Pragmatic JavaScript

Richard Krasso

First Edition

Table of Contents

Chapter 1. Closures

Chapter Overview

JavaScript closures are a potent and sometimes misinterpreted language feature. The main reason for this discourse is attributed to the nature of their complexity. Fundamentally, closures are functions that, even after their outer (enclosing) scope has returned, they still retain access to the variables from within it. This makes it possible to use several powerful and flexible programming patterns like private variables in class objects and factory functions.

While closures can be difficult to learn, this chapter aims to reduce the learning curve by approaching closures from a pragmatic standpoint.

Learning Objectives

By the end of this chapter, you should be able to:

- Define what a closure is and how it is used in a JavaScript program.
- Explain the benefits of closures.
- Identify the use cases for closures.
- Develop a JavaScript program with closures.

Scope

JavaScript scope refers to the rules governing how data can be used in a program. In general, there are two types of scope in a JavaScript program - local and global. Local scope refers to the variables that are defined in a function, object, or code block. They are considered local, because they are only accessible from within their defined function, object or code block. Take for example a vending machine in a rest area. A vending machine offers customers a wide range of products, ranging from crackers, candy, chips, soda, and gum. The items in the vending machine are "local" because they are not accessible by other vending machines. Moreover, if I were to insert money into an adjacent vending machine, the products would only dispense from where I inserted the money. Local scope follows a similar approach. A variable defined in a function is not accessible by other functions or variables within other functions.

```
function helloWorld() {
  let message = "Hello World";
  console.log(message);
} // message is not accessible outside of the function

// Prove it
console.log(message); // undefined since message is not defined in the global scope
```

In this above example, a function named helloWorld() is defined with a variable named message. A string value is assigned to the message variable and console.log() is called to output the message. Then on line 7, another console.log() is called to output the message. The code on line 7 will throw an error because the variable message is not accessible from outside the helloWorld function.

```
Uncaught ReferenceError ReferenceError: message is not defined
    at <anonymous> (/Users/rkrasso/repos/buwebdev/web-330-solutions/week-5/examples/scope.js:7:13)
    at Module._compile (internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (internal/modules/cjs/loader:1310:10)
    at Module.load (internal/modules/cjs/loader:1119:32)
    at Module._load (internal/modules/cjs/loader:960:12)
    at executeUserEntryPoint (internal/modules/run_main:86:12)
    at <anonymous> (internal/main/run_main_module:23:47)
```

As you can see from the above error message, JavaScript threw a "ReferenceError" indicating "message" is not defined. The statement "not defined" means that there is a variable named "message" in your code that has not been declared or assigned a value. The second line in the error message identifies where in the code the error was thrown. In this example, the error was thrown on line 13 of the scope.js file (always use the last number for the line number - scope.js:7:13).

Global scope refers to variables, functions, and objects that are accessible throughout the entire program. These variables, functions, and objects are declared outside of any functions or code blocks in your program. In practice, using global variables is dangerous because it is hard to keep track of them. Imagine you are working on a file with 5,000 lines of code and 100 global variables. It is reasonable to suggest that under these conditions, the state (value) of a variable could get lost.

```
let message = "Hello World";

function helloWorld() {
  console.log(message); // message is accessible in the global scope
}

// Prove it
console.log(message); // "Hello World" since message is defined in the global scope
hellWorld(); // "Hello World" since message is accessible in the global scope from the function
```

In this example a function named helloWorld() is defined and in the body of the function a call to console.log() is invoked to output the message variable. Outside of the function another console.log() is invoked to output the message variable. Running the program prints

```
Hello World
```

Hello World

The variable message is considered globally scoped, because it is declared outside of the helloWorld function. Thus, the variable is accessible from anywhere in the program.

Lexical Scope refers to the position in which a text value is placed. Thus, you can tell the scope of a variable by the place in which it was declared in a program. Having a general understanding of how lexical scope works in JavaScript is critical to understanding how closure scope works.

Closure Scope

Functions are very flexible objects in JavaScript because they provide developers with a mechanism for code reuse. In the body of a function, you can declare an unlimited number of variables, objects, and functions. Functions that are created in the body of a function become siblings of that function. The main function is referred to as the parent function and the sibling functions are referred to as child functions. When a function is defined inside of another function, it has access to the variables defined in its own body and the variables defined in the parent function. Effectively, illustrating how closure scope works. Closure scope refers to the set of variables a closure has access to (covered in the next section).

```
function dogSoundMaker() {
 let sound = "Woof!";

 function playDogSound() {
  console.log(sound); // sound is accessible due to closure scope
 }

 playDogSound();
}

dogSoundMaker(); // "Woof!" since sound is accessible in the closure scope from the function
```

In this example, dogSoundMaker is a parent function that defines a variable named sound. playDogSound is a child function to the parent dogSoundMaker. Even though sound is not defined in the child function, it can still be accessed because it is defined in the parent functions scope. Thus, calling console.log from the child function prints the sound. This is an example of closure scope.

Closures

In JavaScript, a closure is a function that has access to its own scope, the parent functions scope, and the programs global scope. In the example provided under Closure Scope, playDogSound is considered a closure. Closures and closure scope are two separate terms. Closure scope refers to the variables a closure can access. Closure scope is not an object that is created when a function is created, rather it is a concept that describes access. On the other hand, closures are actual function objects that have access to the closure scope.

```
function createComposerGreeting(composer) {
  return function(name) {
    console.log(`Hello, ${name}. Did you know that ${composer} was a great composer?`);
  }
}

const mozartGreeting = createComposerGreeting('Mozart');
mozartGreeting('John')

const beethovenGreeting = createComposerGreeting('Beethoven');
beethovenGreeting('Sarah');
```

In this example, createComposerGreeting is a parent function that takes a composer parameter. It returns an anonymous function (which is a closure) that takes a name parameter. Because of closure scope, the child anonymous function has access to the composer parameter and the name parameter. When createComposerGreeting is called with a composer's name, it returns an anonymous function that can be called later with a person's name. Even after the createComposerGreeting has finished executing it can still access composer because of closure scope.

Factory Pattern

The factory design pattern is a creational pattern that allows you to create objects without defining the type of object to create. To facilitate the creation of objects, a unique method, called a factory method, is used to generate the objects. Factory methods can be created in a child function or created in a parent function and then overridden in the child function.

Imagine you are playing a video game where you can create different types of characters, like a warrior, a mage, a healer, or rogue. Each character has different abilities, spells, and characteristics. Instead of creating each character from scratch, the game provides a "character creation" screen. You just need to select the type of character you want and the game creates it for you with all the default abilities and characteristics of that type. The "character creation" screen is like a factory in programming. It is the central place where

objects (in this case, characters) are created. You just tell it what type of object you want (warrior, mage, healer, rogue) and it handles the details of creating that object for you.

