	FUNDAMENTOS DE PROGRAMACIÓN **EXAMEN DE SEPTIEMBRE. 8 de septiembre de 2015**	**Curso: 2014/15**

CONTEXTO

Se desea implementar parte de un sistema relacionado con la gestión de líneas de autobuses. Para ello se consideran los siguientes tipos:

Tipo **Bus**:

Propiedades:
- matrícula, de tipo String, consultable.
- fecha de inicio del servicio, de tipo LocalDate, consultable (recuerde que LocalDate es mutable).
- número máximo de pasajeros, de tipo Integer, consultable y modificable.
- años de servicio, de tipo Integer, derivada. Indica el número de años completos (de 365 días) que ha cumplido el vehículo.

Orden natural: por matrícula.
Criterio de Igualdad: por matrícula.
Representación como cadena: Matrícula
Constructores: un constructor que tiene un parámetro por cada propiedad básica del tipo y otro a partir de String que recibe una cadena formada por la matrícula, la fecha de inicio del servicio y el número máximo de pasajeros separados por #, como por ejemplo "2536 BFG # 03/06/2002 # 80 ".
Restricciones:
- Ninguna propiedad puede ser nula.
- La fecha de inicio del servicio debe ser anterior a la actual del sistema.
- El número máximo de pasajeros debe ser superior a 0 e igual o inferior a 100 (este valor podría cambiar).

Tipo **Viaje:**

Propiedades:
- trayecto, de tipo List<String>, consultable y modificable. Representa la lista con los nombres de las paradas del viaje, ordenadas de origen a destino.
- distancia total (en km), de tipo Double, consultable y modificable.
- velocidad media (en km/h), de tipo Double, derivada.
- tiempo total (en horas), de tipo Double, consultable y modificable.

Criterio de igualdad: por trayecto.
Representación como cadena: la de la lista de la propiedad trayecto. Ejemplo: "[Sevilla, Madrid, Barcelona]"
Constructores: un constructor que tiene un parámetro por cada propiedad básica del tipo, y otro a partir de String.
Restricciones:
- El trayecto debe tener al menos dos paradas (origen y destino).

Tipo **Persona**:

Propiedades:
- nombre, de tipo String, consultable y modificable.
- apellidos, de tipo String, consultable y modificable.
- dni, de tipo String, consultable y modificable.

Orden natural: por dni, a igualdad por apellidos y si hay empate, por nombre.
Criterio de igualdad: por igualdad de dni, apellidos y nombre.
Representación como cadena: "apellidos, nombre, dni"
Constructores: uno con parámetros, y otro a partir de String con una cadena con el formato de la representación anterior.

Tipo **Conductor,** subtipo de Persona:

Propiedades:
- Fecha de contratación, de tipo LocalDate, consultable y modificable.

Orden natural, **criterio de igualdad y representación como cadena:** los de Persona.
Constructores:
- Un constructor que recibe un String con el formato: "apellidos, nombre, dni - dd/mm/aaaa", donde dd es el día, mm es el mes y aaaa el año de la fecha de contratación.
- Un constructor con un parámetro por cada propiedad básica.

Restricciones:
- Las propiedades no pueden ser nulas.
- La fecha de contratación debe ser anterior a la actual.

EJERCICIO 1 (1.5 puntos)

Escriba, para la clase `BusImpl`, que implementa la interfaz `Bus`:

a) La cabecera y los atributos de la clase.
b) El constructor con parámetros.
c) El constructor a partir de String.
d) Los métodos consultores y modificadores necesarios relacionados con las propiedades matrícula, número máximo de pasajeros, fecha de inicio del servicio y años de servicio.
e) Los métodos `equals` y `compareTo`.

EJERCICIO 2 (1.5 puntos)

Se pide implementar el tipo Conductor reutilizando mediante herencia la clase `PersonaImpl`. Para ello, complete **exclusivamente** los siguientes apartados:

a) Defina la cabecera y los atributos de la clase `ConductorImpl`.
b) Defina los constructores de la clase. Suponga ya implementados los métodos privados necesarios para controlar las restricciones del tipo **(no se pide su implementación)**.
c) Defina los métodos necesarios para completar el resto del tipo siempre y cuando la implementación de dichos métodos sea **estrictamente necesaria**.

