

Desarrollo en el cliente

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez Natalia Madrueño

ruben.rodriguez@urjc.es

URJC

natalia.madrueno@urjc.es

URJC

2025-09-09



Tabla de contenidos

- Orientación a Objetos

Orientación a Objetos

Prototipos vs Clases

JavaScript usa herencia prototípica, no clases tradicionales

- Cualquier objeto puede ser prototipo de otros objetos
- Los objetos heredan directamente de otros objetos
- Cadena de herencia flexible y dinámica
- ES2015+ añadió sintaxis de clases (azúcar sintáctico sobre prototipos)

 **Nota**

A diferencia de Java o C++, JavaScript no tiene clases “reales” en su núcleo

Objeto Literal Simple

La forma más directa de crear objetos

```
1 const enemigo = {  
2     vida: 100,  
3     damage: 15,  
4     atacar() {  
5         return `Enemigo ataca causando ${this.damage} puntos`;  
6     }  
7 };
```

- Definimos propiedades y métodos directamente
- Sintaxis clara y concisa
- Ideal para objetos únicos o configuraciones

Creación Basada en Prototipos

Usando `Object.create()` para heredar

```
1 const goblin = Object.create(enemigo);
2 goblin.vida = 50;
3 goblin.damage = 8;
4
5 console.log(goblin.atacar());
6 // "Enemigo ataca causando 8 puntos"
```

- `goblin` hereda el método `atacar()` de `enemigo`
- Las propiedades propias sobrescriben las heredadas
- `this` se refiere al objeto que invoca el método

Herencia con Prototipos - Concepto

Creando cadenas de herencia

- Cada objeto puede heredar de otro objeto
- Se forma una cadena de prototipos
- JavaScript busca propiedades/métodos en la cadena hasta encontrarlos
- Permite estructurar jerarquías de objetos

Herencia con Prototipos - Base

Creando el objeto base

```
1 const personajeBase = {  
2     mover(x, y) {  
3         this.x += x;  
4         this.y += y;  
5     },  
6     x: 0,  
7     y: 0  
8 };
```

- Define comportamiento común
- Puede ser heredado por múltiples objetos
- Los métodos usan `this` para acceder a propiedades

Herencia con Prototipos - Extensión

Creando un prototipo intermedio

```
1 const prototipoJugador = Object.create(personajeBase);
2 prototipoJugador.atacar = function(objetivo) {
3     return `${this.nombre} ataca a ${objetivo.nombre}`;
4 };
5 prototipoJugador.nombre = "Sin nombre"
```

- Hereda de `personajeBase`
- Añade funcionalidad específica
- Añade un nombre por defecto.
- Forma el segundo eslabón de la cadena

Herencia con Prototipos - Instancia

Creando la instancia final

```
1 const jugador = Object.create(prototipoJugador);
2 jugador.nombre = "Aragorn";
3 jugador.x = 10;
4 jugador.y = 10;
5 jugador.puntuacion = 3;
```

- Tiene acceso a `mover()` y `atacar()`
- Define sus propias propiedades
- Propiedades que ocultan las del prototipo (locales) y nuevas.
- Completa la cadena de herencia

Vision Completa

Base

```

1 let personajeBase = {
2   mover(x, y) {
3     this.x += x;
4     this.y += y;
5   },
6   x: 0,
7   y: 0
8 };

```

Prototipo

```

1 let prototipoJugador = Object.create(personajeBase);
2 prototipoJugador.atacar = function(objetivo) {
3   return `${this.nombre} ataca a ${objetivo.nombre}`;
4 };
5 prototipoJugador.nombre = "Sin nombre"

```

Instancia

```

1 let jugador = Object.create(prototipoJugador);

```

- Acceder a `prototipoJugador.nombre` y a `jugador.nombre`
- añadir los cambios a nivel de instancia y ejecutar de nuevo.
- Comprobar las propiedades con `Object.keys(INSTANCIA)` antes y después.

```

1 jugador.nombre = "Aragorn";
2 jugador.x = 10;

```

Acceso a Propiedades

Notación punto

La forma más común y legible

```

1 const config = {
2   sonido: true,
3   volumen: 0.8,
4   idioma: "es"
5 };
6
7 console.log(config.sonido);
8 config.volumen = 0.5;

