Introducción a Java - Ejercicios

Índice

1. Proyecto básico en Eclipse

Crear un proyecto en Eclipse de nombre `ProyectoBasico`. Introducir los siguientes elementos en dicho proyecto:

- Establecer como directorio de fuentes un directorio `src`, y como directorio de destino un directorio `bin`.
- Crear un paquete `es.ua.jtech.ajdm.basico`.
- Introducir las clases Java que encontrarás en las plantillas de esta sesión en el proyecto, en la carpeta `basico`. Cada clase deberá copiarse al paquete que corresponda.
- Introducir en el proyecto la librería JAR que encontrarás en las plantillas de esta sesión. Hacer que esta librería forme parte del classpath del proyecto.
- Añadidle una nueva clase llamada `Prueba` en el paquete `es.ua.jtech.ajdm.basico`, que tenga el siguiente código:

```
package es.ua.jtech.ajdm.basico;

import es.ua.j2ee.animales.*;
import es.ua.j2ee.insectos.Mosca;
import es.ua.j2ee.plantas.*;

public class Prueba
{
        public static void main(String[] args)
        {
                // Datos del insecto
                Mosca m = new Mosca();
                m.nombre();
                int edad3 = m.edad() + 2;
                System.out.println ("Edad Insecto: " + edad3);

                // Datos del animal
                Elefante e = new Elefante();
                e.nombre();
                int edad2 = e.edad() + 5;
                System.out.println ("Edad Animal: " + edad2);

                // Datos de la planta
                Geranio g = new Geranio();
                g.nombre();
                int edad5 = g.edad();
                System.out.println ("Edad Planta: " + edad5);
        }
}
```

- Ejecutad la nueva clase Prueba.

2. Clases abstractas y herencia

Vamos a ver un ejemplo de clases abstractas y herencia. Se tiene una clase abstracta `Persona`, de la que hereda una clase `Hombre` para definir los métodos abstractos. Además, se tiene otra clase `Anciano`, que hereda de `Hombre` para modificar el valor de alguno de esos métodos. La clase principal del grupo es `EjHerencia`, que se encarga de ejecutarlo todo. Se pide:

- Probar a ejecutar la clase `EjHerencia` y comprobar el resultado producido.
- Convertid el ejemplo para utilizar interfaces en lugar de clases abstractas. Es decir, haced que `Persona` sea una interfaz (para ello no deberá implementar ningún método, sino dejarlos definidos).
- Indicad los cambios que sufren las clases `Hombre` y `Anciano` con eso. Probad el ejemplo con los cambios y comprobad que el resultado que devuelve es el mismo.

3. Aprovechando los elementos de Object

Echa un vistazo a la clase `EjObjetos` de la plantilla. Verás que hay una clase interna llamada `MiObject` que contiene un valor entero y una cadena. Tiene un constructor para poder asignar valor a cada campo. Después, desde la clase principal EjObjetos se crean varios objetos de este tipo, cada uno con un valor entero y una cadena. Después se utiliza el método equals para compararlos entre sí, e indicar si son iguales o no. Finalmente, se imprime por pantalla cada uno de los objetos creados.

Asumimos que dos objetos de tipo `MiObject` son iguales si sus campos enteros son iguales, y si sus cadenas también son iguales. Teniendo en cuenta todo esto:

- Ejecuta el programa. ¿Funcionan bien las comparaciones? ¿Qué resultados obtienes con ellas?
- ¿Qué texto aparece cuando imprimes los objetos `m1`, `m2` y `m3`? ¿Te resultan entendibles?

Probablemente habrás descubierto que las comparaciones no funcionan del todo bien: `m1` y `m2` son aparentemente iguales, y sin embargo dice que son diferentes. También habrás visto que a la hora de imprimir el objeto saca una ristra de caracteres que no se entienden. Vamos a corregir todo esto por separado.

- Para poder comparar bien los objetos `MiObject` entre sí necesitamos definir en esta clase su propio método equals, que redefina el que viene heredado de `Object`. Así que coloca un método equals en esta clase:

```
public boolean equals(Object o)
{
        ...
}
```

  y rellénalo de forma adecuada para que devuelva `true` si el objeto o es igual al actual (`this`), es decir, si tiene su campo entero y su campo cadena igual al actual.

Ayuda: observa que tendrás que convertir el objeto o a tipo `MiObject` para poder comparar. El método equals exige que el objeto que se le pasa como parámetro sea **siempre** de tipo `Object`, para poder hacer bien la herencia y sobreescritura del método equals original. Después, en el código, deberemos convertir este objeto al tipo concreto con que trabajemos:

```
MiObject mo = (MiObject) o;
```

- Para poder imprimir algo entendible al indicar que se imprima el objeto, debemos redefinir su método `toString`. Añade un método `toString` al código de `MiObject`:

```
public String toString()
{
        ...
}
```

  y haz que devuelva una cadena conteniendo el número entero, una coma, y la cadena del objeto `MiObject`.
- Ejecuta de nuevo el programa, y comprueba que funciona correctamente.

