Práctica 2: **Herencia**

Programación II

Abril 2021 (versión 21/03/2021)

DLSI – Universidad de Alicante

Alicia Garrido Alenda

Normas generales

- El plazo de entrega para esta práctica se abrirá el lunes 26 de abril a las 9:00 horas y se **cerrará el viernes 30 de abril 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  practica2.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, 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`** 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 asentar los conocimientos propios del paradigma de programación orientado a objetos relativos a la herencia. Para ello en esta práctica se pide implementar una serie de clases, algunas de ellas relacionadas con las implementadas en la práctica anterior, de las cuales se crearán objetos que se relacionan entre sí de diferentes formas, para comprender de forma práctica el funcionamiento de la herencia en programación.

La sociedad ha evolucionado debido a una serie de cambios medioambientales. Dichos cambios han producido modificaciones en las poblaciones de los fundadores y de los radicales, dando lugar a cambios en los mismos, de manera que los fundadores han evolucionado a **Precursor** y los radicales a **Emprendedor**.

El cambio más notorio que se aprecia en los precursores es que han desarrollado inmunidad a los radicales, de manera que cuando un radical ataca a un precursor, el precursor no pierde su fortaleza, y el radical atacante resulta perjudicado, aunque solo de forma residual.

Por su parte los emprendedores, ante la imposibilidad de obtener la fortaleza de los precursores mediante ataques, han desarrollado la capacidad de almacenar artículos para poder comerciar con los precursores y así obtener pacíficamente la fuerza que necesitan.

Para llevar a cabo los intercambios comerciales entre precursores y emprendedores se han construído unas edificaciones llamadas **Neutral** a las que sólo pueden entrar los individuos evolucionados, los **Converso**, de manera que un emprendedor y un precursor, que son conversos, sólo pueden realizar intercambios comerciales cuando se encuentran dentro del mismo edificio neutral. Obviamente ni un precursor ni un emprendedor pueden estar al mismo tiempo en dos edificios **Neutral** diferentes.

Para esta práctica son necesarias todas las clases de la práctica anterior (**Fundador**, **Radical** y **Articulo**) además de las clases que se definen a continuación y 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).

La **Ubicacion** indica la situación de una localidad dentro de un *Neutral*. Sus variables de instancia son:

- *grada*: número que indica la fila de la localidad dentro de la matriz (`int`);
- *asiento*: número que indica la columna dentro de la fila de la matriz (`int`).

Y los siguientes métodos:

- constructor: se le pasan por parámetro dos enteros; el primero indica el número de grada mientras que el segundo indica el número de asiento.
- *getGrada*: devuelve el valor de *grada*.
- *getAsiento*: devuelve el valor de *asiento*.

Los **Neutral** son edificios en los que se reúnen los *Converso* para hacer negocios. Sus variables de instancia son:

- *nombre*: cadena que indica el nombre del edificio (`String`);

- ⁕ *gradas*: son las localidades que ocupan los converso cuando entran en el edificio (`Converso[][]`).
- ⁕ *caja*: cantidad de dinero efectivo que hay en el edificio (`double`).
- ⁕ *conserje*: converso que se encarga de algunas gestiones del edificio (`Converso`).

Y sus métodos son:

- ⊙ contructor: se le pasan por parámetro dos enteros, un número real y una cadena; el primero es el número de gradas (número de filas de la matriz), el segundo el número de asientos que tendrá cada grada (número de columnas), ambos enteros deben ser mayores que `0`[1]. El tercer parámetro se corresponde con el valor inicial de su *caja*[2]. Por último el cuarto parámetro se corresponde con su *nombre*[3]. Inicialmente no hay conserje.
- ⊙ *contrata*: se le pasa por parámetro un converso. Las restricciones para poder contratarlo son:
  - En caso de ser un precursor, no puede ser el sanador.
  - No puede ser el conserje de otro neutral.
  - El puesto de conserje debe estar libre.

