

PROMISES

in JavaScript

Part 1

The Promise object is used for asynchronous computations. A Promise represents an operation that hasn't completed yet, but is expected in the future.

Promises provide a few **advantages** over callback objects:

- Functional composition and error handling.
- Prevents callback hell and provides callback aggregation.

In plain words

Promise is a placeholder for an asynchronous operation that is ongoing.

Wikipedia says

In computer science, future, promise, delay, and deferred refer to constructs used for synchronizing program execution in some concurrent programming languages. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is not yet complete.

Function Declaration Syntax

Promises make heavy use of functions. When reading documentation on them you may come across different techniques for declaring them.

You can have named functions.

```
function fname (par1, par2)
{
    return par1 + par2;
}
```

You can also declare anonymous functions which return a reference to the function you just declared. This reference can be passed to other functions as a parameter or stored in a variable.

```
function(par1, par2) { return par1 + par2; }
```


You can also use the arrow notation

```
(par1, par2) => { return par1 + par2; }
```

Asynchronous Functions

Asynchronous computation is an important part of modern scripting.

Usually code is synchronous. I.e. code is executed in the order it appears. One bit of code must complete before the next bit of code can start.

However, sometimes you may not know how long some code will take to complete, or it just takes too much time, and you will want the rest of your code to carry on with other work instead of waiting for the that code to finish.

Examples of asynchronous computation:

Event listeners

Loading images/media

Ajax/XMLHttpRequest communications

setTimeout function calls

etc.

Example

```
console.log("Starting")
```

```
var x = new XMLHttpRequest();  
x.open("GET", "data.php");
```

```
x.onreadystatechange = handleIncomingData;
```

```
x.send(null);
```

```
console.log("Carrying on");
```

This function will be executed when the asynchronous code is finished and data returned from the internet request.

Asynchronous
function.

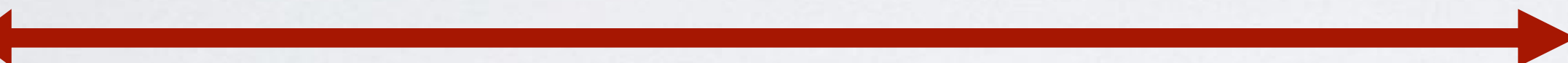
This code doesn't wait for data to come back from the internet request that **x.send(null)** started.

An issue of particular importance is trying to receive data back from the asynchronous code.

A command like the one below can't work because the data from **someAsyncFunction()** (which we assume is asynchronous) won't be ready in time to assign it to x. I.e. **someAsyncFunction()** is not a blocking function.

```
var x = someAsyncFunction();  
  
console.log(x);
```

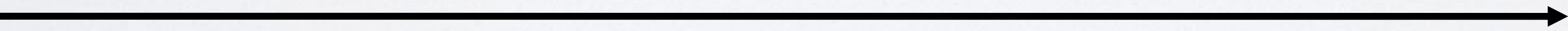

Script starts



someAsyncFunction()
finishes and data is
ready



Assignment tries to
copy data before it is
ready.



Time

Frequently we want some code (e.g. a function) to execute once the asynchronous activity is finished (and **only** when its finished).

E.g. if we retrieve data from a web service using an **XMLHttpRequest** object we cannot not try to use the data until after it has come back from the remote server. and we will have no way of knowing how long this will take.

We usually do this with a **callback**.

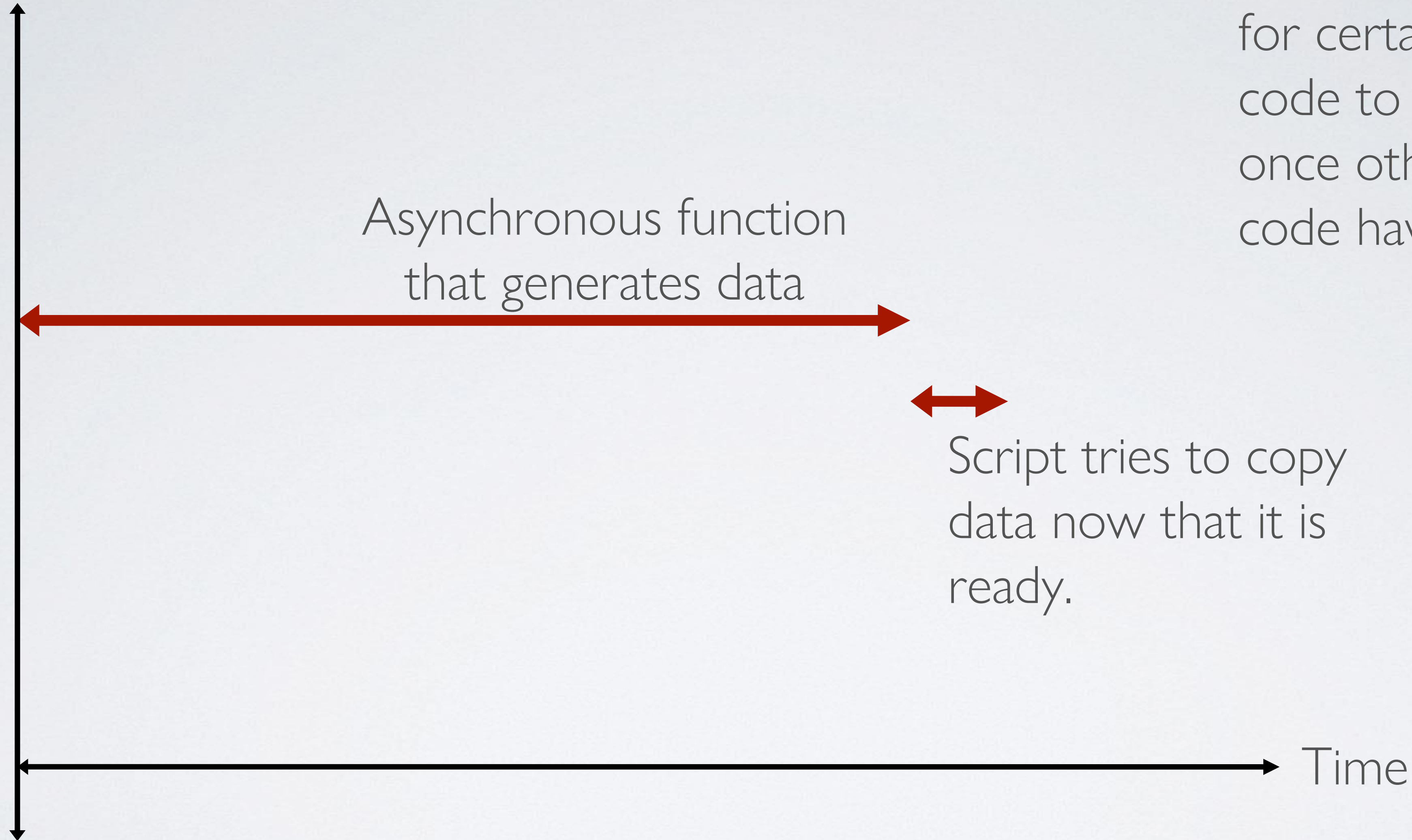
Script starts

Asynchronous function
that generates data

I.e. we need to arrange
for certain parts of our
code to execute only
once other parts of our
code have finished.




Script tries to copy
data now that it is
ready.



```
function getData() {  
  
    var x = new XMLHttpRequest();  
    x.open ("GET", "data.php");  
  
    x.onreadystatechange = function ()  
    {  
        if (x.readyState == 4 && x.status == 200)  
        {  
            incomingData = x.responseText;  
  
            <We want something to happen here  
now that the data has been received>  
        }  
    };  
  
    x.send(null);  
}
```

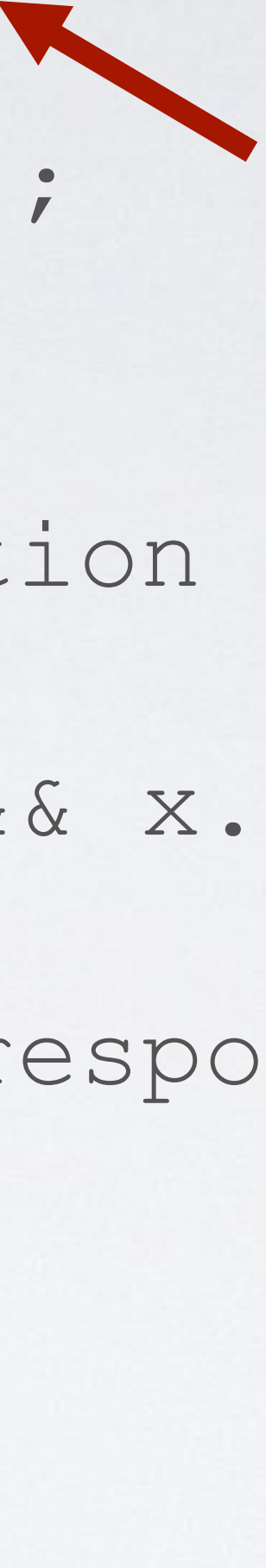


```
function getData() {  
  
    var x = new XMLHttpRequest();  
    x.open ("GET", "data.php");  
  
    x.onreadystatechange = function ()  
    {  
        if (x.readyState == 4 && x.status == 200)  
        {  
            incomingData = x.responseText;  
  
            doSomething();  
        }  
    };  
  
    x.send(null);  
}
```



We can call some function
when the data is ready.

```
function getData(someFunction) {  
    var x = new XMLHttpRequest();  
    x.open ("GET", "data.php");  
  
    x.onreadystatechange = function ()  
    {  
        if (x.readyState == 4 && x.status == 200)  
        {  
            incomingData = x.responseText;  
  
            someFunction() ;  
        }  
    } ;  
  
    x.send(null) ;  
}
```



We can make our code more flexible by passing a function reference to the asynchronous function so that it can be invoked when the asynchronous function finishes.


```
function doSomething()
```

```
{  
    alert("The first bit of data has been received");  
}
```

```
function doSomethingElse()
```

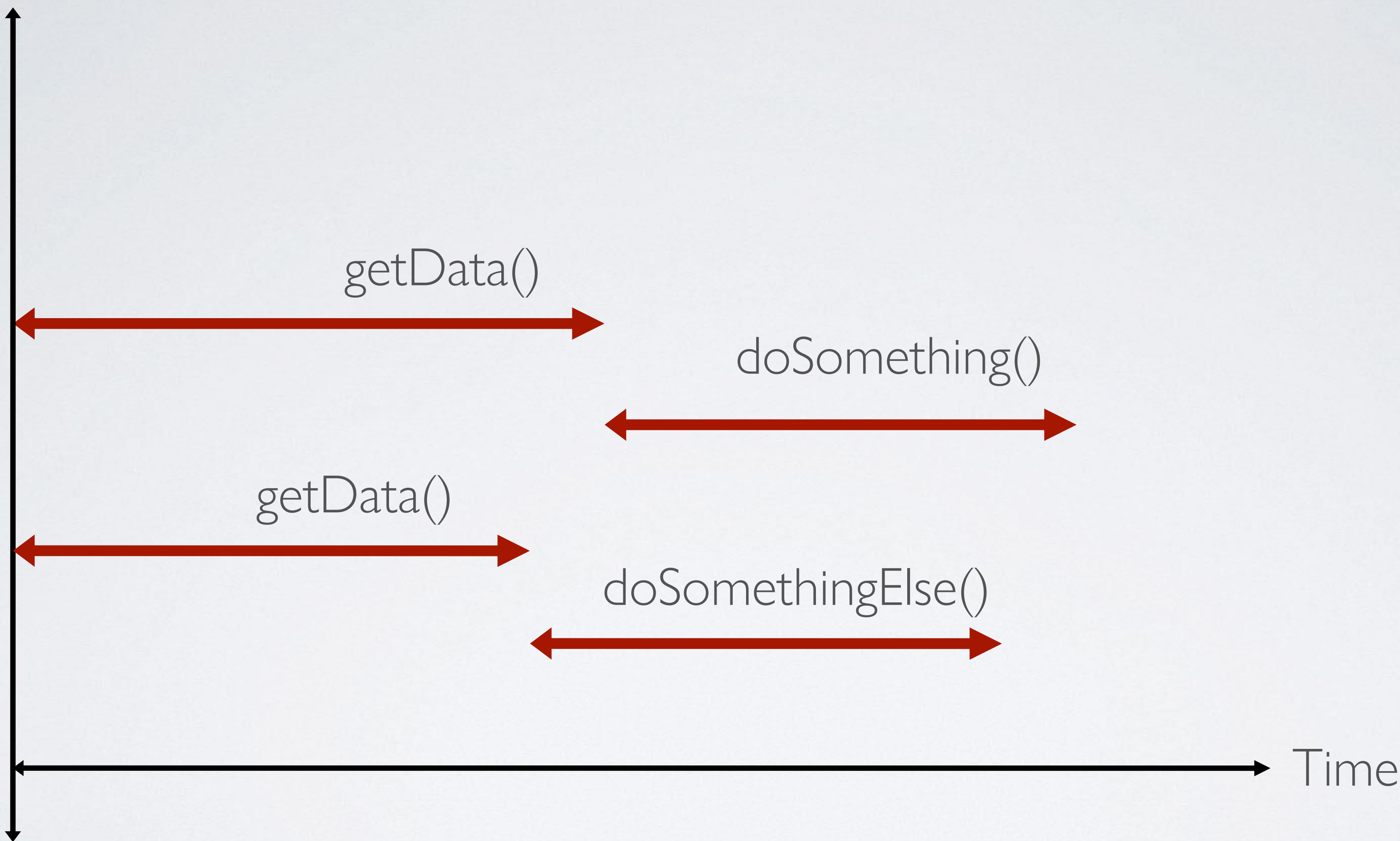
```
{  
    console.log("The second bit of data has been received");  
}
```

```
getData(doSomething);
```

