Práctica 1: **Clases, objetos, relaciones entre objetos**

Programación II

Marzo 2020 (versión 09/02/2020)

DLSI – Universidad de Alicante

Alicia Garrido Alenda

Normas generales

- El plazo de entrega para esta práctica se abrirá el lunes 2 de marzo a las 9:00 horas y se **cerrará el viernes 6 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

El objetivo de esta práctica es aprender a implementar un programa usando el paradigma de programación orientado a objetos. Para ello en esta práctica se pide implementar una serie de clases, de las cuales se crearán objetos que se relacionan entre sí de diferentes formas, para comprender de forma práctica el funcionamiento de este paradigma de programación.

En concreto se tiene que crear un tipo enumerado llamado **Tipo** que indicará los tipos de objetos que puede tener una persona y el valor emocional que le aportan; una clase llamada **Objeto** donde se definen las características que tiene un objeto, además de su tipo, y qué acciones pueden llevar a cabo; una clase **Persona** donde se definen las características que tendrá una persona en esta práctica, entre ellas objetos de la clase *`Objeto`* anterior, y las acciones que puede realizar; una clase **Tienda** que tendrá sus características y acciones, y cuyos objetos se relacionan con objetos de las clases anteriores de diversas formas. Por último también se implementará una clase **Desarrollo** que contendrá un `main`, en el que habrá que crear objetos de estas clases y hacer que interaccionen entre sí.

A continuación se definen más concretamente las clases que es necesario implementar para esta práctica, 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).

ACLARACIÓN: a lo largo de todo el enunciado de la práctica, cuando sea necesario comparar cadenas, se deben ignorar las diferencias entre mayúsculas y minúsculas.

En **Tipo** se establece el tipo de objetos que pueden tener las personas, junto con su valor emocional, por tanto será un tipo enumerado en el que se definirán los tipos de objetos junto con su valor:

RELICARIO	`4.5`
TARJETA	`3.2`
COCHE	`9.0`
CASA	`10`
ROPA	`6.8`
ALIMENTO	`5.5`
ACCESORIO	`7`
RECUERDO	`8`
DINERO	`10`
MASCOTA	`5`

El enumerado **Tipo** tiene una variable de instancia:

- *`valor`*: indica el valor emocional que aporta (`double`);

y los siguientes métodos:

- *`Tipo`*: constructor al que se le pasa por parámetro un número real para asignar a su variable de instancia.
- *`getValor`*: devuelve el valor del tipo concreto.

La clase **Objeto** tiene las siguientes variables de instancia:

- ***tipo***: indica el tipo de objeto (`Tipo`);
- ***nombre***: indica el nombre del objeto (`String`);
- ***influencia***: indica la influencia del valor del objeto (`int`);
- ***poseedor***: indica el nombre de su dueño (`String`); si un objeto no tiene poseedor su valor es `null`.

Y los siguientes métodos:

- constructor: se le pasa por parámetro un tipo, una cadena y un entero que se asignan a sus variables de instancia ***tipo***, ***nombre*** e ***influencia*** respectivamente. Si el tipo pasado por parámetro es `null`, el tipo del objeto será `TARJETA`. Si la cadena pasada por parámetro fuera la cadena vacía o `null`, el nombre del objeto será la cadena "`intrascendente`". Inicialmente un objeto no tiene poseedor.
- ***getTipo***: devuelve el nombre de tipo de objeto que es.
- ***getNombre***: devuelve el nombre del objeto.
- ***getValorIntrinseco***: devuelve el valor del objeto.
- ***getInfluencia***: devuelve la influencia del objeto.
- ***getPoseedor***: devuelve la variable ***poseedor***.
- ***setPoseedor***: se le pasa por parámetro un `String` que se asigna a la variable ***poseedor***.
- ***calculaValorEmocional***: calcula y devuelve el valor emocional del objeto. Para ello aplica la fórmula:

$$valorEmocional = \frac{valor * influencia}{100}$$

- ***getDescripcion***: devuelve en un array dinámico de cadenas todas sus características en el siguiente orden: nombre del tipo del objeto, valor del tipo del objeto, nombre del objeto, influencia del objeto.
- ***cambiaInfluencia***: se le pasa por parámetro un entero que debe sumarse al valor actual de la influencia. El método devuelve cierto si el valor final de la influencia se ha incrementado, y falso en caso contrario.

