Práctica 1

Objetivo

Hacer un repaso de los elementos del lenguaje de programación Java y practicar el desarrollo de métodos recursivos.

Ejecución de programas Java

Un programa en Java es un conjunto de clases diseñadas para colaborar en una tarea, con una clase pública distinguida que contiene un método de clase `public static void main(String[] args)` que desencadena la ejecución del programa. Este programa puede tener argumentos de entrada que se recogen en el array de tipo `String args` y que son introducidos cuando se hace la llamada al programa. Las demás clases pueden estar definidas ad hoc o pertenecer a una biblioteca de clases. Los entornos de programación o IDE ofrecen todo lo necesario para compilar y ejecutar programas, pero una vez que Java está en el `path` del sistema la compilación puede realizarse desde la propia línea de comandos. Para ello hay que abrir la consola en el directorio donde se encuentran los ficheros `.java` y ejecutar la instrucción `javac MiClase.java` y para su ejecución se ejecutaría la instrucción `java MiClase.class`.

Ejercicio 1. Desarrolla una clase ejecutable en Java llamada **SumaAleatoria** que sume una serie de números aleatorios de tipo **double** e imprima el resultado por pantalla (`System.out.println`). Utiliza la clase **Random** y el método **nextDouble()** para generar los números a sumar. En caso de que el array de entrada `args` tenga argumentos se utilizarán tantos sumandos como valor indicado como argumento de entrada. Si no tiene argumentos, se sumarán los números indicados por la *constante de clase* `SUMANDOS`, que estará inicializada a `20`. Compila y ejecuta utilizando la línea de comandos.

Subclases

En Java se pueden definir subclases o clases que heredan estado y comportamiento de otra clase (la superclase) a la que amplían de la forma:

```
class miClase extends superclase {
…
}
```

En Java solo se permite herencia simple, por lo que no puede establecerse un árbol de clases. Todas las clases confluyen en la clase `Object` de `java.lang` que recoge los comportamientos básicos de cualquier clase. Cuando una clase hereda de otra se heredan sus variables y métodos que son públicos o protegidos. Sin embargo, los constructores no se heredan, pero pueden hacerse llamadas a `super(…)` para ejecutar el código contenido en un constructor de la superclase.

Ejercicio 2. Desarrolla una clase `Bicicleta` que tenga dos variables de tipo entero llamadas `frenos` y `velocidad`. La primera indica la capacidad de frenada de la bicicleta y se pasa al constructor de la clase. La última es la velocidad actual y se inicializa a `0`. La clase tiene 3 métodos, `frenar()`, que decrementa el valor de velocidad en la cantidad indicada por `frenos`, `acelerar(int v)`, que incrementa el valor de velocidad en `v` unidades y `toString` que devuelve un `String` en el que consta la capacidad de frenada de la bicicleta y su velocidad actual.

Ejercicio 3. Desarrolla una clase `BicicletaDeMontana` que herede de `Bicicleta`. Esta clase tiene una variable entera llamada `tam` que se le pasará en el constructor. Además, definirá el método adicional `establecerTamano(int t)`, que sustituirá el valor de `tam` por `t`. Se sobrescribirá el método `toString` para concatenar el `String` generado por el método `toString` de `Bicicleta` con el valor de la variable `tam`.

Ejercicio 4. Desarrolla una clase ejecutable `TestBicicleta` que imprima por pantalla el valor retornado por `toString` de diferentes objetos de las clases `Bicicleta` y `BicicletaDeMontana`.

Estructuras de control

En Java disponemos de 4 tipos de sentencias de control: sentencias de repetición, sentencias de selección, sentencias de control de excepciones y sentencias de salto/ramificación. Durante este curso haremos uso de las 3 primeras. Las sentencias de repetición más utilizadas son:

```
while(<exp.booleana>){
      <sentencias>
}
```

```
do <sentencia>
while (<exp. booleana>)
```

```
for(<exp1>; <exp. bool>;<exp2>){
      <sentencias>
}
```

Las dos sentencias de selección más utilizadas son el if y el case. En las siguientes secciones de código podemos ver su sintaxis.

```
if (<exp. bool1>) <sentencia1>
else if (<exp. bool2>) <sentencia2>
...
else <sentenciaN>
```

```
<exp bool> ? <exp1> : <exp2>
```

```
switch (<exp ent>) {
case <exp ent1>: <sentencia1> break;
...
default: <sentenciaN> break;
}
```

