**Creating objects**

In your applications, most of the time you don't work with primitives but rather with objects. You can directly create objects with object literal like this:

var Person = {

firstName: 'John',

lastName: 'Doe'

}

This is sometimes good enough, but in many cases, you want to work with many objects, which share the same characteristics - that is: they have the same properties and methods. These can be users in your applications, bank accounts, cars or anything else. The point is, you'll want to work with 'Jane' and 'John' in the same way because they can both be considered a person - they have a name, surname, age and so on.

Alright, so how do you create objects which have the same methods and properties? Using constructor functions and the new operator.

The new operator

Let's consider the following example:

var **Person** = function (firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

};

var john = new Person('John', 'Doe');

The Person is a **constructor function**. Its whole purpose is to create new objects which represent different persons. We can be sure that each person will have the first name and the last name, which are provided as parameters to this function. Notice that unlike regular functions, constructors begin by convention with a **capital letter**. This way you can easily tell them apart from ordinary functions.

Then we are creating a specific person instance called john with firstName 'John' and lastName 'Doe'. Notice that there is a **new keyword** before the Person constructor function call. It is actually an operator and it is responsible for all the magic when creating new persons. It does actually quite a lot:

1. It creates a new blank object.
2. It makes this to point to this newly created object inside the constructor function
3. It sets the prototype of the newly created object to the constructor function's prototype.
4. It makes the constructor function return the newly created object IF it is not returning anything.

That is the reason why John is created even though the function Person does not have an explicit return statement. That's the reason why using this.firstName=firstName; sets the first name provided to the new object. Without the new keyword, this would point to the global object in our case and not to the newly created instance.

<https://stackoverflow.com/questions/23074875/angularjs-factory-and-service>

Factory and Service is a just wrapper of a provider.

**Factory**

Factory can return anything which can be a class(constructor function), instance of class, string, number or boolean. If you return a constructor function, you can instantiate in your controller.

myApp.factory('myFactory', function () {

// any logic here..

// Return any thing. Here it is object

return {

name: 'Joe'

}

}

**Service**

Service does not need to return anything. But you have to assign everything in this variable. Because service will create instance by default and use that as a base object.

myApp.service('myService', function () {

// any logic here..

this.name = 'Joe';

}

Actual angularjs code behind the service

function service(name, constructor) {

return factory(name, ['$injector', function($injector) {

return $injector.instantiate(constructor);

}]);

}

It just a wrapper around the factory. If you return something from service, then it will behave like Factory.

IMPORTANT: The return result from Factory and Service will be cache and same will be returned for all controllers.

**When should i use them?**

Factory is mostly preferable in all cases. It can be used when you have constructor function which needs to be instantiated in different controllers.

Service is a kind of Singleton Object. The Object return from Service will be same for all controller. It can be used when you want to have single object for entire application. Eg: Authenticated user details.

<https://stackoverflow.com/questions/13762228/confused-about-service-vs-factory>

They are singletons means they are only created once.

Here are some more examples of services vs factories which may be useful in seeing the difference between them. Basically, a service has "new ..." called on it, it is already instantiated. A factory is not instantiated automatically.

Basic Examples

Return a class object which has a single method

Here is a service that has a single method:

angular.service('Hello', function () {

this.sayHello = function () { /\* ... \*/ };

});

Here is a factory that returns an object with a method:

angular.factory('ClassFactory', function () {

return {

sayHello: function () { /\* ... \*/ }

};

});

Return a value

A factory that returns a list of numbers:

angular.factory('NumberListFactory', function () {

return [1, 2, 3, 4, 5];

});

console.log(NumberListFactory);

A service that returns a list of numbers:

angular.service('NumberLister', function () {

this.numbers = [1, 2, 3, 4, 5];

});

console.log(NumberLister.numbers);

The output in both cases is the same, the list of numbers.

<https://stackoverflow.com/questions/15666048/angularjs-service-vs-provider-vs-factory>

**Service vs provider vs factory:**

I am trying to keep it simple. It's all about basic JavaScript concept.