La clase **Persona** tiene las siguientes variables de instancia:

- ***edad***: número real que indica la edad de la persona (`double`);
- ***nombre***: cadena que indica el nombre de la persona (`String`);
- ***genes***: variable que indica si es hombre o mujer, de manera que si es `true` significa que la persona es mujer y en caso de ser `false` indica que es hombre (`boolean`);

- ***estado***: estado anímico de la persona (`double`);
- ***pertenencias***: array dinámico que contiene los objetos que posee actualmente la persona (`ArrayList<Objeto>`);

Las siguientes variables de clase:

- ***Nombres***: tipo enumerado que contendrá los siguientes nombres: `ALEX,MATY,CAS,BLOOM,LAN,JD,SAM,AL,MANI,BEL`;
- ***Edades***: tipo enumerado que contendrá las siguientes edades con su valor asociado:

ADOLESCENTE	16
JOVEN	19.5
TRABAJADOR	30
ESCOLAR	9.5
EMPRENDEDOR	21
JUBILADO	76
ADULTO	38
MADURO	55
MAYOR	64.5
INFANTIL	3

Y los siguientes métodos:

- constructor: se le pasa por parámetro un número real, una cadena y un booleano que se asignan a sus variables de instancia ***edad***, ***nombre*** y ***genes*** respectivamente. Si el número real es menor o igual que `0` o mayor que `99`, se obtiene el valor absoluto de calcular el módulo entre la parte entera del número real pasado por parámetro y `10`, este valor indica la posición de ***Edades*** que se debe asignar a la variable de instancia ***edad*** en este caso. Si la cadena pasada por parámetro es la cadena vacía o `null`, se obtiene el módulo entre la parte entera de la variable de instancia ***edad*** y `10`, este valor indica la posición de ***Nombres*** que se debe asignar a la variable de instancia ***nombre*** en este caso. Inicialmente el estado anímico valdrá `5.0` y se crea el array `pertenencias`.
- ***getNombre***: devuelve el nombre de la persona.
- ***getEdad***: devuelve la edad de la persona.
- ***getEstado***: actualiza el estado anímico de la persona y lo devuelve.
- ***getGenes***: devuelve el carácter '`M`' si se trata de una mujer, o el carácter '`V`' si se trata de un hombre.
- ***getPertenencias***: devuelve un array con todos los objetos que tiene la persona. Si no tiene ningún objeto devuelve `null`.
- ***actualizaEstado***: se recorre las pertenencias de la persona, obteniendo el valor emocional de cada objeto que se suma al estado anímico de la persona.
- ***getNombres***: devuelve un array con todos los nombres del tipo enumerado ***Nombres***.

⊙ *getEdades*: devuelve un array con todos los valores de las edades del tipo enumerado *Edades*.

⊙ *encuentra*: se le pasa por parámetro un objeto que la persona se ha encontrado. Si dicho objeto no tiene poseedor, la persona se lo queda, añadiéndolo al final de sus pertenencias, de manera que el objeto ahora sí tiene poseedor y el método devuelve cierto. En cualquier otro caso, el método devuelve falso.

⊙ *busca*: se le pasa por parámetro una cadena. El método devuelve el primer objeto que encuentra que tenga el mismo nombre (literalmente) entre sus pertenencias. Si no encuentra ninguno devuelve `null`.

⊙ *intercambio*: se le pasa por parámetro una persona, con la que puede intercambiar un objeto, y dos cadenas. Si la propia persona tiene un objeto entre sus pertenencias cuyo nombre coincida literalmente con la primera cadena, y la persona pasada por parámetro tiene a su vez entre las suyas un objeto cuyo nombre coincida literalmente con la segunda cadena, se intercambian dichos objetos[1], y el método devuelve cierto. En cualquier otro caso, devuelve falso.