In the code under the "Closures" section, createComposerGreeting is like the "character creation" screen. It is a factory for creating functions. You tell the function what type of greeting you want (Mozart, Beethoven), and it gives you a function that can greet a person. Each function that is created is a closure that remembers which composer it is asked to use in the greeting. Similar to the "character creation" screen, each function that is created remembers which composer it is asked to use in the greeting. In other words, the closure remembers the default greeting settings (warrior, mage, healer, rogue).

```
function createCharacter(type) {
  return function(name) {
    console.log(`Hello, ${name}. You are a ${type} in this game.`);
  }
}

const warriorCreation = createCharacter('Warrior');
warriorCreation('John'); // Hello, John.  You are a Warrior in this game.

const mageCreation = createCharacter('Mage');
mageCreation('Sarah'); // Hello, Sarah.  You are a Mage in this game.
```

In this example, createCharacter is like the "character creation" screen. It is a factory for creating character functions. You tell it what type of character (Warrior, Mage) you want and it gives you a function that creates that character type. Each function it creates is a closure that remembers the character type it is supposed to use.

The advantages of closures and the factory pattern in JavaScript are:

1. **Memory Efficiency**: Closures can help save memory. In the character creation example, instead of creating separate functions for each character type in the game, we use a single function to generate the character type we want. Effectively, this improves memory efficiency because we do not have duplicated functions taking up memory space.

2. **Encapsulation**: Closures help keep variables that should not be accessed by other parts of your code hidden (private). This is similar to how in a video game you cannot directly change a character's level or stats without some type of cheat code (up, up, down, down, left, right, left, right, b, a, b, a, select, start – kudos to you if you know what this cheat code is for). You have to earn experience points or find items through normal game progression.

3. **Dynamic Function Generation**: Closures allow us to create functions dynamically. In the character creation example, we can create as many character types as we want without having to write new code for each new character type we introduce to the game. For example, let's say we want to add the Warlock type to our game. There is no need for us to modify the factory code to support this new character type.

4. **Maintaining State**: When functions in JavaScript are executed, they use new scope. Closures allow us to keep track of the data. It is just like how the character in a game remembers its level and items each time a new session is started. Imagine if you had to start from level 1 with 0 items each time you started a new game session.

In summary, closures promote efficient memory use, data privacy, dynamic function generation, and state management. The power and flexibility of closures is why it is heavily used in JavaScript frameworks (Node.js I am looking at you) and libraries.

Closure Data Privacy

Function objects are similar in characteristics to the concept of a class in object-oriented programming languages like C++, Java, C# and Python 3+. Later versions of JavaScript introduced the idea of a class with the keyword Class, but fundamentally, it is still considered a function object. The general syntax for defining a function object is:

```
function FunctionIdentifier {
  functionMemberList
}
```

In which functionMemberList consists of variable declarations and/or functions. That is a member of a function can either be a variable (stores data) or a function (manipulates data). The following statements define a course with private variables and functions.

```
// function Course
// private members are declared and set in the body of the function
// public members are returned in the anonymous object literal as a closure
function Course() {
  // private variables are declared here
  let title = "Enterprise JavaScript II";
  let author = "Professor Krasso";
  let description = "A course on advanced JavaScript principles and practices for
enterprise application development.";
  let price = 29.99;

  return {
    getTitle: function() {
```

```
    return title;
  },
  getAuthor: function() {
   return author;
  },
  getDescription: function() {
   return description;
  },
  getPrice: function() {
   return price;
  }
 }
}
```

The members of a function object are classified as private, protected, or public. Technically, in JavaScript there is no concept of private and protected members, but they can be emulated (as shown in this section with data privacy). This chapter focuses exclusively on public and private members. If a member of a function object is a function, it can (directly) access any member of that function object (member variables and member functions). There are two types of parameters associated with a function: formal and actual. Formal parameters are variables declared in the function header.

function functionName(formal parameter list) { statements }

Actual parameters are variables listed in a call to a function. For example,

```
const x = functionName(actual parameter list)
const me = myName("Professor Krasso");
```

In the second line of code, myName function is called with an actual parameter of "Professor Krasso."

Member functions can have formal parameters and function objects can have formal parameters. If the function object has formal parameters, you must include actual parameters in your call to that function object. Likewise, if member functions have formal parameters, you must include actual parameters in your call to that member function. Once a function object is defined, you can declare variables of that object type. In JavaScript, a function object variable is called a function object or function instance. Although, some developers refer to function objects in a more general way by simply calling them objects. I prefer the term function object over plain object. In this book we will use the term function object and function instance for variables of a function object. Thus, we can say the following:

```
let title = "Enterprise JavaScript II"; // the variable title is a member variable
```

```
// the function getTitle is a member function.
getTitle: function () {
  return title;
}

const course = Course(); // the variable course is a function object or function instance.
console.log(course.getTitle()); // course.getTitle is a call to the function objects member function.
```

In the previous section, it was stated that closures help with data privacy through encapsulation. Encapsulation is a way for us to hide the details of an object from the other parts of code in our application. This is referred to as information hiding. To understand why encapsulation is important and why it is something you should learn, consider the following:

Imagine you have a diary with a lock on it. You can write your secrets in the diary and unless someone has a key, they cannot read them. The "secrets" are encapsulated in the diary. Encapsulation is a way for you to control access to that information (lock and key).

```
function Character(type) {
  let health = 100; // private variable

  return {
    getType: function() {
      return type;
    },
    getHealth: function() {
      return health;
    },
    takeDamage: function(amount) {
      health -= amount;
    }
  }
}

const warrior = Character('Warrior');
console.log(warrior.getType()); // "Warrior"
console.log(warrior.getHealth()); // 100
warrior.takeDamage(20);
console.log(warrior.getHealth()); // 80

// prove that health is private
```

warrior.health = 0;
console.log(warrior.getHealth()); // 80 since health is private

In this example, the Character function is an example of encapsulation. And, heath is an example of a "private" member variable. That is heath cannot be accessed outside of the Character function. Instead, you have to use member functions getHealth and takeDamage to adjust the characters health. This is like having a key (member functions) to access the secrets (member variables) in the diary (Character function). This is proven in the last two lines of code. heath is set to 0 and console.log is called to print the updated health value. But, because of encapsulation, the heath value still prints 80. Thus, unless the user has some type of cheat code, they cannot directly change the characters health. The member functions getType, getHealth, and takeDamage within the Character function are closures.

Programming Exercises

In this assignment you will create a character creation screen for a MMORPG (Massively Multiplayer Online Role-Playing Game) using JavaScript closures. HTML, and CSS.

Starter Code: GitHub Repo

1. Create an HTML page with a form that allows the user to enter a character name, select a character gender (male, female, other), and select a character class (warrior, mage, rogue). Use CSS to style your form and make it visually appealing.

