Práctica 1: **Los albores**

Programación II

Marzo 2018 (versión 16/02/2018)

DLSI – Universidad de Alicante

Alicia Garrido Alenda

Normas generales

- El plazo de entrega para esta práctica se abrirá el lunes 12 de marzo a las 9:00 horas y se **cerrará el viernes 16 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. Además, al principio de todos los ficheros fuente entregados se debe incluir un comentario con el `DNI` de la persona que entrega la práctica, con el siguiente formato:

   // DNI *tuDNI* Nombre

   donde *tuDNI* es el DNI del alumno correspondiente (sin letras), **tal y como aparece en UACloud**, y Nombre es el nombre completo; por ejemplo, el comentario podría ser:

   `// DNI 2737189 CASIS RECOS, ANTONIA`

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

En los albores del mundo aparecieron los primeros habitantes, cuya vida era muy rudimentaria, ya que se dedicaban casi exclusivamente a recolectar productos que encontraban en los terrenos, realizar alguna edificación básica y hacer trueques. También adoraban a los seres místicos, a los que agasajaban con tributos. Por su parte, los místicos otorgaban a cambio vigor y hacían que los terrenos fueran fértiles y generasen productos.

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).

Un **Producto** se caracteriza por:

- ⁕ *tipo*: indica el tipo del producto (`int`):
  1. `Vegetal`
  2. `Animal`
  3. `Piedra`
  4. `Madera`
  5. `Edificado`
  6. `Filosofal`
- ⁕ *peso*: indica el peso del producto (`double`);
- ⁕ *colocado*: indicador de colocación que indica si el producto está dentro de alguna parcela o cesta (`boolean`);
- ⁕ *nombre*: nombre del producto (`String`);

Cuando se crea un **Producto** se le pasan por parámetro su peso, tipo y nombre. El peso debe ser mayor que `0`, en otro caso se le asigna un peso estándar de `1.0`. Si el tipo no es uno de los indicados, por defecto el producto será de tipo `Filosofal`. El nombre debe contener una cadena, en otro caso se le asigna por defecto la cadena "`Ordinario`" como nombre al producto. El producto inicialmente no está colocado en ningún sitio, y un producto nunca puede estar en dos sitios al mismo tiempo.

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

- ⊙ *valorKilo*: calcula el valor por kilo del producto y lo devuelve. Para ello aplica la siguiente fórmula:

$$valorK = \frac{\sum_{i=0}^{n-1}(ascii(c_i) - 97)}{n}$$

  donde c_i es el carácter que ocupa la posición i en el nombre del producto y n la longitud del nombre.

- ⊙ *valorProducto*: calcula el valor del producto y lo devuelve. El valor de un producto es su valor por kilo multiplicado por su peso.

- ⊙ *transforma*: se le pasa por parámetro un producto y un entero. Si el objeto sobre el que se invoca este método es de tipo `Filosofal`, transforma el producto pasado por parámetro al tipo indicado por el segundo parámetro, siempre que éste sea distinto al tipo del producto pasado por parámetro[1] y sea un tipo válido. Si se puede realizar la transformación devuelve `1`, y en cualquier otro caso devuelve `-1`.
- ⊙ *getNombre*: devuelve el nombre del producto.
- ⊙ *getColocado*: devuelve el indicador de colocación.
- ⊙ *setColocado*: se le pasa por parámetro un booleano para actualizar su indicador de colocación;
- ⊙ *getPeso*: devuelve el peso.
- ⊙ *getTipo*: devuelve el tipo del producto.

Un **Terreno** se caracteriza por:

- ⋇ *parcelas*: matriz donde estarán los productos que haya en el terreno (`Producto[][]`);

Un **Terreno** se crea pasándole por parámetro dos enteros; el primero indica el número de filas que tendrá la matriz de productos, que debe ser mayor que `0`, en otro caso tomará el valor `3` por defecto. El segundo indica el número de columnas, que debe ser mayor que `0`, en otro caso tomará el valor `2` por defecto. Por defecto un terreno no tiene propietario.