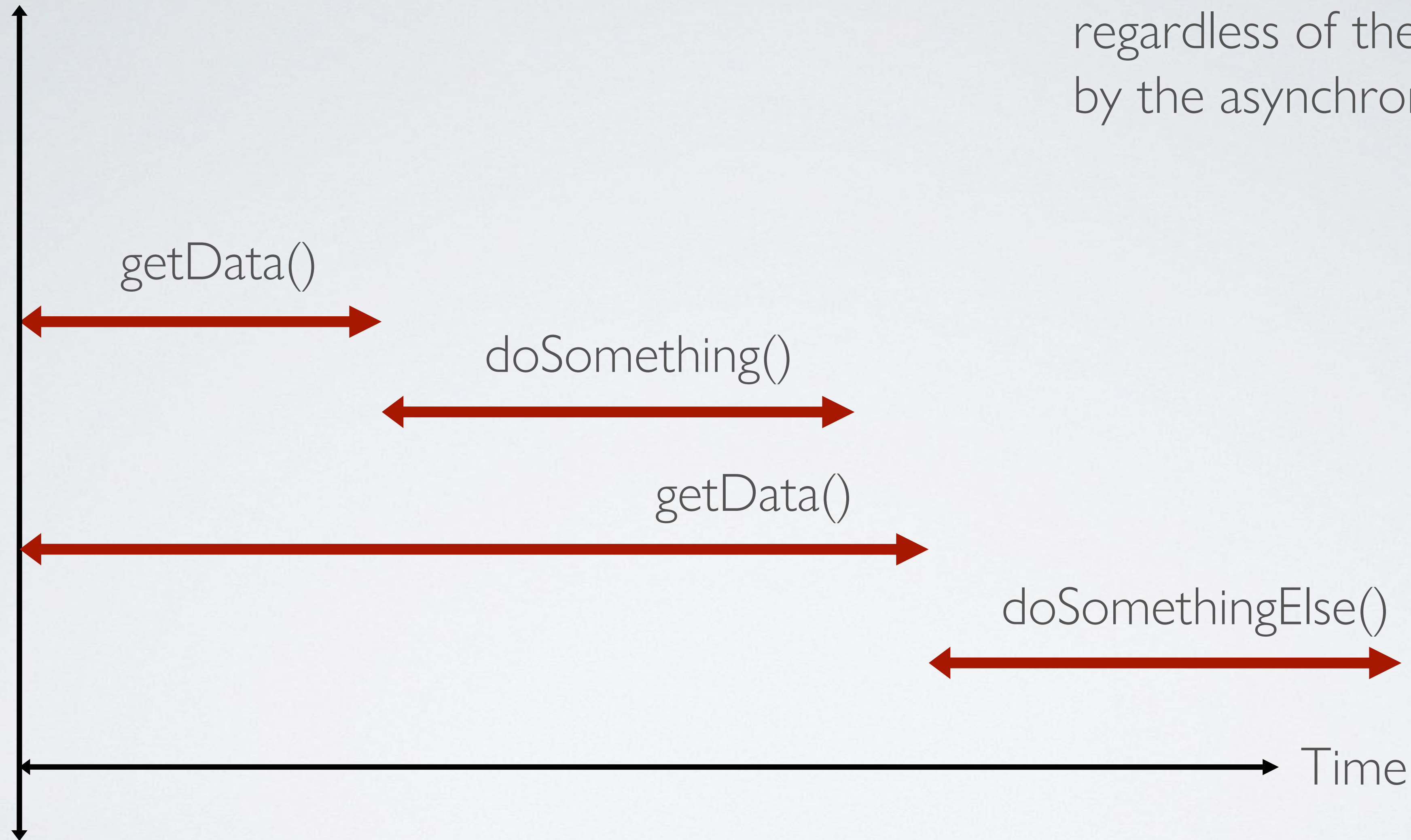
```
getData(doSomethingElse);
```

We can now arrange for any function to be executed when **getData()** is ready.

Script starts

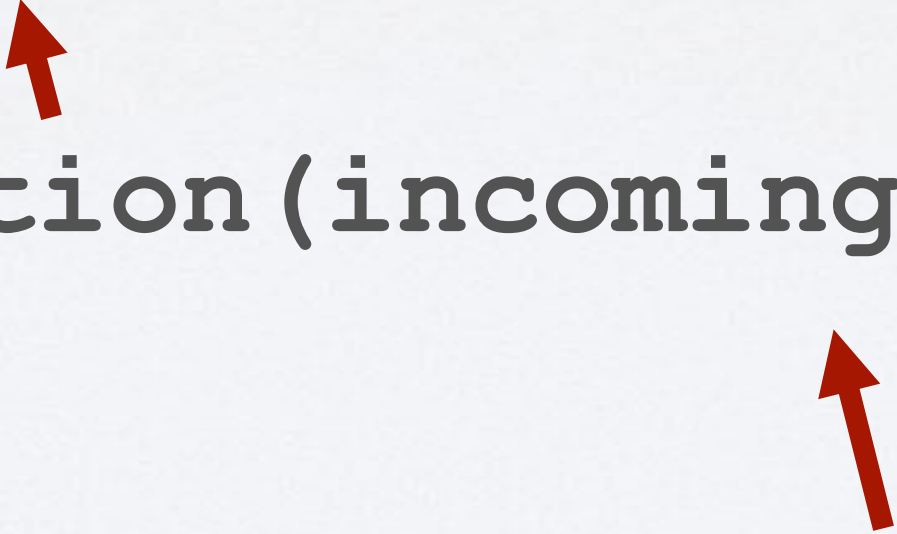


Script starts



This technique will work regardless of the time taken by the asynchronous actions.

```
function getData(someFunction) {  
  
    var x = new XMLHttpRequest();  
    x.open ("GET", "data.php");  
  
    x.onreadystatechange = function ()  
    {  
        if (x.readyState == 4 && x.status == 200)  
        {  
            incomingData = x.responseText;  
            someFunction(incomingData) ;  
        }  
    };  
  
    x.send(null);  
}
```



We can also
send data to the
callback
function.


```
function doSomething(data)
{
    alert("Data has been received: " + data);
}
```

```
function doSomethingElse(data)
{
    console.log("Data has been received");
    console.log(data);
}
```

```
getData(doSomething);
```

```
getData(doSomethingElse);
```

Callbacks and the Pyramid of Doom

Assume we have a simple function that takes some time to finish. We will simulate this with a timeout. We use a callback parameter that can be called when the function is finished.

```
function step1 (callback)  
{  
    console.log ("Starting Step 1");  
  
    setTimeout ( callback , 1000 )  
}
```

Now assume we have another function we want to call when the first function is finished

```
function step2()  
{  
  console.log ("Starting Step 2");  
}
```

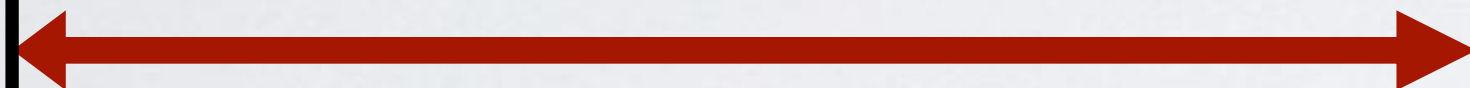

To have the **step2** function execute only after the **step1** function is completed, we can simply do this:

```
step1 (step2) ;
```

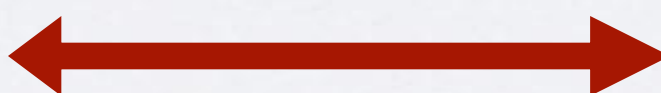
Script starts



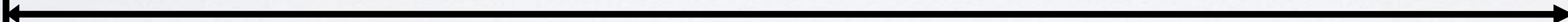
step1()



step2()



Time




```
function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}
```

```
function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}
```

```
function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}
```

```
function step4( )
{
  console.log ("Starting Step 4");
}
```

Now assume we have four functions.

We want them to execute in the following order.

step1

step2

step3

step4

Each function should only start executing when the previous one is finished.

```
function step1(callback)
{
console.log ("Starting Step 1");

setTimeout( callback , 1000);
}

function step2(callback)
{
console.log ("Starting Step 2");

setTimeout( callback , 1000);
}

function step3(callback)
{
console.log ("Starting Step 3");

setTimeout( callback , 1000);
}

function step4( )
{
console.log ("Starting Step 4");
}
```

The output in the console should be:

Starting Step 1
Starting Step 2
Starting Step 3
Starting Step 4

(And it should take at least 3 seconds to finish outputting all the messages).


```
function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}
```

```
function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}
```

```
function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}
```

```
function step4( )
{
  console.log ("Starting Step 4");
}
```

Based on our previous attempt you might try the following:

```
step1 (step2 (step3 (step4) ) ) )
```

However, the log records the following:

Starting Step 3
Starting Step 2
Starting Step 1
Starting Step 4

However, you need to realise that the arguments for a function must be evaluated before you can execute the function.


```
function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}
```

```
function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}
```

```
function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}
```

```
function step4( )
{
  console.log ("Starting Step 4");
}
```

This must be executed first in order to be able to pass the result to step2.



```
step1 (step2 (step3 (step4) ) ) )
```

Starting Step 3

```
function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}
```

```
function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}
```

```
function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}
```

```
function step4( )
{
  console.log ("Starting Step 4");
}
```

**This is next in order to be able
to pass the result to step1.**



```
step1 (step2 (step3 (step4) ) ) )
```

Starting Step 3
Starting Step 2


```
function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}
```

```
function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}
```

```
function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}
```

```
function step4( )
{
  console.log ("Starting Step 4");
}
```

**This is next now its parameter
has been resolved.**



step1 (step2 (step3 (step4))))

Starting Step 3
Starting Step 2
Starting Step 1

```
function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}
```

```
function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}
```

```
function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}
```

```
function step4( )
{
  console.log ("Starting Step 4");
}
```

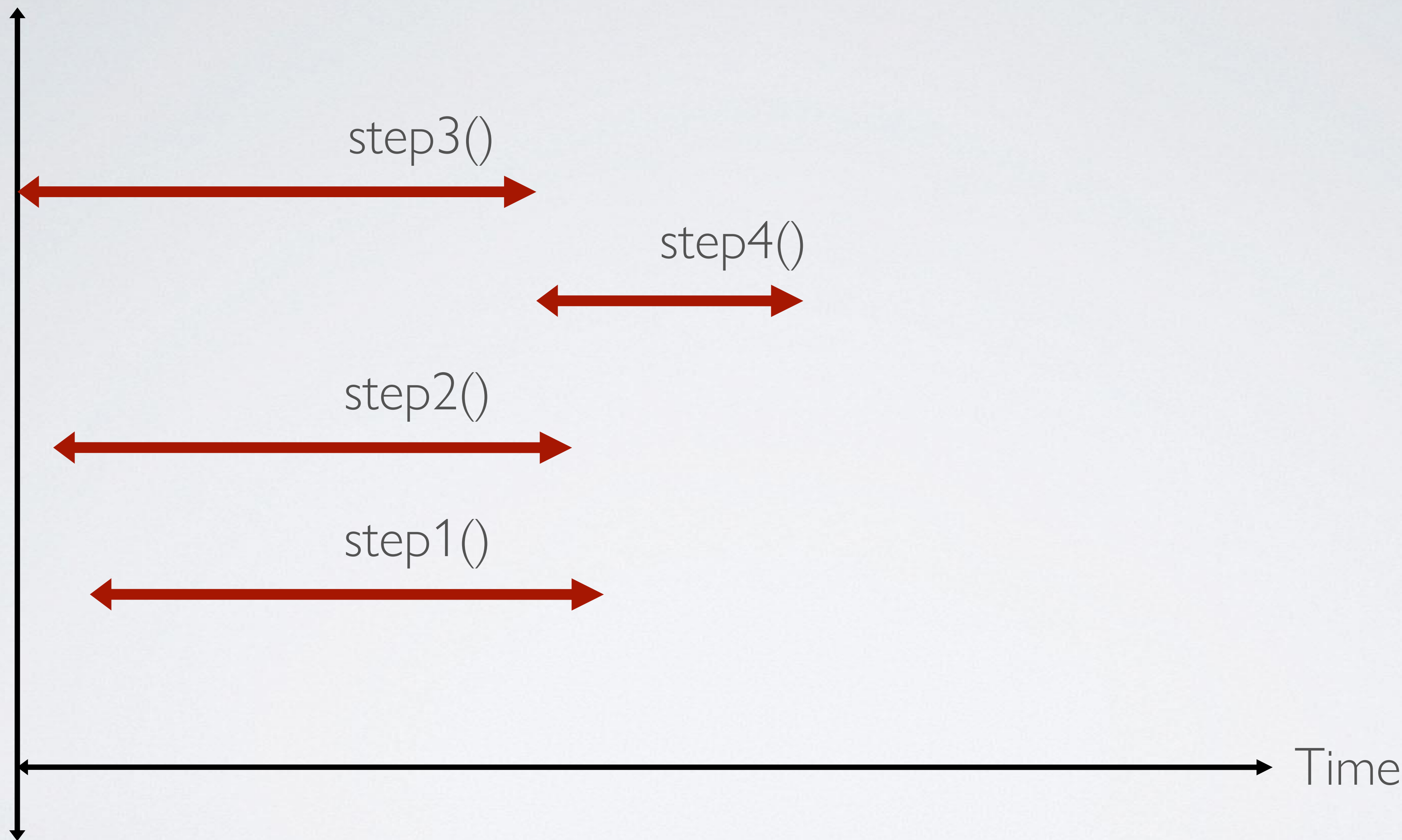
Finally, this executes 1000ms later as the call back of the **step3()** function. This is the only callback that executes *as a callback*.



```
step1 (step2 (step3 (step4) ) ) )
```

Starting Step 3
Starting Step 2
Starting Step 1
Starting Step 4

Script starts



```
function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}
```

```
function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}
```

```
function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}
```

```
function step4( )
{
  console.log ("Starting Step 4");
}
```

step1 (step2 (step3 (step4))))

The problem here is that in order to pass callbacks to **step2** and **step3** we force them to execute immediately, so they are executed at the same time as step1.

