Promeses

¿Què és una promesa?

Una “promesa” o també anomenada “futuro” és un objecte que actua com a proxy en els casos en els que no es poden retornar el veritable valor perquè encara no es coneix però no es pot bloquejar la función esperant a que arribi. És una forma de fer les coses en comptes de les funcions de callback.

Anem a veure un exemple:
Al fer una crida Ajax amb $http, la anomenada a $http no retorna cap valor ja que encara no té aquest valor però tampoc es pot bloquejar esperant a que arribi, doncs en realitat el servei $http si que retorna un valor. Això que retorna és una promesa i, com hem dit, la promesa és un proxy que en un futur contindrà el valor.

Per què utilitzar promeses?

La promesa tindrà en un futur el valor, però, com ens assabentarem de que ara la promesa ja té el valor? Doncs utilitzant una funció de *callback*.
Realment no podem evitar les funcions de callback però si que podem evitar el que es diu la piràmide de la mort (en anglès Pyramid of Doom)). La piràmide de la mort es produeix quan dins d’una funció de callback fem una altre trucada asíncrona que a la seva vegada té una altre funció de callback i dins d’aquesta nova funció de callback fem una altre trucada asíncrona, i així successivament.

Podem veure com queda la piràmide de la mort en el següent exemple.

Obtenim el fitxer json “fichero1.json” i llegim el valor de “importe” però dins tornem a obtenir el fitxer “fichero2.json” i obtindrem el valor del import que sumarem al primer import, …..i així fins a 4 vegades.

```
$scope.importeTotal = 0;
$scope.mensajeFinal = "";

$http({method: 'GET',url: 'fichero1.json'}).success(function(data, status, headers, config) {
  $scope.importeTotal = $scope.importeTotal + data.importe;
  $http({method: 'GET',url: 'fichero2.json'}).success(function(data, status, headers, config) {
    $scope.importeTotal = $scope.importeTotal + data.importe;
    $http({method: 'GET',url: 'fichero3.json'}).success(function(data, status, headers, config) {
      $scope.importeTotal = $scope.importeTotal + data.importe;
      $http({method: 'GET',url: 'fichero4.json'}).success(function(data, status, headers, config) {
        $scope.importeTotal = $scope.importeTotal + data.importe;
        $scope.mensajeFinal = "Ya hemos finalizado la lista de cálculos";
      });
```

```
    });
  });
});
```

Anem a veure com estem afegint funcions dins de les funcions el que fa que el codi estigui cada vegada més indentada el que crea una forma de piràmide i dóna nom al problema.

El problema d'això és que el codi és poc modular, cada una de les funcions de callback està dins de la anterior, fent que separar-les sigui complex.

Anem a veure ara el mateix exemple utilitzant promeses

```
?
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
$scope.importeTotalPromesas = 0;
$scope.mensajeFinalPromesas="";

$http({ method: 'GET',url: 'fichero1.json'}).then(function(resultado) {
  $scope.importeTotalPromesas = $scope.importeTotalPromesas + resultado.data.importe;
  return $http({method: 'GET',url: 'fichero2.json'})
}).then(function(resultado) {
  $scope.importeTotalPromesas = $scope.importeTotalPromesas + resultado.data.importe;
  return $http({method: 'GET',url: 'fichero3.json'})
}).then(function(resultado) {
  $scope.importeTotalPromesas = $scope.importeTotalPromesas + resultado.data.importe;
  return $http({method: 'GET',url: 'fichero4.json'})
}).then(function(resultado) {
  $scope.importeTotalPromesas = $scope.importeTotalPromesas + resultado.data.importe;
  $scope.mensajeFinalPromesas = "Ya hemos finalizado la lista de cálculos con promesas";
})
```

Encara no hem explicat com funcionen les promeses, anem a veure en aquest exemple que ara no hi ha 4 funcions independents i que no estan niuades per el que és més senzill separar-les en funcions independents.

Hem vist en els controladors com fem trucades a $http per obtenir les dades. Això té un problema, la pàgina ja s'ha mostrat però encara estem esperant les dades. L'ideal seria que el controlador tingués les dades que necessita abans de mostrar-se la pàgina i abans d'executar-el controlador. D'aquesta manera no veuríem la pàgina amb dades en blanc esperant la resposta. Gràcies a les promeses i al servei de rutes de AngularJS això és possible.

