
Tabla de contenido

Introduction	1.1
El entorno de trabajo	1.2
JavaScript básico	1.3
Programación orientada a objetos	1.3.1
El modelo de datos de JavaScript	1.3.2
El modelo de ejecución de JavaScript	1.3.3
Ejercicios guiados	1.3.4
Práctica	1.3.5
Guía	1.3.5.1
TDD	1.3.5.2
JavaScript en el navegador	1.4
Navegador y DOM	1.4.1
Canvas	1.4.2
Ejercicios guiados	1.4.3
Práctica	1.4.4
Guía	1.4.4.1

JavaScript para el desarrollo de videojuegos

Esta es una guía de introducción a **JavaScript**, y está orientada al desarrollo de **videojuegos HTML5**.

Esta basada en unos materiales que desarrollamos en una colaboración con la Universidad Complutense de Madrid para la asignatura de *Programación de videojuegos con lenguajes interpretados*. Puedes acceder a los materiales originales de la asignatura a través de [este repositorio en Github](#).

El código fuente de esta guía también está [publicado en Github](#). Si encuentras una errata o quieres sugerir algún cambio, por favor háznoslo saber [abriendo un ticket](#).

Videojuegos en la Web

La llegada de **HTML5** y sus tecnologías asociadas expandió enormemente las capacidades de la Web como **plataforma de videojuegos**. Hasta entonces, la mayoría de juegos web requerían un plugin externo –como Flash o Unity Player–, pero hoy ya no es necesario y los juegos HTML5 se ejecutan en el navegador de forma transparente.

La Web nos ofrece **API** de gráficos 2D y 3D (esta última, basada en el estándar OpenGL ES), de reproducción y sintetización de audio, de acceso a múltiples métodos de entrada (*gamepads*, eventos de *touch*, giroscopios...), etc. En definitiva, todo lo que necesitamos para desarrollar videojuegos.

Existen multitud de **motores y herramientas** para crear videojuegos HTML5. Algunos de los motores más populares, como Unity, Unreal o Game Maker, ya incluyen un exportador HTML5. También existen motores o frameworks específicos para la web, en los que podemos desarrollar con JavaScript, como Phaser o PlayCanvas.

El objetivo de esta guía es proporcionar una base de conocimientos JavaScript para que puedas desarrollar videojuegos web utilizando librerías o motores web existentes.

¿A quién está dirigida esta guía?

- A cualquiera con interés en el desarrollo de videojuegos y que ya tenga unos **conocimientos mínimos de programación** (cualquier lenguaje sirve, como Lua, C o Python): variables, bucles, funciones, condiciones, etc.
 - A programadores de videojuegos que quieran desarrollar videojuegos web con JavaScript.
 - A desarrolladores web que quieran aprender los fundamentos de la programación orientada a objetos con JavaScript.
-

Importante:

Se recomienda leer todos los artículos de una unidad, así como hacer los ejercicios guiados *antes* de realizar la práctica propuesta.

Programación orientada a objetos

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.

[Alan Kay sobre la programación orientada a objetos](#)

Modelado de problemas

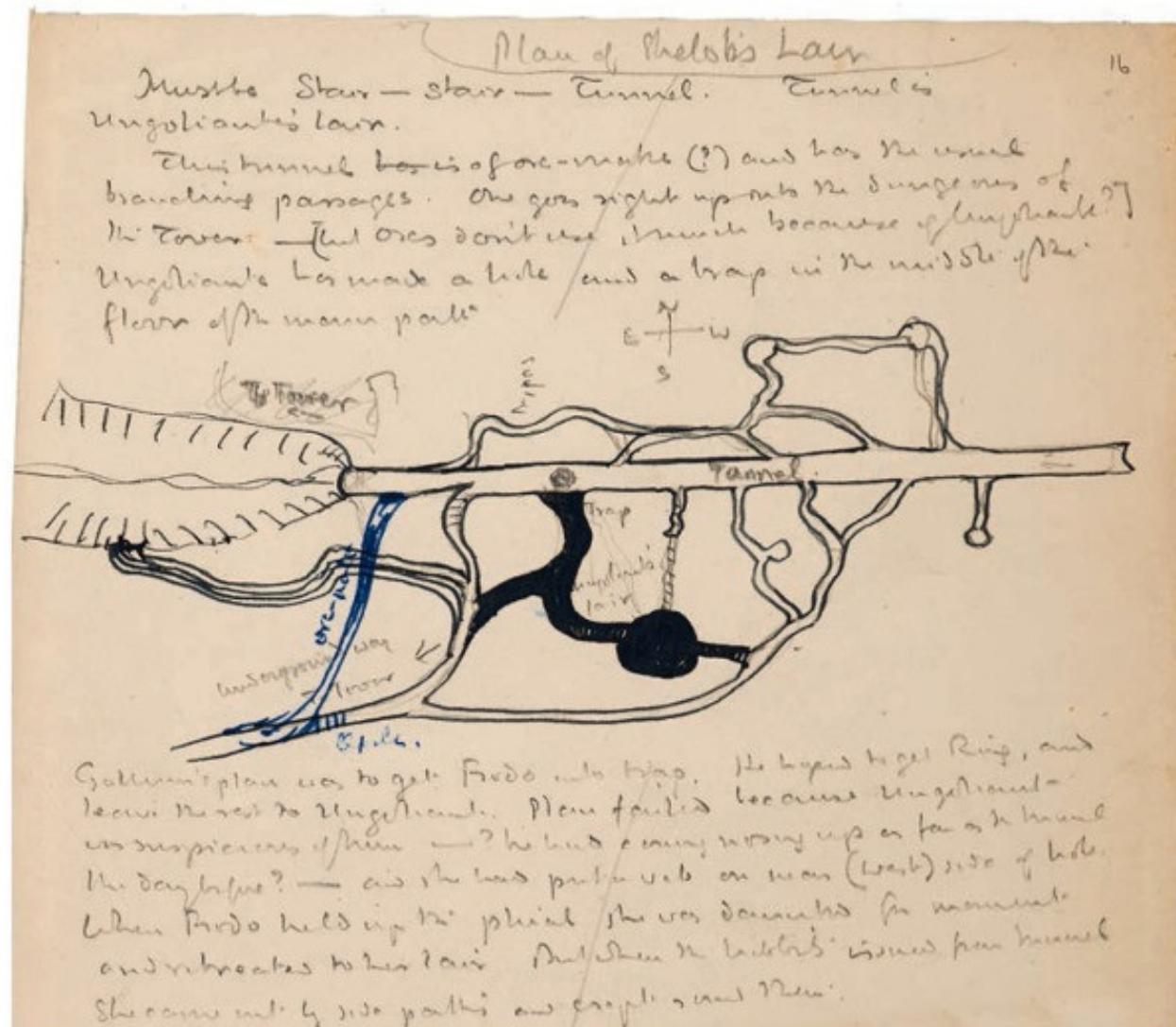
Programar es expresar un problema en un lenguaje de programación dado. Modelar representa un paso intermedio en el que se capturan y organizan los aspectos importantes de un problema.

El modelado de un problema es independiente del lenguaje de programación que se elija, pero el lenguaje seleccionado condiciona la facilidad con la que puedes codificar el modelo.

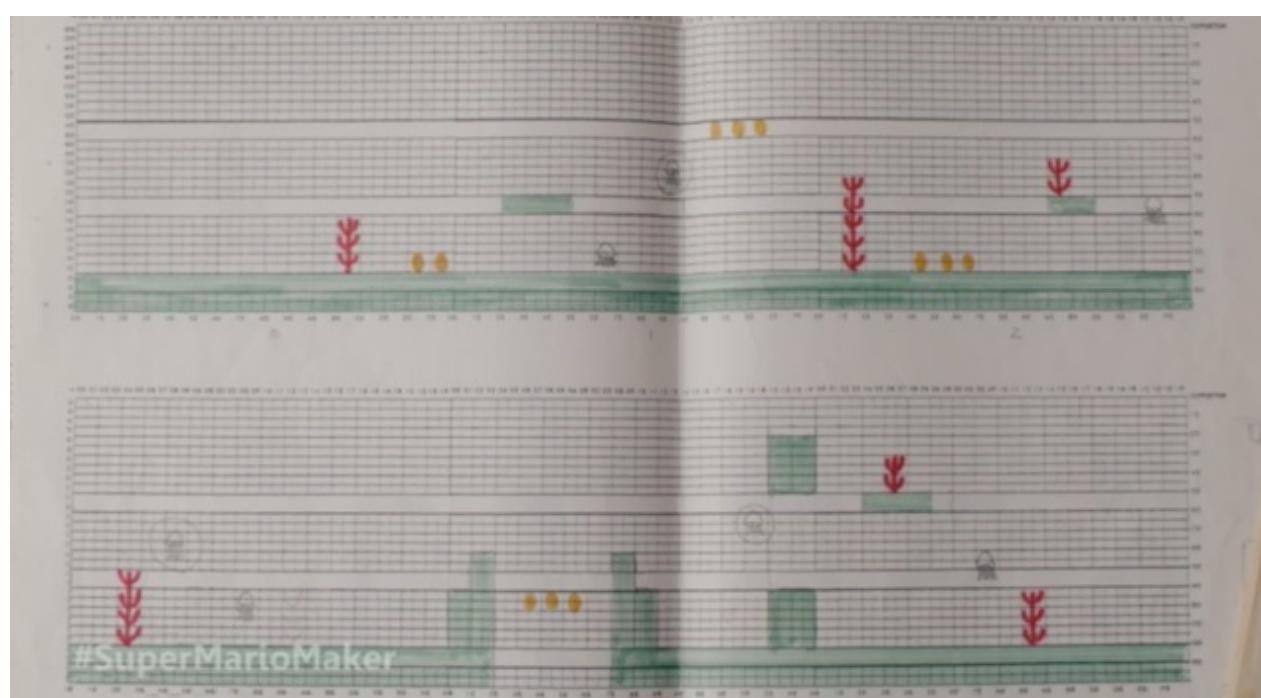
Muchas actividades creativas incluyen modelos intermedios entre la realidad y su expresión en el medio final. Por ejemplo, los *storyboards* se utilizan para planificar las secuencias de acción: capturan los momentos clave de la secuencia.



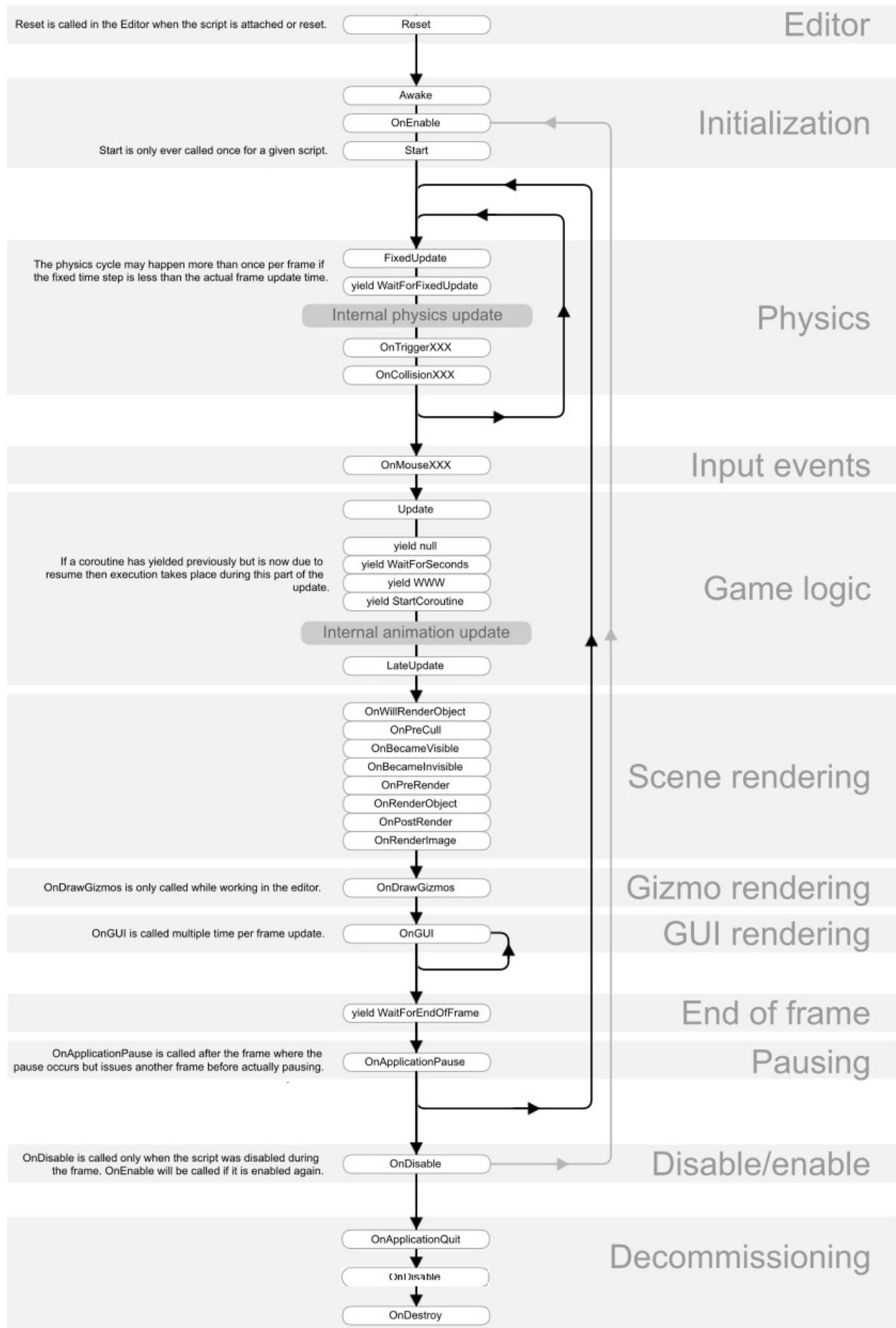
Este mapa muestra el paso de "Ella-Laraña", usado para mantener la coherencia del escenario descrito.



Este es un diseño de un mapa del videojuego Mario. Las herramientas digitales han permitido la automatización de modelos en software.



Y aquí el diagrama de flujo de cómo se llaman las distintas funciones de un script de Unity.



La programación orientada a objetos es una técnica de **modelado de problemas** que pone especial énfasis en dos conceptos: **objetos** y **paso de mensajes**.

Objetos

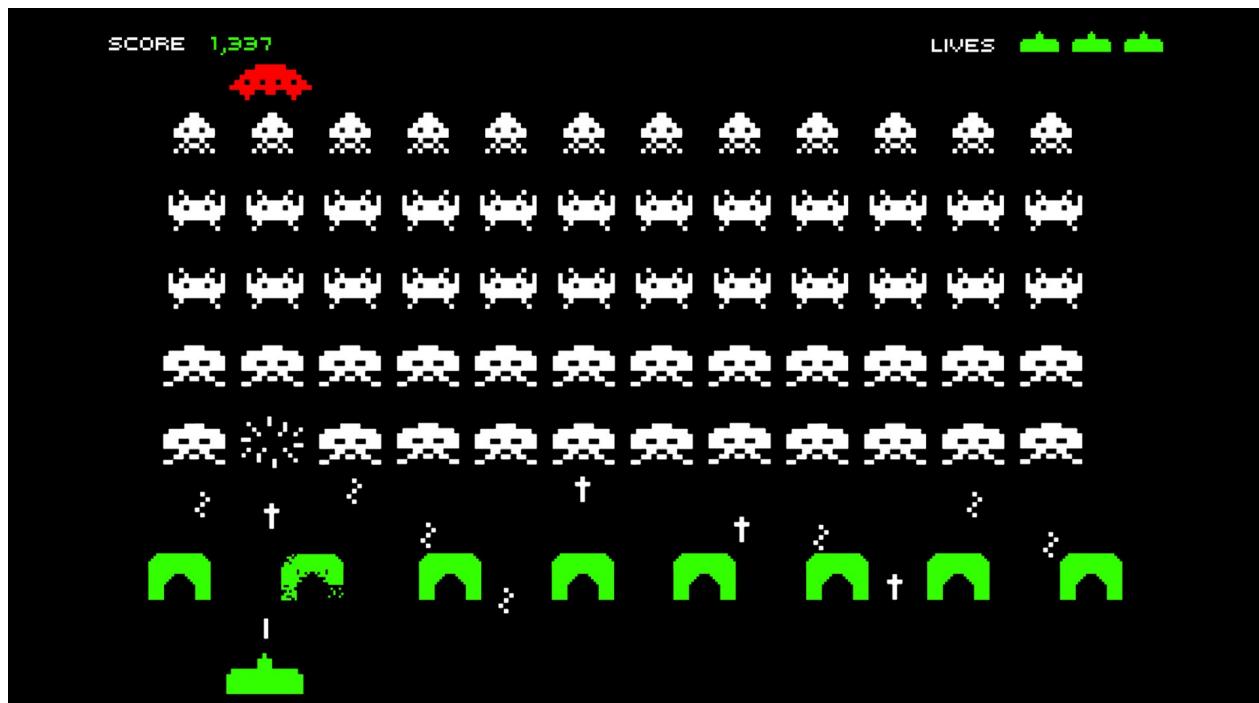
Los objetos son **representaciones de los aspectos de un problema** que desempeñan un rol específico, exponen un conjunto de funcionalidad concreta –la [API](#)–, y además, ocultan cómo realizan esa funcionalidad.

Paso de mensajes

Los mensajes son **peticiones de acción** de un objeto a otro. Estas peticiones parten de un objeto remitente hacia un objeto destinatario y codifican qué **funcionalidad de la API** se precisa.

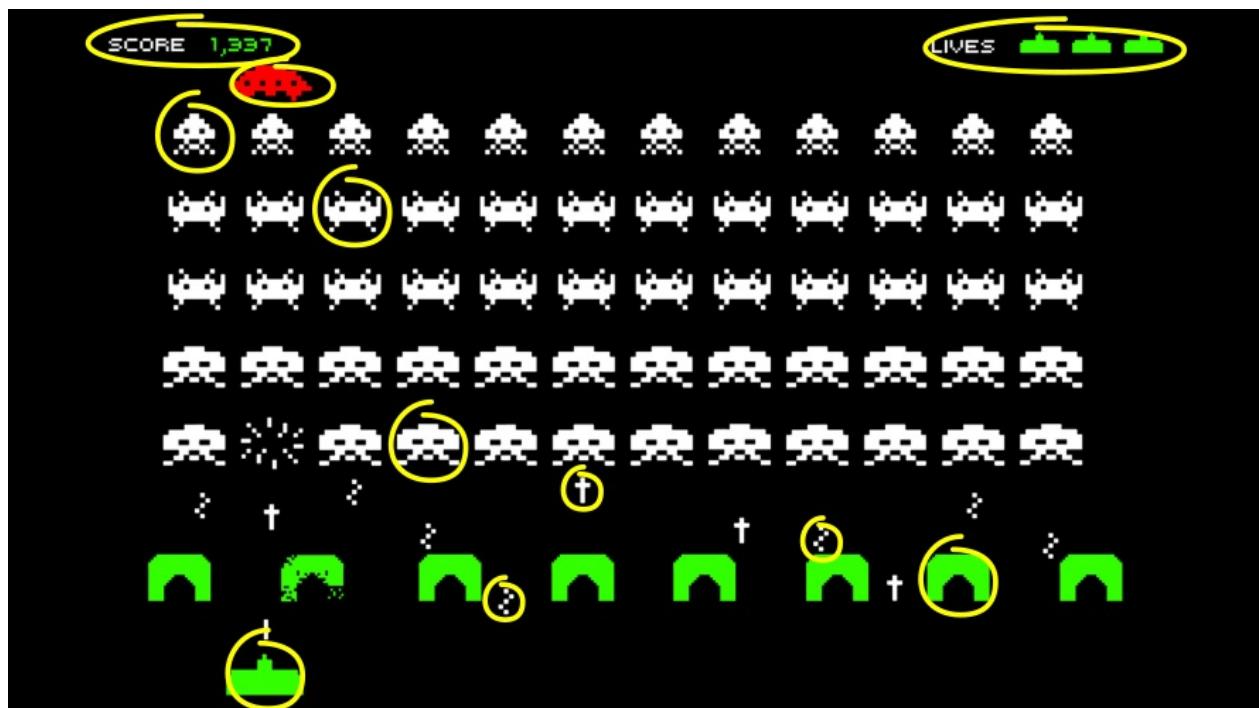
Modelado orientado a objetos

La definición de objetos y las interacciones entre los mismos modelan el problema. A lo largo de esta sección modelaremos informalmente el videojuego [Space Invaders](#).



Identificando objetos

Una técnica para identificar objetos consiste en **poner nombres**.



Estos son algunos objetos a los que podrías poner un nombre: nave amiga, enemigo 1, enemigo 2, enemigo 3, disparo amigo, disparo enemigo 1, disparo enemigo 2, defensa 1, defensa 2, marcador de vidas, marcador de puntuación.

Tipos de objetos e instancias

Queda claro de un vistazo que muchos objetos concretos pertenecen a familias o **tipos** de objetos. Conviene recordar que también se los llama **clases**.

Los tipos **especifican propiedades y comportamientos comunes** a todos ellos aunque individualmente sean distintos.



Algunos **tipos** que podrías identificar son marcadores, escudos, nave amiga, enemigos y disparos.

Los **valores** de un tipo son cada uno de los objetos individuales. El enemigo especial, así como cada uno de los otros enemigos será un valor distinto del **tipo enemigo**.

Cuando utilices la terminología de clases, los valores se convertirán en **instancias de la clase**.

En los modelos de objetos es más conveniente trabajar con tipos de objetos.



Interfaces y métodos

Para tratar de determinar la API de los tipos de objetos puedes guíarte por las interacciones propias del juego.

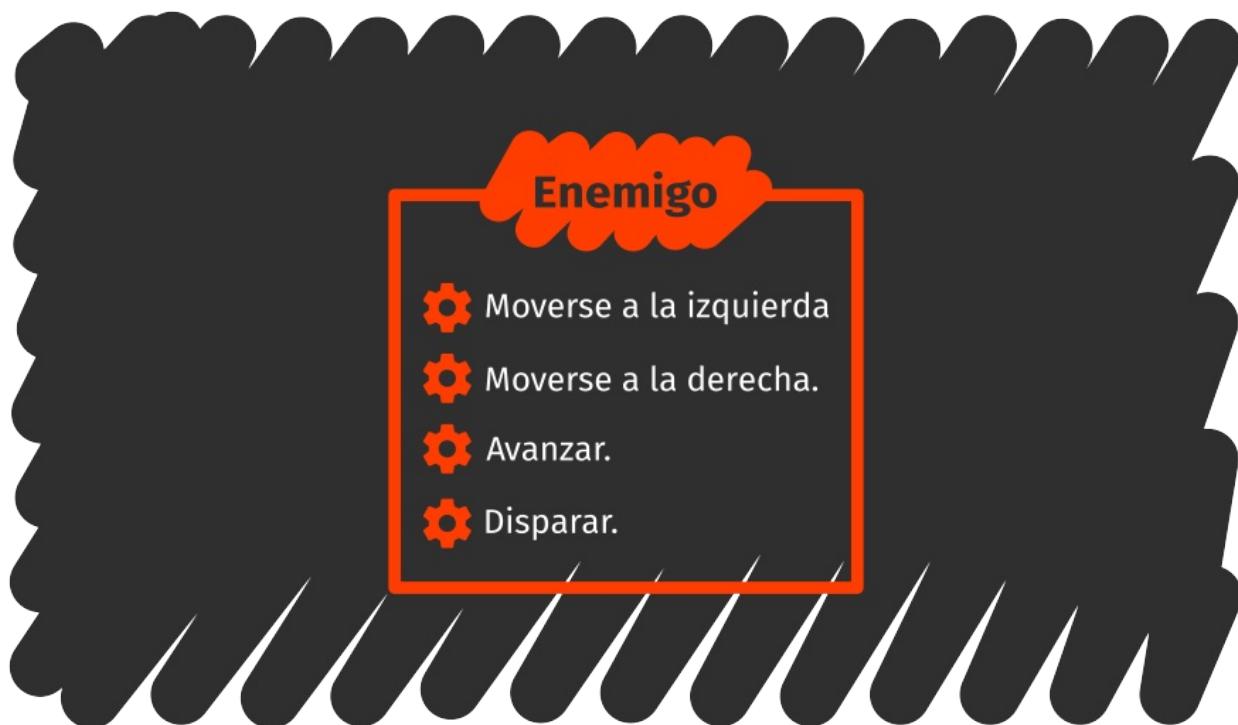
Un ejemplo:

Los enemigos se mueven todos juntos hacia un lado, avanzan una línea y se mueven hacia el otro lado mientras disparan aleatoriamente.

La técnica consiste en **buscar verbos** esta vez: *moverse*, *avanzar* y *disparar*, por ejemplo.

Para poder implementar el comportamiento de los enemigos, estos tienen que poder moverse hacia los lados, avanzar y disparar. Así, tendrán que permitir que les envíen mensajes pidiendo alguna de estas operaciones.

A las acciones que puede realizar un objeto se las denomina **métodos**.



Estado y atributos

Los objetos no sólo pueden realizar acciones, sino que además capturan **características** de las entidades a las que representan.

Cada enemigo, por ejemplo, tiene un gráfico distinto, una puntuación diferente, una posición en pantalla y además recordará en qué dirección se estaba moviendo.

