code examples presented earlier in this lesson:

```
let isReading = true;

function learnJavaScript() {
  console.log(isReading);
}

learnJavaScript();
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/Example5">

We know that when the `learnJavaScript` function is invoked, the local scope of this function is linked to the global scope, but how? The answer is the hidden internal slot named [[Environment]]. This `[[Environment]]` internal slot exists on the functions, and it contains a reference to the outer scope/environment. In other words, this internal slot contains a reference to the scope on which the containing function has *closed over* or formed a "closure."

:::info

There are many hidden internal slots mentioned in the ECMAScript specification. The specification uses these hidden internal slots to define the required behavior. These hidden internal slots, just like abstract operations, may or may not be actual things that are implemented by the different JavaScript engines.

:::

In the code example above, when the `learnJavaScript` function is created, as it is created in the global scope, a reference to the global environment is saved in the internal `[[Environment]]` slot of the function object. Later, when the function is called, a new environment is created for the execution of the code inside the function. This environment (local scope of the function) is linked to the global environment by getting a value that is saved

in the `[[Environment]]` slot of the `learnJavaScript` function and saving it in an internal slot named [[OuterEnv]]. Each environment object has an internal slot that contains a reference to the outer environment.

The linkage of scopes in the above example can be conceptually visualized in the image below:

scope linkage

Let’s revisit one of the earlier examples from this lesson that involved a nested function:

```
let isReading = true;

function learnJavaScript() {
  function stepsToLearnJavaScript() {
    console.log(isReading);
  }

  stepsToLearnJavaScript();
}

learnJavaScript();
```

Here’s a Replit of the code above:

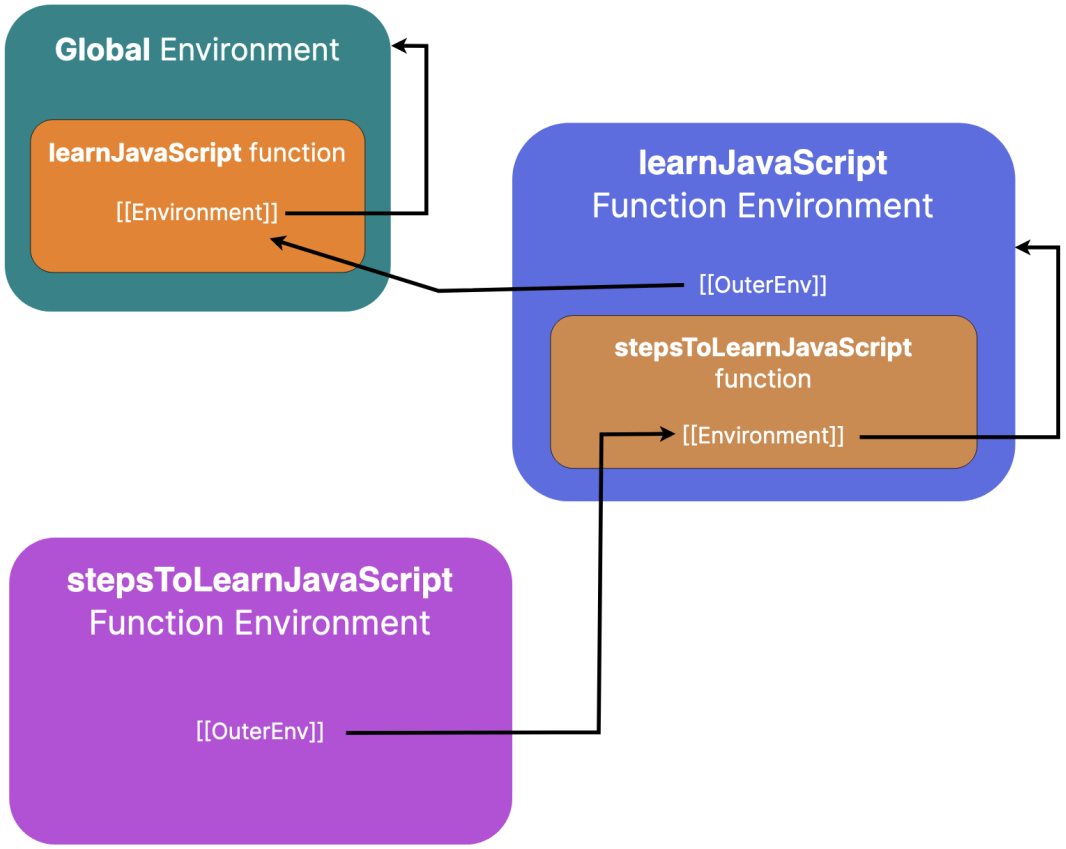
<ReplitEmbed src=”https://replit.com/@newlineauthors/Example6”>

In this code example, we have three different environments:

- The global environment
- The local environment of the `learnJavaScript` function (created when the function is invoked)

- The local environment of the `stepsToLearnJavaScript` function (created when the function is invoked)

The linkage between different scopes can be visualized in the image below:



scope linkage

Hopefully, the above two code examples and images have clarified how different scopes are linked together, forming a scope chain. This scope chain is traversed by the JavaScript engine, if needed, to resolve the scope of any identifier.

This linkage between different environments, i.e., the scope chain, is what makes closures possible in JavaScript. Due to this scope chain, a nested function can still access variables from the outer function even after the execution of the outer function has ended. This outer environment is kept in memory as long as the inner function has a reference to it.

Now, let us revisit the first code example in this lesson, which involves invoking a nested function from a different scope than the one it is defined in.

```
function outerFn() {
  const outerVar = 123;

  return function inner() {
    console.log(outerVar);
  };
}

const innerFn = outerFn();
innerFn();
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/Example7”>

Hopefully, you can now explain how the nested function has access to the `outerVar` variable, even after the completion of the `outerFn` execution.

The `inner` function has a reference to the local scope of the `outerFn` function, saved in its `[[Environment]]` internal slot. When the `inner` function is returned from the `outerFn` function, although the `outerFn` execution has ended, the `inner` function still has a reference to the local scope of the `outerFn`. When the `inner` function is invoked, the value of the `[[Environment]]` slot of the `inner` function is saved in the `[[OuterEnv]]` internal slot of the environment created for the execution of the `inner` function.

To summarize, every time a javascript function is created, a closure is formed, which allows that function to access the scope chain that was in effect when that function was defined. Each time a function is created, javascript saves the reference to the surrounding environment of the function in the internal `[[Environment]]` slot on the function object. When that function is called, a new environment is created for that function call,

and javascript saves the value of `[[Environment]]` slot on the function in the `[[OuterEnv]]` slot of the environment object.

It is a common misconception among beginners that closures are only formed when any function returns a nested function. But that is not the case.

Every time a function is created in JavaScript, it forms a closure over the environment in which that function was created. Forming a closure is a fancy way of saying that when a function is created, it saves a reference to the environment in which it was created.

What is the cause of this misconception? It is because of the following two reasons:

- Many online resources introduce the concept of closures with code examples containing a function that returns a nested function.
- Closures are only noticeable when a function is invoked from a different scope than the one it is defined in.

Most functions are usually invoked from the same scope in which they are defined. This makes the closures go unnoticed. It is only when a function is invoked from a different scope than the one it is defined in that closures become noticeable.

In the following code example, a function is defined and invoked from the global scope, so a closure formed by the function is unnoticeable.

```
let score = 150;

function logScore() {
  console.log(score);
}

logScore();
```

You can see the code above in this Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/closure-misconception-example1">

Unlike the code example above, in the following code example, closure is noticeable as the nested `greet` function has access to the `greetMsg` parameter of the containing `createGreeting` function even after the completion of the `createGreeting` function. The closure is noticeable because the `greet` function is invoked from a different scope than the one it is defined in. The `greet` function is defined in the local scope of the `createGreeting` function, but it is invoked from the global scope.

```
function createGreeting(greetMsg) {
  function greet(personName) {
    console.log(`${greetMsg} ${personName}!`);
  }

  return greet;
}

const sayHello = createGreeting("Hello");
sayHello("Mike");
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/closure-misconception-example2">

What output do you expect from the following code example?

```
for (var i = 1; i <= 3; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
}
```

You might expect the following output:

```
1;
2;
3;
```

Although the above output seems reasonable, it is not the output we get from the code example above. The actual output is as shown below:

```
4;
4;
4;
```

Here is the output of the code example above in a Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/closures-in-loops-example1">

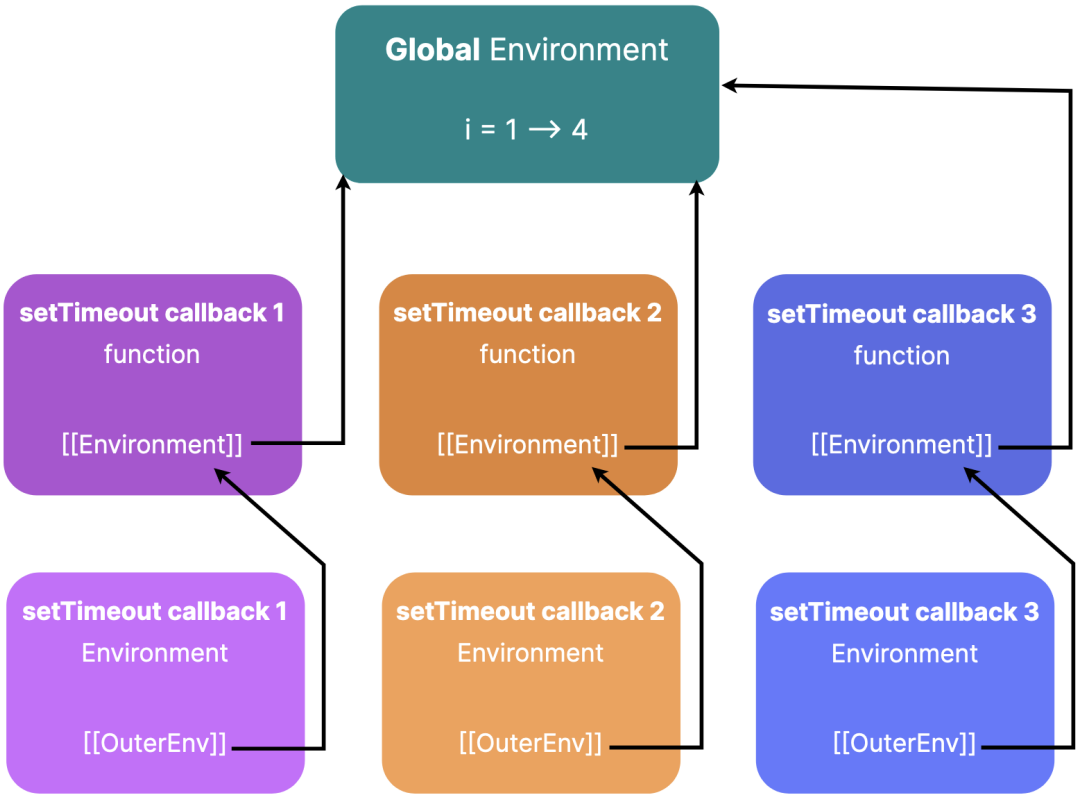
If this output surprised you, then this lesson is for you. Even if it didn't, do you understand the reasons behind this output? In this lesson, we will explore what causes this problem, also known as the "**closures in loop**" problem, and how we can fix it.

What causes this problem?

The code example above suffers from this infamous "closures in loop" problem because the callback function of each `setTimeout` forms a closure over the *same* variable `i`. As there are a total of three loop iterations in our example, `setTimeout` is called three times, so we have three callback functions, all having a closure over the *same* variable `i`.

The callback function of each `setTimeout` call is invoked *after* the loop execution has completed. The value of variable `i` after the last iteration of the loop is "4", and because each callback function of the "setTimeout" has a closure over the *same* variable `i`, all three of them see "4" as the value of the `i`. This is the reason they all log "4" on the console.

The scope linkage for the code example above can be visualized in the image below:



scope linkage

It is important to note that functions form closures over variables, not their values. Closure over variables implies that each function logs the *latest* value of the variable it has closed over; if functions formed closure over values rather than variables, they would log the snapshot of the value in effect when the closure was formed.

In our example, if the closure was over the values of `i` instead of the `i` variable, then each callback would have logged the value of `i` that was in effect in the iteration in which that callback was created. This means we would have gotten the expected output, i.e., `1 2 3` instead of `4 4 4`. But as is evident from the output of the code example, closures are formed over variables. As a result, each callback function of `setTimeout` has a closure over the variable `i`, and when each callback is executed, it sees the latest value of `i`.

How to resolve this problem?

Now that we understand what the “closures in loop” problem is and what causes it let us discuss how this problem was tackled before ES2015.

Pre-ES2015 solution

Before the introduction of ES2015, also known as ES6, one way to solve this problem was to use an IIFE (Immediately Invoked Function Expression). The following code example shows how using an IIFE resolves this problem.

```
for (var i = 1; i <= 3; i++) {
  ((counter) => {
    setTimeout(() => {
      console.log(counter);
    }, 1000);
  })(i);
}

/* output
---------
1
2
3
---------
*/
```

Here is the output of the code example above in a Replit:

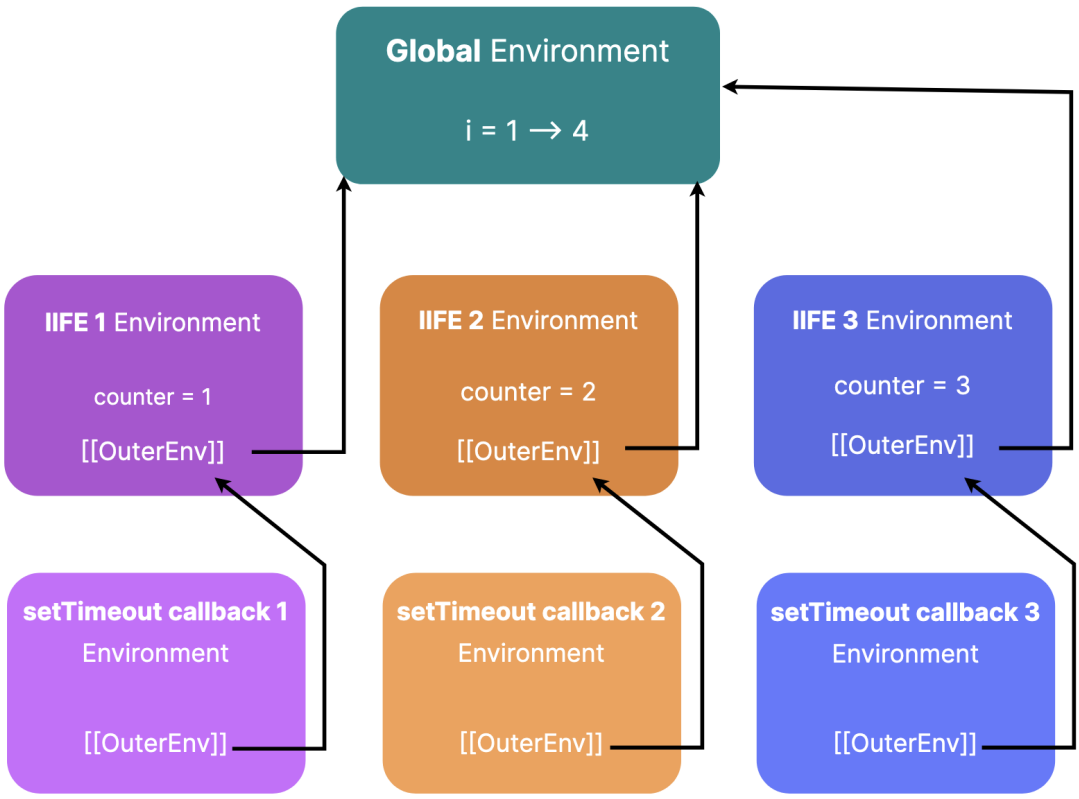
<ReplitEmbed src=”https://replit.com/@newlineauthors/closures-in-loops-example2”>

How does the use of an IIFE solve this problem?

Recall that the problem is caused by the closure of different callbacks over the *same* variable. But with the use of an IIFE, we can pass the value of `i` in each iteration to the IIFE as a parameter. This parameter (`counter`) is then used inside the callback function of the `setTimeout` function. This solves the problem because the `counter` parameter is closed over by each callback

function, and in each iteration, a new IIFE is created, along with the new callback function of `setTimeout`. Each new instance of the IIFE gets passed a new value of `i`, i.e., “1” in the first iteration, “2” in the second iteration, and so on. So now, with the use of an IIFE, each callback function has a closure over a different `counter` variable.

The image below helps visualize the scope chain in the above code example (function objects are not shown in the image below to keep it simple):



scope linkage

Although the callbacks of the `setTimeout` function log the `counter` variable, they still have access to the `i` variable as well. What if we log both `counter` and `i`? How will the output change?

```
for (var i = 1; i <= 3; i++) {
  ((counter) => {
    setTimeout(() => {
```

```
      console.log(counter, i);
    }, 1000);
  })(i);
}

/* output
---------
1 4
2 4
3 4
---------
*/
```

The output of the code example above can be seen in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/closures-in-loops-example3">

Each callback has a closure over a *different* instance of the `counter` variable, but they all still share the *same* variable `i` from the global scope. As a result, they log the latest value of `i`, i.e., "4".

ES2015 solution

ES2015 introduced block-scoped variables and constants with the help of `let` and `const` keywords, respectively. We can solve the "closures in loop" problem simply by replacing the `var` keyword with the `let` keyword in our original code example that had this problem.

```
for (let i = 1; i <= 3; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
}

/* output
---------
1
2
3
---------
*/
```

Here is the output of the code example above in a Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/closures-in-loops-example4">

Using the `let` keyword solves this problem because, unlike each callback function closing over the *same* variable `i`, the `let` being block-scoped causes each iteration of the loop to have a *different* copy of the variable `i`. This is the key idea that solves the problem of "closures in loop". Each iteration has its own *separate* copy of variable `i`, which means that the `setTimeout` callback created in each iteration closes over the copy of variable `i` that is limited to that particular iteration of the loop.

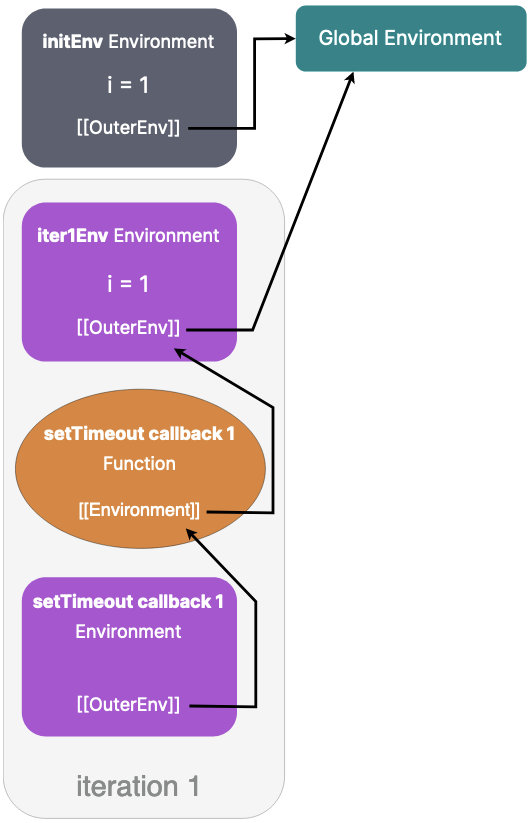
In our code example, we have three iterations of the loop and separate copies of variable `i`, each limited to a particular iteration of the loop. Although it seems that we have a single variable `i`, behind the scenes, each iteration gets its own copy of the variable `i`.

To understand how each iteration gets its own copy of variable `i`, the following steps explain how the above code is executed:

1. Before the execution of the `for` loop starts, an environment object (let's call it `initEnv`) is created, and it contains the variables declared in the initialization part of the `for` loop. In our case, we have only one variable, `i`, so the `initEnv` environment object contains a copy of variable `i`. This environment object is linked to the outer environment by saving a reference to the outer environment in the `[[OuterEnv]]` internal slot. The outer environment, in this case, is the global environment.
2. To start the first iteration of the `for` loop, a new environment object is created (let's call it `iter1Env`). This environment object also gets the global environment as its outer environment. The variable `i` is copied into this newly created `iter1Env` object by copying it from the `initEnv` object (created in step 1).

3. The loop condition `i <= 3` is `true`, so the body of the loop is executed as part of the first iteration of the loop. The callback function for the `setTimeout` is created, saving a reference to the environment object of the first loop iteration `iter1Env` (created in step 2) in the internal `[[Environment]]` slot of the callback function. After that, the callback function is passed to the `setTimeout` function as an argument to be executed later.

   The first iteration has been completed. The following image shows the linkage between different environments created so far:



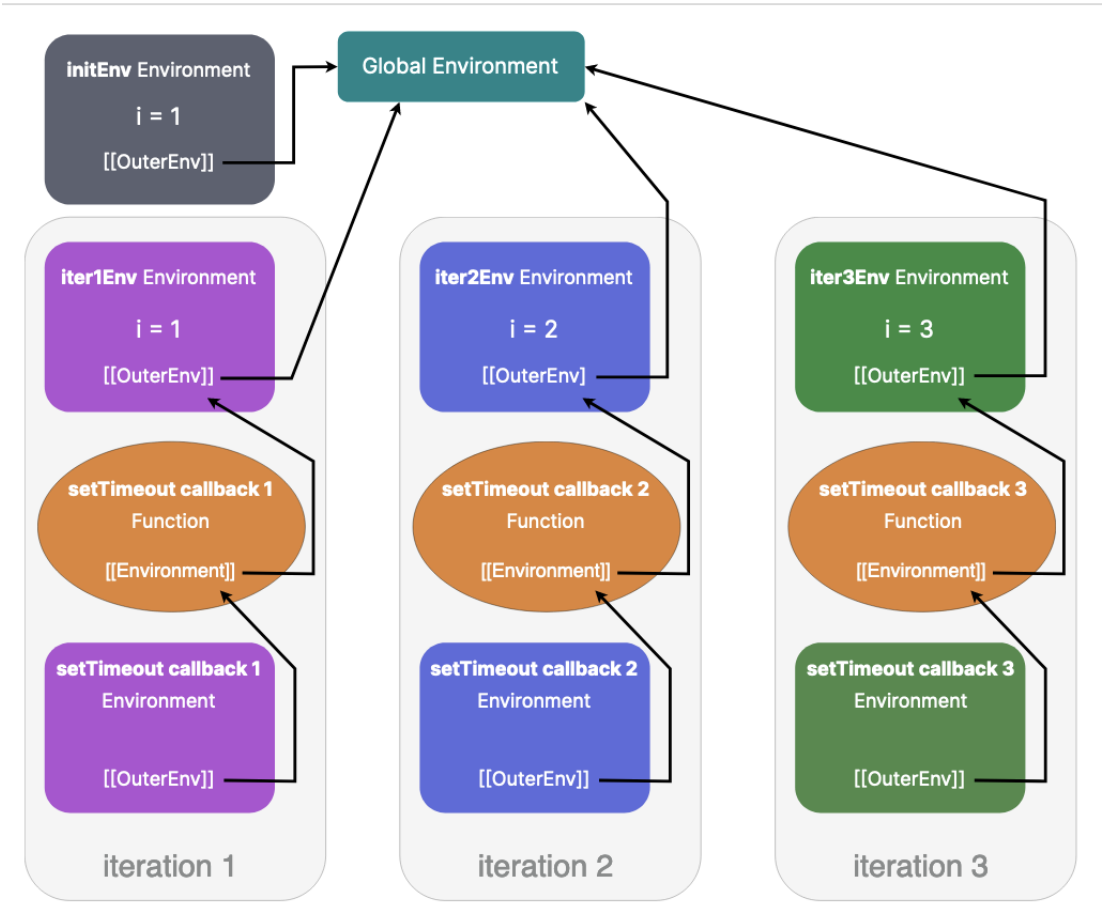
closue in loop - iteration 1

4. Now, to start the second iteration, a new environment object is created (let's call it `iter2Env`). The variable `i` is copied from the `initEnv` environment object (created in step 1), but the value of `i` in `iter2Env` is the value of `i` in `iter1Env` (created in step 2). The value of `i` in

`iter2Env` is then incremented due to the increment part (`i++`) of the `for` loop.

5. As in the previous iteration, a new callback function is created, and the reference to the `iter2Env` is saved in its `[[Environment]]` internal slot. The callback function is then passed to the `setTimeout` function as an argument.
6. This process is repeated for the third and final iteration of the loop.

The following diagram shows how different environments are linked together after three iterations of the loop:



closue in loop - all iterations

Hopefully, the final diagram clarifies why using the `let` keyword solves the "closures in loop" problem. As each environment object created for each

iteration of the loop has its *separate* copy of the variable `i`, the closure of each callback created in different iterations of the loop forms a closure over a separate copy of the variable `i`. As a result, they log the value they have closed over, giving us the expected output, i.e., `1 2 3`.

With the recent additions to the JavaScript language, it is now possible to have private fields and methods in JavaScript. However, before these changes were introduced in the language, closures were the go-to option for hiding data from public access. In object-oriented programming terminology, this is referred to as “data hiding” and “encapsulation”.

Following is an example of how we can have private variables and methods using closures:

```
const bank = (function () {
  // private data
  const accounts = [];

  // private function
  function getInternalBankLogs() {
    // ...
  }

  /***  public functions ***/

  function openAccount(data) {
    // some logic...
    // ...
    accounts.push(newAccount);
  }

  function deposit(accountNum, amount) {
    // ...
  }

  function withdraw(accoutNum, amount) {
    // ...
  }

  return {
    openAccount,
    deposit,
    withdraw
  };
```

```
})();

bank.openAccount({}); // ok
bank.accounts(); // error, not accessible
```

The code inside the IIFE is executed, and an object is returned from the IIFE that is assigned to the `bank` variable. The object returned only contains the data that needs to be accessible from outside the IIFE. The code that is meant to be private is not returned; as a result, that data remains private, limited to the boundary walls of the IIFE. However, thanks to closures, publicly exposed data that is returned from the IIFE can still access the private data inside the IIFE.

Further reading

- Emulating private methods with closures - (MDN)

Prototypes

Inheritance is a general object-oriented programming concept allowing objects to inherit other objects' methods and properties. This reduces code duplication and promotes code sharing between different objects.

Unlike traditional object-oriented programming languages like Java or C#, JavaScript has a different way of dealing with inheritance. Objects in JavaScript are *linked* to other objects, and this linkage allows an object to use the functionality of another object to which it is linked.

The linkage between objects in JavaScript forms a chain. This chain is known as the "**prototype chain**". Think of the scope chain, where each scope is linked to another scope until we reach the global scope. The prototype chain is similar: one object is linked to another object. This other object, in turn, is linked to another object, forming a chain between objects.

The prototype chain allows the sharing of properties between objects, and this is the idea of inheritance in JavaScript, known as the "**prototypal inheritance**". In prototypal inheritance, an object from which other objects inherit properties is known as the "**prototype**" of those objects.

When we create an object literal in JavaScript, it is, by default, linked to the built-in `Object.prototype` object.

