JavaScript

Sesión 8 - **Promesas**

Índice

8.1 ¿Qué es una promesa?

- Objeto que representa un evento único
  - normalmente como resultado de una tarea asíncrona como una llamada AJAX.
- Almacenan un **valor futuro**
  - resultado de una petición HTTP, lectura de un fichero desde disco, etc…

- Permiten escribir código más sencillo, *callbacks* más cortos, y mantener la lógica de la aplicación de alto nivel separada de los comportamientos de bajo nivel.
- Permiten usar *callbacks* en cualquier situación, y no solo con eventos.
- Ofrecen un mecanismo estándar para indicar la finalización de tareas.

8.2 *Promise API*

- *ECMAScript 6*
- http://caniuse.com/#feat=promises
- *jQuery* implementa las promesas mediante los *Deferreds*
- Librerías de terceros:
  - *BlueBird* (https://github.com/petkaantonov/bluebird)
  - Q (https://github.com/kriskowal/q)

Hola Promesa

```
var promesa = new Promise(function(resolver, rechazar) {
  var ok;

  // código con la llamada async

  if (ok) {
    resolver("Ha funcionado"); // resuelve p
  } else {
    rechazar(Error("Ha fallado")); // rechaza p
  }
});
```

- *Revealing Constructor* → sólo los callbacks `resolver` y `rechazar` pueden modificar el estado interno

… *then*

- `then(callbackResuelta, callbackRechazada)`
  - ambos *callbacks* son opcionales
- *thenable* → objeto similar a una promesa, ya que contiene el método `then`

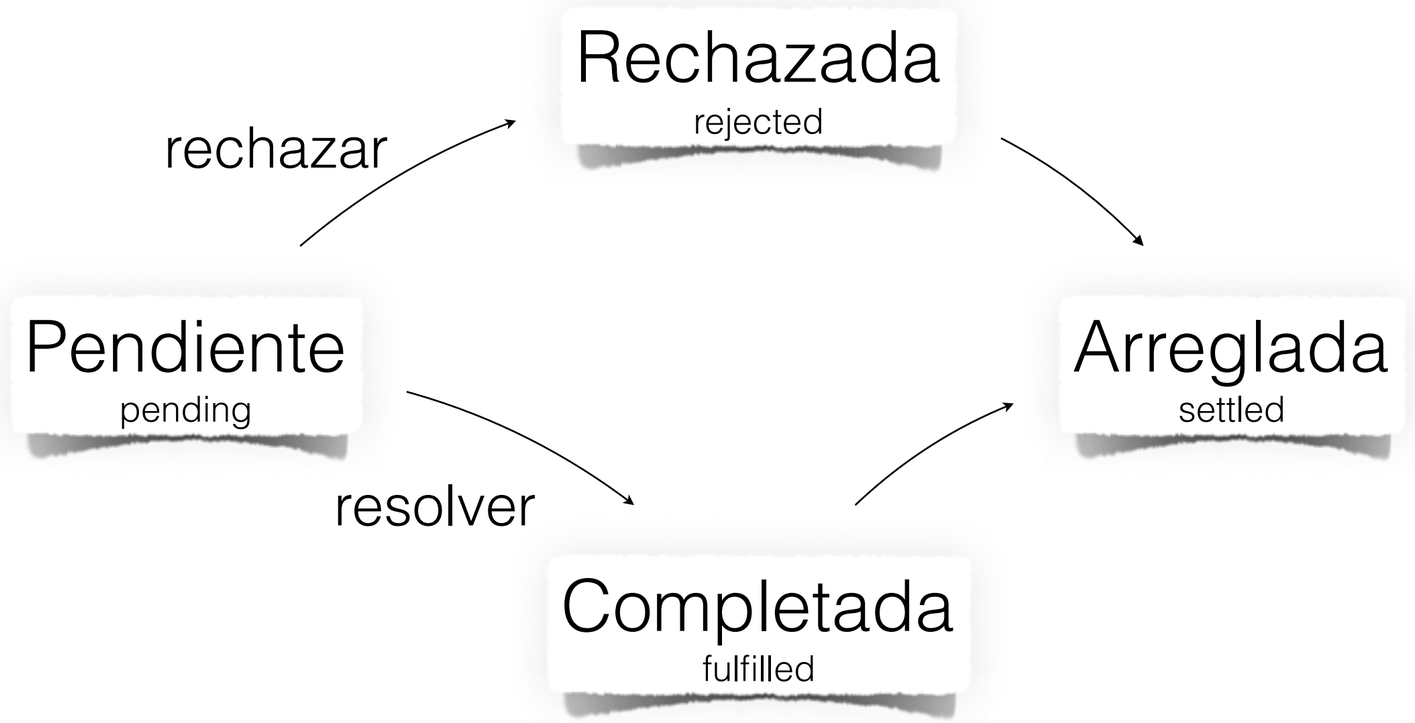
```
promesa.then(
  function(resultado) {
    console.log(resultado); // "Ha funcionado"
  }, function(err) {
    console.error(err); // Error: "Ha fallado"
  }
);
```

```
var p = $.get("http://www.omdbapi.com/?t=Interstellar&r=json");

p.then(function(resultado) {
  console.log(resultado);
});
```