El **estado** no se suele exponer de forma directa en la API. Piensa en el caso de los enemigos: incluso si estos tienen una posición, es preferible tener métodos específicos con los que manipular la posición (como "mover a la izquierda" o "mover a la derecha") en lugar de dar libre acceso a la posición.

A las características de un objeto se las denomina **atributos**.



El proceso de modelado es iterativo: al definir algunas acciones, se introducen nuevos nombres como *posición* o *dirección*, que se convertirán a su vez en nuevos tipos de objetos.

Constructores y creación de objetos

Pensemos ahora en la interacción del disparo:

Cuando el jugador pulsa el botón de disparo, aparece un proyectil delante de la nave amiga que avanza hasta alcanzar la parte superior de la pantalla o colisionar con un enemigo.

El proyectil no estaba ahí antes y tendrá que ser creado en el momento del disparo.

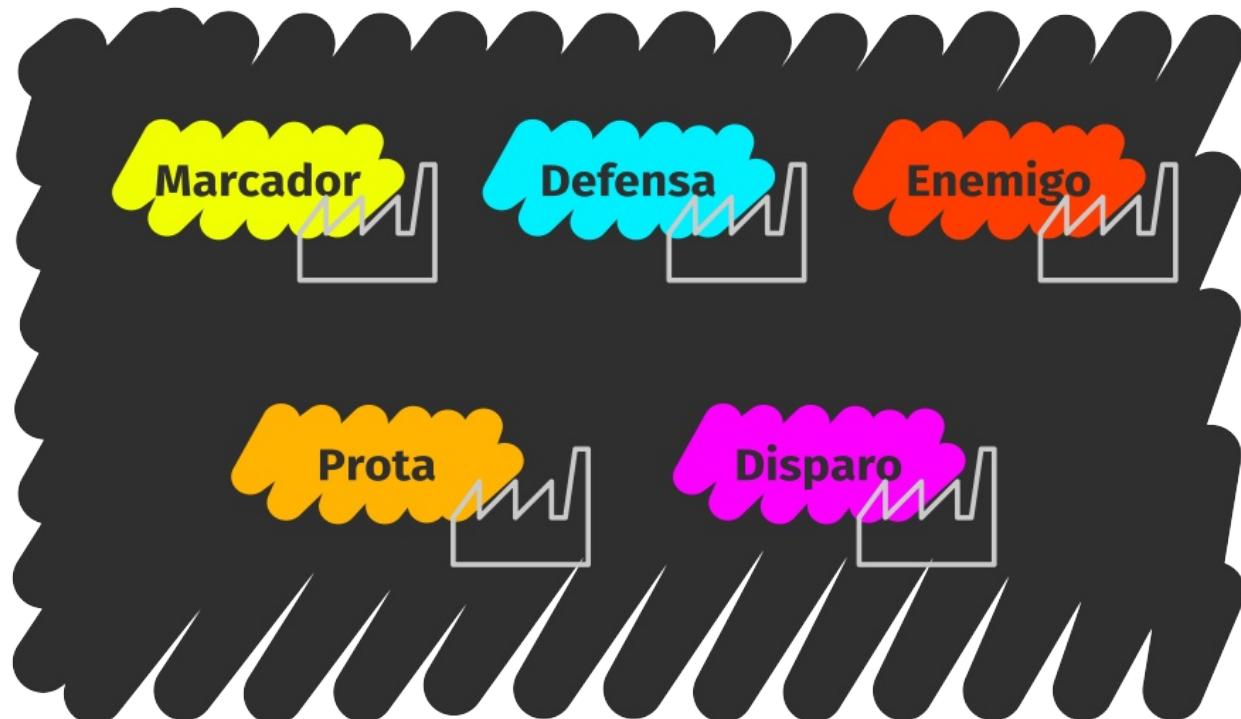
Otro ejemplo, la preparación del nivel antes de jugar:

Aparecen 55 enemigos en pantalla, 5 filas de 11 enemigos con la siguiente configuración: una fila de enemigos de la especie 1, dos filas de la especie 2, una de la especie 3 y una de la especie 4.

Está claro que no queremos escribir los 55 enemigos individualmente. Además, dado que todos pertenecen al tipo enemigo, también es evidente que serán todos muy parecidos.

Lo que se necesita es un mecanismo de **generación automática de objetos**. Cada lenguaje ofrece formas distintas de crear objetos.

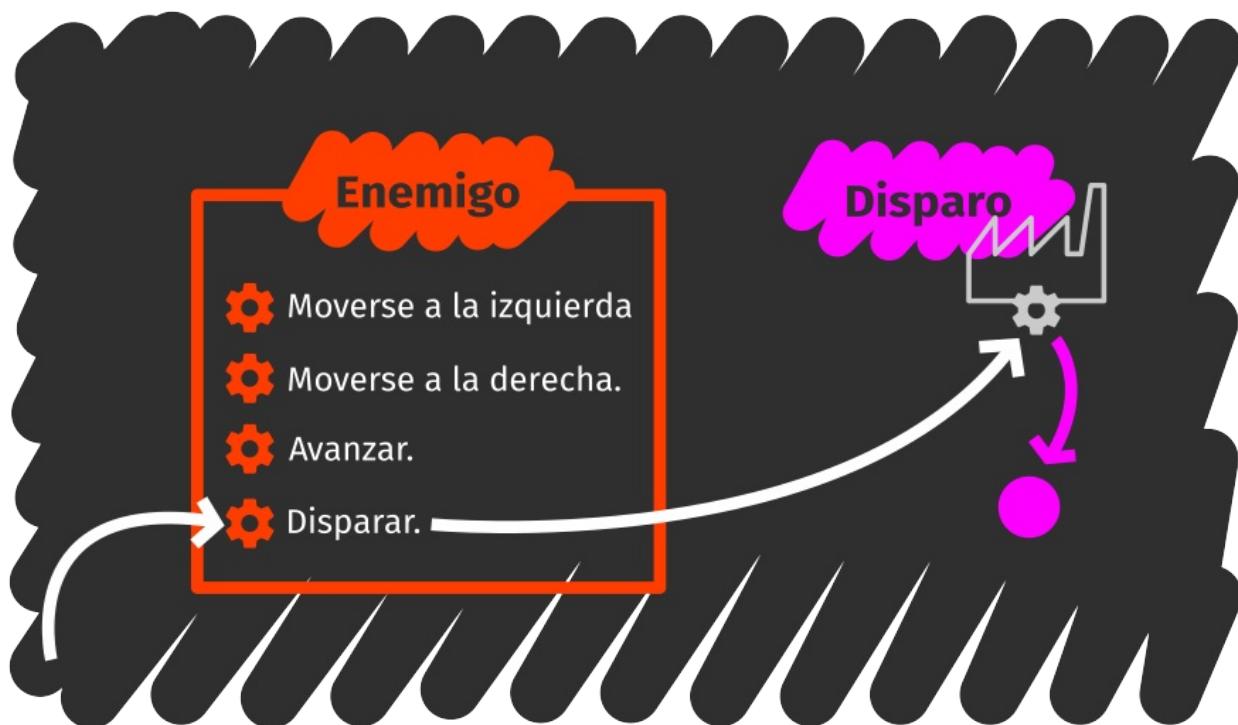
Un mecanismo útil es el de contar con un nuevo objeto el **constructor**, cuya tarea es la de generar objetos de un tipo dado. Habrá pues **un constructor por tipo**.



Los constructores tienen una API muy sencilla: *nuevo objeto*. Este método crea un nuevo objeto de un tipo dado.



Los constructores suelen permitir personalizar partes del objeto que están creando de forma que se le pueda decir algo como "*crea un disparo con esta posición, este gráfico y esta dirección de avance*".

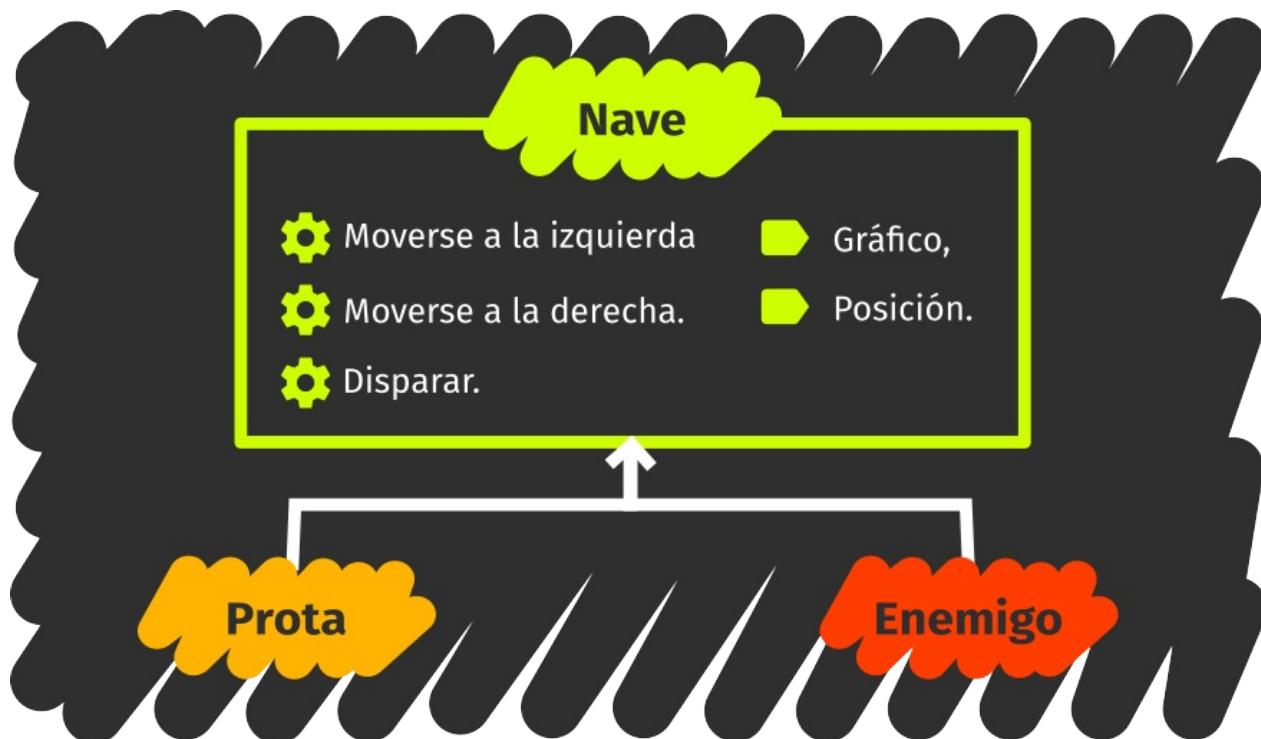


Relaciones entre tipos

Durante el modelado surgen relaciones de forma natural. Los enemigos *tienen* una posición. La nave amiga *crea* disparos.

Como es natural entre las personas, tenderás a **establecer jerarquías** entre objetos creando tipos más generalistas. Por ejemplo: en vez de pensar en enemigos y protagonista por separado, es posible pensar en *naves*.

El tipo `nave` reúne los métodos y atributos comunes de la nave protagonista y enemigos.



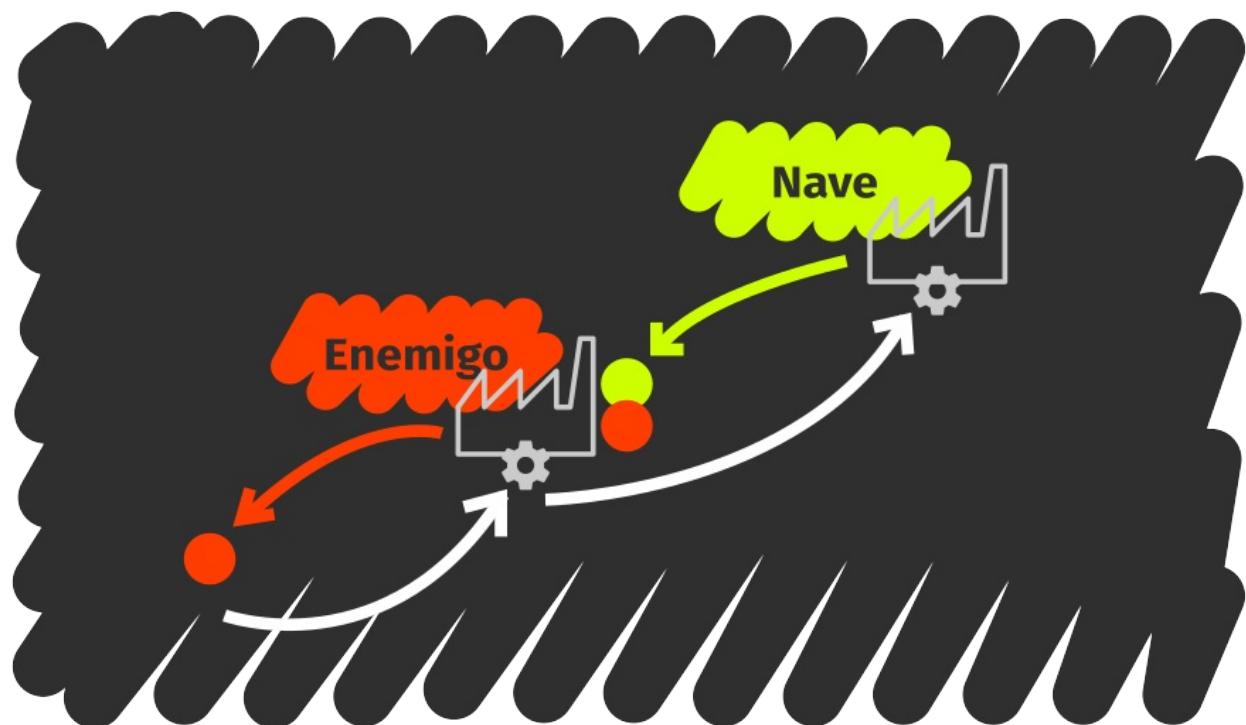
Esta jerarquía establece **relaciones de herencia** también llamadas relaciones "*es un(a)*" dado que el protagonista es una `nave` y el enemigo es una `nave` también.

Se dice que el tipo `enemigo` **extiende** al tipo `nave` añadiendo `avanzar` a la API, así como la puntuación y la última dirección de desplazamiento al estado.

La nave amiga no añade ningún método nuevo pero **redefine o sobreescribe** el método `disparar` para que dispare hacia arriba.

Como hay nuevos tipos, necesitarás nuevos constructores. Los viejos constructores pueden **delegar** parte de la creación del objeto (las partes comunes) a los nuevos.

De esta forma, al pedir un enemigo, el constructor de enemigos pedirá una nave al constructor de naves. Luego tomará esa nave, la modificará para que sea un enemigo y devolverá un enemigo.



El modelo de datos de JavaScript

Conocer un lenguaje de programación se traduce básicamente en conocer su sintaxis, modelo de datos, modelo de ejecución y estilo.

Durante esta lección, [codificarás en JavaScript lo aprendido en la lección anterior](#).

No todos los lenguajes permiten una transcripción 1 a 1 de los conceptos que recogemos en el modelo. Por ejemplo, JavaScript no tiene un mecanismo para crear tipos nuevos, pero tiene otros mecanismos que permiten implementar una funcionalidad similar.

Experimentando con JavaScript

Vas a experimentar con JavaScript, así que necesitarás una forma rápida de inspeccionar expresiones y obtener feedback de lo que estás haciendo. La mejor forma es utilizar la **consola de Node**. Por ejemplo:

```
$ node --use_strict
```

La opción `--use_strict` activa el modo estricto de JavaScript, que simplifica algunos aspectos del lenguaje. El modo estricto recorta algunas características, pero los beneficios son mayores que las pérdidas.

Ahora puedes probar a introducir algunas expresiones:

```
> 40 + 2
42
> var point = { x: 1, y: 1 };
undefined
> point
{ x: 1, y: 1 }
> point.x
1
```

Para limpiar la pantalla presiona `ctrl+l`. Para salir de Node, presiona `ctrl+c` dos veces seguidas. Si escribiendo una expresión Node parece no responder, presiona `ctrl+c` una vez para cancelar la expresión.

Si no quieres lidiar con la consola de Node, siempre puedes escribir un programa y usar `console.log()` para mostrar expresiones por pantalla.

```
// en el fichero prueba.js
console.log(40 + 2);
var point = { x: 1, y: 1 };
console.log(point);
console.log('Coordenada X:', point.x);
```

Ahora ejecuta el programa con Node:

```
$ node prueba.js
42
{ x: 1, y: 1 }
Coordenada X: 1
```

Esta lección asume que utilizarás una única sesión de la consola de Node, a menos que se indique lo contrario.

Para la mayoría de los ejemplos, puedes mantener la misma sesión abierta, pero si te encuentras con algo inesperado, antes de nada prueba a reiniciar la consola. Para reiniciar la consola tienes que **salir y volver a entrar**.

Lo mejor es que se tenga este texto abierto en una ventana (o impreso), y la consola de Node en otra.

Tipos primitivos

Se llaman **tipos primitivos** a aquellos que vienen con el lenguaje y que permiten la creación de nuevos tipos más complejos. En JavaScript, los tipos primitivos son: **booleanos, números, cadenas (strings), objetos y funciones**.

```
// En los comentarios hay más valores posibles para cada uno de los tipos.
var bool = true; // false
var number = 1234.5; // 42, -Infinity, +Infinity
var text = 'I want to be a pirate!'; // "I want to be a pirate"
var object = {}; // [], null
var code = function () { return 42; };
```

Los puedes reconocer porque responden de manera distinta al operador `typeof`. Observa cómo los tipos son cadenas de texto:

```
typeof true;
typeof 1234.5;
typeof 'I want to be a pirate!';
typeof {};
typeof function () { return 42; };
```

En JavaScript se puede declarar una variable y no asignarle ningún valor. En este caso, el tipo de la variable será `'undefined'`.

```
var x;
typeof x;
x = 5; // tan pronto como le demos un valor, el tipo dejará de ser undefined.
typeof x;
```

Objetos en JavaScript

De entre todos los tipos, vamos a prestar especial atención a aquel cuyos valores permiten la **composición** con otros valores. Estos son los de tipo `'object'` (objeto).

En JavaScript, los objetos son colecciones de valores etiquetados. Por ejemplo, si queremos representar el punto `(10, 15)` del plano XY, podemos etiquetar el valor en el eje X con la cadena `'x'` y el valor en el eje Y con la cadena `'y'`.

```
var point = { 'x': 10, 'y': 15 };
```

Cada par etiqueta y valor se llama **propiedad del objeto**. No es algo estricto, pero cuando se habla de las propiedades de un objeto, se suele referir a los valores; mientras que para hablar de las etiquetas se suele decir **nombre de la propiedad**.

Si los nombres de las propiedades se escriben siguiendo las [reglas de formación de identificadores](#) en JavaScript, las comillas no son necesarias y podemos ahorrárnoslas.

```
var point = { x: 10, y: 10 }; // mucho más conveniente.
```

Este es el caso más frecuente, el *recomendado*, y el que usaremos a lo largo de este material; pero conviene saber que, por debajo, **el nombre de la propiedad es una cadena**.

Para acceder a las propiedades de un objeto, usamos los corchetes `[]` con el nombre de la propiedad entre estos:

```
point['x'];
point['y'];
```

De nuevo, si seguimos las reglas de formación de identificadores, podemos usar la **notación de punto** para acceder a la propiedad, mucho más rápida de escribir:

```
point.x;
point.y;
```

Para cambiar el valor de una propiedad se utiliza el operador de asignación:

```
point.x = 0;
point.y = 0;
point['x'] = 0;
point['y'] = 0;
```

Si se accede a una **propiedad que no existe**, obtendrás el valor `undefined`:

```
var label = point.label; // será undefined. Compruébalo con typeof.
```

En cualquier momento podemos crear propiedades nuevas asignándoles algo.

```
point.label = 'origin';
point;
```

Arrays

Las listas o **arrays** son colecciones de **datos ordenados**.

Por ejemplo, la lista de comandos de un menú en un videojuego:

```
var menu = ['Attack', 'Defense', 'Inventory'];
```

En este tipo de objetos, el orden importa. Para acceder a los distintos valores se utiliza el **índice del elemento en la lista**, entre corchetes. Los índices *comienzan en 0*, y no en 1.

```
menu[0];
menu[1];
menu[2];
```

Se puede consultar la longitud de un **array** accediendo a la propiedad `length`.

```
menu.length;
```

Se puede añadir un elemento al final del **array** llamando al método `push`:

```
menu.push('Magic');
```

También se puede quitar un elemento por el final usando el método `pop`:

```
menu.pop();
```

Se puede alterar un **array** (insertar o borrar elementos), en cualquier lugar, usando el método `splice`:

```
// Inspecciona la lista tras cada operación.
menu = ['Attack', 'Defense', 'Inventory'];
menu.splice(2, 0, 'Magic'); // añade Magic antes de Inventory.
menu.splice(2, 2, 'Ench. Inventory'); // reemplaza Magic and Inventory with Ench.
Inventory.
menu.splice(0, 0, 'Wait'); // añade Wait al principio de la lista.
```

Como en el caso de los objetos, podemos cambiar cualquier valor en cualquier momento usando el operador de asignación.

```
menu[0] = 'Special'; // reemplaza Wait con Special
```

También como en el caso de los objetos, podemos acceder a un valor que no existe y recuperarlo o asignarlo en cualquier momento.

```
menu;
menu.length;
var item = menu[10];
typeof item; // será undefined.
menu[10] = 'Secret';
menu;
menu.length;
```

Si asignamos a un índice por encima de la longitud actual, **se extenderá el array hasta ese índice.**

Distinguir entre objetos y arrays

Arrays y objetos tienen tipo `'object'`, así que se ha de usar el método `Array.isArray()` para distinguirlos.

```
var obj = {};// el objeto vacío es tan válido como cualquier otro.
var arr = [];// una lista sin elementos, como te puedes imaginar.
typeof obj; // será object.
typeof arr; // será object.
Array.isArray(obj); // será false.
Array.isArray(arr); // será true.
```

null

Existe un último valor para el tipo objeto, que es `null`. Este valor representa la **ausencia de objeto** y se suele utilizar para:

- En funciones en las que se pregunta por un objeto, indicar que no se ha encontrado tal objeto.
- En relaciones de composición, indicar que el objeto compuesto ya no necesita al objeto componente.

Por ejemplo, en un RPG, podemos preguntar por el siguiente enemigo vivo para comprobar si debemos continuar la batalla:

```
function getNextAliveEnemy() {
  var nextEnemy;
  if (aliveEnemies.length > 0) {
    nextEnemy = aliveEnemies[0];
  }
  else {
    nextEnemy = null;
  }
  return nextEnemy;
}
```

O bien, supón la ficha de personaje de un héroe:

```
var hero = { sword: null, shield: null };
hero.sword = { attack: 20, magic: 5 }; // coge una espada.
hero.sword = null; // suelta la espada.
```

Composición de objetos

Objetos y *arrays* permiten cualquier composición de objetos. Es decir, sus valores pueden ser otros objetos y *arrays*, números, cadenas o funciones.

El siguiente ejemplo muestra una posible ficha de personaje de un RPG:

```
var hero = {
  name: 'Link',
  life: 100,
  weapon: { kind: 'sword', power: 20, magicPower: 5 },
  defense: { kind: 'shield', power: 5, magicPower: 0 },
  // Inventario por slots. Dos slots vacíos y un último con 5 pociones.
  inventory: [
    { item: null, count: 0},
    { item: null, count: 0},
    { item: { kind: 'potion', power: 15 }, count: 5}
  ]
};
```

Algunas propiedades:

```
hero.name; // el nombre del héroe
hero.weapon.kind; // el tipo de arma
hero.inventory[0]; // el primer slot del inventario
hero.inventory[0].item; // qué hay en el primer slot del inventario
hero.inventory[2].item.power; // el poder del item del 3r slot del inventario
```

Identidad de los objetos

En JavaScript, el operador de igualdad es `==` (el triple igual). Esto permite comparar dos objetos y decidir si **son iguales**. También existe el operador de desigualdad `!=` que compara dos objetos y decide si **no son iguales**.