We need to be able to pass a callback without executing it first.


```

function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}

function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}

function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}

function step4( )
{
  console.log ("Starting Step 4");
}

```

We can use anonymous functions to solve this problem. We pass an anonymous function as the callback, which instead can call the function that we originally wanted as the callback.

E.g. We can call **step2** inside the anonymous callback function for **step1**.

```

step1 (
    function ()
    { step2 (); }
)

```



```

function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}

function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}

function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}

function step4( )
{
  console.log ("Starting Step 4");
}

```

This means that **step2()** isn't called when we pass as an argument to **step1()**. Instead it is called whenever the anonymous function is invoked (as the callback for **step1**).

```

step1 (
    function ()
    { step2 () ; }
)

```



```

function step1(callback)
{
  console.log ("Starting Step 1");

  setTimeout( callback , 1000);
}

function step2(callback)
{
  console.log ("Starting Step 2");

  setTimeout( callback , 1000);
}

function step3(callback)
{
  console.log ("Starting Step 3");

  setTimeout( callback , 1000);
}

function step4( )
{
  console.log ("Starting Step 4");
}

```

We can now also add **step3** as the callback to **step2** inside an anonymous function of its own.

```

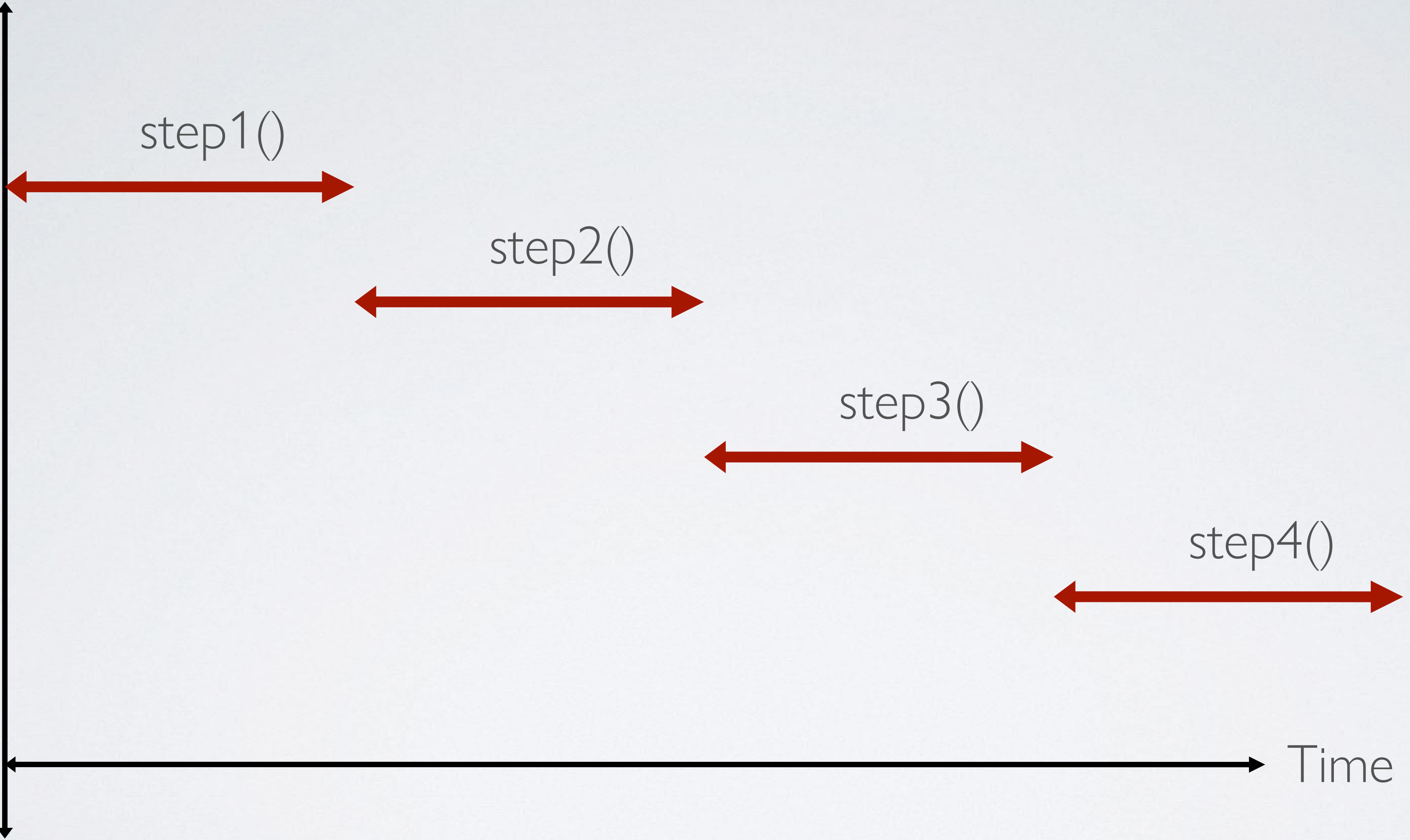
step1(function() {
  step2 (function() {
    step3();
  })
})

```

In order to keep our code legible we can indent it as follows.

```
step1 (function()  
  {  
    step2 (function()  
      {  
        step3 (function()  
          {  
            step4 ();  
          })  
        })  
      })  
    })  
  }) ;
```


Script starts



The code could also be written without named functions.

```
function() {
  console.log ("Starting Step 1");

  setTimeout( function() {

    console.log ("Starting Step 2");

    setTimeout( function() {

      console.log ("Starting Step 3");

      setTimeout( function() {

        console.log ("Starting Step 4");

      }

      , 1000);

    }

    , 1000);

  }

  , 1000);

} ();
```



```
function() {  
  console.log ("Starting Step 1");  
  
  setTimeout( function() {  
  
    console.log ("Starting Step 2");  
  
    setTimeout( function() {  
  
      console.log ("Starting Step 3");  
  
      setTimeout( function() {  
  
        console.log ("Starting Step 4");  
  
      }  
      , 1000);  
    }  
    , 1000);  
  }  
  , 1000);  
} ();
```

It can get quiet hard to follow,
and the resulting code is
sometimes known as the
pyramid of doom.

Promises

As could be seen, **callbacks** can be a complicated solution to what seems like a simple problem: how to get specific code to execute after asynchronous other code has finished.

JavaScript **promises** are another solution that allows us to dictate when code executes relative to other code, but without the possible confusion of nested callbacks.

A **promise** is an object that represents the result of an asynchronous action.

I.e. it represents data now that may not exist *yet*.

To create a promise you pass it a ***executor*** function.

This is a **function** containing your asynchronous code **and has 2 parameters**. These parameters represent callback functions. The promise will automatically pass its own callback functions as arguments to this ***executor*** function.

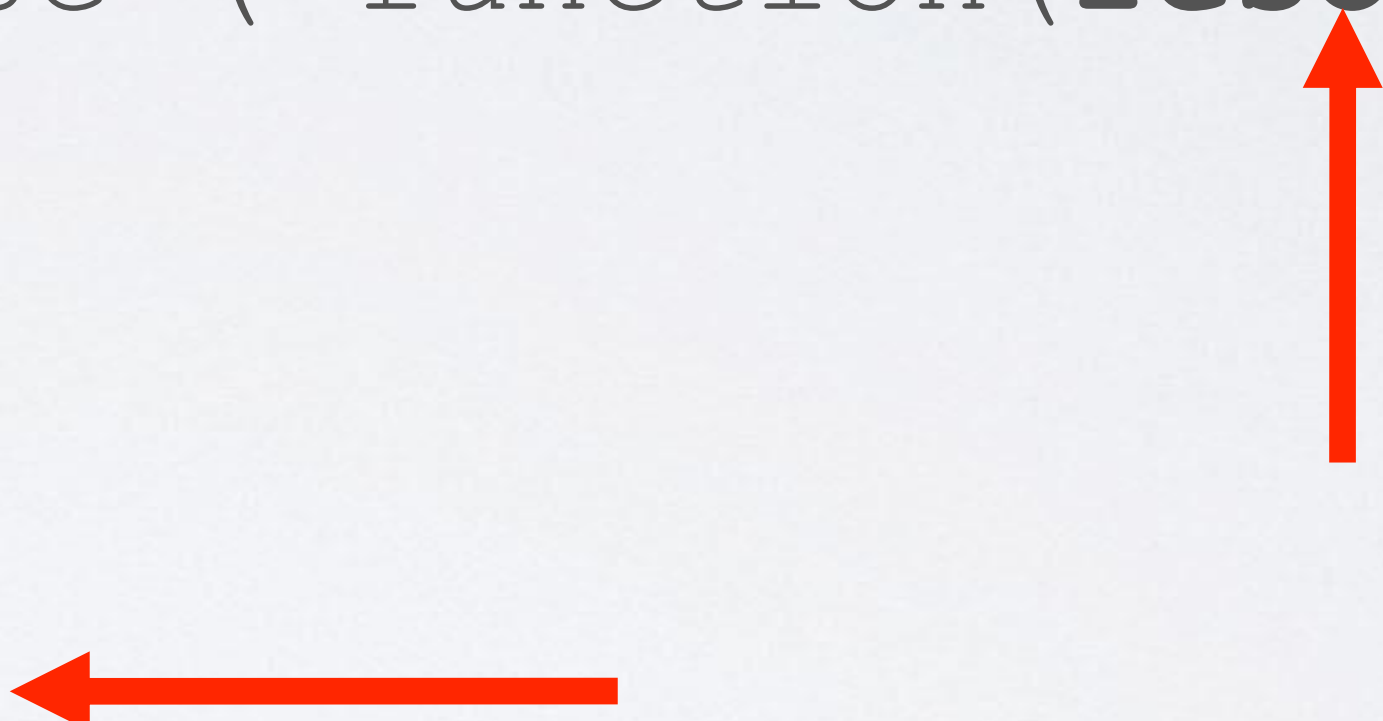
One of these arguments is to be called if your code terminates successfully, the other if there was an error.

```
var promise1 = new Promise ( function(resolve, reject) {  
  
    <your asynchronous code goes here>  
  
    }) ;
```


Resolving a Promise

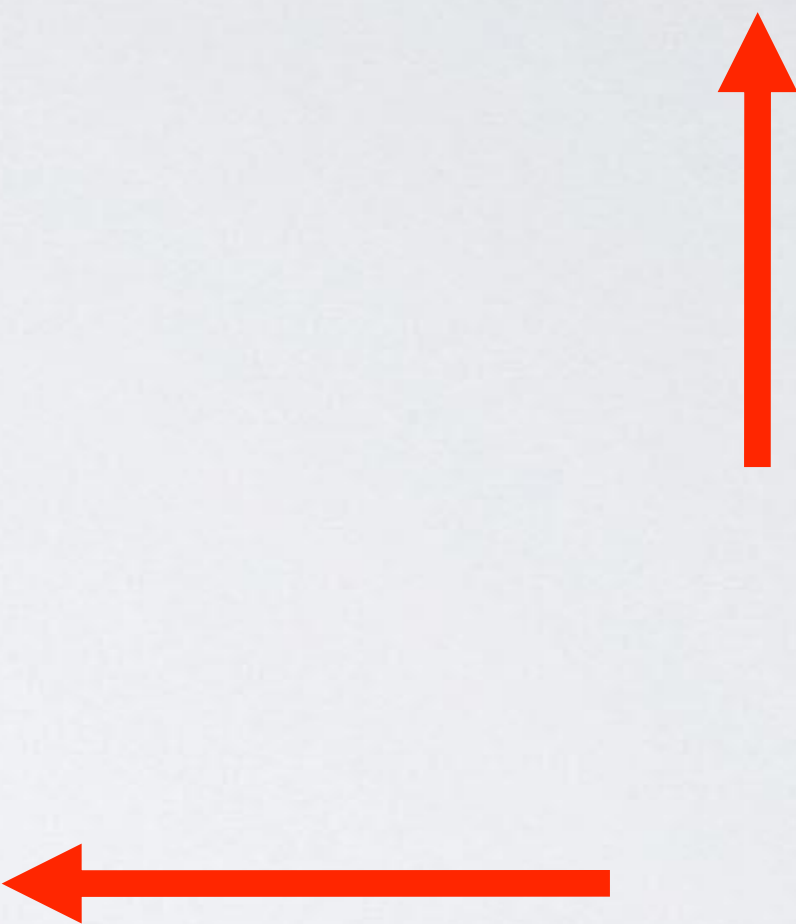
When your code is finished it should call the function from the first parameter. You can pass data to this function so it can pass it on to other parts of your code.

```
var promise1 = new Promise ( function(resolve, reject) {  
  
    setTimeout (  
        () => {  
            var x = 5;  
            resolve(x) ;  
        },  
        1000 )  
  
    } ) ;
```

Two red arrows are present. One arrow points vertically upwards from the bottom towards the **resolve** parameter in the function signature of the Promise constructor. The other arrow points horizontally from the right towards the **resolve(x)** call inside the setTimeout callback.

Rejecting a Promise

```
var promise1 = new Promise ( function(resolve, reject) {  
  
    setTimeout (  
        () => {  
            var x = Math.random();  
  
            if (x < 0)  
                reject("negative value");  
            else  
                resolve(x);  
        },  
        1000 )  
    } ) ;
```



If there is a problem with your code you can call the second parameter function. This will allow your code to react to errors, etc.

Our **step1** function from before can be rewritten to use a promise.

(Note that if you create a promise it executes the code in the executor function immediately.)

```
var step1 = new Promise(function(resolve, reject)
{
    console.log ("Starting Step 1");

    setTimeout( resolve , 1000)
});
```

.then()

If you have a Promise object you can easily tell JavaScript you want to execute some other code when it is finished.

A Promise has 3 states.

fulfilled

Final value is received (including *undefined*). I.e. it has been *resolved*.

rejected

An error occurred. I.e. it has been *rejected*.

pending

Final value is not yet available (i.e. it hasn't been *resolved* or *rejected*).