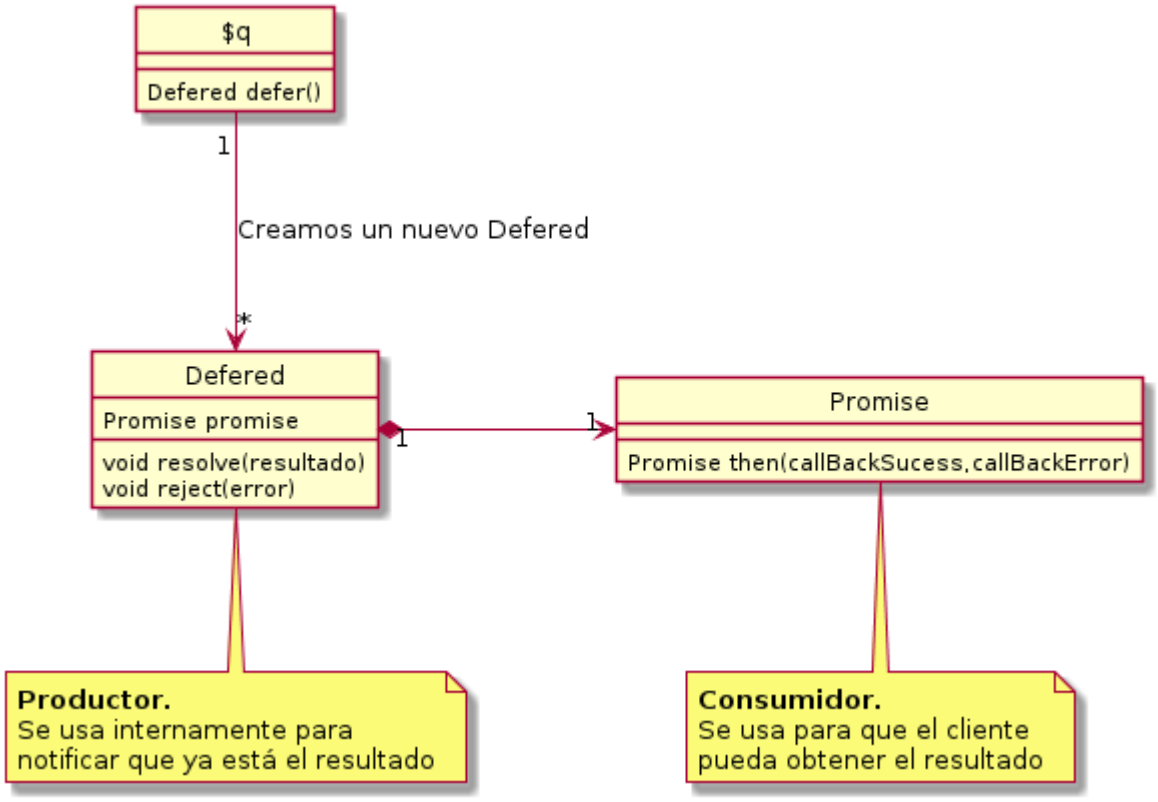
Veure ejercici1.html, ejercici1.js, ejercici91.json, ejercici92.json, ejercici93.json i ejercici94.json.

Promeses en Angular JS

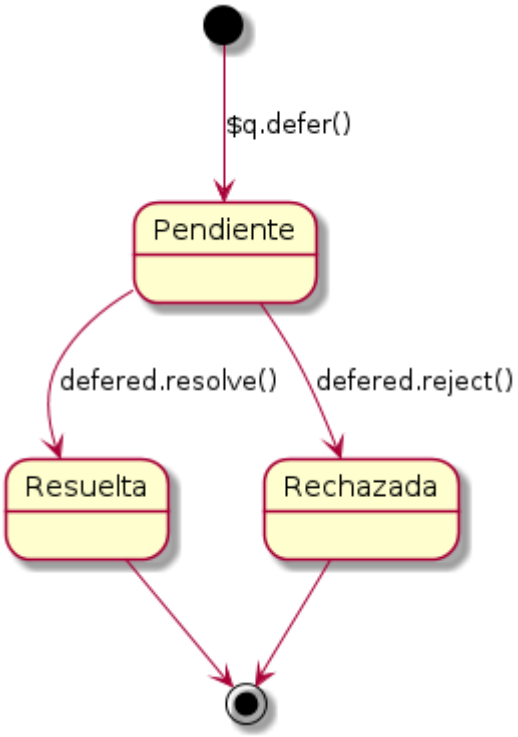
Servei $q

El servei de $q és un servei de AngularJS que conté tota la funcionalitat de les promeses. Tal com s'indica en la documentació, està basat en la implementació de Kris Kowal 's Q .. AngularJS ha fet la seva pròpia versió perquè estigui tot integrat en el propi framework. Jo acostumo a comparar el sistema de promeses al problema del productor-consumidor. La similitud és que hi ha una part que generarà la informació, per exemple el mètode $http i una altra part que consumirà la informació, per exemple el nostre codi. Aquesta separació és important ja que hi ha 2 objectes amb els que haurem de tractar. En la nomenclatura de AngularJS al productor se li anomena defered i al consumidor se li anomena promise. Mitjançant el servei de $q obtenim l'objecte defered cridant el mètode defer() i a partir d'ell obtenim l'objecte promise trucant a la propietat promise.

