

Ejercicio: Programa para gestionar reservas en alojamientos rurales

Resultado de aprendizaje en alternancia:

R.A 7: Desarrolla programas aplicando características avanzadas de los lenguajes orientados a objetos y del entorno de programación.

Tareas:

- A) Identificar en los proyectos de la empresa el uso de **herencia**, diferenciando entre superclases y subclases y analizando su implementación.
- B) Crear nuevas clases que extiendan funcionalidades mediante **herencia**, asegurando la reutilización de código.
- C) Aplicar modificadores como **final (Java)** para bloquear o forzar la herencia según corresponda.
- D) Implementar métodos sobrescritos (**override**) para modificar el comportamiento de métodos heredados.
- E) Analizar el impacto de los **constructores en la herencia**, asegurando la correcta inicialización de objetos en la jerarquía de clases.
- F) Aplicar los principios de encapsulamiento utilizando modificadores de acceso (**private, protected, public**) para gestionar la visibilidad de atributos y métodos.
- G) Documentar el código utilizando herramientas como **JavaDoc**, o el estándar definido en la empresa.
- H) Escribir comentarios claros y estructurados en el código para facilitar su comprensión y mantenimiento por parte del equipo de desarrollo.

Enunciado:

Parte 1 – Cuestionario Teórico (Tarea A)

El cuestionario consistente en 5 preguntas de tipo test se realizará en la plataforma educativa.

Parte 2: Desarrollo práctico (Tareas B a H)

Objetivo del ejercicio:

Crear una **aplicación básica para gestionar reservas en alojamientos rurales**, incluyendo distintos tipos de reserva, cálculo de precios personalizados y seguridad en la gestión de datos. Cada parte del ejercicio trabajará uno de los apartados del RA7 (se indicará la letra correspondiente).

Enunciado

1. (B) Herencia y extensión de clases

Crear una clase base `Reserva` con atributos como `cliente`, `fechaEntrada`, `fechaSalida` y un método `calcularNoches()`.

Después, crea una subclase `ReservaPremium` que añada el atributo

`tarifaPremium` y un nuevo método `calcularPrecioTotal()` que multiplica las noches por una tarifa base y añade el extra premium.

2. (C) Restricción de herencia con `Object.freeze()`

Imagina que la clase `ReservaPremium` no debe ser extendida. Simula el comportamiento del modificador `final` de Java usando `Object.freeze()` y añade un comentario explicando por qué es útil en este caso.

3. (D) Sobrescritura de métodos (`override`)

Crea una subclase `ReservaConDesayuno` que también herede de `Reserva`, y sobrescriba el método `calcularPrecioTotal()` para sumar una tarifa diaria de desayuno a la base. Usa `super.calcularNoches()` para mantener el cálculo reutilizable.

4. (E) Constructores en jerarquías de clases

Asegúrate de que cada subclase llame correctamente al constructor de la clase padre usando `super(...)` y documenta con comentarios cómo se está inicializando cada propiedad.

5. (F) Encapsulamiento con atributos privados

Refactoriza `Reserva` usando campos privados (`#cliente`, `#fechaEntrada`, etc.) y crea getters/setters seguros para acceder y modificar esos datos. Explica en un comentario por qué es importante proteger la información.

6. (G) Documentación con JSDoc

Añade documentación usando comentarios JSDoc para cada clase y método. Por ejemplo:

```
/**
 * Calcula el número de noches entre dos fechas.
 * @returns {number} Número de noches.
 */
calcularNoches() { ... }
```

7. (H) Comentarios estructurados

Añade comentarios claros y ordenados en cada sección del código explicando la lógica aplicada, la razón detrás de las decisiones tomadas, y cómo se conecta con los principios de POO.

Estructura recomendada del proyecto

mi-proyecto/

└─ index.html # Página principal

└─ main.js # Archivo principal con la lógica del programa

└─ utils.js # Funciones de utilidad (opcional)
└─ images # Carpeta para las imágenes
└─ README.md # Instrucciones y explicación de tu solución