Para los tipos `'bool'`, `'string'`, `'number'` y `'undefined'`, dos valores son iguales si tienen **la misma forma**:

```
// Todas estas comparaciones son verdaderas.
"Hola" === "Hola";
"Hola" !== "hola";
true === true;
123 === 123.0;
123 !== "123";
123 === 122 + 1; // primero se resuelve la expresión, luego se compara.
undefined === undefined;
```

Para el tipo `object`, dos objetos son iguales sólo si se refieren al mismo objeto:

```
({} !== {}); // da igual la forma, esto son dos objetos distintos.
({} !== []);
[] !== []; // igual que antes.
[1, 2, 3] !== [1, 2, 3]; // la forma da igual, los objetos son distintos.
null === null; // pero con null funciona porque sólo hay un valor null.
var obj = {};
var sameObj = obj;
var another = {};
sameObj === obj; // funciona porque ambos nombres se refieren al mismo objeto.
sameObj !== another; // insisto, distintos, pese a la forma.
```

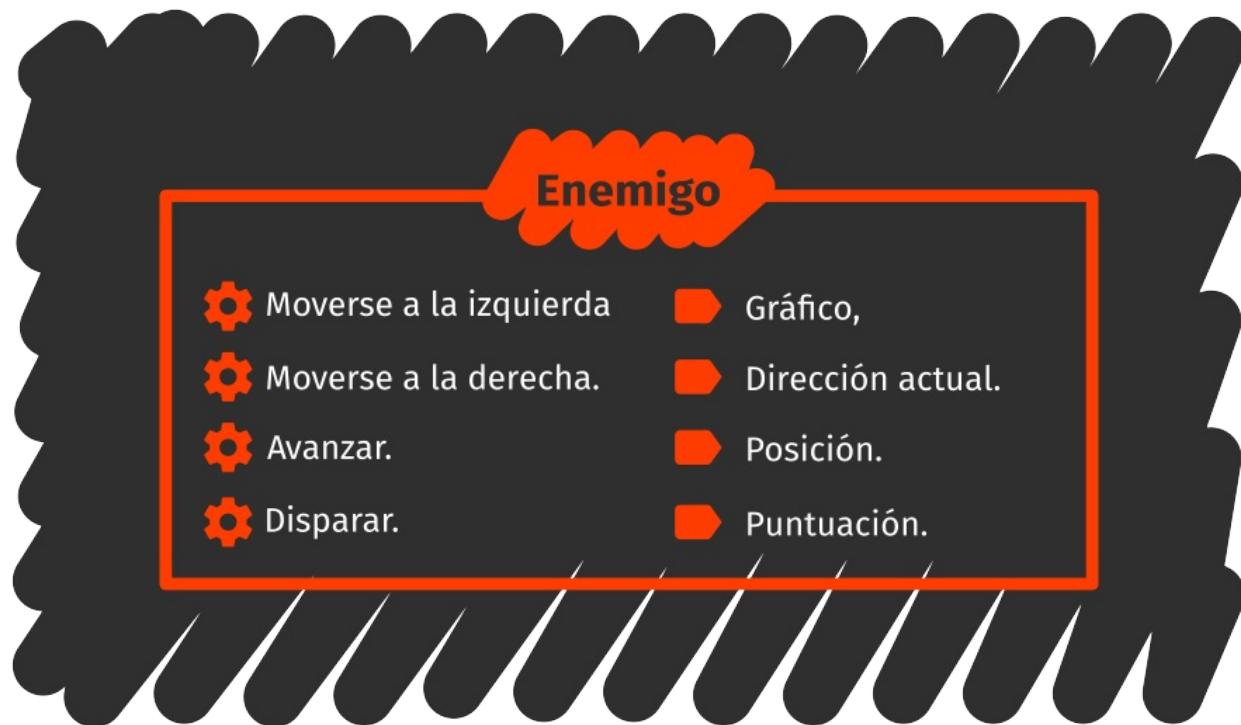
Objetos y paso de mensajes

Los objetos de JavaScript y el poder usar código como un valor más, permiten codificar los conceptos de *objeto* y *paso de mensajes* de la programación orientada a objetos.

Codificando el estado

Con lo que se ha visto hasta ahora deberías tener el conocimiento suficiente para codificar el estado. El **conjunto de atributos** del objeto en el modelo orientado a objetos se traduce en el **conjunto de propiedades** de los objetos JavaScript.

En el ejemplo de *Space Invaders*, el estado de los enemigos formado por:



Se puede codificar mediante:

```
var enemy = {  
    graphic: 'specie01.png',  
    currentDirection: 'right',  
    position: { x: 10, y: 10 },  
    score: 40  
};
```

La primera limitación en JavaScript es que **no se puede restringir el acceso a las propiedades de un objeto** (es decir, no hay propiedades privadas). Así, nada nos impide poder modificar la posición directamente.

```
enemy.position.x = 100; // perfectamente válido.
```

Lo único que se puede hacer es desaconsejar al usuario de ese código que utilice ciertas propiedades. Una práctica muy común en JavaScript es añadir un guión bajo `_` a los atributos que consideramos que son **privados**:

```
var enemy = {  
    _graphic: 'specie01.png',  
    _currentDirection: 'right',  
    _position: { x: 10, y: 10 },  
    _score: 40  
};
```

Pero, insistimos, esto es una convención y se puede seguir accediendo a los atributos que tengan este guión bajo:

```
enemy._position.x = 100; // perfectamente válido también.
```

Codificando la API

Las acciones que forman la API de un objeto, los **métodos**, pueden implementarse como **funciones** en propiedades de un objeto.



```
var enemy = {  
    _graphic: 'specie01.png',  
    _currentDirection: 'right',  
    _position: { x: 10, y: 10 },  
    _score: 40,  
  
    moveLeft: function () { console.log('Going left!'); },  
    moveRight: function () { console.log('Going right!'); },  
    advance: function () { console.log('Marching forward!'); },  
    shoot: function () { console.log('PICHUM!'); } // (es un láser)  
};
```

Enviar un mensaje a un objeto consiste sencillamente acceder a la propiedad del destinatario, que será una función, y llamarla.

```
enemy.shoot(); // primero accedemos con punto, luego llamamos con ().  
enemy.moveLeft();  
enemy.moveLeft();  
enemy.advance();  
enemy['shoot'](); // es lo mismo, acceder con corchetes y llamar con ().
```

Cualquier función puede actuar como método. Para que actúe como un método tan sólo es necesario **llamarla desde la propiedad de un objeto**. Y, como cualquier propiedad de un objeto, podemos cambiarla en cualquier momento:

```
enemy.shoot(); // PICHUM!  
enemy.shoot = function () { console.log('PAÑUM!'); };  
enemy.shoot(); // PAÑUM!
```

Ahora bien, observa el siguiente comportamiento:

```
enemy; // fíjate en la posición.  
enemy.moveLeft();  
enemy; // fíjate en la posición otra vez.
```

Obviamente, echando un vistazo a lo que hace `moveLeft`, no podríamos decir que *cambia el estado* del objeto destinatario del mensaje. ¿Cómo podríamos solucionarlo?

Como cualquier función puede actuar como método, hace falta una forma de **referirse al destinatario del mensaje**, si existe. Cuando se usa como un método, el destinatario se guarda siempre en la variable `this`.

Gracias a ella, podemos implementar los métodos de movimiento:

```
enemy.moveLeft = function () { this._position.x -= 2; };
enemy.moveRight = function () { this._position.x += 2; };
enemy.advance = function () { this._position.y += 2; };
```

Prueba el mismo experimento de antes y observa cómo efectivamente alteramos el estado del objeto.

```
enemy; // fíjate en la posición.
enemy.moveLeft();
enemy; // fíjate en la posición otra vez.
```

El valor `this`

El valor de `this` es uno de los aspectos más controvertidos de JavaScript.

En otros lenguajes, métodos y funciones son cosas distintas y un método *siempre* tiene asociado un –y sólo un– objeto, así que `this` nunca cambia.

Pero en JavaScript, `this` depende de cómo se llame a la función: si se llama como si fuera una función, o si se llama como si fuera un método.

Considera la siguiente función:

```
function inspect() {
  // sólo inspecciona this
  console.log('Tipo:', typeof this);
  console.log('Valor:', this);
}
```

Y prueba lo siguiente:

```
// Piensa qué puede valer this antes de probar cada ejemplo.
var ship1 = { name: 'T-Fighter', method: inspect };
var ship2 = { name: 'X-Wing', method: inspect };
ship1.method();
ship2.method();
inspect();
```

En el último caso, el valor de `this` es `undefined` porque la función no se está usando como un método, por lo que no hay destinatario.

En JavaScript podemos hacer que cualquier objeto sea `this` en cualquier función. Para ello usaremos `apply` en una función.

```
var onlyNameShip = { name: 'Death Star' };
inspect.apply(onlyNameShip); // hace que this valga onlyNameShip en inspect.
```

A `this` se le conoce también como **objeto de contexto**, y en este material usaremos este término de vez en cuando.

Consideraciones adicionales

Nombres y valores

Una **variable es un nombre**. Para el programa, quitando algunas excepciones, los nombres no tienen significado.

Un **valor no es un nombre**. De hecho, sólo las funciones pueden tener nombre con el fin de poder implementar recursividad y un par de cosas más.

Así que no es lo mismo el nombre `uno` que el valor `1`, y por supuesto, no es obligatoria ninguna relación coherente entre el nombre y el valor.

```
var uno = 2; // para el programa tiene sentido, quizás para el programador no.
```

En general, hablando de booleanos, cadenas y números, decimos que los **nombres guardan valores**, mientras que si hablamos de objetos y funciones decimos que los **nombres apuntan a objetos o funciones o son referencias** a objetos o funciones.

Funciones, referencias a funciones y llamadas a funciones

Hay dos formas de definir una función. Una es usando la **declaración de función**

```
function :
```

```
// Introduce una variable factorial que apunta a la función factorial.
function factorial(number) {
  if (number === 0) {
    return 1;
  }
  return number * factorial(number - 1);
} // no hace falta un ';' en este caso.
```

En este caso, el nombre de la función (antes de los paréntesis) es obligatorio. Dar nombre a una función tiene dos implicaciones:

- Permite implementar **llamadas recursivas** como la del ejemplo.
- **Crea un nombre** `factorial` para referirnos a esa función.

La otra forma es usar una **expression de función**. Esta se parece más a como crearíamos otros valores, como números o cadenas:

```
// Introduce una variable recursiveFunction que apunta a OTRA función factorial.
var recursiveFunction = function factorial(number) {
  if (number === 0) {
    return 1;
  }
  return number * factorial(number - 1);
}; // ahora sí hace falta ';', como en cualquier asignación.
```

En este último caso, hay dos nombres. Uno es el nombre de la función `factorial`, que existe para poder referirnos a ella dentro del cuerpo de la función. El otro es la variable `recursiveFunction` que referencia a la función.

La misma función puede referirse desde múltiples variables o, dicho de otra manera, tener muchos nombres:

```
var a = recursiveFunction;
var b = recursiveFunction;
a === b; // es cierto, se refieren a la misma función.
a.name; // el nombre de la función no tiene que ver con el de la variable.
b.name; // lo mismo.
recursiveFunction !== factorial;
```

Tampoco podemos confundir la referencia a la función `factorial` y la llamada a la misma función, por ejemplo: `factorial(10)`.

Con la primera forma **nos referimos al objeto** que encapsula el código que hay que ejecutar. No requiere parámetros porque **no se quiere ejecutar el código** sino solamente referirse a la función.

Con la segunda, **pedimos a la función que se ejecute** y por tanto habrá que aportar todos los parámetros necesarios.

En JavaScript todo es un objeto

Si, como definición alternativa, consideramos como objeto aquello que puede responder a un mensaje, resulta que en JavaScript **todo es un objeto**.

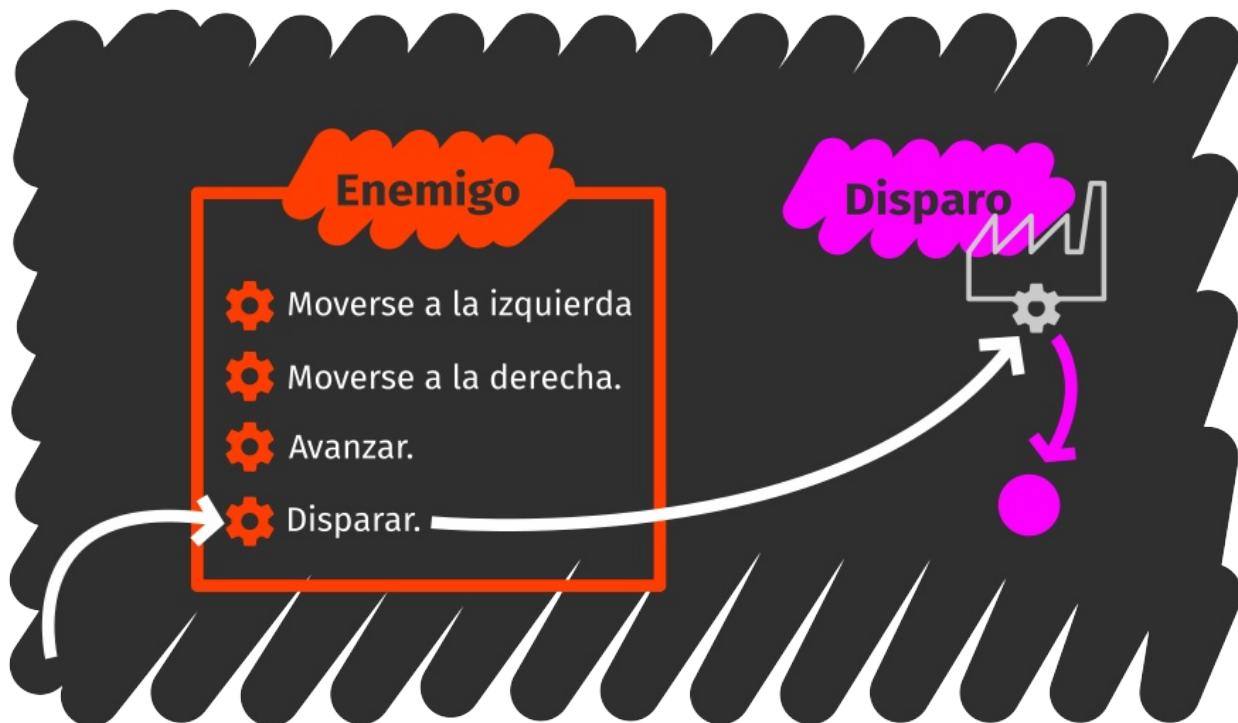
Observa los siguientes ejemplos:

```
true.toString();
3.1415.toFixed(2);
'I want to be a pirate!'.split(' ');
({}).hasOwnProperty('x');
(function (parameter) { return parameter; }).length;
```

Tipos y constructores de objetos

Hemos comentado que JavaScript no permite modelar tipos nuevos y que hace falta "dar un rodeo". Esta es una de las principales diferencias con otros lenguajes orientados a objetos.

Lo que se debe hacer es saltarse el concepto de *tipo* para abordar directamente el de **constructor**.



Vamos a crear dos funciones constructoras: una para puntos y otra para disparos.

```
function newPoint(x, y) {
    var obj = {};
    obj.x = x;
    obj.y = y;

    return obj;
}

function newShot(position, velocity) {
    var obj = {};
    obj._position = position;
    obj._velocity = velocity;
    obj.advance = function () {
        this._position.y += this._velocity;
    };

    return obj;
}
```

La forma de las funciones constructoras es muy similar: crear un objeto vacío, establecer las propiedades del objeto y devolver el nuevo objeto.

Ahora podríamos crear disparos con algo así:

```
// Velocidad positiva para que se mueva hacia abajo.  
var enemyShot = newShot(newPoint(15, 15), 2);  
  
// Velocidad negativa para que se mueva hacia arriba.  
var allyShot = newShot(newPoint(15, 585), -2);  
  
enemyShot !== allyShot;
```

Reaprovechando funcionalidad

El problema con esta aproximación es que estamos creando funciones distintas para comportamientos idénticos: una función por objeto.

```
var s1 = newShot(newPoint(15, 15), 2);  
var s2 = newShot(newPoint(15, 15), 2);  
var s3 = newShot(newPoint(15, 15), 2);  
s1.advance !== s2.advance;  
s2.advance !== s3.advance;  
s3.advance !== s1.advance;
```

Esto es altamente ineficiente, dado que cada función ocupa un espacio distinto en memoria.

Realmente no son necesarias tantas funciones, sino una solamente actuando sobre distintos objetos.

Así que es mejor **crear un objeto que contenga únicamente la API**:

```
var shotAPI = {  
    advance: function () {  
        this._position.y += this._velocity;  
    }  
};
```

Y usarlo en la creación del objeto para compartir los métodos de la API:

```
function newShot(position, velocity) {  
    var obj = {};  
    obj._position = position;  
    obj._velocity = velocity;  
    obj.advance = shotAPI.advance;  
    return obj;  
}
```

Ahora todas las instancias comparten la misma función, pero cada función actúa sobre el objeto correspondiente gracias al valor de `this`:

```
var s1 = newShot(newPoint(15, 15), 2);
var s2 = newShot(newPoint(15, 15), 2);
var s3 = newShot(newPoint(15, 15), 2);
s1.advance === s2.advance; // ahora SÍ son iguales.
s2.advance === s3.advance;
s3.advance === s1.advance;
```

Para hacer todavía más fuerte la asociación entre el constructor y la API, vamos a realizar una pequeña modificación: crear el objeto con la API como una **propiedad de la función constructora**, quedando así todo agrupado en el mismo sitio (la función `newShot`).

```
function newShot(position, velocity) {
    var obj = {};
    obj._position = position;
    obj._velocity = velocity;
    obj.advance = newShot.api.advance;
    return obj;
}

// Una función es un objeto, así que le podemos añadir una propiedad.
newShot.api = {
    advance: function () {
        this._position.y += this._velocity;
    }
};
```

La cadena de prototipos

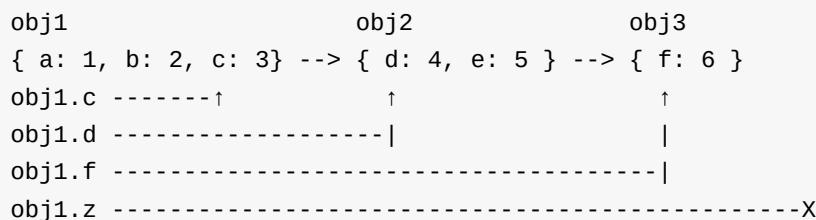
JavaScript posee una característica muy representativa y única del lenguaje: **la cadena de prototipos**.

Puedes experimentar con ella en [Object Playgroun](#)d, una excelente herramienta que te ayudará a visualizarla.

La idea no es complicada: la cadena de prototipos **es una lista de búsqueda para las propiedades**. Cada elemento de la cadena es **prototipo** del objeto anterior.

Cuando accedes a una propiedad de un objeto, esta propiedad se busca en el objeto y, si no se encuentra, se busca en el prototipo del objeto, y así sucesivamente hasta alcanzar la propiedad o el final de esta cadena.

Por ejemplo:



Crear esta jerarquía en JavaScript requiere el uso de `Object.create()`:

```
// La cadena se monta de atrás hacia adelante.  
var obj3 = { f: 6 };  
// Encadenamos obj2 a obj3  
var obj2 = Object.create(obj3);  
obj2.d = 4;  
obj2.e = 5;  
// Encadenamos obj1 a obj2  
var obj1 = Object.create(obj2);  
obj1.a = 1;  
obj1.b = 2;  
obj1.c = 3;  
  
obj1.c;  
obj1.d;  
obj1.f;  
obj1.z; // undefined
```

El método `Object.create()` crea un nuevo objeto vacío (como `{}`) cuyo prototipo es el objeto pasado como parámetro.

Se puede usar el método `hasOwnProperty` para determinar si una propiedad pertenece a un objeto sin atravesar la cadena de prototipos:

```
obj1.hasOwnProperty('c'); // true
obj1.hasOwnProperty('d'); // false
obj1.hasOwnProperty('f'); // false
obj1.hasOwnProperty('z'); // false

obj2.hasOwnProperty('c'); // false
obj2.hasOwnProperty('d'); // true
obj2.hasOwnProperty('f'); // false
obj2.hasOwnProperty('z'); // false

obj3.hasOwnProperty('c'); // false
obj3.hasOwnProperty('d'); // false
obj3.hasOwnProperty('f'); // true
obj3.hasOwnProperty('z'); // false
```

Se puede usar el método `Object.getPrototypeOf()` para obtener el prototipo de un objeto:

```
Object.getPrototypeOf(obj1) === obj2;
Object.getPrototypeOf(obj2) === obj3;
Object.getPrototypeOf(obj3) === Object.prototype;
Object.getPrototypeOf(Object.prototype) === null;
```

Constructores y cadenas de prototipos

Los prototipos se prestan a ser el lugar ideal para contener la API, que es el comportamiento común de todos los objetos de un tipo.

```
var obj = newShot()                               newShot.api
{ _position: { x: 10, y: 10 }, _velocity: 2 } --> { advance: function ... };
obj._position.y -----↑                         ↑
obj.advance -----|                                |
obj.goBack -----x
```

Para crear este enlace, modificaremos nuestro constructor de la siguiente forma:

```
function newShot(position, velocity) {
    // Con esto la API es el prototipo del objeto.
    var obj = Object.create(newShot.api);
    obj._position = position;
    obj._velocity = velocity;

    return obj;
}

newShot.api = {
    advance: function () {
        this._position.y += this._velocity;
    }
};
```

Prueba ahora a crear un nuevo disparo:

```
var shot = newShot({x: 0, y: 0}, 2);
shot; // al inspeccionar shot sólo se muestran las propiedades del objeto.
shot.advance; // pero advance existe en su prototipo.
shot.hasOwnProperty('advance'); // false
Object.getPrototypeOf(shot).hasOwnProperty('advance'); // true
```

Si hacemos esto con todos los constructores, pronto encontraremos un patrón:

1. Crear un objeto para contener la API.
2. Implementar la API como propiedades de este objeto.
3. En el constructor, hacer que este objeto sea el prototipo de un nuevo objeto.
4. Establecer las propiedades del nuevo objeto con el estado.
5. Devolver el nuevo objeto.

Sólo los pasos 2 y 4 involucran diferencias de un constructor a otro, todo lo demás es exactamente igual. Tanto es así, que JavaScript lo tiene en cuenta y viene con los mecanismos para automatizar los pasos 1, 3 y 5.

Primero, JavaScript permite que *cualquier función* pueda usarse como constructor. Por eso, cada vez que escribimos una función, JavaScript crea una **propiedad de la función llamada prototype**, que es un objeto con una única propiedad `constructor` que apunta a la función.

```
function anyFunction() {}
anyFunction.prototype;
anyFunction.prototype.constructor === anyFunction;
```

Esto automatiza el paso 1: ya no es necesario el objeto `api` que preparábamos nosotros manualmente. La propiedad `prototype` es equivalente a la propiedad `api`.

Ahora, al llamar a la función con el operador `new` delante, se crea un **nuevo objeto cuyo prototipo es precisamente la propiedad `prototype`** de la función:

```
var obj = new anyFunction();
var anotherObj = new anyFunction();

// Los objetos son distintos.
obj !== anotherObj;

// Pero sus prototipos son iguales.
Object.getPrototypeOf(obj) === Object.getPrototypeOf(anotherObj);

// Y además son la propiedad prototype de la función.
Object.getPrototypeOf(obj) === anyFunction.prototype;
```

Con esto se automatiza el paso 3: ya no es necesario llamar a `Object.create()` para establecer la cadena de prototipos entre objeto y API (lo conseguimos automáticamente al utilizar el operador `new`).

Finalmente, cuando se llama con `new`, la **función recibe como objeto de contexto (`this`) el elemento que está siendo creado**, lo que nos permite establecer sus atributos.

```
function Hero(name) {
    this.name = name;
    this.sword = null;
    this.shield = null;
}

var hero = new Hero('Link');
hero;
```

Si la función no devuelve nada, el **resultado del operador `new` será el nuevo objeto**. Esto automatiza el paso 5: no es necesario devolver el nuevo objeto, esta devolución se hace implícita al utilizar `new`.