En el següent diagrama UML podem veure les diferents classes que formen part de les promeses:



Un cop creada mitjançant el mètode defer(), una promesa es pot trobar en algun dels següents 3 estats:



- **Pendent: Encara no se sap si es podrà o no obtenir el resultat.**
- **Resulta: S'ha pogut obtenir el resultat. S'arriba a aquest estat, trucant al mètode defered.resolve()**
- **Rebutjada: Hi ha hagut algun tipus d'error i no s'ha pogut obtenir el resultat. S'arriba a aquest estat, trucant al mètode defered.reject ()**

A causa de que la promesa a d'acabar en l'estat Pendent o Rebutjat, és obligatori que sempre cridem al mètode defered.resolve () o defered.reject ().

$q.defer()

Per explicar les promeses farem un exemple d'una funció que accepta com a paràmetre dos valors i de forma asíncrona els retorna. Òbviament per fer una simple suma no cal fer servir promeses ni retornar de manera asíncrona, però servirà com a exemple.

El primer és crear els 2 objectes que es necessiten en les promeses.

```
?
1  function sumaAsincrona(a,b) {
2      var defered=$q.defer();
3      var promise=defered.promise;
4
5      return promise;
6  }
```

- Línia 1: La funció accepta com a arguments 2 nombres, cridats a i b.
- Línia 2: Usant el servei de $q obtenim l'objecte defered trucant al métdo defer(). Per descomptat el servei $q ha de ser injectat d'alguna manera però aquí ho hem obviat per simplificar l'explicació.
- Línia 3: Des del objecte defer s'obté l'objecte promise.
- Línia 5: A qui ens crida li tornem la promesa. Ja tenim els 2 objectes preparats i llestos per ser usats. Ara explicarem més sobre cada un d'ells.

defered

L'objecte defered només s'usa des de dins de la funció asíncrona, per tant el que crida a sumaAsincrona no sap res de l'objecte defered. Com ja hem dit, l'objecte defered farà les funcions de productor de la informació.

Aquest objecte té 2 mètodes. Un d'ells per indicar que s'ha obtingut la informació i per tant fer que la promesa passi a l'estat "Resolt" i un segon mètode per indicar que alguna cosa ha fallat i que no s'ha pogut obtenir la informació i per tant fer que la promesa passi a l'estat "Rebutjat"..

Método	Parámetros	Descripción
`resolve`	`resolve(resultado)`	Anomenarem a aquest mètode per indicar que ja tenim la informació que es va sol·licitar i per tant que la promesa està resolta, sent el paràmetre resultat el que conté la informació sol·licitada.
`reject`	`reject(error)`	Anomenarem a aquest mètode per indicar que no ha estat possible obtenir la informació que es va sol·licitar i per tant que la promesa està rebutjada, contenint el paràmetre error informació relativa a la naturalesa de l'error.

Cal fixar-se que l'objecte defered no diu res relatiu al tipus d'informació que es retorna ni a l'estructura de la mateixa. Éso ja dependrà de cada un dels mètodes que creiem que usin promeses. Tampoc es permet retornar més d'una dada. Si volem retornar més d'una simplement hem de crear un objecte que contingui la informació que vulguem.
Seguim ara amb l'exemple i fem servir el servei de $ timeout per simular una resposta asíncrona de la funció.

```
function sumaAsincrona(a,b) {
    var defered=$q.defer();
    var promise=defered.promise;


    $timeout(function() {
        try{
            var resultado=a+b;
            defered.resolve(resultado);
        } catch (e) {
            defered.reject(e);
        }
    },3000);

```

```
13      return promise;
14  }
15
```

- Línia 5: Posem un timeout perquè passats 3 segons s'executi la funció.
- Línia 7: Aqui és on calculem la informació que sigui necessària. L'únic que fem és sumar els 2 valors que ens van passar.
- Línia 8: Indiquem que tot ha anat bé i retornem el resultat.
- Línia 10: Si alguna cosa ha fallat ho indiquem trucant a reject. En aquest cas passem com a valor de retorn la pròpia excepció que s'ha generat.

promise

Acabem de veure el que cal fer internament per produir la informació en la funció sumaAsincrona. Ara passem a l'altra banda del problema. Vegem què passa en cridar a la nostra funció asíncrona, és a dir en la part del consumidor.

