

Asynchronous Functional Javascript

{ with: categories }

A deep dive into Asynchronous Javascript



Dimitris Papadimitriou

Asynchronous FUNCTIONAL PROGRAMMING IN JS WITH CATEGORIES

A deep dive in Asynchronous JS

Dimitris Papadimitriou

This book is for sale at <https://leanpub.com/functional-programming-in-js-with-categories>

This version was published on 2019-06-08

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.

Feel free to contact me at:

<https://www.linkedin.com/in/dimitrispapadimitriou/>

<https://leanpub.com/u/dimitrispapadim>

<https://medium.com/@dimpapadim3>

<https://github.com/dimitris-papadimitriou-chr>

dimitrispapadim@live.com

Resources

Acknowledgments

<https://pixabay.com/illustrations/parrot-bird-macaw-beak-tropical-2979285/>

© 2020 Dimitris Papadimitriou

Contents

About this book coding conventions	4
1 Introduction to Asynchrony.....	8
1.1 Concurrency, Parallelism, Asynchrony.....	8
1.1.1 Concurrency	8
1.1.2 Parallelism	8
1.1.3 Threads	8
1.1.4 Synchronous and Asynchronous.....	9
1.2 The Problem of Asynchrony	10
1.3 The JavaScript Event Loop	10
1.4 Showcasing of Async Technologies.....	13
1.5 Callbacks	14
1.6 The Continuation monad.....	14
1.7 Promises.....	15
1.8 Async-Await.....	15
1.9 Events.....	16
1.10 The basic Criteria of Comparison.....	18
2 Callbacks	20
2.1 Continuation Passing Style Transformation.....	20
2.2 Continuations and Callbacks.....	21
2.3 Callbacks Example	22
2.4 Including Error Handling.....	24
3 Why Functional Programming	26
3.1 Showcase – Bind for Some Async Structures	26
3.1.1 Multiplying stuff	26
3.2 Composing Continuations	29
3.2.1 The Continuation monad.....	31
3.3 Folding Callbacks and Continuations	32
4 Fluture.js.....	33
4.1.1 Error handling	38
5 Async-Await.....	39
5.1 Async.....	39
5.2 The await keyword.....	40
5.3 Error handling	Error! Bookmark not defined.
5.4 Refactoring Promises to Async/await.....	Error! Bookmark not defined.
5.5 Sequential and Parallel execution	Error! Bookmark not defined.

Purpose

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

— Edsger Dijkstra

OO makes code understandable by encapsulating moving parts.
FP makes code understandable by minimizing moving parts.

—Michael Feathers (Twitter)

One of the main reasons for this book is to transfer in the community of object-oriented developers some of the ideas and advancements happening to the functional side. This book wants to be pragmatic. Unfortunately, some great ideas are coming from academia that cannot be used by the average developer in his day to day coding.

This is not an Introductory book to functional programming, even if I restate some of the most basic concepts in the light of category theory and the object-oriented paradigm. This book covers the middle ground. Nonetheless, it is written in a way that if someone pays attention and goes through the examples, he or she could understand the concepts.

In this book, I will try to simplify the mathematical concepts in a way that can be displayed with code. If something cannot be easily displayed with code probably will not be something that can be readily available to a developer's arsenal of techniques and patterns.

If you think a section is boring, then skip it and maybe finish it later.

One thing I admire in Software Engineers is their ability to infer things. The ability to connect the dots is what separates the exceptional developer from the good one. In some parts of the book, I might indeed have gaps or even assume things that you are not familiar with. Nonetheless, I am sure that even an inexperienced developer will connect the dots and assign meaning, as I usually do when left with unfinished

Category theory?

You might ask: Why use Category theory and the Functional Paradigm in Conjunction to Object oriented Programming? Category theory (and functional programming) provide a simple framework to **unify** different asynchronous concepts and the mechanics behind some of the most popular JS Async libraries.

You will see that concepts like functors monads, pattern matching etc can be found across all the JS async constructs like Callbacks, promises, tasks, futures, observables. This allow us to have a **unified understanding** and not be lost in the details of any specific technology.

About this book coding conventions

A quick stop here. Most of the code in this book use vanilla JavaScript. Instead of the standard way to create a class:

```
var Rectangle = function (height, width) {
  this.height = height;
  this.width = width;
  this.area = function () {
    return this.height * this.width;
  }
}
```

Most of the times I will use this brief notation:

```
const Rectangle = (height, width) => ({
  area: () => height * width,
  scale: (s) => Rectangle (s * height, s * width),
});
```

Object literal notation forces immutability. In this way of writing, you must always return a new instance. In a larger Domain Model, this way of writing might not be viable. Here we do not need to use new to create an object. Also, we cannot access the height and width

variables. If you want to mutate the state, you either must question your design of the object that forces you to think in a defensive programming style. Let's be pragmatic here, this style of coding that promotes purity should **not be an end in itself**, and we should always be able to convince ourselves to go back into an object-oriented style of coding even if it is not that pure. The important thing is to write **functional code that provides business value** to the organization and to the end-user that will eventually use it. We must be pragmatic and utilitarian when we code and abstractionists when we think about code.

Overview

I want to give a very general idea about the structures that we are going to meet in this book in the most simplified way I could come up with. This is the 10.000 feet view of the first half of the book.

1. Two ways to look an evaluation - Push and Pull

Let's start with this simple expression.

```
var s = a => b => a(b)
```

(that's a simple application by the way) we could write the following

```
s(console.log)(5) // prints 5
```

We have an expression that waits for a value, and when we finally give the value, it evaluates the expression. Also, we can separate the two parts

```
var t = s(console.log);  
  
setTimeout(() => { t(5) }, 3000);
```

Run This: [Js Fiddle](#)

We can save the execution in a variable, and at some later time when we have a value, inject it to the expression and finish our execution.

We can write another expression-based on our s

```
s(x => x(5))(console.log)
```

Here we reverse the order and first give the value that we want to use and then at the end the action that we want to perform. The reason behind `x => x(5)` is that because in our s we apply the a on b like this `a(b)` we have to use a callback to reverse the order so tis cancel out [try to create this by yourself]. Now I will rename this x variable to resolve:

```
s(resolve => resolve(5))(console.log)
```

Hopefully, a promise comes to mind. The above expression is equivalent to this promise

```
new Promise (resolve=>resolve(5)).then(console.log)
```

as you would expect this should work in a delayed fashion as well. Or it would not be a true promise

```
s(resolve => setTimeout(() => { resolve(5) }, 3000) )(console.log)
```

Run This: [Js Fiddle](#)

2. Adding a mapping function

We will extend our s.

```
var s = a => f => b => a(f(b))
```

here I added another variable f in the middle of all this. now we can write this

```
s(console.log)(x => x + 2)(5)
```

luckily for us because of the signature we could probably add many f's

```
s(console.log) (x => x + 2) (x => x + 3) (5)
```

so nice... but it does not work. This does though:

```
s(console.log) (s (x => x + 2) (x => x + 2)) (5)
```

Run This: [Js Fiddle](#)