If the Promise has been fulfilled you can direct JavaScript to execute some code.

You do this by using the Promise's **then()** function, passing it a function you want to execute when the Promise is fulfilled (you can specify a function to execute if the Promise is resolved, and another for if it is rejected).

The functions you pass to **then()** are known as **handler functions**.

In this example we resolve the promise after a timeout of 10 seconds.

The code in the executor function starts executing when you create the Promise object. **When the Promise is resolved** the handler function of the **then()** function is **executed**.

```
var promise1 = new Promise(function(resolve, reject)
                                { setTimeout(resolve, 10000); });

function finished()
{
    console.log("finished");
}

promise1.then(finished);
```

Script starts



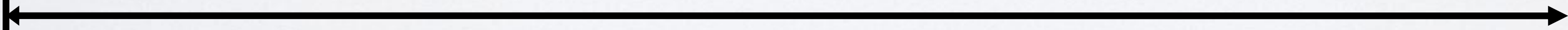
promise1() (+ executor function)



finished()



Time



Promises and their **then()** function provide a simpler way for us to add callback functions to asynchronous code but with more flexibility (and fewer *Pyramids of Doom*)

You can use *anonymous*, *arrow*, and *named* functions with **then()**

```
promise1.then( function ()  
                { console.log(" finished"); } );
```

```
promise1.then( () => {console.log("finished"); } );
```

```
function done() {console.log("finished"); }
```

```
promise1.then(done) ;
```



We can also **pass** information from the **executor function** to a **handler function**.

You do this through the **resolve()** (or **reject()**) functions.

In this example we pass some text to the **resolve** function.

```
var step1 = new Promise(function(resolve, reject)
{
    console.log ("Starting Step 1");


    setTimeout( function() {resolve("Finished step 1");} , 1000)
});
```



Note that **setTimeout** has an alternative syntax for passing parameters to the function parameter.

```
var step1 = new Promise(function(resolve, reject)
{
    console.log ("Starting Step 1");

    setTimeout( resolve , 1000, "Finished step 1")
});
```



The **then()** function is passed a handler function. If that function has a parameter then the data passed to the **resolve()** function in the Promise will be passed on to this parameter.

Remember, invoking the **resolve** function causes the function passed to **then()** to execute.


```
var step1 = new Promise(function(resolve, reject)
{
    console.log ("Starting Step 1");

    setTimeout( function() {resolve ("Finished step 1");} , 1000)
});
```

```
step1.then( function (data)
               { console.log(data); })
```

Console output: **"Finished step 1"**

Regular JavaScript event handlers only catch events that are triggered ***after*** they are attached to an object. E.g. if you add an **onload** event handler to an image ***after*** it loaded then it then the event won't be triggered. The event handler must be attached before the event occurs in order to detect it.

However, if a Promise has been fulfilled (i.e. the **resolve** or **reject** functions are called) ***before*** you added the **then()** function to it, the **then()** function will still detect that fact and act accordingly.

The **then()** function can also be used to catch rejections.

```
promise.then (<resolve function> , <reject function> ) ;
```

As shown before a Promise can be **resolved** or **rejected**.

```
var step1 = new Promise(function(resolve, reject)
{
    var x = Math.random();

    if ( x == 0)

        reject(new Error("x is 0 error"))

    else

        setTimeout( function() {resolve("x =" + x);} , 1000)

});
```


The **then()** function will call the appropriate handler based on whether the promise was resolved or rejected.

```
step1.then( function (data)
            { console.log(data); },

            function (err)
            { console.log(err); }
        )
```

The first function will be called if the **step1** Promise is resolved.

```
step1.then(  
  function (data)  
    { console.log(data); },  
  
  function (err)  
    { console.log(err); }  
)
```

The second function will be called if the Promise is rejected.

.catch();

As we have seen, if the code in your Promise can't be resolved (i.e. there is some error, and it is rejected) then you can let the rest of your code know by rejecting the Promise and catching the error with the second handler function passed to the **then()** function.

You can also catch an error with the **catch()** function.

I.e. if a promise is **rejected** it calls the **catch()** function instead of **then()**

You can use either format:

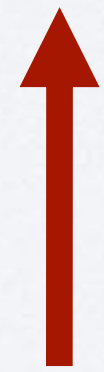
```
promise.then(<resolve function>, <reject function>) ;
```

or

```
promise.then(<resolve function>).catch(<reject function>) ;
```

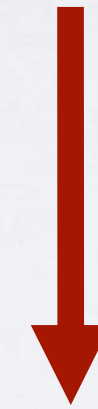
One difference between the two is that in certain circumstances using a separate catch function can also catch errors thrown by the resolve function.

```
promise.then(<resolve function>).catch(<reject function>);
```



Rejections from this function could be caught by this catch


Rejections from this function are **not** caught by this function



```
promise.then(<resolve function>, <reject function>)";
```


Note: in some of the following code examples the Promise contains synchronous code only.

This is just to simplify the code samples. The important part of what follows is the Promise calling the **resolve** or **reject** parameters. In real world examples this would most likely be at the end of some asynchronous action.


```
var step1 = new Promise(function(resolve, reject)
{
    x = Math.random();
    if ( x == 0)
     reject(new Error("x is 0 error"))
    else
        setTimeout( function(){resolve("x = " + x);} , 1000)
});
```

```
step1.then( function (data)
                { console.log(data); })

.catch (function(err)
            { console.log(err); })
```



This code is executed if the Promise invokes the **reject** function.

```
var step1 = new Promise(function(resolve, reject)
{
    x = Math.random();
    if ( x == 0)
        reject(new Error("x is 0 error"))
    else
        setTimeout( function(){resolve("x = " + x);} , 1000)
});
```

```
step1.then( function (data)
           { console.log(data); })

        .catch (function(err)
                { console.log(err); })
```

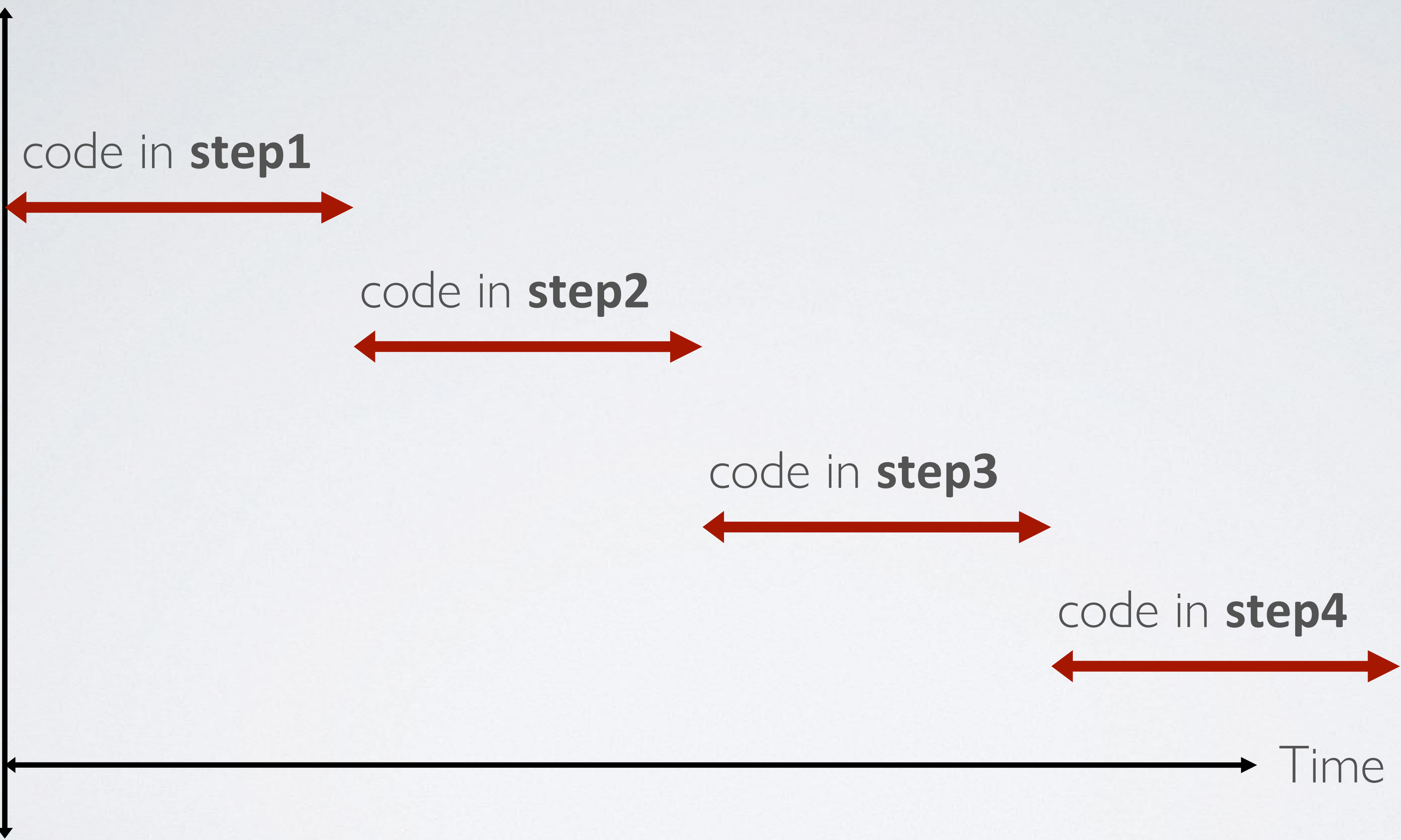
This code is executed if the Promise invokes the **resolve** function.

Chaining/Composition

Remember the **step1()**, **step2()**, **step3()**, **step4()** example from before.

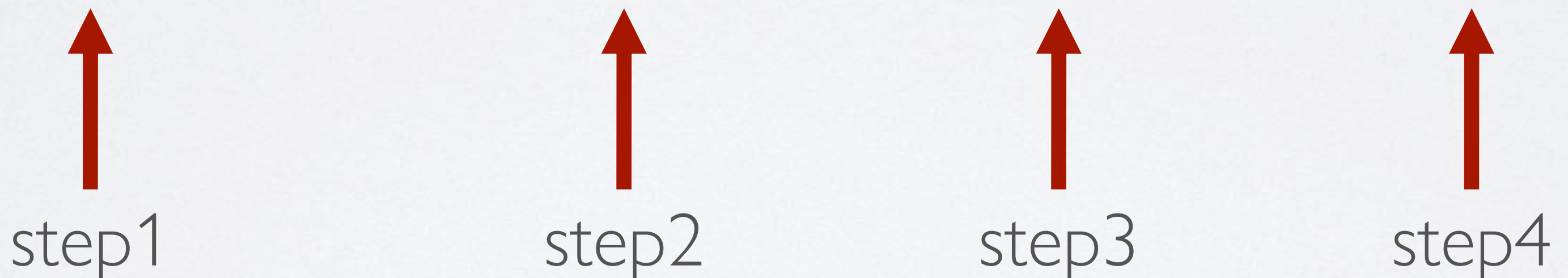
We want to delay the execution of each function until the previous one has finished.

Script starts



We would expect code similar to the following:

```
aPromise.then ( ? ) .then ( ? ) .then ( ? )
```



step1 step2 step3 step4

To get the effect we want, each successive **then()** function **must return a Promise** so the following **then()** will wait until it resolves.

```
var step1 = new Promise(function(resolve, reject)
{
    console.log ("Starting step 1");

    setTimeout(resolve, 1000);
});

var step2 = new Promise(function(resolve, reject)
{
    console.log ("Starting step 2" );

    setTimeout(resolve, 1000);
});

var step3 = new Promise(function(resolve, reject)
{
    console.log ("Starting step 3");

    setTimeout(resolve, 1000);
});

var step4 = new Promise(function(resolve, reject)
{
    console.log ("Starting step 4");

    setTimeout(resolve, 1000);
});
```

You might be tempted to create 4 Promises and call them as follows.

```
step1.then(()=> step2)
        .then(()=> step3)
        .then(()=> step4);
```

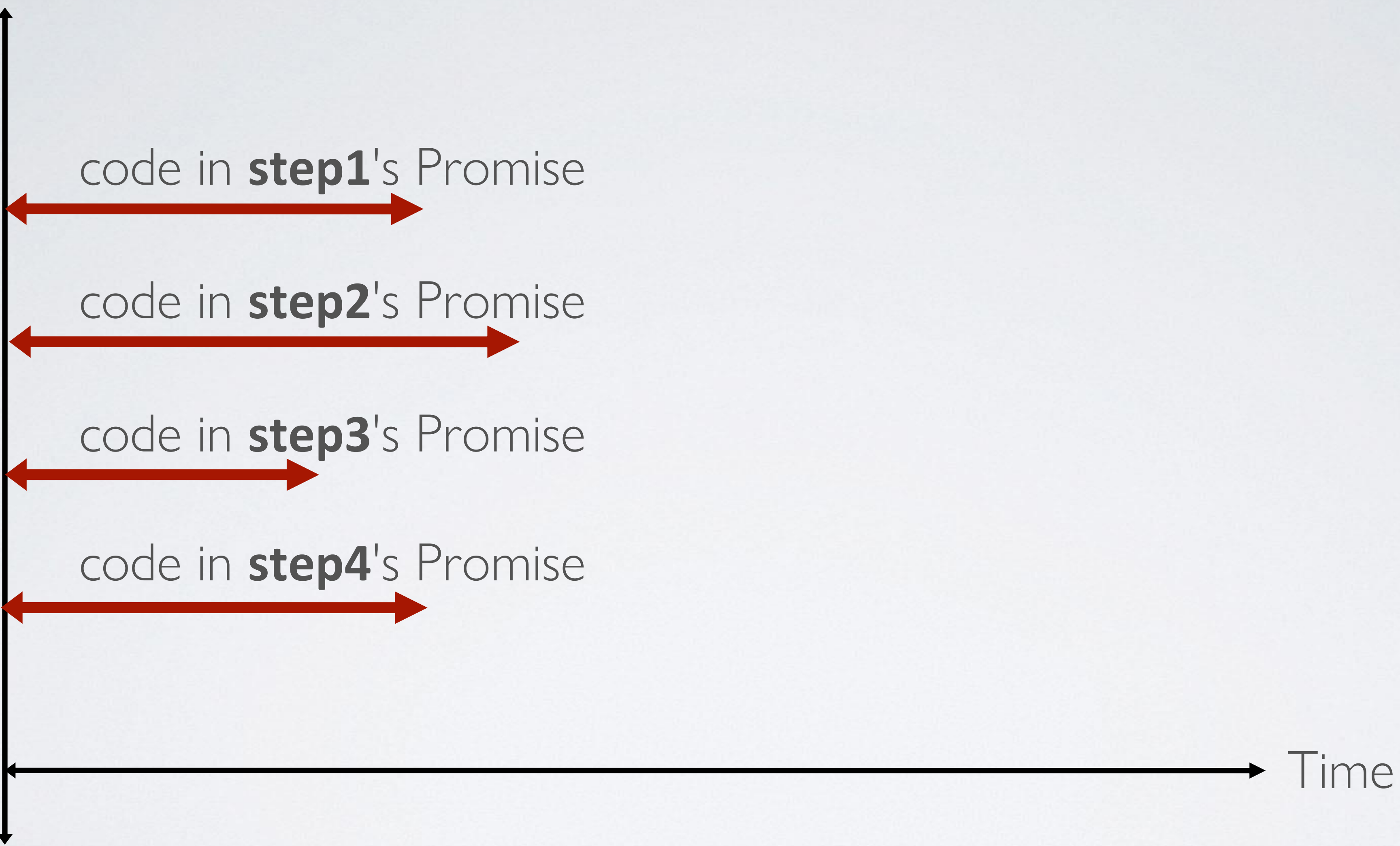
But this **won't work: *creating*** a Promise ***immediately*** starts executing its executor function (so all 4 promises will start executing at the same time if we create them at the same time.).