El primer que hem de fer és cridar a la funció asíncrona i guardar-nos la promesa que ens retorna.

```
1  var promise=sumaAsincrona(5,2);
```

- Línia 1: Trucar a la funció asíncrona i guardar-nos la promesa que ens retorna.Com podem saber ara si s'ha pogut o no obtenir la dada? I com obtenim la dada? L'objecte promise té un mètode anomenat then, el qual accepta que li passem 2 funcions de callback per saber el que ha passat. La primera funció es dirà si s'ha obtingut la informació i per tant si la promesa ha estat resolta. La segona funció es cridarà si alguna cosa ha fallat i per tant si la promesa ha estat rebutjada.

```
1  var promise=sumaAsincrona(5,2);
2
3  promise.then(function(resultado) {
4    $scope.mensaje="El resultado de la promesa es:" + resultado;
5  }, function(error) {
6    $scope.mensaje="Se ha producido un error al obtener el dato:"+error;
7  });
```

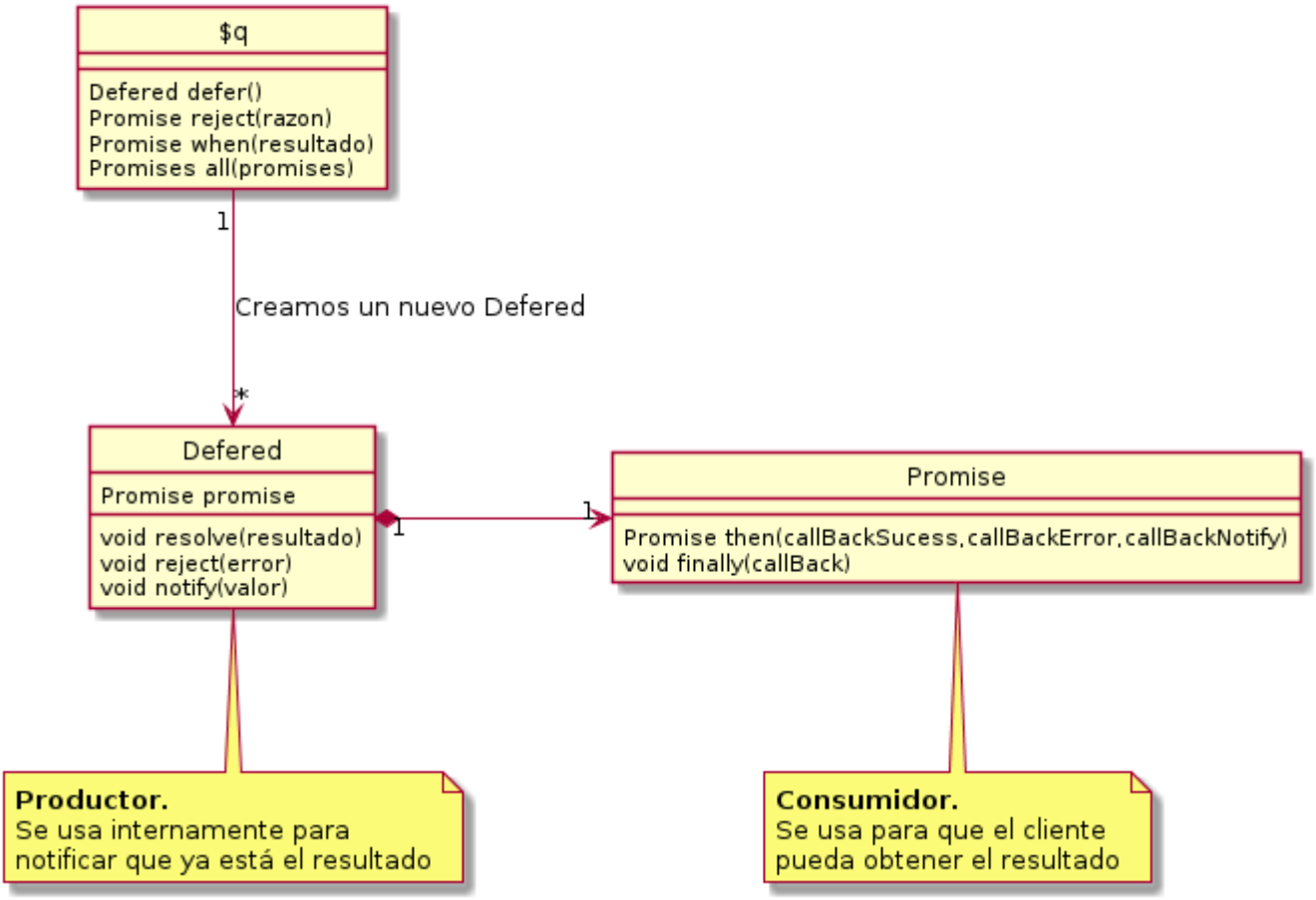
- Línia 3: Fem una crida al mètode then de l'objecte promise i li vam passar les 2 funcions.
- Línia 4: La promesa ha estat resolta i aquí ens guardem al $scope el resultat que ens ha arribat a través de la promesa.
- Línia 5: Especifiquem la funció d'error per quan no es pot obtenir la informació.
- Línia 6: La promesa ha estat rebutjada ja que alguna cosa ha fallat i en aquest cas generem a missatge d'error.

Veure ejercici2.html i ejercici2.js.