2. Write a JavaScript function that creates a new character. The function should take the character name, gender, and class as formal parameters and return an object with the following member functions:

    a. getName: Returns the character's name
    b. getGender: Returns the character's gender.
    c. getClass: Returns the character's class.

3. When a user clicks the "Create Character" form, use your function to create a new character with the entered name, selected gender, and class.

4. Display the character's name, gender, and class on the page. Use CSS to style the information in a way that fits the theme of your game.

Hint:

- Remember to use closures to keep the character's name, gender, and class only accessible through the getName, getGender, and getClass member functions.

Chapter 2. Callback Functions

Chapter Overview

Callback functions in JavaScript are an essential component of contemporary web development. They are widely used to handle asynchronous operations and to trigger code execution when tasks are completed in a JavaScript program. JavaScript's event-driven architecture, which allows for the creation of dynamic and responsive web applications, depends heavily on callback functions. We will examine callback functions in this chapter, covering their definition, usage, and role in JavaScript development.

Learning Objectives

By the end of this chapter, you should be able to:

- Define higher-order functions
- Explain callback functions.
- Distinguish the differences between synchronous and asynchronous flow
- Develop a JavaScript program using callbacks.

Higher-Order Functions

In JavaScript, higher-order functions are functions that either accept functions as formal parameters or return functions when called. To understand the concept of higher-order functions, imagine you are assembling a team of unique individuals, each with their own special skills and abilities. For instance, one individual might be an expert in aerial maneuvers, possess extraordinary strength, and have the ability to see through objects. Another might have enhanced physical capabilities, be a master of martial arts, and an exceptional investigator. Now, you want to create a function that can these skills for a coordinated team action.

In this analogy, a higher-order function is like the coordinated team action. It's a function that takes other functions (the individual skills of the team members) as formal parameters and returns a single function (team action) with their combined skills. Consider the following JavaScript program:

```
function teamAction(memberOne, memberTwo) {
  return function (target) {
    memberOne.action(target);
    memberTwo.action(target);
  };
}

const memberOne = {
  action: function (target) {
    console.log(
```

```
    `Member One uses their aerial maneuvers and extraordinary strength to engage
${target}.`
   );
 },
};

const memberTwo = {
 action: function (target) {
  console.log(
   `Member Two uses their investigative skills to outsmart ${target}.`
  );
 },
};

const coordinatedAction = teamAction(memberOne, memberTwo);
coordinatedAction("Opponent"); // Member One and Member Two both engage
Opponent
```

In this example, teamAttack is a higher-order function. It takes two formal parameters (memberOne.action and memberTwo.action) and returns an anonymous function that combines the actions. The result is a single coordinated action against the Opponent. This is demonstrated in the final line of code, which prints the following message:

Member One uses their aerial maneuvers and extraordinary strength to engage Opponent.
Member Two uses their investigative skills to outsmart Opponent.

JavaScript provides several built-in higher-order functions that operate on arrays. For example,

1. **Array.prototype.map():** This function takes a function as an actual parameter and applies it to each element in the array. The result is a new array with the transformed data.

2. **Array.prototype.filter():** This function takes a function as an actual parameter and returns a new array with the elements that pass the test implemented in the actual parameter. For example, imagine you had an array of individuals (from various teams) and you wanted to a separate array of only your team members, you could use this built-in higher-order function to filter out members from other teams. This assumes, the actual parameter was a function that identified your specific team members.
3. **Array.prototype.reduce():** To understand JavaScript's reduce function, consider the following: Suppose you were in a grocery store with a basket full of apples and, before proceeding to the checkout lane, you needed to know the total weight of all

the apples. Beginning with a total weight of zero, you would weigh each apple and add its weight to the total weight. Array.prototype.reduce does something similar. It takes an array and reduces it to a single data value. It does this by starting at an initial value and slowly reducing the arrays size while applying the function you pass as an actual parameter.

4. **Array.prototype.forEach():** This function iterates over an array executing the actual parameter function on each element in the array.

```
const apples = ["Granny Smith", "Fuji", "Gala", "Honeycrisp"];

apples.forEach(function(apple, index) {
  console.log(`Apple ${index + 1}: ${apple}`);
});

// output
Apple 1: Granny Smith
Apple 2: Fuji
Apple 3: Gala
Apple 4: Honeycrisp
```

Higher-order functions are a key component to the flexibility of a functional programming language. In contrast to other programming languages (C++, Java, C#), functional programming views programs as functions. JavaScript is not a functional programming language in the strict sense of languages like Scheme, Haskell, and Erlang, but it does support functional programming and it is certainly worth learning and being proficient in.

Synchronous Programming

Image you are in a lunch line at school. You cannot get your lunch until the person in front of you has gotten theirs. And, the person behind you cannot get their lunch until you have gotten yours. Everyone has to wait their turn. This is like synchronous programming. Tasks are executed synchronously. That is, one task cannot complete until the task in front of it has completed. If an action on a task takes a long time to complete, all other tasks in the program have to wait until that task completes.

```
const students = ["John", "Sarah", "Alex", "Emma"];

students.forEach(function(student, index) {
  console.log(`${student} gets their lunch. Student number ${index + 1} served.`);
});
```

In this example, each student represents a person in a lunch line. The forEach function is used to synchronously iterate over each student in the line. As each student is served a

message is printed indicating the student was served and the number they were in the line. This is a synchronous operation, because each task (getting lunch) is completed before the next one starts.

Using the previous unique individual's analogy, imagine is facing a series of threats that they must deal with in a specific order. First, Team Member, who is an expert in aerial maneuvers and possesses extraordinary strength must stop a meteor from destroying Earth. After Team Member A has eliminated the meteor threat, Team Member B, who has enhanced physical capabilities and is a master of martial arts, must disarm a bomb in a city. Only after Team Member B has disarmed the bomb, Team Member C, who is an expert in aquatic rescue, can save a sinking ship. Each task depends on the completion of the previous task. They cannot be done all at once or in a different order. Team Member A must go first, then Team Member B, and then Team Member C.

```
function teamMemberATask() {
  console.log("Team Member A stops the meteor.");
}

function teamMemberBTask() {
  console.log("Team Member B disarms the bomb.");
}

function teamMemberCTask() {
  console.log("Team Member C saves the sinking ship.");
}

// Call the functions in order
teamMemberATask(); // Team Member A stops the meteor.
teamMemberBTask(); // Team Member B disarms the bomb.
teamMemberCTask(); // Team Member C saves the sinking ship.
```

Among the topics covered in this chapter, synchronous programming is the easiest to understand and implement. It is natural for humans to organize tasks and operations synchronously; however, the true power of JavaScript is in its ability to execute asynchronous operations.

In the next section we will take a look at asynchronous programming.