Observa como quedaría el constructor de un objeto punto:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
```

Y el del disparo:

```
function Shot(position, velocity) {
    this._position = position;
    this._velocity = velocity;
}

// El prototipo ya existe, pero le añadimos el método advance()
Shot.prototype.advance = function () {
    this._position.y += this._velocity;
};
```

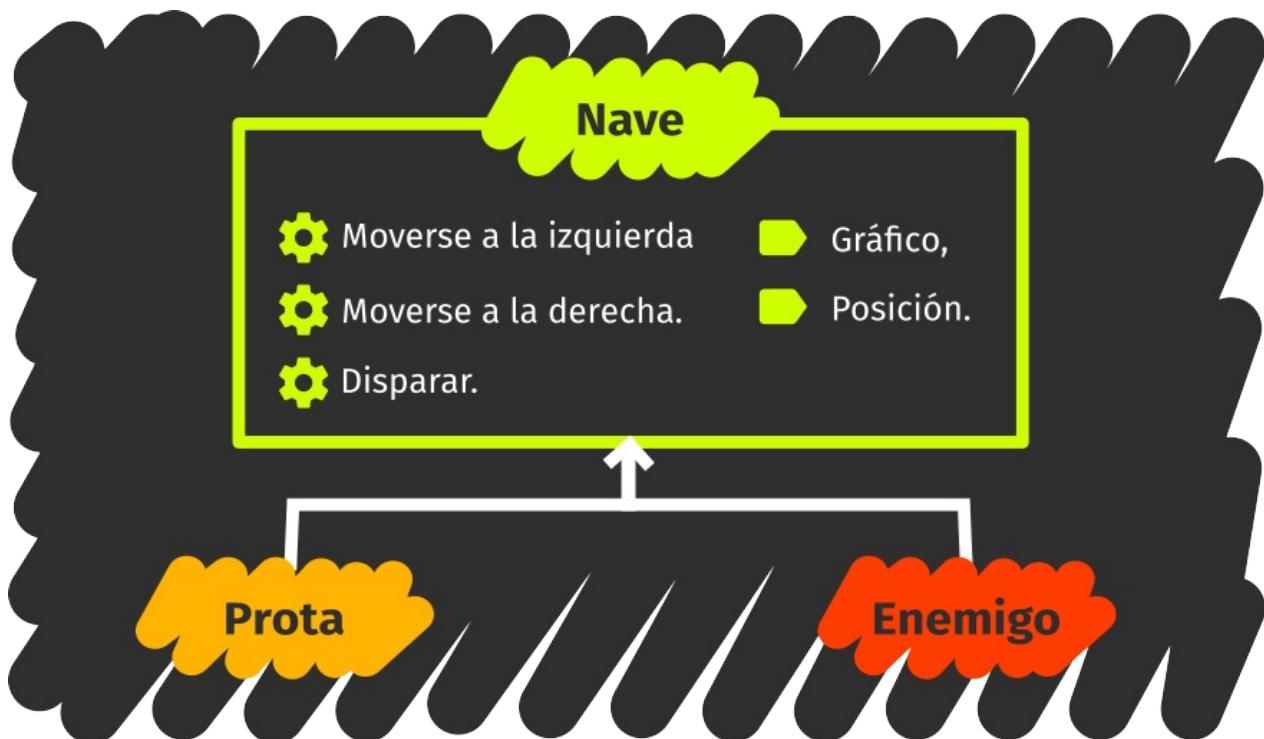
Ahora crear los objetos será cuestión de usar `new`. Emplearemos además nuestro nuevo tipo punto (`Point`) para pasar la posición al disparo:

```
var enemyShot = new Shot(new Point(15, 15), 2);
var allyShot = new Shot(new Point(15, 585), -2);
enemyShot !== allyShot;
```

Herencia

Ya hemos visto cómo crear objetos con atributos y cómo hacerlo eficazmente, usando constructores y la cadena de prototipos.

Veremos ahora cómo crear una **relación de herencia**. Recordemos el ejemplo de los enemigos y la nave protagonista de la lección anterior:



Necesitaremos nuestros puntos y disparos:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}

function Shot(position, velocity) {
    this._position = position;
    this._velocity = velocity;
}

Shot.prototype.advance = function () {
    this._position.y += this._velocity;
};
```

El constructor y los métodos de los enemigos podrían ser:

```
function Enemy(graphic, position, score) {
    this._graphic = graphic;
    this._currentDirection = 'right';
    this._position = position;
    this._score = score;
}

Enemy.prototype.moveLeft = function () { this._position.x -= 2; };
Enemy.prototype.moveRight = function () { this._position.x += 2; };
Enemy.prototype.advance = function () { this._position.y += 2; };

Enemy.prototype.shoot = function () {
    var firePosition = new Position(this._position.x, this._position.y + 10);
    var shot = new Shot(firePosition, 2);
    return shot;
};
```

Y aquí la implementación de la nave aliada:

```
function Ally(position) {
    this._graphic = 'ally.png';
    this._position = position;
}

Ally.prototype.moveLeft = function () { this._position.x -= 2; };
Ally.prototype.moveRight = function () { this._position.x += 2; };

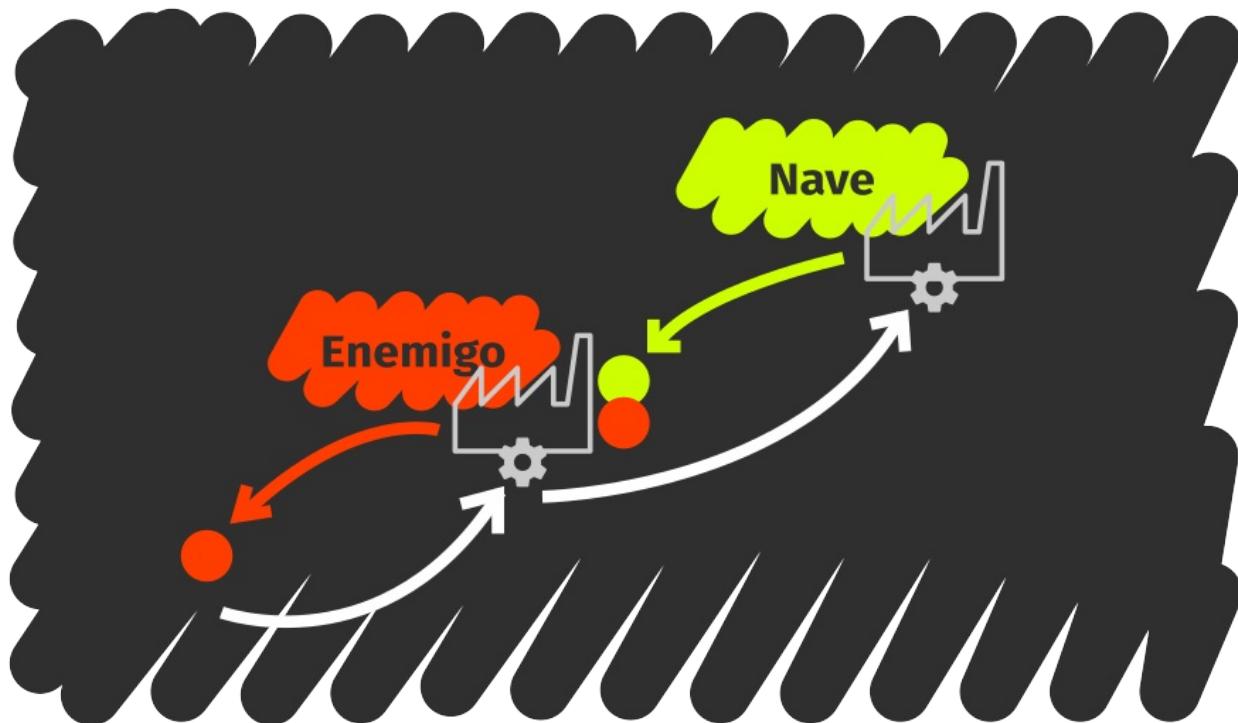
Ally.prototype.shoot = function () {
    var firePosition = new Position(this._position.x, this._position.y - 10);
    var shot = new Shot(firePosition, -2);
    return shot;
};
```

Ahora podemos generalizar y pensar en un constructor que capture las propiedades comunes de ambos tipos:

```
function Ship(graphic, position) {
    this._graphic = graphic;
    this._position = position;
}

Ship.prototype.moveLeft = function () { this._position.x -= 2; };
Ship.prototype.moveRight = function () { this._position.x += 2; };
```

En este caso, probablemente sea mejor no incluir el método de disparar `shoot`, ya que unas naves disparan hacia arriba y otras hacia abajo. Tampoco incluiremos `advance`, puesto que es exclusivo de los enemigos y no de la nave aliada.



Recuerda que ahora los constructores de la nave aliada y los enemigos pedirán primero al constructor de nave que cree una nave y luego la personalizarán.

```
function Enemy(graphic, position, score) {
    Ship.apply(this, [graphic, position]);
    this._currentDirection = 'right';
    this._score = score;
}

function Ally(position) {
    Ship.apply(this, ['ally.png', position]);
}
```

Con `apply` se puede ejecutar una función indicando cuál será su objeto de contexto y sus parámetros.

Con la configuración anterior, las nuevas instancias de enemigos y aliados pasarán primero por el constructor de `Ship`, que establecerá los **atributos comunes** y luego estas instancias serán modificadas cada una por el constructor pertinente para convertirse en enemigos o en aliados.

En cuanto a la API, lo ideal sería contar con una cadena de prototipos de la siguiente manera:

- Los atributos del enemigo (o del aliado) están en la propia instancia.
- La API específica del tipo `Enemy` or `Ally` están en la propiedad `prototype` del constructor de ese tipo.
- La API común a los tipos `Enemy` y `Ally` está en la propiedad `prototype` del constructor `Ship`.

```
var enemy = new Enemy()           Enemy.prototype      Ship.prototype
{ _position: ..., _score: ... } --> { advance: ... } --> { moveLeft: ... }
enemy._score -----↑           ↑                   ↑
enemy.advance -----|           |                   |
enemy.moveLeft -----|         |                   |
```

Como ocurría con el ejemplo en la sección anterior, hay que crear la cadena desde atrás hacia adelante. El enlace entre las instancias y los constructores nos lo proporciona JavaScript al utilizar `new`, pero el enlace entre la propiedad `prototype` de `Enemy` y la de `Ship` **hay que establecerlo manualmente**.

Prueba lo siguiente:

```
// Inspecciona el prototype de Enemy.  
Enemy.prototype;  
  
// Enlaza ambas propiedades prototype.  
Enemy.prototype = Object.create(Ship.prototype);  
  
// Inspecciona la propiedad prototype otra vez y busca diferencias.  
Enemy.prototype;  
  
// Corrige la propiedad constructor.  
Enemy.prototype.constructor = Enemy;  
  
// Añade el método específico del tipo Enemy.  
Enemy.prototype.advance = function () {  
    this._position.y += 2;  
};  
  
// Otro método específico.  
Enemy.prototype.shoot = function () {  
    var firePosition = new Point(this._position.x, this._position.y + 10);  
    var shot = new Shot(firePosition, 2);  
    return shot;  
};
```

Y para el tipo `Ally` :

```
// Lo mismo para el aliado.  
Ally.prototype = Object.create(Ship.prototype);  
Ally.prototype.constructor = Ally;  
  
Ally.prototype.shoot = function () {  
    var firePosition = new Point(this._position.x, this._position.y - 10);  
    var shot = new Shot(firePosition, -2);  
    return shot;  
};
```

Ahora sí, ya podemos crear un enemigo y un aliado usando sus constructores:

```
var enemy = new Enemy('enemy1.png', new Point(10, 10), 40);  
var ally = new Ally(new Point(10, 590));  
  
Object.getPrototypeOf(ally) === Ally.prototype;  
Object.getPrototypeOf(enemy) === Enemy.prototype;  
Ally.prototype !== Enemy.prototype;  
Object.getPrototypeOf(Ally.prototype) === Object.getPrototypeOf(Enemy.prototype);  
Object.getPrototypeOf(Ally.prototype) === Ship.prototype;
```

También podemos comprobar dónde está cada propiedad:

```
enemy.hasOwnProperty('_score');
enemy.hasOwnProperty('advance');
enemy.hasOwnProperty('moveLeft');

Enemy.prototype.hasOwnProperty('_score');
Enemy.prototype.hasOwnProperty('advance');
Enemy.prototype.hasOwnProperty('moveLeft');

Ship.prototype.hasOwnProperty('_score');
Ship.prototype.hasOwnProperty('advance');
Ship.prototype.hasOwnProperty('moveLeft');
```

Polimorfismo

Las relaciones de herencia que acabamos de establecer nos permiten decir que un enemigo es una instancia del tipo `Enemy`, pero también lo es del tipo `Ship`. Una misma instancia tiene **múltiples formas gracias a la herencia**. En programación orientada a objetos a esto se lo llama **polimorfismo**.

Alternativamente, podemos decir que un enemigo es una instancia de `Enemy` porque tiene la API de `Enemy`, o que es una instancia de `Ship` porque tiene la API de `Ship`. Esto es equivalente a decir que las propiedades `prototype` de `Enemy` y `Ship` están en la cadena de prototipos del objeto.

El operador `instanceof` devuelve verdadero si la propiedad `prototype` de la función a la derecha del operador está en la cadena de prototipos del objeto a la izquierda del operador.

```
enemy instanceof Enemy; // Enemy.prototype es el primer eslabón.
enemy instanceof Ship; // Ship.prototype es el segundo.
enemy instanceof Object; // Object.prototype, el tercero.

enemy instanceof Ally; // Ally.prototype no está en la cadena.
```

En lo referente al estado, resulta conveniente saber qué constructor ha construido el objeto para conocer de un vistazo los atributos que contendrá el mismo. Esto es equivalente a determinar cuál es la función cuya propiedad `prototype` es el **primer eslabón** de la cadena de prototipos.

Dado que los objetos prototipo vienen de serie con una propiedad `constructor`, que por defecto apunta a la función que posee al objeto prototipo, basta con acceder a la propiedad `constructor` a través de la instancia.

```
enemy.constructor;
enemy.constructor === Enemy; // fue construido por Enemy, no por Ship.
enemy.constructor !== Ship; // es cierto que Ship fue utilizado, pero nada más.
```

Duck typing

In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.

[Alex Martelli sobre polimorfismo](#)

La cita se refiere a que más que comprobar si algo es una instancia de un tipo, se debería comprobar si tiene la funcionalidad que es necesaria.

JavaScript es tan dinámico que el operador `instanceof` y la propiedad `constructor` sólo tienen sentido si se siguen las convenciones que acabamos de ver.

Nada nos impide borrar la propiedad `constructor` de un prototipo o sobreescribirla en un objeto determinado. De hecho, en las nuevas versiones de JavaScript, el prototipo de un objeto puede cambiar después de que el objeto haya sido construido.

El modelo de ejecución de JavaScript

JavaScript se caracteriza por presentar un **modelo de ejecución asíncrono**, donde los resultados no se encuentran disponibles inmediatamente, sino que lo estarán en un futuro.

Ámbito y *hoisting*

Como en muchos lenguajes, los nombres de las variables pueden reutilizarse y guardar valores distintos, siempre y cuando se encuentren en **ámbitos distintos**.

El ámbito o *scope* de una variable es la porción de código donde puede ser utilizada. Variables con el mismo nombre en ámbitos distintos son variables distintas.

El ámbito en JavaScript es el **cuerpo de la función**, delimitado entre el par de llaves `{` y `}` que siguen a la lista de parámetros de la función.

```
function introduction() {
    // Esta es la variable text.
    var text = 'I\'m Ziltoid, the Omniscent.';
    greetings();
    console.log(text);
}

function greetings(list) {
    // Y esta es otra variable text DISTINTA.
    var text = 'Greetings humans!';
    console.log(text);
}

introduction();
```

En JavaScript, las funciones pueden definirse dentro de otras funciones y, de esta forma, anidar ámbitos.

El anidamiento de funciones es útil cuando se quieren usar **funciones auxiliares**, normalmente cortas.

```
function getEven(list) {
    function isEven(n) {
        return n % 2 === 0;
    }
    return list.filter(isEven);
}

getEven([1, 2, 3, 4, 5, 6]);
```

Como el ámbito es el de la función, el mismo nombre en una función anidada se puede referir a dos cosas:

- 1) Si se usa con `var`, se estará declarando **otra variable distinta**:

```
function introduction() {
    // Esta es una variable text.
    var text = 'I'm Ziltoid, the Omniscient.';

    function greetings(list) {
        // Y esta es OTRA variable text distinta.
        var text = 'Greetings humans!';
        console.log(text);
    }

    greetings();
    console.log(text);
}

introduction();
```

En el caso anterior, decimos que la variable `text` de la función anidada `greetings` *oculta a la variable `text` de la función `introduction`*.

Recuerda que para introducir una nueva variable hay que declararla con `var` antes de usarla (o al mismo tiempo que se asigna).

- 2) Si se omite la palabra `var`, no se crea una nueva variable, sino que se **reutiliza** la que ya existía.

```
function introduction() {
    // Esta es una variable text.
    var text = 'I\'m Ziltoid, the Omniscient.';

    function greetings(list) {
        // Esta es la MISMA variable text que la de afuera.
        text = 'Greetings humans!';
        console.log(text);
    }

    greetings();
    console.log(text);
}

introduction();
```

Hoisting

Hoisting significa "elevación", y en el contexto de la programación nos referimos a un mecanismo que emplean algunos lenguajes respecto a la declaración de nombres.

En JavaScript da igual en qué punto de la función se declara una variable. JavaScript asumirá cualquier declaración como si ocurriese al comienzo de la función.

Es decir que esto:

```
function f() {
    for (var i = 0; i < 10; i++) {
        for (var j = 0; j < 10; j++) {
            console.log('i: ', i, ' j: ', j);
        }
    }
}
```

Es equivalente a esto:

```
function f() {
    var i;
    var j;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 10; j++) {
            console.log('i: ', i, ' j: ', j);
        }
    }
}
```

Fíjate que JavaScript **alza la declaración** de la variable (la lleva al principio), no la inicialización. Es por eso que el siguiente código no falla, pero imprime `undefined`:

```
function f() {
    console.log(i);
    var i = 5;
}
f();
```

En **modo estricto**, usar una variable que no ha sido declarada produce un error.

```
function f() {
    console.log(i);
    i = 5;
}
f();
```

Con las **declaraciones de funciones** esto no pasa: cuando una declaración de función se alza, se alza entera, *definición incluida*.

```
function getEven(list) {
    return list.filter(isEven);

    function isEven(n) {
        return n % 2 === 0;
    }
}

getEven([1, 2, 3, 4, 5, 6]);
```

Esto permite una forma de ordenar el código que quizá sea más clara, situando las funciones auxiliares a continuación de las funciones que las utilizan.

Así, viendo sólo la primera línea de la función, ya podemos conocer qué es lo que realiza.

```
return list.filter(isEven);
```

Y, si aún tenemos dudas, podemos seguir leyendo e investigar qué hace la función auxiliar.

```
function isEven(n) {
    return n % 2 === 0;
}
```

Nótese que para que esta manera de escribir código sea clara, el nombre que utilicemos en las funciones auxiliares tiene que ser adecuado, descriptivo, y dar una pista sobre cuál es el valor de retorno.

Closures

Las funciones son datos y se crean cada vez que se encuentra una instrucción

`function`. De esta forma, podemos crear una función que devuelva funciones.

```
function buildFunction() {
    return function () { return 42; };
}

var f = buildFunction();
var g = buildFunction();

typeof f === 'function';
typeof g === 'function';

f();
g(); // Las funciones hacen lo mismo...

f !== g; // ...pero NO son la misma función
```

Por sí solo, este no es un mecanismo muy potente, pero sabiendo que una función anidada puede acceder a las variables de los ámbitos superiores, podemos hacer algo así:

```
function newDie(sides) {
    return function () {
        return Math.floor(Math.random() * sides) + 1;
    };
}
var d100 = newDie(100);
var d20 = newDie(20);

d100 !== d20; // distintas, creadas en dos llamadas distintas a newDie.

d100();
d20();
```

En JavaScript, las funciones **retienen el acceso a las variables en ámbitos superiores**. Una función que se refiere a alguna de las variables en ámbitos superiores se denomina **closure** o clausura.

Esto **no afecta al valor de `this`**, que seguirá siendo el destinatario del mensaje.

Métodos, closures y `this`

Considera el siguiente ejemplo:

```
var diceUtils = {
    history: [], // lleva el histórico de tiradas.

    newDie: function (sides) {
        return function die() {
            var result = Math.floor(Math.random() * sides) + 1;
            this.history.push([new Date(), sides, result]);
            return result;
        }
    }
}
```

La intención es poder crear dados y llevar un registro de todas las tiradas que se hagan con estos dados.

Sin embargo, esto no funciona:

```
var d10 = diceUtils.newDie(10);
d10(); // ¡error!
```

Y es así porque `this` siempre es el destinatario del mensaje y `d10` se está llamando como si fuera una función y no un método.

Recuerda que podemos hacer que cualquier función tomara un valor forzoso como `this` con `.apply()`; por lo que esto sí funciona, pero no es muy conveniente:

```
d10.apply(diceUtils);
d10.apply(diceUtils);
diceUtils.history;
```

Lo que tenemos que hacer es que la función `die` dentro de `newDie` se refiera al `this` del ámbito superior, no al suyo.

Puedes lograr esto de dos maneras. La primera es un mero juego de variables, guardando el `this` en una variable auxiliar (en este caso, `self`):

```
var diceUtils = {
    history: [], // Lleva el histórico de dados.

    newDie: function (sides) {
        var self = this; // self es ahora el destinatario de newDie.

        return function die() {
            var result = Math.floor(Math.random() * sides) + 1;
            // Usando self nos referimos al destinatario de newDie.
            self.history.push([new Date(), sides, result]);
            return result;
        }
    }
}
```

Esto sí funciona y es mucho más conveniente:

```
var d10 = diceUtils.newDie(10);
var d6 = diceUtils.newDie(6);
d10();
d6();
d10();
diceUtils.history;
```

La segunda forma es usando el método `bind` de las funciones.

El método `bind` de una función devuelve otra función cuyo `this` será el primer parámetro de `bind`. De este modo:

```
var diceUtils = {
    history: [], // Lleva el histórico de dados.

    newDie: function (sides) {
        return die.bind(this); // una nueva función que llamará a die con su
                               // destinatario establecido al primer parámetro.

        function die() {
            var result = Math.floor(Math.random() * sides) + 1;
            this.history.push([new Date(), sides, result]);
            return result;
        }
    }
}
```

Las dos formas son ampliamente utilizadas, pero la segunda se ve escrita muchas veces de este modo:

```
var diceUtils = {
    history: [], // Lleva el histórico de dados.

    newDie: function (sides) {
        return function die() {
            var result = Math.floor(Math.random() * sides) + 1;
            this.history.push([new Date(), sides, result]);
            return result;
        }.bind(this); // el bind sigue a la expresión de función.
    }
}
```

Módulos

Esta sección presenta la característica **módulos**, que es específica de Node.

Una de las principales desventajas de JavaScript ([hasta la próxima versión](#)) es que no hay forma de organizar el código en módulos.

Los módulos sirven para aislar funcionalidad relacionada: tipos, funciones, constantes, configuración...

Node *sí tiene módulos* y, afortunadamente, existen **herramientas que simulan módulos** como los de Node en el navegador.

En Node, los ficheros JavaScript acabados en `.js` son módulos. Node nos permite exponer o **exportar funcionalidad** de un módulo, poniéndola dentro del objeto

```
module.exports :  
  
// En diceUtils.js  
"use strict"; // pone el módulo en modo estricto.  
  
var history = [];  
  
function newDie(sides) {  
    return function die() {  
        var result = Math.floor(Math.random() * sides) + 1;  
        history.push([new Date(), sides, result]);  
        return result;  
    };  
}  
  
// ¡Lo que se exporta realmente es el objeto module.exports!  
module.exports.newDie = newDie;  
module.exports.history = history;
```

Realmente, lo que se exporta es **siempre** `module.exports`, que en principio es un objeto vacío:

```
typeof module.exports;  
module.exports;
```

Ahora podemos ahora **importar ese módulo** desde otro:

```
// En cthulhuRpg.js
"use strict";

var diceUtils = require('./diceUtils');
var d100 = diceUtils.newDie(100);

var howard = {
    sanity: 45,
    sanityCheck: function () {
        if (d100() <= this.sanity) {
            console.log('Horrible, pero lo superaré. Estuvo cerca.');
        } else {
            console.log(
                '¡Ph\nglui mglw\`nafh Cthulhu R\`lyeh wgah\`nagl fhtagn!');
        }
    }
};
howard.sanityCheck();
```

Para importar un módulo desde otro hay que pasar a `require` la **ruta relativa** hasta el fichero del módulo que queremos importar.

Si en lugar de una ruta, pasamos un nombre, accederemos a la **funcionalidad que viene por defecto** con Node (los módulos que forman parte de la librería estándar, como `path` o `fs`) o la que instalamos de terceras partes (por ejemplo, módulos instalados con el gestor de paquetes npm). Usaremos esta forma en un par de ocasiones más adelante.

Diferencias entre ámbitos en Node y en el navegador

Se ha dicho que el ámbito en JavaScript es equivalente a la función, pero sabemos que podemos abrir una consola o un fichero y empezar a declarar variables sin necesidad de escribir una función.

Esto es así porque estamos usando el **ámbito global**. El ámbito global está disponible tanto en el navegador como en Node.

```
// Esta es una variable text en el ámbito GLOBAL.  
var text = 'I'm Ziltoid, the Omniscient.';  
  
// Esta es una función en el ámbito GLOBAL.  
function greetings(list) {  
    // Esta es OTRA variable text en el ámbito de la función.  
    var text = 'Greetings humans!';  
    console.log(text);  
}  
  
greetings();  
console.log(text);
```

Sin embargo, existe una peculiaridad en Node. El ámbito global es realmente *local al fichero*. Esto quiere decir que:

```
// En a.js, text es visible únicamente dentro del FICHERO.  
"use strict";  
var text = 'I'm Ziltoid, the Omniscient.';  
  
// En b.js, text es visible únicamente dentro del FICHERO.  
"use strict";  
var text = 'Greetings humans!';  
  
// En una consola iniciada en el mismo directorio que a y b  
require('./a');  
require('./b');  
text;
```

Programación asíncrona y eventos

Prueba el siguiente ejemplo (copia, pega y espera 5 segundos):

```
var fiveSeconds = 5 * 1000; // en milisegundos.  
  
// Esto ocurre ahora.  
console.log('T: ', new Date());  
  
setTimeout(function () {  
    // Esto ocurre pasados 5 segundos.  
    console.log('T + 5 segundos: ', new Date());  
}, fiveSeconds);  
  
// Esto ocurre inmediatamente después  
console.log('T + delta: ', new Date());
```

Como puedes comprobar, el mensaje se completa pasados 5 segundos porque lo que hace `setTimeout` es llamar a la función tan pronto como pasen el número de milisegundos indicados.

Decimos que una función es un **callback** si se llama en algún momento futuro –es decir, **asíncronamente**– para informar de algún resultado.

En el ejemplo de `setTimeout`, el resultado es que ha pasado la cantidad de tiempo especificada.

Eventos

En esta sección veremos el **módulo** `readline`, que es específico de Node.

La programación asíncrona en JavaScript y otros lenguajes se usa para **modelar eventos**, principalmente esperas por entrada y salida. En otras palabras: hitos que ocurren pero que no sabemos *cuándo* ocurren.

La entrada y salida –a partir de ahora IO (del inglés *input / output*)– no sólo supone lectura de ficheros o peticiones a la red, también incluye esperar una acción del usuario.

