# Práctica 3: **quién es quien**

Programación II

Mayo 2018

DLSI – Universidad de Alicante

versión: 10/05/2018

Alicia Garrido Alenda

## Normas generales

- El plazo de entrega para esta práctica se abrirá el lunes 14 de mayo a las 9:00 horas y se **cerrará el miércoles 23 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 Personaje.java Rasgo.java
     RasgoNoValidoException.java RasgoPreexistenteException.java
     PersonajeIncompletoException.java TableroIncompletoException.java
     PartidaGanadaException.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 de quién es quien. Para ello se implementarán las siguientes clases: **Rasgo**, **Personaje**, **Jugador**, **RasgoNoValidoException**, **RasgoPreexistenteException**, **PersonajeIncompletoException**, **TableroIncompletoException** 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 **Rasgo** se caracteriza por:

- *nombre*: indica el nombre del rasgo (`String`);
- *atributo*: indica la descripción del rasgo en caso de ser necesario (`String`);
- *tenencia*: booleano que indica si se tiene este rasgo (cierto) o no (falso) (`boolean`);

Cuando se crea un **Rasgo** se le pasa por parámetro una cadena para su nombre[1], no tiene descripción para su atributo y se indica que no se tiene el rasgo.

Las acciones que se pueden realizar con un **Rasgo** son las siguientes:

- *setAtributo*: se le pasa por parámetro una cadena para su atributo.
- *loTiene*: pone a cierto la tenencia del rasgo.
- *toString*: devuelve una cadena con la información del rasgo con el siguiente formato `nombre(atributo)`; por ejemplo `ojos(marron)`. En el caso de que el atributo sea `null`, el formato será `nombre(tenencia)`; por ejemplo `gafas(false)`.
- *getNombre*: devuelve su nombre.
- *getAtributo*: devuelve su atributo.
- *getTenencia*: devuelve su tenencia.

Un **Personaje** se caracteriza por:

- *rasgos*: array que contendrá los rasgos del personaje (`ArrayList<Rasgo>`);
- *nombre*: indica el nombre del personaje (`String`);

Cuando se crea un **Personaje** se le pasa por parámetro una cadena[2] para su nombre y crea su array *rasgos* sin ningún rasgo.

[1] Esta cadena nunca será `null` ni la cadena vacía.
[2] Esta cadena nunca será `null` ni la cadena vacía.

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

⊙ *agregaRasgo*: se le pasan por parámetro dos cadenas para crear un rasgo y añadirlo al final de sus rasgos. La primera cadena indica el nombre del rasgo, que sólo puede ser uno de los siguientes:

1. *pelo*
2. *ojos*
3. *gafas*
4. *pendientes*
5. *bigote*
6. *barba*
7. *sombrero*

Si no coincide exactamente con ninguno de estos nombres se lanza y propaga una excepción **RasgoNoValidoException** pasándole por parámetro el nombre de rasgo no válido al constructor de la excepción.

Si el nombre es válido, la segunda cadena puede ser el valor del atributo del rasgo en los siguientes casos:

- *pelo*: indica el color del pelo del personaje (si es `null` significa que es `calvo`). La tenencia tendrá valor `true`;
- *ojos*: indica el color de los ojos del personaje (si es `null` por defecto será `verde`). La tenencia tendrá valor `true`;

O bien el valor de la tenencia del rasgo[3]:

- *gafas*: indica si el personaje lleva gafas ("true") o no ("false"). El atributo tendrá valor `null`;
- *pendientes*: indica si el personaje lleva pendientes ("true") o no ("false"). El atributo tendrá valor `null`;
- *bigote*: indica si el personaje tiene bigote ("true") o no ("false"). El atributo tendrá valor `null`;
- *barba*: indica si el personaje tiene barba ("true") o no ("false"). El atributo tendrá valor `null`;
- *sombrero*: indica si el personaje lleva sombrero ("true") o no ("false"). El atributo tendrá valor `null`;

Si ya existe un rasgo con este nombre en el array de rasgos se lanza y propaga una excepción **RasgoPreexistenteException** pasándole por parámetro al constructor de la excepción dos cadenas: la primera el nombre del rasgo ya existente en rasgos y la segunda su atributo en caso de tratarse de los rasgos `ojos` o `pelo`, o su tenencia en forma de cadena si se trata de cualquier otro rasgo.

