

CODING BASE 2 · QA ENGINEERING

Coding Classes

Del código suelto a las **clases**.

El último escalón antes de la arquitectura profesional de automation.

JS moderno

map / filter

class + extends

Page Object

6 quizzes ★

Ya hablas el idioma. Hoy aprendes a construir.

✓ Coding Base

(sesión 1)

Valores · variables · operadores ·
funciones · if/else · loops ·
arrays · objetos · async/await ·
tu primer test ejecutado.

🎯 Coding Classes

(hoy)

JS moderno (el que ves en
specs reales) → arrays con
superpoderes → **clases y**
herencia → tu primer **Page**
Object.

// El zoom-out sigue creciendo:

función

← *ya la dominas*



class

← *hoy: funciones + datos juntos*



Page Object

← *hoy: una clase que envuelve una página*



Framework KATA

← *próxima parada: el repo profesional*

★ QUIZ relámpago · ¿Sigues caliente de la sesión 1?

¿Cómo accedo al email de este objeto?

```
const creds = { email: "ana@qa.com", password: "secreto123" };
```

A `creds[email]` — con corchetes directos

B `creds.email` — el punto de toda la vida

C `creds→email` — con flecha

💡 El punto: "del objeto creds, dame email". Lo usaste 20 veces en la sesión 1 — y hoy lo vas a usar con `this`: `this.email` significa "de ESTA instancia, dame email". Guarda esa idea.

js moderno

arrays pro

clases

herencia

page object

Template literals: texto con huecos

😬 Como lo hacías ayer

```
const msg = "Hola, " + name + "! Tienes " + bugs + " bugs";
```

Funciona, pero con 3+ variables se vuelve sopa de comillas y signos + .

😎 Desde hoy

```
const msg = `Hola, ${name}! Tienes ${bugs} bugs`;
```

Backticks ``` (no comillas) + `${...}` = hueco donde entra cualquier variable o expresión.



El backtick ``` está junto al 1 en teclado US, o `AltGr + }` en teclado ES/LA. Práctica obligada: lo usarás en cada test.

Arrow functions: ya las usabas sin saberlo

La misma función, en 3 pasos de transformación:

// 1. La clásica de la sesión 1

```
function double(n) { return n * 2; }
```

// 2. Misma función, con flecha

```
const double = (n) => { return n * 2; };
```

// 3. Una sola expresión → return implícito

```
const double = (n) => n * 2;
```

La flecha =>

Se lee: "recibe `n` → devuelve `n * 2`". Función compacta, ideal para pasarla a OTRA función.

👀 ¿Te suena de algún lado?

```
test("login", async ({ page }) => {  
  ...  
});
```

¡Tu test de la sesión 1 ERA una arrow function! Hoy por fin sabes leer cada símbolo.

[js moderno](#)[arrays pro](#)[clases](#)[herencia](#)[page object](#)

Destructuring: desempacar cajas

```
const user = { name: "Ana", role: "admin" };
```

// ayer: uno por uno

```
const name = user.name;
```

```
const role = user.role;
```

// hoy: desempaca de un tirón

```
const { name, role } = user;
```

La regla

Llaves a la IZQUIERDA del `=` significan "extrae estas propiedades del objeto de la derecha". Los nombres deben coincidir.

El misterio resuelto

```
async ({ page }) => { ... }
```

Playwright entrega un objeto LLENO de herramientas.

`( { page } )` = "del paquete que me das, solo desempaco **page**". Misterio de la sesión 1: cerrado. 

JS moderno, a tus dedos

modern.js – editable

⚡ Ejecutable en la versión online

```
const user = { name: "Ana", role: "admin", bugs: 12 };
```

```
// 1) Template literal: texto con ${huecos}
```

```
console.log(`${user.name} reportó ${user.bugs} bugs`);
```

```
// 2) Arrow function compacta
```

```
const double = (n) => n * 2;
```

```
console.log(double(21));
```

```
// 3) Destructuring: desempacar
```

```
const { name, role } = user;
```

```
console.log(`${name} es ${role}`);
```

```
// Reto 1: crea greet = (who) => `Hola, ${who}!` y úsala
```

```
// Reto 2: desempaca también "bugs" en la línea del destructuring
```

★ QUIZ 1 • JS moderno

En `test("login", async ({ page }) => {...})` — ¿qué hace `({ page })`?