Como ejemplo, vamos a implementar una consola de diálogo por líneas. Usaremos el módulo `readline`, que es parte de la librería estándar que viene con Node:

```
// En conversational.js
"use strict";

var readline = require('readline');

var cmd = readline.createInterface({
  input: process.stdin, // así referenciamos la consola como entrada.
  output: process.stdout, // y así, como salida.
  prompt: '(Jº□º) J ' // lo que aparece para esperar la entrada del usuario.
});
```

Lanza ese programa con Node y verás que no hace nada, **pero tampoco termina**. Esto es típico de los programas asíncronos: el programa queda esperando a que pase algo. Pulsa `ctrl+c` para terminar el programa.

Ahora prueba:

```
// Añade al final de conversational.js
console.log('Escribe algo y pulsa enter');
cmd.prompt(); // pide que el usuario escriba algo.

cmd.on('line', function (input) {
  console.log('Has dicho "' + input + '"');
  cmd.prompt(); // pide que el usuario escriba algo.
});

});
```

Lo que has conseguido es **escuchar el evento** `line` que se produce **cada vez que en la entrada se recibe un carácter de cambio de línea**.

Hablando de eventos, la función que se ejecuta asíncronamente recibe el nombre de **listener**, pero tampoco es raro que se la llame **callback**.

Se habla de "**registrar un listener** para un evento", "subscribirse a un evento" o "**escuchar un evento**" a utilizar el mecanismo que permite asociar la ejecución de una función con dicho evento.

Con todo, aún no se puede salir del programa anterior. Necesitamos algunos cambios más:

```
// Añade a conversational.js
cmd.on('line', function (input) {
  if (input === 'salir') {
    cmd.close();
  }
});

cmd.on('close', function () {
  console.log('¡Nos vemos!');
  process.exit(0); // sale de node.
});
```

Hemos añadido un segundo *listener* al evento `line` y **ambos se ejecutarán**. El primero gestiona el funcionamiento por defecto (que es repetir lo que el usuario ha introducido) y el segundo trata específicamente el comando `salir`.

Si la línea es exactamente `salir`, cerraremos la interfaz de línea de comandos. Esto produce un evento `close` y, cuando lo recibamos, utilizaremos el *listener* de ese evento para terminar el programa.

El método `on` es un segundo nombre para `addListener`.

Igual que podemos añadir un *listener*, también podemos eliminarlo con `removeListener`, y quitarlos todos con `removeAllListeners`.

Podemos escuchar un evento **sólo una vez** con `once`.

Emisores de eventos

Ahora veremos la clase `EventEmitter`, que también es específica de Node.

Los eventos no son un mecanismo estándar de JavaScript. Son una forma conveniente de modelar determinados tipos de problema, pero un objeto JavaScript, por sí solo, **no tiene API para emitir eventos**.

En Node contamos con diversas alternativas con el fin de que los objetos emitan eventos:

- Implementar nuestra propia API de eventos.
- Hacer que nuestros objetos **usen** una instancia de `EventEmitter`.
- Hacer que nuestros objetos **sean instancias** de `EventEmitter`.

La primera supondría crear nuestro propio método `on` y los mecanismos para emitir eventos. La segunda y la tercera usan la clase `EventEmitter`, que ya implementa esta API.

Este es un ejemplo de la opción 3 –el cual aprovechamos para repasar la herencia:

```
var events = require('events');
var EventEmitter = events.EventEmitter;

function Ship() {
  EventEmitter.apply(this);
  this._ammunition = 'laser charges';
}

Ship.prototype = Object.create(EventEmitter.prototype);
Ship.prototype.constructor = Nave;

var ship = new Ship();
ship.on; // ¡existe!
```

Ahora que la nave puede emitir eventos, vamos a hacer que dispare y que emita un evento.

```
Ship.prototype.shoot = function () {
    console.log('PICHUM!');
    this.emit('shoot', this._ammunition); // parte de la API de EventEmitter.
};

ship.on('shoot', function (ammunition) {
    console.log('CENTRO DE MANDO. La nave ha disparado:', ammunition);
});

ship.shoot();
```

Emitir un evento consiste en llamar al método `emit`, que hará que se ejecuten los *listeners* que escuchan tal evento.

Los eventos son increíblemente útiles para modelar interfaces de usuario de forma genérica.

Para ello, los modelos deben **publicar** qué les ocurre: cómo cambian, qué hacen... Todo a **base de eventos**. Las interfaces de usuario se **suscribirán** a estos eventos y proporcionarán la información visual adecuada.

Este modelo permite además que varias interfaces de usuario funcionen al mismo tiempo, todas escuchando los mismos eventos. Sin embargo, también permite dividir la interfaz en otras más especializadas, cada una escuchando un determinado conjunto de eventos.

Ejercicios guiados

Prueba estos ejemplos y trata de responder a las preguntas. Si te atascas con lo que hace una función, busca en Internet la función acompañada de "mdn".

1. Valores booleanos en JavaScript.

En JavaScript cualquier valor puede considerarse verdadero o falso según el contexto. Por ejemplo, el `0` es considerado `false` y cualquier número distinto de `0` es `true`:

```
var v = 0; // después de esta prueba, cambia el valor de v por otro número.  
!!v; // la doble negación al comienzo lo convierte en booleano.
```

Descubre qué valores son ciertos y cuales falsos para todos los tipos: números, cadenas, objetos, funciones y `undefined`.

2. Expresiones booleanas en JavaScript.

En JavaScript, las expresiones booleanas son *vagas*, esto significa que en cuanto el intérprete de JavaScript sabe lo que va a valer la expresión, dejamos de evaluar. Por ejemplo, ¿qué crees que le pasará a la siguiente expresión?

```
var hero = { name: 'Link', weapon: null };  
console.log('Hero weapon power is:', hero.weapon.power);
```

Pero, ¿y ahora?

```
var hero = { name: 'Link', weapon: null };  
if (hero.weapon && hero.weapon.power) {  
    console.log('Hero weapon power is:', hero.weapon.power);  
} else {  
    console.log('The hero has no weapon.');
```

En caso de expresiones `&&` (*and* o *y*), la evaluación termina tan pronto como encontramos un término falso.

En caso de expresiones `||` (*or* u *o*), la evaluación termina tan pronto como encontramos un término verdadero.

3. El resultado de las expresiones booleanas.

Contra el sentido común, el resultado de una expresión booleana no es un booleano sino el último término evaluado. Recuerda que la evaluación es *vaga* y JavaScript deja de evaluar tan pronto como puede determinar el resultado de la expresión. Con esto en cuenta, trata de predecir el resultado de las siguientes expresiones:

```
var v;
function noop() { return; };

1 && true && { name: 'Link' };
[] && null && "Spam!";
null || v || noop || true;
null || v || void "Eggs!" || 0;
```

4. Parámetros por defecto.

Puedes ver una aplicación real de lo anterior en esta función para llenar números. En JavaScript no hay parámetros por defecto, pero los parámetros omitidos tienen el valor especial `undefined` que es falso.

```
function pad(target, targetLength, fill) {
  var result = target.toString();
  var targetLength = targetLength || result.length + 1;
  var fill = fill || '0';
  while (result.length < targetLength) {
    result = fill + result;
  }
  return result;
}

// intenta predecir el resultado de las siguientes llamadas
pad(3);
pad(2, 5);
pad(2, 5, '*');
```

5. Buenas prácticas en el diseño de APIs.

Se ha dicho muchas veces que el estado no se debería exponer pero siempre se acaba enseñando este tipo de modelado para los puntos:

```
var p = { x: 5, y: 5 };

function scale(point, factor) {
    point.x = point.x * factor;
    point.y = point.y * factor;
    return p;
}

scale(p, 10);
```

La implementación correcta sería:

```
var p = {
    _x: 5,
    _y: 5,
    getX: function () {
        return this._x;
    },
    getY: function () {
        return this._y;
    },
    setX: function (v) {
        this._x = v;
    },
    setY: function (v) {
        this._y = v;
    }
};

function scale(point, factor) {
    point.setX(point.getX() * factor);
    point.setY(point.getY() * factor);
    return p;
}

scale(p, 10);
```

Pero reconócelo, escribir tanto es un rollo soberano.

6. Propiedades computadas al rescate.

JavaScript permite definir un tipo especial de propiedades llamadas normalmente *propiedades computadas* de esta guisa:

```
var p = {
  _x: 5,
  _y: 5,
  get x() {
    return this._x;
  },
  get y() {
    return this._y;
  },
  set x(v) {
    this._x = v;
  },
  set y(v) {
    this._y = v;
  }
};

function scale(point, factor) {
  point.x = point.x * factor;
  point.y = point.y * factor;
  return p;
}

scale(p, 10);
```

Escribirlo sigue siendo un muermo (menos mal que has estudiado como hacer factorías de objetos) pero utilizarlo es mucho más claro. Así, si ahora decides que sería mejor exponer el nombre de los ejes en mayúscula, puedes hacer:

```
var p = {
    _x: 5,
    _y: 5,
    get X() {
        return this._x;
    },
    get Y() {
        return this._y;
    },
    set X(v) {
        this._x = v;
    },
    set Y(v) {
        this._y = v;
    }
};

function scale(point, factor) {
    point.X = point.X * factor;
    point.Y = point.Y * factor;
    return p;
}

scale(p, 10);
```

¿Se te ocurre la manera de hacer que una propiedad pueda ser de sólo lectura? Es decir, que su valor no pueda cambiarse (asumiendo que el usuario no accederá a las propiedades que comiencen por '_').

Si quisieras añadir una propiedad a un objeto ya existente tendrías que utilizar [Object.defineProperty\(\)](#):

```
var point = {};
Object.defineProperty(point, '_x', { value: 5, writable: true });
Object.defineProperty(point, '_y', { value: 5, writable: true });
Object.defineProperty(point, 'x', {
  get: function () {
    return this._x;
  },
  set: function (v) {
    this._x = v;
  }
});
Object.defineProperty(point, 'y', {
  get: function () {
    return this._y;
  },
  set: function (v) {
    this._y = v;
  }
});
point; // no se observan propiedades...
point.x; // ...pero aquí están.
point.y;
```

¿Te atreves a decir por qué cuando inspeccionamos el objeto no aparecen sus propiedades? ¿Cómo podrías arreglarlo? ¿Cómo harías para que sólo se vieran las propiedades que son parte de la API?

No te lances a usar `Object.defineProperty()` si no tienes **muy claro** qué significan los términos **configurable**, **enumerable** y **writable**.

7. Usando funciones como si fueran métodos.

Hemos visto que cualquier función puede usarse como un método si se referencia como una propiedad de un objeto y entonces se llama. Pero lo cierto es que también puedes hacer que una función cualquiera, sin estar referenciada desde una propiedad, pueda ser usada como el método de un objeto si indicamos explícitamente cual es el objeto destinatario. Esto puede hacerse con `.apply()` y con `.call()`.

```
var ship = { name: 'Death Star' };

function fire(shot) {
  console.log(this.name + ' is firing: ' + shot.toUpperCase() + '!!!!');
}

ship.fire; // ¿qué crees que será esto?
fire.apply(ship, ['pichium']);
fire.call(ship, 'pañum');
```

¿Cuál es la diferencia entre `.apply()` y `.call()` ?

8. Propiedades dinámicas.

La notación corchete para acceder a las propiedades de un objeto es especialmente útil para acceder a propiedades de manera genérica. Por ejemplo, imagina el siguiente código:

```
var hero = {
  name: 'Link',
  hp: 10,
  stamina: 10,
  weapon: { name: 'sword', effect: { hp: -2 } }
};

var enemy = {
  name: 'Ganondorf',
  hp: 20,
  stamina: 5,
  weapon: { name: 'wand', effect: { hp: -1, stamina: -5 } }
};

function attack(character, target) {
  if (character.stamina > 0) {
    console.log(character.name + ' uses ' + character.weapon.name + '!');
    applyEffect(character.weapon.effect, target);
    character.stamina--;
  } else {
    console.log(character.name + ' is too tired to attack!');
  }
}

function applyEffect(effect, target) {
  // Obtiene los nombres de las propiedades del objeto. Búscalos en la MDN.
  var propertyNames = Object.keys(effect);
  for (var i = 0; i < propertyNames.length; i++) {
    var name = propertyNames[i];
    target[name] += effect[name];
  }
}

attack(hero, enemy);
attack(enemy, hero);
attack(hero, enemy);
attack(enemy, hero);
attack(hero, enemy);
```

¿Podrías modificar el efecto del arma del héroe para incapacitar al enemigo pero no matarlo ni dañarlo? Intenta hacerlo sin reescribir el ejemplo entero, es decir, continuando desde el término del ejemplo.

9. Objetos como algo más que objetos.

Los objetos de JavaScript no solo sirven para modelar los objetos de la programación orientada a objetos sino que permiten realizar clasificaciones por nombre. Un histograma, es decir un conteo de un conjunto con repeticiones, es un ejemplo clásico de la utilidad de un objeto JavaScript:

```
function wordHistogram(text) {  
    var wordList = text.split(' ');\n    var histogram = {};\n    for (var i = 0; i < wordList.length; i++) {\n        var word = wordList[i];\n        if (!histogram.hasOwnProperty(word)) {\n            histogram[word] = 0;\n        }\n        histogram[word]++;
    }\n    return histogram;\n}
```

Prueba a usar la función por ti mismo.

Lo que JavaScript llama objetos se conoce en otros lenguajes de programación como mapas o diccionarios y a los nombres de las propiedades se los llama *claves*.

¿Puedes pensar en al menos una aplicación más?

10. Funciones como parámetros.

Las listas de JavaScript tienen algunos métodos que aceptan funciones como parámetros, por ejemplo `.forEach()`. De hecho es común encontrar `.forEach()` cuando se tiene la certeza de que se van a recorrer **todos** los elementos de una lista.

```
function wordHistogram(text) {  
    var wordList = text.split(' ');\n    var histogram = {};\n    wordList.forEach(function (word) {  
        if (!histogram.hasOwnProperty(word)) {\n            histogram[word] = 0;\n        }\n        histogram[word]++;
    });\n    return histogram;\n}  
  
var poem = 'Todo pasa y todo queda, ' +  
          'pero lo nuestro es pasar, ' +  
          'pasar haciendo caminos, ' +  
          'caminos sobre la mar';  
  
wordHistogram(poem);
```

El resultado no es correcto porque al separar las palabras por los espacios estás dejando caracteres que no forman palabras como parte de ellas. Puedes arreglarlo si en vez de partir el texto por los espacios usas una [expresión regular](#) para partir el texto por los límites de las palabras:

```
function wordHistogram(text) {  
  var wordList = text.split(/\b/); // Eso entre / / es una expresión regular.  
  var histogram = {};  
  wordList.forEach(function (word) {  
    if (!histogram.hasOwnProperty(word)) {  
      histogram[word] = 0;  
    }  
    histogram[word]++;  
  });  
  return histogram;  
}  
  
var poem = 'Todo pasa y todo queda, ' +  
          'pero lo nuestro es pasar, ' +  
          'pasar haciendo caminos, ' +  
          'caminos sobre la mar';  
  
wordHistogram(poem);
```

Pero ahora tendrás cosas que no son palabras (como espacios y comas). Puedes filtrar una lista con [.filter\(\)](#) :

```
function isEven(n) { return n % 2 === 0; }  
[1, 2, 3, 4, 5, 6].filter(isEven);
```

Y así quitar lo que no sean palabras:

```
function isWord(candidate) {
  return /\w+/.test(candidate);
}

function wordHistogram(text) {
  var wordList = text.split(/\b/);
  wordList = wordList.filter(isWord);
  var histogram = {};

  wordList.forEach(function (word) {
    if (!histogram.hasOwnProperty(word)) {
      histogram[word] = 0;
    }
    histogram[word]++;
  });
  return histogram;
}

var poem = 'Todo pasa y todo queda, ' +
  'pero lo nuestro es pasar, ' +
  'pasar haciendo caminos, ' +
  'caminos sobre la mar';

wordHistogram(poem);
```

También deberías normalizar las palabras (pasarlas a minúsculas por ejemplo) para no encontrarnos con entradas distintas en el histograma para la misma palabra. Para transformar una lista en otra lista con el mismo número de elementos usamos `.map()`.

```
function isWord(candidate) {
    return /\w+/.test(candidate);
}

function toLowerCase(word) {
    return word.toLowerCase();
}

function wordHistogram(text) {
    var wordList = text.split(/\b/);
    wordList = wordList.filter(isWord);
    wordList = wordList.map(toLowerCase);
    var histogram = {};

    wordList.forEach(function (word) {
        if (!histogram.hasOwnProperty(word)) {
            histogram[word] = 0;
        }
        histogram[word]++;
    });
    return histogram;
}

var poem = 'Todo pasa y todo queda, ' +
    'pero lo nuestro es pasar, ' +
    'pasar haciendo caminos, ' +
    'caminos sobre la mar';

wordHistogram(poem);
```

Una última función te permite transformar una lista en un sólo valor. Esto es precisamente el histograma, una clasificación de todos los valores de la lista. Esta transformación se consigue mediante `.reduce()` :

```
function isWord(candidate) {
    return /\w+/.test(candidate);
}

function toLowerCase(word) {
    return word.toLowerCase();
}

function buildHistogram(inProgressHistogram, word) {
    if (!inProgressHistogram.hasOwnProperty(word)) {
        inProgressHistogram[word] = 0;
    }
    inProgressHistogram[word]++;
    return inProgressHistogram;
}

function wordHistogram(text) {
    var emptyHistogram = {};
    return text.split(/\b/)
        .filter(isWord)
        .map(toLowerCase)
        .reduce(buildHistogram, emptyHistogram);
}

var poem = 'Todo pasa y todo queda, ' +
    'pero lo nuestro es pasar, ' +
    'pasar haciendo caminos, ' +
    'caminos sobre la mar';

wordHistogram(poem);
```

11. Número variables de parámetros

Fíjate en esto:

```
console.log('I\'m', 'Ziltoid');
console.log('I\'m', 'Ziltoid', 'the', 'Omniscient');
Math.max(1);
Math.max(1, 2);
Math.max(1, 2, 3);
```

Como puedes ver, la función acepta un número cualquiera de variables. Podemos hacer lo mismo gracias a la variable implícita `arguments`.

```
function f() {
  console.log('Número de argumentos pasados:', arguments.length);
  console.log('Argumentos:', arguments);
}
f();
f(1);
f('a', {});
f(function () {}, [], undefined);
```

Busca la información sobre `arguments` en la [MDN](#). ¡Te hará falta!

12. Decoradores

A parte de devolverse como parámetros, las funciones pueden ser devueltas desde otras funciones. Considera el siguiente ejemplo:

```
function newLog(label) {
  return function(value) {
    console.log(label + ':', value);
  }
}
```

Esta función crea funciones que llamarán a `console.log()` pero con una etiqueta delante. Podríamos crear métodos `log` por clase, cada uno con un prefijo y así distinguir unos logs de otros.

Sin embargo, advierte el siguiente comportamiento:

```
var log1 = newLog('Default');
var log2 = newLog('Ziltoid');

var p = { x: 1, y: 10 };
log1(p);
log2(p);
log1('Greetings', 'humans!');
```

¿Cuál es el problema? ¿Por qué no funciona el último ejemplo?

Para hacer que funcione, tendrías que llamar a `console.log()` con un número de parámetros que no sabemos a priori. Puedes usar `arguments`, no obstante:

```
function newLog(label) {
  return function() {
    // ¿Por qué tenemos que hacer esto?
    var args = Array.prototype.slice.call(arguments);
    args.splice(0, 0, label + ':');
    console.log.apply(console, args);
  }
}

var log1 = newLog('Default');
var log2 = newLog('Ziltoid');

var p = { x: 1, y: 10 };
log1(p);
log2(p);
log1('Greetings', 'humans!');
```

¿Podrías decir qué hace cada línea en la función `newLog` ?

13. Asincronía y closures

Carga el siguiente código:

```
function scheduleTasks(count) {
  for(var i = 1; i <= count; i++) {
    setTimeout(function () {
      console.log('Executing task', i);
    }, i * 1000);
  }
}
```

Y trata de predecir qué pasará al ejecutar el siguiente código:

```
scheduleTasks(5);
```

¿Hace lo que esperabas? Si no es así, ¿por qué? ¿cómo lo arreglarías? Pista: necesitas la función `.bind()`.

14. Eventos y métodos

Habrá veces en las que tendrás que llamar a un método de un objeto cuando ocurra algo. Por ejemplo, supón que el método avanzar de un supuesto objeto debe llamarse en un intervalo de tiempo. Pongamos cada segundo:

```
var obj = {
  x: 10,
  y: 2,
  advance: function () {
    this.y += 2;
    console.log('Ahora Y vale', this.y);
  }
};

var id = setInterval(obj.advance, 1 * 1000);
```

Este ejemplo falla porque en la última línea **no estamos llamando** a la función sino sólo pasándola como parámetro. La función `setInterval` no tiene idea del destinatario del mensaje y por tanto no puede llamar a la función como si fuera un método.

Podemos arreglarlo con `bind` pero antes para el intervalo con:

```
clearInterval(id);
```

Puedes solucionar el problema con:

```
var id = setInterval(obj.advance.bind(obj), 1 * 1000);
```

15. La función bind()

A estas alturas ya deberías saber cómo funciona `bind` o qué hace. Si aun no lo tienes claro, búscalo en la MDN.

La tarea es la siguiente: crea una función `bind` que simule el comportamiento del método de las funciones `.bind()`. Como se pide una función y no un método, el primer parámetro será la función. Así pues, en vez de usarse así:

```
function die(sides) {
  var result = Math.floor(Math.random() * sides) + 1;
  this.history.push(result);
  return result;
}
var obj = { history: [] };
var d20 = die.bind(obj, 20);
d20();
```

La usarrás de esta otra forma:

```
function die(sides) {  
  var result = Math.floor(Math.random() * sides) + 1;  
  this.history.push(result);  
  return result;  
}  
var obj = { history: [] };  
var d20 = bind(die, obj, 20); // fíjate en que ahora die es el primer parámetro  
d20();
```

Batalla RPG

Los objetivos principales de esta práctica son **acostumbrarte a leer código JavaScript** y que **practiques la implementación de modelos y algoritmos**.

El objetivo secundario es que practiques la metodología **TDD**, del inglés *Test Driven Development* o desarrollo dirigido por tests.

Instalación y tests

Descarga el [esqueleto inicial del proyecto](#) y descomprímelo. En una consola, ve a la raíz del directorio del proyecto (donde se encuentra el archivo `package.json`) y ejecuta:

```
npm install
```

Ahora puedes pasar los tests con:

```
npm test
```

La práctica no estará finalizada hasta que no pases los tests con 0 errores y 0 especificaciones pendientes:

```
$ npm test

> pvli2017-rpg-battle@1.0.0 test /Users/salva/workspace/pvli2017-rpg-battle
> node ./node_modules/gulp/bin/gulp.js test

[09:43:57] Using gulpfile ~/workspace/pvli2017-rpg-battle/gulpfile.js
[09:43:57] Starting 'lint'...
[09:43:57] Finished 'lint' after 71 ms
[09:43:57] Starting 'test'...
[09:43:57] Finished 'test' after 41 ms
.....
.....
88 specs, 0 failures
Finished in 0.4 seconds
```

Calidad del código

Este proyecto exige que tu código cumpla con ciertos estándares de calidad. El conjunto de restricciones responde a la [especificación recomendada por ESLint](#) con dos alteraciones:

- Se debe usar el estilo *came/Case* para identificadores.
- Se deben usar comillas simples para las cadenas.
- Se deben poner espacios entre operandos y operadores.
- No se permiten líneas de más de 100 caracteres.
- No se permiten funciones de más de 40 *statements*

En definitiva esas reglas están ahí para que los programas sean fáciles de leer. La mejor recomendación que puedes seguir es hacer lo que veas que ya está hecho. Cuando contribuyas a código ajeno, mira y copia.

Las dos últimas reglas están pensadas para que no hagas funciones muy largas, tanto en horizontal como en vertical. Ten en cuenta que lo que hemos limitado es el número de *statements*, no de líneas así que el siguiente ejemplo son realmente 4 *statements*:

```
function f() {  
    var x = 1;  
    var y = 2;  
    return x + g(y);  
  
    function g(n) {  
        return n * n;  
    }  
}
```

Al encargado de comprobar el estilo del código se lo llama *linter* y éste se pasa automáticamente junto con los tests. Un error se muestra así:

```
/Users/salva/workspace/pvli2017-rpg-battle/src/TurnList.js  
15:8  error  Identifier 'snake_case' is not in camel case  camelcase
```

Ahí se indica dónde está el error mediante la ruta del fichero, la línea y la columna en formato `fila:columna`.

Metodología y guía de la práctica

Antes de comenzar lee el documento [TDD.md](#) y la guía de la práctica [GUIDE.md](#). El primero presenta la metodología que seguirás para desarrollar la práctica y el segundo te muestra un orden de activación de tests sensato de cara a completar la práctica con éxito.

Descripción de la batalla

El objetivo de la práctica es crear una **biblioteca que modele el funcionamiento de una batalla RPG**.

Los combatientes en la batalla son personajes en distintos bandos que lucharán entre sí. Cuando sólo queden miembros de un bando este será declarado como vencedor.

Al comienzo de la batalla, los personajes son ordenados por su iniciativa, de mayor a menor y los turnos se suceden en este orden hasta que sólo quedan personajes de un bando. Si un personaje ha muerto, es decir, sus puntos de vida son 0, su turno se salta.

Cuando le llega el turno a un personaje este puede elegir entre realizar tres acciones: defender, atacar o lanzar un hechizo.

Si defiende, su defensa aumentará en un 10%. La defensa es importante porque afecta en el modo en que un personaje bloquea el efecto de las armas y los hechizos.

