Práctica 1: **La vida en el campo**

Programación II

Marzo 2018 (versión 13/2/2019)

DLSI – Universidad de Alicante

Alicia Garrido Alenda

Normas generales

- El plazo de entrega para esta práctica se abrirá el lunes 11 de marzo a las 9:00 horas y se **cerrará el viernes 15 de marzo a las 23:59 horas**. No se admitirán entregas fuera de plazo.
- Se debe entregar la práctica en un fichero comprimido de la siguiente manera:

  1. Abrir un terminal.
  2. Situarse en el directorio donde se encuentran los ficheros fuente (`.java`) de manera que al ejecutar `ls` se muestren los ficheros que hemos creado para la práctica y ningún directorio.
  3. Ejecutar:

     ```
     tar cfvz  practica1.tgz   *.java
     ```

Normas generales comunes a todas las prácticas

1. La práctica se debe entregar exclusivamente a través del servidor de prácticas del departamento de Lenguajes y Sistemas Informáticos, al que se puede acceder desde la página principal del departamento (`http://www.dlsi.ua.es`, enlace "Entrega de prácticas") o directamente en `http://pracdlsi.dlsi.ua.es`.

   - **Bajo ningún concepto** se admitirán entregas de prácticas por otros medios (correo electrónico, UACloud, etc.).
   - El usuario y contraseña para entregar prácticas es el mismo que se utiliza en UACloud.
   - La práctica se puede entregar varias veces, pero sólo se corregirá la última entrega.

2. El programa debe poder ser compilado sin errores con el compilador de Java existente en la distribución de Linux de los laboratorios de prácticas; si la práctica no se puede compilar su calificación será 0. Se recomienda compilar y probar la práctica con el autocorrector durante su implementación para detectar y corregir errores.

3. La práctica debe ser un trabajo original de la persona que entrega; **en caso de detectarse indicios de copia con una o más prácticas (o compartición de código) la calificación de la práctica será** `0` y se enviará un informe al respecto tanto a la dirección del departamento como de la titulación.

4. Se recomienda que los ficheros fuente estén adecuadamente documentados, con comentarios donde se considere necesario.

5. La primera corrección de la práctica se realizará de forma automática, por lo que es imprescindible respetar estrictamente los formatos de salida que se indican en este enunciado.

6. El cálculo de la nota de la práctica y su influencia en la nota final de la asignatura se detallan en las transparencias de la presentación de la asignatura.

7. Para evitar errores de compilación debido a codificación de caracteres que **descuenta** `0.5` **de la nota de la práctica**, se debe seguir una de estas dos opciones:

   a) Utilizar EXCLUSIVAMENTE caracteres ASCII (el alfabeto inglés) en vuestros ficheros, **incluídos los comentarios**. Es decir, no poner acentos, ni ñ, ni ç, etcétera.

   b) Entrar en el menú de Eclipse `Edit > Set Encoding`, aparecerá una ventana en la que hay que seleccionar `UTF-8` como codificación para los caracteres.

8. El tiempo estimado por el corrector para ejecutar cada prueba debe ser suficiente para que finalice su ejecución correctamente, en otro caso se invoca un proceso que obliga a la finalización de la ejecución de esa prueba y se considera que dicha prueba no funciona correctamente.

Descripción del problema

Para esta práctica es necesario implementar las clases que se definen a continuación, a las que se pueden añadir variables de instancia y/o clase (siempre que sean privadas) y métodos cuando se considere necesario (que pueden ser públicos o privados).

En **Valores** se establece el valor calórico por kilo de los posibles frutos que hay, por tanto se caracteriza por:

- **Relacion**: clase estática privada que se caracteriza por *nombre* (`String`) y *valor* (`double`), que se le pasan al constructor cuando se crea un objeto. El valor debe ser mayor que `0`, en otro caso se asigna un valor de `0.5`. La cadena debe ser distinta de `null` y de la cadena vacía, en otro caso se le asigna la cadena "`pitaya`".