[3]Si es `null` significa que es "false".

- ⊙ *consulta*: se le pasa por parámetro un rasgo. Se busca en rasgos el rasgo cuyo nombre coincida exactamente que el pasado por parámetro y se comprueba si su atributo y tenencia tienen el mismo valor que el pasado por parámetro; en caso de ser así se devuelve cierto y en cualquier otro caso se devuelve falso.

- ⊙ *completo*: devuelve cierto si el personaje tiene todos los rasgos enumerados en *agregaRasgo*, y falso en cualquier otro caso.

- ⊙ *getNombre*: devuelve el nombre del personaje.

- ⊙ *getRasgos*: devuelve los rasgos del personaje.

- ⊙ *toString*: devuelve una cadena con la información del personaje con el siguiente formato `nombre[rasgo1,rasgo2,...]`. Por ejemplo:

```
Fran[ojos(marron),sombrero(true),pelo(rubio),gafas(false)]
```

Un **Jugador** se caracteriza por:

- ⋇ *tablero*: matriz de personajes donde se colocan los personajes del jugador (`Personaje[][]`);

- ⋇ *elegido*: personaje elegido del tablero para ser adivinado (`Personaje`);

- ⋇ *nombre*: nombre del jugador (`String`);

Inicialmente un **Jugador** se crea pasándole por parámetro una cadena[4] para su nombre y dos enteros para crear su *tablero*: el primer entero indica el número de filas de la matriz, que debe ser mayor que cero, en otro caso tendrá 2 filas. El segundo indica el número de columnas de la matriz, que debe ser mayor que cero, en otro caso tendrá 3 columnas.

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

- ⊙ *situaPersonaje*: se le pasan por parámetro dos enteros y un personaje. El personaje debe estar completo, con todos los rasgos posibles, en otro caso se lanza y propaga la excepción **PersonajeIncompletoException**, pasándole por parámetro al constructor de la excepción el nombre del personaje. El primer entero indica la fila de la matriz donde situar al personaje, y el segundo la columna. Dicha posición debe estar dentro del tablero y no contener ningún personaje previamente: en este caso se sitúa el personaje pasado por parámetro en esa posición y el método devuelve cierto. En cualquier otro caso el método devuelve falso.

- ⊙ *comienzaJuego*: se le pasan por parámetro dos enteros. Primero comprueba si tiene su tablero completo, es decir, tiene un personaje situado en cada posición del tablero. Si no es así se lanza y propaga la excepción **TableroIncompletoException**, pasándole por parámetro al constructor de la excepción el número de personajes que faltan para completar el tablero. A continuación comprueba si los dos enteros pasados por parámetro se corresponden con una posición válida del tablero, el primer entero para la fila y el segundo para la columna. Si no es así el método devuelve falso. Si tiene el tablero completo, la posición indicada por los dos enteros es una posición válida del tablero y no tenía ya un personaje elegido toma como personaje elegido el que ocupa en el tablero la posición indicada por los dos enteros y devuelve cierto.

[4]Esta cadena nunca será `null` ni la cadena vacía.

- ⊙ *pregunta*: se le pasa por parámetro un rasgo. Devuelve cierto si coincide con alguno de los rasgos del personaje elegido y falso en cualquier otro caso.

- ⊙ *consulta*: se le pasan por parámetro tres enteros. Los dos primeros indican una posición del tablero (fila,columna) que debe ser válida. Si la posición es válida, el tercer entero indica el rasgo que se está consultando (siguiendo la numeración indicada en *agregaRasgo*). El método devuelve el rasgo consultado del personaje que ocupa la posición indicada. Si por alguna circunstancia no se puede devolver un rasgo, el método devuelve `null`.

- ⊙ *elimina*: se le pasa por parámetro un rasgo de manera que se elimina del tablero aquellos personajes que no tengan este rasgo. Si en el tablero solo queda un personaje se lanza y propaga la excepción **PartidaGanadaException** pasándole al constructor de la excepción el nombre del jugador y el nombre del único personaje que queda en el tablero. En cualquier otro caso el método devuelve el número de personajes eliminados del tablero.