EJERCICIO 3 (2 puntos)

Implemente en la clase de utilidad Viajes uno o varios métodos que den respuesta a los siguientes apartados, sin usar la clase Stream de Java 8, y **no usando más de un bucle por método**. Se penalizará el uso de métodos auxiliares innecesarios.

a) Dada una lista de ciudades y un conjunto de viajes, devolver un conjunto ordenado por el orden natural de las cadenas, con las ciudades que no están en el trayecto de ninguno de los viajes. **En este apartado no se permite utilizar ningún método auxiliar**.

```
public static SortedSet<String> ciudadesNoCubiertas(List<String> ciudades, Set<Viaje> viajes)
```

b) Dado un conjunto de autobuses, devolver el que admita más pasajeros. Si el conjunto está vacío, se devolverá `null`.

```
public static Bus masPasajeros(Set<Bus> buses)
```

c) Dado un conjunto de conductores, devolver el subconjunto ordenado de aquellos que hayan sido contratados antes de una fecha dada como parámetro.

```
public static SortedSet<Conductor> antesFecha(Set<Conductor> cond, LocalDate fecha)
```

d) Dado un conjunto de autobuses, devolver el tanto por ciento de autobuses que tienen una antigüedad menor a un número determinado de años. Si el conjunto está vacío, se devolverá `null`.

```
public static Double porcentajeAutobusesAntigüedad(Set<Bus> buses, Integer años)
```

e) Dado un conjunto de viajes, devolver cierto si hay algún viaje que tenga entre sus paradas la que se pasa como parámetro. Tenga en cuenta que la parada que se pasa como parámetro y la que se encuentra en un viaje podrían contener mayúsculas o minúsculas indistintamente y ser equivalentes (p. ej. en el conjunto de viajes podría aparecer "Sevilla", mientras que el valor del parámetro podría ser "SEVILLA" o "sevilla").

```
public static Boolean existeParada(Set<Viaje> viajes, String parada)
```

EJERCICIO 4 (1.5 puntos)

Se desea implementar una factoría de objetos de tipo Viaje. Para ello, en una nueva clase de utilidad `Viajes`, implemente las siguientes propiedades poblacionales:

- Media aritmética del número de escalas (paradas entre el origen y el destino, sin incluir éstas) de los viajes creados mediante la factoría. Para proporcionar esta propiedad cree el método `getNumMedioEscalas`. Si no se han creado viajes, se devolverá un valor -1.
- Conjunto de ciudades que son destino para un origen determinado. Para proporcionar esta propiedad cree el método `getDestinosConOrigen`. Si no hay ningún viaje con la ciudad dada como origen, se devolverá el conjunto vacío.

En la misma clase `Viajes`, implemente dos métodos creacionales para el tipo `Viaje`: un método creacional copia y otro a partir de `String`. Llame a ambos métodos `createViaje`.

NOTA: **NO utilice bucles ni métodos de la clase Stream de Java8.**

EJERCICIO 5 (2 puntos)

Implemente en la clase de utilidad `Viajes` los siguientes métodos. **Los apartados b), c), d) y e) deben resolverse utilizando exclusivamente la clase Stream de Java 8.**

a) Un método `numeroViajesPorParadas` que recibe un conjunto de Viaje y devuelve un `Map<String, Integer>` en el que las claves son los nombres de las paradas y los valores el número de viajes en que aparece dicha parada. **En este apartado no se permite utilizar la clase Stream de Java 8**.
b) Un método `getMaximaDuracion` que dado un conjunto de viajes, devuelve la duración del viaje más largo de todos ellos, siendo el viaje más largo el que tiene un mayor número de paradas. En caso de que no existan viajes, el método debe devolver 0.0.
c) Un método `añadirTiempoDescanso` que dado un conjunto de viajes y una parada concreta, añade un tiempo extra determinado, dado en horas, a los viajes cuyo trayecto incluya dicha parada.
d) Un método `getParadas` que dado un conjunto de viajes, devuelve el conjunto de todas las posibles paradas por las que pasan los viajes, sin que haya paradas duplicadas.
e) Un método `hayViajesCirculares` que dado un conjunto de viajes, devuelve un valor booleano indicando si existe algún viaje circular, es decir, algún viaje que empiece y termine en la misma parada.

EJERCICIO 6 (Lenguaje C) (1.5 puntos)

Dados el tipo Logico y el tipo Cadena, capaz de almacenar un máximo de 255 caracteres, responda a las siguientes cuestiones:

a) Defina en lenguaje C:
- Un tipo **Paradas**, con las siguientes propiedades:
  - parada, que es un array para almacenar hasta 100 elementos de tipo Cadena.
  - nParadas, de tipo entero, que almacena el número real de paradas.
- Un tipo **Viaje**, con las siguientes propiedades:
  - trayecto, de tipo Paradas.
  - distanciaTotal, de tipo real (se mide en kilómetros).
  - duracionTotal, de tipo real (se mide en horas).
- Un puntero **PParadas** al tipo Paradas.
- Un puntero **PViaje** al tipo Viaje.
- Un array de viajes **ArrayViajes** que permita almacenar hasta 50 viajes.

