# RELACIÓN EJERCICIOS POO y Ejercicio Resumen



## Ejercicio 1: Sistema de Gestión de Tareas

**Enunciado detallado:**

- Diseña una estructura para manejar tareas en una lista de productividad. Cada tarea debe tener las siguientes propiedades y métodos:
  - **Propiedades**:
    - `id` (un identificador único generado automáticamente, por ejemplo, con un contador estático).
    - `nombre` (nombre de la tarea, tipo `string`).
    - `completada` (booleano que indica si está completada o no).

    - **Métodos**:
        - `toggleCompletada()`: alterna entre completada y no completada.
- Usa las siguientes implementaciones:
    1. **Función Fábrica**: Crea un objeto de tarea con las propiedades y métodos necesarios. Usa closures si es necesario para encapsular información privada.
    2. **Función Constructora**: Usa prototipos para implementar el método `toggleCompletada()`.
    3. **Clase**: Implementa la misma funcionalidad usando sintaxis moderna, incluyendo propiedades privadas si son necesarias.
- **Adicional**: Implementa una función o clase de "Gestor de Tareas" que permita:
    - Agregar tareas.
    - Eliminar tareas por `id`.
    - Listar tareas pendientes y completadas.

## Ejercicio 2: Sistema de Gestión de Productos

**Enunciado detallado:**

- Diseña un sistema para gestionar productos en un inventario. Cada producto debe tener:
    - **Propiedades**:
        - `nombre` (tipo `string`).
        - `precio` (número que indica el precio en euros).
        - `stock` (número que indica la cantidad en inventario).
    - **Métodos**:
        - `actualizarStock(cantidad)`: incrementa o decrementa el stock según el valor de `cantidad`.
- Extiende el sistema para soportar diferentes tipos de productos:
    - **ProductoFísico**:
        - Incluye `dimensiones` como un objeto `{alto, ancho, profundo}`.
    - **ProductoDigital**:
        - No tiene stock (deberías manejar este caso en el método `actualizarStock`).
- Usa las siguientes implementaciones:
    1. **Función Fábrica**: Devuelve un objeto para cada tipo de producto.
    2. **Función Constructora**: Usa prototipos y herencia prototípica para especializar productos.

3. **Clase**: Usa herencia con `extends` para las subclases `ProductoFísico` y `ProductoDigital`.

## Ejercicio 3: Sistema de Empleados y Nóminas

**Enunciado detallado:**

- Diseña un sistema para gestionar empleados. Cada empleado debe tener:
  - **Propiedades**:
    - `nombre` (tipo `string`).
    - `edad` (número).
    - `puesto` (tipo `string`).
  - **Métodos**:
    - `calcularSalario()`: Retorna el salario básico mensual según el puesto.
- Extiende el sistema con una subclase de empleado:
  - **EmpleadoFreelance**:
    - Incluye una tarifa por hora (`tarifaHora`) y un método `calcularSalario(horas)` que calcula el salario según las horas trabajadas.
- Usa las siguientes implementaciones:
  1. **Función Fábrica**: Usa closures para encapsular detalles como la tarifa por hora.
  2. **Función Constructora**: Implementa la herencia usando prototipos.
  3. **Clase**: Usa herencia con `extends` y sobrescribe `calcularSalario()` en `EmpleadoFreelance`.

## Ejercicio 4: Sistema de Vehículos y Flotas

**Enunciado detallado:**

- Diseña un sistema para gestionar vehículos en una flota de transporte. Cada vehículo debe tener:
  - **Propiedades**:
    - `marca` (tipo `string`).
    - `modelo` (tipo `string`).
    - `kilometraje` (número que indica los kilómetros recorridos).
  - **Métodos**:
    - `registrarViaje(kms)`: Incrementa el kilometraje en la cantidad especificada.

- Extiende el sistema para soportar los siguientes tipos de vehículos:
  - **Camión**:
    - Añade una propiedad `capacidadCarga` (número en toneladas).
    - Sobrescribe `registrarViaje()` para incluir un mensaje sobre el peso transportado.
  - **Automóvil**:
    - Añade una propiedad `capacidadPasajeros` (número de pasajeros).
- Usa las siguientes implementaciones:
  1. **Función Fábrica**: Crea objetos para `Camión` y `Automóvil` con comportamiento especializado.
  2. **Función Constructora**: Implementa herencia prototípica para las subclases.
  3. **Clase**: Usa herencia con `extends` y sobrescribe métodos en las subclases.

## Ejercicio 5: Sistema de Gestión de Usuarios

**Enunciado detallado:**

- Diseña un sistema para gestionar usuarios en una aplicación. Cada usuario debe tener:
  - **Propiedades**:
    - `nombre` (tipo `string`).
    - `email` (tipo `string`, debe validarse al asignarlo).
    - `rol` (tipo `string`, puede ser "admin" o "usuario").
  - **Métodos**:
    - `actualizarRol(nuevoRol)`: Cambia el rol del usuario si es válido.
  - Usa encapsulación para proteger el acceso a `email` y solo permitir su modificación mediante un setter.
- Extiende el sistema para incluir un tipo especial de usuario:
  - **UsuarioPremium**:
    - Añade una propiedad `suscripciónActiva` (booleano).
    - Métodos adicionales: `activarSuscripción()` y `cancelarSuscripción()`.
- Usa las siguientes implementaciones:
  1. **Función Fábrica**: Usa closures para encapsular la validación de `email`.
  2. **Función Constructora**: Implementa la herencia prototípica para los usuarios premium.
  3. **Clase**: Usa encapsulación con propiedades privadas (`#email`) y herencia con `extends`.

## Ejercicio 6: Sistema de Reserva de Habitaciones en un Hotel

**Enunciado detallado:**

- Diseña un sistema para gestionar las reservas de habitaciones en un hotel. Cada habitación y reserva debe tener las siguientes propiedades y métodos:

### Habitación:

- **Propiedades**:
  - `numero` (número de la habitación, tipo `number`).
  - `tipo` (tipo de habitación: "individual", "doble", "suite", tipo `string`).
  - `precio` (precio por noche, tipo `number`).
  - `reservada` (booleano que indica si la habitación está reservada).
- **Métodos**:
  - `reservar()`: Marca la habitación como reservada.
  - `liberar()`: Marca la habitación como disponible.

### Reserva:

- **Propiedades**:
  - `id` (identificador único, generado automáticamente).
  - `habitacion` (la habitación reservada, referencia a un objeto de tipo Habitación).
  - `cliente` (nombre del cliente, tipo `string`).
  - `dias` (número de días de la reserva, tipo `number`).
- **Métodos**:
  - `calcularTotal()`: Calcula el costo total de la reserva (`precio * dias`).

### Gestor de Reservas:

- Implementa una clase o función para gestionar todas las reservas:
  - Métodos:
    - `crearReserva(cliente, numeroHabitacion, dias)`: Verifica si la habitación está disponible, la reserva y añade una nueva reserva a la lista.
    - `cancelarReserva(id)`: Busca una reserva por `id` y libera la habitación asociada.

- `listarReservas()`: Devuelve un listado de todas las reservas activas.

### Implementación:

- Usa las siguientes aproximaciones:
  1. **Función Fábrica**: Crea objetos de tipo Habitación, Reserva y Gestor de Reservas con métodos encapsulados.
  2. **Función Constructora**: Usa prototipos para implementar métodos en Habitación y Reserva.
  3. **Clase**: Implementa el sistema usando clases y propiedades privadas (`#reservada`).

### Requisitos Adicionales:

- Usa encapsulación para proteger el estado de las habitaciones (`reservada`) y asegúrate de que solo pueda modificarse mediante los métodos `reservar()` y `liberar()`.
- Usa métodos estáticos en la clase de Reservas para generar IDs únicos.

## Ejercicio 7: Gestión de Usuarios desde una API

**Descripción:**

- Crea una clase `Usuario` que represente un usuario con propiedades como `id`, `nombre`, `email` y `rol`.
- Usa la API JSONPlaceholder para obtener una lista de usuarios.
- Implementa un método estático `Usuario.fetchUsuarios()` para obtener datos desde la API y otro método estático `Usuario.cargarDesdeLocalStorage()` para cargar datos locales.
- Añade un método `info()` para mostrar detalles del usuario.

**Requisitos adicionales:**

- Implementa manejo de errores con `try/catch` y reconexión automática.
- Guarda los datos en `LocalStorage` para usarlos si la API no está disponible.

## Ejercicio 8: Sistema de Reservas de Películas

**Descripción:**

- Crea una clase `Pelicula` con propiedades como `titulo`, `año`, `director` y métodos como `info()` para mostrar detalles.
- Usa la API OMDb API para buscar películas por título.
- Implementa una clase `ReservaPelicula` para gestionar las reservas de películas con propiedades como `cliente` y `diasReservados`.

**Requisitos adicionales:**

- Implementa un método estático `Pelicula.buscarPeliculas(titulo)` que haga un `fetch` a la API.
- Incluye manejo de errores y reintentos en el `fetch`.
- Guarda las películas obtenidas en `LocalStorage`.

## Ejercicio 9: Catálogo de Productos desde una API

**Descripción:**

- Define una clase `Producto` con propiedades como `id`, `nombre`, `precio` y `categoria`.
- Usa la API Fake Store API para obtener un catálogo de productos.
- Implementa una clase `Carrito` para gestionar la compra de productos.

**Requisitos adicionales:**

- Implementa métodos estáticos en `Producto` para cargar datos desde la API y `LocalStorage`.
- Implementa manejo de errores en el `fetch` y reconexiones automáticas.
- Añade métodos en la clase `Carrito` para agregar productos, eliminarlos, y calcular el total.

## Ejercicio 10: Gestión de Publicaciones de un Blog

**Descripción:**

- Crea una clase `Publicacion` con propiedades como `id`, `titulo`, `contenido`, y `autor`.

- Usa la API JSONPlaceholder para obtener una lista de publicaciones.
- Implementa una clase `Blog` que gestione las publicaciones.

**Requisitos adicionales:**

- Implementa un método estático `Publicacion.fetchPublicaciones()` para obtener las publicaciones desde la API.
- Implementa métodos en `Blog` para filtrar publicaciones por autor y buscar por palabra clave.
- Maneja errores en el `fetch`, reconexiones automáticas y guarda los datos en `LocalStorage`.

# EJERCICIO RESUMEN

Diseña un sistema para gestionar **servicios en la nube** (como almacenamiento, cómputo y bases de datos). Implementa los siguientes requisitos:

1. **Clase Base** o equivalente:
   - Propiedades públicas: `nombre`, `costoMensual`.
   - Propiedades privadas: `#id` (identificador único).
   - Métodos:
     - Método estático para generar IDs únicos.
     - Método público `detalles()` para mostrar información básica.
2. **Subclases o equivalentes** para diferentes tipos de servicios:
   - `Almacenamiento` (propiedad adicional: `capacidad` en GB).
   - `Computo` (propiedad adicional: `nucleos` de CPU).
   - `BaseDeDatos` (propiedad adicional: `tamaño` en GB).
3. **Sobrescritura**:
   - Cada tipo de servicio debe sobrescribir el método `detalles()` para mostrar información específica.
4. Usa **getters y setters** para controlar el costo mensual, con validaciones.

## Tabla Comparativa: Implementación del Sistema

| Concepto | Clases | Funciones Constructoras | Funciones Fábrica |
|---|---|---|---|
| **Herencia** | Usamos `class Almacenamiento extends Servicio` para heredar. | Usamos `Object.create()` y `call()` para extender. | Creamos un objeto base y extendemos añadiendo propiedades y métodos. |
| **Polimorfismo** | Sobrescribimos `detalles()` en cada subclase. | Redefinimos `detalles()` en prototipos específicos. | Sobrescribimos el método `detalles()` en el objeto retornado por la fábrica. |

| Concepto | Clases | Funciones Constructoras | Funciones Fábrica |
|---|---|---|---|
| **Propiedades Públicas** | Declaradas directamente en el constructor ( `this.propiedad` ). | Definidas en el constructor ( `this.propiedad` ). | Retornadas como parte del objeto ( `return { propiedad }` ). |
| **Propiedades Privadas** | Usamos `#propiedad` para encapsular ( `#id` ). | Simulamos con closures dentro del constructor. | Usamos variables locales dentro de la función fábrica. |
| **Métodos Públicos** | Declarados en la clase ( `detalles()` ). | Añadidos al prototipo para compartir entre instancias. | Agregados directamente al objeto retornado ( `return { detalles() }` ). |
| **Métodos Estáticos** | Declarados con `static` . | Añadidos directamente al constructor ( `Servicio.generarId` ). | Agregados como propiedades de la función fábrica ( `crearServicio.generarId` ). |
| **Getters y Setters** | Usamos `get` y `set` para controlar `costoMensual` . | Usamos `Object.defineProperty()` o definimos manualmente. | Añadidos directamente al objeto retornado. |
| **Encapsulación** | Propiedades privadas con `#` o closures. | Propiedades privadas simuladas con closures. | Variables internas dentro de la fábrica. |
| **Sobreescritura en Hijos** | Cada subclase redefine el método `detalles()` . | Se redefine `detalles()` en prototipos específicos. | Se sobrescribe `detalles()` en el objeto retornado. |

## Ejemplo de Implementación

Entendido. A continuación, voy a agregar **comentarios detallados** en cada uno de los ejemplos de implementación (clases, funciones constructoras, y funciones fábrica) para explicar cómo se están aplicando los conceptos clave como **herencia**, **polimorfismo**, **métodos estáticos**, **encapsulación**, y más. Cada concepto será señalado claramente en el código.

### Implementación con Clases

```
// Clase base: Servicio
class Servicio {
  // Variable de clase para generar IDs únicos (Variable de Clase)
  static #contadorId = 0;
```

```
  constructor(nombre, costoMensual) {
    // Propiedad privada para encapsulación (Encapsulación)
     this.#id = ++Servicio.#contadorId; // Cada instancia tiene un ID
único
    this.nombre = nombre; // Propiedad pública
    this.costoMensual = costoMensual; // Usamos el setter
  }

  // Propiedad privada
  #id;

  // Método estático para generar IDs únicos (Método Estático)
  static generarId() {
    return ++this.#contadorId;
  }

  // Getter para obtener el costo mensual (Getters y Setters)
  get costoMensual() {
    return this._costoMensual;
  }

  // Setter para modificar el costo mensual con validación
  set costoMensual(valor) {
    if (valor < 0) {
      throw new Error("El costo no puede ser negativo.");
    }
    this._costoMensual = valor;
  }

   // Método público para mostrar información básica del servicio
(Método Público)
  detalles() {
    console.log(`${this.nombre}: $${this.costoMensual}/mes`);
  }
}

// Subclase: Almacenamiento (Herencia)
class Almacenamiento extends Servicio {
  constructor(nombre, costoMensual, capacidad) {
    super(nombre, costoMensual); // Llama al constructor de la clase
base
      this.capacidad = capacidad; // Propiedad específica de la
subclase
  }

  // Sobreescritura del método detalles (Polimorfismo)
  detalles() {
    console.log(
```

```
            `${this.nombre}: $${this.costoMensual}/mes - Capacidad:
${this.capacidad} GB`
    );
  }
}

// Subclase: Computo (Herencia)
class Computo extends Servicio {
  constructor(nombre, costoMensual, nucleos) {
    super(nombre, costoMensual);
    this.nucleos = nucleos; // Propiedad específica de la subclase
  }

  // Sobreescritura del método detalles (Polimorfismo)
  detalles() {
    console.log(
            `${this.nombre}: $${this.costoMensual}/mes - Núcleos:
${this.nucleos}`
    );
  }
}

// Subclase: BaseDeDatos (Herencia)
class BaseDeDatos extends Servicio {
  constructor(nombre, costoMensual, tamanio) {
    super(nombre, costoMensual);
    this.tamanio = tamanio; // Propiedad específica de la subclase
  }

  // Sobreescritura del método detalles (Polimorfismo)
  detalles() {
    console.log(
            `${this.nombre}: $${this.costoMensual}/mes - Tamaño:
${this.tamanio} GB`
    );
  }
}

// *** Ejemplo de uso ***

// Crear instancias de servicios con diferentes subclases
const almacenamiento = new Almacenamiento("Google Drive", 10, 200);
// Almacenamiento
const computo = new Computo("AWS EC2", 50, 8); // Computo
const baseDeDatos = new BaseDeDatos("MongoDB Atlas", 25, 500); //
BaseDeDatos

// Mostrar detalles de cada servicio (Polimorfismo en acción)
almacenamiento.detalles(); // Google Drive: $10/mes - Capacidad: 200
```

```
GB
computo.detalles(); // AWS EC2: $50/mes - Núcleos: 8
baseDeDatos.detalles(); // MongoDB Atlas: $25/mes - Tamaño: 500 GB

// Usar getters y setters para validar y modificar el costo mensual
(Getters y Setters)
console.log(`Costo inicial de Google Drive:
$${almacenamiento.costoMensual}`);
almacenamiento.costoMensual = -5; // Error: El costo no puede ser
negativo
almacenamiento.costoMensual = 15; // Actualización válida
console.log(`Nuevo costo de Google Drive:
$${almacenamiento.costoMensual}`);

// Generar IDs únicos con el método estático (Método Estático)
console.log(`ID único generado: ${Servicio.generarId()}`);
```

### Implementación con Funciones Constructoras

```
// Constructor base: Servicio
function Servicio(nombre, costoMensual) {
  // Variable privada (Encapsulación)
  let id = Servicio.generarId(); // Cada instancia tiene un ID único

  this.nombre = nombre; // Propiedad pública
    this.costoMensual = costoMensual; // Usaremos un setter más
adelante

  // Método público para mostrar información básica del servicio
  this.detalles = function () {
    console.log(`${this.nombre}: $${this.costoMensual}/mes`);
  };

  // Getter para obtener el costo mensual (Getters y Setters)
  Object.defineProperty(this, "costoMensual", {
    get: function () {
      return costoMensual;
    },
    set: function (valor) {
      if (valor < 0) {
        throw new Error("El costo no puede ser negativo.");
      }
      costoMensual = valor;
```

```
    },
  });
}

// Método estático (Método Estático)
Servicio.generarId = (function () {
  let contador = 0; // Variable de clase (Encapsulación)
  return function () {
    return ++contador;
  };
})();

// Constructor Almacenamiento (Herencia)
function Almacenamiento(nombre, costoMensual, capacidad) {
  Servicio.call(this, nombre, costoMensual); // Herencia
  this.capacidad = capacidad; // Propiedad específica de la subclase

  // Sobreescritura del método detalles (Polimorfismo)
  this.detalles = function () {
    console.log(
          `${this.nombre}:  $${this.costoMensual}/mes  -  Capacidad:
${this.capacidad} GB`
    );
  };
}

// Crear instancias de los servicios
const almacenamiento = new Almacenamiento("Dropbox", 15, 500);
almacenamiento.detalles(); // Dropbox: $15/mes - Capacidad: 500 GB
```

### Implementación con Funciones Fábrica

```
// Fábrica base: Servicio
function crearServicio(nombre, costoMensual) {
  // Variable privada para encapsulación (Encapsulación)
  let id = (function () {
    let contador = 0;
    return function () {
      return ++contador;
    };
  })();

  return {
```

```
    nombre, // Propiedad pública
    costoMensual,

    // Getter y setter para costoMensual
    get costoMensual() {
      return costoMensual;
    },
    set costoMensual(valor) {
        if (valor < 0) throw new Error("El costo no puede ser
negativo.");
      costoMensual = valor;
    },

    // Método público para mostrar información básica
    detalles() {
      console.log(`${this.nombre}: $${this.costoMensual}/mes`);
    },

    // Método estático (Método Estático)
    generarId: id,
  };
}

// Fábrica específica: Almacenamiento (Herencia)
function crearAlmacenamiento(nombre, costoMensual, capacidad) {
  const servicio = crearServicio(nombre, costoMensual);

    // Extender el objeto base con propiedades específicas
(Polimorfismo)
  return {
    ...servicio,
    capacidad,
    detalles() {
      console.log(
        `${this.nombre}: $${this.costoMensual}/mes - Capacidad:
${this.capacidad} GB`
      );
    },
  };
}

// Crear instancia del servicio
const almacenamiento = crearAlmacenamiento("OneDrive", 12, 1000);
almacenamiento.detalles(); // OneDrive: $12/mes - Capacidad: 1000 GB
```