So, let us go back to the other way of looking at things

```
s(resolve => resolve(5))(x => x + 2)(console.log)
```

this does not work because everything is reverse, but this could

```
s(resolve => resolve(5))(f => x => f(x + 2))(console.log)
```

now I can define this map operator and use this

```
var map = g => f => x => f(g(x))
```

```
s(next => next(5))(map(x => x + 2))(console.log)
```

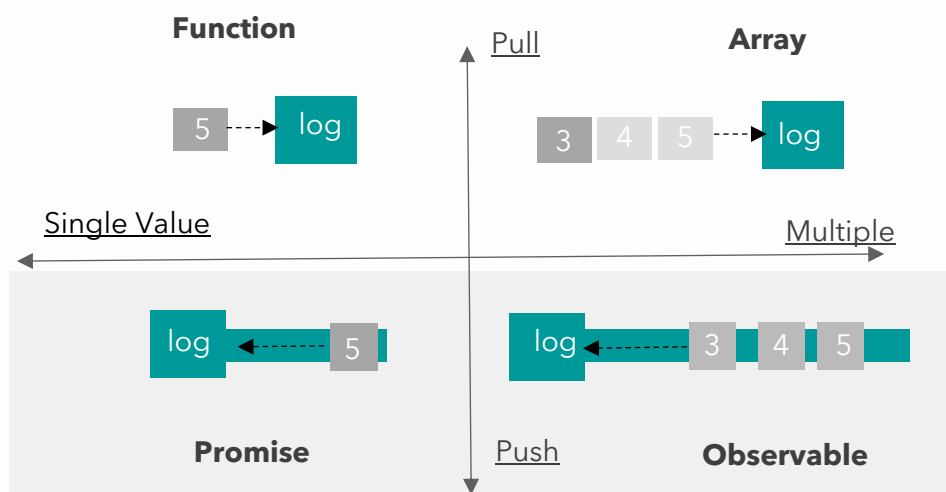
Run This: [Js Fiddle](#)

does it look like this [map from Rx.js](#) ?

We saw two ways of making a computation consisting of two parts a value and an action on the value.

Having that in mind will help you recognize in the first form `s(console.log)(5)` the Reader, the IO and the State functors, and in the second form `s(resolve => resolve(5))(console.log)` the continuation monad, promises, and the observables.

We call the First **pull system** since we already have the value and we just pull them whenever we need them , but in the second case we have the action first so we leave back a callback in which we push the value in a latter time so we can call this **Push system**



1 Introduction to Asynchrony

1.1 Concurrency, Parallelism, Asynchrony

In this first section we are going to have a very brief view on the different concepts around relating to asynchrony.

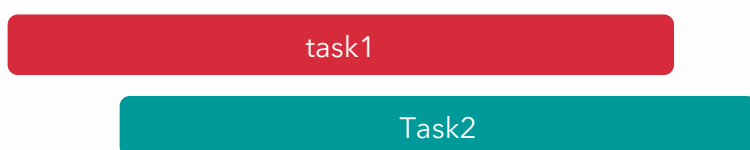
1.1.1 Concurrency

Consider you are given a two tasks task1 and task2. Concurrency means that you execute both tasks even if you alternate the execution and not wait for the one to finish in order to start the other.



1.1.2 Parallelism

Parallelism means performing two or more tasks simultaneously. Parallel computing in computer science refers to the process of performing multiple calculations simultaneously.



1.1.3 Threads

In computer science, a thread of execution **is the smallest sequence of programmed instructions** that can be managed independently the operating system.

JavaScript engine Runs on a single thread. Recent web browsers provide a way to overcome this potential performance issue. The HTML5 specification introduces Web Workers on the browser side to allow JavaScript code to run to multiple threads.

1.1.4 Synchronous and Asynchronous

In a synchronous programming model, tasks are executed one after another. Each task waits for any previous task to complete and then gets executed. This means that a task blocks everything until finished.

Consider the following script:

```
let getClient = () => {
  let n = 10000000000;
  while (n > 0) { n--; }
  return { id: 1, name: "rick" }
}

let client = getClient();

doSomethingElse();

console.log(client.name)
```

This simulate a cumbersome time-consuming process like accessing a Database

The execution will stop here until the `getClient` finish

the `doSomethingElse()` function cannot be called by the js Engine until the `getClient()` finishes its execution.



In an asynchronous programming model, when one task gets executed, you could switch to a different task without waiting for the previous to get completed. Asynchrony can be achieved in a single thread by switching tasks. In the following script the `setTimeout` will execute the `getClient()` code inside with 1 second delay in the meantime the execution will continue and proceed at the `doSomethingElse` function from that point on the JS engine will execute both asynchronously on a single tread by alternate the execution as we will see in the following section.

```
let getClient = () => {
  let n = 10000000000;
  while (n > 0) { n--; }
  return { id: 1, name: "rick" }
}

setTimeout(() => {
  let client = getClient();
  console.log(client.name)
}, 1000);

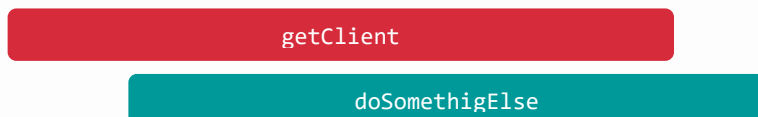
doSomethingElse();
```

This simulate a cumbersome time-consuming process like accessing a Database

This is an **Asynchronous non-parallel** execution



We could also **have an Asynchronous parallel execution** if we could use web workers



1.2 The Problem of Asynchrony

if we modify our initial example

```
var getClient = () => {
    return { id: 1, name: "rick" }
};
var client = getClient();
console.log(client.name)
```

a bit in order to add a time delay around the return statement in the getClient function, then immediately we have a problem. So, the following does not work

```
var getClient = () => {
    setTimeout(() => {
        return { id: 1, name: "rick" }
    }, 1000);
};
var client = getClient();
console.log(client.name)
```

[Run This: Js Fiddle](#)

Now in this situation we get an error because the client is undefined. in the following section we are going to see the mechanics of why this is happening.