Si ataca, lo hará sobre un personaje objetivo que realizará una tirada de defensa. Si el objetivo supera esta tirada no recibirá ningún efecto pero si la falla, recibirá todo el daño del arma.

Si finalmente decide utilizar un hechizo, habrá de seleccionar el hechizo y luego un objetivo. Los hechizos se encuentran en un grimorio común a todo el bando y consumen puntos de maná. Si el personaje no tiene suficientes puntos de maná para seleccionar un hechizo, no puede utilizarlo.

Los bandos

Los bandos tienen un identificador, miembros y hechizos. Al conjunto de hechizos lo llamamos el grimorio del bando. Por ejemplo, la estructura:

```
var setup = {
  heroes: {
    members: [heroTank, heroWizard],
    grimoire: [fire, health]
  },
  monsters: {
    members: [fastEnemy],
    grimoire: [fire, health]
  }
};
```

Contiene dos bandos con identificadores `heroes` y `monsters`. El bando `heroes` tiene dos miembros. `heroTank` y `heroWizard`, mientras que el bando `monsters` sólo uno, `fastEnemy`. Los dos bandos tienen grimorios con los hechizos `fire` y `health`.

Los personajes

Los personajes tienen un **nombre**, un **bando opcional** y una serie de características que determinan su valor en la batalla. Estas características son iniciativa, defensa, puntos de vida y de maná, y máximos de vida y de maná.

La **iniciativa** determina el orden en el que los personajes aparecerán en la lista de turnos. Cuanto más alto, antes aparecen en esta lista.

La **defensa** establece la probabilidad (de 0 a 100) de que un [efecto](#) actúe sobre el personaje.

Los **puntos de maná** sirven para pagar los costes mágicos de los hechizos y los **puntos de vida** indican cuánto daño es capaz de resistir el personaje antes de ser morir (puntos de vida a 0). Ambas características están ligadas a unos valores máximos **puntos de maná máximos** y **puntos de vida máximos** respectivamente que no pueden sobrepasar.

La siguiente tabla resume las características numéricas y sus límites.

Característica	Mínimo	Máximo	Limitado por
Iniciativa	0	-	-
Defensa	0	100	-
Puntos de vida	0	-	Puntos de vida máximos
Puntos de maná	0	-	Puntos de maná máximos

Las acciones

Si durante su turno, un personaje elige **defender**, entrará en estado de defensa mejorada aumentando un 10% su defensa (redondeando arriba). El personaje puede defender tantas veces como quiera, aumentando su defensa en cada turno que defienda pero tan pronto otro personaje lo ataque, perderá su mejora.

Si elige **atacar**, habrá de elegir un personaje objetivo de entre todos los personajes vivos que queden en la batalla. El personaje causará el efecto de su arma al objetivo siguiendo las [reglas de aplicación de efectos](#).

Por último, un personaje puede **lanzar un hechizo**, en tal caso se debe elegir un hechizo que se pueda pagar del grimorio del bando y un objetivo. De nuevo se utilizarán las [reglas de aplicación de efectos](#) con el efecto del hechizo.

Efectos

Armas y hechizos contienen efectos. Un arma contendrá un efecto que **siempre reducirá los puntos de vida**, los hechizos pueden tener efectos variados. Exceptuando el nombre y el bando, **todas las características son susceptibles de verse alteradas por efectos**.

Reglas de efecto

Las reglas de efecto esperan un efecto cualquiera, un objetivo y un indicador de alianza que indica si el efecto lo aplica un aliado del objetivo o no.

El procedimiento es el siguiente:

1. Si el indicador de alianza indica que son aliados, continúa con el paso 2. Si no:
 - i. Genera un número al azar del 1 al 100.

- ii. Si el resultado se encuentra por debajo o es igual a la defensa del objetivo, suspende estos pasos. Si no, continúa con el paso 2.
2. Por cada característica afectada por el efecto
 - i. Suma el valor del efecto al valor de la característica del objetivo.
 - ii. Corrige el valor para que se encuentre entre 0 y el máximo para esa característica, si tiene.

Armas y hechizos

Armas y hechizos son elementos con efectos. La diferencia sustancial entre ellos es que los hechizos tienen un coste en puntos de maná para el lanzador del hechizo. Este coste se aplica tenga o no tenga éxito la defensa del objetivo.

El efecto de los hechizos puede ser variado así como el de las armas pero en el caso de las armas el efecto debe reducir los puntos de vida de alguna forma.

La API de batalla

La batalla tendrá los siguientes métodos:

- `setup` para establecer la configuración inicial como la que hemos mostrados antes.
- `start` para comenzar la batalla.
- `stop` para detener la batalla.

Además la batalla expone los siguientes atributos:

- `characters` para inspeccionar el estado de los personajes.
- `scrolls` para inspeccionar el estado de los personajes.
- `options` para controlar la batalla.

La batalla emitirá los siguientes eventos:

- `start` junto con los identificadores de los combatientes por bando, al comienzo de la batalla.
- `turn` junto con la información del turno, cada vez que le toque a un personaje.
- `info` con el resultado de una acción.
- `end` cuando sólo quede un bando.

En cualquier caso, el objeto de contexto será la `Batalla`.

La interfaz de control

Controlar el curso de la batalla consiste en decidir qué ocurre a cada turno. Para ello tendrás que suscribirte al evento `turn` y elegir entre las alternativas que se nos ofrecen en `options`:

```
var battle = new Battle(setup);
battle.on('turn', function (turn) {

    // Los héroes dañan al primer enemigo disponible.
    if (turn.party === 'heroes') {
        this.options.select('attack');
        var candidates = this.options.list();
        var monsters = candidates.filter(isMonster.bind(this));
        this.options.select(monsters[0]);
    }

    // Los monstruos sólo defienden.
    if (turn.party === 'monsters') {
        this.options.select('defend');
    }

    function isMonster(charId) {
        return this.characters.get(charId).party === 'monsters';
    }
});
battle.start();
```

Las opciones

A cada turno, inicialmente las opciones serán: `defend`, `attack` y `cast`.

Al elegir `defend` se aplican las reglas para la acción defender y pasamos al siguiente turno.

Si se elige `attack`, las siguientes opciones han de mostrar los identificadores de los personajes para seleccionar un objetivo. Cuando se elige el objetivo, se ejecuta la acción atacar y se pasa al siguiente turno.

Si elige `cast`, la siguientes opciones serán los hechizos disponibles (que pueden ser ninguno) y luego la lista de objetivos. De nuevo al elegir el objetivo, se efectúa la acción de lanzar hechizo y se pasa al siguiente turno.

Reglas de formación de identificadores

Los identificadores de los personajes no son los nombres puesto que estos podrían repetirse. Si un bando tuviera dos personajes murciélagos con el nombre `Bat`, sus identificadores sería `Bat` y `Bat 2`. Las reglas de formación de los identificadores son:

1. Recorre todos los bandos. Por cada bando:
 - i. Recorre todos los miembros. Por cada miembros:
 - i. Recupera el nombre del personaje y comprueba si está en el histograma de nombres.
 - ii. Si no está, añádelo con valor cero.
 - iii. Recupera el valor para nombre en el histograma de nombres.
 - iv. Si es 0, el identificador es el nombre.
 - v. Si es mayor que 0, el identificador es el nombre seguido de un espacio y el valor del histograma más uno.
 - vi. Incrementa el valor del histograma en 1.
 - vii. Asigna al personaje ese identificador.

Batalla RPG - Guía de la práctica

La guía de la práctica te sugiere un orden para completar con éxito la implementación de la funcionalidad de la práctica. Prepárate, eso sí, para **leer mucho JavaScript** y ten a mano Google, la MDN y StackOverflow.

1. El modelo de datos

En el juego existen muchos tipos de entidades, algunos relacionados y otros no.

La especificación de tales entidades puede encontrarse en `spec/entities.js` y la implementación en `src/entities.js`, `src/Character.js` y `src/items.js`.

Efectos

Quizá el tipo más sencillo sea el tipo `Effect` en `src/items.js` y especificado en `spec/entities.js`. Comienza por aquí.

Personajes

Continúa activando los tests relacionados con el tipo `Character` y desactiva los demás.

A continuación abre `src/character.js` e implementa las partes que faltan.

Elementos

Continúa con el módulo `src/items.js` activando paulatinamente las *suites* para los tipos `Item`, `Weapon` y `Scroll` que encontrarás en `spec/entities.js`.

Entidades por defecto

Tienes que crear algunos personajes, armas y pergaminos por defecto para que otras prácticas puedan usarlos. La *suite built-in entities* en `spec/entities.js` incluye todas las expectativas de estas entidades.

Ve al archivo `src/entities.js` y completa las que falten. Fíjate que las propiedades son *getters* para que cada acceso a las propiedades te devuelvan un nuevo personaje.

2. La lista de turnos

Esta es fácil. La especificación se encuentra en `spec/TurnList.js` y la implementación en `src/TurnList.js`. Tan sólo rellena los huecos. Es pura algoritmia. Quizá tengas que mirarte cómo funciona la función `Array.sort()` para no implementarte tu propia función de ordenamiento.

3. La vista del personaje

La vista del personaje es una representación de sólo lectura de las estadísticas del mismo. Su especificación está en `spec/charactersView.js` y su implementación en `src/charactersView.js`. Puedes continuar por ahí.

4. El grupo de opciones

El grupo de opciones representa las opciones que se pueden elegir en un momento dado. La especificación está en `spec/charactersView.js` y la implementación, casi completa, en `src/charactersView.js`. Fíjate cómo el tipo `options` extiende `EventEmitter`. Tu misión será implementar el método `.select()` para que al llamarlo se emita un evento acorde con la especificación.

5. La pila de opciones

Un RPG se compone normalmente de varios menús apilados. Por ejemplo, el menú de acciones da paso al menú de hechizos que da paso al menú de objetivos. En cualquier momento podemos regresar al menú anterior. La pila de opciones en `src/optionsStack.js` y especificada en `spec/optionsStack.js` refleja este comportamiento.

La API es la misma que el grupo de opciones pero los métodos realmente sólo se deben redirigir al último menú apilado. Fíjate en cómo se apilan y desapilan menús nuevos.

6. Utilidades

Tranqui. No tienes que hacer nada aquí, tan sólo ten en cuenta que tienes el módulo `src/utils.js` donde puedes colocar más utilidades si encuentras que andas repitiendo el mismo código en muchas partes. Te aconsejo que escribas algún test. Puedes inspirarte en los que ya hay en `spec/utils.js`.

7. La batalla

Has llegado al plato fuerte de la práctica. Hasta aquí era todo preparar los tipos en los que se apoya el tipo `Battle`. Ahora tendrás que implementar la máquina de estados que controla las acciones de batalla: defender, atacar y lanzar un hechizo.

Fíjate que los combatientes, sus armas y los hechizos **no son los que vienen por defecto en `src/entities.js`** sino los que se encuentran en `spec/samplelib.js`.

La especificación de la batalla está en `spec/Battle.js` y la implementación en `src/Battle.js`. Para abordar esta implementación con éxito es necesario que todos los tests hasta ahora pasen.

En esta parte de la práctica no hay recomendaciones sobre qué tests activar primero. Tendrás que experimentar.

Repasa bien el código, muchos de los ejercicios consisten en dar las implementaciones de **funciones auxiliares**. Es el caso de:

- `assignParty`
- `useUniqueNames`
- `isAlive`
- `getCommonParty`

Fíjate entonces en la función `_showAction` que hará que las acciones de batalla estén disponibles en el atributo `options`.

Ahora concéntrate en las acciones. La implementación de `_defend` está casi hecha. Sólo tendrás que completar las funciones `_improveDefense` y `_restoreDefense` para el cálculo de la defensa mejorada.

La acción `defend` rellena la estructura `this._action` con el nombre de la acción, los identificadores del personaje activo y del objetivo, el efecto y la nueva defensa. Todos menos la defensa son necesarios para poder llamar a la función `_executeAction` que ejecutará la acción e informará del resultado.

Durante el proceso de implementación de las acciones, tendrás que implementar también `_showTargets` y `_showScrolls` de manera similar, de acuerdo a la especificación y al [enunciado](#).

8. Calidad del código

Cuando termines y todos tus tests estén en verde habrás terminado la práctica.

Rotula la rama con un *tag* o cambia de rama antes de mejorar la calidad del código.

Lee ahora los errores de estilo que el comando de tests pueda proporcionar y pasa los tests cada vez que realices una modificación para asegurarte que no has roto nada.

Fin

¡Enhorabuena! Has completado la práctica.

TDD: Desarrollo dirigido por tests

Practicando TDD, iremos escribiendo código de forma que los tests pasen. Los tests vienen dados pero están desactivados. La [guía de la práctica](#) recomienda en qué orden activar los tests para completar la práctica poco a poco.

Tests y suites

En esta práctica usamos [Jasmine](#) como framework para tests. En Jasmine escribimos suites y tests. Las suites se pueden anidar y pueden llevar código de inicialización. En general, la API de Jasmine es muy clara y no necesita mayor explicación. De todas formas, aquí tienes un ejemplo:

```
describe('Los suites en Jasmine', function () {  
  describe('pueden anidarse', function () {  
    it('y encierran tests con expectativas', function () {  
      expect(2 + 2).toBe(4);  
    });  
  });  
});
```

Se llama test a un fragmento de código que pone a prueba una funcionalidad específica. El test puede pasar o fallar. En caso de fallo, la consola mostrará por qué ha fallado en la forma de una traza:

```
15.2) Expected 'b' to be 'c'.
      Error: Expected 'b' to be 'c'.
          at stack (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1640:17)
              at buildExpectationResult (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1610:14)
                  at Spec.expectationResultFactory (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:655:18)
                      at Spec.addExpectationResult (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:342:34)
                          at Expectation.addExpectationResult (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:599:21)
                              at Expectation.toBe (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1564:12)
                                  at Object.<anonymous> (/Users/salva/workspace/pvli2017-rpg-battle/spec/TurnList.js:39:36)
                                      at attemptSync (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1950:24)
                                          at QueueRunner.run (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1938:9)
                                              at QueueRunner.execute (/Users/salva/workspace/pvli2017-rpg-battle/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1923:10)
```

La traza contiene el fallo y dónde se ha producido en el conjunto de llamadas desde la más reciente hasta la más vieja. A veces los fallos son producto de implementaciones que no cumplen las expectativas, otras veces serán fallos en tiempo de ejecución y otras serán fallos de sintaxis.

Acostúmbrate a fallar y a encontrar en la traza el punto exacto del código que está bajo tu control para solucionarlo. Para ello busca las carpetas `spec` y `src` entre la traza. El primer número tras la ruta es la línea del fallo.

Activando y desactivando tests

Los tests y las suites pueden desactivarse añadiendo el prefijo `x`. Por ejemplo:

```

describe('Los suites en Jasmine', function () {
  describe('pueden anidarse', function () {
    it('y encierran tests con expectativas', function () {
      expect(2 + 2).toBe(4);
    });

   xit('este test está desactivado', function () {
    });

  });
  xdescribe('la suite y todos sus tests están desactivados.', function () {
  });
});

```

Los tests desactivados no comprueban las expectativas pero Jasmine te informa de que están desactivados.

El ciclo de desarrollo

Cuando estés desarrollando, es conveniente que pases los test a menudo por dos motivos:

- Comprobar que avanzas.
- Comprobar que no has roto nada.

Para ello puedes ejecutar el comando:

```
$ npm run-script watch
```

Esta tarea monitoriza los cambios en los archivos de las carpetas `spec` y `src` y cuando detecte un cambio, lanzara todos los tests.

A veces, el error es tan estrepitoso que rompe la monitorización. En tal caso tendrás que reintroducir el comando manualmente.

Cada vez que realices una modificación y los tests pasen, haz un commit nuevo.

Depurando tests asíncronos

Algunos tests son asíncronos y pueden producir *timeouts*. En general un *timeout* no es un resultado positivo. El problema de los *timeouts* es que pueden ralentizar toda la suite así que la recomendación en estos casos es afrontarlos uno a uno, desactivando el resto y activándolos poco a poco.

Reconocerás un test asíncrono porque lleva un parámetros `done` como en el ejemplo:

```
var EventEmitter = require('events').EventEmitter;

describe('EventEmitter', function () {

  it('emite eventos arbitrarios', function (done) {
    var ee = new EventEmitter();
    ee.on('turn', function(turn) {
      expect(turn.number).toBe(1);
      done();
    });
    ee.emit('turn', { number: 1 });
  });
});
```

Estrategia general para la depuración

Es muy recomendable que mantengas una rama estable donde todos los tests pasen y los que no estén desactivados. Cuando te embarques en la tarea de hacer que un test pase, crea una rama para esa tarea y cuando termines mézclala con la rama estable.

Cuando encuentres un error, intenta seguir los siguientes pasos:

1. Desactiva los tests asíncronos que estén tardando rápido. **Necesitas un ciclo de desarrollo rápido.**
2. **¡¡Lee el error!!.**
3. Busca en la traza el lugar donde se original el error:
 - i. Si es un fallo en una expectativa, localiza el punto de entrada en tu código.
 - ii. Deja trazas con `console.log()` inspeccionando el estado de tus objetos.
4. Salva y relanza los tests a menudo.

Navegador y DOM

¿Por qué?

Si quieras publicar juegos en la web, hay que saber cómo funciona un navegador, y cómo funciona la Web.

Los **navegadores** son el entorno donde se van a ejecutar tus juegos, y hay consideraciones técnicas a tener en cuenta, tanto a nivel de interfaz como de seguridad. Asimismo, los navegadores incorporan **herramientas de desarrollo** que nos serán muy útiles: un depurador, un profiler, un inspector del tráfico de red, etc.

También es necesario tener conocimientos sobre las tecnologías web, aun en el caso de utilizar un motor multiplataforma y exportar a HTML5:

- El juego siempre está contenido en una página web, y quizá interese personalizar su apariencia. Para ello, hace falta usar los lenguajes HTML (para el contenido) y CSS (la apariencia).
- Es útil para añadir ciertas cosas que quizás no incorpore dicho exportador, como tal vez un botón que ponga el juego a pantalla completa o bien precargar fuentes web.
- Siempre que se desarrolla un videojuego multiplataforma, es posible que aparezcan *bugs* sólo en una de estas plataformas. Para depurar un *bug* específico de la web, tendrás que usar las herramientas que proporcionan los navegadores, así como tener ciertos conocimientos de las API web que el juego utiliza (por ejemplo, la API de Gamepad o la de WebGL).
- Si utilizas un motor o framework de terceros, y necesitas arreglar un *bug* en dicho motor o implementar una *feature* que tu juego requiere, obviamente se ha de tener conocimientos de programación web.

Los navegadores

Un **navegador** es un programa que permite acceder a la Web. Hay que tener en cuenta que no todos los navegadores son iguales, y pese a que la Web está basada en **estándares abiertos**, no todos los navegadores implementan estos estándares de la misma manera.

Desde el punto de vista del desarrollador, los componentes más relevantes de un navegador son:

- El **motor de renderizado**, que se encarga de "pintar" una página web en la pantalla. Firefox utiliza Gecko, Safari utiliza Webkit, Chrome y Opera utilizan un *fork* de Webkit conocido como Blink, etc.
- La **máquina virtual de JavaScript**, que se encarga de ejecutar el código JavaScript de la página web. Firefox utiliza SpiderMonkey, mientras que Chrome y otros navegadores –así como Node– utilizan V8.

Esto da lugar a que **no todos los navegadores soporten las mismas características** y a que aparezcan *bugs* del motor de render o de la máquina virtual de JavaScript que son específicos a un navegador en concreto.

Para conocer qué características implementa cada navegador, podemos utilizar, entre otros:

- caniuse.com, rápida e intuitiva
- [MDN](https://developer.mozilla.org), tiene detalles concretos sobre la implementación y diferencias entre navegadores

Lenguajes en la Web

HTML

HTML (*HyperText Markup Language*) se utiliza para crear el **contenido** de una página web: párrafos, títulos, imágenes, vídeos, etc.

Es un lenguaje de marcado basado en etiquetas. Por ejemplo:

```
<h1>Esto es un título</h1>
```

CSS

CSS (*Cascading Style Sheets*) se utiliza para personalizar la **apariencia** de los elementos HTML: colores, fondos, bordes, su posición en la página, división en columnas, el tamaño, márgenes, etc.

Es un lenguaje declarativo basado en reglas. Por ejemplo:

```
h1 {  
    color: red; /* hace que los elementos h1 tengan texto rojo */  
}
```

JavaScript

JavaScript es un lenguaje de programación interpretado, y con él se implementan el **comportamiento y la lógica** de una página web –si hiciera falta.

Es un **lenguaje dinámico** orientado a prototipos, con características funcionales.

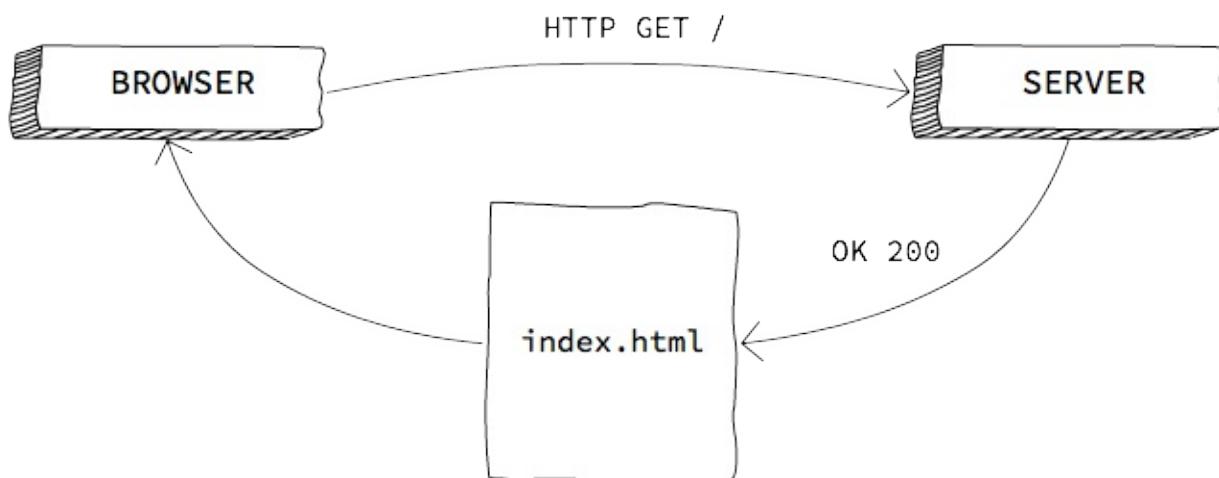
```
console.log("Hello, world!");
```

¿Qué pasa cuando se accede a una página desde un navegador?

Entender cómo funciona el tráfico de red a un nivel básico es necesario para comprender –y poder arreglar o mitigar– ciertas situaciones. Por ejemplo, por qué un juego tarda mucho en cargarse, por qué ciertos assets no se cargan, cómo hacer que la página no se quede en blanco mientras el juego se carga, etc.

Veremos un ejemplo de este flujo paso a paso.

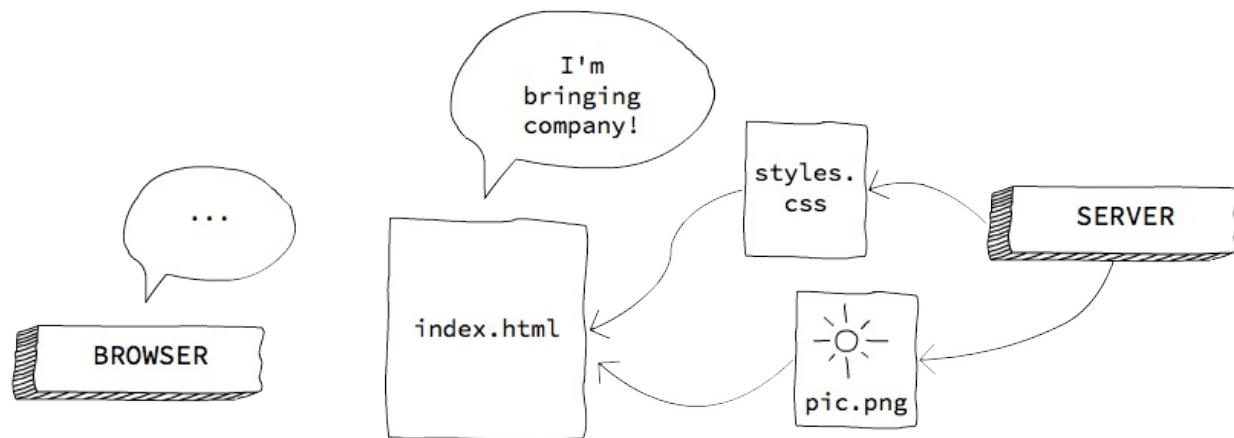
Paso 1. Petición HTTP GET a un servidor



El navegador hace una **petición HTTP GET** a una URL (que sirve para identificar dónde está cierto recurso en la Web), y si el recurso existe, el servidor lo retorna.

Hay que tener en cuenta que los navegadores disponen de una memoria caché, con lo que si ya tienen dicho recurso en memoria y no ha caducado, no realizan una petición al servidor y utilizan el recurso del que ya disponen.

Paso 2. Descarga de archivos



Los archivos HTML pueden hacer referencia a otros recursos... que el navegador deberá pedir al servidor. Si hemos pedido un archivo HTML, se irá **descargando y renderizando** sobre la marcha.