  Si se cumplen todas estas restricciones, el converso se convierte en el conserje del edificio, le pasa un mensaje de contratado y el método devuelve cierto. En cualquier otro caso devuelve falso.
- ⊙ *despide*: con esta acción se despide al converso que ocupa el puesto de conserje, de manera que el puesto queda libre.
- ⊙ *busquedaIzquierda*: busca la primera localidad disponible en la mitad izquierda de las gradas, devolviendo un objeto de tipo *Ubicacion* con su localización si la encuentra. Para buscar en la mitad izquierda de las gradas se aplica la fórmula:

$$centro = (columnas/2) + (columnas\,\%2)$$

  donde $columnas$ es el número de asientos que tiene cada grada del Neutral. La búsqueda se realizará desde la grada 0 hasta la última grada y de izquierda a derecha, pero recorriendo sólo hasta llegar a la columna $centro$ (inclusive).
- ⊙ *busquedaDerecha*: busca la primera localidad disponible en la mitad derecha de las gradas, devolviendo un objeto de tipo *Ubicacion* con su localización si la encuentra. Para buscar en la mitad derecha de las gradas se aplica la fórmula:

$$centro = (columnas/2) + (columnas\,\%2)$$

  donde $columnas$ es el número de asientos que tiene cada grada del Neutral, de manera que la búsqueda se realizará desde la grada 0 hasta la última grada, pero empezando el recorrido de cada grada en la columna $centro$.

[1] El valor por defecto en ambos casos será `2`.
[2] Este valor debe ser mayor que `0`, en otro caso tomará el valor inicial de `300.0`
[3] Si la cadena pasada por parámetro es `null`, el nombre por defecto será "`Aoyama`".

⊙ `ventaDeEntrada`: se le pasa por parámetro el converso que quiere ocupar una de las localidades, la ubicación de la localidad que desea ocupar y el importe que se paga por dicha localidad. Para poder vender entradas el edificio tiene que tener conserje, y en ese caso el método devolverá lo que sobre del importe después de cobrar el coste de la entrada, que sumará a su caja si finalmente el converso en cuestión ocupa la localidad solicitada. Si la venta tiene éxito, se le pasa al conserje un comunicado con el converso que ha comprado la entrada. Por cada comunicado, el conserje cobra al edificio un importe de `0.5`.

Para que un converso pueda ocupar una localidad se tienen que dar las siguientes circunstancias:

- la localidad solicitada debe estar libre (no ocupada por otro converso);
- el converso solicitante no puede estar en otro neutral;
- el importe debe ser mayor o igual que el coste de la entrada;
- el neutral debe poder pagar al conserje antes de cobrar la entrada.

El coste de la entrada[4] de una localidad es:

$$coste = \frac{fila * columna}{ocupantes + 1}$$

donde $fila$ y $columna$ es la ubicación de la localidad que se desea ocupar y $ocupantes$ es el número de conversos que hay en ese momento dentro del edificio.

Si finalmente el converso no ocupa ninguna localidad el método devuelve `-1`.

⊙ `sale`: se le pasa por parámetro el converso que quiere salir del edificio. Si el edificio tiene conserje, puede pagarle por sus servicios (`0.5` por comunicado) y el converso se encuentra en una de sus localidades, ésta quedará vacante. El conserje pasa un mensaje al converso para que salga del neutral y se devuelve la ubicación que ha quedado libre. En otro caso devolverá `null`.

⊙ `localizacion`: se le pasa por parámetro un converso, y si se encuentra dentro del edificio se devuelve su ubicación. En cualquier otro caso, se devuelve `null`.

⊙ `loteria`: con esta acción se quiere premiar a los ocupantes actuales del edificio. Para ello calcula la media de la fortaleza de todos los precursores presentes, la media de la fuerza de todos los emprendedores presentes, las suma y divide entre tres. La cantidad resultante de esta operación es el premio que recibirá cada uno de los conversos presentes en el edificio si se dispone de suficiente caja para repartir el mismo premio a todos. Si no se dispone de suficiente caja para premiar a todos los presentes, solo se premia a los ocupantes de la grada 0. Si aún así no se puede premiar a todos los asistentes de esa grada, no se premia a ningún asistente. Los premios se comunican mediante mensajes `winner` a los agraciados con la cantidad del premio. El método decrementa la `caja` con la cantidad repartida en premios y devuelve el número de conversos agraciados.