4. Recorrido de vectores y listas

La clase `EjColecciones` de la plantilla tiene un método `main`, donde hemos creado un objeto de tipo `Vector<String>`, y le hemos añadido 10 cadenas: `Hola0, Hola1, Hola2...Hola9`. Con este vector deberás hacer lo siguiente:

- Primero, recorrerlo mediante el objeto `Enumeration` que puedes obtener del propio vector. Si observas la API de `Vector`, verás que tiene un método:

```
Enumeration<?> elements();
```

  que devuelve un `Enumeration` para poder recorrer los elementos del vector. Se trata de que obtengas esa enumeración, y formes un bucle como se ha explicado en los apuntes, para recorrerla de principio a fin. Para cada elemento, saca su valor por pantalla (imprime la cadena).
- A continuación de lo anterior, haz otro recorrido del vector, pero esta vez utilizando su `Iterator`. Verás también en la API que el objeto `Vector` tiene un método:

```
Iterator<?> iterator();
```

  que devuelve un `Iterator` para poder recorrer los elementos del vector. Haz ahora otro bucle como el que se explica en los apuntes, para recorrer los elementos del

vector, esta vez con el `Iterator`. Para cada elemento, vuelve a imprimir su valor por pantalla.

- Finalmente, tras los dos bucles anteriores, añade un tercer bucle, donde a mano vayas recorriendo todo el vector, accediendo a sus elementos, y sacándolos por pantalla. En este caso, ya no podrás utilizar los métodos `nextElement`, `hasNext`, ni similares que has utilizado en los bucles anteriores. Deberás ir posición por posición, accediendo al valor de esa posición del vector (puedes utilizar el método que prefieras: `get` o `elementAt`), y sacando el valor obtenido por pantalla.

Una vez tengas los tres bucles hechos, ejecuta el programa, y observa lo que saca cada uno de los bucles por pantalla. ¿Encuentras alguna diferencia en el comportamiento de cada uno? ¿Qué forma de recorrer el vector te resulta más cómoda de programar y por qué?

Nota:
algunas de las técnicas que has utilizado para recorrer el vector se pueden utilizar de la misma forma para recorrer otros tipos de listas. Por ejemplo, puedes obtener el `Iterator` de un `ArrayList` y recorrerlo, o ir elemento por elemento.

5. Captura de excepciones

En el proyecto `Excepciones` de las plantillas de la sesión tenemos una aplicación `Ej1.java` que toma un número como parámetro, y como salida muestra el logaritmo de dicho número. Sin embargo, en ningún momento comprueba si se ha proporcionado algún parámetro, ni si ese parámetro es un número. Se pide:

a) Compilar el programa y ejecutadlo de tres formas distintas:

- Sin parámetros

```
java Ej1
```

- Poniendo un parámetro no numérico

```
java Ej1 pepe
```

- Poniendo un parámetro numérico

```
java Ej1 30
```

Anotad las excepciones que se lanzan en cada caso (si se lanzan)

b) Modificar el código de `main` para que capture las excepciones producidas y muestre los errores correspondientes en cada caso:

- Para comprobar si no hay parámetros se capturará una excepción de tipo `ArrayIndexOutOfBoundsException` (para ver si el *array* de `String` que se pasa en el

`main` tiene algún elemento).
- Para comprobar si el parámetro es numérico, se capturará una excepción de tipo `NumberFormatException`.

Así, tendremos en el `main` algo como:

```
try
{
        // Tomar parámetro y asignarlo a un double
} catch (ArrayIndexOutOfBoundsException e1) {
        // Codigo a realizar si no hay parametros
} catch (NumberFormatException e2) {
        // Codigo a realizar con parametro no numerico
}
```

Probad de nuevo el programa igual que en el caso anterior comprobando que las excepciones son capturadas y tratadas.