NOTA. Para los dos ejercicios siguientes no se preocupe de usar la función fflush, ni de que queden caracteres en el buffer de entrada o de salida.

b) Escriba una función `int pideTrayecto` que teniendo como parámetro de salida un trayecto, pida por teclado el número de paradas que tendrá el trayecto, que debe ser mayor o igual que 2 y menor o igual que 100. Si no se cumple la restricción se visualizará el mensaje: "*Error en el número de paradas del trayecto*" y la función devuelve el valor -1. En caso correcto se pedirán todos los nombres de las paradas que forman el trayecto, de la siguiente manera:

Parada [1]: <*aquí se tecleará el nombre de la primera parada*>
Parada [2]: <*aquí se tecleará el nombre de la segunda parada*>
…

Y así sucesivamente hasta completar el número de paradas del trayecto. El nombre de una parada puede contener espacios en blanco, como por ejemplo: "A Coruña".

c) Escriba una función `int cuentaViajesConParadaEn` que reciba un array de Viaje y una parada y devuelva el número de viajes del array que incluyen dicha parada. Nota: el nombre de la parada pasado como parámetro debe coincidir exactamente con el contenido en el trayecto del viaje.

Anexo: clases e interfaces de la API de Java

```
public class Collections {
  public static void  <T extends Comparable<? super T>>
     sort ( List<T> l);
  public static void sort( List<T> l, Comparator<? super T> c);
  public static void reverse (List<T> list);
  public static List<T> nCopies(int n, T o);
}
```

```
public interface SortedSet<E> extends Set<E> {
   SortedSet<E> headSet(E toElement);
   SortedSet<E> tailSet(E fromElement);
   SortedSet<E> subSet(E fromElement, E toElement);
   E first();
   E last();
}
```

```
public class LocalDate {
   static LocalDate now();
   static LocalDate of(int year, int month, int dayOfMonth);
   static LocalDate parse(String date, DateTimeFormatter f);
   int getYear(); int getMonth(); int getDayOfMonth();
   boolean isBefore(LocalDate d);
}
public class DateTimeFormatter {
   String format(TemporalAccessor temporal)
   static DateTimeFormatter  ofPattern(String pattern);
}
```

```
public class String {
    char charAt (int index);
    boolean contains (CharSequence s);
    int indexOf (char c);
    boolean isEmplty();
    int length();
    Boolean startsWith(String s);
    String[] split(String sep);
    String toUpperCase();
}
```

```
public interface Map<K,V> {
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();
   Set<K> keySet();
   Collection<V> values();
}
```

```
public interface Comparator<T> {
   int compare(T o1, T o2);
   static <T,U extends Comparable<? super U>> Comparator<T>
      comparing(Function<? super T, ? extends U> keyExtractor);
   static <T extends Comparable<? super T>> Comparator<T>
      naturalOrder();
   default Comparator<T> reversed();
   default Comparator<T> thenComparing(Comparator<? super T>
      other);
}
```

```
public class Optional<T> {
   T get();
   boolean isPresent();
   static <T> Optional<T> of(T value);
   T orElse (T other);
   T orElseThrow
      (Supplier<? extends X> exceptionSupplier);
}
```

```
public class Period {
    static Period between(
       LocalDate startDateInclusive, LocalDate endDateExclusive);
    int getDays();
    int getMonths();
    int getYears();
}
```

```
public interface Stream<T> {
   T reduce(T identity, BinaryOperator<T> accumulator);
   Optional<T> reduce(BinaryOperator<T> accumulator);
   <U> U reduce(U identity, BiFunction<U,? super T,U>
      accumulator, BinaryOperator<U> combiner);
   <R> R collect(Supplier<R> resultFactory, BiConsumer<R,?
      super T> accumulator, BiConsumer<R,R> combiner);
   <R,A> R collect(Collector<? super T,A,R> collector);
   boolean allMatch(Predicate<? super T> predicate);
   boolean anyMatch(Predicate<? super T> predicate);
   boolean noneMatch(Predicate<? super T> predicate);
   Optional<T> min(Comparator<? super T> comparator);
   Optional<T> max(Comparator<? super T> comparator);
   long count();
   Optional<T> findFirst();
   Optional<T> findAny();
   void forEach(Consumer<? super T> action);
   Stream<T> filter(Predicate<? super T> predicate);
   DoubleStream mapToDouble(ToDoubleFunction<? super T>
      mapper);
   IntStream mapToInt(ToIntFunction<? super T> mapper);
   LongStream mapToLong(ToLongFunction<? super T> mapper);
   <R> Stream<R> map(Function<? super T,? extends R> mapper);
   IntStream mapToInt(ToIntFunction<? super T> mapper);
   DoubleStream mapToDouble(ToDoubleFunction<? super T>
      mapper);
   <R> Stream<R> flatMap(Function<? super T, ? extends Stream<?
      extends R>> mapper);
   Stream<T> distinct();
   Stream<T> sorted();
   Stream<T> sorted(Comparator<? super T> comparator);
}
```

```
public interface DoubleStream {
    OptionalDouble average();
    double sum();
}
```

```
public interface IntStream {
    OptionalDouble average();
    int sum();
}
```

```
public interface LongStream {
    OptionalDouble average();
    Long sum();
}
```