Remember, these are (mostly) equivalent

`() => step2`

`function() {return step2;`

Script starts




To fix this we need to pass a handler function to each **then()** that **returns a Promise**.

(A *promise factory* is a function that **creates** and **returns** a promise.)

In this example we create one for the first step.

Note that the Promise **is not created here**. We are declaring a function that can create a Promise **when it is called in the future**.

```
var step1 = function()  
    {return new Promise( function(resolve, reject)  
        {  
            console.log ("Starting step 1");  
  
            setTimeout(resolve, 1000);  
        })  
    }  
    ;
```




```
var step1 = function() {return new Promise(function(resolve, reject)  
{  
    console.log ("Starting step 1");  
  
    setTimeout(resolve, 1000);  
}}};
```

```
var step2 = function() {return new Promise(function(resolve, reject)  
{  
    console.log ("Starting step 2" );  
  
    setTimeout(resolve, 1000);  
}}};
```

```
var step3 = function() {return new Promise(function(resolve, reject)  
{  
    console.log ("Starting step 3");  
  
    setTimeout(resolve, 1000);  
}}};
```

```
var step4 = function() {return new Promise(function(resolve, reject)  
{  
    console.log ("Starting step 4");  
  
    setTimeout(resolve, 1000);  
}}};
```

We can now
define a Promise
factory for each
step.

Now when we pass these handler functions to each **then()** function, they are executed once the previous Promise has resolved. So their own Promise is only created (and returned) at that point.

```
step1 () .then (step2) .then (step3) .then (step4) ;
```


If the handler function of a **then()** function returns a Promise then it is only after that Promise resolves/rejects that the handler function of the following **.then()** is executed.

step1 returns a
Promise

When that Promise
resolves we can
execute **step2**

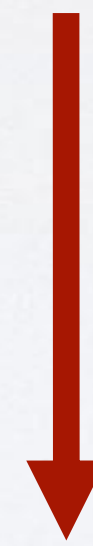


```
step1 ().then (step2) .then (step3) .then (step4) ;
```


This **then()** creates a Promise that resolves when **step2** does



When that Promise resolves we can execute **step3**



```
step1 () .then (step2) .then (step3) .then (step4) ;
```

This **then()** creates a Promise that resolves when **step3** does

When that Promise resolves we can execute **step4**



```
step1 () .then (step2) .then (step3) .then (step4) ;
```


Script starts

```
step1 () .then (step2) .then (step3) .then (step4) ;
```

code in **step1**'s Promise



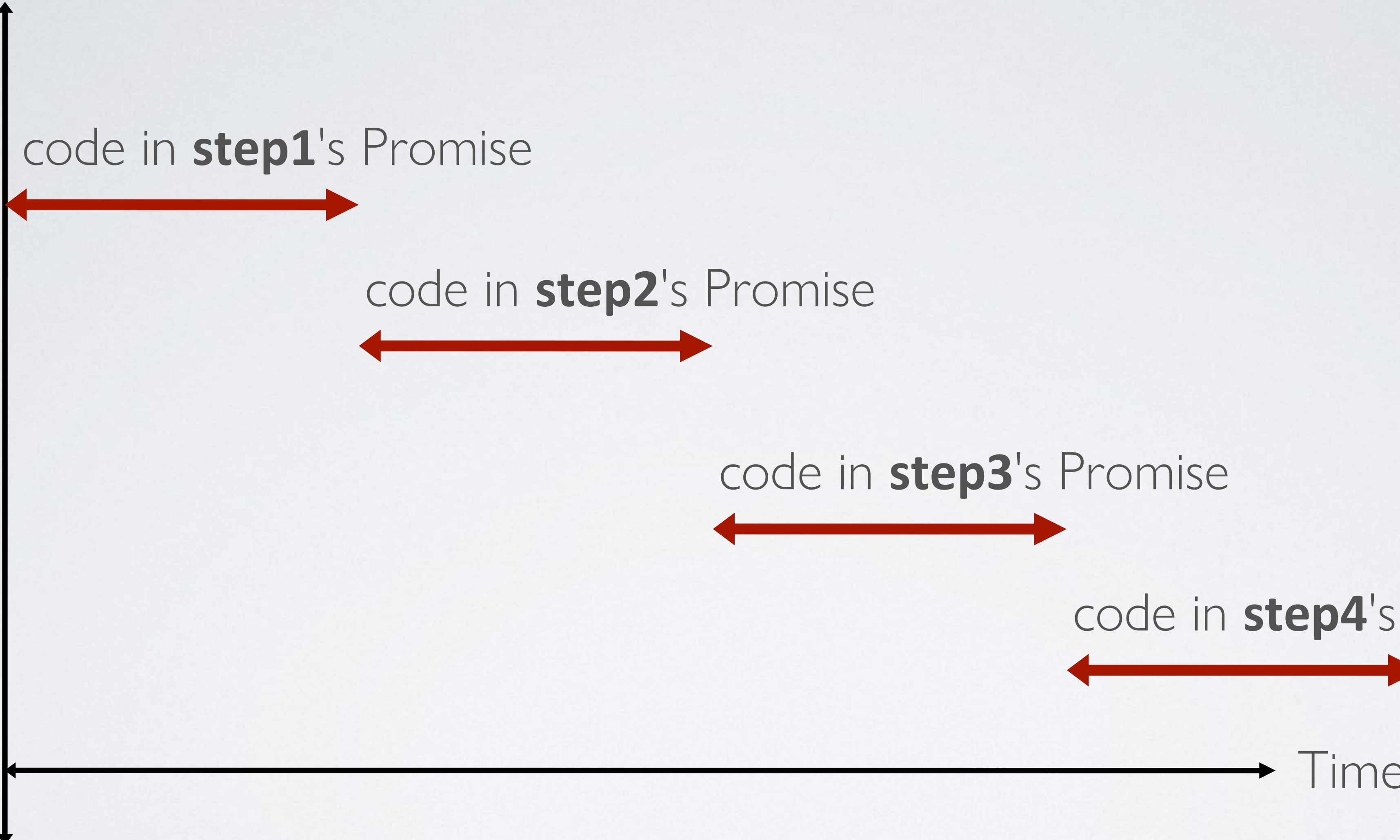
code in **step2**'s Promise



code in **step3**'s Promise



code in **step4**'s Promise



Time

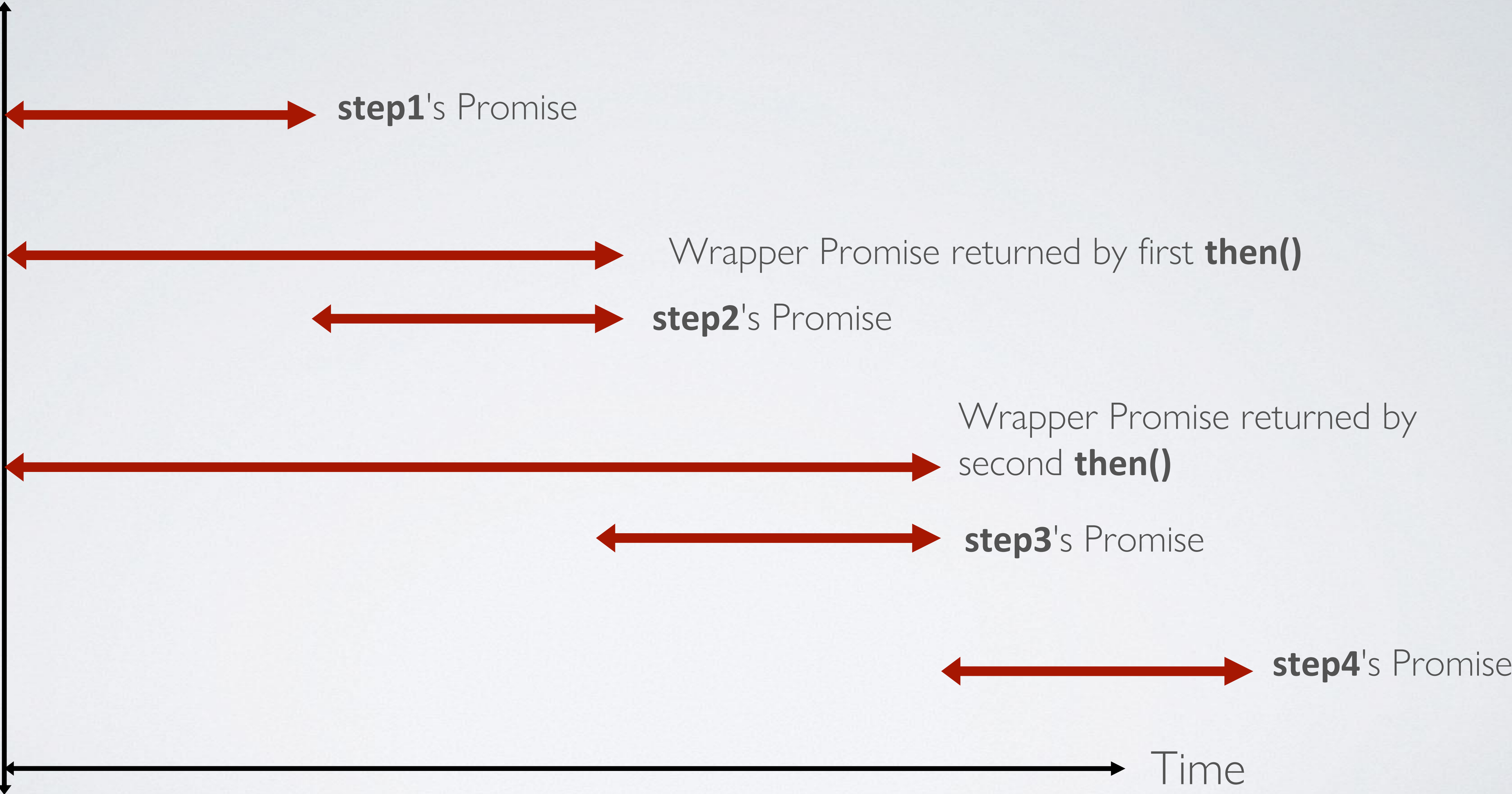
Note that each **then()** function **immediately** creates its own Promise that will be fulfilled just after the handler function's Promise is.

It takes **any data** resolved or rejected by the handler function's Promise and passes it on **as its own data**.

This is because we need a promise straightaway in order to invoke each **then()** function in the chain. So the **then()** function creates a temporary wrapper Promise that is invoked immediately (unlike the handler function's promise) but is resolved once the handler function's promise is.

Script starts

```
step1 () .then (step2) .then (step3) .then (step4) ;
```



Remember, this wrapper function exists mainly so that you have a Promise whose fulfilment will be the same as the handler function's Promise, but - most importantly - is created immediately so that subsequent **then()** methods can be invoked on them straight away.

I.e. the Promises of **step2** and **step3** may not even exist when we try to attach the **then()** function to them.

To get around this, the **then()** function returns the wrapper Promise that is available immediately.

```
step1 () .then (step2) .then (step3) .then (step4) ;
```

E.g. we execute **step1** which returns the first Promise

```
step1().then(step2).then(step3).then(step4);
```



```
step1 () .then (step2) .then (step3) .then (step4) ;
```



This handler
function will
execute when the
Promise from
step1() resolves.

But this **then()** function **immediately** returns a Promise



```
step1 () .then (step2) .then (step3) .then (step4) ;
```

... that resolves only after **step2**'s Promise does.

This **then()** function can now be attached to the previous Promise (returned by the previous **then()**). It also immediately creates its own Promise that resolves only when the **step3** Promise resolves.



```
step1 () .then (step2) . then (step3) .then (step4) ;
```

step3 won't create its own Promise until the previous Promise (based on **step2**'s Promise) resolves.

Similarly, this **then()** function can now be attached to the previous Promise (returned by the previous **then()**).



```
step1 () .then (step2) .then (step3) . then (step4) ;
```

step4 won't be executed until the previous Promises (from **step1**, **step2**, and **step3**) resolves.