- ⊙ *muestraTablero*: muestra por pantalla el tablero. Para ello muestra el número de fila que muestra con el formato `fila iésima:` en una línea y a continuación los personajes contenidos en dicha fila del tablero, uno por línea con el formato indicado en el método *toString* del *Personaje*. Una posible salida de este método sería:

```
fila 0:
Fran[ojos(marron),sombrero(true),pelo(rubio),gafas(false)]
Megan[ojos(azul),sombrero(false),pelo(negro),gafas(true)]
Josh[ojos(verde),pendientes(true),pelo(blanco),bigote(true)]
fila 1:
Katie[ojos(marron),sombrero(false),pendientes(true),pelo(marron),gafas(false)]
Ana[ojos(azul),sombrero(false),pelo(blanco),gafas(true),pendientes(true)]
Hans[ojos(marron),sombrero(false),pelo(rubio,),gafas(false),bigote(true)]
```

- ⊙ *getNombre*: devuelve el nombre del jugador.

- ⊙ *getElegido*: devuelve el elegido del jugador.

La clase **Juego** es una aplicación en la que se desarrollará una partida de quien es quien 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 abra para lectura el fichero de texto que se le pasa como primer parámetro a la aplicación; el formato de este fichero será:

  - nombre del primer jugador en una línea;
  - nombre del segundo jugador en una línea;
  - una línea con dos enteros separados por un espacio en blanco que indican el tamaño del tablero de los jugadores;
  - una línea con la cadena "`personajes`";
  - a continuación vendrán los personajes del primer jugador. Por cada nuevo personaje aparece en el fichero una línea con la cadena "`personaje`", de manera que a partir de aquí hasta la siguiente línea "`personaje`"[5] está la descripción del personaje y su posición

[5] O cualquiera de las otras posibilidades del fichero: "`personajes`", "`partida`" o fin de fichero.

en el tablero del jugador, de manera que en la que en la primera línea estará la posición, en la segunda el nombre del personaje y en las sucesivas sus rasgos, uno por línea. Por ejemplo:

```
personaje
0 0
Fran
ojos marron
sombrero true
pelo rubio
gafas false
pendientes false
bigote false
barba true
```

- una línea con la cadena "`personajes`";
- a continuación vendrán los personajes del segundo jugador, de manera que seguirá el formato indicado anteriormente;
- una línea con la cadena "`partida`";
- una línea con dos enteros separados por un espacio en blanco que indican la posición en el tablero del personaje elegido del primer jugador;
- una línea con dos enteros separados por un espacio en blanco que indican la posición en el tablero del personaje elegido del segundo jugador;
- a partir de aquí están las distintas jugadas de ambos jugadores de manera que en cada línea habrá tres enteros, correspondientes a la fila y la columna de la posición en el tablero del personaje y su rasgo que se consulta;

⊙ 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 dos objetos de tipo ***`Jugador`*** con la información leída de las tres primeras líneas;

⊙ crear los personajes e invocar su método ***`situaPersonaje`*** con la información obtenida de procesar los datos leídos del fichero; 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 y su mensaje asociado, por ejemplo:

```
RasgoPreexistenteException: (sombrero,false)
```

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

⊙ cuando se lee del fichero la línea que contiene la cadena "`partida`" empieza la partida; con la información obtenida de procesar las dos siguientes líneas del fichero se invoca el método ***`comienzaJuego`*** de ambos jugadores; si alguno de los jugadores lanza la excepción ***`TableroIncompletoException`*** 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 y su mensaje asociado, y el tablero de ambos jugadores, primero el del primer jugador que se creó al leer el fichero, y a continuación el del segundo, separados por una línea en blanco, con el mismo formato indicado en el método muestraTablero y termina;

⊙ a partir de aquí empieza el juego; cada línea es la jugada de un jugador, de manera que empieza el primer jugador y continúa el segundo, luego vuelve a jugar el primero y así sucesivamente mientras queden jugadas por procesar en el fichero o se termine el juego. Para cada jugada se tiene que:

- invocar el método ***consulta*** del jugador con la información obtenida de procesar los datos leídos del fichero. Se escribe en el fichero que se ha abierto para escritura el rasgo que devuelve la consulta con el formato descrito en el método ***toString*** de ***Rasgo*** en una línea;
- invocar el método ***pregunta*** del otro jugador pasándole por parámetro el rasgo devuelto por la consulta. Se escribe en el fichero que se ha abierto para escritura el valor devuelto en una línea;
- invocar el método ***elimina*** del jugador si es oportuno, en función del valor devuelto por la pregunta.