First of all, let's talk about **services** in AngularJS!

**What is Service:** In AngularJS, **Service** is nothing but a singleton JavaScript object which can store some useful methods or properties. This singleton object is created per ngApp(Angular app) basis and it is shared among all the controllers within current app.

When Angularjs instantiate a service object, it register this service object with a unique service name. So each time when we need service instance, Angular search the registry for this service name, and it returns the reference to service object. Such that we can invoke method, access properties etc on the service object. You may have question whether you can also put properties, methods on scope object of controllers! So why you need service object? Answers is: services are shared among multiple controller scope. If you put some properties/methods in a controller's scope object , it will be available to current scope only. But when you define methods, properties on service object, it will be available globally and can be accessed in any controller's scope by injecting that service.

So if there are three controller scope, let it be controllerA, controllerB and controllerC, all will share same service instance.

<div ng-controller='controllerA'>

<!-- controllerA scope -->

</div>

<div ng-controller='controllerB'>

<!-- controllerB scope -->

</div>

<div ng-controller='controllerC'>

<!-- controllerC scope -->

</div>

**How to create a service?**

AngularJS provide different methods to register a service. Here we will concentrate on three methods factory(..),service(..),provider(..);

[Use this link for code reference](http://jsbin.com/vaken/16/edit?html,js,output)

**Factory function:**

We can define a factory function as below.

factory('serviceName',function fnFactory(){ return serviceInstance;})

AngularJS provides **'factory('serviceName', fnFactory)'** method which takes two parameter, serviceName and a JavaScript function. Angular creates service instance by invoking the function **fnFactory()** such as below.

var serviceInstace = fnFactory();

The passed function can define a object and return that object. AngularJS simply stores this object reference to a variable which is passed as first argument. Anything which is returned from fnFactory will be bound to serviceInstance . Instead of returning object , we can also return function, values etc, Whatever we will return , will be available to service instance.

Example:

var app= angular.module('myApp', []);

//creating service using factory method

app.factory('factoryPattern',function(){

var data={

'firstName':'Tom',

'lastName':' Cruise',

greet: function(){

console.log('hello!' + this.firstName + this.lastName);

}

};

//Now all the properties and methods of data object will be available in our service object

return data;

});

## Service Function:

service('serviceName',function fnServiceConstructor(){})

It's the another way, we can register a service. The only difference is the way AngularJS tries to instantiate the service object. This time angular uses 'new' keyword and call the constructor function something like below.

var serviceInstance = new fnServiceConstructor();

In the constructor function we can use 'this' keyword for adding properties/methods to the service object. example:

//Creating a service using the service method

var app= angular.module('myApp', []);

app.service('servicePattern',function(){

this.firstName ='James';

this.lastName =' Bond';

this.greet = function(){

console.log('My Name is '+ this.firstName + this.lastName);

};

});

## Provider function:

Provider() function is the another way for creating services. Let we are interested to create a service which just display some greeting message to the user. But we also want to provide a functionality such that user can set their own greeting message. In technical terms we want to create configurable services. How can we do this ? There must be a way, so that app could pass their custom greeting messages and Angularjs would make it available to factory/constructor function which create our services instance. In such a case provider() function do the job. using provider() function we can create configurable services.

We can create configurable services using provider syntax as given below.

/\*step1:define a service \*/

app.provider('service',function serviceProviderConstructor(){});

/\*step2:configure the service \*/

app.config(function configureService(serviceProvider){});

**How does provider syntax internally work?**

1.Provider object is created using constructor function we defined in our provider function.

var serviceProvider = new serviceProviderConstructor();

2.The function we passed in app.config(), get executed. This is called config phase, and here we have a chance to customize our service.

configureService(serviceProvider);

3.Finally service instance is created by calling $get method of serviceProvider.

serviceInstance = serviceProvider.$get()