Java dispone de un mecanismo de ayuda para la comunicación y el manejo de errores conocido como control de excepciones. Cuando se produce un error en un método se genera un objeto de la clase `Exception`, o de alguna heredera, con información sobre el error, se interrumpe el flujo normal de ejecución, y el entorno de ejecución trata de encontrar un tratamiento para dicho objeto dentro del propio método o en uno anterior en la pila de activaciones. Existen tres sentencias relacionadas con el control de excepciones: `try` que delimita un bloque de instrucciones donde se puede producir una excepción, `catch` que identifica un bloque de código asociado a un bloque try donde se trata un tipo particular de excepción, `finally` (cuyo uso es opcional) que identifica un bloque de código que se ejecutará después de un bloque try con independencia de que se produzcan o no excepciones. El código de control de excepciones presenta esta estructura:

```
try {
      <sentencia/s>
} catch(<tipoexcepción> <identif>) {
      <sentencia/s>
} finally {
<sentencia/s>
}
```

Las estructuras de control tienen una gran importancia en el estudio de la complejidad de los programas. En muchos casos determinan cuál es el mejor, peor y caso medio de nuestro programa. En muchos otros determinan directamente el orden de complejidad.

Ejercicio 5. Desarrolla una clase ejecutable llamada `Factorial` que defina un método de clase para calcular de forma iterativa el factorial de un número `n` que se le pasa como argumento.

Ejercicio 6. Desarrolla una clase `MayorDe3` con un método de clase que devuelva el mayor de 3 números que se pasan como argumento usando sentencias `if-then-else` y usando el operador ternario (`<exp bool> ? <exp1> : <exp2>`).

Ejercicio 7. Desarrolla una clase ejecutable `DiaDeLaSemana` que dado un número entre 1 y 7 que se pasa como argumento imprima por pantalla el día de la semana que corresponde. Controla el caso en el que el número que se pasa como argumento no está entre 1 y 7 informando del error. Utiliza la `switch-case` para implementar la selección del día.

Ejercicio 8. Desarrolla una clase ejecutable `Division` que calcule la división de dos argumentos dados. En caso de que el divisor sea *0* se debe capturar la excepción de tipo `ArithmeticException` y mostrar por pantalla la traza del error usando `e.printStackTrace()`. También capturarse la excepción de clase `NumberFormatException` que se lanza por el método `Integer.parseInt` en caso de que los parámetros proporcionados al programa no tengan formato numérico correcto. En esta ocasión se debe escribir un mensaje por pantalla informando de lo ocurrido.

Recordemos que otras excepciones frecuentes son `NullPointerException` y `ArrayIndexOutOfBoundsException`.

Vectores

Durante la asignatura haremos uso de estructuras vectoriales y matriciales. Cuando queramos implementar estas estructuras haremos uso de arrays o listas. La principal diferencia entre ellos es que los arrays se crean con una longitud determinada, mientras que las listas se crean vacías y pueden crecer y decrecer durante la ejecución del programa que las use. Algunas instrucciones útiles para trabajar con **arrays** son las siguientes:

Declaración:

```
int [] array_enteros;
int[][] matriz_enteros;
Punto [] array_puntos;
```

Inicialización:

```
array_enteros = new int[10];
array_puntos = new Punto[23];
matriz_enteros = {{10, 20}, {30, 40}, {50, 60}, {70, 80}};
char[] array_vocales = {'a', 'e', 'i', 'o', 'u'};
```

Copia de arrays:

```
int[] array2 = Arrays.copyOf(array1, array1.length);
System.arraycopy(array1, 0, array2, 0, array1.length);
```

Consulta:

```
vocales[0] -> a
matriz_enteros[0][0] -> 10
```

Otras instrucciones interesantes nos permiten concatenar arrays a través del método `ArrayUtils.addAll(array1,array2)`. Dos arrays pueden compararse a través del método `Arrays.equals(array1,array2)`.

Las lista suelen implementarse utilizando la interfaz `Java.util.List<T>`, que es una colección ordenada de objetos en la que los objetos pueden duplicarse. Los objetos `List` preservan el orden de inserción por lo que podemos acceder a los objetos utilizando su posición en la lista. La interfaz lista es implementada por las

clases `ArrayList`, `LinkedList`, `Vector` y `Stack`. Al ser `List` un interfaz las instancias se pueden crear de una de las siguientes maneras:

```
List<T> a = new ArrayList<T>();
List<T> b = new LinkedList<T>();
List<T> c = new Vector<T>();
List<T> d = new Stack<T>();
```

Los métodos más importantes para trabajar con listas son `add(obj)`, `add(i,obj)`, `set(i,obj)` y `get(i)` que nos permiten añadir un elemento a la lista, modificar el elemento en la posición `i` y obtener una referencia al objeto en la posición `i`. Otros métodos interesantes son `Arrays.asList()`, que nos permite crear una lista a partir de un array, `addAll()` que añade todos los elementos de una lista en otra o `remove(i)` que permite borrar un elemento que está en la posición `i` de la lista. Existen muchos métodos para trabajar con listas, por lo que se recomienda consultar la documentación oficial de Java de la interfaz `Collection<T>` y de la interfaz `List<T>`.