⊙ `getLibres`: devuelve el número de localidades vacías que hay en ese momento en el edificio.

⊙ `getTotal`: devuelve el número de localidades que tiene el edificio.

⊙ `getNombre`: devuelve el nombre del edificio.

⊙ `getCaja`: devuelve el valor de la caja del edificio.

[4] Es un número real.

- ⊙ *getConserje*: devuelve el conserje del edificio.

Los **Converso** pueden ser de dos tipos; o bien de tipo **Precursor**, o bien de tipo **Emprendedor**. Ambos pueden:

- ⊙ *contratado*: se le pasa por parámetro el neutral donde comienza a trabajar de conserje.
- ⊙ *despedido*: el converso deja de su lugar de trabajo libre.
- ⊙ *entra*: se le pasa por parámetro el neutral donde quiere entrar para llevar a cabo sus negocios, devolviendo cierto si consigue entrar y falso en cualquier otro caso. Las acciones que debe realizar para conseguir entrar dependerán del tipo de *Converso* que sea.
- ⊙ *sale*: se le pasa por parámetro el neutral de donde quiere salir, devolviendo la ubicación de donde estaba si consigue salir y `null` en cualquier otro caso. Para poder salir de un neutral se debe cumplir que primeramente estuviera dentro de ese edificio.
- ⊙ *comunicado*: esta acción solo la puede realizar un conserje. Se le pasa por parámetro el converso al que se le tiene que enviar el *guasap*.
- ⊙ *guasap*: se le pasa por parámetro el neutral al que entra para que lo actualice.
- ⊙ *estaDentro*: devuelve cierto si está dentro de algún neutral.
- ⊙ *winner*: se le pasa por parámetro la cantidad correspondiente al premio ganado, que se suma al dinero del converso.
- ⊙ *getTrabajo*: devuelve su lugar de trabajo.
- ⊙ *getOficina*: devuelve su oficina actual.

Un **Precursor** es un *Fundador*, por tanto tiene todas sus características, pero además tiene las variables de instancia:

- ⁕ *oficina*: edificio en el que puede realizar negocios (`Neutral`).
- ⁕ *trabajo*: edificio en el que trabaja de conserje (`Neutral`).

Además un *Precursor* es un *Converso* y tiene que implementar todos los métodos de esta interfaz. Por tanto los métodos a implementar son:

- ⊙ constructor: se le pasan los mismos parámetros y en el mismo orden que al constructor de *Fundador*, para inicializar su nombre, su fortaleza y sus propiedades, e inicialmente no se encuentra dentro de ningún neutral.
- ⊙ *contratado*: se le pasa por parámetro el neutral donde comienza a trabajar de conserje para que actualice su trabajo.
- ⊙ *despedido*: el precursor deja de su lugar de trabajo libre.

- ⊙ `entra`: se le pasa por parámetro el neutral en el que quiere entrar para poder hacer negocios con algún emprendedor. Para entrar realiza la búsqueda de una localidad en la parte izquierda del neutral, de manera que si encuentra una localidad disponible, le solicita al neutral la venta de una entrada. Si consigue la entrada, decrementa su dinero con el coste de dicha entrada y devuelve cierto. En cualquier otro caso devuelve falso.

- ⊙ `sale`: se le pasa por parámetro el neutral de donde quiere salir. Para poder salir de un neutral se debe cumplir que estuviera dentro y que éste sea el mismo que su oficina, de manera que al salir el valor de su oficina quede actualizado. El método devuelve la ubicación donde se encontraba dentro del edificio antes de salir.