⊙ si al invocar ***elimina*** se lanza la excepción ***PartidaGanadaException*** la partida acaba y escribe en el fichero que se ha abierto para escritura el nombre de la excepción, dos puntos y su mensaje asociado. Además también tiene que escribir en el fichero el tablero de ambos jugadores (primero el del primer jugador que se creó al leer el fichero, y a continuación el del segundo), separados por una línea en blanco, con el mismo formato indicado en el método ***muestraTablero***, por ejemplo:

```
PartidaGanadaException: (Hugo,Fran)
fila 0:
Fran[ojos(marron),sombrero(true),pelo(rubio),gafas(false),pendientes(false),
bigote(false),barba(true)]
fila 1:

fila 0:
Megan[ojos(azul),sombrero(false),pelo(negro),gafas(true),pendientes(true),
bigote(false),barba(false)]
Josh[ojos(verde),pendientes(true),pelo(blanco),bigote(true),sombrero(true),
barba(false),gafas(true)]
fila 1:
Ana[ojos(azul),sombrero(false),pelo(blanco),gafas(true),pendientes(true),barba(false),
bigote(false)]
Katie[ojos(marron),sombrero(false),pendientes(true),pelo(marron),gafas(false),
barba(false),bigote(false)]
```

⊙ si no se lanza ninguna excepción que acabe la partida pero no quedan más datos que procesar en el fichero de lectura, la partida acaba y se tienen que escribir en el fichero de escritura el tablero de ambos jugadores (primero el del primer jugador que se creó al leer el fichero, y a continuación el del segundo), separados por una línea en blanco, con el mismo formato indicado en el método muestraTablero;

⊙ por último la aplicación debe cerrar ambos ficheros, el de lectura y el de escritura;

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

```
Hugo
Cinthia
2 3
personajes
personaje
0 0
Fran
ojos marron
sombrero true
pelo rubio
gafas false
pendientes false
bigote false
barba true
personaje
0 2
Megan
ojos azul
sombrero false
pelo negro
gafas true
pendientes true
bigote false
barba false
personaje
0 1
Hans
ojos marron
sombrero false
pelo rubio
gafas false
bigote true
barba false
pendientes false
personaje
1 0
Katie
ojos marron
sombrero false
pendientes true
pelo marron
gafas false
barba false
bigote false
personaje
1 1
Josh
ojos verde
pendientes true
pelo blanco
```

```
bigote true
sombrero true
barba false
gafas true
personaje
1 2
Ana
ojos azul
sombrero false
pelo blanco
gafas true
pendientes true
barba false
bigote false
personajes
personaje
0 0
Megan
ojos azul
sombrero false
pelo negro
gafas true
pendientes true
bigote false
barba false
personaje
0 1
Fran
ojos marron
sombrero true
pelo rubio
gafas false
pendientes false
bigote false
barba true
personaje
0 2
Josh
ojos verde
pendientes true
pelo blanco
bigote true
sombrero true
barba false
gafas true
personaje
1 0
Hans
ojos marron
sombrero false
```

```
pelo rubio
gafas false
bigote true
barba false
pendientes false
personaje
1 1
Ana
ojos azul
sombrero false
pelo blanco
gafas true
pendientes true
barba false
bigote false
personaje
1 2
Katie
ojos marron
sombrero false
pendientes true
pelo marron
gafas false
barba false
bigote false
partida
1 2
0 1
1 0 3
1 1 4
0 1 1
```