Promesas Avançades

Acabem de veure com funcionen les promeses. Per a moltes aplicacions serà necessari només el que acabem de veure, però, AngularJS proporciona més mètodes que ens poden ser útils en certs casos. Passem ara a veure mes funcionalitats de les promeses.

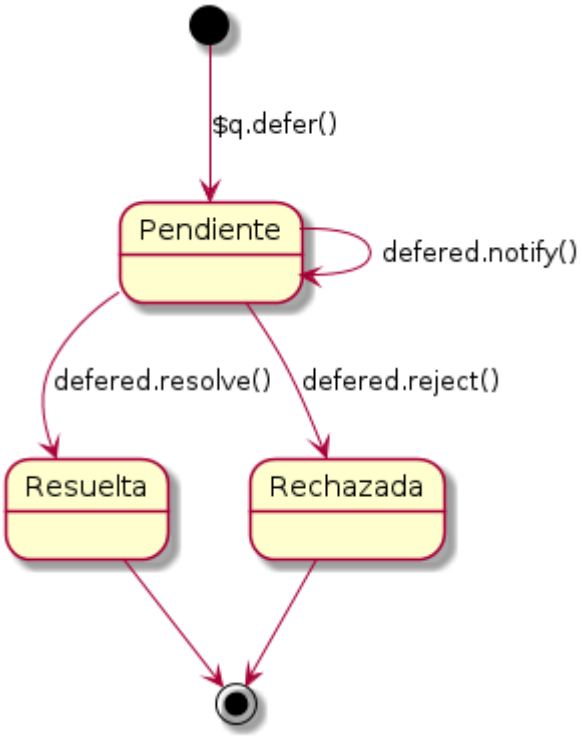
El primer que farem és actualitzar el diagrama de classes d'UML per reflectir tots els nous mètodes que explicarem.



Notificaciones

És possible que mentres la funció asíncrona a la qual hem anomenat està calculant el resultat ens pugui notificar el seu propi progrés. Per això la classe defered té un altre mètode anomenat notify al que li passem un valor. Aquest valor pot ser recollit des de la promesa afegint una tercera funció de callback al mètode then de promise.

El diagrama d'estats de la promesa ara es modifica de la manera següent:



Veiem ara que es pot trucar a defered.notify () totes les vegades que es vulgui i la promesa seguirà en l'estat Pendent.

No és possible trucar a defered.notify () un cop s'hagi resolt la promesa però tampoc abans que es truqui al mètode promise.then.Ésto últim implica que no es pot cridar dins de la funció abans de retornar l'objecte de la classe Promise.

L'exemple del tema anterior l'hem modificat per a incloure un nou text amb el % de progrés de l'operació.

```
1   <!DOCTYPE html>
2   <html ng-app="app">
3     <head>
4       <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.min.js"></script
5       <script src="//code.angularjs.org/1.2.19/i18n/angular-locale_es-es.js"></script>
6       <script src="script.js"></script>
7     </head>
8     <body ng-controller="PruebaController">
9       {{mensaje}}
10      <br>
11      {{progreso}}
```

```
12      </body>
13  </html>
```

- Línea 11: Mostramos el % de progreso de la operación.

```
1   function sumaAsincrona(a, b) {
2     var defered = $q.defer();
3     var promise = defered.promise;
4
5
6     $timeout(function() {
7       defered.notify(0);
8     }, 1);
9     $timeout(function() {
10      defered.notify(33);
11    }, 1000);
12    $timeout(function() {
13      defered.notify(66);
14    }, 2000);
15
16    $timeout(function() {
17      try{
18        var resultado = a + b;
19        defered.notify(100);
20        defered.resolve(resultado);
21      } catch(e) {
22        defered.reject(e);
23      }
24    }, 3000);
25    return promise;
  }
```

```
26
27  var promise = sumaAsincrona(5, 2);
28
29  promise.then(function(resultado) {
30    $scope.mensaje = "El resultado de la promesa es:" + resultado;
31  }, function(error) {
32    $scope.mensaje = "Se ha producido un error al obtener el dato:" + error;
33  }, function(progreso) {
34    $scope.progreso = progreso+"%";
35  });
36
```

- Línies 6,7 i 8: S'ha posat un timeout perquè en passar un mil·lisegon es notifiqui que anem pel 0%.
- Línies 9,10 i 11: Es notifica que anem pel 33% en passar 1000 ms. Línies 12,13 i 14: Es notifica que anem pel 66% al passar 2000 ms.
- Línia 19: Ja hem acabat els càlculs i notifiquem que anem pel 100%. Recorda que aquesta línia cal posar-la abans de defered.resolve ja que, en cas contrari, mai es notificaria.
- Línies 34 i 35: S'afegeix la tercera funció al mètode then la qual serà anomenada cada vegada que ens notifiquin alguna cosa sobre el progrés de l'operació. En aquest cas afegim el caràcter "%" i el posem al% scope perquè es mostri a la pàgina HTML.