```
1 const obj = {};
```

The `Object.prototype` object is the **prototype** of the `obj` object in the code example above.

How are objects linked?

Objects in JavaScript have a hidden internal slot named `[[Prototype]]`. When an object is created, it is linked to another object by saving a reference to the other object in the `[[Prototype]]` internal slot of the newly

created object. The other object whose reference is saved in the internal slot will serve as the “prototype” of the newly created object.

In the code example above, the `[[Prototype]]` slot of the `obj` object contains a reference to the `Object.prototype` object. So, `obj.[[Prototype]]` gives us the prototype of the `obj` object, the object from which `obj` is linked to and inherits the properties. But as `[[Prototype]]` is an internal slot not accessible by JavaScript, later in this lesson, we will see how we can access the prototype of any object.

The “prototype” property

In our discussion of prototypal inheritance so far, you might have noticed the term “prototype” used in two different contexts: one as a property, i.e., `Object.prototype` and the other as a general term used to describe an object that shares its properties with another object. This clash of names creates confusion among many when they first start learning about prototypal inheritance in JavaScript.

As functions are objects in JavaScript, they can have properties just like any other object. The property name “prototype” is one of the functions’ properties. The arrow functions do not have this property.

The prototype property of a function refers to an object that is used as the “prototype” of other objects when that function is invoked as a “constructor function” using the “new” keyword. Again, the term “prototype” has been used in two contexts here:

- The “prototype” property on functions
- An object is referred to as a “prototype” when it is linked and shares its properties with other objects.

The following code example shows that the property named “prototype” exists on functions:

```
1 function Car(name, model) {
2   this.name = name;
```

```
  this.model = model;
}

console.log(Object.getOwnPropertyNames(Car));

// [ "prototype", "length", "name" ]
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/prototypal-inheritance-example1" />

You can probably tell from the code above that the `Car` function is meant to be used as a constructor function. However, it is indeed just a normal function. The "prototype" property is only useful when we invoke a function as a constructor, i.e., with the `new` keyword.

Any properties added to the `Car.prototype` object will be shared by all the instances created from the `Car` constructor function. The `Car.prototype` function will serve as the "prototype" for all the instances of the `Car` constructor function.

Initially, the object pointed to by the prototype property on any function just contains a single property named "constructor". The value of this "constructor" property is a reference to the constructor function. In the case of `Car.prototype` object, `Car.prototype.constructor` refers to the `Car` constructor function.

```
// Car.prototype
{
  constructor: <Car function>
}
```

The following code example verifies that `Car.prototype.constructor` refers to the `Car` function:

```
function Car(name, model) {
  this.name = name;
  this.model = model;
}
```

```
5
6 console.log(Car.prototype.constructor === Car); // true
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/prototypal-inheritance-example2" />

:::note

The constructor property is rarely used, if at all, in the JavaScript code that we write.

:::

Let's add a property on the `Car.prototype` object:

```
1 Car.prototype.start = function () {
2   console.log("starting the engine of " + this.name);
3 };
4
5 const honda = new Car("honda", "1996");
6 const toyota = new Car("toyota", "2000");
7
8 honda.start(); // starting the engine of honda
9 toyota.start(); // starting the engine of toyota
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/prototypal-inheritance-example3" />

When a function is invoked using the `new` keyword, one of the steps during the creation of a new object is that the `[[Prototype]]` internal slot of the newly created object is pointed to the object referenced by the function's prototype property. As a result, the newly created object has access to the properties defined on the object referred to by the constructor function's prototype property.

Getting prototype of any object

The `Object` function has a static method named getPrototypeOf that can be used to get the prototype of any object. It returns the value of the internal `[[Prototype]]` property of the object.

For the `honda` object created in the previous code example, `Object.getPrototypeOf` function returns the `Car.prototype` object because the `Car.prototype` object is the prototype of all the instances of the `Car` constructor function.

```
function Car(name, model) {
  this.name = name;
  this.model = model;
}

Car.prototype.start = function () {
  console.log("starting the engine of " + this.name);
};

const honda = new Car("honda", "1996");

console.log(Object.getPrototypeOf(honda) === Car.prototype); // true
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/prototypal-inheritance-example4" />

Now that we know what prototypal inheritance is let us explore the prototype chain.

Object.prototype - parent of all objects

At the top of the prototypal inheritance hierarchy is the `Object.prototype` object. It is the root object or parent of all objects. When we create an object literal, its prototype is the `Object.prototype` object.

```
const obj = {};

console.log(Object.getPrototypeOf(obj) === Object.prototype);
// true
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/prototype-chain-example1" />

We didn't define any properties on the `obj` object. But we can still call some methods on it.

```
const obj = {};

console.log(obj.toString()); // [object Object]
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/prototype-chain-example2" />

We didn't define a method named `toString` on the `obj` object; how is it accessible in the code example above? You guessed it: it is defined on the `Object.prototype` object, and as `Object.prototype` is the prototype of `obj`, the properties defined on `Object.prototype` are inherited by `obj`, `toString` being one of them.

Objects created in this way are instances of the `Object` constructor function. We can also define `obj` as shown below:

```
const obj = new Object();
```

This has the same effect: it creates an empty object. As discussed in the previous lesson, functions have a prototype property that points to an object that serves as the "prototype" of all instances of that function when that function is invoked as a "constructor". So, the `Object.prototype` object

serves as the “prototype” of all objects created via `new Object()` or through object literal notation.

At this point, you might ask: isn’t `toString` callable on all objects? Yes, it is; some objects inherit it from the `Object.prototype` object, while other objects, such as arrays, inherit it from their prototype, i.e., the `Array.prototype` object, which overrides the `toString` implementation defined in `Object.prototype`.

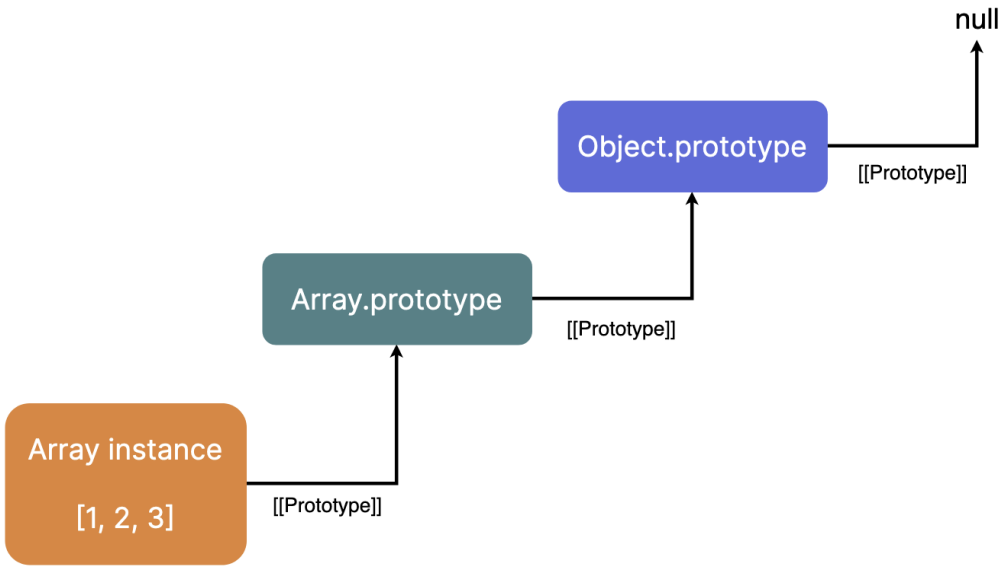
Some objects are directly linked to the `Object.prototype` object, while others are linked indirectly. Arrays, for example, are linked indirectly. Each array instance is directly linked to the `Array.prototype` object. The `Array.prototype` object is linked to the `Object.prototype` object. This forms a prototype chain that ends at the `Object.prototype` object.

```
1 const arrayPrototype = Object.getPrototypeOf([]);
2 const prototypeOfArrayPrototype =
Object.getPrototypeOf(arrayPrototype);
3
4 console.log(arrayPrototype === Array.prototype);
5 // true
6 console.log(prototypeOfArrayPrototype === Object.prototype);
7 // true
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/prototype-chain-example3” />

The prototype chain demonstrated in the code above can be visualized in the image below:



array prototype chain

The `Array.prototype` object contains the methods that are callable on every array, for example, `map`, `filter`, etc. The `Object.prototype` object contains the methods that are available to all objects, for example, the `toString` method.

Just as JavaScript traverses the scope chain to find the declaration of an identifier that can't be found in the current scope, JavaScript traverses the prototype chain to find the properties that cannot be found in the current object. The prototype chain has to end somewhere. Otherwise, JavaScript will keep traversing an endless chain; it ends at the `Object.prototype` object. Accessing the prototype of `Object.prototype` returns `null`.

```
1 console.log(Object.getPrototypeOf(Object.prototype));
2 // null
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/prototype-chain-example4" />

The prototype chain for strings is similar to that for arrays, except that instead of the `Array.prototype` object, there is the `String.prototype` object, which serves as the prototype of all string instances. This `String.prototype` object is, in turn, linked to the `Object.prototype` object, where the `Object.prototype` object serves as the prototype of the `String.prototype` object.

“Function” function

As confusing as it may sound, there’s a function named Function. Functions in JavaScript are objects and are instances of this “Function” constructor function. The `Function.prototype` object provides properties that are accessible by all functions; for example, methods like `bind`, `apply`, etc.

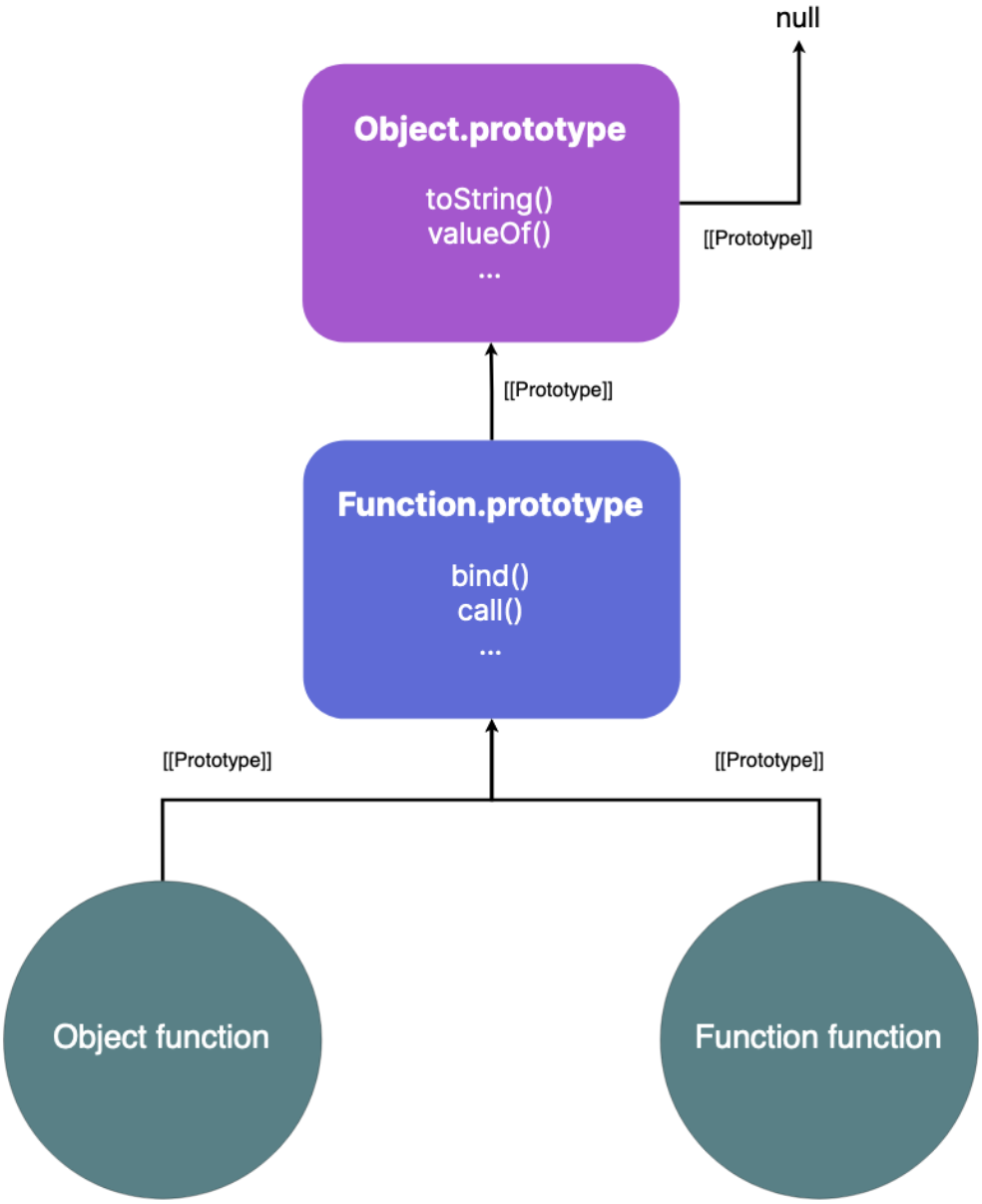
The `Function.prototype` object serves as the prototype for functions, including the `Object` function. Even the `Function` function, to which the `Function.prototype` object belongs, inherits properties from the `Function.prototype` object because `Function` is, after all, just a function itself. So it makes sense to make it inherit from the `Function.prototype` object, which contains the common properties for functions.

```
1 console.log(Object.getPrototypeOf(Object) == Function.prototype);
2 // true
3
4 console.log(Object.getPrototypeOf(Function) ==
Function.prototype);
5 // true
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/prototype-chain-example5” />

The prototype chain described above can be visualized in the following image:



array prototype chain

As the `Object.prototype` object is the root or parent object, it is part of the prototype chain, directly linked to the `Function.prototype` object.

The `__proto__` property is defined on the `Object.prototype` object. It is a getter and a setter that returns or sets the prototype of an object. In other

words, it returns or sets the value of the internal `[[Prototype]]` property of an object.

Although this property can be used to set and get the prototype of an object, its use is discouraged. This property has been deprecated, and better alternatives have been provided to get and set the prototype of an object.

```
const user = { name: "John Doe" };

console.log(user.__proto__);
// logs Object.prototype object
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/proto-property-example1" />

The above code example shows the use of the `__proto__` property as a *getter* to get the prototype of the `user` object. It can also be used as a *setter* to set the value of the `[[Prototype]]` internal property of an object. Again, its use is discouraged, and there are better alternatives. For setting the prototype, we can use the setPrototypeOf method.

The prototype of the `user` object in the code example above is the `Object.prototype` object. As a result, the `user` object has access to the `__proto__` property. As the `__proto__` property, when used as a getter, simply exposes the value of the internal `[[Prototype]]` property of an object, in the case of `user`, it returns the `Object.prototype` object.

Problems with __proto__

There are multiple reasons to avoid using `__proto__` property to get or set the prototype of an object.

As mentioned earlier, the `__proto__` property has been deprecated, and better alternatives exist to set and get the prototype of an object. This property wasn't standardized until 2015, so before that, it existed as a non-

standard feature of the JavaScript language. Even though the `__proto__` property is now part of the ECMAScript specification, this property wasn't standardized by the ECMAScript specification to promote or encourage its use; instead, it was standardized because it already existed in several JavaScript engines running in the browsers.

Another problem with the `__proto__` property is that it may not be available on all objects. You might ask how that is possible. Don't all objects directly or indirectly inherit from the `Object.prototype` object? The reason it may not be available is that we can create objects that do not inherit from any other object (we will discuss this in the next lesson).

So far, we have seen built-in objects, such as `Object.prototype`, automatically being set as prototypes of objects. How can we use our own objects as prototypes for other objects? For example, we have the following object, which we want to use as a prototype of some other object:

```
const propertyPrinter = {
  printOwnPropertyNames: function () {
    // "this" refers to the object on which
    // this function is called
    for (let prop of Object.getOwnPropertyNames(this)) {
      console.log(prop);
    }
  }
};
```

How can we use this object as a prototype of another object? We can use the `setPrototypeOf` method:

```
const propertyPrinter = {
  printOwnPropertyNames: function () {
    // "this" refers to the object on which
    // this function is called
    for (let prop of Object.getOwnPropertyNames(this)) {
      console.log(prop);
    }
  }
};

const user = {
  firstName: "John",
```

```
  lastName: "Doe",
  age: 25
};

// set the prototype of the "user" object
Object.setPrototypeOf(user, propertyPrinter);

// prototype methods are now accessible
user.printOwnPropertyNames();
// firstName
// lastName
// age
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/Custom-prototypes-example2" />

The deprecated `__proto__` property can also be used to achieve the same result, but as it is deprecated, we will instead discuss another option to set the prototype of an object explicitly.

Object.create method

The `Object.create` method is used to create a new object with another object, passed as the first argument, as the prototype of the newly created object. This method lets us explicitly set the prototype of an object. The code example above can be rewritten with `Object.create` as shown below:

```
// create a new object and set "propertyPrinter"
// object as its prototype
const user = Object.create(propertyPrinter);

user.firstName = "John";
user.lastName = "Doe";
user.age = 25;

// prototype methods are accessible
user.printOwnPropertyNames();
// firstName
// lastName
// age
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/Custom-prototypes-example3" />

Null prototype object

All objects ultimately inherit from the `Object.prototype` object because it sits at the top of the prototype chain and is the parent of all objects. However, we can create objects that do not inherit properties from any object. We just have to set `null` as the value of the internal `[[Prototype]]` property using the methods discussed above.

```
const obj = Object.create(null);

console.log(obj.toString());
// Error: toString not defined
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/Custom-prototypes-example4" />

The `obj` object in the above example doesn't have a prototype. If its prototype wasn't explicitly set to `null`, its prototype would have been the `Object.prototype` object, and it would have inherited the `toString` method, but as is evident from the code example above, `obj` doesn't have access to the `toString` method.

The `null` prototype objects may seem useless, but they are useful in some cases. For example, such objects are safe from attacks such as the prototype pollution attack, where a malicious code might add some properties to the prototype chain of an object that could change the normal flow of code execution.

Consider the following simplified example:

```
const user = {};

// malicious code adding "isAdmin"
// property in the prototype object
Object.prototype.isAdmin = true;

if (user.isAdmin) {
  console.log("grant access");
}
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/Custom-prototypes-example5" />

If the prototype of the `user` object was `null`, malicious code wouldn't have an effect on our code.

```
const user = Object.create(null);

// malicious code adding "isAdmin"
// property in the prototype object
Object.prototype.isAdmin = true;

if (user.isAdmin) {
  console.log("grant access");
} else {
  console.log("access denied");
}
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/Custom-prototypes-example6" />

In conventional object-oriented languages like Java or C#, we can extend or inherit a class to reuse its functionality. Extending a class creates a parent-child relationship where the child class extends the parent class. It promotes code reusability.

Until 2015, JavaScript didn't have classes. Constructor functions were used instead. To inherit from a constructor function, JavaScript developers

explicitly created a link between the prototype properties of two different constructor functions by using the `Object.create` method. The following code example shows how one constructor function could extend another constructor function to reuse some functionality:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.introduce = function () {
  console.log(`My name is ${this.name} and I am ${this.age} years old`);
};

function Student(name, age, id) {
  // delegate the responsibility of initializing
  // "name" and "age" properties to the Person
  // constructor
  Person.call(this, name, age);
  this.id = id;
}

// set "Person.prototype" object as the prototype
// of the "Student.prototype" object
Student.prototype = Object.create(Person.prototype);

// set the constructor property on the
// newly created Student.prototype object
Student.prototype.constructor = Student;

const mike = new Student("Mike", 20, 222);
mike.introduce();
```

Here's a Replit embed to run the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/ES6-classes-and-prototypes-example1" />

The following three points are worth noting in the code example above:

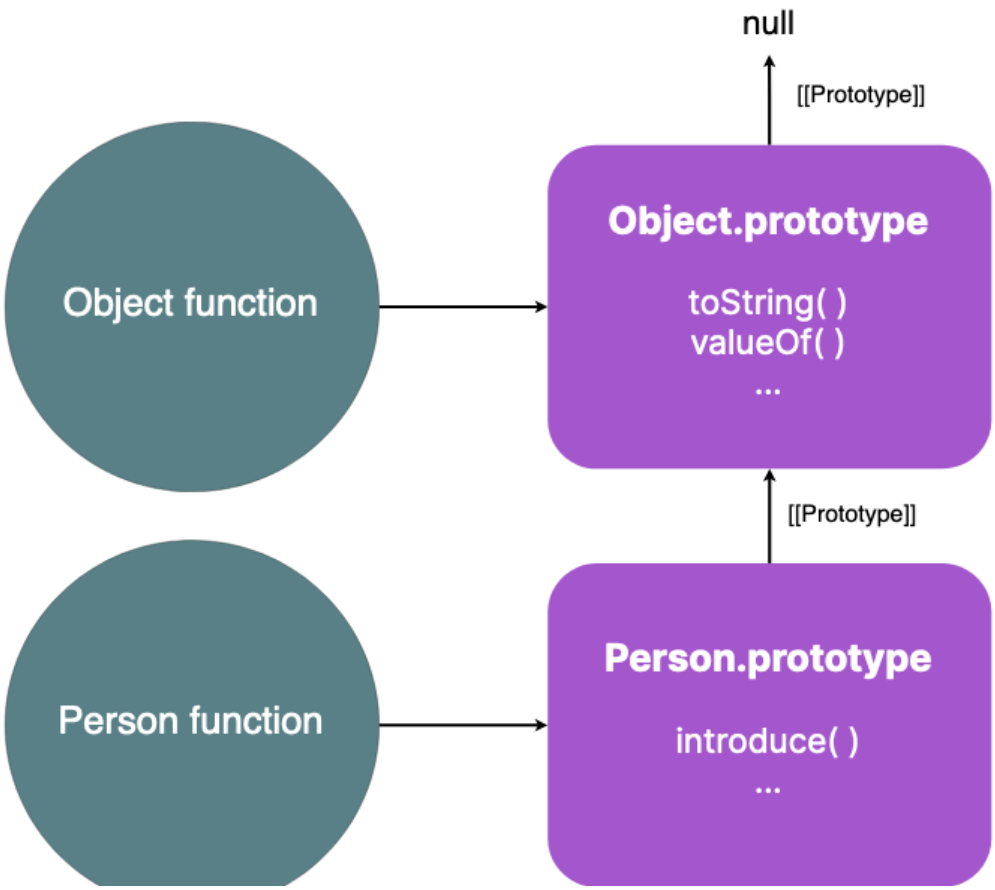
- `Person.call(...)` is invoked inside the `Student` constructor function to *delegate* the responsibility of adding and initializing the `name` and

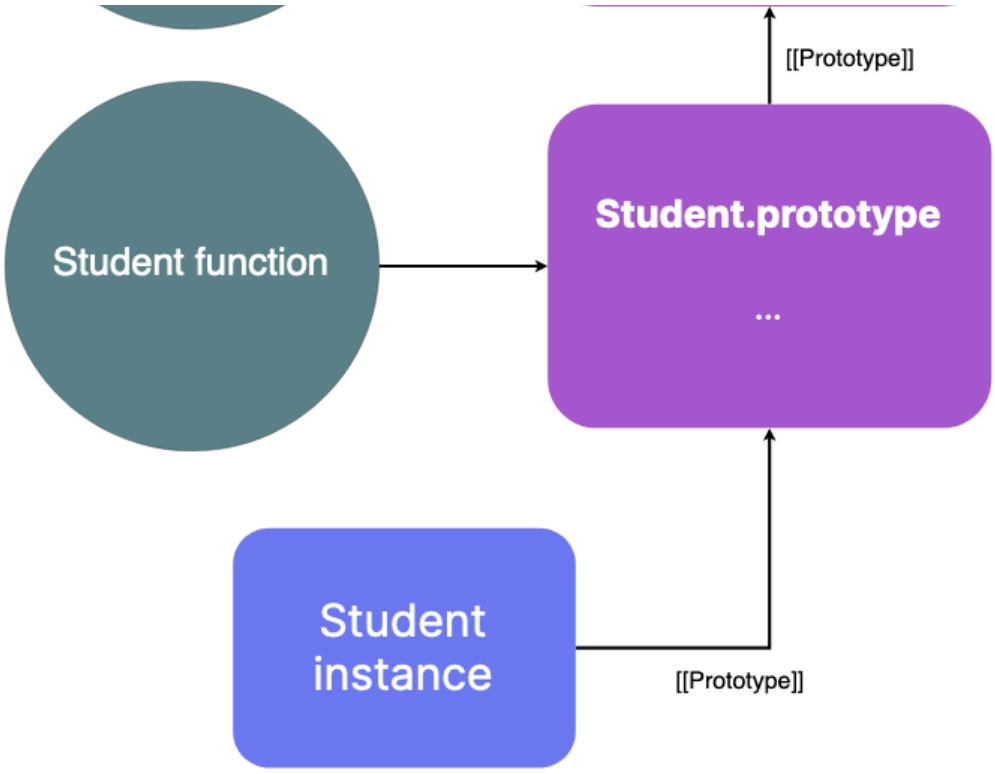
`age` properties on the newly created instance or object of `Student`.

- `Object.create` is used to create a `Student.prototype` object, and `Person.prototype` is set as the prototype of the newly created object.
- Normally, the object referred to by the "prototype" property of a function has a `constructor` property that points back to the function. As the object created by the `Object.create` method doesn't have the `constructor` property, we explicitly added the `constructor` property to the newly created `Student.prototype` object.

The linkage between the `Student.prototype` object and the `Person.prototype` object allows the instances of `Student` to use the properties defined on the `Person.prototype` object.

The image below helps visualize the prototype chain created as a result of our code example:





constructor function prototype chain

Although the code above works, it is error-prone because there are multiple steps to set a prototype link correctly between the two constructors. Imagine having more than two constructors that need to be linked like this. It is easy to forget any steps necessary to set up the prototype link correctly. Ideally, we want a more declarative way of achieving the same result. Ideally, we want a more declarative way of achieving the same result. A declarative solution will allow us to get the same result without having to explicitly create a link between the `Student.prototype` and `Person.prototype` objects.

ES2015 classes

As of 2015, JavaScript has classes. They provide a declarative way of writing code that is less error-prone. Classes come with the extends keyword that helps create a parent-child relationship between classes. The code example above can be rewritten using classes, as shown below:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  introduce() {
    console.log(`My name is ${this.name} and I am ${this.age}
years old`);
  }
}

class Student extends Person {
  constructor(name, age, id) {
    // delegate the responsibility of initializing
    // "name" and "age" properties to the parent class
    super(name, age);
    this.id = id;
  }
}
```

The code above gives us the same result as the one with the constructor functions. It also creates the same prototype linkages. We can verify this with the following comparisons:

```
console.log(Object.getPrototypeOf(mike) === Student.prototype);
// true
console.log(Object.getPrototypeOf(Student.prototype) ===
Person.prototype);
// true
console.log(Object.getPrototypeOf(Person.prototype) ===
Object.prototype);
// true
```

One important thing to mention here is that the classes are just syntactic sugar over the traditional constructor functions. Under the hood, we are still using the constructor functions, but classes allow us to write the code in a more declarative way.