Ejercicio 9. Desarrolla una clase ejecutable llamada `EncuentraComun` con un método de clase llamado `comunes` que reciba dos arrays de `String` y devuelva una lista con los elementos comunes en ambos.

Ejercicio 10. Desarrolla una clase ejecutable `PartirLista`, que cree un objeto `ArrayList<Integer>` lo rellene y lo divida en 2 listas, que contengan, respectivamente, los elementos de la mitad izquierda y de la mitad derecha de la lista original.

Ejercicio 11. Desarrolla una clase `OperacionesListasEnteros` que tenga un método de clase `rotarLista` que dada una lista de enteros (`List<Integer>`) y un entero `e` devuelva la misma lista con los elementos rotados a la izquierda `e` posiciones. Por ejemplo si se pasa como entrada la lista `l={10,20,30,40,50,60,70}` y se rota 3 posiciones, se obtiene la lista `l={40,50,60,70,10,20,30}`.

Recursividad

La recursividad está estrechamente ligada a una de las técnicas más importantes para estructurar programas, el procedimiento. La llamada a un procedimiento altera el flujo de control de un programa, de manera que cuando se llama al procedimiento el programa que lo invoca interrumpe su ejecución y no lo recupera hasta que el procedimiento termina de ejecutarse. Para entender un fragmento de código que contiene una llamada a un procedimiento basta con saber qué hace, no cómo lo hace. Un **procedimiento recursivo** es un procedimiento que se llama a sí mismo, sea directamente o por medio de una cadena de procedimientos distintos. Cada llamada ejecutará el mismo procedimiento, pero para un tamaño de entrada menor. Para el correcto funcionamiento de un algoritmo recursivo es fundamental definir al menos un **caso base**, aquel que puede resolverse sin recursión. Con cada llamada nos debemos acerca a uno de los casos bases de nuestro algoritmo recursivo. Existen distintos tipos de recursión:

- **Recursión lineal:** En este tipo de recursión, cada llamada a la función recursiva sólo hace una nueva llamada recursiva.
- **Recursión en cascada:** Hablamos de recursión en cascada cuando en tiempo de ejecución se realiza más de una llamada recursiva.
- **Recursión anidada:** Hablamos de recursión anidada cuando una o alguna de las llamadas recursivas tiene como parámetro otra llamada recursiva.
- **Recursión de cola:** Un algoritmo recursivo presenta recursión de cola o recursión de extremo final, cuando la última acción efectiva de ese algoritmo es esa llamada recursiva que una vez resuelta hace que termine el algoritmo. Para este tipo de algoritmos recursivos es sencillo encontrar una versión iterativa equivalente.

Ejercicio 12. Añade un nuevo método a la clase `Factorial` desarrollada en el ejercicio 5 que calcule el factorial de un número `n` utilizando recursión lineal.

Ejercicio 13. Desarrolla una clase `Matematicas` con una función de clase `fibonacci` que calcule el número de Fibonacci para una entrada `n` utilizando recursión en cascada.

Ejercicio 14. Añade a la clase `Matematicas` del ejercicio 13 un método de clase llamado `ackermann` que calcule el número de Ackermann para unas entradas `m` y `n` utilizando recursión anidada. El número de Ackerman se calcula utilizando las siguientes expresiones:

$$Ack(0,n) = n + 1$$

$$Ack(m,0) = Ack(m-1,1)$$

$$Ack(m,n) = Ack\big(m-1, Ack(m,n-1)\big) \quad Si\ m,n > 0$$

Ejercicio 15. Añade a la clase `Matematicas` del ejercicio 13 un método `mcd` que calcule el máximo común divisor de los números `m` y `n` (`m` siempre será el mayor de los 2 números) utilizando recursión de cola. Para ello hacer uso de las siguientes expresiones:

$$mcd(m,0) = m$$

$$mcd(m,n) = mcd(n, m\%n)$$

Ejercicio 16. Añadir a la clase `Matematicas` un método **recursivo** `mostrarLista(List<Integer> lista, int i)` que muestre por pantalla, en orden inverso, todos los números de la lista desde la posición `i` hasta el final. **No se debe modificar la lista ni utilizar ninguna estructura de datos auxiliar**.