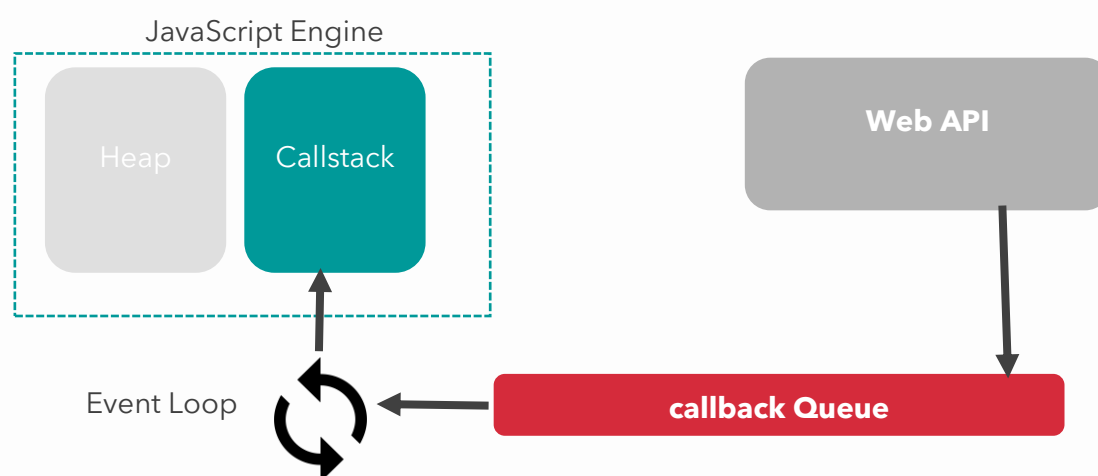
1.3 The JavaScript Event Loop

If you ever created a simple game that runs on a loop, or created any small operating system you should be already familiar with the concept of the Event Loop. The Event Loop has one simple job – to monitor the Call Stack and the Callback Queue. If the Call Stack is

empty, it will take the first event from the queue and will push it to the Call Stack, which effectively runs it. We will briefly explain some of the concepts here but if you are interested in further understanding how the JS engine works you can watch the very popular presentation “[What the heck is the event loop anyway?](#)” by Philip Roberts and also visualize some simple scripts on the <http://latentflip.com/loupe> .

The Basic Circuit consists of four parts for our simple asynchronous scenario

1. The **Call-stack** of the **JavaScript Engine**
2. The **Queue**
3. The **Web API**
4. The **Event Loop**



The JavaScript has a relatively simple way to execute seemingly Asynchronous code on a single thread.

The `setTimeout` is provided to us by the Web API: it lets us delay tasks without blocking the main thread. The callback function that we passed to the `setTimeout`

1) The Call- Stack

The **call stack** is where the **JavaScript engine** stores information about the functions that are currently executed. The [MDN](#) Documentation Describes the Call stack :

“A **call stack** is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions – what function is currently being run and what functions are called from within that function, etc.”

When we write the function in our second script

```
var getClient = () => {
  setTimeout(() => {
    return { id: 1, name: "rick" }
  }, 1000);
}
```

```
};
```

This `() => {...}` lambda which is a function, is added on the call stack **when we invoke it** using `getClient()` [**Pay attention to the fact that the function is added only at the time when we invoked it.**]

The function is executed at that point and the JavaScript Engine Encounters the `setTimeout`.

2) The WebAPI

This `setTimeout` function it not part of the JavaScript Engine but it belongs to the Client-side web APIs. The Web API's are extensions that provide additional functionality. You can take a look at the list of the Web APIs available . Some of the most popular are the DOM API and the fetch API that you may already be familiar with. **The Web API monitors and is responsible for waiting for the `setTimeout` to expire.**

```
setTimeout(() => {
  return { id: 1, name: "rick" }
}, 1000);
```

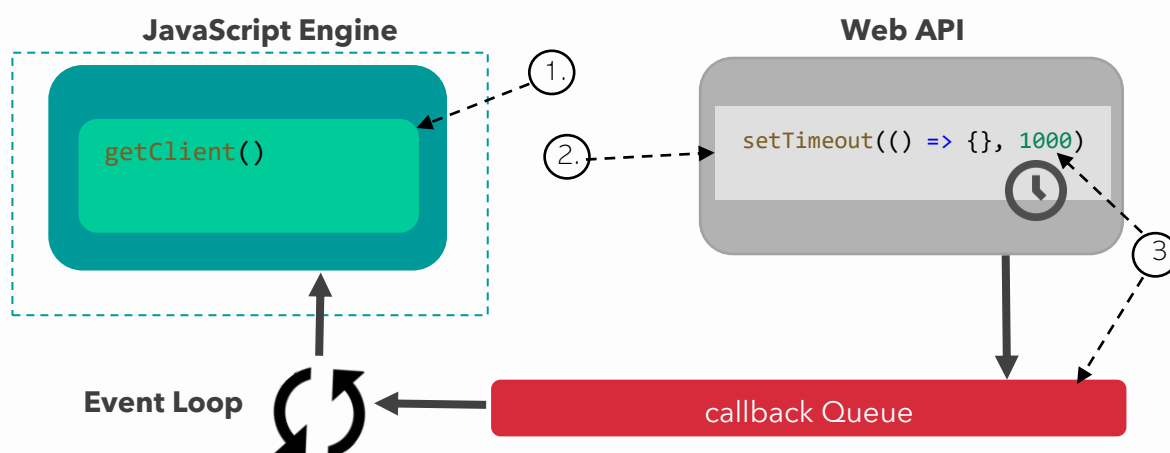
then the callback function of the `setTimeout` (the function inside the set the timeout)

```
((() => { return { id: 1, name: "rick" } } )
```

is added on the **callback Queue**.

The JS engine moves to the next line `console.log(client.name)` and try to execute this by placing it to the callstack.

Let's graphically review this first steps in the following diagram



1. The engine puts `getClient()` on the Call stack and immediately execute it

2. The `setTimeout` is handled by the WebAPI .
3. the WebAPI which waits for 1 sec. in the Meanwhile the Engine continues execution on the next line after the `getClient()`

3) The Queue

The Queue is a place of integration for the JavaScript Engine with the outside world. Different API's can place orders in the Queue that will eventually be executed.

4) The Event Loop

Finally, the event loop constantly searching the queue for tasks that should be placed on the Callback-stack in order to be executed by the Javascript Engine. In essence the **Event Loop connects the queue with the call stack**. If the call stack is **empty**, so if all previously invoked functions have returned their values and have been **popped off the stack**, the **first item in the queue gets added to the call stack**.

And this completes the lifecycle of execution for our simple asynchronous script. [If you still want to watch this in action go to the <http://latentflip.com/loupe>] you can run the simple [asynchronous example here](#), or place the following [that has functions instead of lambdas] and visualize the execution flow:

```
var getClient = function(){
  setTimeout(function() {
    return { id: 1, name: "rick" };
  }, 1000);
};

var client = getClient();
console.log(client);
```

Visualize This: [loupe](#)
Run This: [Js Fiddle](#)

the execution calls the `getClient` but then the execution continues and reaches the console log statement and only after 1 second, we reach the return statement but it's already too late.

1.4 Showcasing of Async Technologies

What follows in this section is a showcase of the ways different solutions tried to solve the asynchrony problem. We are going to see in detail each concept in the following sections. The point of this section is just to let you see how the different concepts evolved from gradually from callbacks to observables.