```

- Sintaxis clara: `objeto.propiedad`
- Recomendada cuando el nombre es conocido
- No funciona con nombres dinámicos

Notación corchetes

Para propiedades dinámicas o especiales

```

1 console.log(config["idioma"]);
2
3 const propiedad = "volumen";
4 console.log(config[propiedad]);

```

- Permite nombres de propiedad dinámicos
- Útil con variables o bucles
- Necesaria para nombres con espacios o caracteres especiales

Añadir Propiedades Dinámicamente

JavaScript permite modificar objetos en tiempo de ejecución

```
1 config.dificultad = "normal";
2 config["nivel-maximo"] = 10;
```

- Se pueden añadir propiedades después de crear el objeto
- Característica de la naturaleza dinámica de JavaScript
- Útil pero puede complicar el seguimiento del código

Iteración sobre Propiedades - For...in

Recorriendo todas las propiedades

```
1 const inventario = {  
2   espada: 1,  
3   pocion: 5,  
4   oro: 150  
5 };  
6  
7 for (let item in inventario) {  
8   console.log(`${item}: ${inventario[item]}`);  
9 }
```

- Itera sobre propiedades enumerables
- Incluye propiedades heredadas del prototipo
- La variable toma el nombre de cada propiedad

Iteración sobre Propiedades - Verificación

Comprobando existencia de propiedades

```
1 if ("oro" in inventario) {  
2     console.log("El jugador tiene oro");  
3 }  
4  
5 if (inventario.hasOwnProperty("espada")) {  
6     console.log("Propiedad propia, no heredada");  
7 }
```

- `in` verifica si existe la propiedad (propia o heredada)
- `hasOwnProperty()` verifica solo propiedades propias
- No necesitamos acceder al valor para verificar

Iteración sobre Propiedades - Arrays

Obteniendo claves y valores como arrays

```
1 const items = Object.keys(inventario);
2 // ["espada", "potion", "oro"]
3
4 const cantidades = Object.values(inventario);
5 // [1, 5, 150]
6
7 const pares = Object.entries(inventario);
8 // [["espada", 1], ["potion", 5], ["oro", 150]]
```

- Útil para trabajar con métodos de array (map, filter, etc.)
- Solo devuelven propiedades propias enumerables

Función Constructor - Definición

El patrón clásico pre-ES2015

```
1 function Jugador(nombre, x, y) {  
2     this.nombre = nombre;  
3     this.x = x;  
4     this.y = y;  
5     this.vida = 100;  
6 }
```

- Se invoca con `new` para crear instancias
- `this` se refiere al nuevo objeto creado
- Inicializa las propiedades de la instancia
- Por convención, nombre en PascalCase

Función Constructor - Métodos

Añadiendo métodos al prototipo

```
1 Jugador.prototype.mover = function(deltaX, deltaY) {  
2     this.x += deltaX;  
3     this.y += deltaY;  
4 };  
5  
6 Jugador.prototype.mostrarPosicion = function() {  
7     return `${this.nombre} está en (${this.x}, ${this.y})`;  
8 };
```

- Los métodos van en el prototipo, no en el constructor
- Se comparten entre todas las instancias (ahorra memoria)
- Tienen acceso a `this` de la instancia

Función Constructor - Uso

Creando instancias

```
1 const player1 = new Jugador("Aragorn", 10, 20);
2 const player2 = new Jugador("Legolas", 15, 25);
3
4 player1.mover(2, 3);
5 console.log(player1.mostrarPosicion());
6 // "Aragorn está en (12, 23)"
```

- Siempre usar `new` para invocar constructores
- Cada instancia tiene sus propias propiedades
- Los métodos son compartidos vía prototipo

Herencia con Constructor - Clase Base

Definiendo el constructor padre

```
1 function Personaje(nombre, vida) {  
2     this.nombre = nombre;  
3     this.vida = vida;  
4 }  
5  
6 Personaje.prototype.saludar = function() {  
7     return `Hola, soy ${this.nombre}`;  
8 };
```

- Constructor base con propiedades comunes
- Métodos compartidos en el prototipo

Herencia con Constructor - Clase Hija

Extendiendo la funcionalidad

```
1 function Guerrero(nombre, vida, fuerza) {  
2     Personaje.call(this, nombre, vida);  
3     this.fuerza = fuerza;  
4 }
```

