IES CHAN DO MONTE
C.S. de Desarrollo de Aplicaciones Multiplataforma
Modulo Acceso a datos

ACTIVIDAD 3: SERIALIZAR OBJETOS Y ACCESO ALEATORIO

Esta práctica tiene como finalidad aplicar de forma integrada los conocimientos sobre ficheros binarios secuenciales y aleatorios, colecciones en Java y estructuras de datos Mapas y Listas). Además, se refuerzan las buenas prácticas de programación, la organización por capas y la coherencia de los datos almacenados.

Objetivos Generales

- Consolidar el manejo de ficheros binarios secuenciales y aleatorios.
- Comprender la interacción entre diferentes tipos de almacenamiento (equipos y corredores).
- Reforzar la aplicación del modelo por capas en una aplicación Java.
- Favorecer el uso de estructuras de datos dinámicas y genéricas (Map, Set,ArrayList).
- Potenciar la capacidad de diseño modular y reutilizable del código.
- Asegurar la coherencia entre los datos de entidades relacionadas (corredores « equipos).

Objetivos Específicos

- Diseñar clases modelo coherentes para las entidades Equipo, Corredor y Patrocinador.
- Implementar un fichero binario aleatorio con registros de tamaño fijo para almacenar equipos.
- Implementar un fichero binario secuencial para almacenar los corredores.
- Controlar la integridad referencial entre los corredores y los equipos mediante validaciones.
- Desarrollar métodos que integren información de ambos ficheros para generar informes combinados.
- Emplear mapas (Map) para agrupar, clasificar y analizar los datos de forma estructurada.
- Utilizar mapas anidados y colecciones como valores para representar relaciones complejas.
- Calcular totales y rankings combinando datos de diferentes fuentes de información.
- Exportar informes a ficheros de texto respetando un formato legible y organizado.

Al finalizar la práctica, el alumnado será capaz de manejar de forma autónoma diferentes tipos de ficheros binarios, garantizar la coherencia entre entidades relacionadas y utilizar colecciones para organizar y analizar la información. Además, comprenderá la importancia deestructurar correctamente una aplicación Java en capas bien definidas.

Gestión de Corredores :

Objetivo: Desarrollar una aplicación para almacenar y gestionar la información de los corredores que participan en las competiciones en España. Esta información debe ser almacenada en un fichero binario secuencial llamado **corredores.dat**, y debe incluir datos personales, la relación con el equipo y un historial de puntuaciones por año. Los corredores se clasifican en dos tipos: Fondistas y Velocistas, los cuales tienen atributos específicos. A continuación, se detallan los requisitos de la información a almacenar:

1. Información a almacenar por corredor:

Cada corredor tendrá los siguientes atributos:

- **dorsal** (int): Identificador único del corredor dentro del sistema. Empezará en 1 y se irá incrementando automáticamente.
- **nombre** (String): Nombre completo del corredor.
- **fechaNacimiento** (LocalDate): Fecha de nacimiento del corredor.
- **equipo** (int): Identificador o código del equipo al que pertenece el corredor. En esta fase, se almacenará solo la referencia (clave) al equipo. Los datos completos del equipo se gestionarán en otro fichero **equipos.dat** (que será accesible de forma aleatoria, pero en esta fase solo se guardará el identificador del equipo). Los equipos estarán enumerados consecutivamente, comenzando desde 1, y habrá un total de 4 equipos.

2. Información de puntuaciones del corredor:

Los corredores tienen un historial de puntuaciones por año. La información de cada puntuación será:

- **anio** (int): Año de la puntuación.
- **puntos** (float): Valor numérico obtenido en ese año.

El historial de puntuaciones de un corredor se guardará en un **array** de objetos de la clase **Puntuacion**, que debe estar ordenado por el año.

3. Tipos concretos de corredor:

Los corredores se clasificarán en dos tipos diferentes, y deben implementarse como clases derivadas de una clase base **Corredor**:

a. Fondista:

- Un **fondista** es un corredor especializado en pruebas de larga distancia, como maratones.
- Atributo adicional: **distanciaMax** (float), que representa la distancia máxima que el corredor puede recorrer (en kilómetros). Ejemplo: 10.0 (10 km), 42.195 (maratón).

b. Velocista:

- Un **velocista** es un corredor especializado en pruebas de corta distancia.
- Atributo adicional: **velocidadMedia** (float), que representa la velocidad media del corredor (en km/h). Ejemplo: 10.34 km/h.