1.5 Callbacks

This is only for understanding callbacks are the base of all the synchronous solution and in this section we're going to see how they progressively evolve to the continuation promises in the object oriented toward this translate to events and the observer design pattern finally the reactive extension observables that we have today

```
var getClient = (resolve) => {
  setTimeout(() => {
    resolve({ id: 1, name: "rick" })
  }, 1000);
};

getClient(client => {
  console.log(client.name)
});
```

Visualize This: [loupe](#)
Run This: [Js Fiddle](#)

1.6 The Continuation monad

This is only for understanding purposes. Continuations are an intermediate step between the callback solution and the Promise solution. It's like a lost ancestor in the evolution of solutions. The only point here is that the callback is now curried (which means removed from the argument and instead is returned as a function that takes the argument as we will see in more detail) if you see the three solutions of the callback continuation and promises at side to side you will see the callback morphing to a promise through the continuation concept you will see that the promise of the continuation

```
var getClient = () => {
  return (resolve => {
    setTimeout(() => {
      resolve({ id: 1, name: "rick" })
    }, 1000);
  });
}
```

Here the callback has been moved inside and returned as a lambda

```

    })
  };

  getClient()(client => {
    console.log(client.name)
  });

```

The above change now allows us to chain the continuation outside the parenthesis

Run This: [Js Fiddle](#)

1.7 Promises

The promises are the object-oriented equivalent of the continuation monad. So, it's basically continuation monad that is widely used in the functional programming but also, we made it into an object .

```

var getClient = () => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ id: 1, name: "rick" })
    }, 1000);
  })
};

getClient().then(client => {
  console.log(client.name)
});

```

The `resolve` now is placed inside the constructor of a `Promise` object

The `Promise` exposes a `then`. Instead of the `()(client =>...`

of the continuation above

Run This: [Js Fiddle](#)

1.8 Async-Await

Async/await is just additional syntactic support for promises. It's surprisingly easy to understand and use if you grasp Promises

```

var getClient = () => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ id: 1, name: "rick" })
    }, 1000);
  })
};

var client = await getClient();
console.log(client.name)

```

The `await` is a syntactic sugar that removes the need for the `then` because the engine treats the rest as been inside a `then` clause

Run This: [Js Fiddle](#)

1.9 Events

Now if we go one step further in an Object-Oriented direction as sidetrack, we can remove the callback from the function and make it a property of the Object that contains the function.

```
class Repository {
  getClient() {
    setTimeout(() => {
      this.resolve({ id: 1, name: "rick" })
    }, 1000)
  }
}

var repo = new Repository();

repo.resolve = ((client) => {
  console.log(client.name)
})

repo.getClient();
```

The `this` Refers to the `Repository`. we call an Event Handler that is attached as a property to the containing object

Assigning an Event handler to the Event. This is the function that was in the `Promise().then`

Run This: [Js Fiddle](#)

If we rewrite the code by just renaming the `resolve` to `onFinished` you might recognize the usual Event driven model of programming where the objects has some Events that its methods can fire and we can assign eventHandlers to those events that would be called when the events are fired.

```
class Repository {
  getClient() {
    setTimeout(() => {
      this.onFinished({ id: 1, name: "rick" })
    }, 1000)
  }
}

var repo = new Repository();

repo.onFinished = ((client) => {
  console.log(client.name)
})

repo.getClient();
```

Run This: [Js Fiddle](#)

The Events of the object-oriented programming is nothing more than assigning the callback as a property of the object.

The Observer Design Pattern

The observer design pattern is just an extension of the event driven design pattern as we saw in the previous section here instead of just one event handler, we can have multiple event handlers that we call observers.

```
class Repository {
  constructor() {
    this.observers = []
  }
  subscribe(observer) {
    this.observers.push(observer)
  }
  notify(value) {
    this.observers.forEach(observer => {
      observer.next(value)
    })
  }
  getClient() {
    setTimeout(() => {
      this.notify({ id: 1, name: "rick" })
    }, 1000)
  }
}

var repo = new Repository();
repo.subscribe(({
  next: (client) => {
    console.log(client.name)
  }
})))

repo.getClient();
```

Run This: [Js Fiddle](#)

Reactive Extensions - Observables

The observables are just promises for multiple values. The observables extract the observer design pattern mechanism into a separate construction that can be reused. So instead of implementing the observer pattern to any specific object that needs observable behavior the reactive extension libraries try to abstract the mechanics in a robust, well designed, reusable library.

```
import Rx from 'rxjs';

const Observable = Rx.Observable;
```

```
const observable = new Observable(subscriber => {
  setTimeout(() => {
    subscriber.next({ id: 1, name: "rick" });
  }, 1000);
});

observable.subscribe({
  next(x) { console.log(x.name); }
});
```

Run This: [Js Fiddle](#)

now we can for example refactor our previous repository example and use the reactive extension library then instead of the explicit implementation of the observer design pattern. this would look like that:

```
const Observable = Rx.Observable;

class Repository {
  getClient() {
    return new Observable(subscriber => {
      setTimeout(() => {
        subscriber.next({ id: 1, name: "rick" });
      }, 1000);
    });
  }
}

var repo = new Repository();

repo.getClient().subscribe(({
  next: (client) => {
    console.log(client.name)
  }
})))
```

Run This: [Js Fiddle](#)

in this example obviously because the repository returns only one value the observable here is overused and we could just have used a promise. But this first example just showcases the difference solutions of the asynchronous problem so you can see the way everything is unified.

1.10 The basic Criteria of Comparison

Any of these tools should obviously provide a solution to the basic problem but also the more probable is to be adopted when they conform to some additional criteria:

- Error handling

- Cancellation
- Resource management [capabilities to dispose resources after Finishing]
- Laziness

Also, the basic Use cases that arise often are

- Applying a function to the outcome without affecting the program
- Composing two asynchronous functions
- Error Handling on the previous two situations

Callbacks

*"Nor did they not perceive the evil plight
In which they were" ~ John Milton - Paradise Lost*

*"Not: not: it is day differs from it is day only in manner of speech"
~ Alexander of Aphrodisias*

2 Callbacks

2.1 Continuation Passing Style Transformation

We are all familiar with the idea that two negatives are equal to a positive. This is not uncommon in many natural languages as well. In some languages, double negatives cancel one another and produce an affirmative; in other languages, doubled negatives intensify the negation. We might say something like "the glass is not empty" which means that the glass is half-full (lets be optimistic).

In the formal language of mathematical logic, for any proposition P the following expression is a tautology $\neg \neg P \rightarrow P$ this tautology is equivalent with the Law of excluded middle $\neg P \vee P$ that says that either P is true or not P .

In Programming unfortunately there is no explicit concept of the negative. This goes deep into theoretical grounds of computer science. You might have heard the Curry-Howard correspondence which is a theory that says that logic and programs have a direct relation. But not classical logic but a specific logic called **Intuitionistic** in which the Principle of excluded middle $\neg P \vee P$ does not hold. Nonetheless It was proven that all classical logic sentences can be translated to Intuitionistic logic with a kind of translation called **double negation translation**. Long story short the negation $\neg P$ is now $P \rightarrow \perp$ (where you can think of \perp as something that cannot be true) which means we cannot prove P and $(P \rightarrow \perp) \rightarrow \perp$ means **we cannot prove that we cannot prove P** this is equivalent to P for the Intuitionistic logic.

