Grado en Ingeniería Informática y Doble Grado en Ingeniería Informática y Administración de Empresas

Programación

2020 - 2021

Ejercicio 1. Para organizar mi biblioteca de libros, quiero hacer un programa que me permita tenerlos colocados y poder localizarlos rápidamente. Para ello, necesito que realices las siguientes tareas:

1) Crear una clase, denominada Libro, que deberá tener tres atributos:
   - titulo: representa el título del libro
   - numero_de_paginas: representa el número de páginas total
   - leido: indica si ya se ha leído el libro o no (por defecto un libro no estará leído)

   Crear un método init que reciba valores para sus tres atributos.
   Crear las propiedades necesarias para los atributos anteriores (se debe comprobar que el número de páginas sea mayor que cero).

2) Crear una clase denominada Biblioteca que nos permitirá tener organizados los libros. La clase deberá tener un atributo denominado los_libros que almacenará una lista con los libros de la biblioteca

   Crear un método init que reciba como parámetro la lista de los libros de la Biblioteca
   Crear un método set_libro que recibe un Libro y una posición y coloca ese libro en la posición indicada en la lista de libros.
   Crear un método dame_titulo que recibe una posición, y devuelve el título del Libro situado en esa posición de la Biblioteca.
   Crear un método denominado dame_libros_ordenados que devuelve una copia de la lista de libros de la biblioteca, ordenados ascendentemente según su número de páginas (se puede utilizar cualquier algoritmo de ordenación de los vistos en clase, no usar el método sort de listas).

3) Crear tres libros que todavía no están leídos:
   "El Quijote", 800 páginas
   "El Lazarillo de Tormes" que tiene 125 páginas
   "Don Juan Tenorio" que tiene 100 páginas
   Crear una biblioteca, que contiene los tres libros anteriores
   Imprimir por pantalla los números de páginas de los libros de la biblioteca (separados por coma), ordenados ascendentemente.

Ejercicio 2. Crear las siguientes clases:

1. Clase `Persona`, con las siguientes características:
   - Atributos: `nombre` de tipo `String`, `dinero` de tipo `float`, `edad` de tipo `int`, y `carnet` de tipo `boolean`.
   - Crear propiedades para `edad`. Se debe comprobar que la `edad` está entre 0 y 150 años.
   - Crear propiedades para `carnet`. Si se va a poner `carnet` a `true` se debe comprobar que la `Persona` sea mayor de edad. En caso contrario no se cambiará el valor de `carnet` y se imprimirá por pantalla: "`no puede tener carnet porque tiene X años`".

Donde `X` debe ser la edad.

- Crear un método init que reciba valores para todos los parámetros. Deberá comprobar que los valores pasados para `edad` y `carnet` tienen sentido.
- Crear un método `str` que devuelva "`A tiene B años, C euros y tiene/no tiene carnet de conducir`" donde `A`, `B` y `C` deben cambiarse por los valores adecuados para cada objeto `Persona`.
- Crear un método `__eq__` que compare dos `Personas`. Dos personas son iguales si su `nombre`, `edad`, `dinero` y `carnet` son iguales.