4. Requisitos adicionales:

- La información de cada corredor debe ser almacenada en el fichero **corredores.dat** en formato binario secuencial.
- El sistema debe ser capaz de leer y escribir datos en este fichero.
- La relación de los corredores con los equipos se debe gestionar mediante la referencia al identificador de un equipo, pero los detalles completos del equipo se gestionarán en otro fichero **equipos.dat** (que será implementado más adelante).
- Cada tipo de corredor (Fondista o Velocista) debe ser almacenado en el fichero como una instancia concreta de la clase correspondiente (es decir, un objeto de la clase **Fondista** o **Velocista**).

5.- Estructura de la aplicación:

La aplicación debe estar estructurada en clases que deleguen responsabilidades de manera clara y modular, siguiendo el principio de **Responsabilidad Única**.

Cada clase se encargará de una sola responsabilidad bien definida. Evitar clases "todo-en-uno" que mezclen dominio, persistencia, formato y validación.

Capas y componentes (visión general)

- Capa modelo (dominio) — responsabilidad: representar los datos y reglas sencillas del dominio.
  - Contendrá las clases que representan un corredor y las estructuras de valor que éste usa (por ejemplo, una clase para una puntuación anual).
  - No realizará I/O ni gestionará persistencia.
- Capa persistencia (I/O) — responsabilidad: acceso secuencial al fichero corredores.dat..
  - Un componente dedicado a escribir objetos (apertura/serialización/escritura/ cierre).
  - Un componente dedicado a leer objetos (apertura/deserialización/recorrido y cierre).
  - Estos componentes no harán validación de negocio ni impresión por consola.
- Capa servicio (lógica coordinadora) — responsabilidad: orquestar operaciones de negocio y validar reglas antes de persistir
  - Creación de objetos a partir de datos de entrada.
  - Usa la capa persistencia para almacenar y recuperar datos.
- Capa aplicación (main) — responsabilidad: interacción con el usuario.
  - Esta clase actuará como el punto de entrada del programa.
  - Llama a los métodos de la Capa Servicios
  - No debe conocer los detalles de serialización ni manejar streams.
- Utilidades pequeñas (helpers) — responsabilidad: tareas concretas y atómicas, por ejemplo:

- **Validadores:** Clases encargadas de validar las entradas del usuario, como la validación de fechas, puntuaciones válidas, o la validez de un dorsal.
- **Conversores:** Utilidades para convertir objetos de un tipo a otro, por ejemplo, conversores entre `LocalDate` y `String` para mostrar fechas en formato legible.

6. Operaciones a realizar:

- Crear la jerarquía de clase para poder serializar objetos de los corredores.
- Crear los métodos necesarios para realizar las operaciones de persistencia y visualización:

Métodos a Implementar:

1. **Método para guardar un corredor en el fichero binario**: Este método se encarga de guardar un corredor (sea de tipo `Fondista` o `Velocista`) en el fichero `corredores.dat` de forma secuencial. Los corredores se almacenan de manera que el dorsal se incremente automáticamente.

   Pon ejemplos de llamada para introducir por lo menos 6 corredores. Ejemplos:

   Corredor corredor1 = new Velocista("Juan Pérez", LocalDate.of(2000, 5, 12), 1, 10.34f);

   Corredor corredor2 = new Fondista("Ana Gómez", LocalDate.of(1995, 3, 22), 2, 42.195f);

   Corredor corredor3 = new Velocista("Carlos Ruiz", LocalDate.of(2002, 11, 30), 3, 9.75f);

   Corredor corredor4 = new Fondista("María López", LocalDate.of(2000, 7, 15), 1, 21.097f);

   Corredor corredor5 = new Velocista("Pedro García", LocalDate.of(1995, 8, 5), 1, 11.20f);

   Corredor corredor6 = new Fondista("Laura Martínez", LocalDate.of(2002, 9, 10), 4, 35.00f);

2. **Método para listar todos los corredores**

   Este método lee todos los corredores almacenados en el fichero corredores.dat y muestra su información por consola. Usará la capa de persistencia para leer los objetos y luego los imprimirá.

3. **Método para mostrar la información de un corredor dado su dorsal**

   Este método permite consultar un corredor a partir de su dorsal. El sistema buscará el corredor en el fichero y si lo encuentra, lo mostrará. Gestionar los posibles errores.

4. **Método para eliminar un corredor por dorsal**

   Este método permite eliminar un corredor del fichero corredores.dat. Se necesitará utilizar un archivo auxiliar que contenga todos los corredores menos el borrado

5. **Método para introducir una nueva puntuación a un corredor:**

   Este método debe permitir agregar una nueva puntuación al historial de puntuaciones de un corredor ya existente en el fichero corredores.dat. Se tendrá que utilizar un archivo auxiliar

   Una vez encontrado el corredor, se debe verificar si el año de la nueva puntuación ya existe en el historial de puntuaciones.

   Si el año no existe, se debe agregar la nueva puntuación al array de puntuaciones del corredor de manera ordenada por año.

   Si el año ya está presente, se debe actualizar la puntuación de ese año.

Gestión de Equipos

Objetivo:
Vamos a añadir a la aplicación existente el código necesario para almacenar y gestionar la información de los **equipos** que participan en competiciones en España. Esta información se almacenará en un **fichero binario de acceso aleatorio** llamado **equipos.dat** y debe incluir datos básicos del equipo y un listado de patrocinadores.

---

1. Información a almacenar por equipo

Cada equipo tendrá los siguientes atributos:

- **idEquipo (int):** Identificador único del equipo. Se guardará en orden consecutivo y también permitirá calcular la posición del registro en el fichero aleatorio.

- **nombre (String): Nombre único** del equipo.
- **numPatrocinadores (int):** Número de patrocinadores que tiene el equipo, útil para la lectura del array de patrocinadores.
- **patrocinadores (Set <Patrocinador>):** Lista de objetos de la clase Patrocinador. Cada patrocinador tendrá:
  - **nombre (String):** Nombre del patrocinador (único dentro del equipo).
  - **donacion (float):** Valor de la aportación del patrocinador.
  - **fechaInicio (LocalDate):** Fecha de inicio del patrocinio.
- **borrado (boolean):** Para implementar borrado lógico si se desea eliminar un equipo sin modificar el resto del fichero.

Notas:

- El número de patrocinadores es variable: un equipo puede tener 0, 1, 2… n patrocinadores.
- Cada registro de equipo tendrá un tamaño máximo definido (por ejemplo, 200 bytes) para permitir acceso aleatorio. Antes de escribir un registro, se debe comprobar que el tamaño total del registro no exceda este límite.
- El uso de **Set** garantiza **que no se repitan patrocinadores con el mismo nombre dentro de un mismo equipo**.
  - Para que funcione correctamente, la clase Patrocinador debe sobrescribir los métodos equals() y hashCode(), definiendo la unicidad por el nombre del patrocinador.
  - Al usar un Set en Java, no se lanza ninguna excepción al intentar añadir un elemento duplicado; simplemente no se añade.

```
// equals y hashCode basados en el nombre (para evitar duplicados por nombre)
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Patrocinador)) return false;
        Patrocinador that = (Patrocinador) o;
        return nombre.equalsIgnoreCase(that.nombre);
    }

    @Override
    public int hashCode() {
        return nombre.toLowerCase().hashCode();
    }
```

2.- Estructura de la Gestión de equipos

La aplicación ya cuenta con paquetes y clases para la gestión de corredores. La gestión de equipos se integrará **siguiendo la misma estructura y principio de responsabilidad única**, añadiendo las clases correspondientes a cada paquete según su función.

1. **En las clases (Capa modelo o dominio)** se añadirán**:**

Las clases Equipo y Patrocinadores

- Estas clases **no realizarán I/O ni gestionarán persistencia**.
- Se pueden añadir métodos auxiliares para calcular el tamaño en bytes de un registro .

---

2. Capa persistencia (I/O)

- **Responsabilidad:** Acceso aleatorio al fichero equipos.dat.
- **Se añade una clase para gestionar el acceso a ficheros aleatorios, apertura, cierre , lectura y escritura**
- **Notas:**
  - Estas clases **no realizarán validación de negocio ni impresión por consola**.
  - Se trabajará con **ficheros de acceso aleatorio** con registros de tamaño fijo =200 bytes por equipo.

---

3. Capa servicio (lógica de negocio)

- **Responsabilidad:** Operaciones de negocio y validar reglas antes de persistir.
- Funciones:
    - Crear un Equipo a partir de datos de entrada.
    - Consultar equipos por nombre o id.
    - Consultar todos los equipos.
    - Añadir, modificar o eliminar patrocinadores.
    - Modificar el nombre de un equipo
    - Gestionar borrado lógico de equipos.
    - Consultar equipos por nombre o id.
    - Usa la **capa persistencia** para almacenar y recuperar datos.
- **Notas:**
    - Aquí se asegura que no se violen reglas de negocio, por ejemplo:
        - No repetir nombres de equipos.
        - No exceder el tamaño máximo de registro en el fichero.
        - Mantener consistencia en el número de patrocinadores.

---

4. Capa aplicación (main)

- **Responsabilidad:** ejemplos de llamadas .
- Punto de entrada del programa para gestionar equipos.
- No conoce los detalles de la persistencia ni maneja streams.

5. Utilidades pequeñas (helpers)

- **Responsabilidad:** Tareas concretas y atómicas.

Métodos a Implementar – Gestión de Equipos

1. Método para guardar un equipo en el fichero binario aleatorio

✓ Este método se encarga de guardar un objeto Equipo en el fichero equipos.dat utilizando acceso aleatorio. Cada equipo se almacenará en una posición calculada a partir de su identificador (idEquipo). Es decir:

- El **primer equipo** tendrá idEquipo = 1 y se almacenará al inicio del fichero (posición 0).
- El **segundo equipo** tendrá idEquipo = 2 y se almacenará a partir de la posición 200, y así sucesivamente.

✓ El tamaño de cada registro será **200 bytes** y el fichero permitirá lectura y escritura en cualquier posición sin recorrerlo completo.

✓ El método debe:

- Validar que el nombre del equipo no esté repetido.
- Asignar automáticamente el idEquipo (autoincremental).
- Calcular el número de patrocinadores (numPatrocinadores) en función del tamaño de la lista.
- Escribir el objeto completo en el fichero usando EquipoRandom.

Ejemplo de llamadas para introducir al menos 5 equipos con sus patrocinadores:

```
Set<Patrocinador> lista1 = new HashSet<>();
lista1.add(new Patrocinador("Coca-Cola", 15000.0f, LocalDate.of(2023, 1, 15)));
lista1.add(new Patrocinador("Nike", 22000.0f, LocalDate.of(2024, 3, 10)));
Equipo equipo1 = new Equipo("Atlético Norte", lista1);
```

```
Set<Patrocinador> lista2 = new HashSet<>();
lista2.add(new Patrocinador("Adidas", 18000.0f, LocalDate.of(2022, 5, 1)));
lista2.add(new Patrocinador("Coca-Cola", 16000.0f, LocalDate.of(2021, 4, 4)));
lista2.add(new Patrocinador("Vodafone", 16500.0f, LocalDate.of(2021, 5, 5)));
Equipo equipo2 = new Equipo("Deportivo Sur", lista2);

Set<Patrocinador> lista3 = new HashSet<>();
lista3.add(new Patrocinador("Renfe", 12000.0f, LocalDate.of(2023, 6, 25)));
lista3.add(new Patrocinador("Mapfre", 9000.0f, LocalDate.of(2023, 9, 1)));
Equipo equipo3 = new Equipo("Coruña Team", lista3);

Set<Patrocinador> lista4 = new HashSet<>();
lista4.add(new Patrocinador("Repsol", 25000.0f, LocalDate.of(2022, 4, 10)));
Equipo equipo4 = new Equipo("Racing Galicia", lista4);

Set<Patrocinador> lista5 = new HashSet<>();
lista5.add(new Patrocinador("Mahou", 17000.0f, LocalDate.of(2023, 11, 5)));
Equipo equipo5 = new Equipo("Union Atlética", lista5);
```

---

2. Método para listar todos los equipos

- ✓ Este método recorre el fichero equipos.dat completo leyendo los registros activos (aquellos cuyo campo borrado sea false) y muestra por consola la información de cada equipo, incluyendo el listado de patrocinadores.
- ✓ Usará los métodos de lectura implementados en la capa de persistencia .

---

3. Método para mostrar la información de un equipo dado su identificador

- ✓ Este método permite consultar toda la información de un equipo a partir de su idEquipo.
- ✓ El sistema calculará la posición del registro en el fichero y lo leerá directamente.
- ✓ Si el registro está marcado como borrado o el ID no existe, mostrará un mensaje de error apropiado.

---

4. Método para eliminar un equipo por identificador (borrado lógico)

- ✓ Este método permite eliminar un equipo del fichero equipos.dat mediante **borrado lógico**.
- ✓ No se eliminará físicamente el registro, sino que se marcará el atributo borrado = true.
- ✓ Esto conserva la estructura del fichero y permite reutilizar posiciones vacías en futuras inserciones.

---

5. Método para añadir o modificar un patrocinador de un equipo

- ✓ Este método permite **agregar un nuevo patrocinador** o **actualizar los datos** de uno existente dentro de la lista de patrocinadores de un equipo.

  Pasos:

- Buscar el equipo por idEquipo en el fichero.
- Comprobar si el patrocinador ya existe (por nombre):
  - Si existe, actualizar su donacion o fechaInicio.
  - Si no existe, agregarlo al final de la lista.
- **Método para borrar un patrocinador de un equipo**

Métodos y Ejercicios de Integración – Gestión Conjunta de Corredores y Equipo

1. -Modificar el método de guardar un corredor para comprobar la existencia del equipo

✓ Antes de guardar un corredor en el fichero secuencial corredores.dat, el sistema debe comprobar que el **equipo indicado existe** en el fichero aleatorio equipos.dat y que **no está marcado como borrado**.

✓ El método no permitirá registrar corredores que hagan referencia a equipos inexistentes o eliminados.

2.- Mostrar corredores agrupados por equipo

✓ Leer todos los corredores y equipos desde los ficheros.

✓ Agrupar los corredores por equipo, mostrando primero el nombre del equipo y debajo el listado de corredores que pertenecen a él.

Ejemplo de salida esperada:

```
Equipo: Atlético Norte
  - Juan Pérez (Velocista)
  - María López (Fondista)
Equipo: Deportivo Sur
  - Ana Gómez (Fondista)
```

---

3. Mostrar número de corredores por equipo

✓ Contar cuántos corredores pertenecen a cada equipo y mostrar un resumen.

Ejemplo de salida:

```
Atlético Norte → 2 corredores
Deportivo Sur → 1 corredor

Total de corredores: 9
```

---

4. Mostrar la suma total de donaciones por equipo y los corredores que patrocinan

✓ Calcular la suma total de las donaciones de los patrocinadores de cada equipo.

✓ Mostrar junto al nombre del equipo el total de donaciones y el listado de corredores que pertenecen a ese equipo.

Ejemplo de salida esperada:

```
Equipo: Racing Galicia
  Total donaciones: 25,000.00 €
  Corredores:
    - María López (Fondista)
    - Pedro García (Velocista)
```

---

5. Buscar los corredores patrocinados por una marca concreta

✓ Pedir al usuario el nombre de una marca patrocinadora (por ejemplo, “Nike”).

✓ Mostrar todos los corredores cuyos equipos están patrocinados por esa marca.

Ejemplo de salida:

```
Patrocinador: Nike
Corredores patrocinados:
  - Juan Pérez (Atlético Norte)
  - Carlos Ruiz (Deportivo Sur)
```

---

6. Mostrar el mapa de puntuaciones promedio por equipo

✓ Calcular el promedio de puntuaciones de todos los corredores de un equipo.

✓ Mostrar los resultados en orden descendente por promedio.

Ejemplo de salida:

```
Promedio de puntuaciones por equipo:
1. Deportivo Sur → 95.2 puntos
2. Racing Galicia → 84.7 puntos
3. Unión Atlética → 70.3 puntos
```