UNIVERSITAT JAUME·I

1º Ingeniería Informática
1º Matemática Computacional

EI1008/MT1008 Programación II

Práctica 2: Programación orientada a objetos

Ejercicio 1

Implementa en Java una clase `Punto` que permita representar puntos formados por dos coordenadas `x` e `y`. Ambas se deben guardar en atributos privados de tipo `double`. La especificación de la clase requiere que tenga los siguientes constructores y métodos públicos:

1. Un constructor `Punto(double x, double y)` que permita crear un nuevo punto dándole sus coordenadas.
2. Un constructor sin parámetros `Punto()` que permita crear un nuevo punto con coordenadas $(0, 0)$.
3. Un método `double getX()` que devuelva la coordenada `x` del punto.
4. Un método `double getY()` que devuelva la coordenada `y` del punto.
5. Un método `Punto desplazar(double desplazamientoX, double desplazamientoY)` que devuelva el nuevo punto obtenido al sumarle al punto los desplazamientos que se le pasen como parámetros. Ten en cuenta que la llamada `p.desplazar(a, b)` no debe modificar `p`.
6. Un método `double distancia(Punto otroPunto)` que permita calcular la distancia entre el punto y otro punto que se le pase como parámetro.

   Recuerda que la distancia entre (x_1, y_1) y (x_2, y_2) es $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Para calcularla puedes utilizar `Math.sqrt` y `Math.pow`.
7. Un método `String toString()` que permita obtener una cadena formada por las coordenadas del punto, encerradas entre paréntesis y separadas por una coma y un blanco. Por ejemplo, para el punto de coordenadas 3,5 y 2,45 se devolvería la cadena `"(3.50, 2.45)"`.

   Fíjate en que, para evitar confundir la coma decimal con la coma que separa la `x` de la `y`, usamos un punto como separador de la parte entera y de la parte decimal de los números. Además, los números se muestran con solo dos decimales. Para conseguir todo ello, puedes devolver la cadena `String.format(Locale.US, "(%.2f,%.2f)", x, y)`.
8. Un método `boolean equals(Object otroObjeto)` que devuelva `true` si el punto tiene las mismas coordenadas que otro que se le pase como parámetro y `false` en caso contrario.

El fichero auxiliar `UsaPuntos.java` es un sencillo ejemplo de prueba que emplea todos los constructores y métodos definidos en la clase `Punto`. Descárgalo y experimenta con él.

Ejercicio 2

Implementa en Java una clase `Restaurante` que permita representar restaurantes. De cada restaurante nos interesa almacenar su nombre (una cadena), su posición (un objeto de la clase `Punto` definida en el ejercicio anterior) y su valoración (un entero). Esos tres datos se deben guardar en atributos privados. La especificación de la clase requiere que tenga los siguientes constructores y métodos públicos:

1. Un constructor `Restaurante(String nombre, Punto posición, int valoración)` que permita crear un nuevo restaurante a partir de los datos recibidos.

2. Un método `String getNombre()` que devuelva el nombre del restaurante.

3. Un método `Punto getPosición()` que devuelva la posición del restaurante.

4. Un método `int getValoración()` que devuelva la valoración del restaurante.

5. Un método `double distancia(Punto p)` que devuelva la distancia entre el restaurante y el punto dado como parámetro. Para ello, el método debe usar el método `distancia` de la clase `Punto`.

6. Un método estático `Restaurante[] leeRestaurantes(String nombreFichero)` que construya y devuelva un vector de restaurantes cuyos datos se obtengan a partir del fichero de texto indicado. La primera línea de este fichero contiene un entero con la cantidad de restaurantes y cada una de las restantes líneas contiene, separados por espacios en blanco, las coordenadas `x` e `y` del punto que determina la posición de un restaurante, la valoración de ese restaurante y su nombre (sin espacios en blanco).

   Para que la lectura de valores flotantes almacenados en el fichero considere como separador el punto en lugar de la coma decimal, puedes utilizar `sc.useLocale(Locale.US)`, donde `sc` es el objeto de la clase `Scanner` que usas en la lectura del fichero de datos.

Ejercicio 3

Queremos escribir un programa que permita averiguar el restaurante más próximo a una posición indicada por el usuario. A partir del nombre del fichero de texto que almacena la información de los restaurantes y de la posición en la que se encuentra el usuario, el programa mostrará en la salida estándar el nombre y la posición del restaurante más cercano en línea recta.