A

Crea un objeto nuevo llamado `page`

B

Desempaca la propiedad `page` del objeto de herramientas que Playwright entrega

C

Es decoración obligatoria de la sintaxis



Destructuring en los parámetros: Playwright llama a tu función pasándole un objeto lleno de fixtures (`page`, `request`, `context...`) y tú desempacas solo lo que usas. Por eso a veces verás `({ page, request })` — desempacando dos.

js moderno

arrays pro

clases

herencia

page object

.map(): transformar cada elemento

```
const users = ["ana", "luis", "mia"];

const emails = users.map(u => `${u}@qa.com`);

// ["ana@qa.com", "luis@qa.com", "mia@qa.com"]
```

Se lee: "**por cada** `u` del array, **devuélveme** ``${u}@qa.com``". Array nuevo, mismo tamaño, elementos transformados.

El loop de ayer

```
const emails = [];
for (const u of users) {
  emails.push(`${u}@qa.com`);
}
```

Mismo resultado, 4 líneas. `map` lo dice en 1: arrow functions pagando dividendos.

En testing

Generar emails de prueba, extraer ids de una respuesta de API, preparar la columna de datos que el test necesita.

js moderno

arrays pro

clases

herencia

page object

.filter() y .find(): elegir con criterio 🎯

🗂️ filter — todos los que cumplen

```
const users = [  
  { name: "Ana", role: "admin" },  
  { name: "Luis", role: "viewer" },  
  { name: "Mia", role: "admin" },  
];  
  
const admins = users.filter(u => u.role === "admin");  
// [Ana, Mia] - array, puede venir vacío []
```

🔍 find — el primero que cumple

```
const luis = users.find(u => u.name === "Luis");  
// { name: "Luis", role: "viewer" }  
// UN objeto - o undefined si nadie cumple
```

✍️ Tu tabla de test data, filtrada con código: "dame solo los admins", "encuentra el usuario inactivo". La condición es la MISMA del if/else de ayer.

Tu test data, dominada

data-power.js – editable

⚡ Ejecutable en la versión online

```
const users = [
  { name: "Ana", role: "admin", bugs: 12 },
  { name: "Luis", role: "viewer", bugs: 3 },
  { name: "Mia", role: "admin", bugs: 7 },
];

const names = users.map(u => u.name);
console.log(names);

const admins = users.filter(u => u.role === "admin");
console.log(`Hay ${admins.length} admins`);

const top = users.find(u => u.bugs > 10);
console.log(`Top bug hunter: ${top.name}`);

// Reto 1: filtra los usuarios con 7+ bugs
// Reto 2: del resultado, saca solo los nombres con map
```

★ QUIZ 2 · Arrays con superpoderes

¿Qué devuelve `[1, 2, 3, 4].filter(n => n > 2)`?

- ☐ A `[1, 2]` — los que NO cumplen
- ☐ B `3` — el primero que cumple
- ☒ C `[3, 4]` — array con todos los que cumplen

💡 `filter` = colador: deja pasar los elementos cuya condición da `true` y devuelve un **array nuevo** con ellos. La opción B describe a `find` (que devolvería `3`, sin corchetes) — esa es exactamente la diferencia entre ambos.

js moderno

arrays pro

clases

herencia

page object

El problema: datos y funciones viven separados

🤪 Todo suelto

```
const tester1 = { name: "Ana", bugs: 0 };
const tester2 = { name: "Luis", bugs: 0 };

function findBug(tester) {
  tester.bugs = tester.bugs + 1;
}

// ¿qué funciones van con qué objetos?
// ¿quién me garantiza que tester tiene .bugs?
```

🧩 El síntoma

Objetos por un lado, funciones por otro. Con 5 objetos y 15 funciones, nadie sabe qué combina con qué.

💡 La idea

Una **clase** junta en un solo lugar: los **datos** (propiedades) + las **funciones que operan sobre ellos** (métodos). Un molde 🍪 que fabrica objetos ya equipados.

Anatomía de una clase

```
class Tester {  
  constructor(name) {  
    this.name = name;  
    this.bugs = 0;  
  }  
  
  findBug() {  
    this.bugs = this.bugs + 1;  
    return `${this.name}: ${this.bugs} bugs`;  
  }  
}
```

class Tester

"Defino el molde Tester." Nombre SIEMPRE con mayúscula inicial — convención universal.

constructor(name)

La función que corre automáticamente **al fabricar** cada objeto. Recibe los datos iniciales.

this

"**Este** objeto en particular" — la galleta que se está fabricando o usando, no el molde.

findBug()

Un **método**: función que vive DENTRO de la clase y trabaja con los datos de `this`. Sin la palabra function.

new: fabricar galletas 🍪

```
const ana = new Tester("Ana");  
const luis = new Tester("Luis");  
  
ana.findBug(); // "Ana: 1 bugs"  
ana.findBug(); // "Ana: 2 bugs"  
luis.findBug(); // "Luis: 1 bugs" ← ¡SU propio contador!
```

🏠 new Tester("Ana")

"Fabrica un objeto con el molde Tester" → corre el constructor con

`name = "Ana"` → entrega la

instancia.