Sample code for creating service using provide syntax:

var app= angular.module('myApp', []);

app.provider('providerPattern',function providerConstructor(){

//this function works as constructor function for provider

this.firstName = 'Arnold ';

this.lastName = ' Schwarzenegger' ;

this.greetMessage = ' Welcome, This is default Greeting Message' ;

//adding some method which we can call in app.config() function

this.setGreetMsg = function(msg){

if(msg){

this.greetMessage = msg ;

}

};

//We can also add a method which can change firstName and lastName

this.$get = function(){

var firstName = this.firstName;

var lastName = this.lastName ;

var greetMessage = this.greetMessage;

var data={

greet: function(){

console.log('hello, ' + firstName + lastName+'! '+ greetMessage);

}

};

return data ;

};

});

app.config(

function(providerPatternProvider){

providerPatternProvider.setGreetMsg(' How do you do ?');

}

);

**Summary:**

**Factory** use a factory function which return a service instance. **serviceInstance = fnFactory();**

**Service** use a constructor function and Angular invoke this constructor function using 'new' keyword for creating the service instance. **serviceInstance = new fnServiceConstructor();**

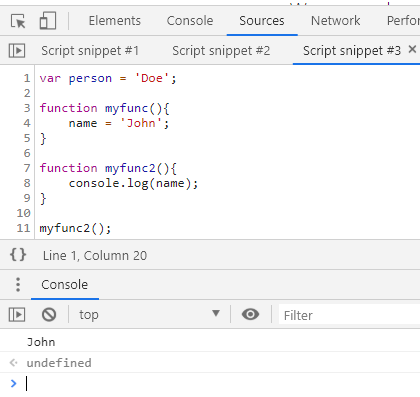
**Provider** defines a providerConstructor function, this providerConstructor function defines a factory function **$get** . Angular calls $get() to create the service object. Provider syntax has an added advantage of configuring the service object before it get instantiated. **serviceInstance = $get();**

innerHTML

innerHTML property sets or returns the HTML content of an element.

innerHTML is a DOM property to insert content to a specified id of an element. It is used in Javascript to manipulate DOM.

Quiz

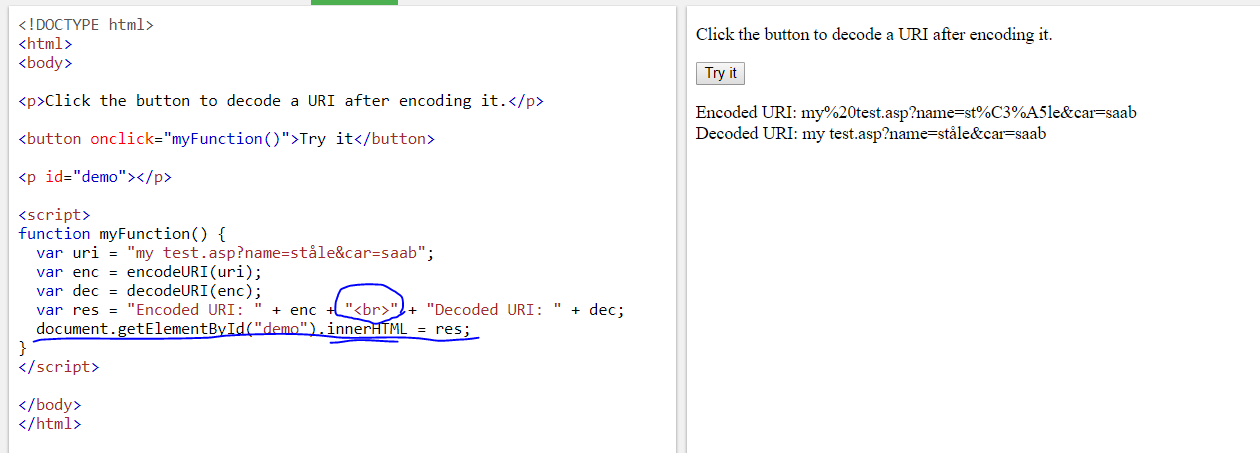
1. Which of the following is an example of anonymous function in JavaScript?
   1. var myFunc = function(){ };
2. A variable declared without var keyword inside a function will become **global** variable.
   1. 
3. What will be the output of the following JavaScript code?

x = 1;

console.log('x = ' + x);

var x;

**Correct Answer:** x = 1



**eval(string)**

**Parameters:**

**String:**

A string representing a JavaScript expression, statement, or

sequence of statements. The expression can include variables

and properties of existing objects.