Skipping some steps we can assume that in JavaScript the void - () represents the \perp and in this way we can write that $((\text{callback}) \Rightarrow \text{callback}(x))$ is equal with x if you look the type of the first object you can see that the `callback` is a function that takes a value and returns nothing : $(a \rightarrow ())$ and the $((\text{callback}) \Rightarrow \text{callback}(5))$ is a function that takes

as an input a callback $(a \rightarrow ())$ and return `void ()` so the overall type is $(a \rightarrow ()) \rightarrow ()$ [same as $(P \rightarrow \perp) \rightarrow \perp$ that we talk about above]

In practical terms the continuation passing style transformation for programming tells us that if we have a function that returns something

```
var f = function (a, b, c) {
  return (_)
}
```

We can make it into an equivalent that does not return anything and instead it takes a callback as an argument that will return the value

```
var f = function (a, b, c, callback) {
  callback(_)
}
```

For example, if we have the squaring function

```
var square = (x) => {
  return (x * x)
};
console.log(square(5))
```

we can rewrite it in a callback style

```
var square = (x, callback) => {
  callback(x * x)
};
```

And use it in this way

```
square(5, x => console.log(x))
```

2.2 Continuations and Callbacks

First of all, I want to clear out that **Callbacks and continuation are referring to the same thing**. *But* in this book, we will follow the convention:

- Callbacks $(a \rightarrow r)$
that the callback is something that has type of $(a \rightarrow r)$ just the function that is passed as an argument to another (higher order) function.

```
var square = function (x, callback) {
  callback(x * x);
}
```

```
}
```

- Continuations $(a \rightarrow r) \rightarrow r$

the continuation would be **a curried callback** having type $(a \rightarrow r) \rightarrow r$

```
var square = function (x) {  
  return callback=>callback(x * x);  
}
```

Here we **curried the callback**

The simplest form looks like this: $(c \Rightarrow c(5))$ where the c should be a function.

The callback is just a Subscription. We want to get back a value of type r but since you don't have it yet I will give you as a **token a physical reified subscription** that you will have to pass inside the value when its available. The Callbacks are in this **sense allow temporal decoupling** by inverting the dependency between the function that does something and the continuation of the program flow that needs this value in order to do more stuff.

Now unfortunately this skews the way that we normally think because now we transferred the burden of controlling the flow to the upstream code, in our squaring function example we normally don't want the function to care about control flow (by passing the result around instead of just returning it) . This looks like the famous Hollywood principle : *"Don't call us, we'll call you"* which is in the core of the inversion of control. In the case of asynchrony this inversion of control is not necessarily good since is driven from our need to fix the async problem and not decoupling domain model aspects. Most of the tools and methods developed like promises and async/await, are just an attempt to partially reduce this effect of unintended inversion.

Callbacks are the idea that everything else seem to build upon.

2.3 Callbacks Example

Let us say that we have this scenario where we have a list of clients and each client is assigned to an employee. Then let us say get the name of the assigned employee for that client.

This is the standard one-to-many relationships and the way to model it is to add on the client entity a property that will store the value of the Id of the related employee



we will use mock repositories that expose `getById` methods in order to search an array with test data and get the item that matches the given Id. The following would be the Repository for the clients.

```
var mockClientRepository = ({
  getById: (id, onFinished) =>
    setTimeout(() => {

      var client =
        [{ id: 1, name: 'rick', age: 29, employeeId: 1 },
        { id: 2, name: 'morty', age: 25, employeeId: 3 }]
        .find(client => client.id === id);

      onFinished(client)

    }, 1000)
})
```

The Employee Repository is the Same

```
var mockEmployeeRepository = ({
  getById: (id, onFinished) =>
    setTimeout(() => {

      var employee = [{ id: 1, name: 'jim', age: 29 },
        { id: 2, name: 'jane', age: 25 }]
        .find(employee => employee.id === id);

      onFinished(employee)

    }, 1000)
})
```

Given the client id we first must search the `mockClientRepository` and retrieve a client and then use the client `employeeId` in order to get the related employee by using the employee repository `getById` method.

```
var assignedEmployee = (clientId, onFinished) => {
  mockClientRepository.getById(clientId, client => {
    if (client) {
      mockEmployeeRepository.getById(client.employeeId, employee => {
        onFinished(employee)
      });
    }
  });
};

assignedEmployee(1, employee => console.log(employee))
```

Run This: [Js Fiddle](#)

this scenario occurs very often because it involves a one to many relationships between two domain model objects.

2.4 Including Error Handling

up until now we did not include the possibility of exceptions. In this section we are going to extend our previous conversation in order to also include exception handling with callbacks.

We are going to include another call back as an argument to our function. This second call back will only be used in order to pass inside the exception if an exception arises.

```
var mockClientRepository = ({
  getById: (id, onError, onFinished) =>
    setTimeout(() => {
      try {

        var client =
          [{ id: 1, name: 'rick', age: 29, employeeId: 1 },
           { id: 2, name: 'morty', age: 25, employeeId: 3 }]
          .find(client => client.id === id);

        onFinished(client)

      } catch (error) {
        onError(error)
      }
    }, 1000)
})
```

In this case when we compose the Callbacks we must take into account the possibility we are into an Error path.

```
var assignedEmployee = (clientId, onError, onFinished) => {
  mockClientRepository.getById(clientId,
    error => onError(`error retriving client:${error}`),
    client => {

      if (client) {
        mockEmployeeRepository.getById(client.employeeId,
          error => onError(`error retriving employee:${error}`),
          employee => {
            onFinished(employee)
          });
      }
    });
}

assignedEmployee(1, error => console.log(error), employee => console.log(employee))
```

Run This: [Js Fiddle](#)

this solves the problem but without any complex domain logic and with the simplest error handling, it already starts to look unreadable.

Functional Programming

for Asynchrony

3 Why Functional Programming

Here we are going to take a short look to three basic functional patterns. Those patterns are essential in order to understand the Asynchronous landscape in a profound and unified way.

- 1 **Functors**: solve the problem of applying [map] a function f .
- 2 **Pattern matching**: Extracting values.
- 3 **Monads**: Composing two things that have a similar effect in our case the effect of asynchrony without changing their initial "form".

But first let us briefly see the concept of **Categories** that form the foundation of functional programming.

3.1 Showcase – Bind for Some Async Structures

In the following sections I want to display the amazing ideas of mathematics in a simple intuitive manner that hopefully give you an aha moment. This is all about the **Composition** of the structures that we already seen. The monad is just a multiplication of programming structures.