Finalizador

Hi ha vegades que ens interessa que s'executi un codi quan es resol la promesa independentment de si ha estat resolta o rebutjada. Per a això la classe Promise disposa d'un mètode anomenat finally que accepta com a argument una funció de callback. Aquesta funció de callback s'anomenarà just abans de cridar a les funcions definides en el mètode then.

A causa de que el mètode es diu finally i aquesta és una paraula reservada en javaScript, pot ser que el navegador no et deixi cridar-la directament així que s'haurà de trucar usant la següent manera:

```
1  promise["finally"](function() {
2      alert("Mensaje del ejemplo.Ésto se llama justo antes de resolver o rechazar la promesa:
3  });
```

- Línia 2: S'executarà just abans de resoldre o rebutjar la promesa, de manera que sempre s'executarà.

Realitzar una promesa de un valor

Un altre cas que pot passar és que la funció a la que anomenem en lloc d'haver de calcular asíncronament el resultat ja el tingui disposible. Un cas molt típic és una crida $ http en què el resultat està cachejat. En aquest cas no tindria sentit fer servir una promesa ja que ja tenim el resultat en el moment de cridar a la funció. No obstant això, com la interfície de la nostra funció no el podem modificar, cal utilitzar una promesa.

Per a aquests casos AngularJS ofereix el mètode $q.when que permet transforma qualsevol valor en una promesa.

Seguint el nostre exemple hem modificat la funció sumaAsincrona perquè quan tots dos arguments són zero, no sigui necessari fer el "càlcul" sinó directament retornem el zero.

```
1   if((a===0) && (b===0)) {
2     return $q.when(0);
3   }
```

- Línia 2: Retornem una promesa que el resultat ja és zero.

Promesas en paralelo

Una altra mètode útil de $ q és all. Aquest mètode permet retornar una única promesa que unifica diverses promeses. Això s'utilitza per quan volem que diverses promeses s'executin en paral·lel i volem esperar fins que totes elles estiguin resoltes.

Hi ha 2 formes de fer això:

- Array de promesas
- Objeto con promesas

Array de promesas

El que fem és obtenir totes les promeses i afegir-les totes a un array. Llavors vam cridar al mètode $ q.all (array) i li vam passar la matriu que conté totes les promeses. Aquest mètode ens retornarà la promesa unificada, la qual només es resoldrà si totes es resolen. Si alguna de les promeses inicials és rebutjada, la promesa unificada també serà rebutjada.

En haver diverses promeses la manera de funcionar canvia de la següent manera:

- En resoldre la promesa, el paràmetre és un array amb cada un dels valors.
- Si alguna de les promeses inicials és rebutjada, a la promesa unificada només li arriba un únic rebuig que serà el de la primera promesa que hagi estat rebutjada. La resta dels rebutjos no es notificaran.
- No es generen missatges de notificació en la promesa unificada.
- Si algun dels elements de l'array de promeses no és una promesa, llavors es dirà a $ q.when per transformar el valor en una promesa.

En el següent exemple podem veure com s'unifiquen diverses promeses en una sola usant un array:

```
1   $scope.mensajeMultipleArray = "Esperando a una promesa múltiple formada por muchas promesa
2   var promesaMultipleArray=$q.all([sumaAsincrona(0,0),sumaAsincrona(1,0),sumaAsincrona(2,0)]
```

```
3
4    promesaMultipleArray.then(function(resultado) {
5      $scope.mensajeMultipleArray = "El resultado de la promesa múltiple formada por muchas
     resultado[2];
6    }, function(error) {
7      $scope.mensajeMultipleArray ="Se ha producido un error en alguna de las multiples promes
8    }, function(progreso) {
9      $scope.progresoMultipleArray = progreso+"%";
10   });
```

- Línia 2: Creem una nova promesa anomenada promesaMultiple que és un array de la unió de les 3 promeses.
- Línia 5: L'argument resultat és un array amb els 3 resultats de les 3 promeses.
- Línia 7: Només serà anomenat una única vegada per a la primera promesa que sigui rebutjada.
- Línia 9: Mai s'anomenarà a aquest codi ja que mai es notifica res en les promeses múltiples.

Objeto con promesas

Lo que hacemos es obtener todas las promesas y añadirlas todas a un objeto, donde cada propiedad del objeto debe ser una promesa. Entonces llamamos al método `$q.all(objeto)` y le pasamos el objeto que contiene las promesas. Este método nos retornará la promesa unificada la cual sólo se resolverá si todas se resuelven. Si alguna de las promesas iniciales es rechazada, la promesa unificada también será rechazada.
Al haber varias promesas la forma de funcionar cambia de la siguiente forma:

- Al resolver la promesa, el parámetro es un objeto cuyas propiedades son cada uno de los valores.
- Si alguna de las promesas iniciales es rechazada , a la promesa unificada sólo le llega un único rechazo, que será el de la primera promesa que haya sido rechazada. El resto de los rechazos no se notificarán.
- No se generan mensajes de notificación en la promesa unificada.
- Si alguna de las propiedades del objeto de promesas no es una promesa , entonces se llamará a `$q.when` para transformar el valor en una promesa.