- `Personaje.call(this, ...)` llama al constructor padre
- Inicializa las propiedades heredadas
- Añade propiedades específicas de `Guerrero`

Herencia con Constructor - Cadena de Prototipos

Estableciendo la herencia correctamente

```
1 Guerrero.prototype = Object.create(Personaje.prototype);
2 Guerrero.prototype.constructor = Guerrero;
3
4 Guerrero.prototype.atacar = function() {
5     return `${this.nombre} ataca con fuerza ${this.fuerza}`;
6 }
```

- `Object.create()` establece el prototipo correcto
- Restauramos la propiedad `constructor`
- Ahora podemos añadir métodos específicos

Herencia con Constructor - Resultado

Usando la herencia completa

```
1 const conan = new Guerrero("Conan", 150, 25);
2
3 console.log(conan.saludar());
4 // "Hola, soy Conan" (heredado de Personaje)
5
6 console.log(conan.atacar());
7 // "Conan ataca con fuerza 25" (propio de Guerrero)
```

- `conan` tiene acceso a métodos de ambos constructores
- Patrón verboso y propenso a errores

Clases ES2015+ - Introducción

Sintaxis moderna y clara

- Introducida en ES2015 (ES6)
- Azúcar sintáctico sobre prototipos
- Más familiar para desarrolladores de otros lenguajes OO
- Sintaxis más estructurada y menos propensa a errores

Clases ES2015+ - Definición Básica

Declarando una clase

```
1 class GameObject {  
2     constructor(x, y) {  
3         this.x = x;  
4         this.y = y;  
5         this.activo = true;  
6     }  
7 }
```

- Palabra clave `class` seguida del nombre
- Método `constructor()` se ejecuta al crear instancias
- Inicializa las propiedades del objeto

Clases ES2015+ - Métodos

Añadiendo comportamiento

```
1 class GameObject {
2     constructor(x, y) {
3         this.x = x;
4         this.y = y;
5         this.activo = true;
6     }
7
8     actualizar(deltaTime) {
9         if (!this.activo) return;
10        // Lógica de actualización
11    }
12
13    destruir() {
14        this.activo = false;
15    }
16 }
```

- Los métodos se definen dentro de la clase
- No se usa `function` keyword
- Automáticamente se añaden al prototipo

Herencia con Clases - Extends

Creando subclases

```
1 class Enemigo extends GameObject {  
2     constructor(x, y, tipo) {  
3         super(x, y); // Llamada al constructor padre  
4         this.tipo = tipo;  
5         this.vida = 50;  
6         this.velocidad = 2;  
7     }  
8 }
```

- `extends` establece la herencia
- `super()` llama al constructor de la clase padre
- Debe ser la primera línea del constructor hijo

Herencia con Clases - Sobrescritura

Extendiendo métodos heredados

```
1 class Enemigo extends GameObject {  
2     // ... constructor ...  
3  
4     actualizar(deltaTime) {  
5         super.actualizar(deltaTime);  
6         if (this.vida > 0) {  
7             this.x += this.velocidad;  
8         }  
9     }  
10 }
```

- Podemos sobrescribir métodos del padre
- `super.nombreMetodo()` llama a la versión del padre
- Permite extender sin reemplazar completamente

Getters y Setters - Concepto

Propiedades calculadas y validación

- Getters: propiedades de solo lectura o calculadas
- Setters: validación al asignar valores
- Se usan como propiedades normales (sin paréntesis)
- Permiten encapsular lógica de acceso

Getters y Setters - Getter

Propiedades de solo lectura

```
1 class Enemigo extends GameObject {
2     constructor(x, y, tipo) {
3         super(x, y);
4         this._vida = 50;
5     }
6
7     get estaVivo() {
8         return this._vida > 0;
9     }
10 }
11
12 const orc = new Enemigo(10, 20, "Orc");
13 if (orc.estaVivo) { // Sin paréntesis
14     console.log("El enemigo sigue vivo");
15 }
```

- Palabra clave `get` antes del nombre
- Se accede como propiedad, no como método
- Útil para valores derivados