One extra thing that the `extends` keyword does is that, apart from setting the linkage between `Student.prototype` and `Person.prototype` objects, it also links the constructor functions. It does this by setting the `Person` class as the prototype of the `Student` class. The following code verifies the

second prototype chain that the `extends` keyword sets up for us behind the scenes.

```
1 console.log(Object.getPrototypeOf(Student) === Person);
2 // true
```

The two prototype chains set up by the `extends` keyword serve two different purposes:

- `Student.prototype ---> Person.prototype` allows the inheritance of the instance properties.
- `Student ---> Person` allows the inheritance of the static properties.

‘this’ keyword

The `this` keyword is among the most confusing concepts in the JavaScript language. The cause of confusion is the different ways in which the value of the `this` keyword is set in different contexts.

In this module, we will try to demystify the `this` keyword. We will understand different ways in which the value of `this` is set. The `this` keyword can be used in different contexts: inside functions, global scope, inside modules, etc. We will explore different contexts and how `this` is set in those contexts.

Function context

The `this` keyword is mostly used inside functions to refer to the object using which the function was invoked. In other words, when a function is invoked as a “method” (invoked using an object), the `this` keyword becomes applicable for referencing the object used to invoke the function.

The `this` keyword is like an implicit parameter passed to a function. Just like explicit function parameters, the value of implicit parameter `this` is set when the function is *invoked*. This is an important point. The value of `this` inside a function depends on *how* that function is *called*.

Consider the following code example:

```
1 const student = {
2   id: 123,
3   name: "John Doe",
4   email: "john@email.com",
5   printInfo: function () {
6     console.log(`${this.id} - ${this.name} - ${this.email}`);
7   }
8 };
9
10 student.printInfo();
11 // 123 - John Doe - john@email.com
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/what-is-this-example1”>

The `printInfo` function in the code example above uses the `this` keyword, and looking at the code, we can tell that the `printInfo` function assumes that the value of `this` inside the `printInfo` function will be an object with three properties: `id`, `name`, and `email`. But as mentioned earlier, the value of `this` inside a function depends on *how* the function is called.

In the code example above, the `printInfo` function is invoked using the `student` object, and when a function is invoked using an object, the `this` inside that function refers to the object using which the function was invoked. So in our code example, `this` inside the `printInfo` refers to the `student` object. As the `student` object has the three properties that are accessed using the `this` keyword inside the `printInfo` function, their values are logged to the console.

What will `this` refer to if the function is not invoked as a “method”? Consider the following code example:

```
function orderFood() {
  console.log("Order confirmed against the name: " +
this.fullName);
}

orderFood();
// Order confirmed against the name: undefined
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/what-is-this-example2”>

What does `this` refer to inside the `orderFood` function?

The answer to the question above depends on whether our code is executed in strict mode. If non-strict mode, `this` inside a function, when not invoked as a method, refers to the global object, which in the case of browsers is the `window` object. However, the value of `this` inside a function is `undefined` in strict mode when not invoked as a method.

Can you guess in which mode the code was executed from the output of the code above? As `this.fullName` evaluated to `undefined`, the code was executed in non-strict mode.

If we execute the code above in strict mode:

```
"use strict";

function orderFood() {
  console.log("Order confirmed against the name: " +
this.fullName);
}

orderFood();
// Uncaught TypeError: this is undefined
```

Here's a Replit of the code above:

<ReplitEmbed src="https://replit.com/@newlineauthors/what-is-this-example3">

An error? Why?

Recall what the value of `this` is inside a "function" in strict mode. It is `undefined`. So `this.fullName` throws an error because we cannot access any properties on the `undefined` value.

Global context

In the global scope, the value of `this` depends on the environment in which our JavaScript code is executed.

JavaScript code can be executed in different environments, for example, browsers, NodeJS, etc. The value of `this` in global scope is different in different environments. In the case of browsers, the value of `this` in the global scope is the `window` object.

In NodeJS, the value of `this` depends on whether we are using the ECMAScript modules or the CommonJS modules. In ECMAScript modules, the value of `this` is `undefined` at the top level of a module. This is because the code in ECMAScript modules is executed in strict mode. In CommonJS modules, at the top level of a module, `this` refers to the `module.exports` object.

:::info

In Node.js, the JavaScript code is technically not executed in a global scope. Instead, it is executed in a module scope, where commonly used modules are CommonJS and ECMAScript modules.

:::

Inside web workers, the value of `this` at the top level refers to the global scope of the web worker, which is different from the global scope containing the window object in the browser. Code inside a web worker is executed in its own separate context with its own global scope.

Constructor function context

When a function is invoked as a constructor function using the `new` keyword, the `this` keyword inside the constructor function refers to the newly created object. The `new` keyword creates a new object and sets the newly created object as the value of `this`. As a result, we can use `this` inside a constructor function to add properties to the newly created object.

```
function Recipe(name, ingredients) {
  this.name = name;
  this.ingredients = ingredients;
}
```

The function above, when invoked as a constructor function, will add two properties: `name` and `ingredients` to the newly created object.

Class context

Code inside a class in JavaScript is executed in strict mode. As a result, the value of `this` inside methods is either `undefined` if not invoked on an object or the class instance itself, which is used to invoke the method.

```
class Shape {
  constructor(color) {
    this.color = color;
  }

  printColor() {
    console.log(this.color);
  }
}

const circle = new Shape("Red");
const printColorFn = circle.printColor;
printColorFn();
// Error: this is undefined
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/what-is-this-example5”>

The code example above throws an error because we have invoked the `printColor` method as a “function”. As mentioned earlier, code inside a class executes in strict mode, so, as within functions in strict mode, `this` inside methods is `undefined`.

DOM event handler context

We already know that `this` inside a function depends on how the function is called. But what about the callback functions that we do not call? I mean the callbacks like the DOM event handlers that we do not call but instead

are called for us by JavaScript whenever the click event is triggered. In such cases, what is the value of `this`?

The event listener callback is invoked with `this` set to the HTML element that triggered the event. Consider the following code example:

```
<button>Submit</button>

<script>
  const btn = document.querySelector("button");

  class FormHandler {
    constructor(submitBtn) {
      submitBtn.addEventListener("click", this.submitForm);
    }

    submitForm() {
      console.log("form submitted");
      console.log(this);
    }
  }

  new FormHandler(btn);
</script>
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/what-is-this-example6”>

Run the above code in a browser and check the console in the browser developer tools. Specifically, note the value of `this` logged by the `submitForm` method. The value of `this` inside this method, when it is invoked as an event listener callback, is the button element and not the instance of the class like we would normally expect.

This can cause a problem if we are not careful when using `this` inside an event listener callback function. Imagine a scenario where we had to call another method within the `FormHandler` class:

```
<button>Submit</button>

```

```
<script>
  const btn = document.querySelector("button");

  class FormHandler {
    constructor(submitBtn) {
      submitBtn.addEventListener("click", this.submitForm);
    }

    submitForm() {
      this.sendRequest();
      // ERROR: this.sendRequest is not a function
    }

    sendRequest() {
      console.log("sending request...");
    }
  }

  new FormHandler(btn);
</script>
```

Calling the `sendRequest` method from within the `submitForm` method throws an error because, as discussed, `this` inside an event handler function, `submitForm` in our case, is an HTML element that triggered the event. So, unlike what we expected, `this.sendRequest` throws an error. It would have worked if `this` inside the `submitForm` method was an instance of the `FormHandler` class. So, how can we call the `sendRequest` method from the `submitForm` method? There are multiple ways to achieve this, but we will discuss them in the later lessons in this module.

Before diving into how `this` works inside arrow functions, let us first explore the problem with using the `this` keyword inside regular functions. Consider the following code example:

```
function Counter(startingValue) {
  this.value = startingValue;
}

Counter.prototype.incrementFactory = function (incrementStep) {
  return function () {
    this.value += incrementStep;
    console.log(this.value);
  };
```

```
};

const counter = new Counter(0);
const increment5 = counter.incrementFactory(5);
increment5(); // NaN
increment5(); // NaN
increment5(); // NaN
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/arrow-functions-and-this-example1" />

Why did we get `NaN` as an output? The reason for unintended output is the incorrect value of `this` inside the function returned from the `incrementFactory` function.

Recall how the value of `this` gets set inside a function. It depends on *how* the function is called. In the code example above, how is `increment5` function called? Is it called as a "method" or as a standalone function? It is called a "function", so the value of `this` depends on whether our code is in strict mode or not. Assuming that our code is in non-strict mode, the value of `this` inside the `increment5` function is the global object, i.e., the `window` object in the case of browsers. So, `this.value` is actually `window.value`, and it is `undefined` because the `window` object, by default, doesn't have a `value` property. As a result, we get the `NaN` value when `undefined` is added to a number, i.e., the value of the `incrementStep` parameter.

How can we fix this problem? How can we ensure that the value of `this` inside the `increment5` function is what we want it to be? There are multiple ways to handle this problem. One way is to save the value of `this` inside the `incrementFactory` function before returning a function, and inside the returned function, use the variable containing the value of `this` instead of directly using `this`. The following code example shows this approach in action:

```
function Counter(startingValue) {
  this.value = startingValue;
```

```
}

Counter.prototype.incrementFactory = function (incrementStep) {
  const thisVal = this; // save `this` value
  return function () {
    // use `thisVar` variable instead of `this`
    thisVal.value += incrementStep;
    console.log(thisVal.value);
  };
};

const counter = new Counter(0);
const increment5 = counter.incrementFactory(5);
increment5(); // 5
increment5(); // 10
increment5(); // 15
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/arrow-functions-and-this-example2" />

The approach shown above was commonly used to fix similar problems where the value of `this` from the surrounding context was needed instead of the one from the current function where `this` was actually used. In the code example above, we needed the value of `this` from the surrounding context of the `incrementFactory` function and not the one inside the `increment5` function.

Arrow functions to the rescue

Another way to solve the problem shown above is to use an arrow function. Let's change the code example above to use an arrow function:

```
function Counter(startingValue) {
  this.value = startingValue;
}

Counter.prototype.incrementFactory = function (incrementStep) {
  // use an arrow function
  return () => {
    this.value += incrementStep;
```

```
    console.log(this.value);
  };
};

const counter = new Counter(0);
const increment5 = counter.incrementFactory(5);
increment5(); // 5
increment5(); // 10
increment5(); // 15
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/arrow-functions-and-this-example3" />

Using an arrow function solves the problem because, unlike regular functions, which get their own value of `this` when they are invoked, arrow functions don't get their own `this` value; instead, the value of `this` inside an arrow function is taken from the surrounding context.

The surrounding context is the environment in which the arrow function is defined. In our code example, the arrow function is created when the `incrementFactory` function is invoked using the `counter` object. So, `this` inside the `incrementFactory` function refers to the `counter` object, and this is the surrounding context of the arrow function returned from the `incrementFactory` function. As a result, the value of `this` inside the arrow function, when it is invoked, is also the `counter` object, and this is what we wanted `this` inside the `increment5` function to be to make our code example work.

Let us revisit the example we discussed in the previous lesson:

```
<button>Submit</button>

<script>
  const btn = document.querySelector("button");

  class FormHandler {
    constructor(submitBtn) {
      submitBtn.addEventListener("click", this.submitForm);
    }
```

```
10
11      submitForm() {
12        this.sendRequest();
13        // ERROR: this.sendRequest is not a function
14      }
15
16      sendRequest() {
17        console.log("sending request...");
18      }
19    }
20
21    new FormHandler(btn);
22  </script>
```

Clicking the `submit` button in the code example above throws an error because the value of `this` inside the `submitForm` method is the button element instead of the instance of the `FormHandler` class. As a result, the `this.sendRequest()` call throws an error because `this` needs to refer to an instance of the `FormHandler` class to allow us to call other methods in this class from within the `submitForm` method. So the problem is, how can we call the `sendRequest` method from the `submitForm` method? We said in the previous lesson that there is more than one way to solve this problem. One of them is to use an arrow function.

To fix the issue, inside the constructor of the `FormHandler` class, we can pass an arrow function instead of `this.submitForm` as a callback function to the click event listener. Inside the arrow function, we can invoke the `submitForm` method to handle the click event.

```
1 class FormHandler {
2   constructor(submitBtn) {
3     submitBtn.addEventListener("click", () => this.submitForm());
4   }
5
6   // methods...
7 }
```

Why did passing an arrow function as a callback fix the issue? We are still invoking the `submitForm` method inside the arrow function, so how is this different from directly passing `this.submitForm` as a callback function?

The reason an arrow function fixed the issue is that, as discussed earlier, arrow functions do not have their own value of `this`; they get it from the surrounding environment. The surrounding environment is the constructor in this case. What's the value of `this` inside the constructor? Its value is an instance of the `FormHandler` class when the constructor is invoked using the `new` keyword. So, instead of `this` referring to an HTML element inside the event handler callback function like it did in the previous example, the value of `this` inside the arrow function is the same as in the constructor, i.e., an instance of the `FormHandler` class. So when we invoke the `submitForm` method inside the arrow function, the value of `this` inside the `submitForm` method is also an instance of the `FormHandler` class. As a result, we can call any other method from inside the `submitForm` method.

In the older version of this code that throws an error, the event handler was `this.submitForm` method. It was not invoked by our code explicitly. Instead, it is invoked by JavaScript whenever the submit button is clicked. We know that the value of `this` inside functions depends on how a function is called. In this case, as we weren't invoking the function explicitly, we couldn't control the value of `this` inside the `submitForm` method. Using an arrow function allowed us to invoke the `submitForm` method explicitly and, consequently, allowed us to control the value of `this` inside it.

Arrow functions are really useful and a welcome addition to the JavaScript language. The problems they solve can also be solved in other ways, but other solutions are more verbose than arrow functions.

So far, we have discussed that the value of `this` depends either on the environment in which our JavaScript code is executed or, in the case of functions, on how a function is called. We have also discussed that arrow functions don't have their own value of `this`; instead, they get their value from the surrounding context.

All the ways we have seen so far for setting the value of `this` automatically set its value. Javascript also provides us with ways to explicitly set `this` to whatever value we want.

We can use any of the following three built-in methods to explicitly set the value of `this`:

- Function.prototype.call()
- Function.prototype.apply()
- Function.prototype.bind()

We won't go into details of how these methods work; you can learn how each of these methods works using the links given above. We will, however, see how explicitly setting `this` can be useful.

Let us revisit the code example from the previous lesson about arrow functions:

```
function Counter(startingValue) {
  this.value = startingValue;
}

Counter.prototype.incrementFactory = function (incrementStep) {
  return function () {
    this.value += incrementStep;
    console.log(this.value);
  };
};

const counter = new Counter(0);
const increment5 = counter.incrementFactory(5);
increment5(); // NaN
increment5(); // NaN
increment5(); // NaN
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/binding-this-example1" />

In the previous lesson, we saw how using an arrow function fixed the code example above. We could also fix this code by explicitly setting the value of `this` to the desired value, i.e., the `counter` object, which is used to invoke

the `incrementFactory` function. Instead of using an arrow function, we could use the `bind` method to set the value of `this`.

```
function Counter(startingValue) {
  this.value = startingValue;
}

Counter.prototype.incrementFactory = function (incrementStep) {
  const incrementFn = function () {
    this.value += incrementStep;
    console.log(this.value);
  };

  // return a function with `this` bound
  // to the object used to invoke the
  // `incrementFactory` method
  return incrementFn.bind(this);
};

const counter = new Counter(0);
const increment5 = counter.incrementFactory(5);
increment5(); // 5
increment5(); // 10
increment5(); // 15
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/binding-this-example2" />

Borrowing methods

Imagine having an object that contains methods that can be useful for other objects as well. How can we use those methods with other objects? One option is to duplicate the definition of methods for each object that needs them. But we don't want duplication. Is there a way to avoid duplication and reuse the existing methods?

```
const john = {
  name: "John",
  sayHello() {
    console.log("Hello, I am " + this.name);
  }
```

```
};

const sarah = {
  name: "Sarah"
};

// borrow method from john
const sayHello = john.sayHello;
sayHello.call(sarah);
// Hello, I am Sarah
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/binding-this-example3" />

We can explicitly set the value of `this` inside a function and use it with other objects using any of the three methods mentioned above. This allows us to avoid duplication and reuse code.

At this point, you might ask: *can we not create a constructor that allows us to create objects and add common methods in the constructor prototype property?* You are right. Creating a constructor is the right way to handle a situation where we want to create similar objects. However, explicitly setting the value of `this` allows us to reuse code between unrelated objects. It is a nice option to have and can be used where appropriate.

Chain constructor calls

Before classes were introduced in JavaScript, the traditional way of inheriting from another constructor function was to explicitly set the prototype chain and reuse the constructor that was being inherited from to add the common properties to the newly created object. The following code example shows how we can delegate the responsibility to an existing constructor to add some properties to the newly created object:

```
function Employee(name, age, id) {
  this.name = name;
  this.age = age;
  this.id = id;
```

```
}

function BankEmployee(name, age, id, bankName) {
  // delegate the responsibility of adding
  // "name", "age", and "id" properties to
  // the Person constructor
  Employee.call(this, name, age, id);
  this.bankName = bankName;
}
```

In the code example above, the `call` method has been used to call the `Employee` constructor, passing in the three properties that the `Employee` constructor can set on the newly created object. But how can we tell the `Employee` constructor to add the properties to the newly created `BankEmployee` object? This is where the first argument passed to the `call` method comes in. We have passed `this` as the first argument. Recall how the value of `this` is set inside a function: it depends on how the function is called. In this case, we expect the `BankEmployee` function to be invoked as a constructor function using the `new` keyword. As a result, `this` inside the `BankEmployee` function will be the newly created object. This newly created object is explicitly set as the value of `this` inside the `Employee` constructor.

In other words, the `Employee` constructor is invoked from inside of the `BankEmployee` constructor, with `this` explicitly set to the newly created `BankEmployee` object. As a result, properties added to `this` inside the `Employee` constructor will actually add the properties to the newly created `BankEmployee` object. This is how we can use the existing constructor function and reduce code duplication.

Revisit “this” problem

Let us revisit the example we discussed in the first lesson.

```
<button>Submit</button>

<script>
  const btn = document.querySelector("button");

  class FormHandler {
```

```
    constructor(submitBtn) {
      submitBtn.addEventListener("click", this.submitForm);
    }

    submitForm() {
      this.sendRequest();
      // ERROR: this.sendRequest is not a function
    }

    sendRequest() {
      console.log("sending request...");
    }
  }

  new FormHandler(btn);
</script>
```

The problem we are trying to fix is that we want to call other methods of the same class from inside the event handler callback, but trying to do so throws an error because `this` inside an event handler is the HTML element that triggered the DOM event. In the previous lesson, we saw how an arrow function could solve this problem. There's another way to fix the issue, and that's to explicitly set the value of `this` inside the `submitForm` method.

```
class FormHandler {
  constructor(submitBtn) {
    submitBtn.addEventListener("click",
this.submitForm.bind(this));
  }

  // methods...
}
```

Explicitly setting the value of `this` inside the `submitForm` method using the `bind` method fixes the problem because it overrides the default value of `this` inside an event callback function. We have explicitly set it to the value of `this` inside the `FormHandler` class constructor, i.e., an instance of the `FormHandler` class.

The difference between passing `this.submitForm` vs. passing `this.submitForm.bind(this)` to the `addEventListener` method is that with `this.submitForm`, the value of `this` inside the `submitForm` method

depends on *how* the `submitForm` method is invoked. In this case, we know that the value of `this` inside it will be the html button element, which is not what we want. On the other hand, passing `this.submitForm.bind(this)` solves the problem because, unlike before, the value of `this` inside the `submitForm` method is explicitly bound to be an instance of the `FormHandler` class.

`globalThis` is a globally available property in JavaScript that allows us to access the global object regardless of which environment our JavaScript code is executed in. It provides us with a standard way to access the global object across different JavaScript environments.

As we know, JavaScript can be executed in different environments, for example, the browser, the Node.js runtime, etc. Each of these environments has a different global object available to the JavaScript code executing in the environment. Before `globalThis`, there was no standard way to access the global object in a cross-environment way. This made our code less portable.

The `globalThis` property made it possible to access the global object without worrying about the environment. If we know that our code can be executed in different environments, using `globalThis` to access the global object is the way to go in modern JavaScript code.

Imagine a scenario where you want to check if the global object contains a certain property regardless of whether your code is executed in a browser or a NodeJS environment. Without a standard way to access the global object, you might write the following code:

```
1 if (typeof window !== "undefined" && window.secretProperty) {
2   // execute code for browser
3 } else if (typeof global !== "undefined" && global.secretProperty)
{
4   // execute code for nodejs
5 }
```

With `globalThis`, we can simplify the above code as shown below:

```
if (globalThis.secretProperty) {
  // execute code
}
```

Although this property is well supported in modern browsers, older versions of browsers don't support it. So, browser support might be taken into consideration when using this property.

"this" vs globalThis

The `globalThis` should not be confused with the `this` keyword. We have discussed in this module how the value of `this` can vary depending on different execution contexts, but `globalThis` is just a standard way to access the global object in different JavaScript environments. Its value only varies depending on the environment in which our code is executed. It isn't affected by how the function is called or whether our code is in strict mode or not.

In ECMAScript modules, code is executed in strict mode. As a result, the value of `this` in module scope is `undefined`, but the value of `globalThis` is the global object of the execution environment; in NodeJS, it is the `global` object, and in the browsers, it is the `window` object.

This module was all about the `this` keyword; we discussed different ways its value is set in different contexts; we also discussed a problem that can arise due to an unexpected value of `this` and explored different options we have to fix such problems.

As `this` can have different values in different contexts, let us summarize what value `this` has in different contexts:

- In the case of an arrow function, the value of `this` is taken from the surrounding context.
- In the case of a regular function, the value of `this` depends on *how* a function is called and whether the code is executed in strict mode or not.

    - If a function is invoked as a constructor using the `new` keyword, the value of `this` is the newly created object.
    - If the value of `this` is explicitly set using `bind`, `call`, or `apply` functions, then the value of `this` is whatever value is passed as the first argument to these functions.
    - If a function is invoked as a "method", the value of `this` is the object used to call the method.
    - If the function is invoked without any object, i.e., as a "function", the value of `this` is the global object in non-strict mode and `undefined` in strict mode.
- In DOM event handler callbacks, the value of `this` is the HTML element that triggered the event.
- In the global scope in browsers, `this` refers to the global `window` object.
- In NodeJS, code at the top level is executed in a module scope. In ECMAScript modules, `this` is `undefined` at the top level of a module because the code in the ECMAScript module is implicitly executed in strict mode. In CommonJS modules, `this` refers to the `module.exports` object at the top level of a module.

Hopefully, this module gave you a better understanding of `this` keyword and helped you understand how its value is set in different contexts. This module also aims to help you understand the kinds of problems an unexpected value of `this` can cause in our code and what different options are available to us to fix such problems.

Symbol

`symbol` is a primitive value that can be created using a function named `Symbol`. What makes this primitive value interesting is that it is guaranteed to be unique. This guarantee of being unique is the selling point of symbols.

With the introduction of symbols in ES2015, two things changed in JavaScript:

- A new type of value was introduced in the language.
- Only strings were allowed to be added to objects as property keys. Now, symbols can be keys as well.

Before we discuss how to use symbols in our code, let us first understand the motivation behind adding symbols to the JavaScript language.

Symbols were originally meant to be used as a mechanism to add private properties to objects and were supposed to be called "**private name objects**". But later, their name was changed to symbols, and they were made a primitive value.

It turned out that each symbol being a unique value is pretty useful because it allows the JavaScript language to be extended and remain backwards compatible. Symbols allow JavaScript to add new properties to objects that cannot conflict with the existing properties on objects that others might have used in their code.

One of the main goals of TC39 is to keep JavaScript backwards compatible. With this goal in mind, any new feature added to the language must not break existing code. Symbols help keep the promise of backward compatibility.

Some features in the JavaScript language require looking up a property on an object. What property keys could have been chosen for such features? Choosing any string property name wasn't possible because someone might

have used that property in their code, and using that property for a new feature might have broken their code.

For example, for converting an object into a primitive value, a special property named Symbol.toPrimitive is looked up on the object by the type conversion algorithm. If such a property exists and its value is a function, its return value is used as the primitive representation of the object. Otherwise, the default mechanism of calling the `toString` and `valueOf` methods in different order is used, as explained in an earlier module on coercion.

Think about how such a feature could have been added to the language with a string property. What name could possibly have been chosen that was guaranteed to not break existing code?

This is where symbols shine. Using symbols as properties enables such features by adding unique properties to objects that cannot possibly break existing code because:

- symbols didn't exist before ES2015, and
- each symbol is a unique value

Symbol values can be created using the `Symbol` function. It's important to note that the `Symbol` function must be invoked without the `new` keyword. Attempting to use the `new` keyword to invoke the `Symbol` function will result in an error. This is because it prevents the creation of an object wrapper around the symbol. Every call to the `Symbol` function must create a new unique symbol value.

```
1 const sym = Symbol();
```

Once a symbol has been created, it can be added to an existing object as a property using the square bracket notation:

```
1 const sym = Symbol();
2
3 const obj = {};
4 obj[sym] = "hello";
```

```
5
6 console.log(obj[sym]); // hello
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/creating-symbols-example2" />

Alternatively, the computed property notation can also be used to add a symbol as a property to the object:

```
1 const sym = Symbol();
2
3 const obj = {
4   [sym]: "hello"
5 };
6
7 console.log(obj[sym]); // hello
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/creating-symbols-example3" />

Symbols and privacy

Regular `string` properties on objects can be accessed in a variety of ways. For instance, consider the following code example:

```
1 const name = "name";
2
3 const person = {
4   [name]: "John Doe"
5 };
6
7 console.log(person.name); // John Doe
8 console.log(person["name"]); // John Doe
9 console.log(person[name]); // John Doe
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/creating-symbols-example4” />

In this example, a property key “name” was added to the `person` object using the computed property name. This property can be accessed in multiple ways, as demonstrated in the code example above. But what happens if the value of the `name` variable is changed from a `string` to a symbol?