En el siguiente ejemplo podemos ver cómo se unifican varias promesas en una sola usando un objeto:

```
?
1  $scope.mensajeMultipleObjetos = "Esperando a una promesa múltiple formada por muchas promesa
2  var promesaMultipleObjetos=$q.all({
3    promesaA : sumaAsincrona(0,0),
4    promesaB : sumaAsincrona(1,0),
5    promesaC : sumaAsincrona(2,0)
```

```
6  });
7
8  promesaMultipleObjetos.then(function(resultado) {
9    $scope.mensajeMultipleObjetos = "El resultado de la promesa múltiple formada por mucha
   resultado.promesaC;
10 }, function(error) {
11   $scope.mensajeMultipleObjetos ="Se ha producido un error en alguna de las multiples promes
12 }, function(progreso) {
13   $scope.progresoMultipleObjetos = progreso+"%";
14 });
```

- Línea 2: Creamos una nueva promesa llamada `mensajeMultipleObjetos` que es la unión de las 3 promesas. Cada promesa es un propiedad del objeto que hemos creado.
- Línea 9: El argumento `resultado` es un **objeto** con los 3 resultados de las 3 promesas. Cada propiedad coincide con el nombre de la propiedad del objeto que contenía las promesas.
- Línea 11: Sólo será llamado una única vez para la primera promesa que sea rechazada.
- Línea 13: Nunca se llamará a este código ya que nunca se notifica nada en las promesas múltiples.

Esta forma de usar las promesas la volveremos a ver en Resolve

Promesas encadenadas

El último tema que queda por ver es la posibilidad de encadenar resultados de promesas.Como vimos al principio de esta unidad, es lo más importante de las promesas ya que permite evitar la *pirámide de la muerte*.
La forma de conseguir ésto es una nueva funcionalidad del método `then` de las promesas. Resulta que las 2 primeras funciones de *callback* del método `then` permiten que se retorne una promesa [3)] y entonces dicha promesa la retorna el método `then`.
En principio no parece que ésto sea una gran avance pero veamos un ejemplo de cómo funciona.

El ejemplo consiste en gracias a la función `sumaAsincrona` calcular la potencia de (2^4) es decir el valor 16. Para ello llamamos a `sumaAsincrona` 4 veces sumando a sí mismo el valor de la llamada anterior.

- Llamar a `sumaAsincrona(1,1)` lo que da un resultado de 2. Usaremos el resultado para la siguiente llamada.
- Llamar a `sumaAsincrona(2,2)` lo que da un resultado de 4. Usaremos el resultado para la siguiente llamada.
- Llamar a `sumaAsincrona(4,4)` lo que da un resultado de 8. Usaremos el resultado para la siguiente llamada.
- Llamar a `sumaAsincrona(8,8)` lo que da un resultado de 16. Ya tenemos el resultado.

Como vemos cada llamada a `sumaAsincrona` se hace con el resultado de la llamada anterior, por lo tanto las llamadas están encadenadas.
Si vemos el código siguiente, gracias a las promesas no es necesario tener función de *callback* que dentro tiene otra función de *callback* y así sucesivamente sino que son funciones de *callback* independientes unas de otras. Sí están relacionadas pero no es necesario anidarlas.
?

```
1   $scope.mensajeEncadenada="Esperando el resultado de promesas encadenadas, esto tardará 12
2   var promesasEncadenadas=sumaAsincrona(1,1);
3
4   promesasEncadenadas.then(function(resultado) {
5     return sumaAsincrona(resultado,resultado);
6   }).then(function(resultado) {
7     return sumaAsincrona(resultado,resultado);
8   }).then(function(resultado) {
9     return sumaAsincrona(resultado,resultado);
10  }).then(function(resultado) {
11    $scope.mensajeEncadenada="El resultado de las promesas encadenadas es:" + resultado;
12  });
```

Cada función de *callback* que resuelve la promesa a su vez retorna otra promesa. Ese objeto promesa se retorna en la función `then` lo que permite encadenar otras funciones de *callback* y así todas las veces que queramos sin que esté ninguna función de *callback* anidada con ninguna otra.

- Línea 2: Se genera la primera promesa.
- Línea 5: En esta función de *callback* se coge el resultado que es 2 y se suma a sí mismo y se retorna otra promesa.
- Línea 7: En esta función de *callback* se coge el resultado que es 4 y se suma a sí mismo y se retorna otra promesa.
- Línea 9: En esta función de *callback* se coge el resultado que es 8 y se suma a sí mismo y se retorna otra promesa.
- Línea 11: Finalmente en la última función de *callback* se muestra el resultado que es de 16.