As with the others, this **then()** also returns a Promise but in this case we don't use it.



```
step1 () .then (step2) .then (step3) . then (step4) ;
```

In order for a chain to work in this way (i.e. asynchronous functions that execute in a specific sequence) we must guarantee 2 things.

The **then()** functions **must be passed a function**.

That function must **return a Promise**.

```
step1 ()  
  .then ((data) => return new Promise(function (resolve, reject{...}))  
    .then(step2)  
      .then(step3) ;
```


Note that, regardless of what a handler functions returns, the **then()** function will **always** return a Promise.

If the handler function returns a synchronous value like this.

```
return 4;
```

then() will wrap it in a Promise (that resolved with that value), and return that promise.

E.g. Effectively it performs the following:

```
return Promise.resolve(4);
```


This allows you to continue to chain the results of each **then()** function regardless of whether they were asynchronous or not.

If the handler function does not return a value, **then()** will still return a Promise that resolves with **undefined**.


If there is **no handler function** the **then()** function will still return a Promise.

It will **have the same status as the Promise it was attached to.**

E.g. this promise will resolve with the data "test"



```
new Promise (function (resolve, reject)
    { setTimeout (resolve, 1000, "test"); })
    .then()
    .then()
    .then (x) => { console.log(x) } );
```



```
new Promise (function (resolve, reject)
  { setTimeout (resolve, 1000, "test"); })
  .then()
  .then()
  .then (x) => { console.log(x) } );
```

This **then()** function will return a Promise that resolved with "test" since it has no handler function of its own and the Promise it is attached to resolved with that value.


```
new Promise (function (resolve, reject)
    { setTimeout (resolve, 1000, "test"); })
```

This **then()** function will **also** return a Promise that resolved with "test" since it has no handler function of its own and the Promise it is attached to resolved with that value.

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

```
new Promise (function (resolve, reject)
  { setTimeout (resolve, 1000, "test"); })
  .then()
  .then()
  .then( (x) => {console.log(x)} ) ;
```



This handler function outputs "test"

If a handler function doesn't return a Promise, the **then()** function will still return a (fulfilled) Promise as we have seen. This allows to us maintain a chaining sequence.

In the example below we assume the **step1**, **step2** and **step3** functions return Promises, while **noPromise** does not.

```
step1 () .then (noPromise) .then (step2) .then (step3) ;
```

Script starts

code in **step1**'s Promise



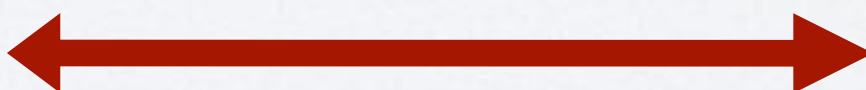
code in **noPromise** (synchronous)



code in **step2**'s Promise



code in **step3**'s Promise



Time

In the example below we assume **step1** and **step2** return Promises, while **noPromise1** and **noPromise2** do not.

```
step1 () .then (noPromise1) .then (noPromise2) .then (step2) ;
```

Script starts

code in **step1**'s Promise



code in **noPromise1** (synchronous)



code in **noPromise2** (synchronous)



code in **step2**'s Promise



Time

Summary

Assume we have a **then()** function attached to a Promise, which is passed a handler Function.

```
somePromise.then (handlerFunction)
```

After **somePromise** resolves the **handlerFunction** is executed.

If **somePromise** resolves with data, and **handlerFunction** has a parameter, then that data is passed to **handlerFunction**.

Chaining

```
somePromise.then(handlerFunction1) . then(handlerFunction2)
```

then() always returns a Promise.

This new Promise can have its own **then()** function.

Wrapper Functions

```
somePromise.then(handlerFunction1).then(handlerFunction2)
```

If the handler function returns a Promise the **then()** function will create (and return) a wrapper Promise that will be fulfilled just after the handler function's one does. It will also resolve/reject with the same arguments.

This means you will immediately have a Promise that resolves the same time as the handler function's returned Promise without having to wait for that handler function to execute in order to create its Promise.

Synchronous Return Values

```
somePromise.then(handlerFunction1).then(handlerFunction2)
```

If a handler function returns a synchronous value (i.e. not a Promise) the **then()** function creates a Promise that has already resolved with this data.

If a handler function returns nothing the **then()** function creates a Promise that has resolved with **undefined**.

No Handler Function Argument

```
somePromise.then() .then(handlerFunction2)
```

If a handler function is not passed a function then the **then()** function creates a Promise with the same state and data as the Promise it was attached to (allowing it to pass its data along the chain)

Promise.resolve()

In some circumstances it may be desirable to create a Promise for a *synchronous value* (i.e. a value that is available immediately).

You can do this with the **Promise.resolve()** function.

It will return a Promise that has been resolved. This can be useful for starting a Promise chain or when returning a value from a function which could be Synchronous or Asynchronous. Returning a Promise either way can simplify your code.

E.g. the example below can return a synchronous value (i.e. the contents of the **value** variable) or an asynchronous value. To simplify matters we can return the synchronous value in a Promise.

```
function start(value)
{
    if (value !== 0) {
        return Promise.resolve(value);
    }
    else
        return new Promise(function (resolve, reject) { ..... });
}
```

Wrap the value variable in Promise and return it.

Assume the code for retrieving asynchronous value goes here.

Since our **start()** function will now return a Promise, whether the value is synchronous or asynchronous, we can have it start a chain (without having to add extra logic to check if the function returned a Promise or not).

```
var x = Math.random();
```

```
start(x).then(function(data) {console.log(data);})
```

We can also use **Promise.resolve()** to create a Promise purely for the purpose of starting a chain.

```
Promise.resolve() .then (step1) .then (step2) ;
```


Promise.all()

```
var imgP = new Promise(  
    function(resolve, reject)  
    {  
        var img = new Image();  
  
        img.onload = function()  
        {  
            resolve();  
        }  
  
        img.onerror = function()  
        {  
            reject();  
        }  
  
        img.src = "pic1.jpg";  
    }  
)
```

We can create a promise that will **resolve** or **reject** depending on whether a image is properly downloaded.

Assuming we have a function **showImage()** that will display the image, we can wait until the image is loaded before calling it.

```
imgP.then(function() {showImage();})
```

or

```
imgP.then(showImage)
```

Assume we want to load several images and must wait **until they all download** before calling a function (**showImages()**) that will display them.

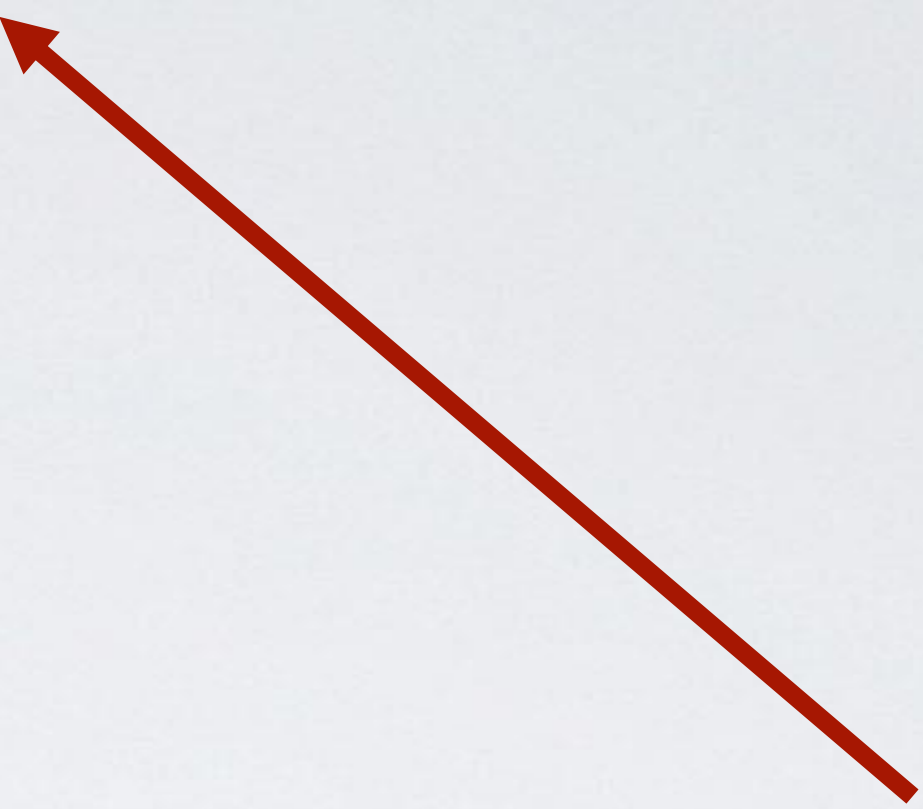

```
var images = ["pic1.jpg", "pic2.jpg", "pic3.jpg", "pic4.jpg", "pic5.jpg"]

var promises = [];

for (var i = 0; i < images.length; i++)
{
    promises[i] = new Promise(
        function(resolve, reject)
        {
            var img = new Image();

            img.onload = function()
            {
                resolve(y);
            }

            img.onerror = function()
            {
                reject(y);
            }
            img.src = images[i];
        }
    )
}
```




We can use an array of the image filenames to create an array of Promises that try to load each image.



The images start downloading when we assign a value to the Image object's **src** property.

Here we indicate that we only want **showImages** to execute when all 5 images have downloaded (i.e. all 5 promises have resolved).

```
function() {return promise[1];}]  
promise[0].then(()=> promise[1])  
              .then(()=> promise[2])  
                .then(()=> promise[3])  
                  .then(()=> promise[4])  
                    .then(showImages);
```



arrow function notation for

The diagram consists of a horizontal red line segment. From the right end of this segment, a red arrow points diagonally upwards and to the right, ending at the first parameter of the function definition 'function()' in the line above.

Note that **we are not** having each image start downloading when the previous one has loaded (we already created the Promises so they had started downloading then).

What the code below is saying is that **showImages** will only execute if all of the previous 5 promises have resolved at some point in the past.

```
promise[0].then(() => promise[1])  
            .then(() => promise[2])  
            .then(() => promise[3])  
            .then(() => promise[4])  
            .then(showImages);
```

Remember: The **then()** function is called if the Promise it is attached to has been fulfilled. It doesn't matter if that Promise was fulfilled ***before*** the **then()** function was attached to it.

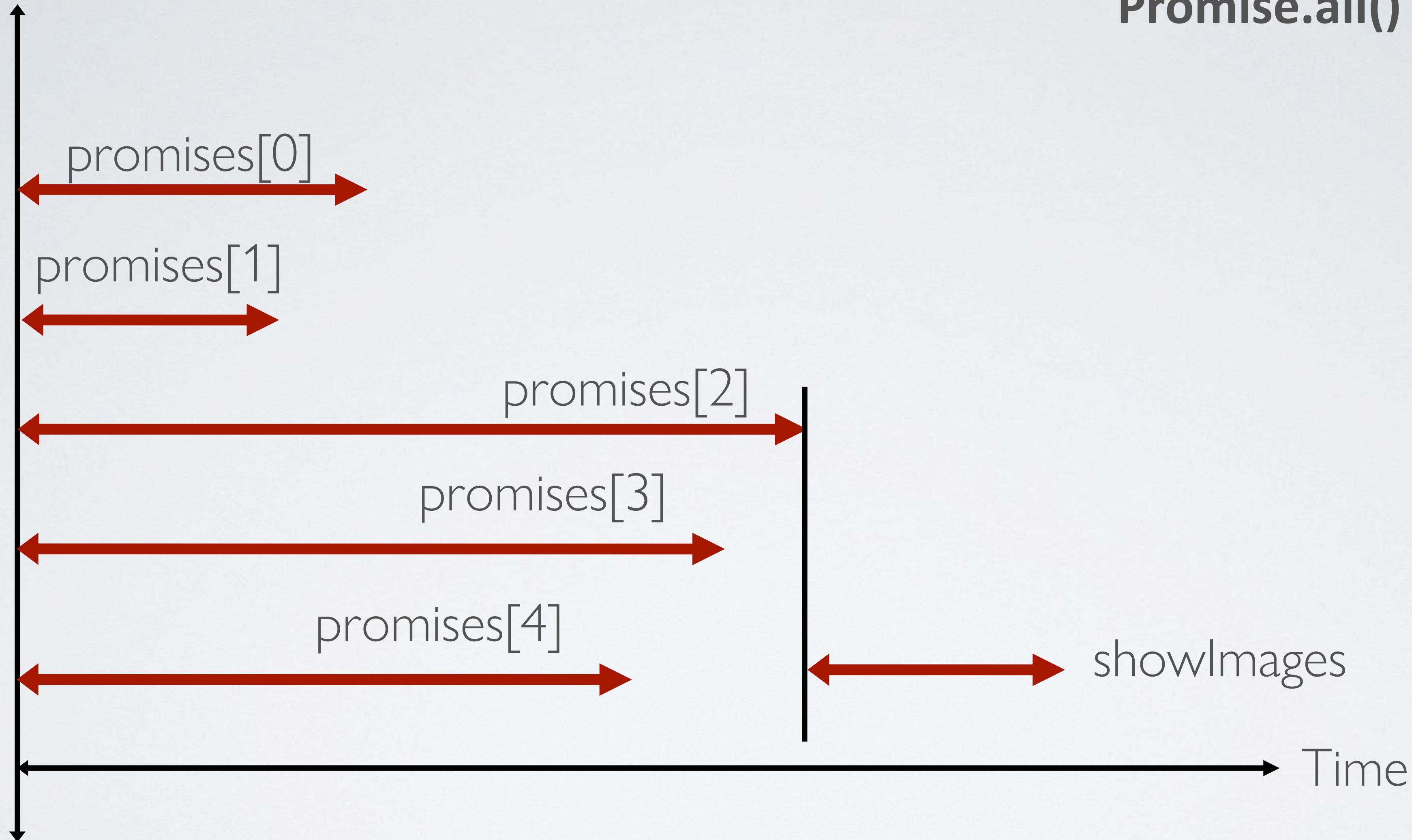
A simpler way of achieving the same effect is to use **Promise.all()**.