Plantilla Base de Código

```
// =====  
// DESARROLLO PRÁCTICO  
// =====  
  
// (B) Clase base y subclases mediante herencia  
  
/**  
 * Clase base para gestionar reservas estándar.  
 */  
class Reserva {  
  #cliente;  
  #fechaEntrada;  
  #fechaSalida;  
  
  /**  
   * @param {string} cliente  
   * @param {Date} fechaEntrada  
   * @param {Date} fechaSalida  
   */  
  constructor(cliente, fechaEntrada, fechaSalida) {  
    this.#cliente = cliente;  
    this.#fechaEntrada = fechaEntrada;  
    this.#fechaSalida = fechaSalida;  
  }  
  
  get cliente() {  
    return this.#cliente;  
  }  
  
  set cliente(nuevoCliente) {  
    this.#cliente = nuevoCliente;  
  }  
  
  /**  
   * Calcula el número de noches entre dos fechas.  
   * @returns {number}  
   */  
  calcularNoches() {  
    const msPorNoche = 1000 * 60 * 60 * 24;  
    return Math.round((this.#fechaSalida - this.#fechaEntrada) / msPorNoche);  
  }  
}  
  
/**  
 * Clase que representa una reserva con tarifa premium.  
 * Hereda de Reserva.  
 */  
class ReservaPremium extends Reserva {  
  constructor(cliente, fechaEntrada, fechaSalida, tarifaPremium) {  
    super(cliente, fechaEntrada, fechaSalida);  
    this.tarifaPremium = tarifaPremium;  
  }  
}
```

```

/**
 * Calcula el precio total con tarifa premium.
 * @returns {number}
 */
calcularPrecioTotal() {
  return this.calcularNoches() * this.tarifaPremium;
}
}

// (C) Bloquear la herencia de ReservaPremium
Object.freeze(ReservaPremium);
// Nota: No se puede extender esta clase. Simula la palabra clave "final".

// (D) Sobrescribir método en subclase

class ReservaConDesayuno extends Reserva {
  constructor(cliente, fechaEntrada, fechaSalida, tarifaBase, tarifaDesayuno) {
    super(cliente, fechaEntrada, fechaSalida);
    this.tarifaBase = tarifaBase;
    this.tarifaDesayuno = tarifaDesayuno;
  }

  /**
   * Sobrescribe el método para añadir desayuno al precio total.
   */
  calcularPrecioTotal() {
    const noches = super.calcularNoches();
    return noches * (this.tarifaBase + this.tarifaDesayuno);
  }
}

// (E) Constructores correctamente encadenados
// (ya incluido en los constructores anteriores con super())

// (F) Uso de atributos privados y getters/setters
// (ya aplicado en la clase Reserva con atributos privados y métodos de acceso)

// (G) JSDoc incluido en métodos y clases anteriores

// (H) Comentarios explicativos a lo largo del código

// =====
// PRUEBAS
// =====

// Aquí puedes instanciar tus clases para hacer pruebas

// const miReserva = new Reserva(...);
// console.log(miReserva.calcularNoches());

```

Ideas para la interfaz del programa

A continuación, tienes varias ideas para crear las pantallas de tu aplicación. No es necesario que sean como las propuestas, pero pueden servirte como orientación.

Pantalla Principal:

Reservas

| Cliente | Fecha de Entrada | Fecha de Salida | Tarifa |
|-------------|------------------|-----------------|--------------|
| Juan Pérez | 2024-05-10 | 2024-05-15 | |
| María Gómez | 2024-06-01 | 2024-06-07 | |
| Juan Pérez | 2024-07-20 | 2024-07-25 | \$150 |
| Ana López | 2024-08-05 | 2024-08-10 | Con desayuno |

Huéspedes

| Nombre | Correo Electrónico |
|-------------|-----------------------|
| Juan Pérez | juan.perez@email.com |
| María Gómez | maria.gomez@email.com |
| Ana López | ana.lopez@email.com |

Cabañas

| Número | Capacidad |
|--------|-----------|
| 101 | 4 |
| 102 | 3 |
| 103 | 2 |

Pantalla de gestión de reservas:

Asignar reserva

Huésped:

Juan Pérez

Cabaña:

Cabaña 1

Fecha de entrada:

2024-05-10

Fecha de salida:

2024-05-15

Cancelar

Asignar

Pantalla de gestión de huéspedes:

Huéspedes

Nuevo

Editar

Eliminar

| Nombre | Correo Electrónico | Teléfono |
|----------------|----------------------------|----------|
| Alice Martínez | alice.martinez@example.com | 555-1234 |
| Bob Sánchez | bob.sanchez@example.com | 555-5678 |
| Carlos Pérez | carlos.perez@example.com | 555-8765 |
| Diana Gómez | diana.gomez@example.com | 555-4321 |

Pantalla de gestión de cabañas:

Cabañas

| Número | Capacidad |
|--------|-----------|
| 1 | 4 |
| 2 | 6 |
| | |

Nueva

Editar

Eliminar

Entrega esperada:

Tu entrega debe incluir:

- Las respuestas del cuestionario.
- El código completo en un único archivo JS o en módulos separados.
- La documentación JSDoc incluida dentro del código.
- Los comentarios explicativos alineados con la lógica del ejercicio.