Estados

- Una vez una promesa se completa o rechaza, se mantendrá en dicho estado (*settled*)
- Este estado es inmutable
- Se puede observar tantas veces como queramos.



Autoevaluación

- ¿Qué saldrá por consola?

```
var promesa = new Promise(function (resolver, rechazar) {
  resolver(Math.PI);
  rechazar(0);
  resolver(Math.sqrt(-1));
});

promesa.then(function (num) {
  console.log("El número es " + num)
})
```

Completar y resolver

- completada != resuelta

- **El argumento que se pasa a `resolver` decide el destino de la promesa.**

- cuando a la función `resolver` se le pasa:
  - un valor, la promesa se completa automáticamente.
  - otra promesa (por ejemplo `promesa.resolver(otraPromesa)`), las promesas se unen para crear una única promesa → su estado depende de la última promesa

    Cuando se resuelve la 2ª promesa (`otraPromesa`), ambas promesas se resolverán.

    Si se rechaza la 2ª promesa, las dos promesas se rechazarán.

`resolver` y `rechazar` inmediatamente

- Tanto resolver como rechazar se pueden llamar sin argumentos
  - El valor de la promesa será `undefined`.

- Para crear una promesa que inmediatamente se resuelva o rechace: `Promise.resolve()` o `Promise.reject()`:

```
new Promise(function(resolver, rechazar) {
  resolver("forma larga");
});
Promise.resolve("forma corta");

new Promise(function(resolver, rechazar) {
  reject("rechazo larga");
});
Promise.reject("rechazo corta");
```

8.2.4 Consumiendo promesas

- Podemos adjuntar tantos callbacks a una promesa como queramos
  - se ejecutarán una vez que la promesa se resuelva o rechace.

```
var usuario = {
  perfilUsuario: null,

  obtenerPerfil: function() {
    if (!this.perfilUsuario) {
      var xhr = new XMLHttpRequest();
      xhr.open("GET","usuario.json", true);
      xhr.onreadystatechange = function() {
        if (xhr.readyState ===  4) {
          perfilUsuario = JSON.parse(xhr.responseText);
        }
      };
      xhr.send(null);
    }
  }
};

usuario.obtenerPerfil();
if (usuario.perfilUsuario) {
  document.getElementById("navbar").innerHTML = usuario.login;
  document.getElementById("titulo").innerHTML = usuario.nombre;
}
```

... AJAX y presentación acopladas...

```
var usuario = {
  perfilUsuario: null,

  obtenerPerfil: function() {
    if (!this.perfilUsuario) {
      var xhr = new XMLHttpRequest();
      xhr.open("GET","usuario.json", true);
      xhr.onreadystatechange = function() {
        if (xhr.readyState ===  4) {
          perfilUsuario = JSON.parse(xhr.responseText);
          document.getElementById("navbar").innerHTML = perfilUsuario.login;
          document.getElementById("titulo").innerHTML = perfilUsuario.nombre;
        }
      };
      xhr.send(null);
    }
  }
};

usuario.obtenerPerfil();
```