**Return Value:**

The completion value of evaluating the given code is

returned by using **eval()**.

If the completion value is empty, **undefined** is returned.

Input : eval(new String('2 + 2'));

Output : returns a String object containing "2 + 2"

Input: var value = eval(new String(a \* b));

         document.write(value);

Output: 16

What is super in java?

The **super** keyword in Java is a reference variable that is used to refer to the immediate parent class object. Whenever you create the instance of the subclass, an instance of the parent class is created implicitly which is referred by super reference variable. The super() is called in the class constructor implicitly by the compiler if there is no super or this.

**Promise**

Understand promise like a progress bar.

The might not have finished yet, and you might not have access to the data to run an operation on, but you want a reference to that in progress task itself so that you could write logic around it.

Example: I want to run 3 separate tasks and then aggregate the results when all three come back and do something with the data, even though the tasks might finish at different times. This is kind of difficult to represent in callback model because you don’t have access to that in progress object.

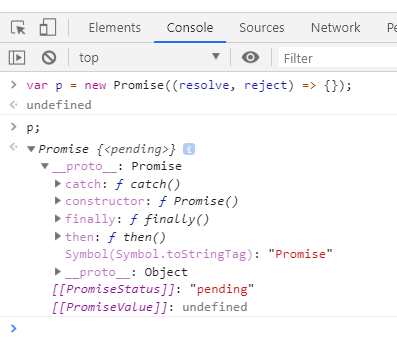
Promise object

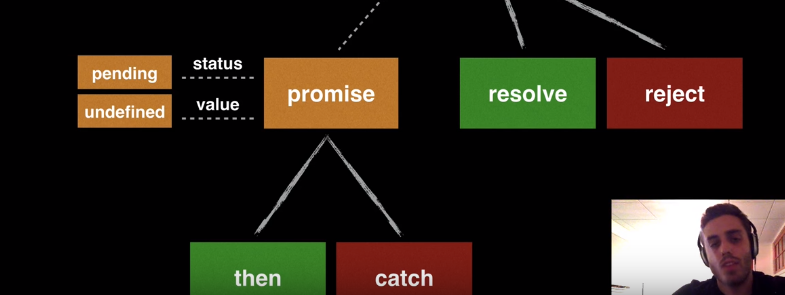
1. Two properties : *[[PromiseStatus]]* && *[[PromiseValue]]. When we create a promise, the*

*[[PromiseStatus]]*: "pending"

*[[PromiseValue]]*: undefined

1. Two functions that you can call – then and catch

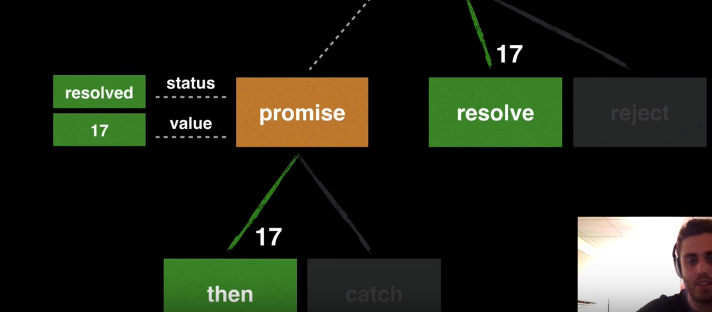




Resolve

What is resolve? If promise is progress bar then, resolve function will say this is no longer in progress, this is finished and got the value from our calculation. So it’s gonna tell everyone that was referencing that progress bar, hey! It’s not in progress anymore …. We have the value, you can go ahead with rest of your execution of your function.