⊙ *adquiere*: se le pasa por parámetro una tienda y un tipo de objeto. La persona desea adquirir de la tienda el objeto de ese tipo con menor coste. El coste de un objeto para una persona es el valor absoluto de restarle al valor intrínseco del objeto, el primer dígito de la edad de la persona. Para poder realizar la transacción la tienda tiene que disponer de objetos de ese tipo y la persona tiene que poderlo pagar. Para poder pagar el coste del objeto la persona debe tener un estado anímico superior al coste del objeto. Si se realiza finalmente la adquisión la persona decrementa su estado anímico con el coste del objeto, que añade al final de sus pertenencias, la tienda elimina de sus existencias dicho objeto e incrementa con el coste del producto sus ganancias. Si se realiza la adquisición el método devuelve cierto. En cualquier otro caso el método devuelve falso.

⊙ *pierde*: se le pasa por parámetro un entero, que indica la posición de sus pertenencias que pierde, y que por tanto se elimina de ellas. El método devuelve el valor emocional del objeto perdido. Si no pierde ningún objeto devuelve `0`.

La clase **Tienda** tiene las siguientes variables de instancia:

⁕ *existencias*: matriz donde estarán los objetos que tiene la tienda (`Objeto[][]`);

⁕ *ganancias*: cantidad que indica las ganancias de la tienda (`double`);

⁕ *negocios*: array dinámico con todas las tiendas creadas (`static ArrayList<Tienda>`).

Y los siguientes métodos:

⊙ constructor: se le pasa por parámetro dos enteros y un número real. Crea la matriz existencias con el número de filas indicado por el primer entero, si es mayor que `0`, o con `3` filas en otro caso, y el número de columnas indicado por el segundo parámetro, si es mayor que `0`, o con `4` columnas en otro caso. El número real se asigna a la variable de instancia `ganancias` si es mayor que `0`, en otro caso se asigna `0`. Se añade la tienda creada al final del array *negocios*.

⊙ getGanancias: devuelve las ganancias.

[1]Es decir, los objetos intercambiados ocuparán la misma posición que ocupaba el objeto por el cual se cambia, y se actualizan los nombres de sus poseedores.

- ⊙ setGanancias: se le pasa por parámetro un número real que, si es mayor que `0`, suma a sus ganancias.
- ⊙ getExistencias: devuelve la matriz de existencias.
- ⊙ getGananciaTotal: devuelve la suma de las ganancias de todas las tiendas existentes.
- ⊙ getObjetos: se le pasa por parámetro un tipo de objeto y devuelve un array dinámico con todos los objetos de ese tipo que hay en las existencias de la tienda en el orden en que los encuentra (recorriendo la matriz por filas). Si no encuentra ninguno devuelve `null`.
- ⊙ *`inventario`*: devuelve un array de enteros, en el que cada posición contiene la cantidad de objetos de ese tipo que tiene la tienda. La posición coincide con el orden definido para los tipos en el enumerado *`Tipo`*. Así por ejemplo, la posición `0` se corresponde con los objetos de tipo `RELICARIO`.
- ⊙ *`almacena`*: se le pasa por parámetro un objeto y dos enteros, que indican la fila y columna donde almacenar el objeto en la matriz. Si dicha posición ya está ocupada por otro objeto, se busca la primera posición libre de la matriz (`null`) desde esa posición, recorriendo por filas la matriz[2]. El método devuelve una cadena con la posición en la que finalmente se almacena el objeto con el formato `(fila,columna)`, o `null` si no se consigue almacenar el objeto.
- ⊙ *`desinventaria`*: se le pasa por parámetro un tipo de objeto, de manera que elimina de sus existencias todos los objetos de ese tipo que encuentra, devolviéndolos en un array en el orden en que los encuentra (recorriendo la matriz por filas). Si no tiene ningún objeto de ese tipo, devuelve `null`.
- ⊙ *`elimina`*: se le pasan dos enteros y devuelve el objeto que ocupa esa posición en la matriz, eliminándolo de ella, donde el primer entero indica la fila y el segundo la columna. Por defecto devuelve `null`.
- ⊙ *`busca`*: se le pasa por parámetro un `String` que es la descripción de un objeto[3]. El método devuelve cierto si encuentra algún objeto con la misma descripción y falso en cualquier otro caso.