Recuperarse de un error

Hemos visto cómo se generan las llamadas si todo funciona correctamente, si falla alguna función se detendrá la cadena de llamadas y ya está . Pero hay una forma de poder tratar el error y seguir con la cadena de llamadas. Podemos añadir una función de *callback* para cuando se produce un error y si dicha función de error retorna una promesa [4] se seguirá la cadena de llamadas como si nada hubiera fallado.
Vamos a modificar nuestro ejemplo para incluir una función en caso de que algo falle en la segunda llamada.

```
?
    $scope.mensajeEncadenada="Esperando el resultado de promesas encadenadas, esto tardará 12
1
    var promesasEncadenadas=sumaAsincrona(1,1);
2
3
    promesasEncadenadas.then(function(resultado) {
4
      return sumaAsincrona(resultado,resultado);
```

```
5     }).then(function(resultado) {
6       return sumaAsincrona(resultado,resultado);
7     },function(error) {
8       return 4;
9     }).then(function(resultado) {
10      return sumaAsincrona(resultado,resultado);
11    }).then(function(resultado) {
12      $scope.mensajeEncadenada="El resultado de las promesas encadenadas es:" + resultado;
13    });
14
```

- Línea 8: Se añade la función de *callback* para cuando algo falla.
- Línea 9: Arreglamos el *error* y retornamos el valor que debería ser , que en este caso es un 4.Al retornar el valor se sigue con las promesas encadenadas como si nada hubiera pasado.

El ejemplo es un poco naíf ya que en caso de fallo retornamos directamente el valor correcto. En un caso real sería más complejo o quizás imposible corregir el fallo y retornar otra promesa. Pero aún así sirve para explicar cómo la función de fallo permite seguir con la cadena de promesas.

Fallo

A diferencia del ejemplo anterior no es normal que podemos arreglar el resultado cuando algo ha fallado sino que simplemente queremos enterarnos si ha fallado algo de la cadena de promesas. AngularJS permite de una forma sencilla enterarnos si algo ha fallado. Simplemente ponemos una función de fallo en la última promesa y si alguna de ellas falla se llamará a dicha función.

```
?
1     $scope.mensajeEncadenada="Esperando el resultado de promesas encadenadas, esto tardará 12
2     var promesasEncadenadas=sumaAsincrona(1,1);
3
4     promesasEncadenadas.then(function(resultado) {
5       return sumaAsincrona(resultado,resultado);
6     }).then(function(resultado) {
7       return sumaAsincrona(resultado,resultado);
8     },function(error) {
9       return 4;
10    }).then(function(resultado) {
      return sumaAsincrona(resultado,resultado);
```

```
11  }).then(function(resultado) {
12    $scope.mensajeEncadenada="El resultado de las promesas encadenadas es:" + resultado;
13  },function(error) {
14    $scope.mensajeEncadenada="Se ha producido un error en alguna de las promesas:"+error;
15  });
16
```

- Línea 14: Definimos una función de fallo en la última promesa de la cadena de promesas.
- Línea 15: Se ejecutará si alguna de las promesas ha fallado.

Por supuesto podría interesarnos saber el paso concreto que ha fallado, en cuyo caso tendremos que añadir tantas funciones de fallo como promesas haya.

Progreso

Otra característica que tienen las promesas encadenadas es que permiten también añadir una función de *callback* en la última promesa por lo que llamará para todas las promesas de la cadena.
El ejemplo lo volvemos ahora a modificar añadiendo dicha función:

```
1   $scope.mensajeEncadenada="Esperando el resultado de promesas encadenadas, esto tardará 12
2   var promesasEncadenadas=sumaAsincrona(1,1);
3
4   promesasEncadenadas.then(function(resultado) {
5     return sumaAsincrona(resultado,resultado);
6   }).then(function(resultado) {
7     return sumaAsincrona(resultado,resultado);
8   },function(error) {
9     return 4;
10  }).then(function(resultado) {
11    return sumaAsincrona(resultado,resultado);
12  }).then(function(resultado) {
13    $scope.mensajeEncadenada="El resultado de las promesas encadenadas es:" + resultado;
14  },function(error) {
15    $scope.mensajeEncadenada="Se ha producido un error en alguna de las promesas:"+error;
16  },function(progreso) {
17    $scope.progresoEncadenada=progreso+"%";
    });
```

- Línea 14: Definimos una función de progreso en la última promesa de la cadena de promesas.
- Línea 15: Se ejecutará cada vez que alguna de las promesas notifica algo.En nuestro ejemplo al haber 4 llamadas se verá : 0% 33% 66% 100%, 0% 33% 66% 100%, 0% 33% 66% 100%, 0% 33% 66% 100%.

Por supuesto podría interesarnos saber por separado las notificaciones de cada promesa,y en ese caso tendremos que añadir tantas funciones de notificación como promesas haya.