2. Clase `Vehiculo`, con las siguientes características:
   - Atributos `matricula` de tipo `String`, `seguro` que solo podrá ser `{NINGUNO, TERCEROS, LUNAS, TOTAL}`, `dueño` de tipo `Persona`, `precio` de tipo `float` y `lunasOK` de tipo `boolean`.
   - Crear una propiedad para `matricula`. Debe comprobar que la cadena que se recibe tiene exactamente 7 caracteres.
   - Crear una propiedad para `seguro` que compruebe que reciba el valor correcto.
   - Crear una propiedad para `dueño`. Deberá comprobar que la `Persona` es mayor de `edad` y tiene `dinero` suficiente para comprar el coche. En caso contrario no hará nada. Se deberá restar el `precio` del `Vehiculo` del `dinero` del `dueño`.
   - Crear un método init que reciba valores para todos los atributos. Debe comprobar que el `dueño` es mayor de edad y tiene `dinero` y que la `matricula` es correcta. En caso contrario dejará estos atributos en blanco.
   - Crear un método `asegurar` que recibe como parámetro el tipo de `Seguro`. Se deberá restar el precio del `seguro` del `dinero` del `dueño`. El precio del `seguro` es un 1% del precio del `Vehiculo` si es a `TERCEROS`, el 1,2% si es con `LUNAS` y el 3% si es `TOTAL`. Se debe comprobar que el dueño tiene dinero para asegurar el vehículo. Si tiene dinero se cambiará el tipo de seguro del vehículo y se decrementará el dinero del dueño, si no, no hará nada.
   - Crear un método `__eq__` para comprobar si dos `Vehiculos` son iguales. Dos `Vehiculos` son iguales si tienen la misma `matricula`.
3. Clase `Taller`, con las siguientes características:
   - Atributos: `vehiculosPendientes` y `vehiculosOK` que será una lista de la clase `Vehiculo`.
   - Un método init que recibe una lista de `Vehiculo` para el primer atributo y otra para el segundo.
   - Un método `llevarAlTaller` que reciba un `Vehiculo` y lo coloque en `vehiculosPendientes`.
   - Un método `repararLunas` que saca el primer elemento de `vehiculosPendientes` y pone las `lunasOK` a `true`. Si el tipo de seguro del `Vehiculo` no es `TOTAL` o `LUNAS`, cobra 150 euros al `dueño` (comprobar que el dueño tiene dinero). A continuación coge ese `Vehiculo` y lo añade al final de la lista `vehiculosOK`.
4. Hacer un programa principal en el que:
   - Se cree una `Persona`

- Se cree un `Vehiculo` que tenga esa `Persona` como `dueño`.
- Se asegure el `Vehiculo`
- Se cree un `Taller`, se lleve ese `Vehiculo` para reparar y se repare
- Se imprima el dueño del primer `Vehiculo` en `vehiculosOK`.

Ejercicio 3. Se quiere crear un programa que ayude al usuario a invertir en Bolsa, para ello se crearán una serie de clases.

Crear la clase `Acción` con las siguientes características:

a) Atributos: `nombre`, `valor`, `valorAnterior`, que deben guardar respectivamente el nombre de la acción, su valor actual y el valor en la sesión anterior. Elegir el tipo más adecuado para estos atributos.

b) Propiedades para los atributos que lo necesiten. Tanto `valor` como `valorAnterior` deben ser siempre mayores que cero.

c) Un método init completo que recibe valores para los tres atributos.

d) Un método `actualizar` que reciba el nuevo `valor` de la `Acción`, lo coloque en el atributo correspondiente y ponga el `valor` antiguo en `valorAnterior`. Si el valor recibido no es mayor o igual que cero, no hará nada.

e) Un método `truncar` que recibe un `float` y devuelve un `String` resultado de truncar el número recibido a 2 decimales. Asumir que el número tiene siempre más de 2 decimales. Ejemplo, recibe `2.14932423` y devuelve "`2.14`". Debe funcionar para cualquier número.

f) Un método `variación` que devuelva como `String` con dos decimales el porcentaje de variación de una acción de un día respecto al anterior.

g) Un método `str` que devuelva: "`<nombre de acción>;<valor>;<variación respecto al día anterior>%`" Por ejemplo, para la `Acción` creada por defecto devolvería: "`Inditex;22.72;-1.00%`". Tanto `valor` como variación deben tener 2 decimales.

h) Un método `__eq__` que recibe por parámetro otro objeto `Acción` y devuelve `true` si ambas son iguales y `false` en caso contrario. Dos acciones son iguales si tienen el mismo `nombre`.

Crear la clase `Bolsa` con las siguientes características:

a) Atributo: `cotizaciones`, de tipo lista de `Acción`.

b) Un método `partirString` que recibe un `String` como parámetro. El parámetro tendrá siempre un formato como "`Inditex;22.72;22.95 Telecinco;10.02;11.02`" (es decir, los `nombres` de cada `Acción`, su `valor` actual y su valor anterior, separando cada `Acción` por un espacio). Devuelve una matriz de `String` en el que hay tantas filas como acciones y cada fila tiene 3 columnas, la primera es el `nombre`, la segunda el `valor` y la tercera el valor anterior. El método debe funcionar para cualquier número de acciones dentro del `String`.

c) Un método `creaAcciones` que recibe un `String` como el del método anterior y devuelve una

lista con los objetos `Acción` correspondientes

d) Un método init que recibe un `String` como el de los dos métodos anteriores y crea la lista `cotizaciones` con el contenido de ese `String`.

e) Un método `buscar` que reciba un `String` con el `nombre` de una `Acción` y si la `Acción` está en `cotizaciones` imprima el `nombre`, `valor` y variación respecto al día anterior de esa `Acción`. Debe devolver también la posición en la que se encuentra esa `Acción` en la lista o -1 si no está.

f) Un método `str` que devuelva el `nombre`, `valor` y variación con el día anterior de cada elemento de `cotizaciones`, cada uno en una línea.