La clase **Desarrollo** no tiene variables de instancia, y tiene un sólo método obligatorio, el `main`, en el que se tiene que:

- ⊙ crear un array dinámico de *`Objeto`*, otro de *`Persona`* y otro de *`Tienda`*;
- ⊙ crear un objeto de tipo `Scanner` para leer de la entrada estándar, de la siguiente manera:

```
Scanner scan=new Scanner(System.in);
```

- ⊙ leer por líneas mientras queden datos por leer, procesando cada línea de la siguiente manera:

[2]Por ejemplo, para una matriz de `3x3`, si se intenta almacenar en la posición `(1,1)` y está ocupada, se mirará en las posiciones `(1,2)`, `(2,0)`, `(2,1)` y `(2,2)` en este orden.

[3]La concatenación de todas sus características, separadas por un espacio en blanco, en el siguiente orden: nombre del tipo del objeto, valor del tipo del objeto, nombre del objeto, influencia del objeto.

1. la primera línea contiene un entero que indica la opción de ejecución del programa, que se comenta más adelante;
2. para el resto de líneas que se leen, aplicar el método `split` a la línea leída usando como separador el ";". El primer elemento del array de cadenas que devuelve este método indica el tipo de objeto que se tiene que crear:
   - Objeto: se tiene que crear un objeto de tipo *`Objeto`*. Los parámetros para su constructor son los demás elementos del array, convirtiendo el último en un entero. Una vez creado se tiene que añadir al array dinámico de objetos creado al principio.
   - Persona: se tiene que crear un objeto de tipo *`Persona`*. Los parámetros para su constructor son los demás elementos del array, convirtiendo el segundo elemento en un número real y el último en un booleano. Una vez creado se tiene que añadir al array dinámico de personas creado al principio.
   - Tienda: se tiene que crear un objeto de tipo *`Tienda`*. Los parámetros para su constructor son los demás elementos del array, convirtiendo el segundo y tercer elemento en números enteros y el último en un número real. Una vez creado se tiene que añadir al array dinámico de tiendas creado al principio.

⊙ una vez se han leído todos los datos y creado todos los objetos, se tiene que:

1. si la opción de ejecución es `1`, se deben ordenar todos los objetos de tipo **Objeto** lexicográficamente de menor a mayor por su nombre[4], y mostrar por pantalla, uno por línea, la descripción de cada objeto, separando los elementos por un espacio en blanco;
2. si la opción de ejecución es `2`, se deben ordenar todos los objetos de tipo **Persona** de mayor a menor por su edad[5], y mostrar por pantalla, uno por línea, los siguientes datos de cada objeto separados por un espacio en blanco: el nombre de la persona, su edad, sus genes y su estado;
3. si la opción de ejecución es `3`, se deben ordenar todos los objetos de tipo **Tienda** de menor a mayor por su capacidad de almacenamiento[6], es decir, por la cantidad de objetos que puede almacenar en sus existencias, y mostrar por pantalla, uno por línea, los siguientes datos de cada objeto separados por un espacio en blanco: capacidad de almacenamiento y ganancias;
4. para cualquier otra opción se tienen que realizar las tres opciones anteriores en el orden en el que están definidas.

▪ Por ejemplo:

datos entrada	salida correspondiente

`2`
`Persona;15.7;Pepito;false`
`Objeto;COCHE;panda;6`
`Persona;21.6;Josefa;true`
`Tienda;5;5;70`
	`Josefa 21.6 M 5.0`
	`Pepito 15.7 V 5.0`

[4]No habrá elementos repetidos.
[5]No habrá elementos repetidos.
[6]No habrá elementos repetidos.

Restricciones en la implementación

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

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

- En **Tipo**
  - `public double getValor()`
- En **Objeto**
  - `public Objeto(Tipo,String,int)`
  - `public String getTipo()`
  - `public String getNombre()`
  - `public double getValorIntrinseco()`
  - `public int getInfluencia()`
  - `public String getPoseedor()`
  - `public void setPoseedor(String)`
  - `public double calculaValorEmocional()`
  - `public ArrayList<String> getDescripcion()`
  - `public boolean cambiaInfluencia(int)`
- En **Persona**
  - `public Persona(double,String,boolean)`
  - `public String getNombre()`
  - `public double getEdad()`
  - `public double getEstado()`
  - `public char getGenes()`
  - `public Objeto[] getPertenencias()`
  - `public void actualizaEstado()`
  - `public static String[] getNombres()`
  - `public static double[] getEdades()`
  - `public boolean encuentra(Objeto)`
  - `public Objeto busca(String)`
  - `public boolean intercambio(Persona,String,String)`
  - `public boolean adquiere(Tienda,Tipo)`
  - `public double pierde(int)`
- En **Tienda**
  - `public Tienda(int,int,double)`
  - `public double getGanancias()`
  - `public void setGanancias(double)`
  - `public Objeto[][] getExistencias()`
  - `public double getGananciaTotal()`
  - `public ArrayList<Objeto> getObjetos(Tipo)`
  - `public int[] inventario()`
  - `public String almacena(Objeto,int,int)`