- ⊙ `comunicado`: esta acción solo la puede realizar si es un conserje. Se le pasa por parámetro el converso al que se le tiene que enviar el `guasap`, al que le pasará como parámetro su lugar de trabajo.

- ⊙ `guasap`: se le pasa por parámetro una variable `Neutral` para que actualice su oficina.

- ⊙ `estaDentro`: devuelve cierto si está dentro de algún neutral, es decir, si tiene oficina.

- ⊙ `winner`: se le pasa por parámetro la cantidad correspondiente al premio que ha ganado y la suma a su dinero.

- ⊙ `getTrabajo`: devuelve su lugar de trabajo.

- ⊙ `calculaIntereses`: los intereses generados por los artículos que un precursor posee entre sus propiedades se obtiene calculando la desviación típica de los valores de sus artículos. Una vez calculado este interés, se suma a su dinero en concepto de beneficios. La desviación típica se obtiene aplicando la siguiente fórmula matemática a los valores de los artículos:

$$\sigma = \sqrt{(1/n \sum_{i=1}^{n} v_i^2) - \bar{V}^2}$$

  donde n es el número de artículos del precursor, v_i es el valor del artículo i y $\bar{V}$ es la media de los valores de sus artículos.

- ⊙ `negocia`: se le pasa por parámetro el emprendedor con el que va a intentar realizar negocios. Para ello ambos deben estar en el interior del mismo neutral. A continuación el precursor consulta las existencias de artículos del emprendedor, selecciona los que ocupan posiciones impares[5] y empieza adquirirlos: para cada artículo el precursor comprueba si tiene dinero para pagar su precio, que es el 50 % del valor del artículo. Si es así, el precursor debe:

  - añadir el artículo a sus propiedades en la primera posición;
  - pasar un mensaje de comprado al emprendedor con el artículo comprado y el pago estipulado para que actualice sus existencias;

  Al finalizar la transacción se devuelve el número de artículos que el precursor ha adquirido.

[5]La posición `0` se considera par.

- ⊙ *esAtacado*: se le pasa por parámetro el radical que lo está atacando, pero al haber desarrollado inmunidad a sus ataques su fortaleza no se ve afectada y devuelve el valor negativo correspondiente al 50 % del valor mínimo de entre todos sus productos como defensa residual.

- ⊙ *negocia*: recibe por parámetro el fundador con el que va a intentar hacer negocio con el intercambio de un artículo. Si el fundador pasado como parámetro es otro precursor, realizan el intercambio del último artículo que tienen en sus pertenencias sin ningún problema, pero si no lo es se aplica la restricción retrógrada de que tienen que ser del mismo sexo, por tanto intercambian como si ninguno fuera precursor. Si se realiza el intercambio se devuelve una cadena formada por el nombre del artículo que inicialmente era del fundador pasado por parámetro, un guión (−) y el nombre del artículo que inicialmente era del propio precursor. Si no se puede realizar el intercambio se devuelve la cadena `no hay intercambio`.

- ⊙ *getOficina*: devuelve el neutral en el que se encuentra.

Un **Emprendedor** es un *Radical*, por tanto tiene todas sus características, pero además tiene las variables de instancia:

- ⁕ *existencias*: lugar donde guarda los artículos que encuentra (`Articulo[]`);

- ⁕ *oficina*: edificio en el que puede realizar negocios (`Neutral`);

- ⁕ *trabajo*: edificio en el que trabaja de conserje (`Neutral`).

- ⁕ *dinero*: número real que representa la cantidad de efectivo de la que dispone (`double`).

Además un *Emprendedor* es un *Converso*, por tanto debe implementar todos los métodos de esta interfaz. Por tanto los métodos a implementar son:

- ⊙ constructor: se le pasan los mismos parámetros y en el mismo orden que al constructor de *Radical*, para inicializar su peso, su fuerza y su nombre, más un entero que indica el tamaño inicial de sus existencias, que debe ser mayor o igual a `0`[6]. Además, inicialmente no se encuentra dentro de ningún *Neutral* y su dinero inicial será de `125.5`.