Asynchronous Programming

In asynchronous programming, tasks are non-blocking, meaning they are executed and do not stop or block the execution of other tasks in a program. In other words, they are executed independently of other tasks. Unlike synchronous programming, where tasks are dependent on the completion of other tasks. To understand asynchronous programming

consider the following, imagine you are at a restaurant and you want a table. You go to the host as give them your name. They tell you it will be a 30-minute wait and take your phone number to text you when the table is ready. You do not have to stand there and wait, instead, you can go the bar to have a drink, browse a nearby shop, chat with friends, or browse videos on your phone. When you get the text, you return to the host and get seated.

This is like asynchronous programming. In asynchronous programming you can start a task (adding your name to the waiting list), and while waiting for a table, you can do other tasks (chat with friends, have a drink at the bar). When the original task is done, you get notified (text message) and you can handle the result (follow the host to your table).

```
function makeReservation(name, callback) {
  setTimeout(() => {
    console.log(`Table is ready for ${name}.`);
    callback();
  }, 5000);
}

console.log("Reservation made under the name Richard. Waiting for the text...");

makeReservation('Richard', function() {
  console.log("Received text. Going to the restaurant to get seated.");
});

console.log("Browsing a nearby shop while waiting for the table...");
```

In this example, makeReservation represents making a reservation at a restaurant. It uses JavaScript's built-in setTimeout function to simulate waiting for a table. The callback function (discussed in the next section) represents the action of receiving a text when the table is ready. For now, do not concern yourself with the callback function and instead focus on the simulated process.

When you run this program, you will see that the reservation is made under the name Richard followed by a message indicating Richard is browsing nearby shops. After 5000 milliseconds (five sections), the callback function is called, simulating a text message, which outputs a message indicating the table is ready and a message acknowledging the response from Richard's phone. This is an example of a non-blocking, asynchronous operation, because other tasks are able to start and end while the makeReservation function is being processed.

In the previous section, we used an example to show how tasks are controlled in synchronous programming. Now, let's use a cooking analogy to demonstrate how asynchronous programming works.

```
function chefATask() {
  console.log("Chef A starts their task.");
  setTimeout(() => {
    console.log("Chef A finishes baking the cake.");
  }, 5000);
}

function chefBTask() {
  console.log("Chef B starts their task.");
  setTimeout(() => {
    console.log("Chef B finishes making the sauce.");
  }, 3000);
}

function chefCTask() {
  console.log("Chef C starts their task.");
  setTimeout(() => {
    console.log("Chef C finishes preparing the salad.");
  }, 1000);
}

// Call the functions in order
chefATask(); // Chef A finishes baking the cake.
chefBTask(); // Chef B finishes making the sauce.
chefCTask(); // Chef C finishes preparing the salad.
```

In this example, there are three tasks that all start at the same time. The setTimeout function is used to simulate the time it would take for each chef to complete their tasks (bake the cake, make the sauce, prepare the salad). It is important to note that the tasks (chefATask, chefBTask, and chefCTask) are now asynchronous. If they were called one after the other, they would start their tasks at the same time, but finish at different intervals. This is a key aspect of asynchronous programming – tasks can start and stop at different times. However, what do you do when you have asynchronous tasks (like the chefs' tasks) but need to control the exaction order. Meaning, you need Chef A's task, which takes 5 seconds to complete, to finish before starting Chef B's 3 second task. And you need Chef B's 3 second task to finish before starting Chef C's 1 second task. There are three options for controlling the execution order of asynchronous tasks: Callback Functions (covered in the next section), Promises (covered in chapter 3), and Async/Await (covered in chapter 4).

In the next section, we will take a look at the first way we can control the execution order of asynchronous tasks: Callback Functions.

Callback Functions

In programming, a callback function is a formal parameter in another function. The code in the formal parameter is not executed until the code in the function is completed. To understand callback functions, consider the following: imagine you ask someone out on date, and they say they have to work this weekend, but they will let you know when they are free. You give them your phone number and ask them to call you once they have had a chance to look over their work schedule. This is an example of how a callback function works. It is a way for code to be executed once a certain task has completed.

Let's consider the following simple callback example,

```
function makeReservation(name, callback) {
  console.log(`Making reservation for ${name}...`);
  setTimeout(() => {
     console.log(`Reservation for ${name} is ready.`);
     callback();
  }, 5000);
}

function goRestaurant() {
  console.log("Going to the restaurant to get seated.");
}

console.log("Reservation made under the name John. Waiting for the text...");
makeReservation("Richard", goRestaurant);
```

In this code example, makeReservation is a function with two formal parameters: a name for the restaurant reservation and a callback function. It uses the setTimeout to simulate the time between making a reservation and being seated at a table. Once the reservation is ready (5000 milliseconds or 5 seconds), the callback function is called. When the makeReservation function is called, two actual parameters are supplied: a string value of Richard and the function goRestaurant. This function is used as the second formal parameter in the makeReservation function, which means it is the callback function that is called from the body of the makeReservation function.

As mentioned earlier, callback functions are also a great way for controlling the execution order of asynchronous operations. If you recall from the previous section, we converted the synchronous task sequence to an asynchronous one using a cooking analogy. However, this introduced a new challenge: the tasks are now being performed out of order. With the introduction of callback functions, we can address this issue. Consider the following:

```
function chefATask(callback) {
```

```
  console.log("Chef A starts their task.");
  setTimeout(() => {
    console.log("Chef A finishes baking the cake.");
    callback(); // Chef A signals Chef B
  }, 5000);
}

function chefBTask(callback) {
  console.log("Chef B starts their task.");
  setTimeout(() => {
    console.log("Chef B finishes making the sauce.");
    callback(); // Chef B signals Chef C
  }, 3000);
}

function chefCTask() {
  console.log("Chef C starts their task.");
  setTimeout(() => {
    console.log("Chef C finishes preparing the salad.");
  }, 1000);
}

// Call the functions in order
chefATask(function () {
  chefBTask(function () {
    chefCTask();
  });
});
```

In this code example, we updated the chefATask and chefBTask by introducing a formal parameter for a callback function. This callback function is called after the setTimeout function inside of each task has completed. Effectively, signaling the next chef to begin their task. The ability to control the execution order of code in an asynchronous program is very powerful and used heavily in modern JavaScript frameworks (Node.js, React.js, Angular). In subsequent chapters we will explore two other options for controlling flow: promises and async/await.

Programming Exercises

In this assignment you will create a restaurant reservation webpage using JavaScript callbacks, HTML, and CSS.

1. Create an HTML page with a form that allows the user to enter their name and select a table number to reserve. Use CSS to style your form and make it visually appealing.

2. In your JavaScript code, create an in-memory object array for each table in the restaurant. Each object should contain properties for table number, capacity, and isReserved.

3. Create a function reserveTable with three formal parameters: table number, a callback function, and a time in milliseconds. The function should check if the table is available. If it is, update the isReserved property of the table and then use setTimeout to wait for the time specified in the formal parameter, then call the callback function with a success message. If the table is not available, it should immediately call the callback function with an error message.