- *clasificacion*: array dinámico en el que se van almacenando los distintos nombres de frutos junto con su valor calórico por kilo (`static ArrayList<Relacion>`).

No requiere constructor puesto que todas las acciones se realizarán a nivel de clase[1]. Las acciones serán:

- *add*: se le pasa por parámetro una cadena y un número real para crear una nueva relación y añadirla al final de su clasificación, siempre que no exista previamente una relación ya creada con dicho nombre[2]. El método devuelve cierto si se añade una nueva relación a la clasificación y falso en cualquier otro caso.

- *consulta*: se le pasa por parámetro una cadena y devuelve su valor asociado[3] en la clasificación. Si no lo encuentra devuelve `-1.0`.

- *getNombres*: devuelve un array dinámico de cadenas con los nombres contenidos en su clasificación en el orden en el que aparecen en ésta.

Una **Coordenada** se caracteriza por:

- *latitud*: indica la latitud de la coordenada (`double`):

- *longitud*: indica la longitud de la coordenada (`double`);

Cuando se crea una **Coordenada** se le pasan por parámetro dos números reales. El primero indica la latitud, que debe tener un valor dentro del rango `[-90,90]`, si no tomará por defecto el valor `0`. El segundo indica la longitud, que debe tener un valor dentro del rango `[-180,180]`, si no tomará por defecto el valor `0`.

Las acciones que se pueden realizar con una coordenada son:

- *iguales*: se le pasa por parámetro una coordenada y devuelve cierto si tanto su latitud como su longitud son iguales a las de la coordenada pasada por parámetro, y falso en cualquier otro caso.

[1] Todos los métodos son `static`

[2] Ignorando diferencias entre mayúsculas y minúsculas

[3] Ignorando diferencias entre mayúsculas y minúsculas

⊙ *distancia*: calcula y devuelve la distancia a la que se encuentra de la coordenada pasada por parámetro. Para ello aplica la fórmula:

$$distancia = \sqrt{(latitud - latitud_c)^2 + (longitud - longitud_c)^2}$$

Si no se puede calcular la distancia, devuelve `-1` por defecto.

Un **Fruto** se caracteriza por:

⁕ *estado*: indica si el fruto es comestible (cierto) o no (falso) (`boolean`):

⁕ *peso*: indica el peso del fruto (`double`);

⁕ *nombre*: nombre del fruto (`String`);

Cuando se crea un **Fruto** se le pasa por parámetro su nombre que debe encontrarse entre los almacenados en la clasificación de *Valores*; si no es así el nombre del fruto será "`pitaya`" y se creará la relación en *Valores* para él con valor `0` si no existe dicha relación previamente. El peso se inicializa a `0` y no es comestible.

Las acciones que se pueden realizar con un **Fruto** son:

⊙ *transforma*: se le pasa por parámetro un entero de manera que aumenta su peso un `20 %` de la cantidad pasada por parámetro, siempre que ésta sea positiva. Si el fruto tiene un peso igual o superior a `0.3` significa que ya es comestible. Si el fruto cambia de estado devuelve cierto; en cualquier otro caso devuelve falso.

⊙ *valorCalorico*: devuelve el valor calórico del fruto, es decir, su peso por el valor establecido en la clasificación de *Valores*.

⊙ *getNombre*: devuelve el nombre del fruto.

⊙ *getEstado*: devuelve la cadena "`comestible`" si el fruto ya es comestible y "`inmaduro`" en cualquier otro caso.

⊙ *getPeso*: devuelve el peso.