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

- ⊙ *genera*: se le pasan por parámetro dos enteros, el primero para la fila y el segundo para la columna, que indican la parcela donde crear el producto con la información indicada por el resto de parámetros: un real para el peso, un entero para el tipo y una cadena para el nombre del producto. El producto se crea y se almacena en la parcela indicada anteriormente actualizando su valor de colocado, si en dicha parcela no había ningún producto. Además no pueden generar productos de tipo `Edificado`. Si se crea y almacena el producto, se devuelve cierto. En cualquier otro caso devuelve falso.
- ⊙ *recoge*: se le pasan por parámetro dos enteros, el primero para la fila y el segundo para la columna, que indican la parcela de donde se quiere recoger un producto, devolviéndolo con su valor de colocado actualizado y dejando la parcela vacía (`null`) siempre que el producto no sea de tipo `Edificado`, ya que este tipo de producto no se puede recoger. Si no se puede recoger un producto por cualquier motivo el método devuelve una referencia vacía (`null`).
- ⊙ *destruye*: se le pasan por parámetro dos enteros, el primero para la fila y el segundo para la columna, que indican la parcela donde se destruye la edificación que pudiera haber, de manera que comprueba si el producto de esa parcela es de tipo `Edificado`, y si es así, deja la parcela vacía (`null`) y devuelve el peso del producto. En cualquier otro caso devuelve `0`.
- ⊙ *coloca*: se le pasan por parámetro un producto y dos enteros. El primer entero indica la fila y el segundo la columna de la parcela donde colocar el producto, siempre que en dicha parcela no haya ya un producto y el producto no esté colocado en otro sitio. El método devuelve cierto si coloca el producto, actualizando su valor de colocado, y falso en cualquier otro caso.

[1] No se puede transformar un producto a su mismo tipo.

⊙ *consultaTipo*: se le pasan por parámetro dos enteros, el primero para la fila y el segundo para la columna, que indican la parcela de donde se quiere conocer el tipo de producto que contiene. El método devuelve el tipo de producto siempre que sea posible, en cualquier otro caso devuelve `-1`.

⊙ *consultaPeso*: se le pasan por parámetro dos enteros, el primero para la fila y el segundo para la columna, que indican la parcela de donde se quiere conocer el peso de producto que contiene. El método devuelve el peso del producto siempre que sea posible, en cualquier otro caso devuelve `-1`.

⊙ *existencias*: se le pasa por parámetro un entero que indica el tipo de producto del cual se devuelve la cantidad de productos que hay en el terreno.

⊙ *calculaDemanda*: para cada tipo de producto que hay en el terreno se calcula su tasa de demanda en función de las existencias del tipo de producto y de las existencias del resto de tipos de productos. Para calcular la tasa del tipo de producto i se debe aplicar la fórmula:

$$t_i = \frac{e_i}{\sqrt{\sum_{j=0}^{n-1} e_j^2}} * 100$$

donde n es el número de tipos de productos que hay en el terreno y e_i son las existencias del producto i. Si sólo hubiera un tipo de producto en el terreno, su tasa de demanda sería `100`.

El método devuelve un array de enteros con la parte entera de la tasa calculada para cada tipo de producto. El orden de los tipos de producto es:

1. `Vegetal`
2. `Animal`
3. `Piedra`
4. `Madera`
5. `Edificado`
6. `Filosofal`

teniendo en cuenta que en el array devuelto solo aparecerán aquellos tipos presentes en el terreno y su tamaño se ajustará a este número de elementos.

⊙ *getFilas*: devuelve el número de filas del terreno.

⊙ *getColumnas*: devuelve el número de columnas del terreno.

Un **Habitante** se caracteriza por:

⁕ *nombre*: cadena que contiene el nombre del habitante (`String`);

⁕ *cesta*: array que contendrá los productos recolectados por el habitante (`ArrayList<Producto>`);

⁕ *vigor*: indica la cantidad de energía vital que tiene el habitante (`double`);

⁕ *sexo*: indica si se trata de una mujer (`'M'`) o un hombre (`'H'`) (`char`);

⁕ *poblacion*: array que contendrá todos los habitantes existentes (`static ArrayList<Habitante>`);

Un **Habitante** se crea pasándole por parámetro una cadena para su nombre[2] y un carácter para determinar su sexo; si dicho carácter no es ninguno de los establecidos por defecto será hombre. Inicialmente un habitante tendrá el máximo vigor posible, que es `100`, creará su cesta y será incluído como el último habitante en la población.

Las acciones que puede realizar un **Habitante** son:

⊙ *recolecta*: se le pasan por parámetro un terreno y el tipo de producto que quiere recolectar. El habitante recorre el terreno por filas, empezando por la posición `(0,0)`, buscando este tipo de producto, siempre que no sea el tipo `Edificado`. La búsqueda decrementa su vigor en `0.25` por parcela recorrida y búsqueda fallida; cuando encuentra un producto del tipo buscado, consulta su peso de manera que si tiene vigor suficiente lo recoge del terreno y lo almacena al final de su cesta y no sigue buscando, ya que recoger un producto decrementa el vigor del habitante en el peso del producto multiplicado por `0.1`, y el vigor de un habitante nunca puede ser negativo. El método devuelve cierto si el habitante consigue almacenar un producto en su cesta, y falso en cualquier otro caso.

⊙ *estudio*: el habitante realiza un estudio de su cesta para conocer sus necesidades y devuelve el tipo de producto del que considera que tiene mayor necesidad. Las prioridades de un habitante a la hora de buscar productos son:

1. `Vegetal`
2. `Animal`
3. `Piedra`
4. `Madera`
5. `Filosofal`

de manera que si no tiene ningún producto de tipo vegetal, este será el tipo de producto que devolverá, y así sucesivamente. Si tiene productos de todos estos tipos en su cesta, devolverá el tipo de producto del cual tenga menor cantidad.

⊙ *vigoriza*: se le pasa por parámetro una cadena. Busca entre los productos de su cesta el primer objeto cuyo nombre coincida con la cadena pasada por parámetro, ignorando diferencias entre mayúsculas/minúsculas, y que sea de tipo vegetal o animal. Si encuentra uno:

- busca en su cesta un producto de tipo madera para cocinarlo, de manera que si lo encuentra lo usa para la cocción y el valor nutritivo del producto será el valor del producto íntegro;
- si no lo encuentra, el producto no se puede cocinar y su valor nutritivo será el `50%` de su valor íntegro;

Una vez ha intentado cocinar el producto, el habitante se alimenta con él, incrementando su vigor con el valor nutritivo del producto (sin exceder nunca el máximo vigor posible) y eliminándolo de su cesta, junto con el producto utilizado para su cocción, en caso de haber cocinado el producto. El método devuelve cuánto le falta al habitante para llegar a tener el máximo vigor posible.

[2]Dicha cadena nunca será `null`.

- ⊙ *edifica*: se le pasa una cadena. Busca en su cesta el primer producto de tipo piedra que tenga. Con él crea un producto de tipo edificado, usando la cadena pasada por parámetro como nombre del nuevo producto y como peso el 50 % del peso de la piedra usada, eliminando de su cesta el producto utilizado para edificar. El método devuelve el producto creado, o `null` en su defecto.

- ⊙ *trueque*: se le pasa por parámetro un habitante con el que realizar un trueque. El habitante busca en su cesta de qué tipo de producto tiene más objetos, ya que intentará hacer el trueque con el primer objeto de este tipo que haya en su cesta[3]. A continuación realiza un estudio sobre su cesta para saber que tipo de producto necesita, y le pasa un mensaje al otro habitante con este tipo de producto y el producto con el que hacer el trueque, que si lo acepta le devolverá un producto para almacenar al final de su cesta. El método devuelve un array con los nombres de los productos intercambiados, primero el devuelto por el otro habitante y segundo por el que se le ha cambiado si se realiza el trueque. En cualquier otro caso devuelve `null`.

- ⊙ *haceTrueque*: se le pasa por parámetro un entero, que indica un tipo de producto, y un producto. Saca de su cesta el primer producto del tipo pasado por parámetro y lo devuelve, almacenando en su lugar el producto pasado por parámetro. Si no encuentra en su cesta ningún producto del tipo indicado, devuelve `null` y no almacena el producto pasado por parámetro.

- ⊙ *tributa*: se le pasa por parámetro un místico al que agasajar. Para ello se quita un producto de cada tipo de la cesta (el primero de cada tipo que se encuentre) y se le pasa un mensaje de culto con ellos y el propio nombre al místico, que devolverá la cantidad de vigor con la que incrementar el propio vigor. El método devuelve la cantidad en la que se ha incrementado el vigor, teniendo en cuenta que el valor máximo que puede alcanzar el vigor es `100`.

- ⊙ *plegaria*: se le pasa por parámetro un místico y un terreno. El habitante comprueba en su cesta qué tipo de alimento es del que tiene mayor carencia (`Animal` o `Vegetal`, si para ambos son iguales se elige `Animal`) y le pasa un mensaje de *transforma* al místico con el terreno y este tipo de producto para que lo haya en abundancia en el terreno, ya que a continuación se dedica con alegría y alborozo a recolectar todos los productos que han sido transformados por el místico, sin que por ello se decremente su vigor, añadiéndolos al final de su cesta. El método devuelve la cantidad de productos recolectados.

- ⊙ *getNombre*: devuelve su nombre.

- ⊙ *getVigor*: devuelve su vigor.

- ⊙ *getCesta*: devuelve su cesta.

- ⊙ *getClan*: devuelve su clan, que es la cadena contenida en el nombre desde el primer espacio en blanco encontrado hasta el siguiente, o bien hasta el final de la cadena. En caso de no poder obtener el clan, se devuelve `null`.

- ⊙ *perteneceClan*: se le pasa por parámetro una cadena y devuelve cierto si dicha cadena coincide con su clan ignorando diferencias entre mayúsculas y minúsculas. En cualquier otro caso devuelve falso.

- ⊙ *getHombres*: devuelve el número de hombres que hay en la población.

[3] Ante un mismo número de productos de distinto tipo, se aplica la prioridad de tipos indicada en *estudio*.

⊙ *getMujeres*: devuelve el número de mujeres que hay en la población.

⊙ *getPoblacion*: devuelve la población de habitantes.

Un **Mistico** se caracteriza por:

⁕ *tributos*: array que contendrá los productos del místico (`ArrayList<Producto>`);

⁕ *adoradores*: array que contendrá los nombres de los habitantes que le rinden culto (`ArrayList<String>`).

Cuando se crea un **Mistico** no se le pasa ningún parámetro, pero cada místico debe crear sus tributos y adoradores.

Las acciones que puede realizar un **Mistico** son:

⊙ *culto*: se le pasa por parámetro un array de productos y una cadena. El array de productos son tributos para el místico, por tanto si dicho array contiene productos, el místico calcula la suma del valor por kilo de todos estos productos y la divide por el número de productos; este cálculo será lo que devuelva el método. A continuación incorpora estos productos en el orden que llegan al final de sus tributos y la cadena al final de sus adoradores, siempre que no exista ya una cadena igual.

⊙ *regenera*: se le pasa por parámetro un terreno y un producto. El producto es un tributo, que si no está colocado en ningún otro sitio, lo añade al final de los tributos que ya tenga, y satisfecho con esto, para cada parcela del terreno el místico creará un producto y lo intentará colocar[4]. La forma de crear productos de los místicos es recorrer sus tributos desde el principio, uno por parcela, creando un producto de iguales características. Si hay más parcelas que productos en los tributos, se vuelve al principio de los tributos para continuar regenerando todas las parcelas hasta completar el terreno. El método devuelve la cantidad de productos colocados en el terreno por el místico.

⊙ *transforma*: se le pasa por parámetro un terreno y un entero. Si el místico dispone de un producto de tipo `Filosofal` entre sus tributos, lo utiliza para transformar todos los productos de tipo `Edificado` que haya en el terreno al tipo indicado por el segundo parámetro, siempre que éste no sea `Edificado`, devolviendo el número de productos transformados.

⊙ *getTributos*: devuelve los tributos.

⊙ *getAdoradores*: devuelve los adoradores.

Restricciones en la implementación

⊛ Todas las variables de instancia de las clases deben ser privadas (no accesibles desde cualquier otra clase).

⊛ Algunos métodos deben ser públicos y tener una *signatura* concreta:

- En **Producto**

[4]Si no añade un tributo nuevo, no regenera el terreno.

- `public Producto(double p,int i, String s)`
- `public double valorKilo()`
- `public double valorProducto()`
- `public int transforma(Producto p,int i)`
- `public int getTipo()`
- `public boolean getColocado()`
- `public void setColocado(boolean b)`
- `public double getPeso()`
- `public String getNombre()`

- En **Terreno**
  - `public Terreno(int i,int j)`
  - `public boolean genera(int i,int j,double d,int k,String s)`
  - `public Producto recoge(int i,int j)`
  - `public double destruye(int i,int j)`
  - `public boolean coloca(Producto p, int i,int j)`
  - `public int consultaTipo(int i,int j)`
  - `public double consultaPeso(int i,int j)`
  - `public int existencias(int i)`
  - `public ArrayList<Integer> calculaDemanda()`
  - `public int getFilas()`
  - `public int getColumnas()`
- En **Habitante**
  - `public Habitante(String s,char c)`
  - `public boolean recolecta(Terreno t,int i)`
  - `public int estudio()`
  - `public double vigoriza(String s)`
  - `public Producto edifica(String s)`
  - `public ArrayList<String> trueque(Habitante h)`
  - `public Producto haceTrueque(int i,Producto p)`
  - `public double tributa(Mistico m)`
  - `public int plegaria(Mistico m,Terreno t)`
  - `public String getNombre()`
  - `public double getVigor()`
  - `public String getClan()`
  - `public boolean perteneceClan(String s)`
  - `public ArrayList<Producto> getCesta()`
  - `public static int getHombres()`
  - `public static int getMujeres()`
  - `public static ArrayList<Habitante> getPoblacion()`
- En **Mistico**
  - `public Mistico()`
  - `public double culto(ArrayList<Producto> a,String s)`

- `public int regenera(Terreno t,Producto p)`
- `public int transforma(Terreno t,int i)`
- `public ArrayList<Producto> getTributos()`
- `public ArrayList<String> getAdoradores()`

⊛ 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`