... prometiendo AJAX...

```
var usuario = {
  promesaUsuario: null,

  obtenerPerfil: function() {
    if (!this.promesaUsuario) {
      this.promesaUsuario = new Promise(function(resolver, rechazar) {
        var xhr = new XMLHttpRequest();
        xhr.open("GET","usuario.json", true);
        xhr.onreadystatechange = function() {
          if (xhr.readyState ===  4) {
            resolver(JSON.parse(xhr.responseText));
          }
        };
        xhr.onerror = function() {
          rechazar(Error("Error obtener usuario"));
        };
        xhr.send(null);
      });
    }
    return this.promesaUsuario;
  }
};
```

```
var navbar = {
  mostrar: function(usuario) {
    usuario.obtenerPerfil().then(function(perfil) {
      document.getElementById("navbar").innerHTML=perfil.login;
    });
  }
}

var titulo = {
  mostrar: function(usuario) {
    usuario.obtenerPerfil().then(function(perfil) {
      document.getElementById("titulo").innerHTML=perfil.nombre;
    });
  }
}

navbar.mostrar(usuario);
titulo.mostrar(usuario);
```

...y devolviendo promesas

- ¿Y si queremos realizar una acción tras mostrar los datos del usuario?

- **Cualquier función que utilice una promesa debería devolver una nueva promesa**

```
// ...
var titulo = {
  mostrar: function(usuario) {
    return usuario.obtenerPerfil().then(function(perfil) {
      document.getElementById("titulo").innerHTML = perfil.nombre;
    });
  }
}
```

8.2.5 Prometiendo `XMLHttpRequest`

```
function ajaxUA(url) {
  return new Promise(function(resolver, rechazar) {
    var req = new XMLHttpRequest();
    req.open('GET', url);

    req.onload = function() {
      if (req.status == 200) {
        resolver(req.response);
      } else {
        rechazar(Error(req.statusText));
      }
    };

    req.onerror = function() {
      reject(Error("Error de Red"));
    };

    req.send();
  });
}
```

```
ajaxUA('heroes.json').then(function(response) {
  console.log("¡Bien!", response);
}, function(error) {
  console.error("¡Mal!", error);
});
```

```
ajaxUA('heroes.json').then(function(response) {
  return JSON.parse(response);
}).then(function(response) {
  console.log("JSON de Heroes:", response);
});
```

```
ajaxUA('heroes.json').then(JSON.parse).then(function(response) {
  console.log("JSON de Heroes:", response);
});
```

8.2.6 Encadenando promesas

- Cada vez que se llama a `then` o `catch` se crea una nueva promesa y se devuelve
  - Estas dos promesas son diferentes

```
var p1,p2;

p1 = Promise.resolve();
p2 = p1.then(function() {
  // ....
});
console.log(p1 !== p2);  // true
```

- Podemos encadenar promesas con el resultado del paso anterior

```
paso1().then(
  function paso2(resultadoPaso1) {
    // Acciones paso 2
  }
).then(
  function paso3(resultadoPaso2) {
    // Acciones paso 3
  }
).then(
  function paso4(resultadoPaso3) {
    // Acciones paso 3
  }
)
```

Ejemplo encadenando promesas

- Si un paso devuelve una promesa en vez de un valor, el siguiente paso recibe el valor empleado para completar la promesa

```
Promise.resolve('Hola!').then(
  function paso2(resultado) {
    console.log('Recibido paso 2: ' + resultado);
    return 'Saludos desde el paso 2';              // Devolvemos un valor
  }
).then(
  function paso3(resultado) {
    console.log('Recibido paso 3: ' + resultado); // No devolvemos nada
  }
).then(
  function paso4(resultado) {
    console.log('Recibido paso 4: ' + resultado);
    return Promise.resolve('Valor completado');   // Devuelve una promesa
  }
).then(
  function paso5(resultado) {
    console.log('Recibido paso 5: ' + resultado);
  }
);

// Consola
// "Recibido paso 2: Hola!"
// "Recibido paso 3: Saludos desde el paso 2"
// "Recibido paso 4: undefined"
// "Recibido paso 5: Valor completado"
```

Orden de ejecución de *callbacks*

- Los promesas permiten gestionar el orden en el que se ejecuta el código respecto a otras tareas.
  - El *callback* `resolver` que recibe el constructor de `Promise` se ejecuta de manera síncrona.
  - Todos los *callbacks* que se le pasan a `then` y a `catch` se invocan de manera asíncrona.

```
var promesa = new Promise(function (resolver, rechazar) {
  console.log("Antes de la funcion resolver");
  resolver();
});

promesa.then(function() {
  console.log("Dentro del callback de completado");
});

console.log("Fin de la cita");

// Consola
// Antes de la funcion resolver
// Fin de la cita
// Dentro del callback de completado
```