Una **Planta** se caracteriza por:

⁕ *estado*: cadena que indica el estado de la planta (`String`). Tomará uno de los siguientes valores:

1. `semilla`
2. `germinado`
3. `brote`
4. `adulta`

⁕ *nombre*: cadena que indica el nombre de la planta (`String`);

⁕ *fruto*: cadena que indica el nombre de los frutos que produce la planta (`String`);

⁕ *plantada*: huerta en la que ha sido plantada (`Huerta`);

⁕ *frutos*: array que contiene los frutos de la planta (`Fruto[]`);

Cuando se crea una **Planta** se le pasa por parámetro dos cadenas y un entero. La primera cadena para el nombre de la planta; si esta cadena es `null` o cadena vacía el nombre de la planta será "`vegetal`". La segunda para el nombre de los frutos que produce; si esta cadena es `null` o cadena vacía el nombre de los frutos será "`pitaya`". El tercer parámetro indica la cantidad máxima de frutos que puede tener, que como mínimo debe ser `1`. Inicialmente su estado será "`semilla`", no está plantada y no tiene frutos.

Las acciones que se pueden realizar con una **Planta** son:

⊙ *abona*: se le pasa por parámetro un entero. Si la planta está plantada en algún lugar, con esta acción se abona la planta de manera que cambia de la siguiente manera:

- si su estado era "`semilla`" lo cambia por "`germinado`";
- si su estado era "`germinado`" lo cambia por "`brote`";
- si su estado era "`brote`" lo cambia por "`adulta`";
- si su estado es "`adulta`" primero crea un nuevo fruto que añade en la primera posición libre (`null`) de sus frutos si es posible, y a continuación transforma los que ya tuviera con la cantidad indicada en el parámetro. Si el peso de algún fruto sobrepasa el valor pasado por parámetro, dicho fruto cae de la planta, por tanto se elimina del array.

Si se cambia el estado de la planta de alguna manera (ya sea estado o frutos o estado de sus frutos), el método devuelve cierto. En cualquier otro caso, devuelve falso.

⊙ *recolecta*: quita de la planta todos los frutos que sean comestibles y los devuelve en un array dinámico ordenados por su peso[4].

⊙ *arranca*: la planta ha sido arrancada de la huerta en la que está, por tanto ya no está plantada en dicha huerta.

⊙ *getNombre*: devuelve el nombre de la planta.

⊙ *getEstado*: devuelve el estado de la planta.

⊙ *getPlantada*: devuelve la huerta donde esta plantada.

⊙ *setPlantada*: se le pasa por parámetro la huerta en la que ha sido plantada.

⊙ *getFrutos*: devuelve el array de frutos de la planta.

⊙ *getFruto*: devuelve el nombre de los frutos que produce.

Una **Huerta** se caracteriza por:

⁕ *huerto*: matriz donde estarán las plantas que se planten en dicha huerta (`Planta[][]`);

⁕ *cuidador*: persona que cuida del estado del huerto (`Persona`);

⁕ *localizacion*: coordenada gps de la localización de la huerta (`Coordenada`);

[4]Cuando hay dos valores iguales, el segundo encontrado se coloca detrás del primero.

⁕ *localizadas*: array dinámico con las huertas que han sido localizadas (`static ArrayList<Huerta>`).

Una **Huerta** se crea pasándole por parámetro dos enteros; el primero indica el número de filas que tendrá la matriz de plantas, que debe ser mayor que `0`, en otro caso tomará el valor `2` por defecto. El segundo indica el número de columnas, que debe ser mayor que `0`, en otro caso tomará el valor `2` por defecto. Inicialmente no tendrá cuidador y no estará localizada.

Las acciones que se pueden realizar con una **Huerta** son:

⊙ *planta*: se le pasa por parámetro un objeto de tipo *Planta* que será plantada en la última posición libre de la matriz[5], siempre que dicha planta no esté ya plantada en otra huerta, el estado de la planta sea "`semilla`", y teniendo en cuenta que en una fila solo puede haber plantas del mismo tipo (mismo nombre), de manera que si en una fila hay plantas con otro nombre, se pasa a la siguiente. Si se puede plantar la planta, le pasa un mensaje a la planta para indicarle donde ha sido plantada y el método devuelve cierto, en cualquier otro caso devuelve falso.

⊙ *recolecta*: se le pasa por parámetro un `String`, que es el nombre de los frutos que se quiere recolectar. El método devuelve un array dinámico con los frutos recolectados con ese nombre[6] en toda la huerta, empezando a recolectar en la posición `(0,0)` del huerto y recorriéndolo por filas.

