Práctica P5b

Colecciones Set y ordenación por defecto (interfaz Comparable)

Cuestiones Previas

En esta práctica modificamos y ampliamos el código desarrollado en la práctica P5a, cambiando el array de plazas bidimensional estático `carSpaces` por dos colecciones Set dinámicas (una `HashSet` y otra `TreeSet`), e incorporamos también la gestión de la carga de la batería de los coches eléctricos/híbridos. Para ello, deberías leer antes los documentos "*Introducción a Comparable*" e "*Introducción a Set*" que se encuentran antes que éste.

Copia todo tu código generado en el paquete `P5a` de la práctica anterior, a un nuevo paquete `P5b` (ubicado dentro del directorio `Practicas`) y realiza en éste todas las modificaciones necesarias para esta práctica.

1 Actividad 1. Gestionando la clase `Parking` con colecciones

En esta actividad modificamos el código desarrollado en la práctica `P5a` para modelar y gestionar las plazas como colecciones `Set` en vez de como un array bidimensional estático. Para ello:

- Modifica en la clase `Parking` la declaración de `carSpaces`. En lugar de un array bidimensional de plazas (objetos `CarSpace`), cámbiala por dos colecciones `Set` de plazas, las siguientes:
  - `busyCarSpaces`: una colección `HashSet` de plazas, para las plazas ocupadas (por tanto, `busyCarSpaces` no es una colección ordenada).
  - `freeCarSpaces`: una colección `TreeSet` de plazas, ordenada por coordenada, para las plazas libres.

    Nota: Para que la JVM pueda mantener ordenados los objetos (`CarSpace`) de un `TreeSet`, es necesario que los pueda comparar entre sí. Para ello, es necesario que la clase `CarSpace` implemente la interfaz `Comparable`. Debes implementar la interfaz `Comparable` en la clase `CarSpace`, definiendo y codificando el método `compareTo`. El código de este método debe comparar dos coordenadas. Una forma adecuada de hacerlo es implementar también la interfaz `Comparable` para la clase `Coordinate`, definiendo y codificando en ésta también el método `compareTo`. Así, en el método `compareTo` de `CarSpace` se llamaría al método `compareTo` de `Coordinate`.

- Incorpora los cambios necesarios en el código de aquellos métodos de la clase `Parking` que hagan referencia a la variable `carSpaces` (sustituida ahora por `busyCarSpaces` y `freeCarSpaces`):
  - Creación del parking. En el constructor de la clase `Parking`:
    - Crea el set `busyCarSpaces`, inicialmente vacío.
    - Crea el set `freeCarSpaces`, y puéblalo con todas las plazas libres (los objetos `CarSpace` con las coordenadas correspondientes, y con matrícula a `null`).
    - A medida que vayas leyendo plazas ocupadas del fichero, mueve la plaza del set `freeCarSpaces` al `busyCarSpaces`.

      Nota. En lugar de recorrer todo el set `freeCarSpaces` buscando la plaza del coche aparcado, puedes usar el método `ceiling` de la clase `TreeSet` para obtenerla de una forma más eficiente.

      Nota: Para que las operaciones de añadir y eliminar sobre el `HashSet busyCarSpaces` funcionen correctamente, en la clase `CarSpace` debes añadir el método `equals`, redefiniendo el de la clase `Object` (usa en el método `equals` de la clase `CarSpace` el `equals` de la clase `Coordinate`).
  - Entradas al parking. En el método `carEntry`, usa el set `freeCarSpaces` para obtener la plaza libre para aparcar el coche. Una vez marcada como no libre, muévela al set `busyCarSpaces`.

    Nota. Para obtener la primera plaza libre para un coche de combustión, puedes usar el método `first` de la clase `TreeSet`. Para un eléctrico/híbrido, puedes usar `higher` o `ceiling`.

- Salidas del parking. En el método `carDeparture`, usa el set `busyCarSpaces` para obtener la plaza ocupada y liberarla. Una vez liberada, muévela al set `freeCarSpaces`.
- Almacenar el parking. En el método `saveParking` debes almacenar en el fichero las plazas ocupadas (que están en el set `busyCarSpaces`), ordenadas por coordenada.

  Nota: Como `busyCarSpaces` no es una colección ordenada, debes crear un nuevo `TreeSet` ordenado a partir de `busyCarSpaces`, y guardar este nuevo `TreeSet` en el fichero. Esta tarea es sencilla, simplemente tienes que crear un nuevo `TreeSet` usando el constructor que admite como parámetro otro `Set` (en este caso `busyCarSpaces`).
- Dibujo del parking. En el método `toMap` debes representar todas las plazas en el dibujo, combinando las libres con las ocupadas.

  Nota: Para ello, igual que en el punto anterior, puedes crear un nuevo `TreeSet` ordenado por coordenada a partir de `busyCarSpaces`, y luego añadirle las plazas de `freeCarSpaces` (usando su método `addAll`). Usa luego ese nuevo `TreeSet` completo como fuente para generar el dibujo del parking.