When we resolve a promise, we resolve it with a value. Then that becomes value of that promise and then any function waiting for that value now uses that value to continue its execution.

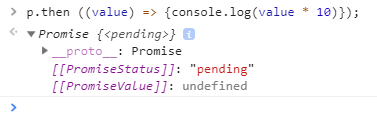


When we resolve a promise with some value say number 17, then it severed its connection with reject function. So if I call reject function after it is resolved then nothing will happen.

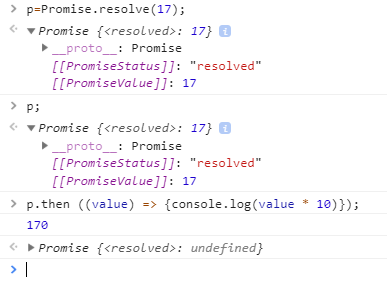
After we resolve a promise, the promise updates its status : resolved and it will update its value to 17. This will also trigger the “then” function with the argument of promise’s value passed in.

This will severe its connection with then function.

Now when I do : p.then ((value) => {console.log(value \* 10)});. Hit enter 🡪 The status is still pending, because then function is not triggered yet. Then function is only going to be triggered once promise is resolved.



When I call resolve,



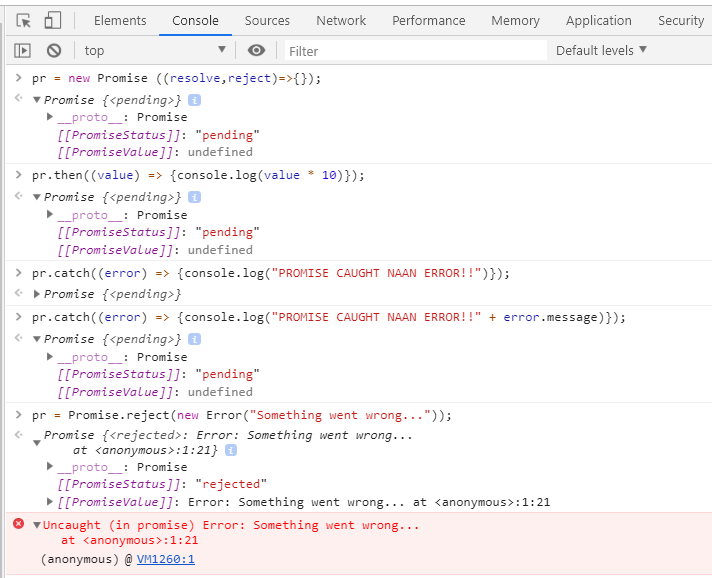
Reject

When you reject a promise, you can pass any value you want but normally you pass the JavaScript error object. Let say something went wrong … our server went down or something, we choose to reject the promise:

* Then resolve branch is severed. So, if you reject a promise then you can no longer ever resolve that promise.
* The status of promise updates to “rejected” and the value becomes that error object.
* We go down the catch and pass the error object in.

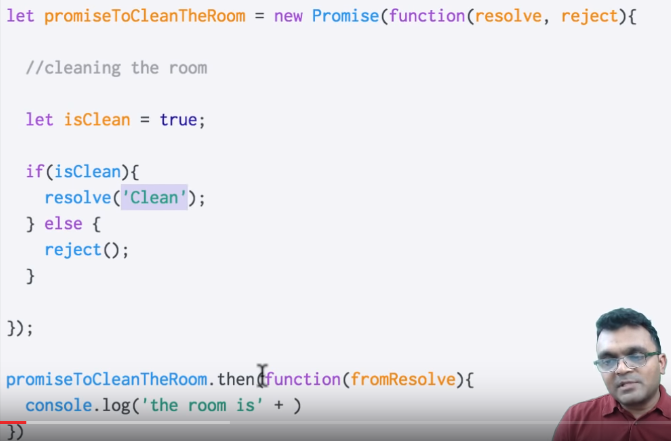
Now anything downstream that was listening to .then function knows … hey! This promise is errored out and you can no longer execute the then function you can catch that function and handle it.