6. (*) Lanzamiento de excepciones

El fichero `Ej2.java` es similar al anterior, aunque ahora no vamos a tratar las excepciones del `main`, sino las del método `logaritmo`: en la función que calcula el logaritmo se comprueba si el valor introducido es menor o igual que 0, ya que para estos valores la función logaritmo no está definida. Se pide:

a) Buscar entre las excepciones de Java la más adecuada para lanzar en este caso, que indique que a un método se le ha pasado un argumento ilegal. (Pista: Buscar entre las clases derivadas de `Exception`. En este caso la más adecuada se encuentra entre las derivadas de `RuntimeException`).

b) Una vez elegida la excepción adecuada, añadir código (en el método `logaritmo`) para que en el caso de haber introducido un parámetro incorrecto se lance dicha excepción.

```
throw new ... // excepcion elegida
```

Probar el programa para comprobar el efecto que tiene el lanzamiento de la excepción.

c) Al no ser una excepción del tipo *checked* no hará falta que la capturemos ni que declaremos que puede ser lanzada. Vamos a crear nuestro propio tipo de excepción derivada de `Exception` (de tipo *checked*) para ser lanzada en caso de introducir un valor no válido como parámetro. La excepción se llamará `WrongParameterException` y tendrá la siguiente forma:

```
public class WrongParameterException extends Exception
{
        public WrongParameterException(String msg) {
                super(msg);
        }
}
```

Deberemos lanzarla en lugar de la escogida en el punto anterior.

```
throw new WrongParameterException(...);
```

Intentar compilar el programa y observar los errores que aparecen. ¿Por qué ocurre esto? Añadir los elementos necesarios al código para que compile y probarlo.

d) Por el momento controlamos que no se pase un número negativo como entrada. ¿Pero qué ocurre si la entrada no es un número válido? En ese caso se producirá una excepción al convertir el valor de entrada y esa excepción se propagará automáticamente al nivel superior. Ya que tenemos una excepción que indica cuando el parámetro de entrada de nuestra función es incorrecto, sería conveniente que siempre que esto ocurra se lance dicha excepción, independientemente de si ha sido causada por un número negativo o por algo que no es un número, pero siempre conservando la información sobre la causa que produjo el error. Utilizar *nested exceptions* para realizar esto.

Ayuda

Deberemos añadir un nuevo constructor a `WrongParameterException` en el que se proporcione la excepción que causó el error. En la función `logaritmo` capturaremos cualquier excepción que se produzca al convertir la cadena a número, y lanzaremos una excepción `WrongParameterException` que incluya la excepción causante.

7. Leer un fichero de texto

Vamos a realizar un programa que lea un fichero de texto ASCII, y lo vaya mostrando por pantalla. El esqueleto del programa se encuentra en el fichero `Ej1.java` dentro del proyecto `Serializacion` de las plantillas de la sesión. Se pide:

a) ¿Qué tipo de flujo de datos utilizaremos para leer el fichero? Añadir al programa la creación del flujo de datos adecuado (variable `in`), compilar y comprobar su correcto funcionamiento.

b) Podemos utilizar un flujo de procesamiento llamado `BufferedReader` que mantendrá un *buffer* de los caracteres leídos y nos permitirá leer el fichero línea a línea además de utilizar los métodos de lectura a más bajo nivel que estamos usando en el ejemplo. Consultar la documentación de la clase `BufferedReader` y aplicar la transformación sobre el flujo de entrada (ahora `in` deberá ser un objeto `BufferedReader`). Compilar y comprobar que sigue funcionando correctamente el método de lectura implementado.

c) Ahora vamos a cambiar la forma de leer el fichero y lo vamos a hacer línea a línea aprovechando el `BufferedReader`. ¿Qué método de este objeto nos permite leer líneas de la entrada? ¿Qué nos devolverá este método cuando se haya llegado al final el fichero? Implementar un bucle que vaya leyendo estas líneas y las vaya imprimiendo, hasta llegar al final del fichero. Compilar y comprobar que sigue funcionando de la misma forma.

8. (*) Gestión de productos

Vamos a hacer una aplicación para gestionar una lista de productos que vende nuestra empresa. Escribiremos la información de estos productos en un fichero, para almacenarlos de forma persistente. Se pide:

a) Introducir el código necesario en el método `almacenar` de la clase `GestorProductos` para guardar la información de los productos en el fichero definido en la constante `FICHERO_DATOS`. Guardaremos esta información codificada en un fichero binario. Deberemos codificar los datos de cada producto (titulo, autor, precio y disponibilidad) utilizando un objeto `DataOutputStream`.

b) Introducir en el método `recuperar` el código para cargar la información de este fichero. Para hacer esto deberemos realizar el procedimiento inverso, utilizando un objeto `DataInputStream` para leer los datos de los productos almacenados. Leeremos productos hasta llegar al final del fichero, cuando esto ocurra se producirá una excepción del tipo `EOFException` que podremos utilizar como criterio de parada.

c) Modificar el código anterior para, en lugar de codificar manualmente los datos en el fichero, utilizar la serialización de objetos para almacenar y recuperar objetos `ProductoTO` del fichero.