4. When the form is submitted, call the reserveTable function with a callback that updates the webpage with the success or error message.

Chapter 3. JavaScript Promises

Chapter Overview

Promises in JavaScript are objects that indicate whether an asynchronous operation has passed or failed and what will happen as a result of a failure. They offer a simpler approach for working with asynchronous operations by handling asynchronous tasks without the need for callback functions. In this chapter, we begin by going over the fundamentals of JavaScript promises, including their definition and usage. Next, we will dive further into the topic of promises by learning how to handle errors, chain promises together, and how to use concurrency to aggregate the results of multiple promises.

Chapter Learning Objectives

By the end of this chapter, you should be able to:

- List the differences between callback functions and promises.
- Explain how promises are used in a JavaScript program.
- Identify the states of a promise.
- Build a JavaScript program using promises.

Introduction to Promises

In JavaScript promises are objects that indicate the completion or failure of an asynchronous operation. Promises are comprised of the following states:

1. **Pending**: The outcome of the promise is yet to be determined. That is, the asynchronous operation has yet to be completed and thus, the promise is in a pending state.

2. **Fulfilled:** The asynchronous operation has completed and the promise has returned an appropriate value.

3. **Rejected:** The asynchronous operation failed, which in turn means the promise cannot be fulfilled. All promises that are rejected will provide some indication of why the failure occurred (see the Error Handling section).

Promises are created using the Promise constructor. The basic syntax for creating a new promise is:

```
const prom = new Promise((resolve, reject) => {});
```

The Promise constructor takes two actual parameters, resolve and reject. resolve represents successfully completed asynchronous operations. And, as the name suggests, reject is for failed asynchronous operations. All promises start in a "pending" state and at some point, either become fulfilled or rejected.

To fully understand promises, consider the following: imagine you want to buy a new video game, but it is not released yet. So, you pre-order it. The pre-order is an example of a promise. The company is basically saying, "We promise to save/sell you a copy of the game, once it is released." In JavaScript, promises are used to handle tasks that take time to complete. Examples range from reading data from a text file, making queries to a database, fetching data from an API, or processing large mathematical calculations. In other words, when you create a promise, you are basically saying, "I promise this take will complete, but it will take some time."

Using the pre-order analogy,

```
let preOrderGame = new Promise((resolve, reject) => {
  let gameIsReleased = true;
  if(gameIsReleased) {
    resolve('Game is released, start playing!');
  } else {
    reject('Game is not released, get a refund.');
  }
});

preOrderGame.then((message) => {
  console.log(message); // This will run if the Promise is fulfilled
}).catch((message) => {
```

```
  console.log(message); // This will run if the Promise is rejected
});
```

In this example, preOrderGame is a promise. If gameIsReleased is true, the promise is fulfilled and the customer can start playing the game. If gameIsReleased is false, the promise will be rejected and the customer should ask for a refund. The gameIsReleased variable is used to simulate a "passing" criteria. In an actual program, the code would be replaced with either some condition or function that would respond to the game being released.

Consider the cooking example from the previous chapter,

```
let chefATask = new Promise((resolve, reject) => {
  // Chef A starts his task
  let taskCompleted = true; // Code to bake the cake
  if (taskCompleted) {
    resolve("Chef A finished baking the cake.");
  } else {
    reject("Chef A failed to bake the cake.");
  }
});

chefATask
  .then((message) => {
    console.log(message); // This will run if the Promise is fulfilled
  })
  .catch((message) => {
    console.error(message); // This will run if the Promise is rejected
  });
```

In this example, chefATask is a promise. If taskCompleted is true, the promise is fulfilled and Chef A has finished baking the cake. If taskCompleted is false, the promise is rejected and Chef A failed to bake the cake. The taskCompleted variable is used to simulate a "passing" criteria. In an actual program, this code would be replaced with a function to back the cake. To this point, we have discussed how to create a promise and its states. In the next section, we will explore how to handle rejections (error handling).

Error Handling

Promises are comprised of the following states: pending, fulfilled, and rejected. States for pending and fulfilled were explored in the previous section. Typically, a promise is rejected when an asynchronous operation has failed. Under these circumstances, the "rejected" promise should provide some type of information that indicates why the promise failed and a path that the program can take as a result of the rejection. To handle rejections in a

promise, you call the reject function during the creation of the promise. Then, once the promise is invoked, the catch function is used to "catch" the error that was thrown. In the body of the catch function the rejection is handled. Looking at the code from the previous section, you can see

reject('Chef A failed to bake the cake.');

The reject function is called with an actual parameter of "Chef A failed to bake the cake." In the catch function:

```
.catch((message) => {
  console.log(message); // This will run if the Promise is rejected
});
```

"message" is the caught actual parameter from the reject function. In our case, it is the string value "Chef A failed to bake the cake." Next, a call to the console.error() function is used to print the message to the terminal/browser window.

Promise Chains

An arrangement of linked promises is called a "Promise Chain." Every promise in the chain depends on how the one before it is resolved. This enables you to carry out several asynchronous tasks in a particular order. The following is an explanation of how promise chains work:

1. Start with a Promise either created by yourself or one returned from a JavaScript library or built-in function.

2. Use the then function on the first promise, which takes an actual parameter that is called when the promise is fulfilled (resolved). The actual parameter receives the resolved value as its actual parameter.

3. The then function returns a promise. This promise resolves to the value returned by the function you passed to then. Thus, allowing you to chain another then.

4. You can chain as many then calls as you want. Each call to then will wait for the previous promise to resolve before being invoked.

5. If any promise in the chain is rejected, the chain is broken and control is passed to the nearest catch handler (see the Error Handling section). All subsequent promises in the chain are ignored.

To understand promise chains, consider the following: imagine you pre-ordered a video game and you also have planned a series of actions to complete once you receive the

game. First, you want to play the game. Then, you will invite a friend over to play the game together. After that, you will post a review online for others to see. Each of these actions depends on the previous one being completed. That is, before you can play the game with a friend you need to try the game yourself. And before posting your review of the game online, you want to play the game with a friend. The code for this type of scenario is

```
let preOrderGame = new Promise((resolve, reject) => {
 let gameIsReleased = true;
 if(gameIsReleased) {
   resolve("Game is released, start playing!");
 } else {
   reject("Game is not released, get a refund.");
 }
});

preOrderGame.then((message) => {
 console.log(message); // This will run if the Promise is fulfilled
 return "Invite friend over to play together.";
}).then((message) => {
 console.log(message);
 return "Post a review online.";
}).then((message) => {
 console.log(message);
}).catch((message) => {
 console.log(message); // This will run if the Promise is rejected
});
```

In this example, if the preOrderGame is fulfilled a message is printed and the string value “Invite friend over to play together” is returned. The first then statement waits for the previous promise to resolve and then prints the message and returns another string value “Post a review online.” The second then statement waits for the previous promise to resolve and then prints the message. If any of the promises in the chain fail, the catch function “catches” the error and prints a message. It is important to note, that in this particular case if any of the promises in the chain fail, the entire chain is broken. It does not matter if the first, second, or third promise fails, the entire chain is broken and rejected.