3.1.1 Multiplying stuff

When we say multiplication, we have in our mind the integers, but we can generalize the concept of multiplication by defining an operation between two "things" that give us as a result one new thing we could write that $\otimes : C \times B \rightarrow C$. In programming there is a special kind of multiplication that we call monoid when we can have items of the same Type \bullet (or Kind) and combine them with some operation \otimes and get something of the same Type $\bullet \otimes \bullet \rightarrow \bullet$

let's see our first weird example immediately because that's the easiest way to understand what I mean here. Let's go to the case of the structure of the callback

if we look at the syntax of our square callback it looks something like this

```
square(2, r => {
});
```

Take a closer look at the Shape of this

```
square(2, r => {
});
```

It looks like an L now if we compose two squares would look like this

```
square(2, r => {
  square(r, r1 => {
    console.log(r1)
  });
});
```

Again if we focus at the shape of the overlapping `square` functions

```
square(2, r => {
  square(r, r1 => {
    console.log(r1)
  });
});
```

We can create a larger function called quadruple that has the same shape (this is takes a callback and return the result at inner parenthesis)

```
var quadruple = (x, resolve) => {
  square(x, r => {
    square(r, r1 => {
      resolve(r1)
    });
  });
};
```

We can imagine that this is a multiplication

```
square(x, r => {
});
```

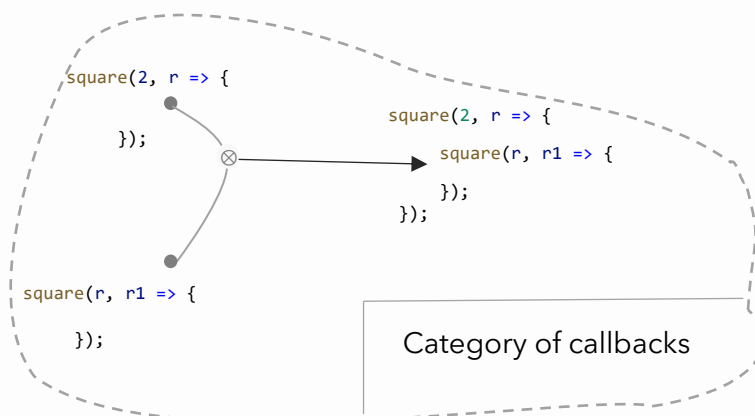
⊗

```
square(x, r => {
});
```

→

```
square(x, r => {
  square(r, r1 => {
  });
});
```

this means that we found a binary “multiplication” operation that when applied to something that belongs to a certain class of objects (aka same category) they give us as a result something of the same category



I know it's weird to think of this combination of syntactical elements as a multiplication, but with enough abstract thinking and imagination you will get used to the idea. This is a monoid. In the language of Category theory, this would be the base to call the callback and the continuation a **Monad**. That we can combine by composition stuff that belongs to the same type (Here the L-shaped callback syntax) and we can get something that is of the same Kind a bigger L callback style structure.

If there was no way to do that in our callback example in a general way, that we would apply to anything with similar form regardless of the naming or argument numbers etc, then we would be forced to treat every case as a separate thing.

Now we can go and write this general helper function `bind`

```
var square = (x, resolve) => {
  setTimeout(() => {
    resolve(x * x)
  }, 1000);
};

var bind = (callbackFunction1, callbackFunction2) => (x, callback) => {
  callbackFunction1(x, r => callbackFunction2(r, callback))
}

bind(square, square)(2, console.log)
```

Run This: [Js Fiddle](#)

and now we can use the `bind` as many times we want

```
bind(bind(square, square), square)(2, console.log)
```

the `bind` is a multiplication between “`square`” functions. You might think of it like writing

```
(square ⊗ (square ⊗ square))
```

Using the bind operation, we can rewrite our client-employee use case example in this way

```
var bind = (callbackFunction1, callbackFunction2) =>
  (x, error, callback3) =>
    callbackFunction1(x, error, r => callbackFunction2(r, error, callback3))

var map = (callbackFunction1) =>
  f =>
    (x, error, callback) =>
      callbackFunction1(x, error, r => callback(f(r)))

var assignedEmployee = bind(map(mockClientRepository.getById)
  (client => client.employeeId), mockEmployeeRepository.getById)

assignedEmployee(1,
  error => console.log("error: " + error),
  employee => console.log(employee))
```

Run This: [Js Fiddle](#)

this is a functional way to compose Callbacks, but it is not useful in practice even if all the newer libraries exhibit a similar syntax.

3.2 Composing Continuations

Let us now see how to compose the continuation form of the square:

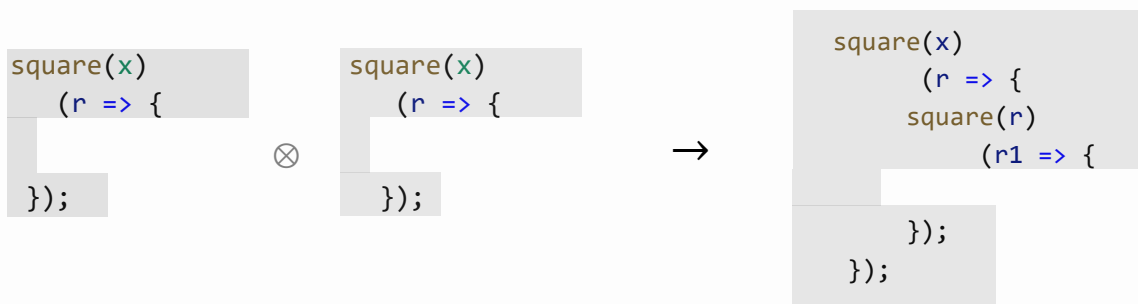
```
var square = (x) => {
  return resolve => {
    setTimeout(() => {
      resolve(x * x)
    }, 1000);
  }
};
```

The Full composition of two squares looks like this

```
square(2)(r => {
  square(r)(r1 => {
    console.log(r1)
  });
});
```

This is pretty much almost the same as the callback composition of the previous section. We couldn't get rid of the nested structure. Still the arrow antipattern make its appearance.

Nonetheless, let's try one more trick. Because of the similarity of the composition to the individual continuation $(a \rightarrow r) \rightarrow r$ [which is the `return resolve => resolve(...)`] we hope that we can compose it in a monadic way. This means that we can hope to create a bind



```
var bind = (cont1, cont2) => {
  return x => resolve => {
    cont1(x)(r1 => cont2(r1)(r2 => resolve(r2)))
  }
}
```

With this definition now we can rewrite the composition like this

```
bind(square, square)(3)(console.log)
```

which is ok but since we already used currying to define the continuation maybe we can use carrying on the definition of the bind to fit better. We can replace `(cont1, cont2) =>` with `(cont1) => (cont2) =>`

```
var bind = (cont1) => (cont2) => {
  return x => resolve => {
    cont1(x)(r1 => cont2(r1)(r2 => resolve(r2)))
  }
}
```

```
bind(square)(square) (3) (console.log)
```

Run This: [Js Fiddle](#)

this would look so beautiful if we would like to compose more continuations

```
bind(square)(bind(square)(square) (3)) (console.log)
```

unfortunately, because of the syntax of the language this set of additional parenthesis ruin everything.

3.2.1 The Continuation monad

If we wanted to rewrite the above derivation of continuation monad in our usual object literal notation the result would be the following:

```
var Continuation = (cont) => ({
  bind: (f) => {
    return Continuation(resolve => {
      cont(res => f(res).run(res2 => resolve(res2)));
    })
  },
  map: (f) => Continuation(resolve => cont(res => resolve(f(res)))),
  run: (alg) => cont(alg)
});

Continuation.of(x)=>Continuation(resolve => resolve(x));
```

This is almost equivalent to a simple promise without the (rejection path and explicit state management) and we can use this in the same way we use a Promise.

```
Continuation.of(5).run(console.log)//5

Continuation(resolve => resolve(5)).map(x => x + 2).run(console.log)//7

Continuation(resolve => resolve(5))
  .bind(x => Continuation(resolve => resolve(x + 2)))
  .run(console.log)//7

//this is equivalent
Continuation.of(5).bind(x=>Continuation.of(x+2)).run(console.log)
```

Run This: [Js Fiddle](#)

Because promises are natively supported in JavaScript there is no reason to use Continuation.