- ⊙ *contratado*: se le pasa por parámetro el neutral donde comienza a trabajar de conserje para que actualice su trabajo.

- ⊙ *despedido*: el emprendedor deja de su lugar de trabajo libre.

- ⊙ *entra*: se le pasa por parámetro el neutral en el que quiere entrar para poder hacer negocios con algún precursor. Para entrar realiza la búsqueda de una localidad en la parte derecha del edificio, de manera que si encuentra una localidad disponible, le solicita al neutral la venta de una entrada. Si consigue la entrada, decrementa su dinero con el coste de dicha entrada y devuelve cierto. En cualquier otro caso devuelve falso.

- ⊙ *sale*: se le pasa por parámetro el neutral de donde quiere salir, devolviendo cierto si consigue salir y falso en cualquier otro caso. Para poder salir de un neutral se debe cumplir que estuviera dentro y que sea el mismo que su oficina, de manera que al salir el valor de su oficina quede actualizado. El método devuelve la ubicación donde se encontraba dentro del edificio antes de salir.

[6]El valor por defecto será `0`.

- ⊙ *comunicado*: esta acción solo la puede realizar si es un conserje. Se le pasa por parámetro el converso al que se le tiene que enviar el *guasap*, al que le pasará como parámetro su lugar de trabajo.
- ⊙ *guasap*: se le pasa por parámetro una variable *Neutral* para que actualice su oficina.
- ⊙ *estaDentro*: devuelve cierto si está dentro de algún neutral, es decir, si tiene oficina.
- ⊙ *winner*: se le pasa por parámetro la cantidad correspondiente al premio que ha ganado y la suma a su dinero.
- ⊙ *getTrabajo*: devuelve su lugar de trabajo.
- ⊙ *genera*: recibe por parámetro el nombre (`String`) del artículo que va a generar. Crea un artículo con ese nombre y busca la primera posición libre que tenga en sus existencias para guardarlo. Si no tiene ninguna posición libre, el emprendedor redimensiona sus existencias aumentando su tamaño en uno para dar cabida al nuevo artículo generado. A continuación actualiza el valor del artículo recién creado, utilizando como `String` de referencia el nombre del artículo que ocupe la posición anterior en las existencias o la cadena vacía (“”) si no hubiese artículo en esa posición.
- ⊙ *comprado*: se le pasa por parámetro el artículo vendido y la cantidad que ha cobrado al venderlo, de manera suma esta cantidad a su dinero y quita el artículo de sus existencias.
- ⊙ *ataca*: se le pasa por parámetro el fundador del que quiere su fortaleza. Debido a la evolución, el emprendedor no intenta extraer la fortaleza del fundador por la fuerza, sino que intenta realizar una especie de venta. Para ello, si dispone de artículos en sus existencias, selecciona el primer artículo de sus existencias y le solicita al fundador que cree un artículo con el nombre del artículo seleccionado. Si el fundador lo crea, el emprendedor quita de sus existencias el artículo seleccionado, lo devuelve e incrementa su fuerza con lo que le devuelve el fundador como resultado de la creación del artículo. Si el fundador no crea el nuevo artículo por algún motivo, el método devuelve `null` por defecto.
- ⊙ *invita*: Si recibe la invitación de un radical, se comportará exactamente igual que si fuera un radical y el hijo será un *Radical*. Si recibe la invitación de un emprendedor, el hijo será un *Emprendedor*, que tendrá el mismo peso, fuerza y nombre que si fuera un radical, y que tendrá como tamaño inicial de sus existencias el tamaño actual de las existencias de su padre, pero sin ningún artículo almacenado. Pero eso sí, solo acepta la invitación de otro radical/emprendedor una sola vez, nunca repite.
- ⊙ *getOficina*: devuelve el neutral en el que se encuentra.
- ⊙ *getExistencias*: devuelve un array dinámico con sus existencias.
- ⊙ *getDinero*: devuelve su dinero para que pueda ser consultado.