Recall the cooking analogy we used in the previous sections. We had to use callback functions to control the execution order for the chefATask, chefBTask, and chefCTask. We can use promise chains to achieve the same results.

```
function chefATask() {
 return new Promise((resolve) => {
  console.log("Chef A starts their task.");
  setTimeout(() => {
```

```
    console.log("Chef A finishes baking the cake.");
    resolve();
   }, 5000);
  });
}

function chefBTask() {
  return new Promise((resolve) => {
   console.log("Chef B starts their task.");
   setTimeout(() => {
    console.log("Chef B finishes making the sauce.");
    resolve();
   }, 3000);
  });
}

function chefCTask() {
  return new Promise((resolve) => {
   console.log("Chef C starts their task.");
   setTimeout(() => {
    console.log("Chef C finishes preparing the salad.");
    resolve();
   }, 1000);
  });
}

// Call the functions in order
chefATask()
  .then(() => {
   return chefBTask();
  })
  .then(() => {
   return chefCTask();
  })
  .then(() => {
   console.log("All tasks completed successfully!");
  })
  .catch((error) => {
   console.error("An error occurred:", error);
  });
```

In this code, each chef's task is a function that returns a promise. Each promise resolves after the setTimeout duration finishes. The promises are then chained together using the then function. When the chefATask promise is fulfilled, the chefBTask is returned and when

the chefBTask promise is fulfilled, the chefCTask promise is fulfilled. If any of the tasks are rejected, the entire chain is broken and the catch function “catches” the error.

Chaining promises is an excellent strategy for controlling the execution order of asynchronous operations. And, it is arguably easier to read than the approach we used with callback functions. However, how do we handle scenarios where we want to wait for all of the promises in the chain to be executed before handling errors (promise rejections)? In other words, what if we still wanted Chef C to prepare the salad regardless if Chef B fails to make the sauce? A solution to this problem will be explored in the next section.

Concurrency

Within programming, concurrency refers to the capacity for several components or units of an algorithm, program, or problem to be executed partially or out of sequence without compromising the overall result. In distributed or multicore systems, this is a method to increase a program’s speed and performance. To understand concurrency, consider the following: imagine you are at a cafeteria with multiple lunch lines. Each lunch line has its own server who prepares the plate. That is all servers work independently to serve their customers. While one server is putting mashed potatoes on a customer’s plate another server from a different line is putting a hamburger on their customer’s plate. And, another server from a different line is putting tacos on their customer’s plate. All of these tasks are happening at the same time. This is an example of concurrency.

JavaScript is a single-threaded programming language, meaning it processes operations one at a time. This is similar to reading a book. You start at the beginning and read one word at a time until you reach the end. You do not skip around or read multiple words at a time. Originally, JavaScript was designed to handle simple operations like validating input fields in a form or showing simple messages in a webpage. Under these use cases, single-threaded processing was sufficient in satisfying these requirements. However, as the language and technology grew, so did the need for support for currency and asynchronous operations.

Concurrency is achieved in JavaScript through the use of callback functions, promises, and async/await (covered in the next chapter). Each of these design paradigms allows JavaScript to perform more complex tasks like, handling network requests without blocking the single thread and making the website unresponsive. Let’s take a look at a simple program that simulates concurrency using the Promise.all function.

```
function serveMashedPotatoes() {
 return new Promise((resolve) => {
  setTimeout(() => {
    console.log("Server 1: Mashed potatoes served.");
    resolve('Mashed potatoes served');
  }, 2000);
```

```
  });
}

function serveHamburger() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Server 2: Hamburger served.");
      resolve('Hamburger served');
    }, 3000);
  });
}

function serveTacos() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Server 3: Tacos served.");
      resolve('Tacos served');
    }, 1000);
  });
}

// Call the functions concurrently
Promise.all([serveMashedPotatoes(), serveHamburger(),
serveTacos()]).then((messages) => {
  console.log("All tasks completed successfully!");
  console.log(messages); // This will log an array: ['Mashed potatoes served',
'Hamburger served', 'Tacos served']
}).catch((error) => {
  console.log("An error occurred:", error);
});
```

In this example, each server task is a function that returns a promise, similar to the code in the cooking example. The promise resolves after the setTimeout duration finishes. The tasks are ran concurrently using the Promise.all function. And a single then and catch function are used for fulfilled and rejected operations. If any task fails (rejected), the catch handler “catches” the error and prints a message and the error object. The tasks are run “concurrently” and all promises are invoked at the same time. If one of the tasks fails (rejected), then all promises fail. Promise.all fails if any of the promises it’s waiting on fail. If you want to wait for all promises to settle regardless of whether a promise is resolved or rejected, you would use Promise.allSettled instead.

```
function serveMashedPotatoes() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
      console.log("Server 1: Mashed potatoes served.");
      resolve('Mashed potatoes served');
    }, 2000);
  });
}

function serveHamburger() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Server 2: Hamburger served.");
      resolve('Hamburger served');
    }, 3000);
  });
}

function serveTacos() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Server 3: Tacos served.");
      resolve('Tacos served');
    }, 1000);
  });
}

Promise.allSettled([serveMashedPotatoes(), serveHamburger(),
serveTacos()]).then((results) => {
  results.forEach((result) => {
    if (result.status === "fulfilled") {
      console.log(result.value);
    } else {
      console.log("An error occurred:", result.reason);
    }
  });
});
```

In this code example, Promise.allSettled is used to wait for all of the promises to finish before handling any potential errors. With this approach, the results array contains the results of all the promises. Therefore, you must iterate over each result object to determine whether the promise was fulfilled or not. The advantage of this approach is we can now programmatically decide what happens when a single promise in a chain of promises fails. The use cases for promise chains, Promise.all, and Promise.allSettled are:

1. **Promise Chain:** Use promise chains when you need one task to complete before the second task can be started. And the second task must complete before the third task can be started.

2. **Promise.all:** Use Promise.All when you need all of the tasks to run concurrently, but you need to wait until all of them are done before moving on to something else in your program.

3. **Promise.allSettled:** Use Promise.allSettled when you need all of the tasks to run concurrently (same as Promise.all) and you need to wait until all of them are done before moving on to something else in your program. But, this time, even if one of the promise fails, you still want the other promise to run.

Programming Exercise

In this assignment you will create a Chef dashboard that displays information about a chef and handles errors when a chef's data cannot be retrieved using JavaScript promises, HTML, and CSS.