No te pedimos implementar este programa, sino descargar el fichero `RestauranteMasProximo.java`, que contiene una posible solución al problema planteado, y estudiar su código y experimentar con él. Comprueba que se comporta correctamente al usar tu implementación de las clases `Restaurante` y `Punto`. Por ejemplo, si empleas el fichero `restaurantes.txt` y el punto $(21, 15)$, el programa debería indicar que el restaurante más cercano es `Restaurante_Marade`, situado en el punto $(20, 18)$.

Ejercicio 4

Basándote en el ejercicio anterior, escribe un programa similar que también pregunte la distancia máxima que se quiere andar y muestre el restaurante con mejor valoración situado a una distancia inferior o igual a la distancia dada.

Tu solución debe incluir un método estático `restauranteMejorValorado` que tenga como parámetros un vector de restaurantes, v, un punto, p, y una distancia, d, y devuelva como resultado el restaurante de v mejor valorado cuya distancia a p sea menor o igual a d. En caso de que no haya ningún restaurante situado a la distancia máxima que se quiere andar, el método debe devolver `null`.

Ejercicio 5

Implementa en Java una clase `Fecha` que permita representar fechas formadas por día, mes y año. Esos tres datos se deben guardar en atributos privados de tipo `int`. La especificación de la clase requiere que tenga los siguientes constructores y métodos públicos:

1. Un constructor `Fecha(int día, int mes, int año)` que permita crear una nueva fecha. De momento, puedes suponer que los datos proporcionados por el usuario de la clase representan una fecha válida.

2. Un constructor `Fecha(Fecha otraFecha)` que permita crear una nueva fecha que sea una copia de la que se le pasa como parámetro.

3. Un método `String toString()` que permita obtener una cadena que representa la fecha con el formato `día/mes/año`.
4. Un método `boolean equals(Object otroObjeto)` que devuelva `true` si la fecha es igual a la que se le pase como parámetro y `false` en caso contrario.
5. Un método `int compareTo(Fecha otraFecha)` que devuelva un valor negativo, cero o positivo según la fecha sea anterior, igual o posterior, respectivamente, a `otraFecha`.
6. Un método de acceso `int getDía()` que devuelva el día.
7. Un método de acceso `int getMes()` que devuelva el mes.
8. Un método de acceso `int getAño()` que devuelva el año.
9. Un método estático `boolean esAñoBisiesto(int año)` que devuelva `true` cuando el año dado sea bisiesto y `false` en caso contrario. Recuerda que un año es bisiesto si es divisible por 4 y no es divisible por 100, o si es divisible por 400.
10. Un método estático `int díasMes(int mes, int año)` que devuelva la cantidad de días de un determinado mes, teniendo en cuenta si es febrero y pertenece a un año bisiesto.
11. Un método estático `Fecha hoy()` que devuelva la fecha del sistema. Para ello, puedes ayudarte del siguiente bloque de código, tras importar la clase `java.util.Calendar`:

    ```
    Calendar calendario = Calendar.getInstance();
    int día = calendario.get(Calendar.DAY_OF_MONTH);
    int mes = calendario.get(Calendar.MONTH) + 1;
    int año = calendario.get(Calendar.YEAR);
    ```

12. Un método `Fecha díaSiguiente()` que cree y devuelva una nueva fecha correspondiente al día siguiente. Ten en cuenta que la llamada `f.díaSiguiente()` no debe modificar `f`.

El fichero auxiliar `UsaFechas.java` es un sencillo ejemplo de prueba que emplea todos los constructores y métodos definidos en la clase `Fecha`. Descárgalo y experimenta con él.

Ejercicio 6

Modifica la clase `Fecha`[1] para que lance una excepción *no verificada* de tipo `ExcepcionFechaNoValida` cuando los valores pasados al constructor o a los métodos estáticos `esAñoBisiesto` y `díasMes` no se correspondan con una fecha válida.

Para evitar problemas con los cambios de calendario a lo largo de la historia, considera que solo serán válidos los años comprendidos entre 1800 y 2500, ambos incluidos. Haz que esta información se almacene en sendos atributos estáticos, como se indica a continuación:

```
public static final int PRIMER_AÑO = 1800;
public static final int ÚLTIMO_AÑO = 2500;
```

El fichero auxiliar `LeerFecha.java` pide repetidamente al usuario tres valores (día, mes y año) hasta que los mismos representen una fecha correspondiente al período indicado. Descárgalo y comprueba que se comporta correctamente. A continuación, modifica su código para que indique si la fecha leída es anterior, igual o posterior a la fecha actual, es decir, la fecha que nos proporciona el método estático `hoy()`.

[1] Si deseas conservar la versión de la clase `Fecha` del ejercicio anterior, puedes guardarla en un paquete diferente, por ejemplo, en `practica2.ejercicio5`.