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

```
gafas(false)
true
pendientes(true)
true
pelo(rubio)
true
fila 0:
Fran[ojos(marron),sombrero(true),pelo(rubio),gafas(false),pendientes(false),bigote(false),
barba(true)]
Hans[ojos(marron),sombrero(false),pelo(rubio),gafas(false),bigote(true),barba(false),
pendientes(false)]
fila 1:

fila 0:
Megan[ojos(azul),sombrero(false),pelo(negro),gafas(true),pendientes(true),bigote(false),
barba(false)]
Josh[ojos(verde),pendientes(true),pelo(blanco),bigote(true),sombrero(true),barba(false),
```

```
gafas(true)]
fila 1:
Ana[ojos(azul),sombrero(false),pelo(blanco),gafas(true),pendientes(true),barba(false),
bigote(false)]
Katie[ojos(marron),sombrero(false),pendientes(true),pelo(marron),gafas(false),barba(false),
bigote(false)]
```

Las clases **RasgoNoValidoException**, **RasgoPreexistenteException**, **PersonajeIncompletoException**, **TableroIncompletoException** y **PartidaGanadaException** heredan de la clase *`Exception`* de java.

Los constructores de las clases **RasgoPreexistenteException** y **PartidaGanadaException** reciben por parámetro dos cadenas. El mensaje que se tiene que devolver al invocar el método *`getMessage`* es "`(s1,s2)`", donde `s1` sería el valor de la primera cadena y `s2` el valor de la segunda cadena que se le pasan al constructor por parámetro.

Los constructores de las clases **RasgoNoValidoException** y **PersonajeIncompletoException** reciben por parámetro un `String`. El mensaje que se tiene que devolver al invocar el método *`getMessage`* es el `String` que se le pasa al constructor por parámetro.

El constructor de la clase **TableroIncompletoException** recibe por parámetro un entero. El mensaje que se tiene que devolver al invocar el método *`getMessage`* es el entero que se le pasa al constructor por parámetro convertido a cadena.

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 **Rasgo**
  - `public Rasgo(String s)`
  - `public void setAtributo(String s)`
  - `public void loTiene()`
  - `public String getNombre()`
  - `public String toString()`
  - `public String getAtributo()`
  - `public boolean getTenencia()`
- Clase **Personaje**
  - `public Personaje(String s)`
  - `public void agregaRasgo(String s1,String s2) throws RasgoNoValidoException, RasgoPreexistenteException`
  - `public boolean consulta(Rasgo r)`
  - `public boolean completo()`
  - `public String getNombre()`

  - `public ArrayList<Rasgo> getRasgos()`
  - `public String toString()`
- Clase **Jugador**
  - `public Jugador(String n,int f,int c)`
  - `public boolean situaPersonaje(int i,int j,Personaje p) throws PersonajeIncompletoException`
  - `public boolean comienzaJuego(int i,int j) throws TableroIncompletoException`
  - `public boolean pregunta(Rasgo r)`
  - `public Rasgo consulta(int i,int j,int k)`
  - `public int elimina(Rasgo r) throws PartidaGanadaException`
  - `public void muestraTablero()`
  - `public String getNombre()`
  - `public Personaje getElegido()`
- Clase **RasgoNoValidoException**
  - `public RasgoNoValidoException(String s)`
- Clase **RasgoPreexistenteException**
  - `public RasgoPreexistenteException(String s1,String s2)`
- Clase **PersonajeIncompletoException**
  - `public PersonajeIncompletoException(String n)`
- Clase **TableroIncompletoException**
  - `public TableroIncompletoException(int i)`
- Clase **PartidaGanadaException**
  - `public PartidaGanadaException(String s1,String s2)`
- 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 1`" de la cual queremos obtener por separado los tres datos que contiene para invocar el método *`consulta`* de un jugador. 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 f1 = Integer.parseInt(elems[0]);
 int c1 = Integer.parseInt(elems[1]);
 int r = Integer.parseInt(elems[2]);
// con estos datos ya se puede invocar el metodo
 Rasgo carac=jugon.consulta(f1,c1,r);
```

# 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 `p01.java` y `p02.java`, programas en Java con un método `main` que realizan una serie de pruebas sobre la práctica, y los ficheros `p01.txt` y `p02.txt` con la salida correcta a las pruebas realizadas en `p01.java` y `p02.java`, respectivamente.
- 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 salen los mensajes "`Prueba p01: ok`" y "`Prueba p02: 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 `p02.txt` y `p02.tmp`, en función de 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`