Cuando un archivo JS se acaba de descargar, su código se parsea y se **ejecuta**. Mientras se ejecuta, el navegador queda *bloqueado*.

Paso 3. Carga finalizada

Cuando todas las imágenes, scripts, CSS, etc. se han cargado, se dispara el **evento `load`** de `window`, que es un objeto global definido en el estándar de JavaScript que incorporan los navegadores.

Es muy común incluir el código que inicializa la ejecución del programa en el *handler* de ese evento.

```
window.onload = function () {  
    // Initialise the program  
};
```

Cómo incluir JavaScript en una página web

Una página web es, en esencia, un archivo HTML que puede incluir otros recursos, como las hojas de estilo CSS y también código JavaScript.

Estructura básica de un documento HTML

Aquí un ejemplo de un documento HTML con un poco de contenido –en este caso, un título (`<h1>`) y un párrafo (`<p>`).

```
<!doctype html>
<html>
  <!-- el head es metadata -->
  <head>
    <title>Cancamusa</title>
    <meta charset="utf-8">
  </head>
  <!-- el body es contenido -->
  <body>
    <h1>Monkey Island</h1>
    <p>Mira detrás de ti, ¡un mono de tres cabezas!</p>
  </body>
</html>
```

Como ves, HTML se basa en incluir unas etiquetas dentro de otras. Hay una etiqueta raíz, `<html>`, que contiene a su vez dos etiquetas, `<head>` y `<body>`. Estas dos etiquetas son *necesarias* en toda página web.

También se puede observar que en la primera línea hay una etiqueta especial:

```
<!doctype html>
```

Con esto le indicamos al navegador que efectivamente estamos utilizando un formato HTML5 o superior –antes había otros estándares, como XHTML.

`<head>` contiene **metadatos** relativos a la página, como el título (aparece en las pestañas del navegador, o en los resultados de búsqueda de Google y otros), la codificación de caracteres, etc. Desde aquí también podemos incluir otros archivos, como hojas de estilos CSS.

`<body>` posee el **contenido** de la página, lo que ve el usuario. En el ejemplo anterior hay un elemento de título y un párrafo.

¿Dónde va el código JavaScript?

Se puede incluir archivos –y código *inline*– de JavaScript tanto en el `<head>` como en el `<body>`, pero esto afecta a cómo se carga la página.

Por defecto, cuando se inicia la carga de un archivo JavaScript el navegador deja de renderizar el HTML hasta que dicho archivo se descargue por completo, se *parsee* y se ejecute.

Por tanto, si lo descargamos al principio (en el `<head>` o al principio de `<body>`) es posible que la página web aparezca en blanco o que retrasemos la carga de algunos assets que quizás sean más importantes.

Si por el contrario, incluimos el JavaScript al final de `<body>` la página web se renderizará por completo –con lo cual el usuario puede ver *algo*– pero el archivo tardará más en ejecutarse.

No hay una solución objetivamente óptima para todos los casos. Dependiendo de las características del videojuego (o de la página) interesarán un punto de carga diferente.

Nota: esto es ha sido una explicación muy simplificada. Puedes encontrar información más completa en [este artículo de Jake Archibald](#).

Scripts inline y externos

Podemos incluir código JavaScript inline en el HTML con la etiqueta `<script>`.

```
<script>
    console.log("Hello, world!");
</script>
```

También podemos tener el código JavaScript en un archivo `.js` separado. Se incluyen estos archivos con la etiqueta `<script>`:

```
<script src="js/game.js"></script>
```

Cómo se ejecuta el código JavaScript

Modelo asíncrono basado en eventos

En el navegador, JavaScript sigue un **modelo asíncrono** basado en eventos. Hay que tener en cuenta que, mientras el código JavaScript se está ejecutando, el navegador **bloquea todo** lo demás, incluida la UI (la interfaz gráfica de usuario).

El modelo asíncrono nos permite programar JavaScript de forma que los bloqueos sean imperceptibles al usuario. La idea es que el código JavaScript no se esté ejecutando siempre (y así la interfaz no estará bloqueada), sino que nos **subscribimos a ciertos eventos** y ejecutamos sólo código cuando estos eventos se producen.

Lógicamente, tenemos la responsabilidad de que el código no tarde mucho tiempo en ejecutarse, porque entonces sí que el usuario notaría que el navegador se ha quedado bloqueado. Los navegadores, además, suelen bloquear –u ofrecer al usuario esta posibilidad– los scripts de JavaScript que no han terminado al cabo de cierta cantidad de tiempo.

Algunos ejemplos de eventos a los que podemos subscribirnos son: la carga de la página, click en un botón o en un enlace, pasar el ratón por encima de un elemento, cuando se ha descargado cierto contenido, etc. También podemos lanzar nuestros propios eventos personalizados.

Un único hilo

Otro aspecto a considerar es que, *normalmente*, nuestro código JavaScript se ejecutará en **un solo hilo** (hay excepciones, como el uso de WebWorkers). Es por ello por lo que si hay dos suscripciones a un mismo evento, *no* se ejecutarán de forma simultánea, sino una detrás de otra. Y mientras un evento se ejecuta, el resto del código *espera*.

Veamos un ejemplo:

```
button.onclick = function (evt) {
    console.log("Click");
}

// ...

button.trigger('click');
// el botón no se desactiva hasta que el handler de "click"
// haya acabado de ejecutarse
button.disabled = true;
```

También puedes acceder a este [snippet de código online](#).

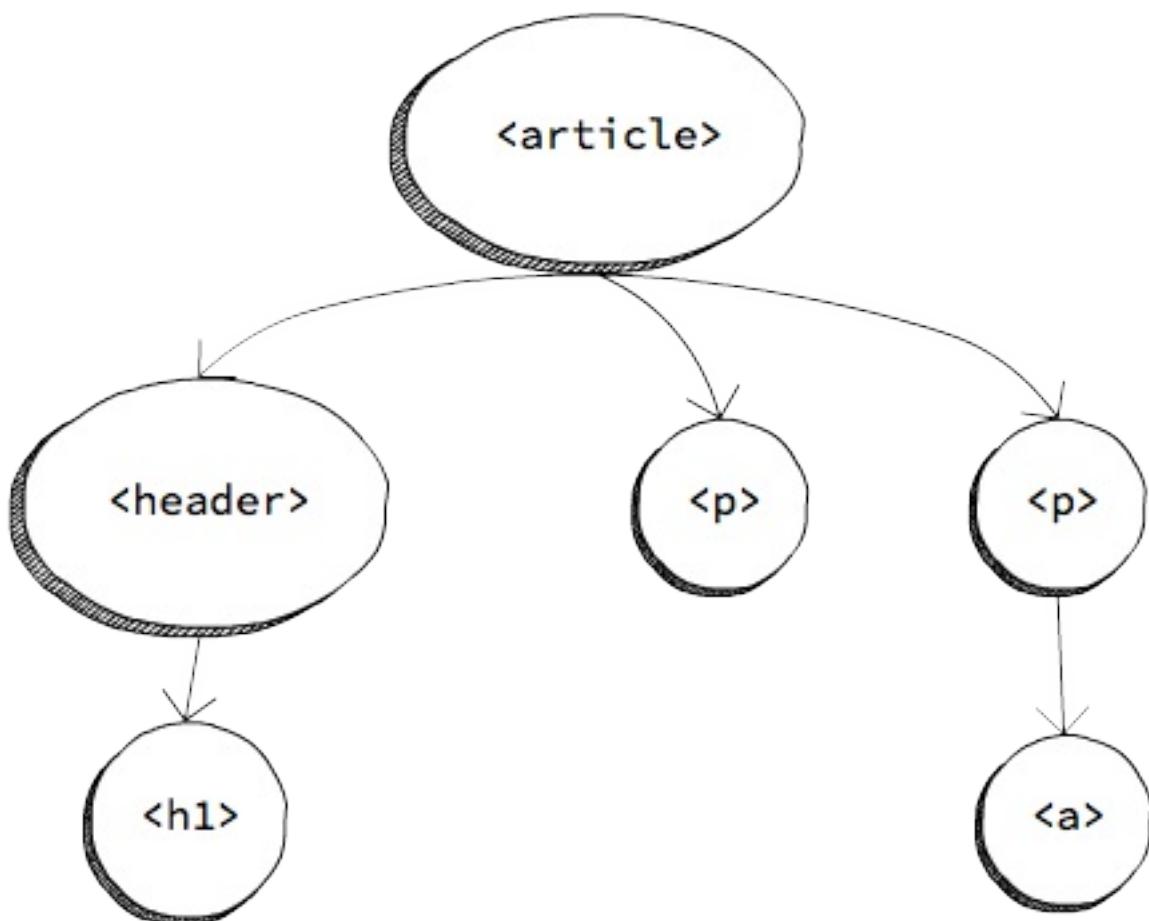
El DOM

Los documentos HTML presentan una **estructura de árbol**. Al incluir unas etiquetas dentro de otras, tenemos una relación jerárquica, ya que algunos elementos son "padres" de otros.

Por ejemplo, pongamos el siguiente fragmento de código HTML:

```
<article>
  <header>
    <h1>Un título molón</h1>
  </header>
  <p>Bla bla bla.</p>
  <p>
    Más bla, bla, bla y
    <a href="http://wikipedia.org">aquí un enlace</a>.
  </p>
</article>
```

Su representación en forma de árbol sería la siguiente:



El DOM (*Object Document Model*) es una **interfaz** que implementan los navegadores para que podamos **interactuar con dicho árbol** y con los elementos HTML que lo pueblan. Podemos tanto manipular los elementos HTML (cambiando su contenido, sus propiedades o bien llamando sus métodos propios), como manipular el árbol en sí, insertando, cambiando e eliminando elementos.

Acceder a elementos del DOM

Acceder a un elemento (o varios) del DOM es una de las operaciones más frecuentes que se hacen.

Por ID

Sólo selecciona un elemento (los ID deben ser únicos), en base a su atributo `id` de HTML:

```
<button id="show-fullscreen">Fullscreen</button>
```

```
var button = document.getElementById('show-fullscreen');
```

Acceder a elementos por selector CSS

Esta forma usa la sintaxis de los selectores CSS para localizar uno (o varios) elementos.

```
// selecciona el primer párrafo que encuentra
var paragraph = document.querySelector('p');
// selecciona el primer elemento con clase .warning
var label = document.querySelector('.warning');
// selecciona TODOS los párrafos
var allPars = document.querySelectorAll('p');
```

Puedes encontrar más información sobre selectores CSS para usar con `querySelector` en [la MDN](#).

Iterar sobre una lista de elementos

Hay que tener en cuenta que `querySelectorAll` no devuelve un array, sino una `NodeList`, que es un objeto diferente.

No podemos utilizar métodos de `Array` sobre una `NodeList`, pero tiene la propiedad `length` y el operador `[]`, así que podemos iterar sobre ella mediante un bucle:

```
var buttons = document.querySelectorAll('button');
for (var i = 0; i < buttons.length; i++) {
    buttons[i].style = "display: none"; // hide buttons
}
```

Aunque también podemos iterar con `Array.forEach` si lo utilizamos con `apply ...`

Navegar el árbol del DOM

Una vez que hemos accedido a un elemento, podemos navegar –recorrer– el árbol del DOM a partir de él.

- Se accede al **padre** de un elemento con la propiedad `parentNode`.

- Se accede a la lista de **hijos** de un elemento con `childNodes`.
- Se accede al **hermano** anterior o siguiente con `previousSibling` y `nextSibling`.

Con esto podemos recorrer todo el DOM en cualquier dirección.

Propiedades interesantes de elementos del DOM

innerHTML

Es el *interior* o contenido del elemento. Puede haber desde simplemente texto, hasta código HTML. Si usamos código HTML, estaríamos de hecho creando nuevos elementos HTML en el DOM al vuelo.

```
button.innerHTML = 'Aceptar';
// -> <button>Aceptar</button>
p.innerHTML = 'Párrafo con <b>negrita</b>';
// -> <p>Párrafo con <b>negrita</b></p>
```

style

`style` nos permite aplicar **estilos CSS inline**. Estos estilos tienen la máxima prioridad, así que son muy útiles para ocultar/mostrar elementos, por ejemplo.

```
var previousDisplay = button.style.display;
button.style="display:none"; // oculta cualquier elemento
button.style="display:inline-block"; // muestra el botón
```

Nota: `display:none` es universal, pero para mostrar un elemento se ha de elegir entre varios valores, los más comunes son `inline`, `inline-block` y `block`, pero hay otros.

classList

`classList` nos permite acceder a las **clases CSS** de un elemento, pudiendo añadir, quitar o alternar (*toggle*) clases. Esto es muy útil para cambiar el aspecto de la UI en función de las interacciones del usuario.

```
button.classList.add('loading');
button.classList.remove('loading');
button.classList.contains('loading'); // query
button.classList.toggle('loading'); // doesn't work on IE
```

Ejemplo: [snippet de código online](#).

Manipular el DOM

Manipular el DOM (esto es, insertar y eliminar elementos) nos permite alterar dinámicamente el contenido de una página web.

Insertar elementos

Ya hemos visto que se pueden insertar elementos HTML nuevos a través de la propiedad `innerHTML`, pero también los podemos crear desde cero, con `createElement`:

```
var button = document.createElement('button');
button.innerHTML = 'Start';
button.setAttribute('type', 'button');

// <button type="button">Start</button>
```

Es importante tener en cuenta que cuando creamos un elemento con `createElement` este se encuentra **huérfano** y no lo veremos renderizado en la página. Para que aparezca, hay que añadirlo al DOM como "familiar" de algún otro elemento –usando para ello `appendChild`, `insertBefore`, etc.

```
document.body.appendChild(button);
```

Ejemplo: [snippet de código online](#).

Eliminar elementos

Para eliminar un elemento del DOM, bien podemos reemplazarlo por otro –usando `replaceChild`, o bien podemos eliminarlo del todo.

Del mismo modo que se pueden crear elementos insertando una string con código HTML en la propiedad `innerHTML`, también se pueden eliminar todos los hijos –y contenido– de un elemento asignando una cadena vacía a `innerHTML`:

```
document.body.innerHTML = ''; // remove all the body content
```

También podemos eliminar un elemento del DOM con `remove`, y un hijo suyo con `removeChild`:

```
var button = document.querySelector('button');
button.remove();
```

Eventos

Los eventos del DOM son una parte crucial de la programación web, ya que es la manera que tenemos de programar el comportamiento de una página web sin bloquear la UI.

Los elementos del DOM disparan eventos a los que podemos suscribirnos, y ejecutar entonces el comportamiento que queremos asociado a dicha acción. Por ejemplo, un click en un botón, un cambio del texto de un `<input>`, cuando se selecciona una checkbox, etc.

No sólo los elementos HTML disparan eventos, sino que otros objetos globales, como `window`, también lo hacen. Por ejemplo: `load`, `resize`, etc.

Hay dos maneras de escuchar (o suscribirse a) eventos del DOM:

- Usando el método `Event.addEventListener`
- Usando los *on-event handlers* (p. ej: `onclick`, `onfocus`, etc.)

On-event handlers

Los *on-event handlers* eran originalmente la única manera de suscribirse a eventos, aunque hoy en día disponemos de la alternativa de `addEventListener`. El inconveniente de usar un *on-event handler* es que sólo podemos tener **un único handler** (es decir, el código que se ejecutará cuando el evento se dispara) por evento.

La manera de suscribirse y cancelar la suscripción con *handlers* es mediante una **asignación**:

```
// suscripción
button.onclick = function (evt) { /* ... */ };
// cancelar la suscripción
button.onclick = null;
```

Documentación [en la MDN](#).

Event listeners

Usando *event listeners* podemos **subscribirnos varias veces** al mismo evento, pudiendo tener varios comportamientos asociados a una misma acción.

Esta es la manera **recomendada y más segura**, especialmente si se usa código de terceros, ya que no podríamos eliminar de manera accidental un *handler* existente.

Para subscribirnos a un evento usamos el método `addEventListener`, y para cancelar la suscripción usamos `removeEventListener` (al que hay que pasarle la función de callback para que sepa qué suscripción en particular cancelar).

```
var sayHi = function () { /* */ };
// suscripción
button.addEventListener('click', sayHi);
// cancelar la suscripción
button.removeEventListener('click', sayHi);
```

Documentación [en la MDN](#).

Bubbling

El **bubbling** es la metáfora con la que explicamos cómo se comportan los eventos del DOM.

Cuando un elemento dispara un evento, **se propaga** hacia arriba en el árbol del DOM, como si fuera una burbuja. Es decir, que un evento disparado por un elemento en concreto, podrá ser visto también por su nodo padre y el resto de sus ancestros.

Por ejemplo, en el siguiente código, si el usuario hace click en el botón se dispararía el evento `click`, al que podríamos subscribirnos tanto desde el `<button>` como desde la `<section>` –como ocurre en este caso.

```
<section>
    <button>Click me</button>
</section>
```

```
var section = document.querySelector('section')
section.addEventListener('click', function () {
    console.log('Clicked...');

});
```

Puedes probar un [snippet de código online](#).

Interrumpir el *bubbling*

Hay veces que no nos interesa que se produzca esta propagación, o sólo queremos que se produzca de forma parcial.

Los *callbacks* de los eventos pueden recibir **un argumento**, que será un objeto de tipo `Event` y contiene información sobre el evento (como el elemento originario de que se disparase), así como métodos. Uno de estos métodos, `stopPropagation`, interrumpe el *bubbling*.

```
button.addEventListener('click', function (evt) {
    evt.stopPropagation();
});
```

Cancelar el evento

También puede ocurrir que necesitemos **cancelar el evento** para evitar las acciones por defecto asociadas a él (por ejemplo, el envío de un formulario, o el cambio de página al hacer click en un enlace).

Nótese que esto *no* es interrumpir el *bubbling*. Una vez disparado, el evento ya se ha producido y ya ha comenzado su propagación. Para interrumpir el *bubbling* debemos usar `Event.stopPropagation`, como ya hemos visto.

Los eventos se cancelan usando `Event.preventDefault`:

```
<a href="file.zip" download>Download zip</a>
```

```
var link = document.querySelector('a');
link.addEventListener('click', function (evt) {
    // the browser won't detect the link has been clicked
    evt.preventDefault();
});
```

Aviso: hay que tener mucho cuidado al hacer `preventDefault`, puesto que podemos empeorar la experiencia de usuario. Cancelar los links suele ser una mala idea, anular los clicks con el botón derecho para que el usuario no pueda guardar una imagen es una idea aún peor.

Usos legítimos de `preventDefault` podrían ser: validar un formulario en el cliente antes de enviarlo, anular ciertas acciones del teclado –por ejemplo, scroll hacia abajo con la barra espaciadora– en un videojuego que use esa tecla, etc.

Documentación en la MDN

La documentación de todo lo que se ha visto está en la Mozilla Developer Network (MDN): <http://developer.mozilla.org>

Un truco para acceder más rápidamente a la MDN desde un buscador, es añadir `mdn` a cualquier búsqueda:

The screenshot shows a search interface with a search bar containing "addEventListener mdn" and a blue search button. Below the search bar, there's a navigation menu with "All" selected, followed by "Videos", "Images", "Shopping", "News", "More ▾", and "Search tools". A progress bar indicates "About 25,300 results (0.71 seconds)". The main content area displays a search result for "EventTarget.addEventListener() - Web APIs | MDN - Mozilla Developer ...". The result includes a snippet of text: "Aug 30, 2016 - The EventTarget.addEventListener() method registers the specified listener on the EventTarget it's called on. The event target may be an ... You visited this page on 10/11/16.".

Canvas

Con la llegada de HTML5, se introdujeron varios elementos HTML y API's de JavaScript que hacen posible la programación de videojuegos con tecnologías y estándares web, entre ellos el elemento `<canvas>` y su API JavaScript, que nos permite pintar gráficos en pantalla, tanto 3D como 2D.

El elemento `<canvas>`

Esta etiqueta HTML crea un canvas con el ancho y el alto indicado. Sobre este canvas, podremos dibujar posteriormente usando la API de Canvas.

```
<canvas width="320" height="200"></canvas>
```

Las **dimensiones** del canvas especificadas en la etiqueta HTML con `width` y `height` no tienen por qué corresponderse con las dimensiones en pantalla, sino que se corresponden a una unidad virtual. Por defecto `1` unidad equivale a `1` pixel, pero con CSS podemos realizar un escalado. Por ejemplo, el siguiente código escalaría nuestro canvas anterior a un `200%` del tamaño original.

```
canvas {  
    width: 640px;  
    height: 480px;  
}
```

Podemos usar esto a nuestro favor y jugar con los escalados, o adaptar nuestro juego a distintos tamaños de pantalla. Por ejemplo, en este artículo se explica cómo usar un escalado para videojuegos retro con pixel art: [Retro, crisp pixel art in HTML5 games](#).

La API de Canvas

La API de Canvas nos permite realizar operaciones de dibujo 2D en un elemento `<canvas>`. Con ella podemos renderizar imágenes, manipular píxeles, dibujar primitivas gráficas, curvas, etc.

Dos recursos importantes son:

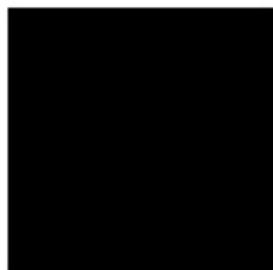
- [Documentación en la MDN](#).
- [Canvas Deep Dive](#): libro online conciso, pero bastante completo.

Contextos

Para realizar cualquier operación de pintado, necesitamos obtener un **contexto 2D** de un `<canvas>`. Para ello, utilizaremos el método `getContext` y le indicaremos que necesitamos un contexto 2D.

Ejemplo:

```
var ctx = document.querySelector('canvas').getContext('2d');
ctx.fillRect(10, 10, 100, 100);
```



Ver el [snippet de código](#) online.

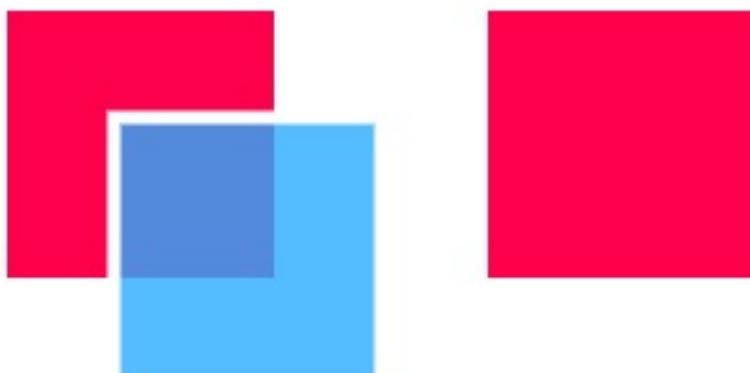
Colores, bordes, etc.

La manera de trabajar que tiene la API de Canvas es similar a la de un programa de dibujo. En el **contexto** indicamos los estilos (color de fondo, grosor de borde, etc.) que queremos que las "herramientas" (en este caso, las operaciones de dibujado) usen.

Hay que tener en cuenta que esto tiene "memoria", y que los estilos se conservan de una operación a la siguiente. Es decir, que si establecemos en el contexto que a partir de ahora usaremos el color rojo, *todas* las operaciones de dibujado posteriores usarán este color, no sólo la siguiente.

Ejemplo:

```
// red rectangles
ctx.fillStyle = '#FF004D';
ctx.fillRect(10, 10, 100, 100);
ctx.fillRect(190, 10, 100, 100);
// blue rect with white border
ctx.fillStyle = 'rgba(41, 173, 255, 0.8)';
ctx.fillRect(50, 50, 100, 100);
ctx.strokeStyle = '#fff';
ctx.lineWidth = 5;
ctx.strokeRect(50, 50, 100, 100);
```



Ver el [snippet de código](#) online.

Imágenes

Podemos renderizar imágenes en el canvas, pero para ello **deberán cargarse previamente**. El método más trivial es usar una imagen cargada con ``, pero podemos obtener imágenes de otras fuentes: la webcam del usuario, un `<video>`, otro

elemento `<canvas>` , etc.

Ejemplo:

```

<canvas width="300" height="300"></canvas>
```

```
window.onload = function () {
    var img = document.getElementById('kitten');
    var ctx = document.querySelector('canvas')
        .getContext('2d');

    ctx.drawImage(img, 0, 0);
}
```

Puedes ver la imagen creada con `` , y el elemento `<canvas>` a su lado con la misma imagen dibujada:



El ejemplo funciona porque el **evento** `load` **de** `window` se dispara cuando todas las imágenes incluidas en el documento HTML se han cargado, así que sabemos con seguridad que está ya disponible para ser pintada en el canvas.

Ver el [snippet de código](#) online.

Cargar imágenes al vuelo

Tener que crear una etiqueta `` en nuestro HTML por cada imagen que queramos cargar es laborioso. La mayoría de motores o librerías de videojuegos o gráficos crean objetos `Image` dinámicamente con JavaScript, que no se añaden al DOM –por lo que no son visibles.

Los pasos a seguir para cargar una imagen de esta manera serían:

1. Usamos el constructor `Image`.
2. Nos subscribimos al evento de `load` (para pintar la imagen cuando se haya cargado).
3. Establecemos el atributo `src` de la imagen para iniciar la carga.

```
window.onload = function () {
    var img = new Image();
    img.addEventListener('load', function () {
        ctx.drawImage(img, 0, 0);
    });
    img.src = 'https://placekitten.com/g/300/300';
}
```



Ver el [snippet de código](#) online.

