Evidencias varios 13

```html
</head>
<body>
    <h1> Iteración Avanzada</h1>

    <section id="generators-advanced-iteration">
        <h2>12. Generators, advanced iteration</h2>

        <section id="generators-12-1">
            <h3>12.1 Generators</h3>
            <p>Este bloque demuestra el uso de funciones generadoras en JavaScript.</p>
            <div class="resultado" id="resultado-12-1">
                <!-- Aquí se mostrarán los resultados de los generadores -->
                Cargando resultados del generador...
            </div>
        </section>

        <section id="generators-12-2">
            <h3>12.2 Async iteration and generators</h3>
            <p>Este bloque explora la iteración asíncrona y los generadores asíncronos.</p>
            <div class="resultado" id="resultado-12-2">
                <!-- Aquí se mostrarán los resultados de la iteración asíncrona -->
                Cargando resultados de iteración asíncrona...
            </div>
        </section>
    </section>

    <footer>
        <p>&copy; 2023 [Tu Nombre Completo]. Prácticas de JavaScript. Grado 10, [Tu Grupo].</p>
    </footer>

    <script src="main.js"></script>
</body>
</html>
```

```javascript
// TEMA: 14.1 Proxy y reflejo — 10/05/2024
// Este bloque demuestra cómo usar un Proxy para interceptar operaciones en un objeto.
// Un Proxy permite interceptar y personalizar operaciones fundamentales para objetos,
// como búsquedas de propiedades, asignaciones, enumeración, invocación de funciones, etc.
// Aprendí que los proxies son poderosos para validación, logging, o incluso para crear objetos vi
console.log("--- 14.1 Proxy y reflejo ---");

let user = {
    name: "Juan",
    age: 30
};

let handler = {
    get(target, prop, receiver) {
        console.log(`GET ${prop}`); // Intercepta la lectura de propiedades
        return Reflect.get(target, prop, receiver);
    },
    set(target, prop, value, receiver) {
        if (prop === 'age' && value < 18) {
            console.warn("La edad no puede ser menor de 18.");
            return false; // Rechaza la operación
        }
        console.log(`SET ${prop} = ${value}`); // Intercepta la escritura de propiedades
        return Reflect.set(target, prop, value, receiver);
    }
};

let userProxy = new Proxy(user, handler);

console.log(userProxy.name); // Acceso a la propiedad, se activa el 'get'
userProxy.age = 25;          // Modificación de la propiedad, se activa el 'set'
console.log(userProxy.age);

userProxy.age = 15;          // Intento de modificar a un valor inválido
console.log(userProxy.age);  // La edad no debería haber cambiado
document.getElementById('output-14-1').innerText = "Ver consola para resultados de Proxy (F12)";
```

```javascript
// TEMA: 14.3 Curry — 10/05/2024
// Este bloque demuestra la técnica de currificación (Curry), que transforma una función que toma múlti
// argumentos en una secuencia de funciones, cada una tomando un solo argumento.
// Aprendí que la currificación mejora la reusabilidad de funciones y permite crear funciones más
// especializadas a partir de funciones genéricas.
console.log("\n--- 14.3 Curry ---");

// Función sin currificación
function add(a, b, c) {
    return a + b + c;
}
console.log("add(1, 2, 3): " + add(1, 2, 3));

// Función currificada
function curry(func) {
    return function curried(...args) {
        if (args.length >= func.length) { // Si ya tenemos suficientes argumentos
            return func.apply(this, args);
        } else { // Si no, devolvemos una nueva función que espera los argumentos restantes
            return function(...args2) {
                return curried.apply(this, args.concat(args2));
            };
        }
    };
}

let curriedAdd = curry(add);

console.log("curriedAdd(1)(2)(3): " + curriedAdd(1)(2)(3));
console.log("curriedAdd(1, 2)(3): " + curriedAdd(1, 2)(3));
console.log("curriedAdd(1, 2, 3): " + curriedAdd(1, 2, 3));

const add5 = curriedAdd(5);
console.log("add5(10)(20): " + add5(10)(20)); // Esto es add(5, 10, 20)
```

```javascript
// TEMA: 14.5 Bigint — 10/05/2024
// Este bloque demuestra el uso del tipo de dato `BigInt`, introducido para manejar números enteros
// de una magnitud arbitraria, más allá del límite de `Number.MAX_SAFE_INTEGER` (2^53 - 1).
// Aprendí que los `BigInt` se crean añadiendo `n` al final de un número entero o usando `BigInt()`.
// No se pueden mezclar con `Number` en operaciones directas; hay que convertir explícitamente.
console.log("\n--- 14.5 Bigint ---");

const maxSafe = Number.MAX_SAFE_INTEGER;
console.log("Número máximo seguro (Number.MAX_SAFE_INTEGER): " + maxSafe);
console.log("maxSafe + 1 === maxSafe + 2:", maxSafe + 1 === maxSafe + 2); // true, ¡error de precisión!

const largeNumber = 9007199254740991n; // Sufijo 'n' para BigInt
const largerNumber = largeNumber + 2n; // Operaciones con BigInt
console.log("Número grande con BigInt: " + largerNumber);
console.log("largeNumber + 1n === largeNumber + 2n:", (largeNumber + 1n) === (largeNumber + 2n)); // fal

const anotherBigInt = BigInt("1234567890123456789");
console.log("Otro BigInt desde string: " + anotherBigInt);

try {
    console.log(largeNumber + 1); // Esto causará un TypeError
} catch (e) {
    console.error("Error al mezclar BigInt y Number:", e.message);
}

document.getElementById('output-14-5').innerText = "Ver consola para resultados de BigInt (F12)";


// TEMA: 14.6 Unicode, componentes internos de cadena — 10/05/2024
// Este bloque explora cómo JavaScript maneja los caracteres Unicode y los componentes internos de las c
// JavaScript utiliza UTF-16 para representar cadenas, donde la mayoría de los caracteres comunes ocupan
// (16 bits), pero los caracteres suplementarios (emojis, etc.) ocupan 2 unidades de código (pares subro
// Aprendí sobre `codePointAt()` para obtener el punto de código Unicode real y cómo `length` puede ser
console.log("\n--- 14.6 Unicode, componentes internos de cadena ---");
```