Crear un programa `Prueba` con las siguientes características:

a) Una función `ordenar`, que reciba una lista de `Acción` y utilizando el algoritmo de la **burbuja** lo ordene de forma que las acciones de menor `valor` sean las primeras.

b) Hacer lo siguiente:

   a) Pedir al usuario que por teclado introduzca los datos de una lista de acciones en el formato: `nombre;valor;valor anterior`. Para terminar debe teclear la palabra “fin” (será válida cualquier combinación de mayúsculas o minúsculas). Guardar las acciones en un `String` separándolas por un espacio.

   b) Crear un objeto `Bolsa` usando el `String` anterior.

   c) Usar la función `ordenar` para ordenar las acciones del atributo `cotizaciones` del objeto `Bolsa` creado.

   d) Imprimir el objeto `Bolsa`.

Ejemplo de ejecución:

```
Introduce las acciones. Introduce "fin" para terminar
Inditex;22.72;22.95
Telecinco;10.02;11.02
Antena3;12.12;12.04
FiN
Telecinco;10.02;-9.07%
Antena3;12.12;0.66%
Inditex;22.72;-1.00%
```

Ejercicio 4. El objetivo es crear un programa para jugar a un juego con cierto parecido al dominó. Tenemos una serie de fichas rectangulares que tienen un color en cada extremo, de entre cinco posibles (**R**ojo, **V**erde, **A**zul, **M**arrón o **B**lanco). El número de fichas se le pedirá al usuario y los colores de cada uno de los dos lados se generarán pseudo-aleatoriamente de forma que el problema tenga solución (en el enunciado se darán los detalles). Al iniciar la partida las fichas se colocan en línea aleatoriamente. El usuario debe colocar las fichas de forma que el color de la parte derecha de una ficha sea el mismo que el de la parte izquierda de la siguiente ficha y que la línea de fichas comience y termine por Blanco, es decir, la ficha al inicio de la línea tendrá la parte izquierda blanca, mientras la del final de la línea tendrá blanco en la parte derecha. De esta forma todas

las fichas estarán conectadas por su color. A continuación se puede ver un ejemplo de partida con 5 fichas. Se muestra la inicial del color de cada ficha, si dos fichas están conectadas (coincide el color de la derecha de una ficha con el de la izquierda de la siguiente), la inicial se muestra en mayúsculas, si no están conectadas, la inicial se muestra en minúsculas:

```
¿Número de fichas?
5
Movimientos: 0
[B m][v b][m B][B v][m m]
¿Origen?
3
¿Destino?
0
Movimientos: 1
[B v][b m][v b][m b][m m]
¿Origen?
2
¿Destino?
1
Movimientos: 2
[B V][V B][B M][M b][m m]
¿Origen?
4
¿Destino?
5
¡Posición incorrecta!
Movimientos: 2
[B V][V B][B M][M b][m m]
¿Origen?
3
¿Destino?
4
Movimientos: 3
[B V][V B][B M][M M][M B]
¡Has ganado!
```

Crear la clase `Ficha` con las siguientes características:

a) **(0,2 puntos)** Atributos: dos listas de dos posiciones, una denominada `colores` para guardar los colores de cada parte de la `Ficha` y otraa denominado `conectada`, para guardar si la `Ficha` está conectada con la de la derecha o con la de la izquierda. Elegir el tipo más adecuado. Asumir que la posición 0 de cada lista representa la izquierda de la `Ficha` y la 1 la derecha.

b) Un método `ponerBlanco` que no recibe parámetros y que pone ambos lados de la `Ficha` en blanco (el color será `BLANCO` para ambos)

c) Un método `setColor` que recibe una lista de `Color` y pone la `Ficha` con esos colores. Si no tiene el tamaño adecuado se deberá imprimir un mensaje de error por pantalla y poner la `Ficha` en blanco.

d) Un método init que recibe un array de `Color` y crea la `Ficha` con esos colores. Si el array no tiene el tamaño adecuado se deberá imprimir un mensaje de error por pantalla y poner la `Ficha` en blanco.