⊙ *abona*: se le pasa por parámetro un entero y un `String`, de manera que se abonan con la cantidad indicada por el primer parámetro, siempre que éste sea mayor que `0`, las plantas del huerto, empezando por la posición `(0,0)`, y cuyo nombre coincida con la cadena pasada por parámetro[7]. El método devuelve la cantidad de plantas que han cambiado de estado de alguna manera (estado, número de frutos o estado de sus frutos) después de haber sido abonadas.

⊙ *consulta*: se le pasan dos enteros y devuelve el nombre de la planta que ocupa esa posición en el huerto, donde el primer entero indica la fila y el segundo la columna. Por defecto devuelve `null`.

⊙ *arranca*: se le pasa por parámetro un `String` y dos enteros. El primer parámetro indica el nombre de la planta[8] que se va a arrancar de la huerta, el segundo indica la fila del huerto, y el tercero la columna del huerto. Si en la posición indicada por el segundo y tercer parámetro hay una planta con ese nombre, se arranca la planta del huerto y se devuelve. En cualquier otro caso se devuelve `null`.

⊙ *localiza*: se le pasan por parámetro dos reales para su localización; el primero indica su latitud y el segundo su longitud. Una vez localizada se debe añadir al final del array *localizadas*. Una huerta sólo puede ser localizada una vez, es decir, no puede cambiar su localización.

⊙ getAdultas: devuelve un array dinámico con el nombre de todas las plantas que haya en el huerto cuyo estado sea "`adulta`", sin que haya nombres repetidos en dicho array, en el orden en el que se encuentren en el huerto, empezando por la posición `(0,0)` y recorriéndolo por filas.

[5]Empezando por la posición `(0,0)` y recorriendo por filas.
[6]Ignorando diferencias entre mayúsculas y minúsculas.
[7]Ignorando diferencias entre mayúsculas y minúsculas
[8]Ignorando diferencias entre mayúsculas y minúsculas.

- ⊙ getLocalizacion: devuelve la localización de la huerta.
- ⊙ getLocalizadas: devuelve el array dinámico de huertas localizadas.
- ⊙ getCuidador: devuelve la persona que cuida del huerto.
- ⊙ setCuidador: se le pasa por parámetro la persona que cuida del huerto a partir de este momento.
- ⊙ getHuerto: devuelve el huerto.

Una **Persona** se caracteriza por:

- ⁕ *`nombre`*: cadena con el nombre de la persona (`String`);
- ⁕ *`huerta`*: huerta que cuida (`Huerta`);

Una **Persona** se crea pasándole por parámetro una cadena con su nombre, que por defecto será "`John Doe`", y no está al cuidado de ninguna huerta inicialmente.

Las acciones que se puede realiza una **Persona** son:

- ⊙ *`planta`*: se le pasan por parámetro una planta y una huerta, de manera que la persona intenta plantar la planta en la huerta. El método devuelve cierto si la acción se realiza con éxito, y falso en cualquier otro caso.
- ⊙ *`paseo`*: si no tiene una huerta a su cuidado, se da un paseo por las huertas localizadas y decide cuidar la primera que encuentra que no tiene un cuidador, devolviendo la coordenada donde está localizada dicha huerta. Si ya tiene una huerta a su cuidado, devuelve la coordenada de su huerta. En cualquier otro caso devuelve `null`.
- ⊙ *`malasHierbas`*: recorre la huerta por filas, empezando en la fila `0`, arrancando todas la plantas adultas que hay que no tienen ningún fruto. El método devuelve un array dinámico con las plantas arrancadas.
- ⊙ *`abona`*: se le pasa por parámetro un entero y un `String`, de manera que se abonan con la cantidad indicada por el primer parámetro las plantas de la huerta cuyo nombre coincida con la cadena pasada por parámetro. El método devuelve la cantidad de plantas que han cambiado de estado de alguna manera (estado, número de frutos o estado de sus frutos) después de haber sido abonadas.
- ⊙ getNombre: devuelve el nombre de la persona.
- ⊙ getHuerta: devuelve la huerta que la persona que cuida en este momento.