```
const name = Symbol();

const person = {
  [name]: "John Doe"
};

console.log(person.name); // undefined
console.log(person["name"]); // undefined
console.log(person[name]); // John Doe
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/creating-symbols-example5” />

It will be noticed that `person.name` and `person["name"]` output `undefined`, but the last `console.log` statement logs the name as expected. So, what’s the difference between the first two statements and the last one? The first two statements log `undefined` because symbol properties cannot be accessed until access to the original symbol is obtained. Only the last `console.log` statement in the code example above used the original symbol, which is why it logged the name instead of `undefined`.

So, if the original symbol is not accessible, does that make the symbol property a private property? What happens if the properties of an object are iterated over? Let’s find out if the symbol properties of an object can be obtained.

```
const name = Symbol();

```

```
const person = {
  [name]: "John Doe",
  age: 20
};

// only sees the "age" property
for (const prop in person) {
  console.log(prop);
}

console.log(Object.keys(person)); // ["age"]

console.log(Object.getOwnPropertyNames(person)); // ["age"]
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/creating-symbols-example6" />

Looking at the output of the above code example, it might be assumed that symbol properties are indeed private. However, this assumption would be incorrect. Symbol properties are not private, as the next code example demonstrates.

```
const name = Symbol();

const person = {
  [name]: "John Doe",
  age: 20
};

console.log(Object.getOwnPropertyDescriptors(person));

console.log(Object.getOwnPropertySymbols(person));
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/creating-symbols-example7" />

The above code example shows that symbol properties can be discovered using methods like `Object.getOwnPropertyDescriptors` or

`Object.getOwnPropertySymbols`. While symbol properties may be a bit more inconvenient to access compared to `string` properties, they are not private. JavaScript has true private properties, and symbols are not intended to be used as private properties.

Adding a description to symbols

When creating symbols, there is the option to provide a description for each symbol. This description can be useful for debugging purposes. The following code example creates a symbol with a description:

```
const propSymbol = Symbol("property symbol");
```

The description is passed as an argument to the `Symbol` function. The provided description can then be accessed using the property named `description` on symbols.

```
const propSymbol = Symbol("property symbol");

console.log(propSymbol.description);
// property symbol
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/creating-symbols-example9" />

The `description` property can only be used to get the property of a symbol; it cannot be used to set the description. The following code example shows that assigning a value to the `description` property of a symbol doesn't change its value. The `description` property is actually a getter, and a setter isn't defined for this property. As a result, the description can only be obtained but not set using this property.

```
const propSymbol = Symbol("property symbol");

console.log(propSymbol.description);
// property symbol

```

```
propSymbol.description = "123";

console.log(propSymbol.description);
// property symbol
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/creating-symbols-example10" />

Each call to the `Symbol` function creates a new unique symbol. However, JavaScript also allows us to create symbols that can be shared across files or realms. This is where the concept of a **global symbol** repository comes in.

We can use the `Symbol.for` method to create a global symbol in the global symbol repository.

```
const globalSymbolKey = "my-global-Symbol";
const mySymbol = Symbol.for(globalSymbolKey);

console.log(mySymbol === Symbol.for(globalSymbolKey));
// true
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/global-symbols-example" />

We pass a key to `Symbol.for` method, and using this key, we can retrieve the symbol associated with it in the global symbol repository. If the key exists in the repository, the `Symbol.for` method returns the symbol associated with it; otherwise, it creates a new symbol in the repository and associates it with the given key.

If we have a global symbol, we can get the key it is associated with using the `Symbol.keyFor` method.

```
const globalSymbolKey = "my-global-Symbol";
const mySymbol = Symbol.for(globalSymbolKey);

```

```
4 console.log(Symbol.keyFor(mySymbol));
5 // my-global-Symbol
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/global-symbols-example2" />

The JavaScript language uses several built-in symbols to make different features work, for example, the `Symbol.toPrimitive` described in an earlier lesson in this module. Such symbols are referred to as "well-known symbols" by the ECMAScript specification.

While the complete list of well-known symbols in the JavaScript language can be found in the ECMAScript specification, this lesson describes some of the well-known symbols.

Symbol.toPrimitive

As explained in an earlier lesson in this module, `Symbol.toPrimitive` represents a symbol property that is used by the object to primitive conversion process in JavaScript. Its value is a function that is passed a *hint* or the *preferred type* of the primitive value to represent the object being converted into a primitive value. The return value is used as the primitive value of the object.

The following code example hooks into the object to the primitive conversion process of the `movie` object and returns a different representation of the object based on the value of the `hint` argument.

```
1 const movie = {
2   name: "Jurassic Park",
3   releaseDate: "09,June,1993",
4
5   [Symbol.toPrimitive](hint) {
6     if (hint === "number") {
7       return new Date(this.releaseDate).getTime();
8     } else {
9       return this.name;
```

```
    }
  }
};

console.log(Number(movie));
console.log(String(movie));
```

You can see the output of the above code example in this Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/well-known-symbols-example1" />

Symbol.toStringTag

The default implementation of the `Object.prototype.toString` method isn't very useful for user-defined objects.

```
console.log({}.toString()); // [object Object]
```

For some built-in objects, the default implementation of the `toString` method is also not useful. As a result, many objects override the default `toString` implementation.

```
const arr = [1, 2, 3];
// overridden implementation
console.log(arr.toString()); // "1,2,3"

// default implementation from Object.prototype
console.log(Object.prototype.toString.call(arr)); // [object Array]
```

You can see the output of the above code example in this Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/well-known-symbols-example3" />

Notice the "Array" part in the output of the default `toString` implementation. It is known as the tag. For some built-in objects, the tag is the type of the value, for example, "Array" in the case of arrays and "String" in the case of strings.

In the case of user-defined objects, the default `toString` output, as shown above, is `[object Object]` where the tag is “Object” - not very useful.

The well-known symbol `Symbol.toStringTag` allows us to change the value of the tag.

```
const task = {
  title: "exercise",
  isComplete: false,
  [Symbol.toStringTag]: "Task"
};

console.log(task.toString()); // [object Task]
```

You can see the output of the above code example in this Replit:

<ReplitEmbed src=”https://replit.com/@newlineauthors/well-known-symbols-example4” />

Symbol.isConcatSpreadable

The `[Symbole.isConcatSpreadable]` property is looked up by the `concat` method of arrays to determine if the elements of the array or array-like object passed to the `concat` method should be *spread* or flattened.

```
const arr = [1, 2, 3];
console.log([].concat(arr));
// [1, 2, 3]

arr[Symbol.isConcatSpreadable] = false;
console.log([].concat(arr));
// [[1, 2, 3]]
```

You can see the output of the above code example in this Replit:

<ReplitEmbed src=”https://replit.com/@newlineauthors/well-known-symbols-example5” />

As the output of the above code example shows, the default behavior for arrays is to spread their elements. This default behavior can be overridden

by setting `Symbol.isConcatSpreadable` to `false`.

For array-like objects, the default behavior is to not spread their properties. This can be overridden by setting `Symbol.isConcatSpreadable` to `true`.

```
const obj = {
  0: 123,
  1: 456,
  length: 2,
  [Symbol.isConcatSpreadable]: true
};

console.log([].concat(obj));
// [123, 456]
```

You can see the output of the above code example in this Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/well-known-symbols-example6" />

There are other well-known symbols that allow us to hook into different built-in operations in JavaScript. The complete list can be seen in the ECMAScript specification or MDN - Symbol.

Asynchronous JavaScript

In this module, we will cover asynchronous programming in JavaScript. We will learn what asynchronous programming means and how it was traditionally done in JavaScript. We will also discuss the problems with the traditional way of handling asynchronous code in JavaScript and how promises, introduced in ES2015, changed the way we handle asynchronous code in JavaScript. We will discuss promises in detail and also learn about the `async-await` syntax that simplifies using promises.

What does asynchronous mean?

In the context of programming, “asynchronous” means that the program starts a potentially long-running task and is free to do other tasks while the long-running task is executed in the background. The key point to understand here is that the program doesn’t have to wait for the long-running task to be completed; it is free to do other tasks. Once the task is complete, the program is notified and presented with the result of the task.

Asynchronous programming is needed because of the problems with traditional synchronous programming. In synchronous programs, each instruction is executed one after the other in a sequential manner. Instructions are executed in the order they appear in the program. As a result, synchronous programs are easier to reason about, but they also present problems that are solved by asynchronous programming.

The problem with synchronous programs is that a potentially long-running task will block the program’s execution until its completion. This presents problems like poor program performance, poor user experience, and inefficient resource utilization.

Although asynchronous programs solve the problems presented by synchronous programs, asynchronous programs come with their own set of challenges, like error handling, managing the shared state and resources, managing the coordination between different parts of the program, etc.

Asynchronous JavaScript

Before we dive into how asynchronous code can be written in JavaScript and how it is handled behind the scenes, let's take a step back and see the problem we face in JavaScript if we execute some long-running code, like a loop in the following example:

```
function block() {
  const start = new Date();

  while (new Date() - start < 3000) {
    // simulate long running operation
    // that takes approximately 3 seconds
  }
}

console.log("Before long running operation");
// gets logged immediately

block();

console.log("After long running operation");
// gets logged after approximately 3 seconds
```

You can see the code above in this Replit:

<ReplitEmbed src="https://replit.com/@newlineauthors/overview-example1">

JavaScript is a single-threaded language, which has its pros and cons. JavaScript developers generally don't have to worry about problems that come with multi-threaded programs, like race conditions and deadlocks. However, the limitation of the single thread is evident in the code example above.

The code example above is designed to simulate some long-running code that takes approximately 3 seconds to complete. During these 3 seconds, the main thread on which our JavaScript code is executed is blocked; nothing else executes during those 3 seconds. If this JavaScript code is attached to

an HTML file and executed in a browser, the UI will freeze until the loop ends.

To see the frozen UI in action, try adding the above Javascript code to a file named `index.js` and attaching it to an HTML file containing the following HTML:

```
<h1>hello world</h1>
<button onclick="alert('hello')">Click</button>
```

You can see the code above in this Replit:

<ReplitEmbed src=”https://replit.com/@newlineauthors/overview-example2”>

On the initial page load, you will notice that the button is not clickable for a few seconds. The UI will stay frozen until the JavaScript code has been executed, specifically the long-running loop.

This presents a horrible user experience in web applications. The single thread puts a hard limit on what we can do with JavaScript on the main thread.

Nowadays, JavaScript engines are good enough to execute the code at a very good speed; engines are designed to heavily optimize the JavaScript code to execute it as efficiently as possible. Still, one needs to be mindful that the main thread should not be blocked by any code that could take long enough to make the delay noticeable.

:::info

JavaScript also allows us to execute some code in another thread, independent of the main thread, using the web workers.

:::

Next, let’s discuss the traditional way of writing asynchronous code in JavaScript using callbacks. We will also discuss the problems with using

callbacks.

Using callback functions to handle asynchronous code has been the traditional way of writing asynchronous code in JavaScript. A callback function is a function that is passed to another function as an argument and is intended to be invoked after some asynchronous operation. The function that receives the callback function as an argument typically invokes the callback function with the result of the asynchronous operation or the error if the asynchronous operation fails.

Operations like HTTP requests are asynchronous, but they aren't handled by JavaScript. The code we write initiates the asynchronous operation; the actual asynchronous operation is handled by the browser in the case of client-side JavaScript, background threads, or the operating system itself in the case of the NodeJS runtime.

In simple words, asynchronous operations take place in the background (outside of JavaScript land), and in the meantime, other things can execute on the main thread (in JavaScript land).

When the asynchronous operation is completed, our JavaScript code is notified, leading to the execution of the callback function that we provided at the time of initiating the asynchronous operation.

This is how JavaScript gets around its limitation of a single thread. The asynchronous operations are actually handled by the runtime (browser, NodeJS, etc.). In the meantime, JavaScript can do other things.

The following code example shows the use of a callback function by sending an HTTP request to a fake REST API:

```
function fetchUser(url) {
  const xhr = new XMLHttpRequest();

  xhr.addEventListener("load", function () {
    // check if the operation is complete
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        // Request succeeded
```

```
        const data = JSON.parse(xhr.responseText);
        console.log(data);
      } else {
        // Request failed
        const error = new Error("Failed to fetch todo");
        console.log(error);
      }
    }
  });

  xhr.open("GET", url);
  xhr.send();
}

fetchUser("https://jsonplaceholder.typicode.com/todos/1");
```

You can run the code above in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/callbacks-example1" />

A callback function passed to the `addEventListener` method handles the result of the HTTP request.

Different DOM events, for example, the click event, are also handled asynchronously using the callback functions. A callback is registered as an event handler and is invoked later whenever the event is triggered.

```
const submitBtn = document.getElementById("submit");

submitBtn.addEventListener("click", function (event) {
  // code to handle the click event
});
```

Similarly, different timer functions like `setTimeout` are also provided with a callback function that is intended to be invoked after the specified amount of time has elapsed.

```
setTimeout(function () {
  console.log("logged after 2 seconds");
}, 2000);
```

You can run the code above in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/callbacks-example3" />

There is a common misconception that the callback provided to `setTimeout` is invoked exactly after the specified amount of time has passed. The time we specify when invoking `setTimeout` is the *minimum* amount of time after which the provided callback will be invoked.

In the code example provided above, we have specified 2 seconds (2000 milliseconds) as the time after which the callback should be invoked. But the callback will not be invoked exactly after 2 seconds. Imagine a scenario where we also have a long-running loop that takes approximately 4 seconds to run. As we discussed in the previous lesson, JavaScript is single-threaded, so a long-running loop will block the main thread, which means that the callback function provided to `setTimeout` cannot be executed until the loop ends. The following code example demonstrates this scenario:

```
setTimeout(function () {
  console.log("logged after approximately 4 seconds instead of 2");
}, 2000);

const start = new Date();

// takes approximately 4 seconds to execute
while (new Date() - start < 4000) {}
```

You can run the code above in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/callbacks-example4" />

The above code example shows that the `setTimeout` callback or the callback provided to the `setInterval` function may not be invoked after the specified time has passed; some other code can block them from being invoked after the specified time.

JavaScript is single-threaded; only one thing can execute at a time on the main thread. Currently, executing code cannot be interrupted to execute some other code, like the callback function of `setTimeout` in the code example above.

The callback functions are at the heart of asynchronous code in JavaScript. Using callbacks works, but they also come with problems.

Problems with callbacks

Using the traditional way of using callbacks to handle asynchronous code presents multiple problems that make it hard to use callbacks effectively.

Callback hell

Imagine a scenario where multiple asynchronous operations need to start sequentially because each operation depends on the result of the previous operation. To handle such a scenario, we have to nest callbacks, which, depending on the number of asynchronous operations, can lead to code that is hard to read and maintain. The following code example shows this in action:

```
asyncOperation1((result1) => {
  asyncOperation2(result1, (result2) => {
    asyncOperation3(result2, (result3) => {
      asyncOperation4(result3, (result4) => {
        // ...more nested callbacks and operations
      });
    });
  });
});
```

This is what’s referred to as the **Callback Hell** or the **Pyramid of Doom** because the nesting at each level creates a structure that looks like a pyramid.

Looking at the code example above, it is not hard to imagine that it will get harder to read as more operations are added to the sequence of

asynchronous operations. Not only is it hard to read, but it is also hard to maintain and refactor. Note that the code example above does not include any error handling; add that to the code above, and you will have the following:

```
asyncOperation1((error1, result1) => {
  if (error1) {
    // handle error
  } else {
    asyncOperation2(result1, (error2, result2) => {
      if (error2) {
        // handle error
      } else {
        asyncOperation3(result2, (error3, result3) => {
          if (error3) {
            // handle error
          } else {
            asyncOperation4(result3, (error4, result4) => {
              if (error4) {
                // handle error
              } else {
                // ...more nested asynchronous operations and
callbacks
              }
            });
          }
        });
      }
    });
  }
});
```

The code above fits its name: “Callback Hell”. No one wants to deal with such a code. It is hard to reason about. We will see in later lessons in this module how the same code can be rewritten using promises and async-await syntax to make it more readable and maintainable.

Error handling

Writing error handling code using callbacks, as shown above, is not a pleasant experience. As shown in the code above, we need to handle errors in each callback, which can lead to duplication of error handling logic.

There is no central place where we can catch and handle errors for all the asynchronous operations.

In this lesson, we discussed how callbacks are used to write asynchronous code in JavaScript. Though there are better alternatives like promises and async-await that solve the problems with callbacks discussed above, callbacks are still commonly used. Although promises solve the problems with callbacks, they still use callbacks, but in a more manageable way that helps us avoid the callback hell.

As we know already, the JavaScript language is single-threaded. Long-running code on the main thread can block the thread; in the case of browsers, blocking the main thread means that browsers cannot respond to user interactions and cannot render changes on the UI. This is why the screen freezes when some long-running code blocks the main thread. In the case of NodeJs, in the context of application servers, blocking the main thread means that the server cannot handle the incoming HTTP requests until the main thread is unblocked.

To get around the limitation of a single thread where JavaScript code executes, as discussed in the previous lesson, any asynchronous operation is handled in the background, and in the meantime, the main thread can do other things.

Asynchronous operations like HTTP requests are handled by the browser in the background, and when the HTTP request is completed, our JavaScript code is executed using the callback we provided at the time of starting the HTTP request. The same is true for other asynchronous operations, like file handling in NodeJS. Every asynchronous operation in NodeJs is either handled by the internal thread pool of NodeJS or the operating system itself.

If we consider the main thread as the “JavaScript world”, then the asynchronous operations actually happen outside the JavaScript world. Once the operation is completed, to get back into the JavaScript world, callbacks are used, which are invoked to execute the JavaScript code in response to the successful completion or failure of the operation.

So, in short, the code we write that executes on the main thread only initiates the asynchronous operation. Instead of waiting for the operation to complete, the main thread is free to do other things. The asynchronous operation is handled in the background by the environment in which our JavaScript code is executed. It can be a browser or a runtime like NodeJS. But how does the execution get back from the background to the main thread where our code is executed? This is where the **event loop** comes into the picture.

What is an event loop?

The event loop in JavaScript is one of those concepts that is explained abstractly to make it easier for others to understand. This lesson will aim to create a solid understanding of the event loop.

The event loop helps to execute asynchronous operations in a non-blocking manner. When an asynchronous operation is completed in the background to execute the JavaScript callback that we provided at the time of initiating the asynchronous operation, it needs to be pushed onto the call stack.

The call stack is a stack data structure used to keep track of the currently executing code. Every function call is added to the stack as a stack frame. The frame is popped off the stack when the function execution ends.

Let us understand the event loop using the following code example:

```
setTimeout(() => {
  console.log("hello world");
}, 2000);

console.log("after setTimeout");

// output:
// ------
// after setTimeout
// hello world
```

You can run the code above in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/event-loop-example1" />

The code above logs "after setTimeout" before "hello world" which is inside the callback of `setTimeout`. The following steps explain how the code above executes:

1. To execute the code, a task is created and pushed onto the call stack. This is what's commonly referred to as the "global execution context".
2. Once the code execution starts, the first thing to do is to invoke the `setTimeout` function, passing in a callback that is to be invoked after approximately 2 seconds. Calling `setTimeout` starts a timer in the background that will expire after 2 seconds in our code example. In the meantime, the main thread continues executing the code instead of waiting for the timer to expire. This is why "after setTimeout" is logged before "hello world".
3. Next, the `console.log` is executed, logging "after setTimeout" on the console.
4. At this point, the synchronous execution of our code has ended. As a result, the task created (step 1) to execute the code is popped off the call stack. Now, JavaScript is ready to execute any scheduled callbacks. This point is important: *no asynchronous callback can be invoked until the synchronous execution of the code has ended*. Remember, only one thing executes at a time on the main thread, and the currently executing code cannot be interrupted.
5. After the synchronous execution ends, let us assume that by this time the timer has expired (in reality, our code execution will end long before 2 seconds). As soon as the timer expires, a task is enqueued in a **task queue** to execute the callback of `setTimeout`. The task queue is where different tasks are queued until they can be pushed onto the call stack and executed.
6. The **event loop** is the entity that processes the tasks in the task queue and pushes each of them to the call stack to execute them. Tasks are processed in the order they are enqueued in the task queue. In our case, there is only one task in the task queue. This task is pushed onto the call stack, but the event loop only pushes the tasks onto the call stack if

the call stack is empty. In our case, the call stack is empty, so the callback of `setTimeout` can be executed. As a result, “hello world” is logged on the console.

The role of the event loop, as described in the above steps, is to process the tasks in the task queue if the call stack is empty and there are one or more tasks in the task queue waiting to be executed. So, the event loop is an entity that allows asynchronous code to be executed in JavaScript in a non-blocking manner. The event loop can be thought of as a loop that continuously checks if there are any tasks waiting to be executed.

The event loop is what connects the two worlds: the “JavaScript world”, where our code executes, and the “background world”, where the asynchronous operations are actually executed.

The above steps can be visualized in the following image:

function parameter scope

Take your time to understand exactly what happens behind the scenes. The timer is intentionally shown to take longer than 2 seconds to make the visualization easier to understand. Understanding the steps above before

seeing the image will make it easy to understand how our code example executes.

Any user interaction like the click event requires scheduling a task; the same is true for executing the callbacks of timing functions like `setTimeout`. Tasks are queued in the task queue until the event loop processes them. The task queue is also referred to as the **event queue** or the **callback queue**.

The event loop processes a single task during its single turn, commonly referred to as the “event loop tick” or just “tick”. The next task is processed during the next turn or tick of the event loop. The browser may choose to render UI updates between tasks.

The event loop can have multiple sources of tasks, and the browser decides which source to process tasks from during each tick of the event loop. Another queue is known as the microtask queue, which we will discuss later in this module. The event loop also processes microtasks, but there is a difference in how the event loop processes tasks and microtasks. The difference will be clear when we discuss the microtask queue.

In this lesson, we discussed what an event loop is and how tasks are processed: a single task per tick of the event loop.

Tool to visualize the event loop

The following is a great tool to visualize the workings of an event loop using our code:

- loupe

Further reading

The following resources are recommended to further our understanding of the event loop:

- The event loop (MDN)
- What the heck is the event loop anyway? | Philip Roberts (youtube video)
- Jake Archibald on the web browser event loop, setTimeout, micro tasks, requestAnimationFrame (youtube video)

Promises introduced in ES2015 have transformed the way we handle asynchronous code in JavaScript. Promises are meant to address the problems we discussed with callbacks.

A promise represents an object that acts as a *placeholder* for a value that is typically produced as a result of an asynchronous operation. In other words, a promise object represents the successful completion or failure of an asynchronous operation. There is a common misconception among beginners that promises to make our code asynchronous; they do not. Think of promises as a notification mechanism that notifies us about the success or failure of some operation that is *already* asynchronous. Promises wrap asynchronous operations and allow us to execute code when an asynchronous operation is successfully completed or when it fails. That's all a promise does. Nothing more, nothing less. It is only meant to observe the asynchronous operation and notify us when that operation is completed.

Before we learn how we can create promise objects, let us first learn how we can deal with promises using the built-in `fetch` function that allows us to make HTTP requests from the JavaScript code running in the browser.

When the `fetch` function is called, instead of making the calling code wait for the HTTP request to complete, it returns a promise object. We can associate callback functions with the returned promise object to execute code when the HTTP request is complete. We still use callbacks with promises, but the problems we discussed with callbacks in an earlier lesson in this module don't exist when using promises. Compared to callbacks, promises provide a clean and structured way to handle asynchronous operations in JavaScript.

```
1 const p1 = fetch(/* some url */);
```

The promise returned by the `fetch` function can be thought of as the `fetch` function *promising* us to supply a value when the HTTP request completes some time in the future. In the meantime, the main thread is free to do other things.

What can we do with the returned promise? We can register callbacks with the promise object that will be invoked when the network request completes. We can register separate callbacks to handle the success or failure of the network request.

Promise states

Promises can be in one of the following three states in their lifecycle:

- **pending:** the initial state in which promises typically start when they are created. It indicates that the asynchronous operation associated with the promise is in progress.
- **fulfilled:** means that the asynchronous operation associated with the promise has been completed successfully.
- **rejected:** means that the asynchronous operation associated with the promise has failed.

During the lifecycle of a promise, its state changes from pending to either fulfilled or rejected. The state of a promise is saved in the hidden internal slot named [[PromiseState]].

A promise in the pending state is considered *unsettled*. Once the promise transitions from the pending state into either a fulfilled or rejected state, it is said to have *settled*.

Promise instance methods

There are three instance methods we can call on promise instances:

- `Promise.prototype.then()`
- `Promise.prototype.catch()`

- `Promise.prototype.finally()`

then method

The `then` method is used to register a callback that is invoked asynchronously once the promise is fulfilled, i.e., the asynchronous operation wrapped by the promise completes successfully. This method allows us to execute code upon the successful completion of an asynchronous operation. Consider the following code example:

```
const p1 = fetch(/* some url */);

p1.then((response) => {
  // code to execute if the promise fulfills
});
```

The `then` method accepts two callback functions as arguments: the **fulfillment handler** and the **rejection handler**. The fulfillment handler is the first argument, as shown in the code example above. The rejection handler is the optional second argument that is invoked if the promise on which the `then` method is called gets rejected.

```
const p1 = fetch(/* some url */);

p1.then(
  (response) => {
    // code to execute if the promise fulfills
  },
  (error) => {
    // code to execute if the promise is rejected
  }
);
```

The fulfillment handler is passed the result of the asynchronous operation as an argument. In other words, the fulfillment handler receives the value with which the promise is fulfilled. In the case of an HTTP request, the promise is fulfilled with the server response, so the fulfillment handler receives the server response as an argument. On the other hand, the rejection handler receives the rejection reason as an argument if the promise is rejected.

catch method

We learned in the previous section that we can pass the rejection handler as the second argument to the `then` method to handle the promise rejection. There is another option to register the rejection handler, and that is through the `catch` method. Instead of passing the rejection handler to the `then` method, we can call the `catch` method on the promise to register the rejection handler.

```
const p1 = fetch(/* some url */);

p1.then((response) => {
  // code to execute if the promise fulfills
});

p1.catch((error) => {
  // code to execute if the promise is rejected
});
```

The `catch` method is similar to the `then` method that is called with only a rejection handler, as shown below:

```
p1.then(null, (error) => {
  // code to execute if the promise is rejected
});
```

However, using the `catch` method to register a rejection handler is more common than using the second argument of the `then` method.

finally method

Imagine a scenario where we want to send an HTTP request to a server, and while the request is in progress, we show a loading spinner to the user to indicate that data is being loaded. When the request completes, either successfully or unsuccessfully, we want to hide the loading spinner. To achieve this, we will have to duplicate the code that hides the loading spinner in the fulfillment and rejection handlers, as shown in the following code example:

```
const p1 = fetch(/* some url */);

p1.then((response) => {
  // hide the loading spinner
  document.getElementById("spinner").style.display = "none";
});

p1.catch((error) => {
  // hide the loading spinner
  document.getElementById("spinner").style.display = "none";
});
```

We want to avoid code duplication, and the `finally` method can help us remove the code duplication. The `finally` method allows us to execute code regardless of promise fulfillment or rejection. Just like the `then` and `catch` methods, the `finally` method also accepts a callback function that is invoked asynchronously after promise fulfillment as well as promise rejection. The callback passed to the `finally` method is the perfect place for the code that we want to execute regardless of whether the asynchronous operation fails or completes successfully. We can refactor the code above as shown below:

```
const p1 = fetch(/* some url */);

p1.then((response) => {
  // code to execute if the promise fulfills
});

p1.catch((error) => {
  // code to execute if the promise is rejected
});

p1.finally(() => {
  // hide the loading spinner
  document.getElementById("spinner").style.display = "none";
});
```

Unlike the callbacks of the `then` and `catch` methods, the callback passed to the `finally` method receives no arguments.