Ejercicio 7

Queremos escribir un programa que le proponga al usuario adivinar una fecha secreta y, finalmente, le informe del número de intentos que ha necesitado. Cada vez que el usuario se equivoque, el programa debe ayudarle diciéndole si la fecha secreta es anterior o posterior a la que ha introducido.

No te pedimos implementar este programa, sino descargar el fichero auxiliar `AdivinarFecha.java`, que contiene una posible solución al problema planteado, y estudiar su código y probarlo. A continuación, modifica el programa para limitar la fecha secreta al año actual, es decir, al año de la fecha proporcionada por el método estático `hoy()`.

Ejercicio 8

Implementa en Java una clase `Tarea` que permita representar tareas formadas por una fecha y una descripción. Estos datos se deben guardar en atributos privados. La especificación de la clase requiere que tenga los siguientes constructores y métodos públicos:

1. Un constructor `Tarea(Fecha fecha, String descripción)` que permita crear una nueva tarea a partir de los datos recibidos.
2. Un método `Fecha getFecha()` que devuelva la fecha asociada a una tarea.
3. Un método `String getDescripción()` que devuelva la descripción asociada a una tarea.
4. Un método `String toString()` que permita obtener una cadena formada por la fecha asociada a la tarea, el carácter dos puntos, un blanco y la descripción de la tarea. Por ejemplo, para una tarea con fecha 24/2/2025 y descripción *Dentista*, se devolvería la cadena `"24/2/2025: Dentista"`.

A continuación, implementa una clase `Agenda` que permita representar agendas de tareas pendientes. Las tareas se guardarán **ordenadas por fecha** de menor (más antigua) a mayor (más reciente) en un vector privado de objetos de la clase `Tarea` antes definida. La longitud de este vector deberá coincidir en todo momento con la cantidad de tareas almacenadas (en particular, la longitud será cero cuando la agenda esté vacía). La especificación de la clase requiere que tenga los siguientes constructores y métodos. Todos ellos serán públicos, excepto `posiciónInserción`, que será privado:

1. Un constructor sin parámetros `Agenda()` que permita crear una nueva agenda inicialmente vacía.
2. Un método privado `int posiciónInserción(Tarea tarea)` que devuelva la posición del vector de tareas en la que debería insertarse la tarea dada para que el vector siga ordenado. Cuando la agenda ya contenga otras tareas con esa misma fecha, la nueva tarea debería añadirse a continuación de ellas.
3. Un método `void añadir(Tarea tarea)` que añada la tarea dada a la agenda de modo que siga ordenada por fecha. Este método debe hacer uso del método privado `posiciónInserción`.
4. Un método `int cantidad()` que devuelva la cantidad de tareas almacenadas en la agenda.
5. Un método `Tarea[] consultar(Fecha fecha)` que devuelva un vector con todas las tareas correspondientes a la fecha dada. La longitud del vector devuelto será cero cuando la agenda no contenga ninguna tarea con esa fecha.
6. Un método `void borrarPasadas(Fecha fecha)` que borre todas las tareas anteriores a la fecha dada, sin incluirla.
7. Un método `void borrar()` que borre todas las tareas anteriores al día de hoy. Para ello, debes llamar al método `borrarPasadas`.

8. Un método `String toString()` que permita obtener una representación de la agenda en forma de cadena. En esta cadena aparecerá representada cada una de las tareas de la agenda (con el formato propio de la clase `Tarea`), seguida de un salto de línea, como se observa en el siguiente ejemplo:

```
24/2/2025: EI1007-C1
24/2/2025: Recoger paquete
24/2/2025: Dentista
10/3/2025: EI1008-C1
17/3/2025: EI1010
14/4/2025: EI1008-C2
28/4/2025: EI1007-C2
```

El fichero auxiliar `UsaAgenda.java` es un sencillo ejemplo de prueba que emplea todos los constructores y métodos definidos en la clase `Fecha`, salvo el método `borrar()`. Descárgalo y comprueba que funciona correctamente con tu implementación de las clases `Tarea` y `Agenda`. A continuación, modifica el código como creas conveniente para probar también el método `borrar()`.

Ejercicio 9

Implementa en Java una clase `LineaPoligonal` que permita representar líneas poligonales. Una línea poligonal está formada por un conjunto de cero o más segmentos, de modo que el extremo final de cada segmento coincide con el extremo inicial del siguiente.