Rúbrica de Evaluación

| Criterio | Tareas vinculadas | Excelente (4) | Bien (3) | Suficiente (2) | Insuficiente (1) |
|---|-------------------|--|---|--|---|
| Identificación de estructuras de herencia | A | Identifica correctamente superclases y subclases en el contexto de JavaScript, explicando su uso y relación. | Identifica superclases y subclases pero con una explicación parcial. | Reconoce algunos elementos de herencia sin justificar su utilidad. | No identifica ni comprende la estructura de herencia. |
| Creación de nuevas clases mediante herencia | B | Crea subclases correctamente con propiedades y métodos heredados y extendidos; aplica reutilización eficaz del código. | Crea subclases con herencia funcional pero sin optimización o extensión completa. | Crea subclases con errores menores o poco funcionales. | No consigue crear subclases o comete errores graves. |
| Aplicación de modificadores para herencia | C | Aplica correctamente restricciones de herencia (ej. <code>Object.freeze</code>) y explica su propósito con claridad. | Aplica restricciones parcialmente o sin justificar adecuadamente. | Usa restricciones incorrectamente o sin relación con el contexto. | No utiliza ningún mecanismo para controlar la herencia. |

| | | | | | |
|--|-------|---|--|---|---|
| Sobrescritura de métodos heredados | D | Implementa métodos sobrescritos correctamente, usando <code>super</code> y adaptando funcionalidad según el nuevo contexto. | Sobrescribe métodos pero con lógica poco ajustada o sin usar <code>super</code> correctamente. | Intenta sobrescribir pero sin lograr una modificación efectiva. | No implementa sobrescritura de métodos. |
| Uso de constructores y encadenamiento o con <code>super()</code> | E | Implementa constructores en jerarquía de clases con uso correcto de <code>super()</code> y buena inicialización. | Usa <code>super()</code> pero con errores menores o poca claridad en la construcción de objetos. | Usa constructores sin <code>super()</code> o con inicialización incompleta. | No consigue encadenar constructores correctamente. |
| Encapsulamiento o con modificadores de acceso | F | Utiliza <code>#</code> , getters y setters correctamente para proteger y exponer atributos según convenga. | Usa encapsulamiento o con fallos menores o uso incompleto. | Usa solo parte del sistema de acceso o de forma poco segura. | No aplica ningún tipo de encapsulamiento. |
| Documentación con Javadoc o JSDoc | G | Utiliza JSDoc de forma precisa, completa y clara en clases, métodos y atributos. | Documenta parcialmente o con errores menores en formato o claridad. | Documenta con estilo no estandarizado o poco útil. | No documenta o la documentación es irrelevante. |
| Comentarios estructurados en el código | H | Los comentarios son útiles, están bien ubicados y ayudan a entender el funcionamiento del código. | Los comentarios son adecuados pero algo escasos o poco detallados. | Comenta el código de forma mínima o con poca claridad. | No incluye comentarios o estos son irrelevantes. |
| Prueba y funcionamiento del programa | Todas | El programa funciona completamente, produce resultados correctos y cubre todos los casos planteados. | El programa funciona con fallos menores o no cubre todos los casos. | El programa funciona parcialmente o de forma inconsistente. | El programa no funciona o tiene errores críticos. |
| Calidad del código y estilo | Todas | El código es legible, bien estructurado, sigue buenas prácticas y utiliza nombres descriptivos. | El estilo es mayormente correcto pero con detalles mejorables. | El código es funcional pero difícil de leer o inconsistente. | El código es desordenado o no sigue ninguna convención. |

Escala de puntuación:

- **40 – 35 puntos → Excelente**

Demuestra un dominio sólido de los principios avanzados de la POO en JavaScript, con un código bien estructurado, funcional y claramente documentado.

- **34 – 28 puntos → Bien**

Buen entendimiento de los conceptos, con una implementación correcta y

completa en la mayoría de los apartados. Puede haber leves fallos o áreas mejorables.

- **27 – 20 puntos → Suficiente**

Entiende los fundamentos pero comete errores o presenta carencias en varios criterios. El código funciona parcialmente o es poco claro.

- **19 o menos → Insuficiente**

No demuestra comprensión suficiente de los conceptos. La solución es incompleta, incorrecta o inoperativa.