Restricciones en la implementación

- ⊛ Todas las variables de instancia o clase de las clases deben ser privadas (no accesibles desde cualquier otra clase).
- ⊛ Algunos métodos deben ser públicos y tener una *`signatura`* concreta:

- En **Valores**
  - `public static boolean add(String,double)`
  - `public static double consulta(String)`
  - `public static ArrayList<String> getNombres()`
- En **Coordenada**
  - `public Coordenada(double,double)`
  - `public boolean iguales(Coordenada)`
  - `public double distancia(Coordenada)`
- En **Fruto**
  - `public Fruto(String)`
  - `public boolean transforma(int)`
  - `public double valorCalorico()`
  - `public String getNombre()`
  - `public String getEstado()`
  - `public double getPeso()`
- En **Planta**
  - `public Planta(String,String,int)`
  - `public boolean abona(int)`
  - `public ArrayList<Fruto> recolecta()`
  - `public String getNombre()`
  - `public void arranca()`
  - `public String getEstado()`
  - `public Huerta getPlantada()`
  - `public void setPlantada(Huerta)`
  - `public Fruto[] getFrutos()`
  - `public String getFruto()`

- En **Huerta**
  - `public Huerta(int,int)`
  - `public boolean planta(Planta)`
  - `public ArrayList<Fruto> recolecta(String)`
  - `public int abona(int,String)`
  - `public String consulta(int,int)`
  - `public Planta arranca(String,int,int)`
  - `public void localiza(double,double)`
  - `public Coordenada getLocalizacion()`
  - `public Persona getCuidador()`
  - `public ArrayList<String> getAdultas()`
  - `public static ArrayList<Huerta> getLocalizadas()`
  - `public void setCuidador(Persona)`

  - `public Planta[][] getHuerto()`

- En **Persona**
  - `public Persona(String)`
  - `public boolean planta(Planta,Huerta)`
  - `public Coordenada paseo()`
  - `public ArrayList<Planta> malasHierbas()`
  - `public int abona(int,String)`
  - `public String getNombre()`
  - `public Huerta getHuerta()`

⊛ Ninguno de los ficheros entregados en esta práctica debe contener un método `public static void main(String[] args)`.

⊛ Todas las variables de clase e instancia deben ser privadas. En otro caso se restará un `0.5` de la nota total de la práctica.

Clases y métodos de Java

Algunas variables de instancia, parámetros y valores devueltos en esta práctica son de un tipo concreto de Java: el tipo `ArrayList`. Un `ArrayList` es un array que se redimensiona de forma automática conforme se le añaden y/o quitan elementos, y que puede contener cualquier tipo de elementos. El tipo de elementos que contendrá el array se define en la declaración de una variable de este tipo entre `<` y `>`.

Para poder usarlos es necesario importar la librería correspondiente al principio:

```
import java.util.ArrayList;
```

Algunos métodos de esta clase que pueden ser útiles/necesarios para esta práctica son:

- `boolean add(E e)`: añade `e` al final del array y devuelve cierto.
- `void clear()`: borra todos los elementos del array.
- `E get(int pos)`: devuelve el elemento que ocupa la posición `pos` del array.
- `boolean isEmpty()`: devuelve cierto si el array no contiene ningún elemento, y falso en otro caso.
- `int size()`: devuelve el número de elementos que contiene el array.
- `E remove(int pos)`: borra del array el elemento que ocupa la posición `pos` y lo devuelve.
- `E set(int pos,E e)`: sustituye el elemento que ocupa la posición `pos` en el array por el que se le pasa por parámetro, devolviendo el elemento que había originalmente.