Restricciones en la implementación

- ⊛ Los métodos deben ser públicos y tener una *signatura* concreta:
  - En **Converso**
    - `public abstract void contratado(Neutral)`

- `public abstract void despedido()`
- `public abstract boolean entra(Neutral)`
- `public abstract Ubicacion sale(Neutral)`
- `public abstract void comunicado(Converso)`
- `public abstract boolean estaDentro()`
- `public abstract void guasap(Neutral)`
- `public abstract void winner(double)`
- `public abstract Neutral getTrabajo()`
- `public abstract Neutral getOficina()`

- En **Ubicacion**
  - `public Ubicacion(int,int)`
  - `public int getGrada()`
  - `public int getAsiento()`
- En **Neutral**
  - `public Neutral(int,int,double,String)`
  - `public boolean contrata(Converso)`
  - `public void despide()`
  - `public Ubicacion busquedaIzquierda()`
  - `public Ubicacion busquedaDerecha()`
  - `public double ventaDeEntrada(Converso,Ubicacion,double)`
  - `public Ubicacion sale(Converso)`
  - `public Ubicacion localizacion(Converso)`
  - `public int loteria()`
  - `public int getLibres()`
  - `public int getTotal()`
  - `public String getNombre()`
  - `public double getCaja()`
  - `public Converso getConserje()`
- En **Precursor**
  - `public Precursor(String,double,Articulo[])`
  - `public void contratado(Neutral)`
  - `public void despedido()`
  - `public boolean entra(Neutral)`
  - `public Ubicacion sale(Neutral)`
  - `public void comunicado(Converso)`
  - `public void guasap(Neutral)`
  - `public boolean estaDentro()`
  - `public void winner(double)`
  - `public Neutral getTrabajo()`
  - `public void calculaIntereses()`
  - `public int negocia(Emprendedor)`
  - `public double esAtacado(Radical)`

- `public String negocia(Fundador)`
- `public Neutral getOficina()`

- En **Emprendedor**
  - `public Emprendedor(double,double,String,int)`
  - `public void contratado(Neutral)`
  - `public void despedido()`
  - `public boolean entra(Neutral)`
  - `public Ubicacion sale(Neutral)`
  - `public void comunicado(Converso)`
  - `public void guasap(Neutral)`
  - `public boolean estaDentro()`
  - `public void winner(double)`
  - `public Neutral getTrabajo()`
  - `public void genera(String)`
  - `public void comprado(Articulo,double)`
  - `public Articulo ataca(Fundador)`
  - `public Radical invita(Radical,String)`
  - `public Neutral getOficina()`
  - `public ArrayList<Articulo>  getExistencias()`
  - `public double getDinero()`

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

⊛ Sólo se permite el uso del modificador `protected` en algunas variables de las clases de la práctica anterior (sólo aquellas de las que se hereda) y exclusivamente para aquellas variables a las que es necesario que las subclases accedan. Todas las demás variables deben ser privadas. En otro caso se restará un `0.5` de la nota total de la práctica o si se detecta un mal uso del modificador `protected` en la práctica.

Probar la práctica

- En el Campus Virtual se publicará un corrector de la práctica con una prueba (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 19 de abril**. 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 `correctorPublicarP2.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar:

  ```
  tar xfvz correctorPublicarP2.tgz
  ```

  De esta manera se creará una carpeta `publicar` que contendrá:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica2-pruebas`: dentro de este directorio están los ficheros con extensión `.java`, programas en Java con un método main que realiza 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 directorio `publicar` que se ha creado, 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.
  - Ficheros con extensión `.tmp.err`: estos ficheros debe estar vacíos por regla general. Sólo contendrán 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.
  - Ficheros con extensión `.tmp`: ficheros de texto con la salida generada por pantalla al ejecutar el fuente correspondiente, por ejemplo `p01.tmp` contendrá la salida generada al ejecutar el programa `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, 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 p01.txt p01.tmp`