8.2.8 Gestión de errores

- Los rechazos y los errores se propagan a través de la cadena de promesas → efecto dominó.
- Posibilidades para capturar los errores
  - utilizar el método `then` y pasar *callback* como segundo argumento
  - o método `catch`

```
Promise.reject(Error("Algo ha ido mal")).then(
  function paso2() {
    console.log("Por aquí no pasaré");
  }
).then(
  function paso3() {
    console.log("Y por aquí tampoco");
  }
).catch(
  function (err) {
    console.error("Algo ha fallado por el camino");
    console.error(err);
  }
);

// Consola
// Algo ha fallado por el camino
// Error: Algo ha ido mal
```

Excepciones y promesas

- Una promesa se rechaza:
  - mediante la función `rechazar` del constructor
  - con `Promise.reject`
  - si el *callback* pasado a `then` lanza un `Error`
  - o si el constructor lanza un `Error`

- Al lanzar el objeto `Error` para rechazar una promesa, la pila de llamadas se captura
  - facilita el manejo del error en el `catch`.

```
rechazarCon("¡Malas noticias!").then(
  function paso2() {
    console.log("Por aquí no pasaré")
  }
).catch(
  function (err) {
    console.error("Y vuelve a fallar");
    console.error(err);
  }
);

function rechazarCon(cadena) {
  return new Promise(function (resolver, rechazar) {
    throw Error(cadena);
    resolver("No se utiliza");
  });
}
// Consola
// Y vuelve a fallar
// Error: ¡Malas noticias!
```

Autoevaluación

- ¿Qué saldrá por consola?

```
var promesaJSON = new Promise(function(resolver, rechazar) {
  resolver(JSON.parse("Esto no es JSON"));
});

promesaJSON.then(function(datos) {
  console.log("¡Bien!", datos);
}).catch(function(err) {
  console.error("¡Mal!", err);
});
```

`catch !== then(undefined, función)`

- `then(func1, func2)`, llamará a `func1` o a `func2`, nunca a las dos.
- `then(func1).catch(func2)` llamará a ambas si `func1` rechaza la promesa, ya que son pasos separados de la cadena.

```
paso1().then(function() {
  return paso2();
}).then(function() {
  return paso3();
}).catch(function(err) {
  return recuperacion1();
}).then(function() {
    return paso4();
  }, function(err) {
    return recuperacion2();
}).catch(function(err) {
  console.error("No me importa nada");
}).then(function() {
  console.log("¡Finiquitado!");
});
```

Promesas en paralelo

- Si ejecutamos un conjunto de promesas mediante un bucle, se ejecutarán en paralelo, en un orden indeterminado finalizando cada una conforme al tiempo necesario por cada tarea.

```
var cuentas = ["/banco1/12345678", "/banco2/13572468", "/banco3/87654321"];

cuentas.forEach(function(cuenta) {
  ajaxUA(cuenta).then(function(balance) {
    console.log(cuenta + " Balance -> " + balance);
  });
});

// Consola
// Banco 1 Balance -> 234
// Banco 3 Balance -> 1546
// Banco 2 Balance -> 789
```

Sincronizando promesas

- *"Cuando todas estas cosas hayan finalizado, haz esta otra".*
- `Promise.all(arrayDePromesas).then(function(arrayDeResultados)`
  - Devuelve una nueva promesa que se cumplirá cuando lo hayan hecho todas las promesas recibidas.
  - Si alguna se rechaza, la nueva promesa también se rechazará. El resultado es un array de resultados que siguen el mismo orden de las promesas recibidas.

```
var cuentas = ["/banco1/12345678", "/banco2/13572468", "/banco3/87654321"];

var peticiones = cuentas.map(function(cuenta) {
  return ajaxUA(cuenta);
});

Promise.all(peticiones).then(function (balances) {
  console.log("Los " + balances.length + " han sido actualizados");
}).catch(function(err) {
  console.error("Error al recuperar los balances", err);
})
// Consola
// Los 3 balances han sido actualizados
```

Secuencia de promesas

- Encadenando N promesas
- Array de promesas mediante un bucle / recursión

```
function secuencia(array, callback) {
  var seq = Promise.resolve();

  array.forEach(function (elem) {
    seq = seq.then(function() {
      return callback(elem);
    });
  });
}

secuencia(cuentas, function (cuenta) {
  return ajaxUA(cuenta).then(function(balance) {
    console.log(cuenta + " Balance -> " + balance);
  });
})
```

```
function secuencia(array, callback) {
  function cadena(array, indice) {
    if (indice === array.length) {
      return Promise.resolve();
    } else {
      return Promise.resolve(callback(array[indice])).
       then(function() {
         return cadena(array, indice + 1);
       });
    }
  }
  return cadena(array, 0);
}
```

8.2.11 Carrera de promesas

- ¿Y si nos interesará el que nos devuelve el resultado más rápidamente ?