3.3 Folding Callbacks and Continuations

Monoids have this very desirable property that if we keep applying this binary operation, we can always reduce the computation to a single element of the same type:

$$(M \otimes \dots (M (\otimes (M \otimes M))) \rightarrow M$$

this is called folding. We already know folding as **reduce** from the js Array type. We can use the array.reduce in order to execute sequentially Callbacks. We can do that with any Monad as well since monads are monoids [we will see an example of folding Either monads for validation purposes in the end of the Promises chapter]

if you are familiar with the array.reduce you can verify that if we have an array of Callbacks like this [square, square, square] we can use reduce to fold it into a single callback

```
var fold = callbackArray => callbackArray
    .reduce((accumulation, callback) => bind(accumulation, callback), id)
```

where the bind was defined earlier

```
var bind = (callbackFunction1, callbackFunction2) => (x, callback) =>
    callbackFunction1(x, r => callbackFunction2(r, callback))
```

And the id is the identity element of the "callback" monoid. The only property of the id is that id we use bind on a callback and the id it gives us back the callback. an approximation for this given our bind is `var id = (x, callback) => callback(x)`. Anyway, this boils down to:

```
var fold = callbackArray => callbackArray.reduce(bind, (x, callback) => callback(x))
```

you can test it here

```
fold([square, square, square])(3, console.log)
```

Run This: [Js Fiddle](#)

the same hold for the continuations. This is equivalent to the [async.series](#) function of the async.js library.

Other Async Libraries

4 Fluture.js

The tagline of the Fluture on the github repository is: "Fantasy Land compliant (monadic) alternative to Promises" we already saw that `Promise.then` is actually implemented in such a way that we can call it a monadic but this fact remains hidden in the semantic overload of the `then`. The following two Libraries that we are going to see

In the discussion in the Fluture there is a topic regarding the [Comparison between the Futures and the Promises](#)


On the surface Futures are just like Promises, but with the different behaviors of the `.then` method extracted into three distinct functions, each with *a single responsibility*.

- [map](#): Transforms the success value inside the Future. This happens in Promise if you return anything that doesn't look like a Promise from `f` in `.then(f)`.
- [chain](#): Absorb the state of another Future into the main Future (flat map). This happens in Promise if you return something that looks like a Promise from `f` in `.then(f)`.
- [fork](#): Evaluates the computation using the given continuations. Promise are automatically evaluated (more on that [below](#)).

We can create a new Future in the same way that we Create a Promise.

```
Future((reject, resolve) => {
  var t = setTimeout(() => {
    resolve(5)
  }, 1000);
  return () => clearTimeout(t)
})
```

Returning a Cancellation
function that can be used
latter



The only difference is that it also returns a function which is used for cancellation and clearing any resources.

We can also create a Future out of a Promise using the [attemptP](#)

```
attemptP (() => Promise.resolve (4))
```

The basic operation in order to chain operations for Future is the pipe, for example using the map operation on a Future we should just write

```
attemptP (() => Promise.resolve (4))
  .pipe(map(x=>x+3))
```

For pattern matching the value or Error of the future we pipe the fork operation

```
attemptP (() => Promise.resolve (4))
  .pipe(map(x=>x+3))
  .pipe(fork
    (x => console.log(x + ' rejection'))
    (x => console.log(x + ' resolution')))
```

The Future library also allow us to use the Curried version of fork where we prepend the operation

```
fork
  (x => console.log(x + ' rejection'))
  (x => console.log(x + ' resolution'))
  (attemptP (() => Promise.resolve(5))
    .pipe(map(x => x + 5))
  )
```

This is the curried version where we apply the fork on the Future. If you are not used to use Curried functions, this looks kind of weird and you might prefer to use the classical notation using `.pipe(fork)`.

Finally, in order to monadically bind two futures we use the chain operations:

```
var firstFuture = Future((reject, resolve) => {
  const t = setTimeout(resolve, 20, 2)
  return () => clearTimeout(t)
});

var secondFuturef = x => Future((reject, resolve) => {
  const t = setTimeout(resolve, 20, x + 3)
  return () => clearTimeout(t)
})

firstFuture
  .pipe(chain(secondFuturef))
  .pipe(fork
    (x => console.log(x + ' rejection'))
    (x => console.log(x + ' resolution')))
```

Now we can rewrite our classical client-Employee use case example using Futures

```

var mockClientRepository = ({
  getById: (id) =>
    Future((reject, resolve) => {
      setTimeout(() => {

        var client =
          [{ id: 1, name: 'rick', age: 29, employeeId: 1 },
           { id: 2, name: 'morty', age: 25, employeeId: 3 }]
          .find(client => client.id === id);

        if (client)
          resolve(client)
        else
          reject("no client found")

      }, 1000)
      return () => { }
    })
})

var mockEmployeeRepository = ({
  getById: (id) =>
    Future((reject, resolve) => {
      setTimeout(() => {
        var employee = [{ id: 1, name: 'jim', age: 29 },
                        { id: 2, name: 'jane', age: 25 }]
        .find(employee => employee.id === id);

        if (employee)
          resolve(employee)
        else
          reject("no employee found")

      }, 1000)
      return () => { }
    })
})

var assignedEmployee = (clientId) =>
  mockClientRepository.getById(clientId)
    .pipe(map(client => client.employeeId))
    .pipe(chain(mockEmployeeRepository.getById))

assignedEmployee(1)
  .pipe(fork
    (x => console.log('error:' + x))
    (x => console.log(x.name)))

```

Web Workers

for parallelism

The JavaScript Engine is Single Threaded which might be limiting if we want to use multiple threads to delegate work, this was made possible with the **web workers**. In the case of the client-side JavaScript, the need for multiple threads is usually not critical, because complex tasks can be delegated to the server side and then have the results delivered asynchronously whenever the computation is completed. But for node.js the need for multiple thread support is essential. JavaScript engines like the chrome's v8 used in Chrome and in Node.js. In this section we are going to take a look at the web workers from the client-side perspective through the web workers API. Nonetheless node.js Worker Threads API is pretty much similar.

The way to use workers is very easy to understand. The main thread used by the UI can create a web worker by

```
var worker = new Worker('worker.js');
```

the main thread communicates with the worker by sending messages. the worker exposes a postMessage method that we can use to send data to the worker

```
worker.postMessage(5);
```

The web worker receives the message by subscribing to a onmessage method

```
onmessage = function(event) { ... };
```

Where the event has a data property event.data that contains the data send by the main thread. The worker then can do some work and finally send back data to the main thread with a postMessage method of its own. The data are transferred at the main thread by having an event handler on the worker on the onmessage event it exposes.