    - `public Objeto[] desinventaria(Tipo)`
    - `public Objeto elimina(int,int)`
    - `public boolean busca(String)`
  - En **Desarrollo**
    - `public static void main(String[] args)`

⊛ Solo el fichero correspondiente a la clase **Desarrollo** puede contener un método `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.

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 las cadenas, Java dispone del tipo ***String***, que tiene métodos implementados para realizar operaciones con dicho tipo. Algunos de ellos son:

- `int compareTo(String cadena)`: compara dos cadenas lexicográficamente. Devuelve `0` si son iguales, un número mayor que `0` si el parámetro es menor (precede) a la cadena con la cual se invoca el método y un número negativo en otro caso. Se invoca:

  ```
  String cadena1=new String(‘‘perro’’);
  String cadena2=new String(‘‘mesa’’);
  int n=cadena1.compareTo(cadena2);
  ```

  donde si `n` es mayor que `0` significa que `cadena2` precede lexicográficamente a `cadena1`.

- `int compareToIgnoreCase(String cadena)`: compara dos cadenas lexicográficamente ignorando las diferencias debidas a mayúsculas y minúsculas. Devuelve `0` si son iguales, un número mayor que `0` si el parámetro es menor (precede) a la cadena con la cual se invoca el método y un número negativo en otro caso. Se invoca:

  ```
  String cadena1=new String(‘‘perro’’);
  String cadena2=new String(‘‘mesa’’);
  int n=cadena1.compareToIgnoreCase(cadena2);
  ```

  donde si `n` es mayor que `0` significa que `cadena2` precede lexicográficamente a `cadena1`.

- `boolean equals(Object cadena)`: compara la cadena con la cual se invoca el método con el objeto que se pasa por parámetro, devolviendo cierto si y solo si el parámetro no es `null`, es un `String` y contiene la misma secuencia de caracteres que la cadena con la cual se invocó el método. Se invoca:

  ```
  boolean bool=cadena1.equals(cadena2);
  ```

- `boolean equalsIgnoreCase(Object cadena)`: compara la cadena con la cual se invoca el método con el objeto que se pasa por parámetro, devolviendo cierto si y solo si el parámetro no es `null`, es un `String` y contiene la misma secuencia de carácteres que la cadena con la cual se invocó el método ignorando las diferencias debidas a mayúsculas y minúsculas. Se invoca:

  ```
  boolean bool=cadena1.equalsIgnoreCase(cadena2);
  ```

- `String[] split(String s)`: devuelve un array de `String` resultado de separar el `String` sobre el que se invoca en tantos `String` como veces contenga el `String` pasado por parámetro, es decir, utiliza el `String` pasado por parámetro como separador. Se invoca:

  ```
  String a=new String(‘‘abc:def:ghi:jkl’’);
  String[] b=a.split(‘‘:’’); //b es un array de 4 posiciones
  // b -> {‘‘abc’’,‘‘def’’,‘‘ghi’’,‘‘jkl’’}
  ```

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
```

Para leer de la entrada estándar de forma sencilla en Java, se puede utilizar la clase `Scanner`. Un ejemplo de lectura utilizando esta clase es el siguiente, en que el se lee por líneas mientras queden datos por leer:

```
import java.util.*;
public class Lectura{
  public static void main(String[] args){
   Scanner scan=new Scanner(System.in);
   String line=null;
   int c=0;
   while(scan.hasNext()){ // consulta si quedan datos por leer
     line=scan.nextLine(); // lee una linea y la guarda en la variable line
     System.out.println(c+" -> "+line); // procesa la variable line
     c++;
   }
  }
}
```

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 **a partir del 24 de febrero**. 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 -b p01.txt p01.tmp`