We can pass the array of promises to the **Promise.all()** function. This function returns a Promise that will wait until all the promises in the array have resolved before resolving itself.

```
Promise.all(promises) . then(showImages)
```

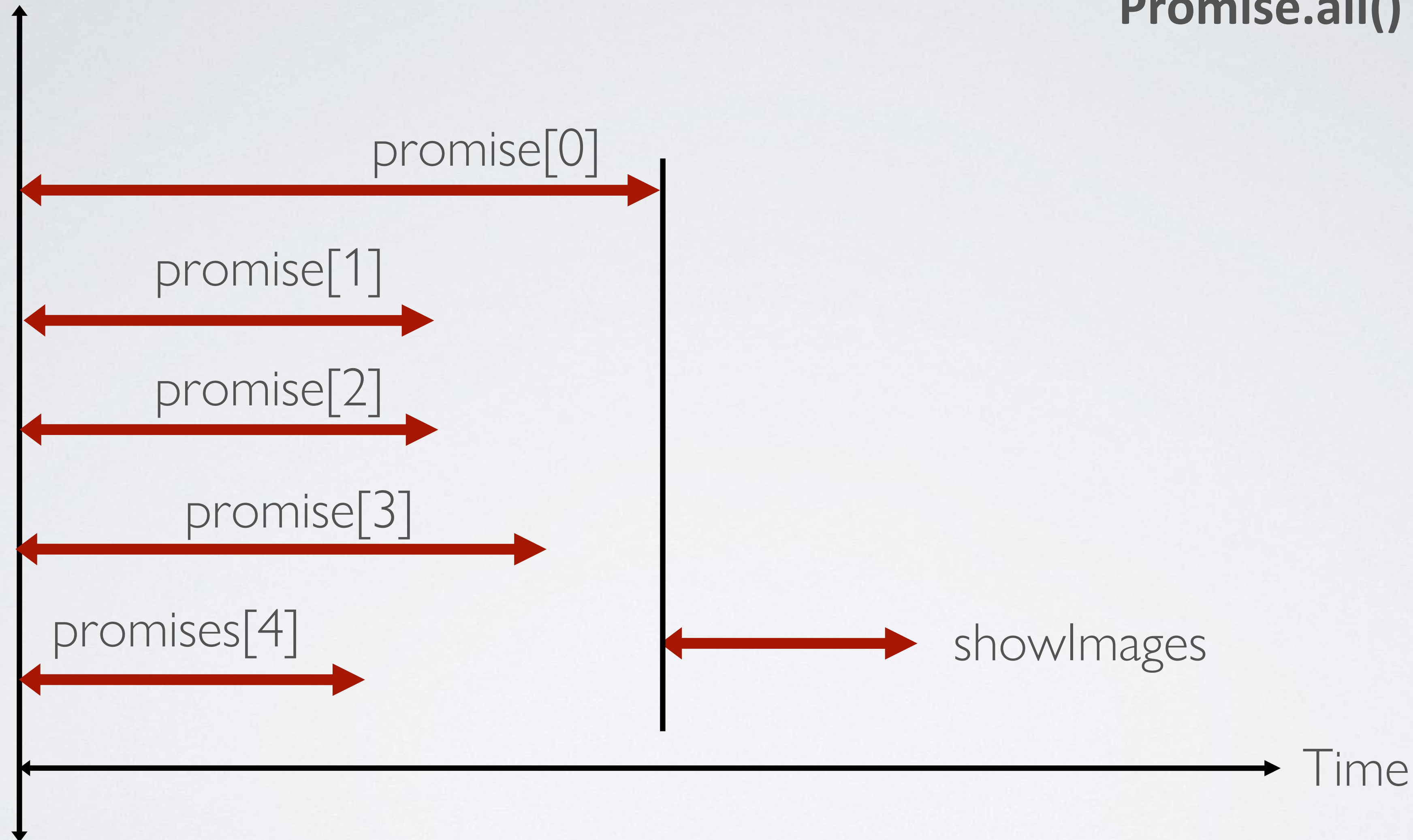
Script starts

Promise.all()



Script starts

Promise.all()



If **any** of the Promises in the array are rejected we can handle it straight away.

```
Promise.all(promises)
  .then(function() {showImages();})
  .catch(function()
    {console.log("at least one image didn't load");})
```


We can also arrange to send data from all the promises in the array to the **then()** handler function parameter.

```

for (var i = 0; i < images.length; i++)
{
    promises[i] = new Promise(

        function(resolve, reject)
        {
            var img = new Image();

```

Copy the value of **i** on
account of closure

→ **var y = i;**

Here we pass the (copy
off the) index of the image
to the resolve function.

→ **resolve(y) ;**

```

    img.onload = function()
    {

```

```

        reject(y);
    }
    img.src = images[i];

```

```

    }

```

```

)

```

```

}

```


Each promise in the array has the data it resolved with added to an array that **Promise.all(...)** will resolve with.

```
Promise.all(promises).then(function(dataArray) {console.log(dataArray[0]);})
```

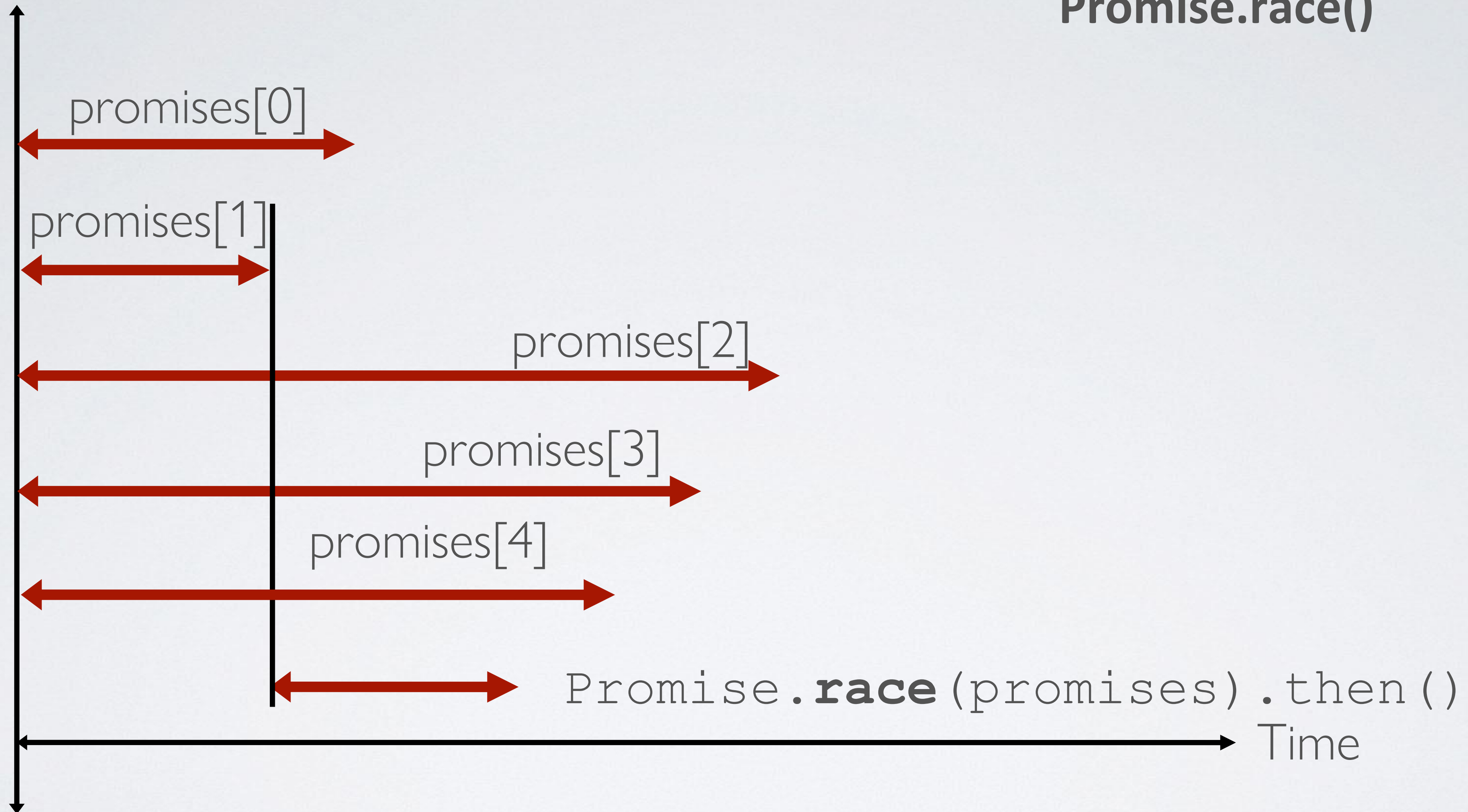
Promise.race()

We can also wait until the first Promise in the array has resolved using the **Promise.race()** function. This returns a Promise that resolves when the first Promise in the array resolves.

```
Promise.race(promises).then(function() {  
    console.log("the first of the images has downloaded");  
})
```

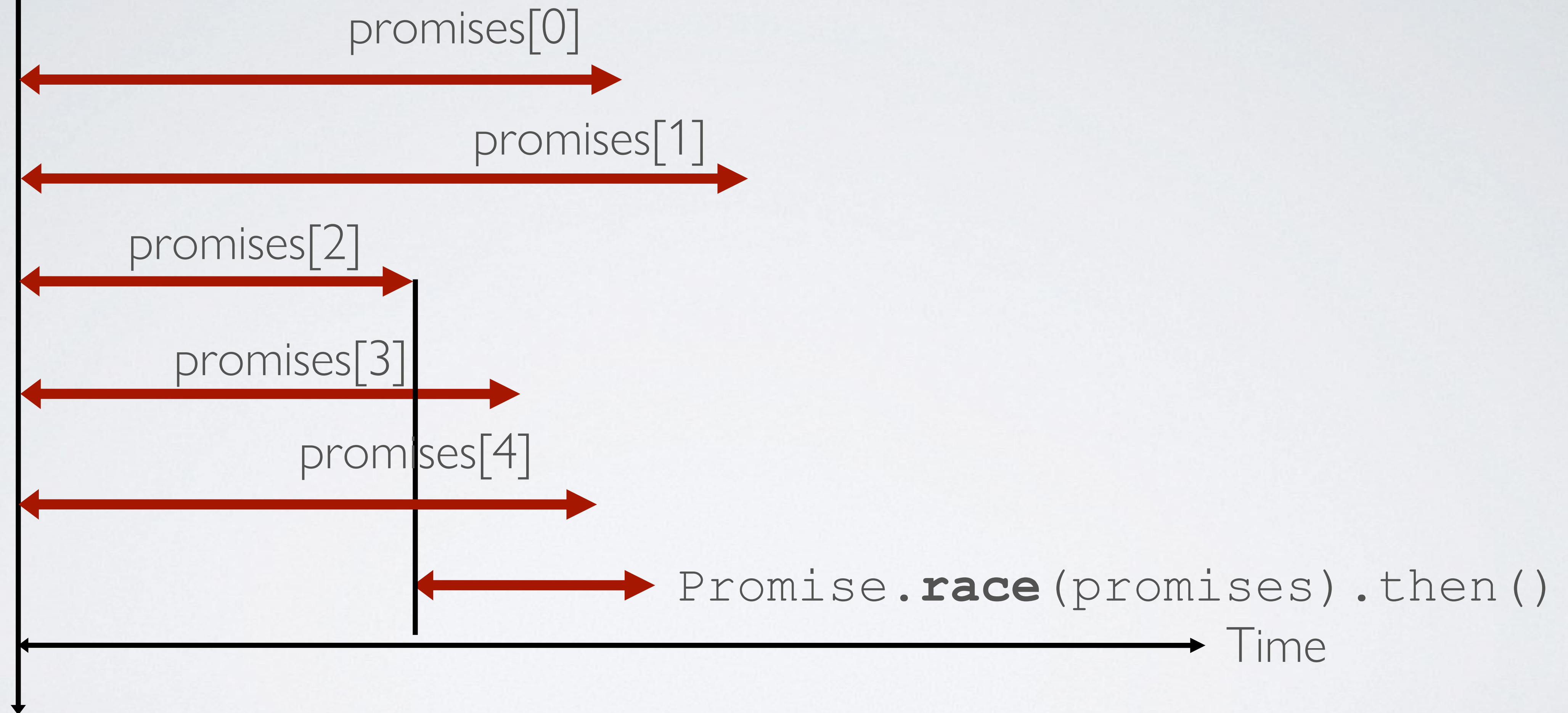
Script starts

Promise.race()



Script starts

Promise.race()



async functions

The following features (the **async** and **await** keywords) can be used to simplify how you use Promises.

async

JavaScript supports asynchronous functions. You denote these with the **async** keyword.

These are functions that don't block your execution. I.e. JavaScript **does not wait until they finish** before carrying on with the next statement in your code.

You can specify that a function is asynchronous as follows:

```
async function getData() { ... }
```

or

```
var getData = async function getData() { ... }
```


E.g.

```
async function getData() { ... }
```

```
function function1() { ... }
```

```
function function2() { ... }
```

```
getData();
```

```
function1();
```

```
function2();
```

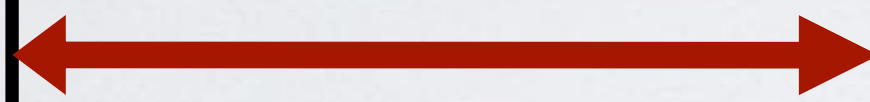
Script starts



getData()



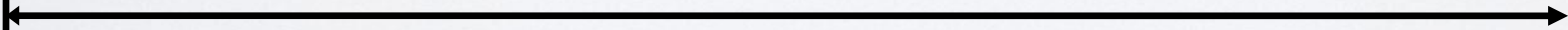
function1()



function2()



Time



Since the function is asynchronous you can't use them to retrieve data in expressions as you normally would (since JavaScript won't wait until the data is ready before carrying on with the expression/next statement).

```
async function getData()  
{  
    someCode(); // assume this code takes time to execute  
    return 5;  
}
```


```
var x = getData();
```

```
console.log(x);
```

```
async function getData()  
{  
    someCode();  
    return 5;  
}
```

```
var x = getData();
```

```
console.log(x);
```



This function may still be executing when we try to access the value the function returns.


```
async function getData()  
{  
    someCode();  
    return 5;  
}
```

~~var x = getData();~~

```
console.log(x);
```

Script starts

getData()


return 5

x = ?