Main.js

```
var worker = new Worker(worker.js');

worker.onmessage = function(event) {
  document.getElementById('result').textContent = event.data;
};

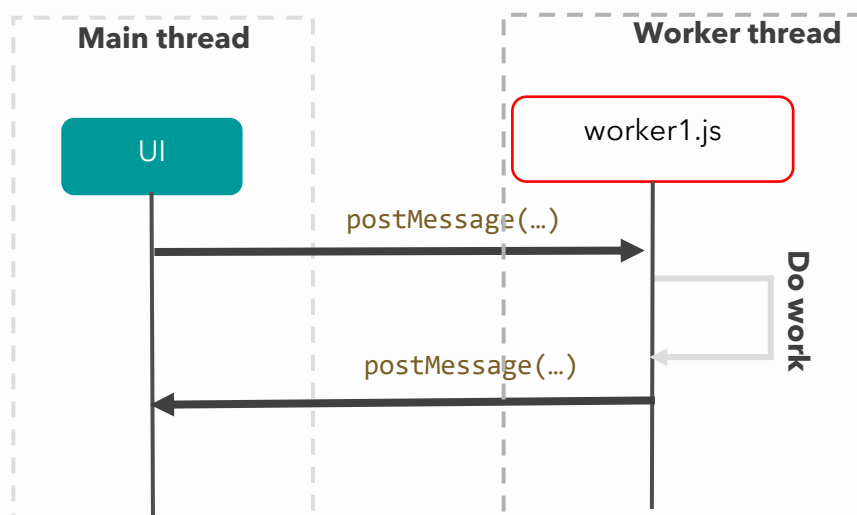
worker.postMessage(5);
```

Run This: [Js Fiddle](#)

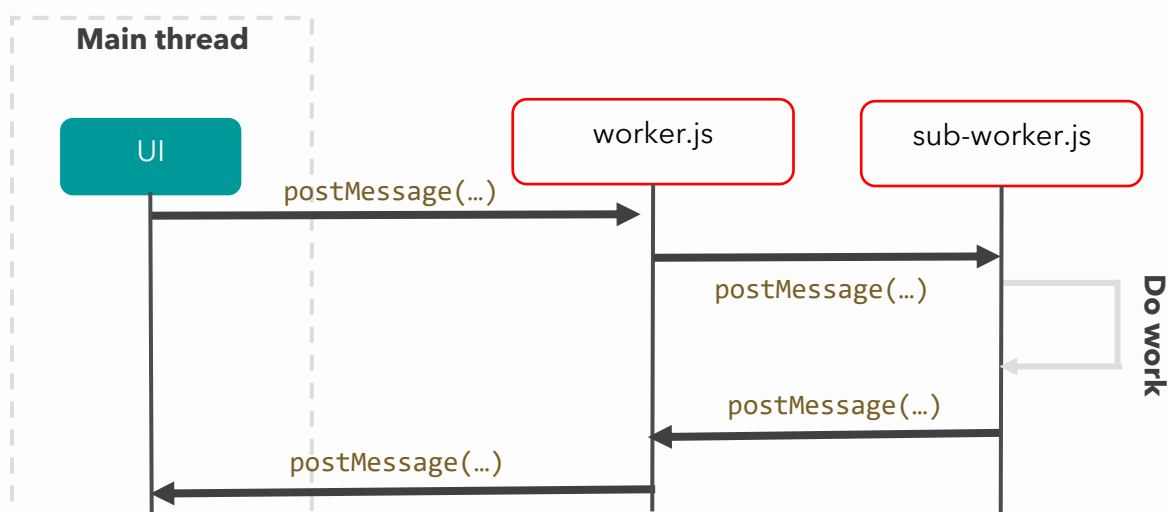
worker.js

```
onmessage = function(e) {
  postMessage(e.data + 5);
};
```

Run This: [Js Fiddle](#)



The workers can also create sub workers and delegate task to them. The same message passing mechanics apply to a worker-sub worker communication using `postMessage` and `onmessage` you can play around with the [Js Fiddle](#)



Run This: [Js Fiddle](#)

4.1.1 Error handling

In the case of Errors, the error in whatever worker file occurs it is pushed in the main thread at the `worker.onerror` event handler

```
var worker = new Worker('lib/worker.js');

worker.onmessage = function(event) {
  log(`${event.data}`);
};

worker.onerror = function(error) {
  log(`error: ${error.message}`);
};
```

Run This: [Js Fiddle](#)

You can try to some cases by raising errors at the sub-worker thread here [jsFiddle](#). You can see that if it is not handled and passed in a controlled manner it directly reaches the error handler of the parent worker.

Async-Await

5 Async-Await

This section is based on the understanding of promises. Async/await is just additional syntactic support for promises. It's surprisingly easy to understand and use. The MDN Documentation Describes the `async/await` :

"These features [`async/await`] basically act as syntactic sugar on top of promises, making asynchronous code easier to write and to read afterwards."

[We will also explore the the concepts of co-routines and generators that form the foundations of the `async/await`. For simplicity this is left out at the first installment but will be added in the next release.]

So, let's start from the Async functions first.

5.1 Async

The `async` keyword is placed before the `function` keyword and makes the function an **asynchronous function**. The async function **returns an Promise** as a result. If we write this function below and call it the result is not `{ id: 1, name: "rick" }` but instead `Promise.resolve({ id: 1, name: "rick" })`

```
async function getClient() {
  return { id: 1, name: "rick" }
}
```

We can use `.then` because the result is wrapped into a promise

```
getClient().then(console.log)
```

Run This: [Js Fiddle](#)

and because of the monadic bind property of the `Promise.then` you expect that if we return a promise the result would be the same because the `Promise` would be flatmapped . That's why the following gives us the same result:

```
async function getClient() {
  return Promise.resolve({ id: 1, name: "rick" })
}
```


And we can use the **async with the arrow notation** as well

```
var getClient = async () => { id: 1, name: "rick" }
```

Now the other important thing about `async` is that **we can only use the await keyword inside an async function**. And that brings us to the `await` Keyword.

5.2 The await keyword

The `await` can be put in front of any `Promise` to pause your code on that line until the promise settles

1. If the Promise is fulfilled, `await` returns the fulfillment value.
2. If the Promise is rejected, `await` throws the rejection value.

Look at the following code here I just added the `await` in front of the `Promise.resolve`

```
var getClient = async () => {
  var client = await Promise.resolve({ id: 1, name: "rick" })
  return client
}

getClient().then(console.log)
```

Run This: [Js Fiddle](#)

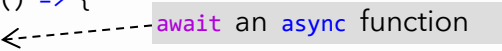
when the engine executes this code, it stops at the `await` and waits for the `Promise` to return something (or throw an exception). Notice that because the `await` is inside the `async` function the overall execution is not blocked. The `getClient()` "is a Promise" so it doesn't block the engine.

Obviously because the `async function` return a promise we can use the `await` before calls to `async` functions as well. This is the most usual case.

```
var mockFetch = async () => ({ id: 1, name: "rick" })

var getClient = async () => {
  var client = await mockFetch()
  return client;
}

getClient().then(console.log)
```



Run This: [Js Fiddle](#)