Más sobre la carga de imágenes

Los elementos `` que tengan como estilo `display:none` son invisibles al usuario, pero se *siguen cargando* igualmente. Es habitual usar esto para ocultar las imágenes que se dibujen en el canvas. Como se ha comentado anteriormente, el evento `load` de `window` se dispara cuando –entre otras cosas– todas las imágenes del DOM se han cargado, sean visibles o no. Por ello, podemos cargar imágenes usando únicamente HTML y CSS, aunque es algo laborioso.

Para cargar imágenes desde JavaScript, creando nuevas instancias `Image`, se suelen utilizar [Promesas](#), o bien un contador para detectar cuándo se han cargado todas.

Otras operaciones de la API de Canvas

La API de Canvas es extensa y nos permite hacer muchas otras operaciones, entre ellas:

- Dibujar curvas y paths complejos
- Pintar gradientes
- *Clipping*
- Transformaciones
- Manipular píxeles
- Etc.

WebGL

WebGL es una API que nos permite operar con **gráficos 3D** en un elemento `<canvas>`. Su filosofía es completamente diferente a la API de Canvas, y es una API bastante más compleja –pero potente.

WebGL es una implementación en JavaScript de **OpenGL ES 2.0**, que es un estándar de la industria para el pintado de gráficos 3D.

Para poder usarla, debemos obtener un **contexto 3D** de un elemento `<canvas>`. Para ello, debemos especificar el parámetro `webgl -o experimental-webgl` en algunos navegadores– en la llamada a `getContext`.

Hay [documentación en la MDN](#) sobre esta API.

Gráficos 2D en WebGL

En el desarrollo de videojuegos, es habitual utilizar una API de gráficos 3D también para videojuegos 2D. La razón no es otra que rendimiento: ciertas operaciones realizadas con gráficos 3D son más eficientes, como el *Z-ordering*, rotaciones, transparencias, etc.

Es posible **simular un mundo 2D** usando una API de gráficos 3D, como WebGL. El "truco" es utilizar una proyección ortográfica, que no deformen los objetos con la perspectiva. Los gráficos 2D serían entonces polígonos con una textura dentro de este espacio 2D.

Muchas librerías y motores de juegos 2D en JavaScript –entre ellos Pixi y Phaser– ofrecen la posibilidad de utilizar WebGL como API de gráficos en lugar de la API de Canvas.

Recursos

Para aprender más sobre WebGL recomendamos los siguientes recursos:

- [WebGL en la MDN](#): documentación, guías, tutoriales, etc.
- [Introduction to WebGL parte 1](#) y [parte 2](#)
- [WebGL fundamentals](#): tutoriales paso a paso

Librerías gráficas

Hay dos librerías extremadamente populares para el manejo de gráficos con JavaScript. Son únicamente librerías de gráficos, así que no implementan otras funcionalidades necesarias para un videojuego. Sin embargo, constituyen un buen punto de partida bien para desarrollar un motor propio, bien para cuando sólo se necesite pintar gráficos en un proyecto en concreto.

Muchos motores las utilizan como capa gráfica, así que conviene conocerlas si se quiere modificar un motor o acceder a *features* de estas librerías no expuestas por el motor.

Gráficos 2D: Pixi.js

- www.pixijs.com
- Funciona por defecto con WebGL, pero tiene fallback a Canvas 2D.
- Phaser utiliza Pixi para renderizar gráficos

Gráficos 3D: THREE.js

- www.threejs.org
- Es la librería de referencia para trabajar con WebGL, y simplifica mucho el uso de esta API.
- Facilita renderizar gráficos para ser usados con WebVR (API Web para realidad virtual)
- Hay infinidad de tutoriales, libros, etc. disponibles.

Animaciones

Hasta ahora hemos visto cómo renderizar gráficos estáticos en un canvas, pero en un videojuego las imágenes están en movimiento.

En el desarrollo de videojuegos, idealmente vamos a intentar renderizar las imágenes en el canvas **60 veces por segundo** (60 FPS o *frames per second*). Para este cometido **no nos sirven** `setTimeout` ni `setInterval`, ya que no son precisas y no tenemos garantizada su ejecución (puedes ver las razones [en la documentación](#)).

La manera adecuada para renderizar animaciones en un canvas es usando `requestAnimationFrame`.

requestAnimationFrame

Esta función acepta un *callback* que se llamará automáticamente la siguiente vez que el navegador **pueda pintar** en pantalla. En este *callback* incluiremos tanto las operaciones de dibujo que queramos realizar, como una nueva llamada a `requestAnimationFrame`, para establecer así un **bucle** continuo.

Ejemplo:

```
function render() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.fillRect(x, 25, 50, 50);
    x = (x + 1) % canvas.width;

    requestAnimationFrame(render);
}

requestAnimationFrame(render);
```

Puedes ver una comparación del mismo código de dibujado ejecutándose con `requestAnimationFrame` y con `setInterval`. Podrás observar que la animación con `requestAnimationFrame` es más fluida, especialmente si cambias de pestaña en el navegador, lo pasas a segundo plano, etc.

- [Snippet de código](#) con `requestAnimationFrame`
- [Snippet de código](#) con `setInterval`

Tiempo delta

El **tiempo delta** es como se llama en desarrollo de videojuegos al tiempo que ha transcurrido entre el *frame* actual y el anterior. Este tiempo es necesario que sea preciso ya que mucha lógica del juego depende de él: animaciones, físicas, etc.

Mientras que en programación web se usa normalmente `Date` para manejar fechas, crear instancias de `Date` no sólo no es eficiente, sino que estos objetos no tienen la resolución suficiente para un videojuego, que necesita decimales de milisegundos.

Existe una alternativa, y es usar objetos de *timestamp*, `DOMHighResTimeStamp`, como los que devuelve el uso de la interfaz de `Performance`. Estos *timestamps* tienen un margen de error de únicamente 5 microsegundos. Además, para nuestra conveniencia, `requestAnimationFrame` llama a nuestro **callback con un timestamp**.

En la práctica, este parámetro nos permite calcular el tiempo delta de forma precisa. El *timestamp* que obtenemos contiene el número de milisegundos transcurrido desde la primera llamada a `requestAnimationFrame`. Si vamos almacenando cuál era el valor del *timestamp* en el *frame* anterior, podemos calcular el tiempo delta:

Ejemplo:

```
const SPEED = 60; // pixels per second
var oldTimestamp = 0;

function render(timestamp) {
  var delta = (timestamp - oldTimestamp) / 1000.0;
  oldTimestamp = timestamp;

  // ...
  x = (x + SPEED * delta) % canvas.width;
  requestAnimationFrame(render);
}
```

Puedes probar el [snippet de código online](#).

Algunas **consideraciones** a tener en cuenta sobre el tiempo delta:

- Siempre se ha de poner una **cota superior** al valor del tiempo delta (p.ej: 250ms) para evitar *glitches* en la lógica del juego. Un "salto" grande en el tiempo delta puede causar fallos en las colisiones, funcionamiento anormal en el motor de físicas, etc.
- A veces es recomendable saltarse el *update* de nuestro juego un *frame* (por ejemplo, [como hace Phaser](#)).

Cliente web para batallas RPG

A continuación se expone la práctica propuesta para esta unidad. Tras leer este enunciado, se recomienda *encarecidamente* consultar la [guía de la práctica](#) para su realización.

Enunciado

La práctica consiste en implementar un cliente visual (en este caso, [una página web](#)) para el juego de batallas de la práctica de la unidad anterior.

RPG Battle

Heroes	Monsters
Tank (HP: 80/80, MP: 30/30)	Slime (HP: 40/40, MP: 50/50)
Wizz (HP: 40/40, MP: 100/100)	► Bat (HP: 5/5, MP: 0/0)
	Skeleton (HP: 100/100, MP: 0/0)
	Bat 2 (HP: 5/5, MP: 0/0)

Fight!

Attack
 Defend
 Cast

Select action

Como punto de partida, se ha provisto de un [href="start-here.zip" target="_blank">esqueleto de proyecto](#) con los siguientes archivos:

- `index.html` : código HTML de partida
- `styles.css` : hoja de estilo
- `js/main.js` : el archivo de partida JavaScript
- `js/rpg.js` : un archivo con el código de la librería de batallas de la práctica.

También puedes usar tu propio código (consulta la sección de *Adaptación del código de la práctica anterior*).

En este código inicial se incluye ya implementado lo siguiente:

- Carga desde el archivo HTML de los recursos JavaScript y CSS.
- Esqueleto HTML con una interfaz ya hecha. Puedes modificar este HTML para añadir más cosas, o cambiar elementos de la UI que no te convengan, pero no es obligatorio.
- Creación de una instancia de `Battle`, así como el setup de las *parties* y la suscripción a los eventos más relevantes. La información de dichos eventos se imprime por consola (lo cual puedes eliminar/modificar a tu gusto).

Hay que implementar las siguientes *features*:

- Mostrar los personajes de ambas *parties*, con sus ID's, puntos de vida y de maná.
- Marcar qué personaje está seleccionado, cambiando su estilo o añadiendo un carácter especial (p.ej: `*`).
- Marcar qué personajes están muertos, cambiando su estilo o añadiendo un carácter especial (p.ej: `+`).
- Implementar el menú de batalla con sus siguientes estados: selección de acción, selección de objetivo y selección de hechizo.
- Mostrar información de qué ha pasado cada turno (p.ej `Bat attacked Wizz, but missed`).
- Mostrar un mensaje al final de la batalla indicando cuál es el bando ganador.

Para implementar estas *features* básicas, es recomendable seguir el procedimiento marcado por la [guía de la práctica](#).

Otras características opcionales que se podrían implementar, serían:

- En el menú de selección de objetivo, mostrar en un color diferente los personajes de cada *party*.
- Al terminar la batalla, mostrar un botón o enlace para empezar una nueva (esto se puede hacer simplemente recargando la página).
- Crear la composición de una o ambas *parties* de manera aleatoria.

Adaptación del código de la práctica anterior

En la versión actual de JavaScript no hay ningún mecanismo para gestionar módulos (por ejemplo, usando la función `require` como en Node). Es por esto que no podemos utilizar ni `require` ni `module.exports`.

Además, ciertas partes que forman parte de la librería estándar de Node, como el módulo `events` para implementar eventos, no forman parte del estándar JavaScript.

Hay una herramienta, [Browserify](#) que nos permite transformar módulos de Node –con sus dependencias– en código que funciona en el browser. También incluye **polyfills**.

Instrucciones

Opción A: usar el código propio

Si has acabado la práctica anterior, puedes utilizar ese código en esta. Sigue los pasos que hay a continuación para adaptar ese código de Node a código que puedas ejecutar en el navegador.

- Como vamos a necesitar dos módulos, `Battle` y `entities` (junto a sus dependencias), tenemos que crear un archivo "raíz" con esos dos. Crea en la raíz del directorio de la práctica anterior un archivo `export.js` con el siguiente contenido:

```
module.exports = {
  "Battle": require('./src/Battle.js'),
  "entities": require('./src/entities.js')
};
```

- De nuevo en la raíz del directorio de la práctica anterior, instala Browserify con npm:

```
npm install --save-dev browserify
```

- Comprueba que el archivo `package.json` se ha modificado y que ahora aparece Browserify listado dentro de `devDependencies`. Por ejemplo:

```
"devDependencies": {
  "browserify": "^13.1.0"
}
```

4. Edita `package.json` para añadir un comando de script más, que ejecutará Browserify:

```
"scripts": {  
  "bundle": "browserify export.js --standalone RPG > rpg.js"  
}
```

5. Ejecuta dicho comando, que generará un archivo `rpg.js` en el raíz de ese directorio.

```
npm run bundle
```

6. Ahora puedes copiar `rpg.js` al directorio `js` de la práctica 2. Cuando se cargue el archivo con una etiqueta `<script>` en el navegador, habrá un objeto global `RPG`, con dos propiedades: `entities` y `Battle`.

Opción B: usar una implementación de terceros

Puedes emplear el archivo `rpg.js` –incluido ya en el código fuente de partida–, que contiene una implementación de la práctica anterior.

Auto-reload del navegador

Si quieres que el navegador **automáticamente recargue** la página cuando modifiques un archivo, lo puedes conseguir fácilmente con [Browsersync](#). No es obligatorio, pero quizás te resulte más cómodo programar así.

Puedes instalar Browsersync de manera global (para así utilizarlo desde cualquier directorio) vía npm con el flag `-g`:

```
npm install -g browser-sync
```

Una vez instalado, puedes lanzarlo en el directorio raíz de la práctica. El siguiente comando ejecuta browsersync, lanza un servidor local y activa la auto-recarga del navegador cuando haya algún cambio en los ficheros HTML, CSS o JavaScript:

```
browser-sync start --server --files=".html,.js/*.js,.css"
```

Para más información, consulta la [documentación de Browsersync](#).

Guía

Nota: lee primero el enunciado de la práctica antes de leer este documento.

Esta es una guía paso a paso con sugerencias para afrontar la práctica. Es recomendable **leer este documento entero** –especialmente el apartado de *Consideraciones*– antes de ponerse a realizar el primer paso.

También es recomendable tener fresca en la cabeza la API de la práctica anterior de las batallas RPG con Node: repasa el enunciado de la práctica o su código fuente.

Durante el desarrollo de esta práctica, si te atascas sobre qué llamadas a la API hay que hacer, siempre puedes consultar el archivo `index.js` de la práctica anterior – donde hay programado un cliente de dicha API.

1. Mostrar los personajes

En el código HTML de base, hay dos listas `<ul class="character-list">` que hay que rellenar con elementos ``, uno por personaje. Debemos indicar el nombre del personaje junto con sus atributos de vida y maná. Más adelante nos será útil poder referirnos a ese personaje, así que es conveniente almacenar en un atributo `data` su ID de personaje.

```
<ul class="character-list">
    <!-- ... -->
    <li data-chara-id="bat_2">
        bat (HP: <strong>5</strong>/5, MP: <strong>0</strong>/0)
    </li>
</ul>
```

Aunque las listas `` están creadas en el archivo HTML, falta llenar su contenido, lo cual se ha de hacer dinámicamente con JavaScript. Con `Battle.prototype.characters.allFrom` se puede acceder a las vistas (`charactersView`) de los personajes de una party.

Como habrá que refrescar el contenido de estas listas, es recomendable implementar una función que haga esto, para poder llamarla más adelante desde otros sitios.

Por ahora, se puede comenzar con mostrar los personajes al inicio de cada turno. Para ello, hay que suscribirse al evento `turn` de `Battle`.

RPG Battle

Heroes

Tank (HP: 80/80, MP: 30/30)
Wizz (HP: 40/40, MP: 100/100)

Monsters

Slime (HP: 40/40, MP: 50/50)
Bat (HP: 5/5, MP: 0/0)
Skeleton (HP: 100/100, MP: 0/0)
Bat (HP: 5/5, MP: 0/0)

Documentación relacionada

- [Element.innerHTML](#) en la MDN.
- [Document.querySelector](#) en la MDN

2. Mostrar el personaje seleccionado

El CSS está preparado para que un elemento con la clase `active` aparezca resaltado, indicando que es el turno de dicho personaje.

En el callback evento `turn` de `Battle` se nos pasa la información de cada turno. Uno de los datos del evento es la ID del personaje activo, con lo que podemos usar un `querySelector` para seleccionar el `` de dicho personaje y añadir la clase `active`.

Monsters

Slime (HP: 40/40, MP: 50/50)

► Bat (HP: 5/5, MP: 0/0)

Skeleton (HP: 100/100, MP: 0/0)

Bat (HP: 5/5, MP: 0/0)

Documentación relacionada

- [Element.classList](#) en la MDN
- [Using data attributes](#) (guía en la MDN)

3. Mostrar el menú de acciones de batalla

Dentro de la sección `<section class="battle-menu">` hay tres formularios, que aparecen ocultos gracias a un estilo `display: none` aplicado *inline*. Cada formulario representa una "fase" del menú de batalla, y presentará una lista de opciones al jugador.

Se puede obtener una lista de las opciones disponibles en un momento dado con el método `list` de `Battle.prototype.options`. Con dicha lista, se pueden generar una serie de *radio buttons* para que el jugador pueda elegir la opción deseada.

Esta lista de *radio buttons* tendrá el siguiente aspecto –pero se ha de generar dinámicamente con JavaScript:

```
<ul class="choices">
  <li><label><input type="radio" name="option" value="attack"> attack</label></li>
  <li><label><input type="radio" name="option" value="defend"> defend</label></li>
  <li><label><input type="radio" name="option" value="cast"> cast</label></li>
</ul>
```

Hemos de generar esta lista dinámicamente dentro de las acciones a tomar cuando se dispare el evento `turn` de `Battle`. En este caso, la lista a generar es de **las acciones** que puede tomar un personaje, lo que se corresponde con el formulario con su atributo `name` a `select-action`.

- Attack**
- Defend**
- Cast**

Select action

4. Seleccionar una acción

Los *radio buttons* tienen un funcionamiento peculiar, puesto que comparten el atributo `name`. Esta es la manera que tiene el navegador de "agruparlos", de forma que sólo pueda estar activo un único *radio button* en un momento dado.

Se puede acceder fácilmente al valor (`value`) de un **grupo de radio buttons** a través del formulario al que pertenecen. Por ejemplo, para el menú de batalla hemos establecido un `name` para el grupo de "option". Suponiendo que su formulario está almacenado en la variable `form`, podríamos tener:

```
var action = form.elements['option'].value;
battle.options.select(action);
```

Obviamente hay que realizar esto cuando el jugador haya pulsado el botón de *Select action*. Para ello, hay que suscribirse al evento `submit` del formulario. Es imprescindible que además anulemos dicho evento con `preventDefault` para evitar que el navegador recargue la página.

Debemos además **validar** el formulario, para asegurarnos de que el jugador *ha seleccionado* una opción. HTML5 nos permite hacer validaciones sencillas con JavaScript en el lado del cliente. Para requerir un campo, debemos añadir el atributo `required` a dicho elemento. En el caso de un grupo de *radio buttons*, basta con añadir `required` a uno cualquiera:

```
<input type="radio" name="option" value="attack" required>
```

Para comprobar que este paso está hecho correctamente, puedes suscribirte al evento `info` de `Battle` e imprimir los datos de dicho evento. Con lo que hay hecho ahora debería funcionar la acción de *Defend*, puesto que no requiere ningún paso más.

```
INFO Object { action: "defend", activeCharacterId: "bat", targetId: "bat", newDefense: 88, success: true }
```

Documentación relacionada

- [HTML Forms Guide](#), una lista de artículos y tutoriales sobre formularios HTML5.
- [`Event.preventDefault`](#) en la MDN
- [`HTMLFormControlsCollection`](#) en la MDN (para información sobre `HTMLFormElement.elements`)

5. Seleccionar un objetivo

Se hace de forma similar a seleccionar una acción: se ha de generar la lista de opciones dinámicamente con JavaScript, así como interceptar el evento `submit` de este formulario y llamar a `battle.options.select` con el objetivo seleccionado por el jugador.

Un añadido que tiene este formulario es un enlace que nos permite **cancelar la acción actual**. Para que funcione, hemos de interceptar el evento `click` de dicho enlace, usar `preventDefault` para que el navegador no haga nada, y llamar a `battle.options.cancel`.

Por supuesto, hay que controlar qué menú está visible cada momento. Esto lo podemos hacer cambiando su estilo CSS inline, a través de la propiedad `display`:

```
actionForm.style.display = 'none'; // oculta el formulario de acciones
```

```
targetForm.style.display = 'block'; // muestra el formulario de objetivos
```

Para comprobar que funciona, prueba a elegir atacar un objetivo en los menús y comprueba el mensaje que emite el evento `info` de `Battle` en la consola:

```
INFO Object { action: "attack", activeCharacterId: "bat", effect: Object, targetId: "Tank", success: false }
```

Como re-renderizamos los personajes de las *parties* en cada turno (cada vez que `Battle` emite `turn`), si el ataque ha tenido éxito, deberíamos ver cómo han disminuido sus puntos de vida.

Documentación relacionada

- Propiedad CSS `display` en la MDN

6. Seleccionar un hechizo

Esto es muy similar a seleccionar una acción o un objetivo. Hay que generar, de nuevo, la lista de opciones (en este caso, hechizos disponibles) dinámicamente, y ocultar / enseñar el menú que corresponda según el flujo.

Seleccionar un hechizo ocurre tras haber seleccionado la opción *Cast* en el menú de acciones, y una vez seleccionado el hechizo debemos mostrar el formulario de seleccionar un objetivo.

La particularidad de este menú es que **puede ser que no haya ninguna opción disponible**. En este caso, debemos desactivar el botón del formulario si la lista de opciones está vacía (y activarlo en caso contrario). Para ello, hay que usar la propiedad `disabled` del botón.



Select spell or Cancel

7. Panel de información

Hay que mostrar información al usuario sobre el resultado de una acción concreta: si el ataque tuvo éxito o no, cuánto daño causó, etc.



Wizz casted fireball on Skeleton and caused -25 hp.

En el archivo HTML hay un párrafo con ID `battle-info`, cuyo contenido puedes modificar para mostrar estos mensajes de información de batalla.

Esta información de batalla la podemos obtener suscribiéndonos al evento `info` de `Battle`, que nos proporcionará los datos de qué personaje actuó, contra quién, si tuvo éxito, etc.

Para facilitar la tarea, se ha incluido una función llamada `prettyEffect`, que devuelve una string "bonita" (en lugar de `[Object object]`) con los efectos aplicados en el ataque.



```
var effectsTxt = prettyEffect(effect || {});  
// ej: -> '-5 hp, +5 mp'
```

8. Marcar personajes como muertos

Ahora que los personajes pueden atacar, podemos mostrar cuándo uno ha muerto. La hoja de estilos CSS incluye una clase `dead` que, aplicada a un elemento, lo muestra como "muerto" (en este caso, tachado).

Monsters

~~Slime (HP: 40/40, MP: 50/50)~~

~~Bat (HP: 0/5, MP: 0/0)~~

~~Skeleton (HP: 100/100, MP: 0/0)~~

► **Bat (HP: 5/5, MP: 0/0)**

Para ello, se ha de modificar el código que se ha programado en el paso 1 para mostrar los personajes, de forma que cada elemento de la lista (``) tenga la clase `dead` si el personaje está muerto (los puntos de vida están a cero).

9. Final del juego

Hay que mostrar un mensaje en el panel de información que indique que el juego ha acabado y quién ha ganado. Además, hay que volver a mostrar los personajes para que se muestre el resultado final de la batalla (con una de las *parties* con todos los personajes muertos).

El juego se acaba cuando `Battle` emite el evento `end`, por lo que debemos implementar dicho código en un callback de este evento.

Heroes

Tank (HP: 80/80, MP: 30/30)

Wizz (HP: 40/40, MP: 10/100)

Monsters

~~Slime (HP: 0/40, MP: 35/50)~~

~~Bat (HP: 0/5, MP: 0/0)~~

~~Skeleton (HP: 0/100, MP: 0/0)~~

~~Bat 2 (HP: 0/5, MP: 0/0)~~

Battle is over! Winners were: Heroes

Consideraciones

Template strings

Las *template strings* (o *template literals*) son una característica de ES6, pero que [ya está implementada](#) en la mayoría de navegadores modernos. Pueden resultarte útiles en esta práctica, ya que soportan **interpolación de expresiones** y declaraciones **multilinea**.

En su manera más básica, se definen igual que una string normal, pero usando backticks ```:

```
`Hello, world!`
```

Se pueden tener strings multilíneas sin necesidad de romper la cadena ni usar el operador de concatenación:

```
`Hello,  
world!`
```

Para interpolar expresiones, debemos poner la expresión a interpolar entre los caracteres `{}$` :

```
var name = 'Darth Vader';  
`Hello, ${name}`;
```

No sólo podemos utilizar variables, sino otro tipo de expresiones:

```
`Hello, ${name.toUpperCase()}, here's a calculation: ${2 + 2}`
```

Recordemos también que el operador ternario es también una expresión, y nos permite interpolar condiciones sencillas:

```
`Hello, you are ${life > 0 ? 'alive' : 'dead'}`;
```

Un ejemplo práctico que muestra la diferencia entre usar template strings y strings normales para generar código HTML a usar dentro de la propiedad `innerHTML` :

```
var list = document.querySelector('ul.shopping-list');
var data = {name: 'Banana', amount: 3, price: 0.5};

// template strings
list.innerHTML += `<li class="${data.amount > 0 ? 'bought' : ''}">
  ${data.name}, ${data.amount * data.price}€
</li>`;

// regular strings
list.innerHTML += '<li class="' + (data.amount > 0 ? 'bought' : '') + '">' +
  data.name + ', ' + data.amount * data.price + '€' +
  '</li>';
```

Documentación: [Template strings](#) en la MDN.

Atributos data

Las ID's de los personajes repetidos tienen un espacio en blanco (p.ej: `bat 2`). Para poder usar esto en un `querySelector`, se deben utilizar comillas:

```
document.querySelector('[data-chara-id="bat 2"]');
```

Para acceder a ese atributo data usando la propiedad `dataset` hay que tener en cuenta que los guiones se transforman en [camel case](#). Así, si se usa como atributo data `data-chara-id`, se accedería a él de la siguiente manera:

```
var el = document.querySelector([data-chara-id="bat 2"]);
console.log(el.dataset.charaId); // imprimiría bat 2
```

Documentación: [Using data attributes](#) en la MDN.