Getters y Setters - Setter

Validación al asignar

```
1 class Enemigo extends GameObject {  
2     // ... constructor ...  
3  
4     set vida(valor) {  
5         this._vida = Math.max(0, valor);  
6         if (this._vida === 0) {  
7             this.destruir();  
8         }  
9     }  
10  
11    get vida() {  
12        return this._vida;  
13    }  
14 }
```

- Palabra clave `set` antes del nombre
- Permite validar y ejecutar lógica adicional
- Por convención, la propiedad real usa `_` como prefijo

Getters y Setters - Uso

Asignación transparente

```
1 const goblin = new Enemigo(5, 10, "Goblin");
2
3 goblin.vida = 30;      // Usa el setter
4 console.log(goblin.vida); // Usa el getter
5
6 goblin.vida = -10;     // Setter lo convierte a 0
7 // El setter también llama a destruir()
```

- Sintaxis de propiedad normal
- El setter ejecuta validación automáticamente
- Transparente para el código que usa la clase

Métodos Estáticos - Concepto

Métodos de la clase, no de instancias

- Pertenecen a la clase, no a objetos individuales
- Se invocan sobre la clase misma
- No tienen acceso a `this` de instancia
- Útiles para funciones de utilidad o métodos factoría

Métodos Estáticos - Definición

Palabra clave `static`

```
1 class Enemigo extends GameObject {  
2     // ... resto del código ...  
3  
4     static crearOrc() {  
5         const orc = new Enemigo(0, 0, "Orc");  
6         orc.vida = 80;  
7         orc.velocidad = 1.5;  
8         return orc;  
9     }  
10  
11    static crearGoblin() {  
12        const goblin = new Enemigo(0, 0, "Goblin");  
13        goblin.vida = 30;  
14        goblin.velocidad = 3;  
15        return goblin;  
16    }  
17 }
```

- Prefijo `static` antes del método
- Patrón factoría para crear instancias preconfiguradas

Métodos Estáticos - Uso

Invocación sobre la clase

```
1 const orc = Enemigo.crearOrc();
2 const goblin = Enemigo.crearGoblin();
3
4 // NO se puede llamar sobre instancias
5 const enemigo = new Enemigo(5, 5, "Slime");
6 // enemigo.crearOrc(); // ✗ Error
```

- Se llaman con `NombreClase.metodoEstatico()`
- No están disponibles en las instancias
- Similares a métodos de clase en Java/C++

Uso de Clases - Creación

Instanciando objetos

```
1 const goblin = new Enemigo(10, 20, "Goblin");
2 goblin.vida = 30;
3
4 const orc = Enemigo.crearOrc(); // Método estático
```

- Constructor normal con `new`
- Podemos usar setters para asignar valores
- Los métodos estáticos crean instancias preconfiguradas

Uso de Clases - Polimorfismo

Tratamiento uniforme de objetos diferentes

```
1 const enemigos = [goblin, orc];
2
3 enemigos.forEach(enemigo => {
4     enemigo.actualizar(16);
5     if (enemigo.estaVivo) {
6         enemigo.mover(1, 0);
7     }
8});
```

- Todos los enemigos comparten la misma interfaz
- Podemos tratarlos de forma uniforme
- Cada uno puede tener comportamiento específico

Ventajas de Clases ES2015+

Por qué usar la sintaxis moderna

- **Sintaxis más clara:** Familiar para otros lenguajes OO
- **Herencia simplificada:** `extends` y `super()` más directos
- **Menos errores:** Estructura más rígida previene problemas comunes
- **Getters y setters integrados:** Encapsulación natural
- **Métodos estáticos:** Organización clara de funciones de utilidad
- **Mejor soporte de herramientas:** IDEs y linters funcionan mejor



Tip

La sintaxis de clases es la **recomendada** para proyectos nuevos

Resumen: Prototipos vs Clases

Dos formas, mismo resultado

Aspecto	Prototipos	Clases ES2015+
Sintaxis	Verbosa y manual	Clara y estructurada
Herencia	<code>Object.create()</code> y <code>call()</code>	<code>extends</code> y <code>super()</code>
Métodos	<code>Funcion.prototype.metodo</code>	Dentro de la clase
Uso actual	Legacy code	Recomendado

 **Importante**

Las clases son azúcar sintáctico: por debajo siguen usando prototipos