- **`Promise.race(arrayDePromesas)`**
- Reduce el array de promesas y devuelve una nueva promesa con el primer valor disponible.
- Se examina cada promesa hasta que una de ellas finaliza, ya sea resuelta o rechazada, la cual se devuelve.

```
function obtenerDatos(url) {
  var tiempo = 500;  // ms
  var caduca = Date.now() + tiempo;

  var datosServer = ajaxUA(url);

  var datosCache = buscarEnCache(url).then(function (datos){
    return new Promise(function (resolver, rechazar) {
      var lapso = Math.max(caduca: Date.now(), 0);
      setTimeout(function () {
        resolver(datos);
      }, lapso);
    })
  });

  var fallo = new Promise(function (resolver, rechazar) {
    setTimeout(function () {
      rechazar(new Error("Error al acceder a " + url));
    }, tiempo);
  });

  return Promse.race([datosServer, datosCache, fallo]);
}
```

8.3 Fetch API

- ES6
- API para realizar peticiones AJAX que directamente devuelvan un promesa.

- *Google Chrome* y *Mozilla Firefox*.
  - http://caniuse.com/#search=fetch

- El objeto `window` ofrece el método `fetch`
- Argumentos:
  1. la URL de la petición
  2. (opcional) objeto literal que permite configurar la petición

```
// url (obligatorio), opciones (opcional)
fetch('/ruta/url', {
  method: 'get'
}).then(function(respuesta) {

}).catch(function(err) {
  // Error :(
});
```

Hola *Fetch API*

```
fetch('http://www.omdbapi.com/?s=batman', {
    method: 'get'
}).then(function(respuesta) {
  if (!respuesta.ok) {
    throw Error(respuesta.statusText);
  }
  return respuesta.json();
}).then(function(datos) {
    var pelis = datos.Search;
    for (var numPeli in pelis) {
        console.log(pelis[numPeli].Title + ": " + pelis[numPeli].Year);
    }
}).catch(function(err) {
  console.error("Error en Fetch de películas de Batman", err);
});
```

Cabeceras

- Objetos `Headers`
- Similar a un mapa

```
var headers = new Headers();

headers.append('Content-Type', 'text/plain');
headers.append('Mi-Cabecera-Personalizada', 'cualquierValor');

headers.has('Content-Type'); // true
headers.get('Content-Type'); // "text/plain"
headers.set('Content-Type', 'application/json');

headers.delete('Mi-Cabecera-Personalizada');

// Add initial values
var headers = new Headers({
  'Content-Type': 'text/plain',
  'Mi-Cabecera-Personalizada': 'cualquierValor'
});
```

```
var peticion = new Request('/url-peticion', {
  headers: new Headers({
    'Content-Type': 'text/plain'
  })
});

fetch(peticion).then(function() { /* manejar la respuesta */ });
```

Petición

- `method`: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`
- `url`: URL de la petición
- `headers`: objeto `Headers` con las cabeceras asociadas
- `body`: datos a enviar con la petición
- `referrer`: *referrer* de la petición
- `mode`: `cors`, `no-cors`, `same-origin`
- `credentials`: indica si se envían cookies con la petición: `include`, `omit`, `same-origin`
- `redirect`: follow, error, manual
- `integrity`: valor integridad del subrecurso
- `cache`: tipo de cache (`default`, `reload`, `no-cache`)

Ejemplo petición

```
var request = new Request('/heroes.json', {
  method: 'GET',
  mode: 'cors',
  headers: new Headers({
    'Content-Type': 'text/plain'
  })
});

fetch(request).then(function() { /* manejar la respuesta */ });
```

```
fetch('/heroes.json', {
  method: 'GET',
  mode: 'cors',
  headers: new Headers({
    'Content-Type': 'text/plain'
  })
}).then(function() { /* manejar la respuesta */ });
```