Creating promises

We can create new promise objects using the `Promise` constructor, as shown below:

```
const p = new Promise((resolve, reject) => {
  // initiate asynchronous operation...
});
```

The `Promise` constructor takes a callback function as an argument, referred to as the **executor** function, that is invoked *synchronously* to create the promise object. It is common to incorrectly assume that any code inside the executor function is executed asynchronously, but that is not the case. The executor function is invoked synchronously to create the promise object. The code inside the executor function should be any code that starts some asynchronous operation. The newly created promise object will observe that asynchronous operation. The promise object will notify us about the success or failure of the asynchronous operation that is initiated inside the executor function.

How is the asynchronous operation linked to the newly created promise? Through the `resolve` and `reject` functions that, the executor function receives as arguments. The parameters could be given different names, but it is common practice to name them `resolve` and `reject` to clearly indicate their purpose. The `resolve` function is used to resolve or fulfill the promise, and the `reject` function is used to reject the promise. Let us take a look at a concrete example that will clarify how we can create a promise object that wraps around an asynchronous operation and is resolved or rejected depending on whether the asynchronous operation succeeds or fails.

```
const p = new Promise((resolve, reject) => {
  const xhr = new XMLHttpRequest();

  xhr.addEventListener("load", function () {
    // check if operation is complete
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        // Request succeeded
        const data = JSON.parse(xhr.responseText);
        // call the resolve function with the data
        // as an argument to fulfill the promise
        // with the data
```

```
        resolve(data);
      } else {
        // Request failed
        const error = new Error("Failed to fetch todo");
        // call the reject function with the rejection
        // reason or an error as an argument
        reject(error);
      }
    }
  });

  const url = "https://jsonplaceholder.typicode.com/todos/1";
  xhr.open("GET", url);
  xhr.send();
});

// register fulfillment handler
p.then((todo) => {
  console.log(todo);
});

// register rejection handler
p.catch((error) => {
  console.log(error.message);
});
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promises-example9” />

The code example above shows how a promise object can be wrapped around an asynchronous operation and be resolved or rejected when the asynchronous operation succeeds or fails. In response to the promise being fulfilled or rejected, the appropriate handler (fulfillment or rejection) is invoked asynchronously.

It is important to note that the promise won’t be fulfilled or rejected until the `resolve` or `reject` functions are called within the executor function. As a result, any registered fulfillment or rejection handlers won’t be called until the promise is settled.

The code example above might not make you see how promises are an improvement over the traditional way of using callbacks to handle asynchronous code, but wait until we discuss promise chaining and `async-await` syntax in the upcoming lessons in this module. These two topics will help you see how promises address the two main problems with callbacks: Callback hell and error handling.

Using promises with callback-based API

If you noticed in the code example in the previous section, we just wrapped a promise around the callback-based `XMLHttpRequest` API. We wrapped a promise around it so that we could interact with it using promises.

Similar to what we did above, we can convert any callback-based API into a promise-based API. All we need to do is place the callback-based code in the executor function and call the `resolve` and `reject` functions at appropriate places to fulfill or reject the promise. Following is an example of wrapping a promise around `setTimeout` to add an artificial delay in the code:

```
function timeout(delayInSeconds) {
  const delayInMilliseconds = delayInSeconds * 1000;

  return new Promise((resolve) => {
    setTimeout(() => resolve(), delayInMilliseconds);
  });
}

timeout(2).then(() => {
  console.log("done"); // logged after 2 seconds
});
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promises-example10” />

In the code example above, we wrapped `setTimeout` in a promise to add an artificial delay in the code. To resolve the promise after the specified delay

(2 seconds in our code above), we call the `resolve` function inside the callback function of `setTimeout`. Note that we didn't call or use the `reject` function because we didn't need it to reject the promise. We just want the promise to be fulfilled after the specified delay.

Promise specification

Promises/A+ is a standard that defines the behavior of promises in JavaScript. It ensures that different implementations of promises in different environments conform to the standard behavior defined in the specification to ensure consistency in the behavior of promises across different environments.

Promise vs thenable

If you read the promise specification, you will find the word "**thenable**" mentioned multiple times. A thenable is any object that has defined a method named `then` but is not a promise. It is a generic term for objects with a method named `then`. As promises have a method named `then`, we can say that all promises are thenables, but the reverse is not true: every thenable is not a promise.

To summarize the difference, thenable is an object with a `then` method, and promise is an object with a `then` method that conforms to the Promises/A+ specification.

In the previous lesson, we discussed the following instance methods of promises:

- `Promise.prototype.then()`
- `Promise.prototype.catch()`
- `Promise.prototype.finally()`

We discussed what each of these methods allows us to do, but what we didn't discuss is what each of these methods returns. Their return value is

important because it allows for promise chaining, which is the topic of this lesson.

Each of the instance methods of promises returns a new promise, which enables us to create a chain of method calls, effectively creating a chain of asynchronous operations. Promise chaining helps resolve the two main problems we face when using callbacks: “Callback Hell” and error handling.

The following code example shows how we have registered fulfillment and rejection handlers with the promise returned by the `fetch` function:

```
const p = fetch(/* some url */);

// register a fulfillment handler
p.then((response) => {
  // code...
});

// register a rejection handler
p.catch((error) => {
  // code...
});
```

As each promise instance method returns a new promise, we can rewrite the above code as shown below:

```
fetch(/* some url */)
  .then((response) => {
    // code...
  })
  .catch((error) => {
    // code...
  });
```

The refactored code achieves the same result as the first code example, but technically, the first code example is different compared to the refactored code. In the first code example, the rejection handler is registered on the promise returned by the `fetch` function, whereas in the refactored code, the rejection handler is registered on the promise returned by the `then` method.

The promise chain in the refactored code can be split into different parts, as shown below to make it easier to understand:

```
const pFetch = fetch(/* some url */);

const pThen = pFetch.then((response) => {
  // code...
});

const pCatch = pThen.catch((error) => {
  // code...
});
```

Notice the promises for which fulfillment and rejection handlers have been registered. The fulfillment handler is registered on the promise returned by the `fetch` function, but unlike the first code example, the rejection handler is registered on the promise returned by the `then` method. So does it mean that if the promise returned by the `fetch` function is rejected, there is no rejection handler registered to handle the rejection? No, the rejection handler registered on the `pThen` promise will handle the rejection. To understand how it works, we have to understand how the promise chain works.

Further code examples in this lesson will use the following function that simulates an HTTP request that takes approximately two seconds to complete:

```
function fakeRequest(isSuccessRequest = true) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (isSuccessRequest) {
        const data = { name: "John Doe", favouriteLanguage:
"JavaScript" };
        resolve(data);
      } else {
        const error = new Error("request failed");
        reject(error);
      }
    }, 2000);
  });
}
```

This above function will make it easy for us to understand the promise chaining. The function takes a boolean parameter that specifies whether we want our fake request to be fulfilled or rejected. The default value of the parameter is `true`, so we only need to pass the argument if we want the request to fail. Inside the function, a promise is returned that wraps around the `setTimeout` to simulate a request that takes approximately two seconds to complete. After the timer expires, the promise is fulfilled or rejected, depending on the value of the `isSuccessRequest` parameter.

With the `fakeRequest` function defined, let us dive into the world of promise chaining.

then promise

Calling the `then` method on a promise registers a fulfillment handler on that promise. The `then` method itself returns a new promise that is different from the original promise on which the `then` method is called.

```
const pRequest = fakeRequest();

const pThen = pRequest.then((response) => {
  console.log(response);
});

console.log(pThen === pRequest); // false
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promise-chaining-example5” />

The promise returned by the `then` method fulfills or gets rejected based on the following two questions:

- What happens to the promise on which the `then` method is called? In our case, the `then` method is called on the `pRequest` promise.
- What is returned from the fulfillment or rejection handlers passed to the `then` method? Which handler will affect the promise returned by

the `then` method depends on which handler is invoked when the original promise on which the `then` method is called settles.

Keeping the above two questions in mind, let us discuss the different scenarios that can affect the promise returned by the `then` method:

Scenario 1: Original promise gets fulfilled

If the original promise on which the `then` method is called is fulfilled, the promise returned by the `then` method depends on what happens inside the fulfillment handler. Following are different things a fulfillment handler can do that affect the promise returned by the `then` method:

- If the fulfillment handler is registered and it returns a value that is not a promise or a thenable, then the promise returned by the `then` method gets fulfilled with that returned value.

```
// pRequest will be fulfilled
const pRequest = fakeRequest();

const pThen = pRequest.then((response) => {
  console.log(response);
  return "success";
});

pThen.then((data) => {
  console.log(data); // success
});
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promise-chaining-example6” />

- If the fulfillment handler doesn’t explicitly return any value, the promise returned by the `then` method is fulfilled with `undefined` as the fulfillment value.

```
// pRequest will be fulfilled
const pRequest = fakeRequest();
```

```
3
4 const pThen = pRequest.then((response) => {
5   console.log(response);
6 });
7
8 pThen.then((data) => {
9   console.log(data); // undefined
10 });
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example7" />

- If the `then` method is called on a promise but the fulfillment handler isn't provided, the promise returned by the `then` method gets fulfilled with the same fulfillment value as the original promise.

```
1 // pRequest will be fulfilled
2 const pRequest = fakeRequest();
3
4 // fulfillment handler not provided
5 const pThen = pRequest.then();
6
7 pThen.then((data) => {
8   // logs the value with which pRequest fulfilled
9   console.log(data);
10 });
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example8" />

- If the fulfillment handler throws any value or an error, the promise returned by the `then` method gets rejected with the thrown value as the rejection reason or value.

```
1 // pRequest will be fulfilled
2 const pRequest = fakeRequest();
3
4 const pThen = pRequest.then((response) => {
5   throw new Error("something bad happened");
```

```
});

pThen.catch((error) => {
  console.log(error.message); // something bad happened
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example9" />

- If the fulfillment handler returns a promise, the promise returned by the `then` method gets *resolved* to the promise returned by the fulfillment handler. One promise getting *resolved to another promise* simply means that the fate of one promise (let's call it `p1`) depends on the other promise (let's call it `p2`). If `p2` is fulfilled, `p1` also gets fulfilled with the same fulfillment value. If `p2` gets rejected, `p1` also gets rejected with the same rejection value. Promise `p1` will wait for `p2` to settle before it also settles and will eventually meet the same fate as `p2`.

```
// pRequest will be fulfilled
const pRequest = fakeRequest();

const pThen = pRequest.then((response) => {
  // return a promise that will be fulfilled
  return fakeRequest();
});

pThen.then((response) => {
  // logs the fulfillment value of
  // promise returned from the fulfillment
  // handler of pRequest promise
  console.log(response);
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example10" />

In the above code example, `pThen` promise returned by the `then` method gets *resolved* to the promise returned by the fulfillment handler of the `pRequest` promise.

Scenario 2: Original promise gets rejected

If the original promise on which the `then` method is called is rejected, the promise returned by the `then` method depends on the following scenarios:

- If only the fulfillment handler is passed to the `then` method, the promise returned by the `then` method also gets rejected with the same rejection reason or value with which the original promise was rejected.

```
// pRequest will get rejected
const pRequest = fakeRequest(false);

const pThen = pRequest.then((response) => {
  console.log(response);
});

pThen.catch((error) => {
  console.log(error.message); // request failed
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example11" />

- If a rejection handler is passed to the `then` method, then the promise returned by the `then` method depends on what happens inside the rejection handler. This works similarly to how it works in the case of the fulfillment handler:
  - If the rejection handler returns a non-promise value, the promise returned by the `then` method gets fulfilled with the value returned by the rejection handler.
  - If the rejection handler doesn't explicitly return any value, the promise returned by the `then` method is fulfilled with `undefined`

as the fulfillment value.
    - If the `then` method is called on the original promise but the rejection handler isn't provided, the promise returned by the `then` method gets rejected with the same rejection value as the original promise.
    - If the rejection handler throws any value or an error, the promise returned by the `then` method gets rejected with the thrown value as the rejection reason or value.
    - If the rejection handler returns a promise, the promise returned by the `then` method gets resolved to the promise returned by the rejection handler, the same as in the case of the fulfillment handler discussed earlier.

Now that we have discussed how the promise returned by the `then` method settles in different scenarios, next we will discuss the promise returned by the `catch` method.

catch promise

The `catch` method is used to register a rejection handler for a promise in which it is called. Just like the `then` method, the `catch` method also returns a new promise, and just like the `then` method promise, the promise returned by the `catch` method settles depending on the following two questions:

- What happens to the promise on which the `catch` method is called?
- What is returned from the rejection handler passed to the `catch` method?

Scenario 1: Original promise gets fulfilled

Suppose the original promise on which the `catch` method is called is fulfilled. In that case, the rejection handler registered using the `catch` method isn't called, and the promise returned by the `catch` method gets fulfilled with the same fulfillment value as the original promise.

```
// pRequest will be fulfilled
const pRequest = fakeRequest();

const pCatch = pRequest.catch((error) => {
  console.log(error.message);
});

pCatch.then((data) => {
  // logs the fulfillment value with
  // which pRequest promise fulfilled
  console.log(data);
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example12" />

Scenario 2: Original promise gets rejected

The rejection handler registered using the `catch` handler is called when the original promise on which the `catch` method is called gets rejected. If the original promise is rejected, then the promise returned by the `catch` method, just like the promise returned by the `then` method, depends on what happens inside the rejection handler:

- If the rejection handler returns a non-promise value, the promise returned by the `catch` method gets fulfilled with the value returned by the rejection handler.

```
// pRequest will get rejected
const pRequest = fakeRequest(false);

const pCatch = pRequest.catch((error) => {
  return "default value";
});

pCatch.then((data) => {
  console.log(data); // default value
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example13" />

- If the rejection handler doesn't explicitly return any value, the promise returned by the `catch` method is fulfilled with `undefined` as the fulfillment value.

```
// pRequest will get rejected
const pRequest = fakeRequest(false);

const pCatch = pRequest.catch((error) => {
  console.log(error.message); // request failed
});

pCatch.then((data) => {
  console.log(data); // undefined
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example14" />

- If the `catch` method is called on the original promise but the rejection handler isn't provided, the promise returned by the `catch` method gets rejected with the same rejection value as the original promise.

```
// pRequest will get rejected
const pRequest = fakeRequest(false);

// rejection handler not registered
const pCatch = pRequest.catch();

pCatch.catch((error) => {
  // logs the rejection value of the
  // original pRequest promise
  console.log(error.message); // request failed
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example15" />

- If the rejection handler throws any value or an error, the promise returned by the `catch` method gets rejected with the thrown value as the rejection reason or value.

```
// pRequest will get rejected
const pRequest = fakeRequest(false);

const pCatch = pRequest.catch((error) => {
  throw error;
});

pCatch.catch((error) => {
  console.log(error.message); // request failed
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example16" />

- If the rejection handler returns a promise, the promise returned by the `catch` method gets resolved to the promise returned by the rejection handler, the same as in the case of the fulfillment handler discussed earlier.

```
// pRequest will get rejected
const pRequest = fakeRequest(false);

const pCatch = pRequest.catch((error) => {
  // return a promise that will get fulfilled
  return fakeRequest();
});

pCatch.then((data) => {
  // logs the fulfillment value of
  // promise returned from the rejection
  // handler of pRequest promise
  console.log(data);
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example17" />

finally promise

The `finally` method is used to register a callback function that is invoked asynchronously after the promise settles. The callback passed to the `finally` method is invoked regardless of whether the promise on which it is called is fulfilled or rejected. Just like the `then` and `catch` methods, the `finally` method also returns a new promise, and the settlement of the promise returned by the `finally` method depends on the following two questions:

- What happens to the promise on which the `finally` method is called?
- What is returned from the callback function passed to the `finally` method?

Scenario 1: Original promise gets fulfilled

Suppose the original promise on which the `finally` method is called is fulfilled. In that case, the promise returned by the finally method also gets fulfilled with the same fulfillment value as the original promise, provided that the following conditions are met:

- The `finally` callback doesn't throw an error or a value.
- The `finally` callback doesn't return a rejected promise or a promise that eventually gets rejected.

```
1 // pRequest will get fulfilled
2 const pRequest = fakeRequest();
3
4 const pFinally = pRequest.finally(() => {
5   console.log("finally called");
6 });
7
8 pFinally.then((data) => {
```

```
  // logs the fulfillment value of
  // the original pRequest promise
  console.log(data);
});
```

Here’s the code in the Replit:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promise-chaining-example18” />

Note that the `finally` callback didn’t explicitly return any value, but the `finally` promise fulfilled with the fulfillment value of the original `pRequest` promise. This behavior is different than that of the `then` and `catch` methods; their promise is fulfilled with the value `undefined` if their callback implicitly returns `undefined`.

Scenario 2: Original promise gets rejected

Suppose the original promise on which the `finally` method is called is rejected. In that case, the promise returned by the `finally` method also gets rejected with the same rejection value as the original promise, provided that the same two conditions mentioned previously are met:

- The finally callback doesn’t throw an error or a value.
- The finally callback doesn’t return a rejected promise or a promise that eventually gets rejected.

```
// pRequest will get rejected
const pRequest = fakeRequest(false);

const pFinally = pRequest.finally(() => {
  console.log("finally called");
});

pFinally.catch((error) => {
  // logs the rejection value of
  // the original pRequest promise
  console.log(error.message);
});
```

Here’s the code in the Replit:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promise-chaining-example19” />

Scenario 3: shadowing original promise settlement

Unlike the `then` and `catch` methods, the return value of the `finally` callback is ignored, and the promise returned by the `finally` method simply meets the same fate as the original promise on which it is called; if the original promise fulfills, the `finally` promise also fulfills; if the original promise gets rejected, the `finally` promise also gets rejected.

However, the two conditions mentioned in the first two scenarios of the `finally` method are exceptions to this rule. If the `finally` callback throws an error, the `finally` promise gets rejected with the thrown value.

```
// pRequest will get rejected
const pRequest = fakeRequest(false);

const pFinally = pRequest.finally(() => {
  throw new Error("finally error");
});

pFinally.catch((error) => {
  console.log(error.message); // finally error
});
```

You can run the above code in the Replit below:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promise-chaining-example20” />

The rejection of the `finally` method shadowed the rejection of the original `pRequest` promise. The same thing will happen if the `pRequest` promise is fulfilled. Throwing an error from the `finally` callback simply rejects the `finally` promise, regardless of what happens to the original promise.

Similarly, returning a promise that is rejected from the `finally` callback rejects the `finally` promise, regardless of what happens to the original promise.

```
// pRequest will get fulfilled
const pRequest = fakeRequest();

const pFinally = pRequest.finally(() => {
  // return a promise that will get rejected
  return fakeRequest(false);
});

pFinally.catch((error) => {
  // logs the rejection value of the
  // promise returned from the finally callback
  console.log(error.message); // request failed
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example21" />

Making sense of promise chaining

We have discussed every scenario that can reject or fulfill the promise returned by each of the promise instance methods. Finally, we can make sense of how the promise chain works. We will go through a series of examples to solidify our understanding.

Example 1

```
fakeRequest()
  .then((response) => {
    console.log(response);
    return "hello world";
  })
  .then((data) => {
    console.log(data);
    return "123";
  })
  .catch((error) => {
```

```
  console.log(error.message);
});
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example22" />

What output do you expect in the above code example? Keep in mind that each instance method returns a new promise, and its fulfillment or rejection depends on two things:

- What happens to the original promise on which the method is called?
- What happens inside the callback function of that method?

Considering the different scenarios discussed above that can fulfill or reject the promise returned by each of the promise instance methods, try to test your understanding by guessing the output of the above code. Below is an explanation of the output produced by the above code:

1. Starting from the top of the promise chain, the promise returned by the `fakeRequest` function will be fulfilled, resulting in the invocation of its fulfillment handler that is registered using the first invocation of the `then` method. As a result, the following is logged on the console:

```
{
  favouriteLanguage: "JavaScript",
  name: "John Doe"
}
```

2. The promise returned by the `fakeRequest` function has been settled. The next promise in the chain is the one returned by the first `then` method call. As the original promise is fulfilled, the first `then` promise now depends on what happens inside its callback function. As it returns the string "hello world", the first `then` promise will get fulfilled with "hello world" as the fulfillment value.

As a result, its fulfillment handler is called, which was registered using the second `then` method call. The fulfillment handler of the first `then` promise receives its fulfillment value, i.e., “hello world”, as an argument. This results in “hello world” getting logged to the console. The console output so far is shown below:

```
{
 favouriteLanguage: "JavaScript",
 name: "John Doe"
}

"hello world"
```

3. At this point, two promises in the promise chain have settled. The next promise in the chain is the one returned by the second `then` method. Just like the first `then` method, the promise returned by the second `then` method depends on the original promise on which it is called, i.e., the promise returned by the first `then` method. As the original promise is fulfilled, the second `then` promise now depends on what happens inside its callback function. Its callback returns the string “123”. So the second `then` promise fulfills with “123” as its fulfillment value.

   But there is no fulfillment handler registered for the second `then` promise; only the rejection handler is registered using the `catch` method. As a result, no fulfillment handler is invoked in response to the fulfillment of the second `then` method. The promise chain moves to the last promise in the chain, i.e., the one returned by the `catch` method.

4. The promise returned by the `catch` method is called on the promise returned by the second `then` method. As the second `then` promise is fulfilled, the `catch` promise also gets fulfilled with the same fulfillment value as the second `then` promise. But there is no fulfillment or rejection handler registered for the `catch` promise, so its fulfillment is simply ignored. The final output of the code is shown below:

```
{
 favouriteLanguage: "JavaScript",
 name: "John Doe"
}

"hello world"
```

Example 2

```
fakeRequest()
  .then((response) => {
    console.log(response);
    return fakeRequest();
  })
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.log(error.message);
  });
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example23" />

Below is an explanation of the output produced by the above code:

1. Starting from the top of the promise chain, the promise returned by the `fakeRequest` function will be fulfilled, resulting in the invocation of its fulfillment handler that is registered using the first invocation of the `then` method. As a result, the following is logged on the console:

```
{
   favouriteLanguage: "JavaScript",
   name: "John Doe"
}
```

2. The promise returned by the `fakeRequest` function has been settled. The next promise in the chain is the one returned by the first `then` method call. As the original promise is fulfilled, the first `then` promise

now depends on what happens inside its callback function. As it is returning a new promise by calling the `fakeRequest` function, the first `then` promise will get *resolved* to the promise returned from its callback function. The `then` promise will wait for the promise returned from its callback function to settle before settling itself.

The promise returned from the callback function of the first `then` method will be fulfilled after approximately two seconds. As soon as it is fulfilled, the promise returned by the `then` method will also get fulfilled with the same fulfillment value as the promise returned by its callback. As a result, its fulfillment handler is called, which was registered using the second `then` method call. The fulfillment handler of the first `then` promise receives its fulfillment value as an argument, which is logged inside the fulfillment handler. The console output so far is shown below:

```
{
   favouriteLanguage: "JavaScript",
   name: "John Doe"
}

{
   favouriteLanguage: "JavaScript",
   name: "John Doe"
}
```

3. At this point, two promises in the promise chain have settled. The next promise in the chain is the one returned by the second `then` method. Just like the first `then` method, the promise returned by the second `then` method depends on the original promise on which it is called, i.e., the promise returned by the first `then` method. As the original promise is fulfilled, the second `then` promise now depends on what happens inside its callback function. Its callback implicitly returns `undefined`. So the second `then` promise fulfills with `undefined` as its fulfillment value.

   But there is no fulfillment handler registered for the second `then` promise; only the rejection handler is registered using the `catch`

method. As a result, no fulfillment handler is invoked in response to the fulfillment of the second `then` method. The promise chain moves to the last promise in the chain, i.e., the one returned by the `catch` method.

4. The promise returned by the `catch` method is called on the promise returned by the second `then` method. As the second `then` promise is fulfilled, the `catch` promise also gets fulfilled with the same fulfillment value as the second `then` promise. But there is no fulfillment or rejection handler registered for the `catch` promise, so its fulfillment is simply ignored. The final output of the code is shown below:

```
{
    favouriteLanguage: "JavaScript",
    name: "John Doe"
}

{
    favouriteLanguage: "JavaScript",
    name: "John Doe"
}
```

Example 3

```
fakeRequest(false)
  .then((response) => {
    console.log(response);
  })
  .catch((error) => {
    console.log(error.message);
  });
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example24" />

Below is an explanation of the output produced by the above code:

1. Starting from the top of the promise chain, the promise returned by the `fakeRequest` function will get rejected, but there is no rejection handler registered for this promise; only a fulfillment handler is registered. As a result, no rejection handler will be invoked for this promise. We move on to the next promise in the chain.
2. The promise returned by the `fakeRequest` function has been settled. The next promise in the chain is the one returned by the first `then` method call. As the original promise is rejected and no rejection handler was passed to the `then` method, the promise returned by the `then` method will also get rejected with the same rejection value as the original promise. As a result, its rejection handler, registered using the `catch` method, is invoked, logging the following on the console:

   ```
   1 "request failed"
   ```

   The error object with which the first promise got rejected is the same value with which the promise returned by the `then` method also got rejected. The promise chain moves to the last promise in the chain, i.e., the one returned by the `catch` method.

3. At this point, two promises in the promise chain have settled. The next promise in the chain is the one returned by the `catch` method. As the `then` promise is rejected, the promise returned by the `catch` method depends on what happens inside its callback. Its callback implicitly returns `undefined`, resulting in the `catch` promise getting fulfilled with `undefined` as a fulfillment value. But there is no fulfillment handler registered for the `catch` promise, so its fulfillment is simply ignored. The final output of the code is shown below:

   ```
   1 "request failed"
   ```

One thing to note in the above code example is that the rejection of the promise returned by the `fakeRequest` function was eventually handled by the rejection handler registered for the promise returned by the `then` method. This is one of the powers of promise chaining. Unlike callbacks, where we had to check for the error in every callback, with promise

chaining, we can register one rejection handler, and it can handle the rejection of all the promises that come before it in the promise chain. We could have multiple `then` method calls in the promise chain and only one rejection handler at the end of the promise chain, registered using the `catch` method. This makes error handling easy to manage in a promise chain.

Example 4

```
fakeRequest()
  .then((response) => {
    return fakeRequest(false);
  })
  .catch((error) => {
    return { data: "default data" };
  })
  .then((data) => {
    console.log(data);
  })
  .then(() => {
    throw new Error("error occurred");
  })
  .catch((error) => {
    console.log(error.message);
  });
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-chaining-example25" />

Below is an explanation of the output produced by the above code:

1. Starting from the top of the promise chain, the promise returned by the `fakeRequest` function will be fulfilled, resulting in the invocation of its fulfillment handler that is registered using the first invocation of the `then` method, passing the fulfillment value to its fulfillment handler as an argument.
2. The promise returned by the `fakeRequest` function has been settled. The next promise in the chain is the one returned by the first `then` method call. As the original promise is fulfilled, the first `then` promise

now depends on what happens inside its callback function. As it is returning a new promise by calling the `fakeRequest` function, the first `then` promise will get *resolved* to the promise returned from its callback function. The `then` promise will wait for the promise returned from its callback function to settle before settling itself.