e) Un método `getColor` que recibe un entero y devuelve el `Color` de la parte izquierda o derecha de la `Ficha`, según el número recibido.

f) Un método `getInicialColor`, que reciba un `Color` y devuelva su inicial.

g) Un método `str` que devuelva una cadena que representa a la `Ficha` tal y como aparece en el ejemplo de la primera página. Por ejemplo para una `Ficha` Roja-Verde que está conectada con la anterior pero no con la posterior, devolverá `[R v]`

Crear la clase `Tablero` con las siguientes características:

a) Atributos: `fichas`, de tipo lista de `Ficha`, y `movimientos`, cuyo tipo debe ser elegido por el alumno.

b) Un método init que reciba el número de `fichas` del `Tablero` y cree la lista (pero no las fichas). Si el número es menor de 2, creará un `Tablero` de 10 fichas.

c) Un método `obtenerColorAleatorio`, que devuelva un `Color` de entre los posibles, elegido aleatoriamente.

d) Un método `obtenerColorFichaAnterior` que recibe la posición de una `Ficha` y devuelve el `Color` de la derecha de la ficha anterior. Si la posición de la `Ficha` es la primera (y por lo tanto no tiene `Ficha` anterior) devolverá `BLANCO`. Hacer también un método `obtenerColorFichaSiguiente` que devuelva el color de la izquierda de la `Ficha` posterior. Si es la última, devolverá `BLANCO`

e) Un método `rellenarTablero`, que crea las `fichas` de forma que la partida sea resoluble. Para ello, la primera `Ficha` tendrá `BLANCO` en su izquierda y un color aleatorio en la derecha. La segunda, a su izquierda el mismo color que la primera tenía a la derecha y un color aleatorio a la derecha, y así sucesivamente hasta la última que tendrá `BLANCO` a su derecha. Ejemplo con 5 fichas: `[B M][M R][R B][B V][V B]`

f) Un método `comprobarConexiones` que comprueba todas las `fichas` mirando para cada `Ficha` si está conectada con la de su izquierda y la de su derecha, rellenando la lista `conectada` de esa `Ficha` convenientemente. Si todas las fichas están conectadas, devolverá `true`, si hay alguna sin conectar, devolverá `false`.

g) Un método `comprobarPosicion` que recibe un entero y comprueba si es una posición válida de la lista de `fichas`. Devolverá `true` o `false`, según sea válida o no. Si no es válida imprimirá además por pantalla: `¡Posición incorrecta!`

h) Un método `moverFicha` que recibe la posición inicial de la `Ficha` y la posición de destino. Primero comprueba si ambas posiciones son correctas, si no lo son, no hace nada. Si lo son coloca la `Ficha` de la posición inicial en la posición de destino, **moviendo** el resto convenientemente. Ejemplo:

Si tenemos `[B M][M V][V R]` y aplicamos `moverFicha(0,2)` tendremos `[M V][V R][B M]`

Debe funcionar tanto si el origen es menor que el destino como si es mayor (tal y como se muestra en el ejemplo de la primera página).

Cada vez que se haga un movimiento se debe incrementar en 1 el atributo `movimientos`.

i) Un método `barajarFichas` que **intercambie** aleatoriamente tantas `fichas` como longitud tiene el array `fichas`, de forma que lo desordene. Al contrario que el método anterior este método tiene que intercambiar las fichas dos a dos, sin mover el resto de fichas no implicadas en el cambio.
Ejemplo:
Si tenemos `[B M][M V][V R]` e intercambiamos la 0 y la 2 tendremos `[V R][M V][B M]`
En este caso, como tenemos 3 fichas habría que hacer tres intercambios aleatorios como el mostrado.

j) Un método `str` que devuelva el número de movimientos y el estado actual del `Tablero`, tal y como muestra el ejemplo de la primera página.

Crear un programa principal que pida al usuario que introduzca por teclado el número de fichas del `Tablero` y lo cree. A continuación debe rellenarlo, barajar las fichas, comprobar las conexiones entre fichas e imprimirlo. Luego irá preguntando al usuario los movimientos que quiere realizar, comprobando cada vez si todas las fichas están conectadas. En caso afirmativo terminará, si no, volverá a pedir un movimiento (debe comportarse como el ejemplo de la primera página)