1. Create an HTML page with three sections. Each section will display the information for a different chef. Use CSS styling to bring your dashboard to life.

2. In your JavaScript code, create an array of chef objects. Each object should contain properties for name, specialty, weakness, and restaurant location.

3. Create three functions, each function should return a promise that "retrieves" the data for a different chef in the object array. Use the setTimeout to simulate a delay in retrieving the data. Set each function's timer to a different value starting with 2 seconds and expanding from there.

4. Use Promise.allSettled to make sure all chef data is retrieved before updating the webpage. The function should handle both fulfilled promises and rejected promises.

5. If a promise is fulfilled, update the webpage with the chef data. If a promise is rejected, display an error message in the webpage under the section where the promise failed. For example, if the first promise fails, the error message should be placed under the first section in the webpage. If the second promise fails, the error message should be placed under the second section in the webpage, and so on.

Hint:

- To simulate a promise rejection, you can randomly reject promises in your retrieve functions using Math.random() or you could replace one of the resolves with a

rejection. Just make sure you uncomment the resolve line and comment the reject line when you want to rest fulfilled states.

Chapter 4. Async/Await

Chapter Overview

One of JavaScript's most powerful features is its ability to handle asynchronous code execution, which allows for faster and non-blocking processes. However, asynchronous code can be difficult to manage. When it comes to handling asynchronous actions, JavaScript developers have traditionally turned to callbacks and promises. The async/await feature of ECMAScript 2017 (ES8) greatly simplifies and makes controlling asynchronous programming more intuitive. This chapter will introduce the basics of async/await and examine how to use it in JavaScript code to create simpler, more efficient applications.

Chapter Learning Objectives

By the end of this chapter, you should be able to:

- List the differences between callback functions, promises, and async/await operations.
- Identify use cases for async/await operations.
- Explain how error handling is used in an async/await operation.
- Build a JavaScript program using async/await.

Async/Await vs. Promises

In Chapter 3, it was stated that, "JavaScript promises are objects that indicate the completion or failure of an asynchronous operation." And, that promises are comprised of the following states: pending, fulfilled, and rejected. Here is how you create a Promises in JavaScript:

```
let promise = new Promise(function(resolve, reject) {
  // Asynchronous operation
});
```

Promises are like pre-ordering a video game from a store. The store is "promising" to hold a copy of the game for you so when it is release you can play it.

Async/Await is like a VIP ticket at a music concert. When you have a VIP ticket, you do not have to wait in line; you can go directly to the front and wait in the VIP lounge. In other words, you can "await" in the VIP lounge until the concert starts.

In JavaScript, async and await are keywords and extensions of Promises. They allow for you to work with Promises in a more efficient and concise way. Here is how you declare an asynchronous operation using the async keyword.

```
async function myFunction() {
  // Asynchronous operation; you can use the await keyword here...
}
```

The await keyword is used to pause the execution of the function until the Promise is resolved or rejected. For example,

```
let value = await promise;
```

The term "syntactic sugar" in programming refers to the syntax used in programming languages that is intended to make ideas easier to convey and understand. It makes the language "sweeter" by allowing ideas to be stated more clearly and concisely. In JavaScript, the async/await syntax is syntactic sugar for Promises. That is, they do not add any new functionality to the language, rather it makes asynchronous code easier to read and write. In "Chapter 3" I showed an example of how to use promises to simulate the pre-order video game analogy. Here is an example of how we could write this program using async/await.

```
let preOrderGame = new Promise((resolve, reject) => {
  let gameIsReleased = true;
  if(gameIsReleased) {
     resolve('Game is released, start playing!');
  } else {
     reject('Game is not released, get a refund.');
  }
});

async function playGame() {
  let message = await preOrderGame;
  console.log(message); // This will run if the Promise is fulfilled
}

playGame();
```

In this code example, playGame is an async function. In the body of this function, we use await to pause the execution of the asynchronous operation until the preOrderGame has either fulfilled or been rejected. If preOrderGame is fulfilled, the value is assigned to the message variable and printed to the console window. One item of note, then can only be used on code blocks that return a promise. This is because it is part of the Promise

prototype and can only be used on instances of Promise or on objects that implement a then method that follows the Promise specification. In other words, you cannot do this:

```
playGame().then(message => {
  console.log(message);
});
```

But, if we convert the playGame function to return a promise, we can. For example,

```
let preOrderGame = new Promise((resolve, reject) => {
  let gameIsReleased = true;
  if(gameIsReleased) {
     resolve("Game is released, start playing!");
  } else {
     reject("Game is not released, get a refund.");
  }
});

async function playGame() {
  let message = await preOrderGame;
  console.log(message); // This will run if the Promise is fulfilled
  return message;
}

playGame().then(message => {
  console.log("Game status: " + message);
});
```

In this code example, we await preOrderGame and assign it to the message variable, but instead of printing it to the console, we return it. By doing so, we covert the playGame function into a JavaScript promise, which allows us to use then to handle the fulfilled state and catch for the rejection state. But, as you probably guessed, this “technically” defeats the purpose of using async/await. In the next section we will explore a better approach for handling errors when using async/await.

Error Handling

Handling errors in an asynchronous operation that uses promises is accomplished through then and catch. In operations that use async/await, errors are handled through try/catch blocks. Let us take the preOrderGame example from the previous section. To add error handling, we would update the code to the following:

```
let preOrderGame = new Promise((resolve, reject) => {
  let gameIsReleased = true;
```

```
 if(gameIsReleased) {
   resolve("Game is released, start playing!");
 } else {
   reject("Game is not released, get a refund.");
 }
});

async function playGame() {
 try {
  let message = await preOrderGame;
  console.log(message); // This will run if the Promise is fulfilled
 } catch (error) {
  console.error(error); // This will run if the Promise is rejected
 }
}

playGame();
```

The only difference between this code segment and the previous one is the inclusion of a try/catch block. Anytime you write code that uses async/await, you should always include a try/catch block to handle the fulfilled and rejection states. Otherwise, if the promise is rejected, an unhandled error will be thrown that could potentially halt the execution of your program. Here is another example using async/await with our cooking tasks, this time incorporating error handling:

```
function chefATask() {
 return new Promise((resolve) => {
  console.log("Chef A starts their task.");
  setTimeout(() => {
   console.log("Chef A finishes baking the cake.");
   resolve();
  }, 5000);
 });
}

function chefBTask() {
 return new Promise((resolve) => {
  console.log("Chef B starts their task.");
  setTimeout(() => {
   console.log("Chef B finishes making the sauce.");
   resolve();
  }, 3000);
 });
}
```

```
function chefCTask() {
  return new Promise((resolve) => {
    console.log("Chef C starts their task.");
    setTimeout(() => {
      console.log("Chef C finishes preparing the salad.");
      resolve();
    }, 1000);
  });
}

// Call the functions in order
async function performTasks() {
  try {
    await chefATask();
    await chefBTask();
    await chefCTask();
    console.log("All tasks completed successfully!");
  } catch (error) {
    console.error("An error occurred:", error);
  }
}

performTasks();
```