2 Actividad 2. Modificando la carga de la batería

En esta actividad incorporamos la gestión de la carga de la batería, consistente en actualizar el nivel de carga de batería de los coches eléctricos e híbridos, a partir del tiempo de aparcamiento (diferencia entre la hora de entrada y de salida al parking de un coche). Ese nuevo nivel debe reflejarse en el fichero que almacena todos los coches de la ciudad.

2.1 Actualizando las clases `ElectricCar` e `HybridCar`

Modifica el método `increaseBatteryChargeLevel` creado en la práctica P3 para las clases `ElectricCar` y `HybridCar`. Ahora, en lugar de recibir un porcentaje de incremento de la batería, recibirá el tiempo que ha pasado conectado a un cargador eléctrico:

```
public void increaseBatteryChargeLevel(float chargeTime) {...}
```

el método debe calcular el nuevo nivel de batería de acuerdo con la fórmula siguiente:[1]

```
newLevel= actualLevel + (chargeTime * chargerPower / BATTERY_CAPACITY) * 100
```

donde:

- **chargeTime**, es el tiempo en *horas* (valor *float*) que el coche estuvo cargando la batería (argumento `chargeTime` del método).
- **chargerPower** es la potencia del cargador en KW. Su valor es el asociado al atributo POWER de la clase `ElectricCharger`.
- **BATTERY_CAPACITY** es la capacidad de la batería del coche en kWh. Será la misma para todos los coches eléctricos (100 kWh), y la misma para todos los coches híbridos (15 kWh). Para modelar estos valores, define los atributos estáticos adecuados (constantes) en las clases `ElectricCar` y `HybridCar`.

Si el nuevo nivel de batería (`newLevel`) es mayor de 100, recórtalo a 100.

[1] El termino *chargeTime * chargerPower* representa el incremento de carga de la batería durante el tiempo que el coche estuvo aparcado y recargando.

2.2 Actualizando la clase CarDB

Crea ahora en la clase `CarDB` un nuevo método para actualizar el nivel de la batería de uno de los coches almacenados en el `ArrayList` que gestiona:

```
public void increaseCarBatteryChargeLevel(String plate, String entryTime, String departureTime) {...}
```

Su código debe hacer lo siguiente:

- Localizar en la colección `cityCars` guardada en los objetos de la clase `CarDB` el coche con la matrícula indicada (argumento `plate`).
- Calcular el tiempo en horas (como valor `float`) entre la hora de entrada (argumento `entryTime`) y salida (argumento `departureTime`). Para calcular ese tiempo, puedes emplear el siguiente método:

```
private float intervalInHours(String inTime, String outTime) {
        int hi = Integer.parseInt(inTime.split(":")[0].trim());
        int mi = Integer.parseInt(inTime.split(":")[1].trim());
        int ho = Integer.parseInt(outTime.split(":")[0].trim());
        int mo = Integer.parseInt(outTime.split(":")[1].trim());

        int dif = (ho*60+mo)-(hi*60+mi);
        return ((float)dif/60);
}
```

- Invocar el método anterior `increaseBatteryChargeLevel(float chargeTime)` del coche eléctrico o híbrido, donde `chargeTime` es el tiempo en horas calculado en el punto anterior.

Invoca este método inmediatamente después de que el coche salga del parking[2], cuando el coche sea eléctrico o híbrido. En ese momento conoces tanto la matrícula, como las horas de entrada y salida del coche del parking.

3 Actividad 3. Clase principal

Crea la clase `P5b`, que contendrá el método `main` y será la clase principal de la práctica. En el método `main`:

- Recibe y lee seis argumentos, los nombres de seis ficheros:
  - `file1`: un fichero existente con la estructura y contenido de un parking.
  - `file2`: un fichero existente de entradas y salidas.
  - `file3`: el nombre del fichero en el que guardar el parking.
  - `file4`: un fichero existente con los coches existentes en la ciudad.
  - `file5`: el nombre del fichero donde guardar los coches existentes en la ciudad.
  - `file6`: el nombre del fichero donde guardar el dibujo del parking.
- Crea un objeto `cdb` de la clase `CarDB` (asígnalo a una variable global) e invoca su método `readCityCarsFile` para leer el fichero `file4` con los coches de la ciudad y que estos se almacenen en el `ArrayList cityCars` del objeto creado `cdb`.
- Crea un objeto de la clase `Parking` (asígnalo a una variable global `miParking`) a partir del fichero `file1`.
- Actualiza `miParking` a partir del fichero de entradas/salidas `file2` (llamando al método `processIO`).
- Guarda `miParking`, invocando su método `saveParking` con `file3`.
- Guarda la colección `cityCars` en el fichero `file5`, invocando su método `saveCarsToFile`.
- Crea el dibujo de `miParking` invocando su método `toMap` y salva el resultado en el fichero `file6`.

[2] Es decir, dentro del método `processIO` de la clase P5b, después de invocar al método `carDeparture`.