The promise returned from the callback function of the first `then` method will be rejected after approximately two seconds. As soon as it is rejected, the promise returned by the `then` method will also get rejected with the same rejection value as the promise returned by its callback. As a result, its rejection handler is called, which was registered using the first `catch` method call. The rejection handler of the first `then` promise receives its rejection value as an argument.

3. At this point, the first two promises in the promise chain have settled. The next promise in the chain is the one returned by the first `catch` method. As the promise on which the `catch` method is called gets rejected, the first `catch` promise now depends on what happens inside its callback function. It returns an object literal. As a result, the first `catch` promise fulfills the returned object as its fulfillment value.

   Note that the `catch` method doesn't necessarily have to be at the end of the promise chain; however, the `catch` method is most commonly placed at the end of the promise chain. Depending on the requirement, the `catch` method can be placed anywhere in the chain. In our code example, it is called after the first `then` method to handle the possible rejection of the first `then` promise by returning the default data and letting the chain continue. If the rejection of the first `then` promise wasn't handled, all the `then` promises after the first `then` method would also be rejected with the same rejection value as the first `then` promise, and the rejection would finally be handled in the last `catch` method call.

4. At this point, the first three promises in the promise chain have settled. The next promise in the chain is the one returned by the second `then` method call. As the promise (the first `catch` promise) on which the

second `then` method is called is fulfilled, the promise returned by the second `then` method now depends on what happens inside its callback function. Its callback logs the fulfillment value of the first `catch` promise and implicitly returns undefined, resulting in the second `then` promise getting fulfilled with `undefined` as the fulfillment value. Following is the console output up to this point:

```
{
  data: "default data"
}
```

5. At this point, the first four promises in the promise chain have settled. The next promise in the chain is the one returned by the third `then` method call. As the promise (the second `then` promise) on which the third `then` method is called is fulfilled, the promise returned by the third `then` method now depends on what happens inside its callback function. Its callback throws an error, resulting in the third `then` promise getting rejected with the thrown error as the rejection reason or value. As a result, its rejection handler, registered using the last `catch` method, is invoked, passing in the rejection value as an argument.
6. As the promise (the third `then` promise) on which the last `catch` method is called gets rejected, the last `catch` promise depends on what happens inside its callback function. Its callback logs the rejection value of the third `then` promise and implicitly returns `undefined`, resulting in the last `catch` promise getting fulfilled with `undefined` as the fulfillment value. But there is no fulfillment handler registered for the last `catch` promise, so its fulfillment is simply ignored. The final console output of the code is shown below:

```
{
  data: "default data"
}

"error occurred"
```

Hopefully, the examples above, along with the earlier discussion in this lesson on different scenarios that can reject or fulfill the promise returned by each of the promise instance methods, have laid a solid foundation for understanding the promise chains and making use of them in your own code.

Rejection handler in then vs catch

In the previous lesson, it was mentioned that a rejection handler can be registered by passing a second argument to the `then` method, as shown below:

```
fakeRequest().then(null, (error) => {
  // handle error
});
```

There is one thing that should be kept in mind when registering a rejection handler using the `then` method: the rejection handler registered using the `then` method is not invoked if the promise returned by the `then` method, to which the rejection handler is passed as an argument, gets rejected. The following code example shows this in action:

```
fakeRequest().then(
  (response) => {
    // rejects the promise returned
    // by the "then" method
    throw new Error("error");
  },
  (error) => {
    // this callback is not invoked
    console.log(error.message);
  }
);
```

The rejection handler registered using the `then` method is only invoked if the original promise on which the `then` method is called gets rejected. As a result, we have an *unhandled promise rejection* in the above code example, which, in the worst case, can terminate the program. As a result, always

remember to handle all the possible promise rejections in your code when working with promises.

In this lesson, we will discuss a couple of common use cases of the two static methods of promises and learn how they can be useful. Other promise static methods are also useful, but in my opinion, the following two use cases are the most common ones:

- Making concurrent requests
- Implementing request timeout

Concurrent requests

Imagine a scenario where we want to initiate multiple HTTP requests all at once and wait for their collective results. We cannot use a promise chain, as shown below, because it will execute each request in a sequential manner.

```
const url1 = "https://jsonplaceholder.typicode.com/todos/1";
const url2 = "https://jsonplaceholder.typicode.com/todos/2";
const url3 = "https://jsonplaceholder.typicode.com/todos/3";

function parseFetchResponse(response) {
  if (response.ok) {
    return response.json(); // returns a promise
  } else {
    throw new Error("request failed");
  }
}

fetch(url1)
  .then(parseFetchResponse)
  .then((data1) => {
    console.log(data1);
    // initiate second request
    return fetch(url2);
  })
  .then(parseFetchResponse)
  .then((data2) => {
    console.log(data2);
    // initiate third request
    return fetch(url3);
  })
  .then(parseFetchResponse)
```

```
  .then((data3) => {
    console.log(data3);
  })
  .catch((error) => {
    console.log(error.message);
  });
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/static-promise-methods-example1" />

As each request in the above code is independent, instead of initiating the requests in a sequential manner, we want concurrent requests. This can be achieved using the `Promise.all` method. It allows us to start each request one after the other without waiting for one request to complete before starting the other one. As a result, all three requests are initiated concurrently, and we can wait for their collective results.

The `Promise.all` method takes an iterable of promises as an input and returns a promise that fulfills once all the promises passed to it are fulfilled. The fulfillment value of the promise returned by this method is an array of fulfillment values of all the promises passed to this method as input. The promise this method returns gets rejected if any of the input promises gets rejected. We can rewrite the above code example using the `Promise.all` method as shown below:

```
const url1 = "https://jsonplaceholder.typicode.com/todos/1";
const url2 = "https://jsonplaceholder.typicode.com/todos/2";
const url3 = "https://jsonplaceholder.typicode.com/todos/3";

function parseFetchResponse(response) {
  if (response.ok) {
    return response.json();
  } else {
    throw new Error("request failed");
  }
}

Promise.all([
  fetch(url1).then(parseFetchResponse),
  fetch(url2).then(parseFetchResponse),
```

```
  fetch(url3).then(parseFetchResponse)
])
  .then((dataArr) => {
    console.log(dataArr);
  })
  .catch((error) => console.log(error.message));
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/static-promise-methods-example2" />

Request timeout

An HTTP request can sometimes just hang due to some problem on the server. We do not want the request to be in a pending state for longer than a few seconds. To avoid longer request pending times, we can implement the request timeout feature, which lets our code know that a request is taking longer than expected. This allows us to take the appropriate action.

Using the `Promise.race` method, we can initiate an HTTP request with a timeout. This method, just like the `Promise.all` method, takes an iterable of promises and returns a promise that fulfills when any of the promises out of one or more promises provided to this method as an input fulfills. Similarly, the promise returned by this method is rejected when any of the input promises is rejected.

The following code example shows the request timeout implemented using the `Promise.race` method:

```
// simulate a request that takes
// approximately 8 seconds to complete
function delayedRequest() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("hello world");
    }, 8000);
  });
}

// timeout promise that is rejected
```

```
// after approximately 3 seconds
function timeout() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const error = new Error("request timed out");
      reject(error);
    }, 3000);
  });
}

Promise.race([delayedRequest(), timeout()])
  .then((response) => {
    console.log(response);
  })
  .catch((error) => console.log(error.message));
```

You can run the above code in the Replit below:

<ReplitEmbed src="https://replit.com/@newlineauthors/static-promise-methods-example3" />

In this lesson, we discussed only two static methods, but it's worth learning about the other static methods available on the `Promise` constructor. Each static method has its own use cases; we just discussed two use cases that I think are the most commonly needed.

While promises have changed the way we write asynchronous code in JavaScript, we still use callbacks to register fulfillment and rejection handlers with promises. Some people might view using callbacks with promises as verbose, even though promises solve the problem of "Callback Hell" and the problem of error handling using the traditional way of using callbacks. What if there was an easier, more concise, and more intuitive way to deal with promises? What if we could get rid of callbacks when using promises? Enter `async await`!

The `async await` can be considered a syntax sugar over the traditional way of using promises. It allows us to deal with promises using code that executes asynchronously but looks synchronous. It also allows us to write more concise code that is easier to reason about, as the code doesn't include callbacks, and the flow of the code looks like that of synchronous code.

Let’s take a look at the following code example that uses promise chaining:

```
function fetchTodo(url) {
  fetch(url)
    .then((response) => {
      if (response.ok) {
        return response.json();
      } else {
        throw new Error("request failed");
      }
    })
    .then((data) => {
      console.log(data);
    })
    .catch((error) => {
      console.log(error.message);
    });
}

const url = "https://jsonplaceholder.typicode.com/todos/1";
fetchTodo(url);
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example1” />

The code above is certainly an improvement over the traditional way of using callbacks for writing asynchronous code, but it still has room for improvement, at least in terms of readability. The `async await` syntax can be used to rewrite the above code example as shown below:

```
async function fetchTodo(url) {
  try {
    const response = await fetch(url);

    if (response.ok) {
      const data = await response.json();
      console.log(data);
    } else {
      throw new Error("request failed");
    }
  } catch (error) {
    console.log(error.message);
  }
```

```
}

const url = "https://jsonplaceholder.typicode.com/todos/1";
fetchTodo(url);
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example2” />

The revised code achieves the same result but is more readable, doesn’t use any callbacks, and is easier to reason about as compared to the earlier example that uses promise chaining. Although the code looks like synchronous code, it is asynchronous. Let us understand how `async await` works.

Two things should be noted in the code example above that uses the `async await` syntax: the `async` keyword in the function signature and the `await` keyword inside the function. Following are the two main steps to using the `async await` syntax:

1. Mark any function as “async” using the `async` keyword. This is needed because the `await` keyword can only be used inside an “async” function.
2. Use the `await` keyword inside the `async` function to wait for any promises to settle.

:::info While the `await` keyword is mostly used inside an `async` function because the `await` keyword was only allowed inside `async` functions until a recent change in the language that allows using the [await keyword at the top-level of a module]

:::

async functions

An `async` function allows the use of the `await` keyword inside its body. An `async` function is different from a non-async function because an `async` function always returns a promise. An `async` function implicitly creates and returns a promise, similar to how each promise instance method creates and returns a new promise. The following code verifies this claim:

```
async function foo() {}

const result = foo();
console.log(result instanceof Promise); // true
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example3” />

The fulfillment or rejection of the promise returned by an `async` function depends on what happens inside its body, similar to how the promise returned by each of the promise instance methods depends on the events occurring within its callback function. The following points summarize the settlement of the `async` function promise:

- Returning any non-promise value from an `async` function leads to the fulfillment of the `async` function promise, using the returned value as the fulfillment value.

  ```
  async function foo() {
    return 123;
  }

  foo().then(console.log); // 123
  ```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example4” />

- Not returning any value from the function implicitly returns `undefined`. This leads to the function promise getting fulfilled with

`undefined` as the fulfillment value.

```
async function foo() {}

foo().then(console.log); // undefined
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/async-await-example5" />

- Throwing an error inside the `async` function rejects the `async` function promise, using the thrown value as the rejection reason.

```
async function foo() {
  throw new Error("some error occurred");
}

foo().catch((error) => console.log(error.message)); // some error occurred
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/async-await-example6" />

- Returning a promise from the `async` function results in the `async` function promise getting *resolved* to the promise returned inside the function body. As we learned about one promise *resolving* to another promise in one of the earlier lessons in this module, the promise created by the `async` function will wait for the promise, returned inside its body, to settle. Eventually, the `async` function promise will be fulfilled or rejected depending on what happens to the promise returned inside the `async` function.

```
// returns a promise that is fulfilled
// after approximately 2 seconds
function getPromise() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("hello world");
```

```
    }, 2000);
  });
}

async function foo() {
  return getPromise();
}

foo().then(console.log); // hello world
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example7” />

await keyword

The `await` keyword, also referred to as the `await` operator, is used to wait for a promise to settle. The following is an example of using the `await` keyword to wait for a promise to settle:

```
// assume that the following statement
// is inside an async function
const response = await fetch(url);
```

The `await fetch(url)` is an expression that will either evaluate the fulfillment value of the promise returned by the `fetch` function or it will throw the rejection value if the promise returned by the `fetch` function gets rejected. The thrown value can either be caught in the `catch` block of the surrounding `try-catch` block or, if `try-catch` is not wrapped around the `await` statement, rejection of the *awaited* promise can reject the `async` function promise, allowing the calling code to handle the promise rejection.

Unlike promise chaining, where we have to register the fulfillment handler to get the fulfillment value of the promise, the `await` expressions evaluate the promise fulfillment value, which we can save in a variable. But how does it work? Isn’t it blocking the main thread while waiting for the promise to settle?

Whenever an `async` function is called, it is executed synchronously until the first `await` expression is encountered. The function's execution is suspended or paused until the awaited promise is settled. Instead of blocking the main thread, the function's execution is paused, and in the meantime, the main thread is free to do other things. When the promise is eventually settled, the function's execution is resumed, resuming the code execution after the `await` expression if the promise is fulfilled or throwing the rejection value of the promise if the awaited promise is rejected.

What's important to note is that the code inside the `async` function is executed synchronously until the first `await` expression. What if the `async` function doesn't have the `await` keyword inside it? Will the function execute synchronously? Yes, it will, but keep in mind that the `async` function always returns a promise, and it will either get fulfilled or rejected depending on what happens inside the `async` function. This means that the following code doesn't work as one might expect:

```
async function foo() {
  return "123";
}

const result = foo();
```

The `async` function in the above code example didn't use the `await` keyword, so the code inside it is executed synchronously, but does it return the value "123" synchronously as well? No, it doesn't. The function is `async`, which means it returns a promise, so the `result` in the above example contains the promise and not the value "123". To get the fulfillment value of the promise, we can either use promise chaining as shown below:

```
async function foo() {
  return "123";
}

foo().then(console.log); // "123"
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/async-await-example10" />

Or `await` the promise returned by the `foo` function using the `async await` syntax as shown below:

```
async function foo() {
  return "123";
}

async function bar() {
  const result = await foo();
  console.log(result); // "123"
}

bar();
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/async-await-example11" />

Multiple await expressions

An `async` function is not limited to using the `await` keyword only once inside its body. You can use the `await` keyword as many times as you want inside an `async` function. The only thing to be aware of regarding multiple `await` expressions is that they are not executed in parallel; instead, they are executed in sequence, one after the other. The function execution will be paused at each `await` expression, and the next `await` expression can only be executed after the ones before it has been executed.

```
// returns a promise that is fulfilled
// after approximately 1 second
function promisifiedRandomNumber() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // generate a random number within range: 0 - 9
      const randomNum = Math.floor(Math.random() * 10);
      resolve(randomNum);
    }, 1000);
```

```
  });
}

async function random() {
  const num1 = await promisifiedRandomNumber();
  const num2 = await promisifiedRandomNumber();
  const num3 = await promisifiedRandomNumber();

  console.log(num1, num2, num3);
}

random();
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example12” />

Each of the `await` expressions in the above code example takes approximately 1 second, so the function takes approximately 3 seconds to evaluate all the `await` expressions, logging their value at the end.

```
// returns a promise that is fulfilled
// after approximately 1 second
function promisifiedRandomNumber() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // generate a random number within range: 0 - 9
      const randomNum = Math.floor(Math.random() * 10);
      resolve(randomNum);
    }, 1000);
  });
}

async function random() {
  const randomSum =
    (await promisifiedRandomNumber()) + (await
promisifiedRandomNumber());
  console.log(randomSum);
}

random();
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example13” />

The two `await` expressions in the above code example are also not executed in parallel; instead, they are executed one after the other, from left to right.

Suppose we want concurrent asynchronous operations inside an `async` function. In that case, we can use the `Promise.all` function, providing all the promises as input and awaiting the promise returned by `Promise.all`.

```
// returns a promise that is fulfilled
// after approximately 1 second
function promisifiedRandomNumber() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // generate a random number within range: 0 - 9
      const randomNum = Math.floor(Math.random() * 10);
      resolve(randomNum);
    }, 1000);
  });
}

async function random() {
  const promiseArr = [
    promisifiedRandomNumber(),
    promisifiedRandomNumber(),
    promisifiedRandomNumber()
  ];
  const randomNumsArr = await Promise.all(promiseArr);
  console.log(randomNumsArr);
}

random();
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example14” />

Error handling

To handle promise rejections inside an `async` function, we can wrap the `await` expressions with the `try-catch` block as shown below:

```
async function getUsersAndTasks() {
  try {
    const users = await fetchUsers();
    const tasks = await fetchTasks();
  } catch (error) {
    // handle the error
  }
}
```

If any of the promises awaited in the `try` block are rejected, the code after that `await` expression won't be executed, and the execution will jump to the `catch` block. The `await` keyword throws the promise rejection value, allowing the `catch` block to catch the rejection.

Alternatively, we can omit the `try-catch` block, but in this case, the code that calls the `async` function must handle the promise rejection, either by using the promise chaining:

```
async function getUsersAndTasks() {
  const users = await fetchUsers();
  const tasks = await fetchTasks();

  // do something with users and tasks.
}

getUsersAndTasks().catch((error) => {
  /* handle the error */
});
```

or using the `try-catch` block in the calling code if we are using the `async await` syntax:

```
async function getUsersAndTasks() {
  const users = await fetchUsers();
  const tasks = await fetchTasks();

  // do something with users and tasks.
}

async function initApp() {
```

```
  try {
    await getUsersAndTasks();
  } catch (error) {
    // handle the error
  }
}
```

Returning vs awaiting promise

Forgetting to `await` a promise inside an `async` function can lead to bugs, causing unexpected output. The code below shows the problem it can cause:

```
// returns a promise that either
// fulfills or gets rejected randomly
function getPromise() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() < 0.5) {
        resolve("success");
      } else {
        reject(new Error("failed"));
      }
    }, 1000);
  });
}

async function foo() {
  getPromise();
}

foo()
  .then(() => console.log("foo promise fulfilled"))
  .catch(() => console.log("foo promise rejected"));
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example18” />

The `getPromise` function returns a promise that can be rejected, but the promise returned by the `foo` function is always fulfilled. Why? The `foo` function has a problem: it didn’t return or `await` the promise returned by the

`getPromise` function. As a result, the `foo` function doesn't wait for the promise returned by the `getPromise` function to settle; instead, it just calls the `getPromise` function, and the function execution ends, implicitly returning `undefined`, leading to the `foo` function promise getting fulfilled with `undefined` as the fulfillment value.

Further code examples will use the `getPromise` function defined above.

To catch the rejection of the promise returned by the `getPromise` function, we have the following options:

- return the promise returned by the `getPromise` function.

```
async function foo() {
  return getPromise();
}

foo()
  .then(() => console.log("foo promise fulfilled"))
  .catch(() => console.log("foo promise failed"));
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/async-await-example19" />

Returning the promise *resolves* the `foo` function of the promise returned inside its body. As a result, whatever happens to the promise returned by `getPromise`, the `foo` function promise meets the same fate.

- await the promise returned by the `getPromise` function.

```
async function foo() {
  await getPromise();
}

foo()
  .then(() => console.log("foo promise fulfilled"))
  .catch(() => console.log("foo promise failed"));
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example20” />

As mentioned earlier, if the awaited promise is rejected, its rejection value is thrown inside the `async` function. As there is no `try-catch` block inside the `foo` function, the promise rejection causes the `foo` function promise to also get rejected with the same rejection reason.

However, one thing to note in this code example is that if the promise returned by `getPromise` is fulfilled, the `foo` function promise doesn’t fulfill with its fulfillment value; instead, it fulfills with `undefined` as the fulfillment value because we didn’t explicitly return anything from the `foo` function, and we know what happens to the `async` function promise when we don’t explicitly return any value inside the function: the `async` function promise gets fulfilled with `undefined` as the fulfillment value.

- await the promise returned by the `getPromise` function and surround it with the `try-catch` block.

```
async function foo() {
  try {
    await getPromise();
  } catch (error) {
    console.log("inside catch block of foo function");
    return "error caught in foo";
  }
}

foo()
  .then(() => console.log("foo promise fulfilled"))
  .catch(() => console.log("foo promise failed"));
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-await-example21” />

Awaiting the `getPromise` call will catch the promise rejection, causing the `catch` block to execute. However, the promise returned by the `foo` function will always be fulfilled. Why? Because the `catch` block didn't throw an error or return a promise that gets rejected. As a result, the `foo` function promise always fulfills with the return value of the `catch` block. We can throw the error from the `catch` to fix this problem. Having said that, if all we do inside the `catch` block is throw the error, then it's better to just omit the `try-catch` block and let the promise rejection automatically reject the `foo` function promise.

Another problem in this code example is that we didn't explicitly return any value in case the promise was fulfilled and the `catch` block was never executed. So the `foo` function promise will be fulfilled with `undefined`. Adding the `return` keyword before the `await` expression will do the job.

:::caution Can't we just do `return getPromise();` instead of `return await getPromise();`? We could have if the `await` expression wasn't wrapped in the `try-catch` block. What difference does `try-catch` make? With the `try-catch` block, `return getPromise();` will lead to the `catch` block inside the `foo` function never executing. For the `catch` block inside the `foo` function to execute, we need to `await` the promise inside the `try` block instead of just returning it. For more details, read: [await vs return vs return await](#) :::

Awaiting non-promise value

The `await` keyword is usually used to wait for a promise to settle, but it can also be used with a non-promise value. The following code example shows this action:

```
const printRandomNumber = async () => {
  const randomNum = await Math.floor(Math.random() * 10);
  console.log(randomNum);
};

printRandomNumber();
```

```
7
8 console.log("before printing random number");
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/async-await-example22" />

If you execute the above code example, you will note that the `console.log` statement at the end of the code example is logged *before* the random number is printed, even though the function is called *before* the last `console.log` statement. Why is that? We haven't awaited any promises, so what's happening here?

When the `await` keyword is used with a non-promise value, a new promise is created, and that promise is fulfilled with the value we used with the `await` keyword. In our code example, we have awaited a random number; it is not a promise, so a new promise is created and fulfilled with the generated random number. Code after the `await` expression is executed as though it were in a fulfillment handler. As a result, when the promise is fulfilled, the code after the `await` expression is not immediately executed. It is executed asynchronously, and as we learned in the lesson about event loop, any asynchronous code is only executed after the synchronous execution of our code ends. The last `console.log` statement is executed as part of the synchronous execution of our code. As a result, it is logged before the random number.

Using `await` with a non-promise value is hardly useful, but just be aware that it is possible, and the value is implicitly wrapped in a promise.

We learned in one of the earlier lessons in this module that executing DOM event listeners and `setTimeout` or `setInterval` callbacks require scheduling a "task". Tasks are enqueued in a task queue until the event loop processes them. What about the promise fulfillment or rejection handlers? Does their execution also require scheduling a task? Not exactly a task, but a "microtask".

Microtasks, which the ECMAScript specification calls jobs, are scheduled for things that have higher priority than “tasks”. Microtasks are processed after:

- each callback, provided that the callstack is empty.
- each task

Whereas “tasks” are executed in the order they are enqueued in the task queue, only one task is executed per one turn or tick of the event loop.

Another important thing to note about microtasks is that while only one task is processed per tick of the event loop, microtasks are processed until the microtask queue is empty. If a task schedules another task, it won’t be processed until the next turn or tick of the event loop, but in the case of microtasks, if any microtask is queued by a microtask, the queued microtask will also be processed. This means that the event loop can get stuck in an infinite loop if each microtask keeps queuing another microtask.

Consider the following code example:

```
1 console.log("start");
2
3 setTimeout(() => {
4   console.log("setTimeout callback with 500ms delay");
5 }, 500);
6
7 Promise.resolve()
8   .then(() => {
9     console.log("first 'then' callback");
10  })
11  .then(() => {
12    console.log("second 'then' callback");
13  })
14  .then(() => {
15    console.log("third 'then' callback");
16  });
17
18 setTimeout(() => {
19  console.log("setTimeout callback with 0ms delay");
20 }, 0);
21
22 console.log("end");
23
```

```
/*
start
end
first 'then' callback
second 'then' callback
third 'then' callback
setTimeout callback with 0ms delay
setTimeout callback with 500ms delay
*/
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/microtasks-example1” />

Executing the above code requires scheduling tasks and microtasks. The following steps explain how different tasks and microtasks are scheduled to execute the code above:

1. A task is created to execute the script, starting the synchronous execution of the code.
2. The first `console.log` statement is executed, logging “start” on the console.

```
output:
-------
start
```

3. Next, we have the `setTimeout` call with a 500ms delay. This starts a timer in the background, and its expiration will result in a task to execute the `setTimeout` callback getting queued in the task queue.

```
task queue:
-----------
[task(execute setTimeout callback)]

output:
-------
start
```

4. Moving on with the synchronous execution of the code, Promise.resolve is called, which creates a resolved promise. To execute its fulfillment handler, a microtask or job is queued in the microtask queue.

```
task queue:
-----------
[task(execute setTimeout callback)]

microtask queue:
----------------
[job(execute fulfillment callback)]

output:
-------
start
```

5. Next, we have a `setTimeout` call with a 0ms delay. This also schedules a task to execute its callback.

```
task queue:
-----------
[
    task(execute setTimeout callback),
    task(execute setTimeout callback)
]

microtask queue:
----------------
[job(execute fulfillment callback)]

output:
-------
start
```

6. Finally, the synchronous execution reaches its end with the final `console.log` statement, logging “end” on the console. At this point, the callstack is empty, and the event loop can start processing the scheduled tasks and microtasks.