Time

To make the data returned from an **async** function accessible the function will automatically return a **Promise** that resolves with the value returned by the function.

```
async function getData()  
{  
    someCode();  
    return 5;  
}
```



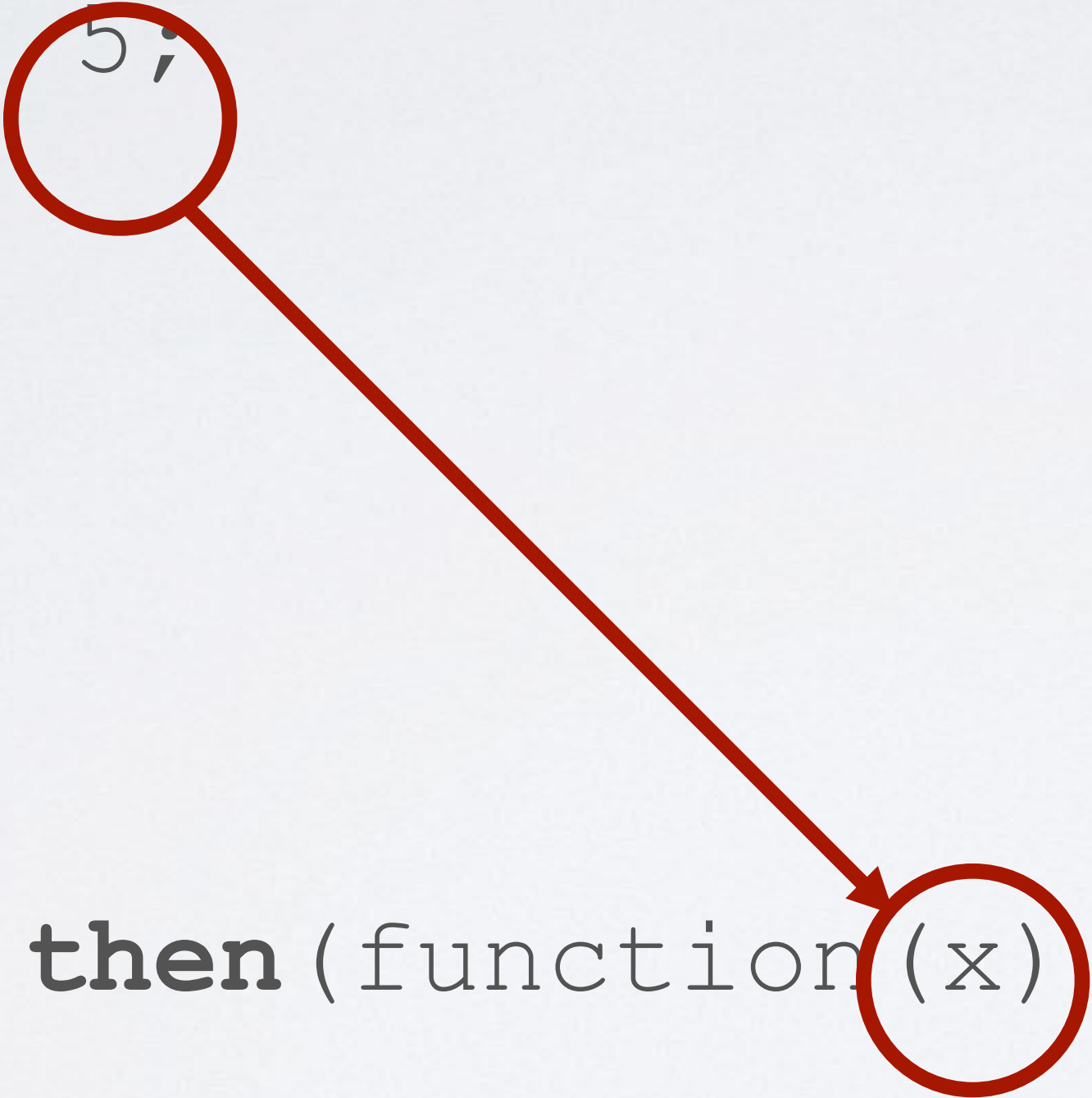
I.e. when you call an **async** function it ***immediately*** returns a **Promise**. When the function code returns a value the async function will resolve the **Promise** with that value (which can then be passed to the **then()** function's handler function).

```
getData().then(function(x) { console.log(x); });
```



```
async function getData()  
{  
    someCode();  
    return 5;  
}
```

```
getData().then(function(x) { console.log(x); });
```



I.e. when you call an **async** function it ***immediately*** returns a **Promise**. When the function code returns a value the async function will resolve the **Promise** with that value (which can then be passed to the **then()** function's handler function).

Script starts

`getData()`

`return 5`

`function(x) { console.log(x); }`

Time

`getData().then(function(x) { console.log(x); });`

We could have achieved this without **async** functions.

However, **async** functions simplify our code.

Without async

```
function getData()  
{  
  return new Promise(function (resolve, reject)  
  {  
    someCode();  
    resolve(5);  
  })  
};
```

```
getData().then(function(x) { console.log(x); });
```



```
function getData()  
{  
  return new Promise(function (resolve, reject)  
  {  
    someCode();  
    resolve(5);  
  })  
};
```

vs.

```
async function getData()  
{  
  someCode();  
  return 5;  
}
```

await

Inside an **async** function you can use the **await** keyword to simplify the use of promises in your code.

await allows you to easily retrieve the values resolved by a Promise. It will also wait until that Promise has resolved before going onto the next statement in the function.

await can only be used in **async** functions.

```
function createShortDelay() {  
    return new Promise(function(resolve, reject)  
        {setTimeout(function() {resolve("Finished Short Delay");}  
            ,1000)  
        }  
    );  
}
```

```
async function test()  
{  
    var str = await createShortDelay();  
    console.log(str);  
}
```

This function
returns a **Promise**
that resolves after a
second has elapsed.

await appears before an expression. It can also appear in the RHS of an assignment.

await is followed by an expression that results in a Promise.

In this example, the **createShortDelay()** function will return a Promise

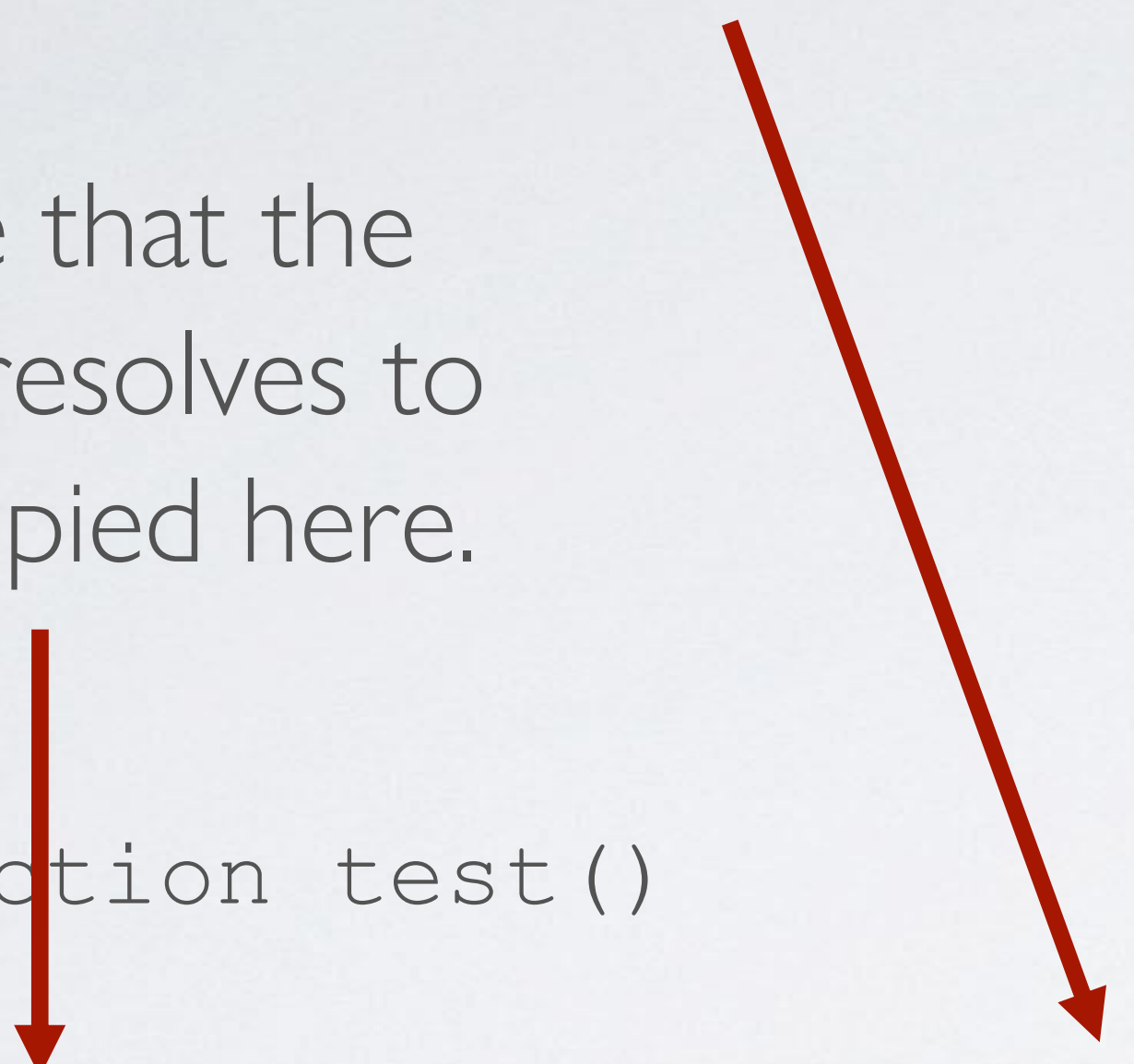
```
async function test()  
{  
  var str = await createShortDelay();  
  
  console.log(str);  
}
```

This statement won't execute until the Promise in the previous **await** statement resolves.


This function can
resolve to some
value.

The value that the
promise resolves to
will be copied here.

```
async function test()  
{  
    var str = await createShortDelay();  
  
    console.log(str);  
}
```

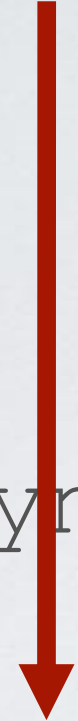


await is useful for simplifying the work of retrieving the value that the Promise on the right-hand side of the expression resolves to.



```
async function test()  
{  
  var str = await createShortDelay();  
  
  console.log(str);  
}
```

You don't need to get a value from **await**. You can just use it to pause the function until the expression resolves.



```
async function test()  
{  
  await createShortDelay();  
  console.log("after short delay");  
}
```



l.e. **await** is used whenever you want to wait until asynchronous code terminates before moving on to the next statement.

```
async function test2()  
{  
    await createLongDelay() ;  
  
    console.log("after long delay");  
  
    await createShortDelay() ;  
  
    console.log("after short delay");  
}
```

Assuming our two functions return promises we could have implemented the previous example as follows by using **then()**.

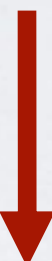
```
function test2()  
{  
    createLongDelay()  
  
    .then(function() {  
        console.log("after long delay");  
    })  
    .then(function () {  
        return createShortDelay();  
    })  
    .then(function() {  
        console.log("after short delay");  
    })  
}
```

We must return the Promise returned by the async function so the **then()** function can use it.




Note that these two handler functions will execute at the same time since the first one doesn't return a Promise. I.e. **then()** doesn't have to wait for anything to resolve (so it will return a Promise that has already resolved).

```
function test2()  
{  
  createLongDelay()
```



```
    .then(function() {  
        console.log("after long delay"); })
```



```
    .then(function () {  
        return createShortDelay(); })
```

```
    .then(function () {  
        console.log("after short delay"); })
```


```
}
```

A non-Promise expression following the **await** keyword is converted to a *resolved Promise*.

```
async function test3()  
{  
    var x = await "not a promise";  
  
    return x;  
}
```


A non-Promise expression following the **await** keyword is converted to a ***resolved Promise***.

```
async function test3()  
{  
  var x = await "not a promise";  
  
  return x;  
}
```

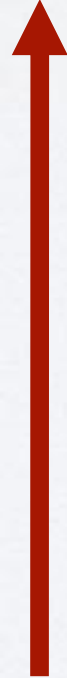


Promise.resolve("not a promise")

Rejecting a Promise in an *await* Expression

Sometimes a promise might **reject** instead of **resolve**.

```
function createShortDelay() {  
    return new Promise(function(resolve, reject)  
        {setTimeout(function(){reject("an Error occurred");}  
            ,1000)  
        }  
    );  
}
```



If the Promise in the **await** expression **rejects** then it will throw an **error** that you catch.

```
async function()  
{  
    try {  
        var str = await createShortDelay();  
        console.log(str);  
    }  
    catch (e)  
    {  
        console.log("Error");  
    }  
}
```


If you don't catch the error inside the function (as in the previous example) the error then propagates outside the function by causing the promise that is returned by the function to reject. We can detect this with the **catch** function of the promise.

```
async function test()  
{  
    var str = await createShortDelay();  
  
    console.log(str);  
}
```

```
test().then(function(){console.log("it worked");})  
      .catch(function(){console.log("it failed");})
```

References:

<https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html>

<https://developers.google.com/web/fundamentals/primers/promises>

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises