Práctica 3: **Dominó**

Programación II

Mayo 2019

DLSI – Universidad de Alicante

versión: 16/04/2019

Alicia Garrido Alenda

Normas generales

- El plazo de entrega para esta práctica se abrirá el lunes 20 de mayo a las 9:00 horas y se **cerrará el viernes 24 de mayo 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 todos los ficheros que hemos creado para la práctica y ningún directorio.
  3. Ejecutar:

     ```
     tar cfvz  practica3.tgz   Jugador.java Ficha.java Saco.java
     Tablero.java ObjetoNoValidoException.java FichaRepetidaException.java
     JugadaIncorrectaException.java SacoIncompletoException.java
     PartidaGanadaException.java CierreException.java Juego.java
     ```

  4. No entregar más ficheros con extensión `.java` que los que se indican en el punto anterior.

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, Campus Virtual, etc.).
   - El usuario y contraseña para entregar prácticas es el mismo que se utiliza en el Campus Virtual.
   - 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 inmediatamente antes de entregarla.

3. La práctica debe ser un trabajo original de la persona que entrega; **en caso de detectarse indicios de copia de una o más prácticas (o compartición de código) la calificación de la práctica será** `0`.

4. Los ficheros fuente deben estar adecuadamente documentados, con comentarios donde se considere necesario. Como mínimo se debe realizar una breve descripción (2 líneas máximo) del funcionamiento de cada método (exceptuando los métodos `getter` y `setter`).

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 el Campus Virtual**, 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.

Descripción del problema

El objetivo de esta práctica es implementar el juego del dominó. Para ello se implementarán las siguientes clases: **Ficha**, **Tablero**, **Saco**, **Jugador**, **ObjetoNoValidoException**, **FichaRepetidaException**, **SacoIncompletoException**, **JugadaIncorrectaException**, **CierreException** y **PartidaGanadaException**.

Además se tiene que implementar la clase **Juego**, que contendrá un main en el que se jugará una partida completa leyendo la información de un fichero de texto y escribirá en otro fichero de texto el resultado de la partida.

Pasemos ahora a una descripción más detallada de estas clases.

Un **Ficha** se caracteriza por:

- *primera*: número de la primera parte de la ficha (`int`);
- *segunda*: número de la segunda parte de la ficha (`int`);

Cuando se crea una **Ficha** se le pasa por parámetro dos enteros, que deben estar entre `0` y `6`, incluídos, de manera que el primero es la primera parte de la ficha y el segundo la segunda. Si alguno de los números no está en el rango indicado se lanza y propaga la excepción *ObjetoNoValidoException*, a cuyo constructor se le pasan por parámetro los dos enteros.

Las acciones que se pueden realizar con una **Ficha** son las siguientes:

- *getPrimera*: devuelve el número de la primera parte de la ficha.
- *getSegunda*: devuelve el número de la segunda parte de la ficha.
- *creaInversa*: devuelve una ficha con las partes intercambiadas. Si se lanza alguna excepción, ésta se debe propagar.
- *toString*: devuelve una cadena con la información de la ficha con el siguiente formato `[primera,segunda]`, si por ejemplo la variable uno tiene el número `1` y la variable dos el número 2, devolverá la cadena `[1,2]`.

Un **Tablero** se caracteriza por:

- *fichas*: array dinámico que contendrá las fichas jugadas (`ArrayList<Ficha>`);

Cuando se crea un **Tablero** no se le pasa ningún parámetro e inicialmente no contiene fichas.

El comportamiento de un *Tablero* se define mediante las siguientes acciones:

- *getSecuencia*: devuelve un array dinámico de enteros con la secuencia de números contenidas en las fichas, empezando por la posición `0` de sus fichas. Por ejemplo si en el tablero están colocadas las siguientes fichas:

  `[1,2],[2,2],[2,3]`

  devolvería {`1,2,2,2,2,3`}.

⊙ *getInicio*: devuelve el número de la primera parte de la primera ficha del tablero. Si no hay fichas en el tablero devuelve `-1`.

⊙ *getFin*: devuelve el número de la segunda parte de la última ficha del tablero. Si no hay fichas en el tablero devuelve `-1`.

⊙ *coloca*: se le pasan por parámetro una ficha y un entero, de manera que si el entero es:

- `0`: la ficha se coloca al principio de las fichas;
- distinto de `0`: la ficha se coloca al final de las fichas.

La ficha que se coloca tiene que tener el mismo número en la parte que quede adyacente al de la ficha junto a la que se pone en el tablero, en otro caso se lanza y propaga la excepción *JugadaIncorrectaException* cuyo mensaje asociado será el contenido de las dos fichas que no tienen números adyacentes iguales y que se le pasan por parámetro a su constructor, la que se intenta colocar como primer parámetro, y la existente en el tablero como segundo parámetro.

Que dos fichas tengan el mismo número en la parte que quede adyacente significa:

- si es añadir al final: el número de la segunda de la última ficha del tablero tiene que coincidir con el número de la primera de la ficha a colocar;
- si es añadir al principio: el número de la primera de la primera ficha del tablero debe coincidir con el número de la segunda de la ficha a colocar.

Una vez colocada la ficha se comprueba si se da la situación de cierre: mismo número en la primera de la primera ficha del tablero que en la segunda de la última ficha del tablero, y que las siete fichas que contienen ese número estén colocadas en el tablero; si se da esta situación se lanza y propaga *CierreException*, cuyo mensaje asociado será el número que ha provocado el cierre que se le pasa por parámetro a su constructor. Si la ficha se coloca, el método devuelve cierto, en cualquier otro caso falso.

⊙ *toString*: devuelve una cadena con la información de todas sus fichas con el siguiente formato {`[primera,segunda]`,`[primera,segunda]`}. Por ejemplo:

```
{[1,2],[2,2],[2,3]}
```

Un **Saco** se caracteriza por:

⁕ *fichas*: array dinámico que contendrá las fichas del juego (`ArrayList<Ficha>`);

Cuando se crea un **Saco** no se le pasa ningún parámetro e inicialmente no contiene fichas.

El comportamiento de un *Saco* se define mediante las siguientes acciones:

⊙ *roba*: quita de sus fichas la que ocupa la primera posición y la devuelve. Si no quedan fichas en el saco, devuelve `null`.

⊙ *meteFicha*: se le pasa por parámetro una ficha para meterla en el saco, teniendo en cuenta que no puede contener fichas repetidas. Por ejemplo si ya existe una ficha `[1,2]`, no se puede meter una ficha `[2,1]`, porque en el dominó es la misma ficha. Si se presenta esta situación se lanza y propaga *FichaRepetidaException*, cuyo mensaje asociado será el contenido de

la ficha repetida que se le pasa por parámetro al constructor de la excepción. El máximo de fichas que se pueden meter son 28, por tanto si se consigue meter la ficha el método devuelve cierto, y en cualquier otro caso falso.

⊙ *completado*: si en algún momento el saco ha llegado a tener las 28 fichas necesarias para el juego, devuelve cierto[1]. En cualquier otro caso devuelve falso.

⊙ *toString*: devuelve una cadena con la información de todas sus fichas con el siguiente formato {[primera,segunda],[primera,segunda]}. Por ejemplo:

```
{[4,2],[6,0],[2,3]}
```

Un **Jugador** se caracteriza por:

⁕ *propias*: array dinámico de fichas donde se colocan las fichas del jugador (ArrayList<Ficha>);

⁕ *nombre*: nombre del jugador (String);

Inicialmente un **Jugador** se crea pasándole por parámetro una cadena para su nombre. Si la cadena es null o la cadena vacía, se lanza y propaga la excepción *ObjetoNoValidoException*, a cuyo constructor se le pasa por parámetro la cadena No name. Inicialmente un jugador no tiene fichas.

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

⊙ *roba*: se le pasa por parámetro un saco del que roba una ficha y la añade al final de las que ya tenía. El método devuelve cierto si añade una nueva ficha y falso en cualquier otro caso.

⊙ *elige*: se le pasa por parámetro un tablero del que consulta el inicio y el final. Si el tablero ya contiene fichas colocadas, se busca cuáles de sus fichas tienen esos números en alguna de sus partes. Si tiene varias, se elige la que sume el mayor número entre sus dos partes, la quita de sus propias fichas y la devuelve. Si hay más de una elige la primera de ellas que se haya robado.

Si no tiene ninguna, el método devuelve null.

Si el tablero no tiene ninguna ficha colocada se elige la ficha con valor máximo sumando sus partes; si hay más de una se elige la última de ellas que se haya robado.

⊙ *juega*: se le pasa por parámetro un tablero. El jugador tiene que elegir una ficha y colocarla en el tablero pasado por parámetro al principio o al final, de manera que una de las partes de la ficha tenga el mismo número que su posible ficha adyacente en el tablero.

Si puede colocarla tanto al principio como al final, tiene prioridad colocarla al principio siempre que no provoque una situación de cierre, por tanto si ponerla al principio significa que se da la situación de cierre, se pone al final. Si no es posible colocarla debe crear la inversa de la ficha y ver donde la coloca en el tablero. Si continua sin poderse colocar la ficha el método devuelve null. Si consigue colocar la ficha en el tablero y todavía le quedan fichas, devuelve la ficha colocada; si después de colocar la ficha, el jugador no tiene más fichas lanza *PartidaGanadaException* cuyo mensaje asociado será el nombre del jugador que se le pasa por parámetro a su constructor. Si al colocar la ficha en el tablero se lanza alguna excepción, ésta debe ser propagada.

[1] Aunque en el momento actual no las tenga porque ya se han robado fichas del saco.

⊙ *valorMaximo*: se calcula la suma de las partes de cada ficha del jugador, y se devuelve el que sea máximo. Por ejemplo, si un jugador tiene estas fichas:

```
{[1,2],[5,0],[2,3]}
```

devolverá `5`. Si el jugador no tiene fichas devuelve `-1`.

⊙ *getNombre*: devuelve el nombre del jugador.

⊙ *toString*: devuelve una cadena con la información del jugador con el formato `Nombre{[primera,segunda],[primera,segunda]}`. Por ejemplo:

```
Galadriel{[5,2],[6,3],[1,4]}
```

La clase **Juego** es una aplicación en la que se desarrollará una partida de dominó a partir de la información leída de fichero y escribirá el resultado de dicha partida en otro fichero de texto. Para ello esta clase debe tener un método `main` en el que:

⊙ se le pasan dos parámetros[2];

⊙ se abra para lectura el fichero de texto que se le pasa como primer parámetro a la aplicación; el formato de este fichero será:

- número de jugadores; una línea con un entero que debe estar entre 2 y 4 (inclusives) que indica el número de jugadores que se deben crear;
- a continuación vendrán los nombres de los distintos jugadores, uno por línea, hasta encontrar una línea con la cadena "`saco`";
- a continuación vendrá la información de las fichas que se tienen que meter en el saco. Para cada ficha en el fichero habrá una línea con dos enteros separados por un espacio en blanco;

⊙ se abra para escritura el fichero de texto que se le pasa como segundo parámetro a la aplicación;

⊙ al leer el fichero la aplicación debe crear tantos objetos de tipo *Jugador* como se indique en la primera línea (si el número no está entre 2 y 4, por defecto se crean 2 jugadores) con la información leída en las siguientes líneas hasta encontrar la línea "`saco`"; si hay menos líneas que jugadores indicados, se crean con la cadena vacía "". Si se lanza la excepción *ObjetoNoValidoException* no se puede comenzar el juego, por tanto la aplicación termina, pero antes escribe en el fichero que se ha abierto para escritura el nombre de la excepción, dos puntos, su mensaje asociado y nueva línea ("`\n`"). Si hay más líneas que jugadores indicados, simplemente se descarta la información sobrante;

⊙ se crea el saco al leer la línea del fichero que contiene la palabra "`saco`"; a continuación por cada línea leída del fichero se debe crear una ficha y meterla en el saco, escribiendo en el fichero que se ha abierto para escritura una línea con la acción, la información de la ficha y lo que devuelve el método. Por ejemplo:

```
mete [6,6] true
```

[2]No es necesario comprobarlo; se asegura que se le pasan siempre.

Si se lanza alguna excepción, la aplicación simplemente escribe en el fichero que se ha abierto para escritura el nombre de la excepción, dos puntos, su mensaje asociado y nueva línea, por ejemplo:

```
FichaRepetidaException: [6,6]
```

y descarta los datos que generan dicha excepción y continúa leyendo y procesando el fichero;

- ⊙ cuando se ha terminado de leer el fichero de lectura, empieza la partida; se crea el tablero y se comprueba si se ha completado el saco, si no es así se lanza la excepción ***SacoIncompletoException***, cuyo mensaje asociado es el número de fichas que hay en el saco que se le pasa por parámetro a su constructor, y no se puede comenzar el juego, por tanto la aplicación termina, pero antes escribe en el fichero que se ha abierto para escritura el nombre de la excepción, dos puntos, su mensaje asociado y nueva línea, y a continuación el contenido del saco con el formato indicado en su método toString, nueva línea y termina;
- ⊙ cada jugador debe robar `7` fichas del saco, una cada uno alternativamente, empezando por el primer jugador creado, escribiendo en el fichero abierto para escritura una línea por cada vez que se realice la acción con quien roba y el resultado de la acción, por ejemplo:

```
Galadriel roba true
```

- ⊙ a partir de aquí empieza la partida; alternativamente cada jugador juega, y si no consigue colocar ficha en el tablero tendrá que robar del saco. Empieza la partida el jugador que tenga la ficha con el valor máximo y continúa su siguiente. Si hay más de un jugador que tenga el valor máximo tiene prioridad el orden en el que se crearon dichos jugadores; es decir si tenemos dos jugadores con un valor máximo de `9`, uno fue el segundo en crearse y otro el tercero, empieza la partida el jugador que se creo en segundo lugar. Si por ejemplo se crean tres jugadores en este orden: `Uriel`, `Galadriel` y `Amenadiel`, y tanto `Galadriel` como `Amenadiel` tienen la ficha con el valor máximo, el orden para realizar las jugadas sería: `Galadriel`, `Amenadiel`, `Uriel`. Se debe escribir en el fichero abierto para escritura quien realiza la acción y el resultado de ésta, por ejemplo:

```
Galadriel juega [5,4]
Amenadiel juega [4,4]
Uriel juega null
Uriel roba true
```

- ⊙ la partida acaba cuando se lance la excepción **PartidaGanadaException** o **CierreException**, que se debe escribir en el fichero abierto para escritura (el nombre de la excepción, dos puntos, su mensaje asociado y nueva línea), a continuación el estado del tablero y los jugadores (en el orden en que se crearon) con el formato descrito en sus respectivos métodos `toString`, cada uno en una nueva línea.
- ⊙ por último la aplicación debe cerrar ambos ficheros, el de lectura y el de escritura;
- ⊙ el main no tiene que propagar ninguna excepción, las tiene que capturar y tratar;

Un ejemplo de fichero de lectura para la aplicación sería el siguiente:

```
2
Hugo
Cinthia
saco
2 3
0 0
0 2
0 1
1 1
1 2
3 2
```

El fichero de salida que se generaría para este fichero de entrada sería el siguiente:

```
mete [2,3] true
mete [0,0] true
mete [0,2] true
mete [0,1] true
mete [1,1] true
mete [1,2] true
mete [3,2] FichaRepetidaException: [3,2]
SacoIncompletoException: 6
{[2,3],[0,0],[0,2],[0,1],[1,1],[1,2]}
```

Las clases **ObjetoNoValidoException**, **FichaRepetidaException**, **SacoIncompletoException**, **JugadaIncorrectaException**, **CierreException** y **PartidaGanadaException** heredan de la clase *`Exception`* de java.

La clase **ObjetoNoValidoException** tiene dos constructores:

- un contructor que recibe por parámetro dos enteros. En este caso el mensaje que se tiene que devolver al invocar el método *`getMessage`* es "`[i,j]`", donde `i` sería el valor del primer entero y `j` el valor del segundo entero que se le pasan al constructor por parámetro.
- un contructor que recibe por parámetro una cadena. En este caso el mensaje que se tiene que devolver al invocar el método *`getMessage`* es la cadena que se ha pasado por parámetro.

El constructor de la clase **JugadaIncorrectaException** recibe por parámetro dos fichas. El mensaje que se tiene que devolver al invocar el método *`getMessage`* es "`[i,j]!=[k,l]`", donde `[i,j]` serían los valores de la primera ficha y `[k,l]` los valores de la segunda ficha que se le pasan al constructor por parámetro.

Los constructores de las clases **CierreException** y **SacoIncompletoException** reciben por parámetro un entero. El mensaje que se tiene que devolver al invocar el método *`getMessage`* es "`i`", donde `i` es el entero que se le pasa al constructor por parámetro.

El constructor de la clase **FichaRepetidaException** recibe por parámetro una ficha. El mensaje que se tiene que devolver al invocar el método *`getMessage`* es "`[i,j]`", donde `[i,j]` serían los

valores de la ficha que se le pasa al constructor por parámetro.

El constructor de la clase **PartidaGanadaException** recibe por parámetro una cadena. El mensaje que se tiene que devolver al invocar el método *`getMessage`* es la cadena que se le pasa al constructor por parámetro.

Para obtener el nombre de un objeto de cualquier tipo de excepción se puede hacer de la siguiente manera:

```
String nombre=objetoDeTipoExcepcion.getClass().getName();
```

Restricciones en la implementación

⊛ Algunos métodos deben tener una *`signatura`* concreta:

- Clase **Ficha**
  - `public Ficha(int,int) throws ObjetoNoValidoException`
  - `public int getPrimera()`
  - `public int getSegunda()`
  - `public Ficha creaInversa() throws ObjetoNoValidoException`
  - `public String toString()`
- Clase **Tablero**
  - `public Tablero()`
  - `public boolean coloca(Ficha,int) throws JugadaIncorrectaException, CierreException`
  - `public ArrayList<Integer> getSecuencia()`
  - `public int getInicio()`
  - `public int getFin()`
  - `public String toString()`
- Clase **Saco**
  - `public Saco()`
  - `public boolean meteFicha(Ficha) throws FichaRepetidaException`
  - `public Ficha roba()`
  - `public boolean completado()`
  - `public String toString()`
- Clase **Jugador**
  - `public Jugador(String n) throws ObjetoNoValidoException`
  - `public boolean roba(Saco)`
  - `public Ficha elige(Tablero)`

    - `public Ficha juega(Tablero) throws PartidaGanadaException,CierreException, JugadaIncorrectaException,ObjetoNoValidoException`
    - `public int valorMaximo()`
    - `public String getNombre()`
    - `public String toString()`
- Clase **ObjetoNoValidoException**
    - `public ObjetoNoValidoException(int,int)`
    - `public ObjetoNoValidoException(String)`
- Clase **FichaRepetidaException**
    - `public FichaRepetidaException(Ficha)`
- Clase **JugadaIncorrectaException**
    - `public JugadaIncorrectaException(Ficha,Ficha)`
- Clase **SacoIncompletoException**
    - `public SacoIncompletoException(int)`
- Clase **CierreException**
    - `public CierreException(int)`
- Clase **PartidaGanadaException**
    - `public PartidaGanadaException(String)`
- Clase **Juego**
    - `public static void main(String args[])`

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

Segmentación de una cadena en distintos elementos

Una vez tenemos una línea del fichero en una variable de tipo `String`, por ejemplo la variable `linea`, hay que segmentarla para obtener de ella los distintos elementos que necesitamos para procesar la información contenida de manera conveniente.

Esta segmentación la podemos hacer usando el método `split` de la clase String. Este método devuelve un array de `Strings` a partir de uno dado usando el separador que se especifique. Por ejemplo, supongamos que tenemos la variable `linea` de tipo `String` que contiene la cadena "`1 1`" de la cual queremos obtener por separado los dos datos que contiene para crear una ficha. Para ello primero debemos indicar el separador y después segmentar usando el método `split` de la siguiente manera:

```
//indicamos el separador de campos: un espacio en blanco
 String separador = "[ ]";
//segmentamos
 String[] elems = linea.split(separador);
//convertimos a entero cada cadena
//contenida en elems
 int p1 = Integer.parseInt(elems[0]);
 int p2 = Integer.parseInt(elems[1]);
// con estos datos ya se puede crear una ficha
 Ficha f=new Ficha(p1,p2);
```

Probar la práctica

- En el Campus Virtual se publicará un corrector de la práctica con dos 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 las profesoras de la asignatura mediante tutorías del campus virtual, para obtener la salida correcta a esa prueba. En ningún caso se modificará/corregirá el código de las pruebas.
- El corrector viene en un archivo comprimido llamado `correctorP3.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar:

  ```
  tar xfvz correctorP3.tgz
  ```

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica3-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 extensión `txt` con el mismo nombre, con la salida correcta a la pruebas realizadas en el `java` 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.
  - `p01.tmp.err`: este fichero debe estar vacío por regla general. Sólo contendrá información si el corrector emite el mensaje "`Prueba p01: Error de ejecucion`" y contendrá los errores de ejecución producidos al ejecutar el fuente `p01` con los ficheros de una práctica particular.
  - `p01.tmp`: fichero de texto con la salida generada por pantalla al ejecutar el fuente `p01` con los ficheros de una práctica particular.

  Si en la prueba no sale el mensaje "`Prueba p01: ok`" o algún otro de los comentados anteriormente, significa que hay diferencias en las salidas, por tanto se debe comprobar que diferencias puede haber entre los ficheros `p01.txt` y `p01.tmp`, o bien entre los ficheros donde el corrector haya indicado que hay diferencias. Para ello ejecutar en línea de comando, dentro del directorio `practica3-prueba`, la orden: `diff -w p01.txt p01.tmp`