```
task queue:
-----------
[
    task(execute setTimeout callback),
```

```
    task(execute setTimeout callback)
]

microtask queue:
----------------
[job(execute fulfillment callback)]

output:
-------
start
end
```

7. As mentioned earlier, microtasks are processed after each task as well as after each callback, provided that the callstack is empty. The synchronous execution of the code is a task, and when it ends, the callstack is empty, so any microtasks in the microtask queue are ready to be processed by the event loop. We have only one microtask in the microtask queue. It will be processed by logging “first ‘then’ callback” on the console.

```
task queue:
-----------
[
    task(execute setTimeout callback),
    task(execute setTimeout callback)
]

microtask queue:
----------------
[]

output:
-------
start
end
first 'then' callback
```

8. The callback function of the first `then` method implicitly returns `undefined`, and as a result, the promise returned by the `then` method is fulfilled with `undefined` as the fulfillment value. This queues another microtask in the microtask queue to execute the fulfillment handler of the promise returned by the first `then` method.

```
task queue:
-----------
[
    task(execute setTimeout callback),
    task(execute setTimeout callback)
]

microtask queue:
----------------
[job(execute fulfillment callback)]

output:
-------
start
end
first 'then' callback
```

9. As mentioned earlier, microtasks are processed until the microtask queue is empty, so the newly queued microtask will also be processed, logging “second ‘then’ callback” on the console.

```
task queue:
-----------
[
    task(execute setTimeout callback),
    task(execute setTimeout callback)
]

microtask queue:
----------------
[]

output:
-------
start
end
first 'then' callback
second 'then' callback
```

10. Similar to step 8, the callback function of the second `then` method implicitly returns `undefined`, and as a result, the promise returned by the `then` method is fulfilled with `undefined` as the fulfillment value. This queues another microtask in the microtask queue to execute the

fulfillment handler of the promise returned by the second `then` method.

```
task queue:
-----------
[
    task(execute setTimeout callback),
    task(execute setTimeout callback)
]

microtask queue:
----------------
[job(execute fulfillment callback)]

output:
-------
start
end
first 'then' callback
second 'then' callback
```

11. This results in the “third ‘then’ callback” getting logged on the console.

```
task queue:
-----------
[
    task(execute setTimeout callback),
    task(execute setTimeout callback)
]

microtask queue:
----------------
[]

output:
-------
start
end
first 'then' callback
second 'then' callback
third 'then' callback
```

12. We didn’t do anything with the promise returned by the third `then` method, so its fulfillment is ignored. At this point, all the microtasks

have been processed, and the microtask queue is empty. The event loop can now process the first task in the task queue.

13. The first task in the task queue is that of the second `setTimeout` call because it had less delay than the first one, so it was queued before the task of the other `setTimeout` callback, which had a 500ms delay. Processing it results in a “setTimeout callback with 0ms delay” being logged on the console.

```
task queue:
-----------
[task(execute setTimeout callback)]

microtask queue:
----------------
[]

output:
-------
start
end
first 'then' callback
second 'then' callback
third 'then' callback
setTimeout callback with 0ms delay
```

14. Finally, the last task in the task queue is that of the first `setTimeout` call with a 500ms delay, resulting in “setTimeout callback with 500ms delay” getting logged on the console.

```
task queue:
-----------
[]

microtask queue:
----------------
[]

output:
-------
start
end
first 'then' callback
second 'then' callback
third 'then' callback
```

```
setTimeout callback with 0ms delay
setTimeout callback with 500ms delay
```

:::note In this module, we have used the term “resolved” to refer to a promise that is waiting for another promise to settle. In other words, we have used the term “resolved” to refer to a promise in a pending state.

Having said that, the term “resolved” can also be used to refer to a promise that has either been fulfilled or rejected. For more details, read: promises-unwrapping - States and Fates

:::

Further reading

The following are links to some of the Stackoverflow questions that I answered that are related to microtasks and explain the execution of code examples similar to the one discussed above:

- How to explain the output order of this code snippet?
- JavaScript quiz of printing sequence with a combination of promise.then and async function
- Promise chain .then .catch
- Asynchronous Execution Order in JavaScript

The following is an article that explains the execution of tasks and microtasks with the help of interactive examples:

- Tasks, microtasks, queues, and schedules

In this lesson, we will discuss common promise-related anti-patterns that should be avoided. Following is a list of anti-patterns we will discuss:

- Unnecessary use of the Promise constructor
- Incorrect error handling
- Converting promise rejection into fulfillment
- Async executor function

Unnecessary use of the Promise Constructor

One of the most common mistakes made by JavaScript developers, especially those who don't have much experience with promises, is creating promises unnecessarily using the `Promise` constructor function. Let's take a look at an example:

```
function fetchData(url) {
  return new Promise((resolve, reject) => {
    fetch(url)
      .then((res) => res.json(res))
      .then(resolve)
      .catch(reject);
  });
}
```

The above code will work if you pass a URL to the `fetchData` function and then wait for the promise to resolve, but the use of the `Promise` constructor is unnecessary in the above code example. The `fetch` function already returns a promise, so instead of wrapping the `fetch` function call with the promise constructor, we can re-write the above function as shown below:

```
function fetchData(url) {
  return fetch(url).then((res) => res.json(res));
}
```

The revised version of the `fetchData` function is concise, easy to read, free from the creation of any unnecessary promises, and allows the code that calls the `fetchData` function to catch and handle any errors. The older version of the fetchData function also allowed the calling code to handle errors, but the revised version does it without using the catch method call.

Unnecessary use of the promise constructor can lead to another problem: if we forget to add the `catch` method call to the promise chain inside the `Promise` constructor, then any error thrown during the HTTP request won't be caught. Forgetting to call the `reject` function inside the executor function can hide the failure of the asynchronous operation inside the executor function.

Incorrect Error Handling

When writing code that uses promises, one of the most important rules to keep in mind is to *either catch and handle the error or return the promise to allow the calling code to catch and handle it*. This fundamental rule can help you avoid hidden bugs in the code that uses promises.

Let's take a look at an example of incorrect handling of errors that breaks the above rule:

```
function fetchData(url) {
  fetch(url).then((response) => response.json());
}

fetchData("https://jsonplaceholder.typicode.com/todos/1")
  .then((data) => console.log(data))
  .catch((error) => console.log(error));
```

Here's a Replit of the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-anti-patterns-example3" />

The above code throws an error because the `fetchData` function doesn't return the promise. It also doesn't allow the calling code to do any kind of error handling.

There are two ways to fix the above code:

- Return the promise from the `fetchData` function by adding the `return` keyword before `fetch(...)`.

  ```
  function fetchData(url) {
    return fetch(url).then((response) => response.json());
  }
  ```

  Here's a Replit of the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-anti-patterns-example4" />

As the above function just makes the HTTP request and returns the response data after calling the `json()` method on the response object, the calling code is responsible for using the response data as well as handling any error.

```
fetchData(/* some url */)
  .then((data) => {
    /* do something with the data */
  })
  .catch((error) => {
    /* handle error */
  });
```

- Handle the error inside the `fetchData` function by chaining the `catch` method to the `then` method.

```
function fetchData(url) {
  fetch(url)
    .then((response) => response.json())
    .then((data) => {
      /* do something with the data */
    })
    .catch((err) => {
      /* error handling code */
    });
}
```

and you call the function above as shown below:

```
fetchData(/* some url */);
```

Converting promise rejection into fulfillment

Each method on the `Promise.prototype` object returns a new promise. If we are not careful, we can write code that can implicitly convert promise rejection into promise fulfillment. Let's take a look at an example:

```
function getData(url) {
  return Promise.reject(new Error()).catch((err) => {
    console.log("inside catch block in getData function");
  });
}

getData()
  .then((data) => console.log("then block"))
  .catch((error) => console.log("catch block"));
```

Here's a Replit of the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/promise-anti-patterns-example8" />

What output do you expect? The output is shown below:

```
"inside catch block in getData function"

"then block"
```

We called Promise.reject inside the `getData` function, so instead of logging "then block", why didn't "catch block" get logged? Instead of the catch block, why was the callback function of the `then` method invoked? Let's understand how the above code executes:

1. `getData` function is invoked.
2. `Promise.reject(new Error())` creates a rejected promise.
3. As a result of the promise rejection, the callback function of the `catch` method is invoked.
4. "inside catch block in getData function" gets logged on the console.
5. As the callback function of the `catch` method didn't explicitly return anything, the callback function implicitly returns `undefined`.
6. The promise returned by the `catch` method is fulfilled with the return value of its callback function, i.e., `undefined`.
7. This fulfilled promise is returned to its calling code by the `getData` function.
8. As the promise returned by the `getData` function is fulfilled with the value `undefined`, the `then` method's callback is invoked in the calling

code, which logs “then block”.

See this [stackoverflow post](), which explains this behavior in more detail.

Although the above code is a contrived example, imagine if there was a `fetch` function call instead of `Promise.reject` in the `getData` function; if the HTTP request is successful, our code will work without any problem, but if the HTTP request fails, the `catch` method in the `getData` function will convert promise rejection into promise fulfillment. As a result, instead of returning a rejected promise, the `getData` function will return a fulfilled promise.

:::info

Sometimes, you do want to convert promise rejection into promise fulfillment to handle the rejection and let the promise chain continue. This is fine if done intentionally. Just be aware that promise rejection can turn into promise fulfillment if you are not careful. Doing this will certainly lead to bugs in your code.

:::

Suppose you are wondering why the promise returned by the `catch` method got fulfilled instead of getting rejected. In that case, the answer is that, as explained in the previous lesson, the promise returned by the `then` or `catch` method gets fulfilled if their callback function explicitly or implicitly returns a value instead of throwing an error or returning a rejected promise or a promise that eventually gets rejected.

So, how can we fix the above code example to avoid this problem? There are two ways to fix this problem:

- Throw the error from the callback function of the `catch` method.

```
function getData(url) {
  return Promise.reject(new Error()).catch((err) => {
    throw err;
```

```
  });
}
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promise-anti-patterns-example9” />

This will reject the promise returned by the `catch` method, and the `getData` function will return this rejected promise. As a result, as expected, `catch` method callback in the calling code will be invoked.

- Remove the `catch` method call.

```
function getData(url) {
  return Promise.reject(new Error());
}
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/promise-anti-patterns-example10” />

This will also invoke the catch block in the calling code because now the `getData` function returns the result of calling `Promise.reject`, and as mentioned before, `Promise.reject` creates a rejected promise. :::tip Personally, I recommend using this approach instead of throwing the error from the `catch` method callback. Just allow the calling code to catch and handle the errors. The `catch` method callback that just re-throws the error is unnecessary. :::

Async executor function

When creating a new promise using the `Promise` constructor, we pass a function to the promise constructor. This function is known as the **executor** function. The executor function should never be an `async` function. Why is that?

Suppose the executor function is an `async` function. In that case, any errors thrown by the `async` executor function will not be caught, and the thrown error won't cause the newly-constructed promise to reject.

```
const p = new Promise(async (resolve, reject) => {
  throw new Error("error");
});

p.catch((e) => console.log(e.message));
```

In the above code example, as the executor function is an `async` function, the error thrown inside it doesn't reject the newly-created promise `p`. As a result, the callback function of the `catch` method, called on promise `p`, never gets called.

If the executor function is a synchronous function, then any error thrown inside the executor function will automatically reject the newly created promise. Try removing the `async` keyword in the above code example and observe the output.

Another thing to note is that if you find yourself using `await` inside the executor function, this should be a signal to you that you don't need the promise constructor at all (remember the first anti-pattern discussed above).

Iterators and Generators

Built-in objects like arrays can be iterated over using the for…of loop. Instead of iterating over an array with a simple `for` loop where we have to access the value using an index, increment the index after each iteration, and also know when to end the iteration so that we don't access indexes that are out of bounds, with `for...of` loop, everything is handled for us. We don't have to worry about the indexes or the loop termination condition.

```
1 const arr = [1, 2, 3];
2
3 for (const num of arr) {
4   console.log(num);
5 }
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/iterators-example1" />

How does `for...of` loop help us iterate over an array? How does it know when to end the iteration?

To understand this, we need to understand the following two concepts:

- Iterables
- Iterators

Iterables

Iterable is an object that implements the iterable protocol. According to the iterable protocol, an object is iterable if it defines the iteration behavior that can be used by the `for...of` loop to iterate over the values in the object. The object can implement a method that is referred to by the property represented by Symbol.iterator; it is one of the well-known symbols to

define the iteration behavior. The well-known symbols were discussed in the module related to symbols.

In the case of arrays, this method is defined in the `Array.prototype` object. This method defines the iteration behavior that is appropriate for arrays. Other objects can also implement this method to define an iteration behavior that is appropriate for them.

What does the `Symbol.iterator` return that can be used by constructs like `for...of` loop to iterate over an object? It returns an iterator object.

Iterators

Iterators are objects that implement the iterator protocol. According to the iterator protocol, an object is an iterator if it implements a method named `next` that takes zero or one argument and returns an object with the following properties:

- `done`: indicates whether the iterator can produce or return another. If it can, its value is `false`, otherwise, `true`. The `true` value is the same as omitting this property in the object returned by the `next` method.
- `value`: the value returned by the iterator object. This property can be omitted or its value can be `undefined` if the value of the `done` property is `true`.

The following are examples of iterator objects with the above mentioned properties:

```
{ value: 45, done: false }

// or

{ value: undefined, done: true }
```

When we iterate over an array using the `for...of` loop, it internally gets the iterator from the array and keeps calling its `next` method until the iterator has returned all values. With the `for...of` loop, we use the iterator

indirectly. We can also use the iterator directly. Arrays are iterables, and we know that iterables implement the `Symbol.iterator` method that returns the iterator object that contains the `next` method. The following code example shows how we can get the array iterator and use it directly to get the values in the array:

```
const arr = [2, 4, 6, 8, 10];

// get the array iterator object
const arrayIterator = arr[Symbol.iterator]();

// get the first iterator result object
let result = arrayIterator.next();

// keep getting new iterator result objects
// until the "done" property of the iterator
// result object is false
while (!result.done) {
  console.log(result.value);
  result = arrayIterator.next();
}

/*
2
4
6
8
10
*/
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/iterators-example2” />

Each built-in iterator object provides iterators that define a specific iteration behavior for the iterable object. The following is an example of using the `Map` object’s iterator directly:

```
const myMap = new Map();
myMap.set("a", 1);
myMap.set("b", 2);
myMap.set("c", 3);

```

```
// get the array iterator object
const mapIterator = myMap[Symbol.iterator]();

// get the first iterator result object
let result = mapIterator.next();

// keep getting new iterator result objects
// until the "done" property of the iterator
// result object is false
while (!result.done) {
  console.log(result.value);
  result = mapIterator.next();
}

/*
["a", 1]
["b", 2]
["c", 3]
*/
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/iterators-example3" />

The value of the `value` property on the iterator result object for the `Map` object is a key-value pair contained in an array. You can use Map.prototype.values() to get an iterator that just returns values in the `Map`, or Map.prototype.keys() to get an iterator that returns all the keys in the `Map`.

Iterator prototype

Each iterator object inherits from the respective iterator prototype object. For example, the array iterator inherits from the Array Iterator prototype object. Similarly, the string iterator inherits from the String Iterator prototype object. All iterator prototype objects inherit from the `Iterator.prototype` object.

```
const arr = [2, 4, 6, 8, 10];

// get the array iterator object
```

```
const arrayIterator = arr[Symbol.iterator]();

// this is the prototype object shared by all array iterators
const arrayIteratorPrototype =
Object.getPrototypeOf(arrayIterator);

console.log(Object.getOwnPropertyNames(arrayIteratorPrototype));
// ["next"]
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/iterators-example4” />

The code example above shows one way to get the array iterator prototype object, and it also shows that the `next` method is inherited from the array iterator prototype object. If we get the prototype of the array iterator prototype object, we will get the `Iterator.prototype` object, which is shared by all iterator prototype objects.

```
const arr = [2, 4, 6, 8, 10];

// get the array iterator object
const arrayIterator = arr[Symbol.iterator]();

// this is the prototype object shared by all array iterators
const arrayIteratorPrototype =
Object.getPrototypeOf(arrayIterator);

// this is the Iterator.prototype object shared by all iterator
prototypes
const iteratorPrototype =
Object.getPrototypeOf(arrayIteratorPrototype);
```

:::info We cannot access the `Iterator.prototype` object directly because it is a hidden global object that all built-in iterators inherit from.

:::

The `Iterator.prototype` object itself is an iterable object, which means that it implements the iterable protocol. But the `Symbol.iterator` method that it implements simply returns the iterator on which this method is called.

This means that any iterator object itself is iterable, meaning that we can use it with constructors like the `for...of` loop.

```
const arr = [1, 2, 3];

const arrayIterator = arr[Symbol.iterator]();

// use the array iterator object instead of the array
for (const num of arrayIterator) {
  console.log(num);
}

/*
1
2
3
*/
```

Here's a Replit where you can run the above code:

<ReplitEmbed src="https://replit.com/@newlineauthors/iterators-example6" />

Making custom iterable objects

At this point, we have learned enough about iterables and iterators to make custom iterable objects. To make a custom iterable object, we need to implement `Symbol.iterator` method that returns the iterator object containing the `next` method. Let's consider an example of student objects that we want to make iterable so that we can easily print their properties with the `for...of` loop.

```
function Student(name, age, id, courses) {
  this.name = name;
  this.age = age;
  this.id = id;
  this.courses = courses;
}
```

This is the `Student` constructor that will be used to make student objects. To make all the student objects iterable, we need to implement the

`Symbol.iterator` method in the `Student.prototype` object, as shown below:

```
Student.prototype[Symbol.iterator] = function () {
  // "this" refers to the student object on which this method is called
  const currentStudent = this;
  const studentProps = Object.getOwnPropertyNames(currentStudent);
  let propIndex = 0;

  const studentIterator = {
    next: () => {
      if (propIndex < studentProps.length) {
        const key = studentProps[propIndex];
        const value = currentStudent[key];
        propIndex++;
        const formattedValue = `${key.padStart(7)} => ${value}`;

        return {
          value: formattedValue,
          done: false
        };
      }

      return {
        value: undefined,
        done: true
      };
    }
  };

  return studentIterator;
};
```

Now, if we try to iterate over any student instance, we will get the formatted values as we defined in the student iterator's `next` method, as shown below:

```
const jack = new Student("Jack", 20, "21A", ["Maths", "Biology", "Physics"]);

for (const val of jack) {
  console.log(val);
}

/*
```

```
   name => Jack
    age => 20
     id => 21A
courses => Maths,Biology,Physics
*/
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/iterators-example9” />

We can define the iteration behavior according to whatever logic we want and format the value returned in the iterator result object in whatever way we want. This gives us the flexibility to define the iteration behavior for any object(s) that is appropriate for a particular object or for a group of related objects.

Remember that each iterator prototype object, for example, the array iterator prototype object, inherits from the `Iterator.prototype` object, but the `studentIterator` object doesn’t. As a result, the student iterator object is not iterable.

```
const jack = new Student("Jack", 20, "21A", ["Maths", "Biology", "Physics"]);

const studentIterator = jack[Symbol.iterator]();

for (const val of studentIterator) {
  console.log(val);
}

// ERROR...
```

Here’s a Replit where you can run the above code:

<ReplitEmbed src=”https://replit.com/@newlineauthors/iterators-example10” />

We can fix this either by explicitly setting up the prototype chain link between the `Iterator.prototype` object and our `studentIterator` object,

or an easier way is to just implement the `Symbol.iterator` method in the `studentIterator` object to make it iterable:

```
Student.prototype[Symbol.iterator] = function () {
  // code omitted to keep code example short

  const studentIterator = {
    next() {
      // code omitted to keep code example short
    },
    [Symbol.iterator]() {
      return this;
    }
  };

  return studentIterator;
};
```

Now the `studentIterator` object is iterable, so we can use it with the `for...of` loop if we want to.

There's one more improvement we can make to the above code example. We have defined `Symbol.iterator` method in the `Student.prototype` object, but it is enumerable, which is not ideal. We can make it non-enumerable by defining it using the Object.defineProperty method, as shown below:

```
Object.defineProperty(Student.prototype, Symbol.iterator, {
  value: function () {
    // copy the code inside the Symbol.iterator method from above
  },
  configurable: true,
  writable: true
});
```

Generators are special functions in JavaScript that can suspend their execution at different points in their execution and then resume from the point at which they were paused. Just like iterators, generator functions can be used to produce a sequence of values that can be consumed by constructs like the `for...of` loop. In fact, a generator function returns a generator

object, which is an iterator. So, we can use the return value of a generator function just like any other iterator.

Following is an example of a generator function that produces odd numbers from 0 to 10:

```
function* odds() {
  for (let i = 1; i <= 10; i += 2) {
    yield i;
  }
}

for (const num of odds()) {
  console.log(num);
}

/*
1
3
5
7
9
*/
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/generators-example1” />

Note the following two things in the generator function above:

- The `function*` syntax marks a function as a generator function. Space between `*` and `function` is also valid syntax: `function *`.
- The `yield` keyword produces a value. The value on the right of the `yield` keyword is what we get when we call the `next` method on the iterator object returned from the generator function. The `yield` keyword also marks a place where a generator function is paused.

Infinite sequence

Generator functions can also be used to create an infinite sequence. as shown below:

```
function* randomNumberGenerator(max) {
  while (true) {
    yield Math.floor(Math.random() * max);
  }
}

const randomNumGen = randomNumberGenerator(10);

// log 10 random numbers
for (let i = 0; i < 10; i++) {
  console.log(randomNumGen.next().value);
}
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/generators-example2” />

The code example above logs only 10 random numbers, but it is possible to use the generator to generate an infinite number of random numbers. This is possible because generator functions are evaluated lazily; their execution is paused until a new value is requested.

In the above code example, we are consuming the generator using the `next` method. Note that calling the generator function doesn’t execute it; instead, it returns a generator object. The generator object is an iterator as well as an iterable. So we can use it like an iterator by calling the next method on it. Calling the `next` method on a generator object returns the values that the generator yields. Multiple calls to the `next` method keep returning the value that the generator yields until the generator object stops yielding the values. At this point, we get an object with the `done` property set to `true`. However, the generator function above can yield values infinitely.

Implementing iterators

As a generator function returns an iterator object, generator functions make it convenient to write iterators. We can rewrite the student iterator example in the previous lesson using a generator function, as shown below:

```
function Student(name, age, id, courses) {
  this.name = name;
  this.age = age;
  this.id = id;
  this.courses = courses;
}

Student.prototype[Symbol.iterator] = function* () {
  // "this" refers to the student object on which this method is called
  const currentStudent = this;
  const studentProps = Object.getOwnPropertyNames(currentStudent);

  for (let i = 0; i < studentProps.length; i++) {
    const key = studentProps[i];
    const value = currentStudent[key];
    const formattedValue = `${key.padStart(7)} => ${value}`;

    yield formattedValue;
  }
};

const jack = new Student("Jack", 20, "21A", ["Maths", "Biology", "Physics"]);

for (const val of jack) {
  console.log(val);
}

/*
   name => Jack
    age => 20
     id => 21A
courses => Maths,Biology,Physics
*/
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/generators-example3” />

Compare the implementation of the `Symbol.iterator` method in the previous lesson and the new one that uses a generator function to implement the student iterator. Code is simpler, concise, and easy to read. Generator functions really make it easy to implement iterators.

As an added bonus, the iterator object returned by the generator function is automatically iterable; as a result, the student iterator is iterable without us needing to manually set up the prototype chain link or implement the `Symbol.iterator` method in the iterator object.

Consuming values

While simple iterators only produce values, generators can also consume values. We can pass a value to the generator using the `next` method. The value passed to the generator becomes the value of the `yield` expression. Consider the following code example of a generator consuming a value:

```
function* myGenerator() {
  const name = yield "What is your name?";
  yield `Hello ${name}!`;
}

const gen = myGenerator();
console.log(gen.next().value); // What is your name?
console.log(gen.next("John").value); // Hello John!
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/generators-example4” />

The first `next` method call yields the first value, i.e.,”What is your name?”. At this point, the generator function pauses. The value of the `yield` expression will be calculated depending on how we call the `next` method again. If we pass an argument to the second call to the `next` method, that argument will become the value of the `yield` expression. In the above code example, the argument provided is the string “John”, so the value of the

first `yield` expression is “John”, and that is saved in the `name` constant inside the generator function.

The second `next` method call provides a value for the first `yield` expression and results in the generator function yielding the next value, which is also a string in the code example above. The second `yield` value is calculated using the value of the `name` constant. As a result, we get “Hello John!” as the second value from the generator function.

:::note We didn’t pass any argument to the first `next` method call; this is because any value provided to the first `next` method call is ignored. The first `next` method call cannot provide a value for the first `yield` expression. The value of the first `yield` expression will be provided by the second `next` method call; similarly, the value of the second `yield` expression can be provided by the third `next` method call, and so on. :::

The following is an example of a generator that produces random numbers infinitely and allows us to pass the upper limit of the range in which the random number should be produced. The number we pass is not included in the range.

```
function* generatorRandomNumber(limit) {
  while (true) {
    const randomNumber = Math.floor(Math.random() * limit);
    limit = yield randomNumber;
  }
}

const randomNumGenerator = generatorRandomNumber(10);

console.log(randomNumGenerator.next());
console.log(randomNumGenerator.next(20));
console.log(randomNumGenerator.next(40));
console.log(randomNumGenerator.next(60));
console.log(randomNumGenerator.next(80));
console.log(randomNumGenerator.next(100));
```

Here’s a Replit of the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/generators-example5" />

Delegating to other iterators

Another cool thing about generators is that they allow us to delegate the responsibility of producing values to other iterators, such as generators. To delegate the responsibility, we need to use the `yield*` operator. This operator takes any iterator and yields values from it until the iterator is done. The following is an example of using the `yield*` operator to delegate the responsibility of producing even or odd numbers to respective generator functions:

```
function* evens() {
  yield 2;
  yield 4;
  yield 6;
}

function* odds() {
  yield 1;
  yield 3;
  yield 5;
}

function* printNums(isEven) {
  if (isEven) {
    yield* evens();
  } else {
    yield* odds();
  }
}

for (const num of printNums(false)) {
  console.log(num);
}

// 1
// 3
// 5
```

Here's a Replit of the above code in action:

<ReplitEmbed src="https://replit.com/@newlineauthors/generators-example6" />

The `printNums` generator delegates the responsibility of producing values to one of the other two generator functions depending on the value of the `isEven` parameter.

Further reading

- Generator (MDN article)
- yield* (MDN article)

We learned about iterables and iterators in the first lesson of this module. An iterable is an object that implements the iterable protocol, and an iterator is an object that implements the iterator protocol. The iterators we learned about were synchronous.