🍪 Cada instancia, su memoria

`ana.bugs` y `luis.bugs` son contadores **independientes**. El molde es uno; las galletas, muchas.

👁️ Spoiler

`page` de Playwright... es una instancia de una clase.

`page.click()` = método. Llevas una sesión entera usando clases.

this, paso a paso

```
class User {  
  constructor(email) {  
    this.email = email;  
  }  
  greet() {  
    return `Hola, ${this.email}`;  
  }  
}  
  
const ana = new User("ana@qa.com");  
console.log(ana.greet());
```

▶ Siguiente paso

↶ Reiniciar

paso 0 / 6

📦 MEMORIA

— vacía —

Presiona "Siguiente paso": vas a ver a quién apunta **this** en cada momento.

consola...

Fabrica tus propias instancias

tester-class.js – editable

⚡ Ejecutable en la versión online

```
class Tester {
  constructor(name) {
    this.name = name;
    this.bugs = 0;
  }

  findBug() {
    this.bugs = this.bugs + 1;
    return `${this.name} encontró un bug! Total: ${this.bugs}`;
  }
}

const ana = new Tester("Ana");
console.log(ana.findBug());
console.log(ana.findBug());

const luis = new Tester("Luis");
```

Después de este código, ¿cuánto vale `luis.bugs`?

```
const ana = new Tester("Ana");  
const luis = new Tester("Luis");  
ana.findBug();  
ana.findBug();  
ana.findBug();
```

A

3 — los contadores se comparten

B

0 — cada instancia tiene su propia memoria

C

undefined — luis no tiene bugs

💡 `new` fabrica objetos **independientes**: los 3 `findBug()` fueron sobre `ana` (`this = ana`), así que `luis.bugs` sigue en su valor inicial 0 del constructor. El molde es uno, cada galleta lleva su propia cuenta. 🍪

[js moderno](#)[arrays pro](#)[clases](#)[herencia](#)[page object](#)

Nuevo dolor: clases que se repiten 🙄

```
class Admin {  
  constructor(name) { this.name = name; }  
  login() { return `${this.name} entró`; }  
  deleteProject() { return "💣 borrado"; }  
}
```

```
class Viewer {  
  constructor(name) { this.name = name; }  
  login() { return `${this.name} entró`; }  
  watch() { return "👁 mirando"; }  
}
```



constructor y login() están **copiados y pegados**. ¿Te suena? Es el mismo dolor de los test cases repetidos de la sesión 1 — y la solución tiene la misma forma: escribir lo común UNA vez.

extends: heredar lo común

```
class User {  
  constructor(name) { this.name = name; }  
  login() { return `${this.name} entró`; }  
}  
  
class Admin extends User {  
  deleteProject() { return "💣 borrado"; }  
}  
  
const ana = new Admin("Ana");  
ana.login();           // heredado del padre ✅  
ana.deleteProject();  // propio de Admin ✅
```

extends

"Admin **es un** User, más sus extras." Hereda constructor, propiedades y métodos del padre — gratis.

super(...)

Si el hijo define su PROPIO constructor, la primera línea debe ser

`super(name)` : "padre, haz tu parte primero". Si no define constructor, hereda el del padre directo.

Por qué te importa

El framework KATA al que vas es una cadena de `extends` : tu

Page **hereda** docenas de helpers ya escritos. Entender esto = leer KATA sin miedo.

Herencia en acción

inheritance.js – editable

⚡ Ejecutable en la versión online

```
class User {
  constructor(name) {
    this.name = name;
  }
  login() {
    return `${this.name} entró al sistema`;
  }
}

class Admin extends User {
  constructor(name) {
    super(name); // "padre, haz tu parte primero"
    this.power = "total";
  }
  deleteProject() {
    return `${this.name} borró el proyecto 💥`;
  }
}
```

★ QUIZ 4 • Herencia

En `class Admin extends User` — ¿qué hace `super(name)`?

A

Crea una instancia nueva de User

B

Llama al constructor del padre para que inicialice su parte (`this.name`)

C

Borra los métodos heredados del padre

💡 `super(name)` = "padre, ejecuta TU constructor con este dato". Así User asigna `this.name` y luego Admin agrega lo suyo. Sin esa llamada, el lenguaje lanza error: el hijo no puede usar `this` antes de que el padre haga su parte.