Podemos representar una línea poligonal mediante un vector privado de puntos. Por ejemplo, un vector que contenga los puntos $(2, 3)$, $(4, 5)$, $(6, 7)$ y $(8, 9)$, en ese orden, representa una línea poligonal formada por tres segmentos en la que el primer segmento va del punto $(2, 3)$ al punto $(4, 5)$, el segundo segmento va del $(4, 5)$ al $(6, 7)$ y el tercer segmento va del $(6, 7)$ al $(8, 9)$.

Para mejorar la eficiencia de las operaciones que añaden y quitan puntos, utilizaremos otro atributo privado que indique la cantidad de puntos de la línea. De esta manera distinguiremos entre la **talla** o capacidad del vector (la cantidad máxima de puntos que puede almacenar) y la **ocupación** del vector indicada por el nuevo atributo (la cantidad de puntos válidos que realmente contiene). Si el valor de este nuevo atributo es $n > 1$, representará una línea poligonal formada por $n - 1$ segmentos cuyos extremos son los puntos almacenados en las n primeras posiciones del vector, de talla $m \geq n$. Si el valor del nuevo atributo es cero o uno no tenemos ningún segmento, pero consideraremos que se trata de una línea poligonal válida de longitud cero.

La especificación de la clase requiere que tenga los siguientes constructores y métodos públicos:

1. Un constructor sin parámetros `LineaPoligonal()` que permita crear una nueva línea poligonal que inicialmente estará vacía. Para ello, debe crear un vector de puntos de una determinada talla inicial (es habitual emplear un valor 8 o 10) que represente una línea sin puntos.

2. Un método `void añadir(Punto punto)` que permita extender la línea poligonal añadiéndole al final el punto dado como parámetro.

   El método debe *redimensionar* el vector de puntos únicamente cuando sea necesario (es decir, cuando su talla m coincida con el número de puntos n), de modo que su nueva talla sea el doble de la anterior. Si en el vector hay espacio disponible para un nuevo punto, este simplemente se añadirá al final.

   Importante: Fíjate en que este es el único método que accede al atributo `length` del vector de puntos; el resto de métodos trabajará con el atributo que representa la ocupación del vector.

3. Un método `void quitar(int posición)` que permita reducir la línea poligonal quitándole el punto que ocupa la posición dada como parámetro, contando desde cero inclusive. La línea no debe cambiar si la posición es menor que cero o mayor o igual que la cantidad de puntos.

   El método debe suprimir el punto correspondiente sin disminuir nunca la talla del vector[2] ni hacer uso de ningún otro vector auxiliar.

4. Un método `void quitar(Punto punto)` que permita reducir la línea poligonal quitándole el punto dado como parámetro. Para ello, debes llamar al método anterior. Si el punto aparece varias veces en la línea, sólo se debe quitar su primera aparición. La línea no debe cambiar si no contiene el punto dado.

5. Un método `void trasladar(double desplazamientoX, double desplazamientoY)` que permita modificar la línea poligonal desplazando cada uno de sus puntos.

6. Un método `double longitud()` que devuelva la longitud de la línea poligonal. Si la línea tiene cero puntos o un punto, su longitud es cero. Si tiene dos o más puntos, su longitud es la suma de las longitudes de los segmentos que la forman. La longitud de cada segmento es la distancia entre sus dos puntos extremos.

7. Un método `String toString()` que permita obtener una representación de la línea poligonal en forma de cadena. En dicha representación aparecen representados todos los puntos de la línea, separados por dos guiones. La representación de cada punto es la propia de la clase `Punto`. Por ejemplo, la cadena `(2.00, 3.00)--(4.00, 5.00)--(6.00, 7.00)--(8.00, 9.00)` representa la línea poligonal definida por esos cuatro puntos como extremos de sus tres segmentos. Los casos especiales en que tenemos cero puntos o un punto se representan mediante la cadena vacía o mediante la cadena que representa un punto, respectivamente.

8. Un método `boolean equals(Object otroObjeto)` que devuelva `true` si la línea tiene los mismos puntos y en el mismo orden que otra que reciba como parámetro y `false` en caso contrario. Si dos líneas contienen los mismos puntos, pero en orden inverso, consideraremos que son diferentes.

El fichero auxiliar `UsaLineaPoligonal.java` es un sencillo ejemplo de prueba que emplea todos los constructores y métodos definidos en la clase `LineaPoligonal`. Descárgalo y comprueba que funciona correctamente con tu implementación de las clases `LineaPoligonal` y `Punto`.

[2] Aunque en este ejercicio te pedimos que no lo hagas, una estrategia habitual consiste en reducir a la mitad la talla m del vector cuando el número de puntos n llega a valer $m/4$.