Enviando datos

```
fetch('/submit', {
  method: 'post',
  body: new FormData(document.getElementById('formulario-cliente'))
});
```

```
fetch('/submit-json', {
  method: 'post',
  body: JSON.stringify({
    email: document.getElementById('email').value
    comentarios: document.getElementById('comentarios').value
  })
});
```

```
fetch('/submit-urlencoded', {
  method: 'post',
  headers: {
    "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
  },
  body: 'heroe=Batman&nombre=Bruce+Wayne'
});
```

Respuesta

- Objeto `Response`

- `type`: indican el origen de la petición. Dependiendo del tipo, podremos consultar diferente información:
  - `basic`: proviene del mismo origen, sin restricciones.
  - `cors`: acceso permitido a origen externo.
  - `opaque`: origen externo que no devuelve cabeceras CORS,
- `status`: código de estado (200, 404, etc.)
- `ok`: *Booleano* que indica si la respuesta fue exitosa (en el rango 200-299 de estado)
- `statusText`: código de estado (OK)
- `headers`: objeto `Headers` asociado a la respuesta.

- `clone()`: clona el objeto `Response`
- `error()`: devuelve un nuevo objeto `Response` asociado con un error de red.

Tipos de respuesta

- `json()`: Devuelve una promesa que se resuelve con un objeto JSON.
- `text()`: Devuelve una promesa que se resuelve con un texto (USVString).
- Otros tipos: `formData()`, `blob()`, `arrayBuffer()`

```
fetch('heroes.json').then(function(response) {
  return response.json();
}).then(function(datos) {
  console.log(datos); // datos es un objeto JavaScript
});
```

```
fetch('/siguientePagina').then(function(response) {
  return response.text();
}).then(function(texto) {
  // <!DOCTYPE ....
  console.log(texto);
});
```

7.3 jQuery *Deferreds*

- Implementación que hace *jQuery* de las promesas
- Independiente de la versión de ECMAScript del navegador.

- Un *Deferred* es una promesa con métodos que permiten a su propietario resolverla o rechazarla
- Todas las promesas de otros propietarios son de sólo lectura.

`$.Deferred()`

- Operaciones:
  - `state()` → estado de la promesa
  - **`resolve`**`()` → resuelve la promesa
  - **`reject`**`()` → rechaza la promesa

```
var deferred = new $.Deferred();

deferred.state();  // pending
deferred.resolve();
deferred.state();  // resolved
deferred.reject();
```

- Si al método constructor le pasamos una función, ésta se ejecutará tan pronto como el objeto se cree, y la función recibe como parámetro el nuevo objeto `Deferred`.
  - Permite crear un envoltorio que realiza una tarea asíncrona y que dispare un *callback* cuando haya finalizado:

```
function realizarTarea() {
   return $.Deferred(function(def) {
      // tarea async que dispara un callback al acabar
   });
}
```

`promise()`

- Permite obtener una promesa pura.
- Similar a `Deferred`, excepto que faltan los métodos de `resolve()` y `reject()`.
- Se emplea para dar soporte a la encapsulación
  - Si una función devuelve un `Deferred`, puede ser resuelta o rechazada por el programa que la invoca.
  - Si sólo devolvemos la promesa pura correspondiente al *Deferred*, el programa que la invoca sólo puede leer su estado y añadir *callbacks*, no puede modificar su estado.
- Enfoque que sigue *jQuery* con `$.ajax()`

```
var obteniendoProductos = $.get("/products");

obteniendoProductos.state();  // "pending"
obteniendoProductos.resolve();  // undefined
```

Manejadores de Promesas

- Una vez tenemos una promesa, podemos adjuntarle tantos *callbacks* como queremos mediante los métodos:
  - `done()` → se lanza cuando la promesa se resuelve correctamente mediante `resolve()`
  - `fail()` → se lanza cuando la promesa se rechaza mediante `reject()`
  - `always()` → se lanza cuando se completa la promesa, independientemente que su estado sea resuelta o rechazada

```
promesa.done(function() {
   console.log("Se ejecutará cuando la promesa se resuelva.");
});

promesa.fail(function() {
   console.log("Se ejecutará cuando la promesa se rechace.");
});

promesa.always(function() {
   console.log("Se ejecutará en cualquier caso.");
});
```