[js moderno](#)[arrays pro](#)[clases](#)[herencia](#)[page object](#)

Mira este spec real. ¿Notas algo? 🔍

login.spec.ts - 3 tests

```
test("login exitoso", async ({ page }) => {  
  await page.fill("#email", "ana@qa.com");  
  await page.fill("#password", "secreto123");  
  await page.click("#login-btn");  
  await expect(page.locator("#welcome")).toBeVisible();  
});
```

*los mismos
3 pasos...*

```
test("login con password inválida", async ({ page }) => {  
  await page.fill("#email", "ana@qa.com");  
  await page.fill("#password", "maLaMaLa");  
  await page.click("#login-btn");  
  await expect(page.locator("#error-msg")).toBeVisible();  
});
```

...otra vez...

// test 3, test 4, test 5... ¿y si mañana #login-btn cambia de id? 💀

[js moderno](#)[arrays pro](#)[clases](#)[herencia](#)[page object](#)

Page Object: una clase que envuelve una página

```
class LoginPage {  
  constructor(page) {  
    this.page = page;    // guarda el "control remoto"  
  }  
  
  async login(email, password) {  
    await this.page.fill("#email", email);  
    await this.page.fill("#password", password);  
    await this.page.click("#login-btn");  
  }  
}
```

La jugada

TODO lo que se puede hacer en la página de login, encapsulado en UNA clase. Los selectores viven aquí y solo aquí.

constructor(page)

Recibe el `page` de Playwright y lo guarda en

`this.page` — el control remoto del navegador, ahora de la instancia.

Todo lo de hoy, junto

class ✓ · constructor ✓ · this ✓ · método async ✓ · await ✓. Cero conceptos nuevos — solo composición.

El refactor: antes / después

✗ ANTES — 8 LÍNEAS POR TEST

```
test("login exitoso", async ({ page }) => {  
  await page.fill("#email", "ana@qa.com");  
  await page.fill("#password", "secreto123");  
  await page.click("#login-btn");  
  await expect(page.locator("#welcome"))  
    .toBeVisible();  
});
```

✓ DESPUÉS — EL TEST CUENTA UNA HISTORIA

```
test("login exitoso", async ({ page }) => {  
  const loginPage = new LoginPage(page);  
  await loginPage.login("ana@qa.com", "secreto123");  
  await expect(page.locator("#welcome"))  
    .toBeVisible();  
});
```



El test ahora se lee como un test case manual: "crea la página de login, entra con estas credenciales, verifica la bienvenida".
Y si #login-btn cambia → corriges **1 línea** dentro de LoginPage, no 50 tests.

Tu Page Object, ejecutándose en vivo

https://bunkai-bank.test/login

**Bunkai Bank**

Acceso a tu cuenta

EMAIL

PASSWORD

Entrar

login-page.spec.ts — ¡código real, clases reales!

⚡ Ejecutable en la versión online

```
class LoginPage {
  constructor(page) {
    this.page = page;
  }
  async login(email, password) {
    await this.page.fill("#email", email);
    await this.page.fill("#password", password);
    await this.page.click("#login-btn");
  }
}

test("login con Page Object", async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.login("ana@qa.com", "secreto123");
  await expect(page.locator("#welcome")).toBeVisible();
});
```

► Ejecuta — esta vez tu CLASE controla el cursor...

★ QUIZ 5 • El integrador

El dev renombra `#login-btn` a `#signin-btn`. Tienes 50 tests que hacen login usando `LoginPage`. ¿Cuántos lugares corriges?

A 50 — uno por test

B Ninguno — Playwright lo detecta solo

C 1 — el selector vive solo dentro de `LoginPage`

💡 Esa es TODA la promesa del Page Object: los selectores viven en un único lugar. Los 50 tests llaman `LoginPage.login(...)` y ni se enteran del cambio. Acabas de entender por qué TODA arquitectura profesional de automation (incluido KATA) se construye sobre clases.

KATA es esto mismo... con esteroides 🥋

TestContext config, datos fake, utilidades

↓ extends

UiBase helpers de Playwright ya escritos

↓ extends

★ LoginPage ¡ESTO ya lo sabes construir!

↓ usada por

TestFixture el ({ page }) pero con tus clases

↓ usada por

login.spec.ts tests que cuentan historias



Cada capa es una clase que extends la anterior — la cadena de herencia que acabas de dominar. Tu LoginPage de hoy es el corazón del framework profesional.

MISIÓN CUMPLIDA

Tu puntaje final

— / 6



Re-practica en casa

Este archivo funciona offline: re-juega los 4 playgrounds y los retos.



Reto puente

Escribe un `SignupPage` imaginario en papel: ¿qué métodos tendría? ¿qué selectores guardaría?



Próxima sesión

Playwright real contra **UPEX Dojo**: instalas el kit y tus Page Objects controlan un navegador DE VERDAD.



Hoy: JS moderno → map/filter/find → clases → this → herencia → **tu primer Page Object**. El gap más grande del camino: cerrado.