JavaScript also has asynchronous iterators that work similarly to synchronous iterators. The synchronous iterators contain a method named `next` that, when called, returns an object with two properties: `done` and `value`. The asynchronous iterators also contain a method named `next`, but instead of returning an object with the previously mentioned properties, it returns a promise.

Objects implement the `Symbol.iterator` method to implement the iterable protocol. Objects can implement the async iterable protocol by implementing the Symbol.asyncIterator method.

The following is an example of using an async iterator to fetch users from an API:

```
function fetchUsers(userCount) {
  // keep max user count to 10
  if (userCount > 10) {
    userCount = 10;
  }

  const BASE_URL = "https://jsonplaceholder.typicode.com/users";
```

```
  let userId = 1;

  const userAsyncIterator = {
    async next() {
      if (userId > userCount) {
        return { value: undefined, done: true };
      }

      const response = await fetch(`${BASE_URL}/${userId++}`);

      if (response.ok) {
        const userData = await response.json();
        return { value: userData, done: false };
      } else {
        throw new Error("failed to fetch users");
      }
    },
    [Symbol.asyncIterator]() {
      return this;
    }
  };

  return userAsyncIterator;
}

async function getData() {
  const usersAsyncIterator = fetchUsers(3);

  let userIteratorResult = await usersAsyncIterator.next();

  while (!userIteratorResult.done) {
    console.log(userIteratorResult.value);
    userIteratorResult = await usersAsyncIterator.next();
  }
}

getData();
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-iterators-example1” />

In the above code example, we are using the async iterator directly instead of using a construct like the `for...of` loop. Later in this module, we will see how we can consume an async iterator using a loop.

Note in the above code example that the `userAsyncIterator` object inside the `fetchUsers` function implements the `Symbol.asyncIterator` method to implement the async iterable protocol. We discussed in the lesson about iterators that each built-in iterator inherits from the `Iterator.prototype` object and is an iterable itself. To make our custom iterators also iterables, we can make our iterators inherit from the `iterator.prototype` object, or we could implement the `Symbol.iterator` method in our iterator object to make it an iterable. Similarly, async iterators inherit from the `AsyncIterator.prototype` object, which is also a hidden global object, just like the `Iterator.prototype` object. We could make our `userAsyncIterator` object inherit from the `AsyncIterator.prototype` to make it an iterable, or we could implement the `Symbol.asyncIterator` method, as shown in the above code example.

In short, the main difference between a synchronous iterator and an async iterator is that the `next` method of an async iterator returns a promise. Next, we will discuss the asynchronous generators that, just like regular generators, can make it easy for us to implement the async iterators.

Further reading

- AsyncIterator (MDN article)

Similar to the difference between synchronous iterators and async iterators, the main difference between a regular generator and an async generator is that an async generator yields a promise.

Async generators are a combination of `async` functions and generators. As a result, we can use both the `await` and the `yield` keywords inside an async generator. Calling an async generator returns an AsyncGenerator object that implements both async iterator as well as async iterable protocols. The next method of the `AsyncGenerator` object returns a promise.

Let us rewrite the async iterator example in the previous lesson to use an async generator:

```
async function* fetchUsers(userCount) {
  // keep max user count to 10
  if (userCount > 10) {
    userCount = 10;
  }

  const BASE_URL = "https://jsonplaceholder.typicode.com/users";

  for (let userId = 1; userId <= userCount; userId++) {
    const response = await fetch(`${BASE_URL}/${userId}`);

    if (response.ok) {
      const userData = await response.json();
      yield userData;
    } else {
      throw new Error("failed to fetch users");
    }
  }
}

async function getData() {
  const usersAsyncGenerator = fetchUsers(3);

  let userGeneratorResult = await usersAsyncGenerator.next();

  while (!userGeneratorResult.done) {
    console.log(userGeneratorResult.value);
    userGeneratorResult = await usersAsyncGenerator.next();
  }
}

getData();
```

Here’s a Replit of the code above:

<ReplitEmbed src=”https://replit.com/@newlineauthors/async-generators-example1” />

:::info Inside the `fetchUsers` function, instead of awaiting the result of `response.json()` and then yielding the `userData` object, we could simply yield `response.json()`. This will also work because returning a promise from an `async` function resolves the `async` function promise to the promise returned inside its body. The same principle applies here: if we `yield` a

promise, the `async` generator promise will be resolved to the promise that is `yield` inside it.

:::

The async generators make it really easy to implement async iterators, and this is how you would normally implement async iterators.

for await…of loop

So far, we have consumed the async iterators directly by calling the `next` method. The `for...of` loop, which helps us iterate over the iterable objects, has a counterpart known as the [for await…of] loop that helps us iterate over the async iterable. The following code example shows how we can rewrite the `getData` function to use the `for await...of` loop:

```
async function getData() {
  for await (const user of fetchUsers(3)) {
    console.log(user);
  }
}
```

The `for await...of` loop can only be used in a context where we can use the `await` keyword, i.e., inside an `async` function and a module.

Further reading

- [async function* (MDN article)]

Debugging JavaScript

Debugging is the process of finding and fixing problems or errors in code. Most software developers, if not all, spend more time debugging code than writing it. Therefore, debugging is a must-have skill for any software developer. It is impossible to write bug-free code, so learning to debug your own or someone else's code effectively can greatly enhance your productivity as a software developer. We can't escape debugging as software developers, so we might as well learn to do it effectively.

In this module, we will learn different ways to debug JavaScript code. Every JavaScript developer uses the `console.log` and `alert` functions to debug their code, and it is fine to do so, but there are other ways to debug JavaScript. The goal of this module is to introduce the following three ways that can help us debug JavaScript code effectively:

- Using the `debugger` statement
- Using breakpoints in browser developer tools
- Using VS Code debugger

:::note

The debugging strategies mentioned above depend on the debugger to help us debug our code effectively.

:::

The next few lessons introduce each of the above-mentioned methods.

The `debugger` statement allows us to set up a point in our code where the debugger can pause the execution of our code. This is like setting up breakpoints in our code where the code execution can be paused, allowing us to inspect the values of different variables in our code.

Run the following code example in the browser and enter the value “18”. The code below works fine but gives incorrect output for the value “18”.

```
function isOldEnoughToDrive() {
  const age = prompt("What is your age?");
  let result;

  debugger;

  if (age === 18) {
    result = "You are just about the right age to drive!";
  } else if (age < 18) {
    result = "Not allowed to drive";
  } else if (age > 18) {
    result = "Allowed to drive!";
  } else {
    result = "invalid age value provided";
  }

  const resultElm = document.querySelector("#result");
  resultElm.innerHTML = result;
}

isOldEnoughToDrive();
```

```
<body>
  <h2 id="result"></h2>
  <script src="index.js"></script>
</body>
```

Here’s a Replit of the above code in action:

<ReplitEmbed src=”https://replit.com/@newlineauthors/debugger-statement-example1” />

:::info

You can open the above code example in the browser using the VS Code’s live server extension.

:::

The value “18” is a valid value, but we get “invalid age value provided” as an output. Why is that? Let us debug that using the `debugger` statement.

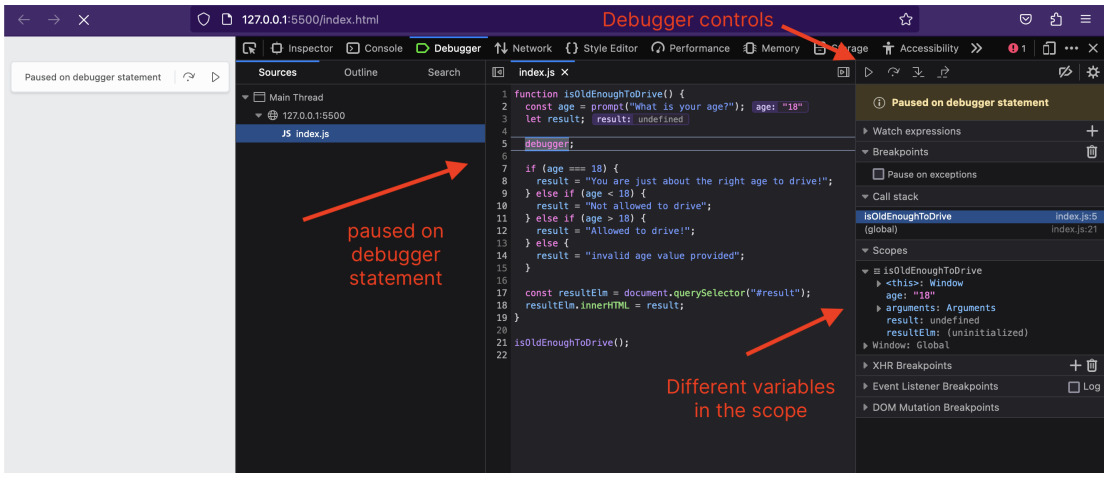
:::note

The Firefox browser was used to show different debugging strategies in this module. You can follow along using Firefox, Microsoft Edge, or the Chrome browser.

:::

You might have noticed the `debugger` statement already added to the code. It is only needed when debugging the code and can be removed after you’re done debugging the code. But it didn’t do anything in the code; our code didn’t pause at the `debugger` statement. Why is that? For the `debugger` statement to pause the code execution, we need to have the browser developer tools opened. Just open the [browser’s developer tools].

Once opened, refresh the browser window, and you will notice the paused code execution, as shown in the image below:



screenshot of code paused on debugger statement

:::info You might need to drag the developer tools window to increase its width in order to match the layout shown in the image above. The narrow

width of the window can show a different layout of the different areas highlighted in the image above. :::

Now that the code execution is paused, we can focus on two areas of the debugger: the debugger controls and the values of different variables in the current scope; both areas are highlighted in the image above.

The debugger controls allow us to execute the code one line at a time, making it easier for us to see how each line of code executes and how it affects the values of different variables.

:::info You can hover over each button in the debugger controls to know what it does. :::

You can also see the call stack above the “Scopes” section in the image above. This allows us to view how the current function was called. We can also hover over different variables in our code to view their values.

:::info You can change the values of different variables in the “Scopes” section by double-clicking on the value. This allows us to see how our code behaves if the values of variables in the current scope are changed. :::

Let us debug why our code doesn’t work when the value is “18”. Note the value of the `age` variable (hover over it or look at the “Scopes” section); you will see that its value is a string and not a number. That means the `prompt` function, which takes the user’s input, returns a string. So when we get to the first `if` condition, i.e., `age === 18`, it doesn’t evaluate to `true`. Can you guess why? Because comparing a string with a number using the triple equals (strict equality) operator always evaluates to `false` and you probably knew that, but if you didn’t, the debugger helped you know that the value of `age` is a string and you are comparing it to a number, so it did help you better understand your code.

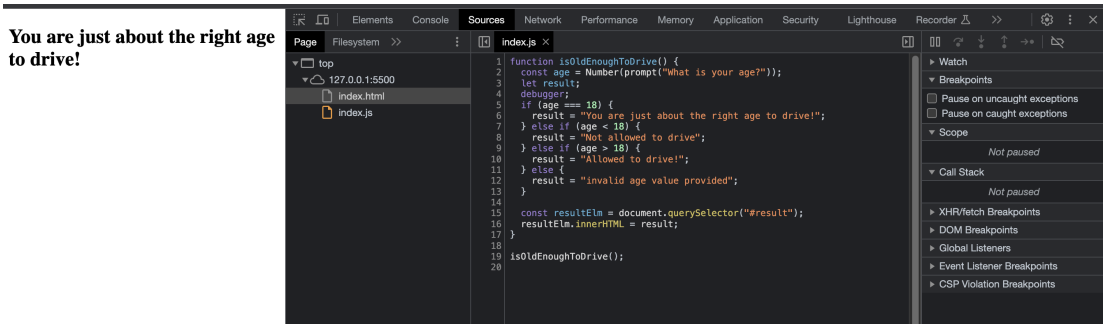
Now that we know the problem, we can fix it by converting `age` to a number before comparing it:

```
1 const age = Number(prompt("What is your age?"));
```

This was a simple example to show you how the `debugger` statement can be used to debug our code. The debuggers built into browsers are really powerful, and it is worth exploring every feature of them to enhance your debugging skills.

In the previous lesson, we used the `debugger` statement to pause the code execution and debug our code. There's another way to pause the code execution, and that is by using breakpoints. A breakpoint acts just like the `debugger` statement, but the difference is that we don't have to write any special keywords in our code. Instead, we open the JavaScript code in the browser's developer tools and set breakpoints in the browser's developer tools.

As shown in the previous lesson, our JavaScript code was opened in the "Debugger" tab. In Chrome, the corresponding tab is named "Sources". The overall functionality of the debugger is more or less the same for both browsers. The following is a screenshot of the "Sources" tab in the Chrome browser containing our JavaScript code:



screenshot of sources tab in chrome dev tools

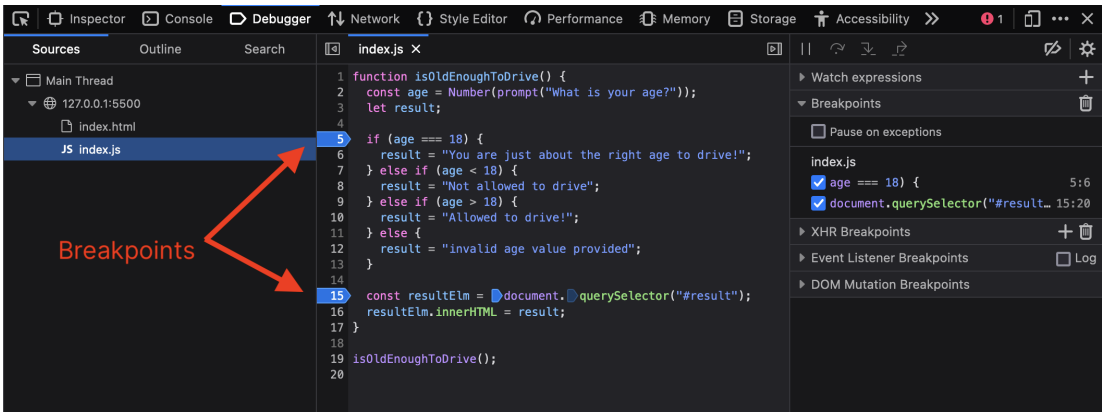
Now, instead of using the `debugger` statement to pause the code execution, let us use the breakpoints. We will use the same example as in the previous lesson but without the `debugger` statement. To set breakpoints, we first need to open our code in the browser, open the developer tools, and open the "Sources" or "Debugger" tab if you are using the Firefox browser or Chrome (other browsers will also have a similar tab).

:::info

If you are following along from the previous lesson, you probably already have the code opened in the browser; if not, you can open the code example in the previous lesson in the browser. You can use VS Code's live server extension to open the code example.

:::

Once the code is opened in the browser's developer tools, setting up a breakpoint is as simple as clicking on the line number in the code. The following image shows two breakpoints set up in the code:



screenshot of breakpoints in browser dev tools

Just click on the line number at which you want to set the breakpoint and refresh the browser window. Just like with the `debugger` statement, when the execution reaches the breakpoint set in our code, the code execution will be paused, and from there on, we can use the different features provided by the browser debugger to debug our code.

Further reading

- Debug JavaScript (chrome devtools docs)

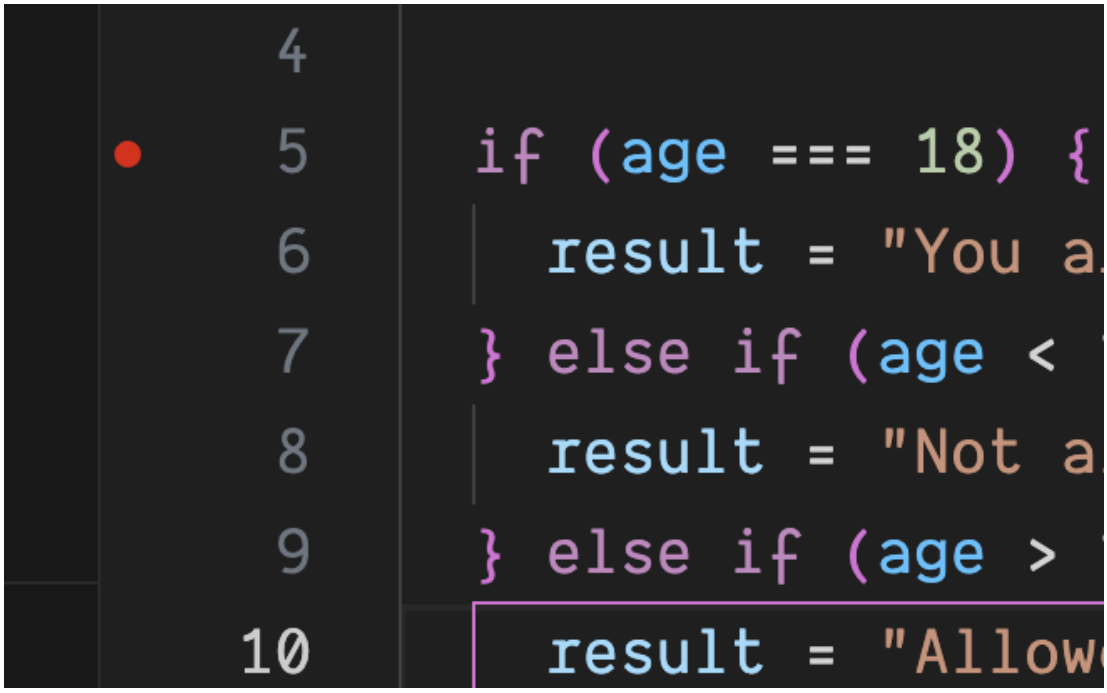
Visual Studio Code (VS Code) is one of the most commonly used editors these days due to the features and flexibility that it provides with the help of

the many extensions that are available to use with it. Among the many features that VS Code provides, one is the built-in debugger that allows us to debug our code within VS Code. Apart from the built-in debugger, there are many extensions available for debugging code written in different languages.

To use VS Code to debug our code, open the same code example in VS Code that we have been working on within the last two lessons. Once opened, create a folder named ".vscode" in the folder containing our code (HTML and JavaScript files). Inside the ".vscode" folder, create a file named "launch.json" and paste the following JSON into this file:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "chrome",
      "request": "launch",
      "name": "Launch Chrome against localhost",
      "file": "${workspaceFolder}/index.html"
    }
  ]
}
```

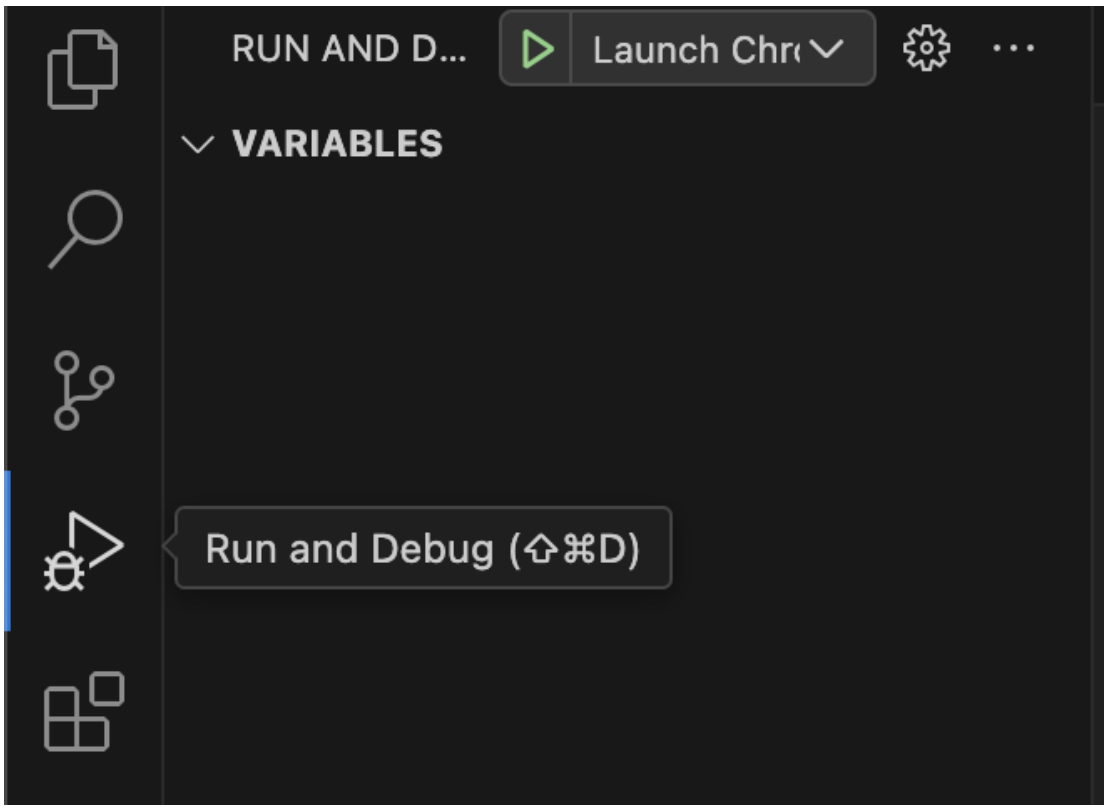
Before we can run the debugger, we need to setup breakpoints. We can either use the `debugger` statement in the code or set breakpoints by clicking on the line number in the JavaScript code file opened in VS Code. The following image shows a breakpoint added in the JavaScript code file that is opened in VS Code:



screenshot of breakpoint added in VS Code

The red dot in the image above is a breakpoint added by clicking on line number 5.

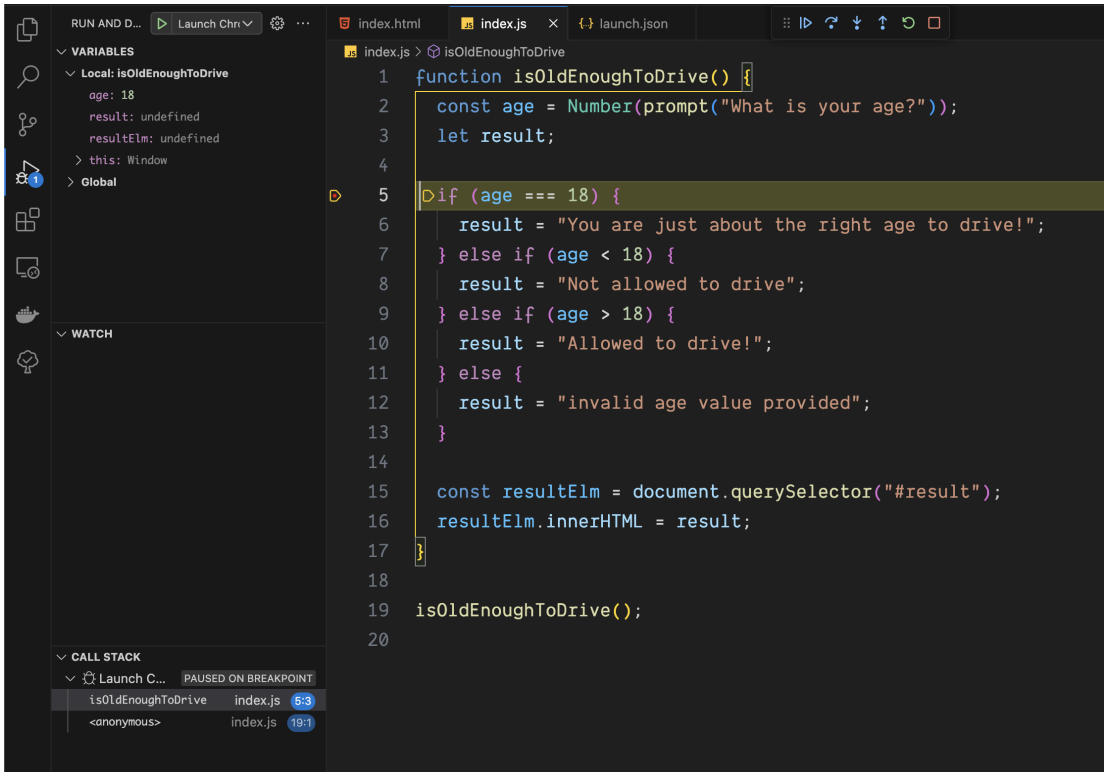
After this, open the “Run and Debug” option in VS Code, as shown in the image below:



screenshot of Run and Debug option in VS Code

Once the “Run and Debug” window opens, as shown in the image above, click on the green play button at the top in the image above. This will open up the Chrome browser, and the debugger will pause the code execution when it reaches the breakpoint. As shown in the image above, the breakpoint is added at line 5, so the debugger will pause the execution after taking user input. Depending on where you added the breakpoint, the code execution will be paused whenever it reaches that point.

The following image shows the state of VS code when code execution is paused at the breakpoint:



screenshot of VS code debugger

You can see the debugger controls at the top, which works similarly to the debugger controls in the browser. The highlighted line shows the point where the code execution paused, and there is a call stack and the variables in the current scope in the left sidebar. We can advance the debugger forward or resume it using the debugger controls and see the flow of execution and values of different variables in the scope to debug our code.

Further reading:

There is a lot more that you can do with the debugger in VS Code. The “launch.json” file that we created manually can be created automatically by VS Code with the click of a button. This and other things possible with the VS Code debugger are explained in the VS Code documentation:

- Debugging (VS Code docs)

Wrap up

Congratulation! You have reached the end of this course. Hopefully, this course was able to meet your expectations, if not exceed them.

Despite all its quirks, JavaScript is an amazing language to learn. The aim of this course is not to cover the JavaScript language in its entirety. Instead, it aims to cover the core topics that are often not understood well enough, especially by beginners. There is so much about JavaScript that couldn't be covered in this course. But with a solid understanding of its core topics, one can surely continue learning it further.

Learning JavaScript opens the door for working not only on the frontend using modern frontend frameworks but also on the backend using technologies like NodeJS.

Next Steps

Following are some of the resources that can be used to continue your journey of learning JavaScript:

- JavaScript reference on MDN is a great resource for referencing different JavaScript topics. Instead of trying to memorize every JavaScript topic, use MDN to read about it when needed.
- ECMAScript specification is mostly meant for those who implement the specification in the browsers. Having said that, it can be used to further dive into a specific topic to understand how it actually works or is supposed to work under the hood.
- exploringjs.com is a great website where you can find books related to JavaScript. The great thing about this website is that the books are free and can be read online.

In addition to the above-mentioned resources, the following are a few of my favorite books (paid resources) that can serve as great resources for learning JavaScript:

- Javascript: The Definitive Guide by David Flanagan
- Javascript: The New Toys by T.J. Crowder
- Professional JavaScript for Web Developers by Nicholas C. Zakas

With that, I shall thank you for investing your time in this course, and hopefully it was able to help you gain a deeper understanding of the JavaScript language.