Encadenando Promesas - `then()`

```
promesa.done(function() {
  console.log("Se ejecutará cuando la promesa se resuelva.");
}).fail(function() {
  console.log("Se ejecutará cuando la promesa se rechace.");
}).always(function() {
  console.log("Se ejecutará en cualquier caso.");
});
```

- `promesa.then(doneCallback, failCallback, alwaysCallback);`

```
promesa.then(function() {
  console.log("Se ejecutará cuando la promesa se resuelva.");
}, function() {
  console.log("Se ejecutará cuando la promesa se rechace.");
}, function() {
  console.log("Se ejecutará en cualquier caso.");
});
```

Orden de *callbacks*

- El orden en el que se adjuntan los *callbacks* definen su orden de ejecución.

```
var promesa = $.Deferred();
promesa.done(function() {
  console.log("Primer callback.");
}).done(function() {
  console.log("Segundo callback.");
}).done(function() {
  console.log("Tercer callback.");
});

promesa.fail(function() {
  console.log("Houston! Tenemos un problema");
});

promesa.always(function() {
  console.log("Dentro del always");
}).done(function() {
  console.log("Y un cuarto callback si todo ha ido bien");
});
```

```
"Primer callback."
"Segundo callback."
"Tercer callback."
"Dentro del always"
"Y un cuarto callback si todo ha ido bien"
```

http://jsbin.com/wanavo/1/edit?html,js,console,output

Prestando promesas del futuro

- Para separar la creación de una promesa del *callback* de lógica de aplicación → reenviar los eventos de `resolve`/`reject` desde la promesa POST a una promesa que se encuentre fuera de nuestro alcance.
- En vez de necesitar varias líneas con código anidado del tipo `promesa1.done(promesa2.resolve());` → Usar **`then()`**.

- `promesa.then(doneCallback, failCallback, alwaysCallback);`
  - Devuelve una nueva promesa que permite filtrar el estado y los valores de una promesa mediante una función
  - Es una **ventana al futuro** → permite adjuntar comportamiento a una promesa que todavía no existe.

Ejemplo `then()`

```
var enviandoObservaciones = new $.Deferred();
var guardandoObservaciones = enviandoObservaciones.then(function(input) {
  return $.post("/observaciones", input);
});
```

```
$("#observaciones").submit(function() {
  enviandoObservaciones.resolve($("textarea", this).val());
  return false;
});

enviandoObservaciones.done(function() {
  $("#contenido").append("<div class='spinner'>");
});
guardandoObservaciones.then(
  function() {  // done
    $("#contenido").append("<p>¡Gracias por las observaciones!</p>");
  }, function() { // fail
    $("#contenido").append("<p>Se ha producido un error al contactar con el servidor.</
p>");
  }, function() { // always
    $("#contenido").remove(".spinner");
});
```

Intersección de Promesas - `$.when()`

- Mismo funcionamiento que `Promise.all()`

- **`$.when`**`(promesa1, promesa2, promesa3, …)` → intersección de promesas
- Devuelve una nueva promesa que cumple estas reglas:
  - Cuando todas las promesas recibidas se resuelven, la nueva promesa esta resuelta.
  - Cuando alguna de las promesas recibidas se rechaza, la nueva promesa se rechaza.

- Permite crear un punto de sincronización de promesas

Ejemplo `$.when()`

```
$("#contenido").append("<div class='spinner'>");
$.when( $.get(“/encryptedData"), $.get("/encryptionKey"))
   .then(function() { // done
      // ambas llamadas han funcionado
   }, function() { // fail
      // una de las llamadas ha fallado
   }, function() { // always
      $("#contenido").remove(".spinner");
});
```

AJAX mediante *Deferreds*

- El objeto `jxXHR` que se obtiene de los métodos AJAX como `$.ajax()` o `$.getJSON()` implementan el interfaz `Promise`
  - Vamos a poder utilizar los métodos `done`, `fail`, `then`, `always` y `when()`.

```
function getDatos() {
  var peticion = $.getJSON("http://www.omdbapi.com/?s=batman&callback=?");
  peticion.done(todoOk).fail(function() {
    console.log("Algo ha fallado");
  });
  peticion.always(function() {
    console.log("Final, bien o mal");
  });
}

function todoOk(datos) {
  console.log("Datos recibidos y adjuntándolos a resultado");
  $("#resultado").append(JSON.stringify(datos));
}
```

http://jsbin.com/ponaca/edit?html,js,console,output

¿Preguntas?