1. I create a new promise object 🡪 pr = new Promise ((resolve,reject)=>{});
2. Add .then pr.then((value) => {console.log(value \* 10)}); 🡪 status still “pending”
3. Add .catch function pr.catch((error) => {console.log("PROMISE CAUGHT NAAN ERROR!!" + error.message)}); 🡪 status still “pending”
4. Reject the promise 🡪pr = Promise.reject(new Error("Something went wrong...")); Then,
5. *[[PromiseStatus]]*: "rejected"
6. *[[PromiseValue]]*: Error: Something went wrong... at <anonymous>:1:21



Promise is created with a callback function. Callback function takes two arguments --- resolve & reject. There are 3 parts in promise definition:

1. Task (which returns a token (true/false))
2. Token = true? Yes, Call resolve [ .then function is triggered]
3. Token = true? No, Call reject [ .catch function is triggered]







step1 = function() {

return new Promise(function(resolve,reject) {

resolve("Step 1. Boil Water ");

});

};

step2 = function(mes) {

return new Promise(function(resolve,reject) {

resolve(mes + ", Step 2. Add Tea & Sugar ");

});

};

step3 = function(mes) {

return new Promise(function(resolve,reject) {

resolve(mes + "Step 3. Serve hot Tea!!!");

});

};

step1().then(function(a){console.log(a); return step2(a);})

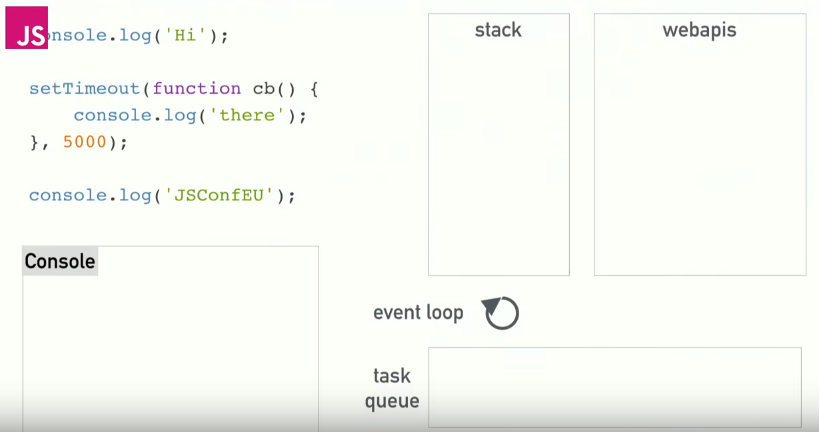
.then(function(b){console.log(b); return step3(b);}) // this "then" called with resolve message "b" from step1

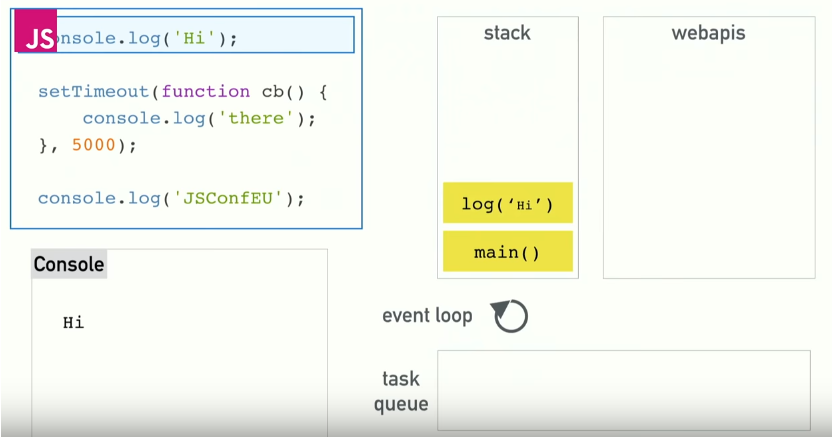
.then(function(c){ console.log("finished "+c)}); // this "then" called with resolve message "c" from step2