Un ejemplo de declaración y uso de este tipo de arrays sería:

```
ArrayList<Integer> array; // declaracion donde se indica que va a contener enteros

array=new ArrayList<Integer>(); // se crea el objeto, indicando tambien el tipo de los
                                // elementos que va a contener

array.add(4); // agrega 4 al final
array.add(8); // agrega 8 al final
array.add(12); // agrega 12 al final
int elem=array.get(1); // obtiene el elemento que ocupa la pos 1
System.out.println(elem); // muestra por pantalla 8
```

Para trabajar con funciones matemáticas, Java dispone de la clase *Math*, que tiene métodos implementados para realizar diversas operaciones matemáticas. Algunos de ellos son:

- `int abs(int i)`: devuelve el valor absoluto del entero pasado por parámetro. Se invoca:

  ```
  int i=-7;
  int j=Math.abs(i); // j contiene el valor 7
  ```

- `double pow(double x,double y)`: devuelve el resultado de elevar x a y (x^y). Se invoca:

  ```
  double x=3,y=2;
  double z=Math.pow(x,y); // z contiene el valor 9.0
  ```

- `double sqrt(double x)`: devuelve la raíz cuadrada del valor pasado por parámetro. Se invoca:

  ```
  double x=36;
  double z=Math.sqrt(x); // z contiene el valor 6.0
  ```

Probar la práctica

- En UACloud se publicará un corrector de la práctica con un conjunto mínimo de pruebas (se recomienda realizar pruebas más exhaustivas de la práctica).
- Podéis enviar vuestras pruebas (fichero con extensión `java`, con un método `main` y sin errores de compilación) a los profesores de la asignatura mediante tutorías de UACloud, para obtener la salida correcta a esa prueba. En ningún caso se modificará/corregirá el código de las pruebas. Los profesores contestarán a vuestra tutoría adjuntando la salida de vuestro `main`, si no da errores.
- El corrector viene en un archivo comprimido llamado `correctorP1.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar:

  ```
  tar xfvz correctorP1.tgz
  ```

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica1-prueba`: dentro de este directorio están los ficheros con extensión `.java`, programas en Java con un método `main` que realizan una serie de pruebas sobre la práctica, y los ficheros con el mismo nombre pero con extensión `.txt` con la salida correcta para la prueba correspondiente.

- Una vez que tenemos esto, debemos copiar nuestros ficheros fuente (sólo aquellos con extensión `.java`) al mismo directorio donde está el fichero `corrige.sh`.
- Sólo nos queda ejecutar el *shell-script*. Primero debemos darle permisos de ejecución si no los tiene. Para ello ejecutar:

```
chmod +x corrige.sh
corrige.sh
```

- Una vez se ejecuta `corrige.sh` los ficheros que se generarán son:
  - `errores.compilacion`: este fichero sólo se genera si el corrector emite por pantalla el mensaje "`Error de compilacion: 0`" y contiene los errores de compilación resultado de compilar los fuentes de una práctica particular. Para consultar el contenido de este fichero se puede abrir con cualquier editor de textos (gedit, kate, etc.).
  - Fichero con extensión `.tmp.err`: este fichero debe estar vacío por regla general. Sólo contendrá información si el corrector emite el mensaje "`Prueba p01: Error de ejecucion`", por ejemplo para la prueba `p01`, y contendrá los errores de ejecución producidos al ejecutar el fuente `p01` con los ficheros de una práctica particular.
  - Fichero con extensión `.tmp`: fichero de texto con la salida generada por pantalla al ejecutar el fuente correspondiente, por ejemplo `p01.tmp` contendrá la salida generada al ejecutar el fuente `p01` con los ficheros de una práctica particular.

  Si en la prueba no sale el mensaje "`Prueba p01: Ok`", por ejemplo para la prueba `p01`, o algún otro de los comentados anteriormente, significa que hay diferencias en las salidas para esa prueba, por tanto se debe comprobar que diferencias puede haber entre los ficheros `p01.txt` y `p01.tmp`. Para ello ejecutar en línea de comando, dentro del directorio `practica1-prueba`, la orden: `diff -w p01.txt p01.tmp`