In this code example, performTasks is an asynchronous function that uses await on each of the check tasks. The order of execution is controlled because await is used to pause the execution until each of the chef tasks have either fulfilled or been rejected. If all tasks are fulfilled, “All tasks completed successfully” is printed to the console window. If any of the tasks are rejected, the catch block “catches” the rejection and prints an error message to the console window. The main advantage of this approach is its readability. The async/await version:

```
async function performTasks() {
  try {
    await chefATask();
    await chefBTask();
    await chefCTask();
    console.log("All tasks completed successfully!");
  } catch (error) {
    console.error("An error occurred:", error);
  }
}
```

Is easier to read and understand than the promise-based version:

```
chefATask()
 .then(() => {
  return chefBTask();
 })
 .then(() => {
  return chefCTask();
 })
 .then(() => {
  console.log("All tasks completed successfully!");
 })
 .catch((error) => {
  console.error("An error occurred:", error);
 });
```

But, if you are not convinced, imagine if we had 10 chef tasks that needed to be executed in order. First, let's look at the promise-based version, often referred to as "Promise Hell":

```
chefATask()
 .then(chefBTask)
 .then(chefCTask)
 .then(chefDTask)
 .then(chefETask)
 .then(chefFTask)
 .then(chefGTask)
 .then(chefHTask)
 .then(chefITask)
 .then(chefJTask)
 .then(() => {
  console.log("All tasks completed successfully!");
 })
 .catch((error) => {
  console.log("An error occurred:", error);
 });
```

As you can see, the code becomes harder to read as additional tasks are added to the program. Now, let's take a look at the callback-based version, often referred to as "Callback Hell":

```
chefATask((error) => {
 if (error) {
  console.log("An error occurred:", error);
 } else {
  chefBTask((error) => {
```

```
    if (error) {
      console.log("An error occurred:", error);
    } else {
      chefCTask((error) => {
        if (error) {
          console.log("An error occurred:", error);
        } else {
          // ... continue this pattern for all tasks
          chefJTask((error) => {
            if (error) {
              console.log("An error occurred:", error);
            } else {
              console.log("All tasks completed successfully!");
            }
          });
        }
      });
    }
  });
 }
});
```

Again, the code becomes even more difficult to read and manage as more tasks are added. Finally, let's look at the async/await version:

```
async function performTasks() {
 try {
   await chefATask();
   await chefBTask();
   await chefCTask();
   await chefDTask();
   await chefETask();
   await chefFTask();
   await chefGTask();
   await chefHTask();
   await chefITask();
   await chefJTask();
   console.log("All tasks completed successfully!");
 } catch (error) {
   console.error("An error occurred:", error);
 }
}

performTasks();
```

In this version, each task is awaited in sequence. If any task fails (i.e., the promise it returns is rejected), execution will stop and the error will be caught and logged. If all tasks succeed, "All tasks completed successfully!" will be logged. As you can see, the readability is significantly improved with async/await. In the next section we will explore best practices and when you would use one strategy (callback functions, promises, and async/await) over the other.

Best Practices

Having examined the distinctions among callback functions, JavaScript promises, and async/await, it is now appropriate to delve into the best practices and optimal scenarios for employing callbacks, promises, and async/await.

1. **Callbacks:** Callbacks play a fundamental role in understanding the asynchronous behavior of JavaScript. They are frequently used in event handlers, timers, and low-level I/O operations, but can create "callback hell" when managing intricate asynchronous control flows. In enterprise-level applications, callbacks are typically used when a straightforward, single-level asynchronous operation is required, or when interacting with libraries or APIs that rely on callbacks.

2. **Promises:** Asynchronous operations can be handled more effectively using promises compared to callbacks. Promises offer better error handling and help to avoid complex and nested code blocks. They are ideal for managing a sequence of asynchronous operations, and allow for improved error propagation and chaining. Promises are widely used in modern web APIs and libraries.

3. **Async/Await:** Async/await is a programming construct that simplifies writing asynchronous code by making it appear and behave like synchronous code. This greatly enhances code readability and facilitates error handling through the use of try/catch blocks. Async/await is particularly useful when dealing with multiple asynchronous operations that have dependencies or when handling complex control flow. It is quickly becoming the preferred approach for writing asynchronous code in enterprise-level applications.

4. **Error Handling:** Irrespective of the approach used, it is crucial to handle errors appropriately. Callbacks necessitate error handling in each callback, while Promises employ the catch function to handle errors, and async/await permits the use of standard try/catch blocks for error handling and propagating errors.

5. **Interoperability:** It is important to note that callbacks, Promises, and async/await can work together seamlessly. Callbacks can be transformed into Promises to function with Promise-based code, and async/await is compatible with any function that provides a Promise return.

6. **Testing and Debugging:** Testing and debugging have varying implications for each approach. Promises and async/await can simplify the testing process since they can be directly returned and awaited in test cases.

7. **Performance Considerations:** Improper usage of async/await code can result in performance issues, such as unnecessary sequential execution, despite it being more readable. Optimal usage of async/await can be attained by having a thorough understanding of the Promise specification and how they work in a JavaScript program.

When developing JavaScript applications at an enterprise level, comprehending the distinctions between callbacks, Promises, and async/await is critical. The decision of which method to use frequently hinges on the applications specific use cases, the intricacy of the asynchronous control flow, and the type of code base you are working with.

Programming Assignment

Create a webpage that displays information about movies from an in-memory array using JavaScript async/await, HTML, and CSS.

1. **HTML:** Build a form for the user to enter a movie title, and sections to display the movie's title, director, release year, and a brief synopsis.

2. **CSS:** Style your webpage to your liking. Make sure the movie information is clearly visible and the layout is user-friendly.

3. **JavaScript:**

    a. Define an array of movie objects. Each object should have properties for the title, director, release year, and synopsis.

    b. Define a function named fetchMovie(title). This function should return a Promise that simulates fetching data for the given title from the array. Use setTimeout to simulate the delay of a network request. The resolved value should be the movie object that matches the title.

    c. Define an async function named displayMovie. This function should be called when the user submits the form. It should use await to wait for fetchMovie to complete, then update the HTML of the page to display the movie information.

4. **Error Handling:** Your fetchMovie function should reject with an error message if the movie title is not found in the array. Your displayMovie function should handle these errors and display an appropriate message on the webpage.

References

Copilot. (n.d.). OpenAI. *Microsoft Copilot*. computer software. Retrieved December 19, 2023, from https://www.microsoft.com/en-us/microsoft-copilot.

“JavaScript Reference.” *MDN Web Docs*, developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.