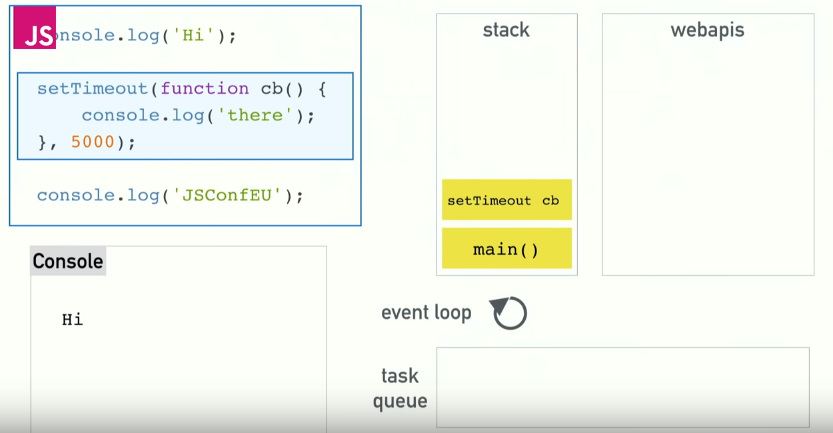
JavaScript Event Loop

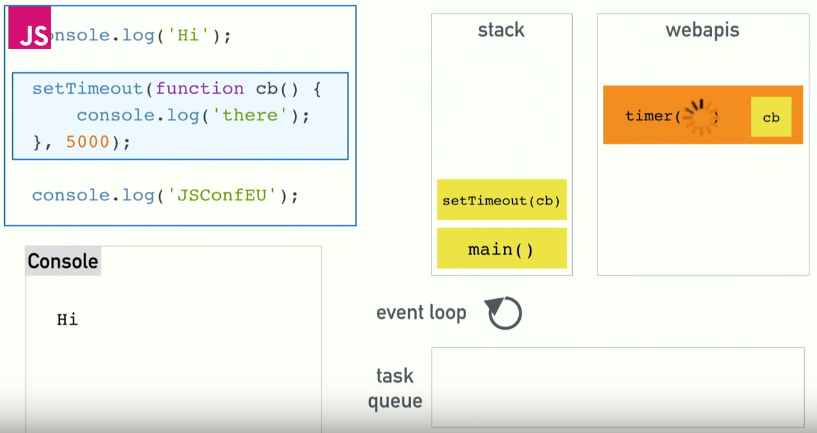
Async Call

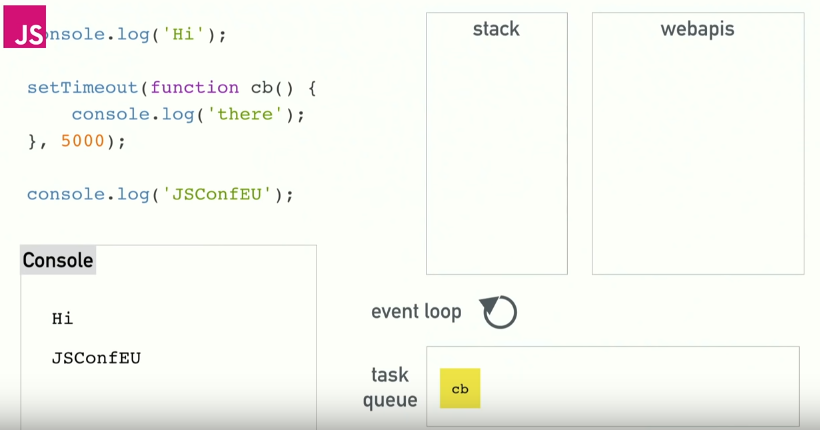
Examples of web APIs: setTimeout, AJAX requests, DOM (document)











Cb is picked up by event loop and moved from task queue to stack

