Objetivos Herencia. Utilización de interfaces y clases abstractas.

Índice

1. Interfaces

Una **interfaz** se define del mismo modo que una clase cambiando `class` por `interface`. Los métodos que define la interfaz no tienen cuerpo y terminan con un punto y coma tras la lista de parámetros. Es responsabilidad de las clases que la implementen definir el cuerpo de los métodos. Se pueden declarar variables dentro de una interfaz y son `final` y `static`. Una interfaz se puede declarar como `public` o sin modificador de acceso (por lo que solo podrá ser vista por clases que estén en el mismo paquete o en el mismo directorio).

Si no se pueden crear objetos de una interfaz, ¿cual es su utilidad?. La utilidad de una interfaz es que define los requisitos mínimos que debe cumplir cualquier objeto que creemos a partir de una clase que la implemente. Con ello estamos seguros que el objeto tiene como mínimo el conjunto de métodos que declara la interfaz, pero cada clase que la implemente especifíca cómo se llevan a cabo las tareas que se han definido en la interfaz.

La diferencia entre una `interface` y una clase abstracta, es que ésta última puede definir la implementación para alguno de sus métodos, con lo cual las clases que la extiendan heredaran el comportamiento definido en estos métodos y tienen libertar para especificar otros. Sin embargo en el caso de la interfaz, podríamos decir que la clase que la implemente se compromete a implementar todos los métodos impuestos por la interfaz.

Podríamos buscar la siguiente similitud con el mundo real: todos los coches disponen de un conjunto de elementos comunes: el volante (al que podemos asociar un método girar), una serie de pedales (a los que podemos asociar los métodos pisar y soltar), un cambio de marchas (al que podemos asociar el método cambiar pasando como argumento la marcha hacia la que queremos cambiar), etc. Aunque básico, con esto (y un motor) se puede hacer funcionar un coche. Estas *restricciones* se pueden definir en un interface:

```
public interface Coche{
   void girar(float grados);
```

```
    void pisarEmbrague();
    void pisarAcelerador();
    void pisarFreno();
    void soltarEmbrague();
    void cambiarMarcha(int n);
}
```

Un determinado fabricante, llamémosle X, decide fabricar un coche, llamémosle CocheX, y por lo tanto debe decidir cómo realizar cada una de esas acciones. Un ejemplo se muestra en la siguiente porción de código, donde se muestra la elección del fabricante para llevar a cabo la acción `girar(···)`.

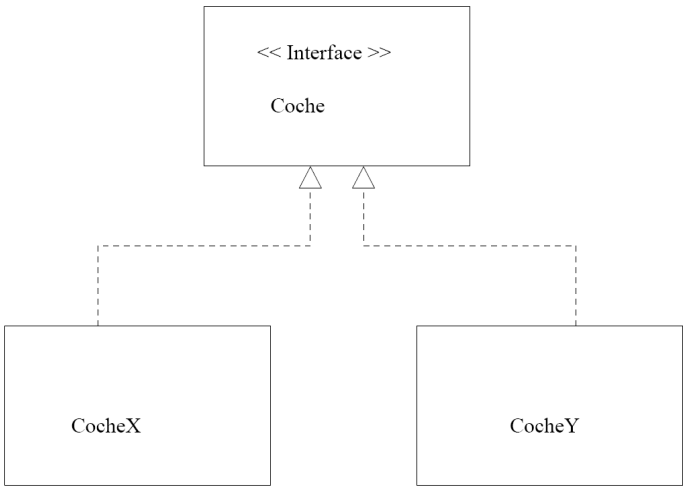
```
public class CocheX implements Coche{
    // Elementos y comportamiento particular
    void girar(float grados){
        si el giro es violento activar ESP
        si hay peligro de deslizamiento activar el ASR
        sino girar las ruedas
    }
    // El resto de las acciones definidas en la interfaz
}
```

Otro fabricante Y, con su modelo CocheY decide realizar estas acciones de otro modo, tal y como se muestra a continuación.

```
public class CocheY implements Coche{
    // Elementos y comportamiento particular
    void girar(float grados){
        girar las ruedas
    }
    // El resto de las acciones definidas en la interfaz
}
```

Lo importante es que para ser un `Coche`, las dos clases `CocheX` y `CocheY` deben implementar todos y cada uno de los requisitos impuestos por la interfaz. Con esto nos aseguramos de que todas aquellas clases que implementen `Coche` realizan como mínimo un conjunto común de acciones. Desde el punto de vista del usuario esto es importante, ya que una vez conocidas cuales son las acciones que se pueden llevar a cabo sobre un coche (independientemente de cual haya sido la solución tecnológica aplicada para realizar esas acciones), podrá conducir cualquiera siempre que sea un `Coche`, independientemente del que se conduzca en un instante dado.

La siguiente figura muestra cómo se puede representar esa jerarquía mediante un diagrama. Las flechas discontinuas indican que se trata de una implementación.



Resumiendo

Una interfaz declara (pero no implementa) las acciones mínimas que poseerá un objeto creado a partir de una clase que la implemente. Es responsabilidad de las clases que implementan la interfaz definir cómo se llevan a cabo todas y cada una de las acciones declaradas por la interfaz.

2. Clases abstractas

Java permite definir uno o varios métodos en una clase sin implementarlo/s declarandolo/s como `abstract`. Un método abstracto no tiene cuerpo, simplemente tiene su definición y finaliza con un punto y coma. Hay unas cuantas reglas a tener en cuenta al trabajar con métodos abstractos y las clases abstractas que los contienen:

- Una clase que contenga un método abstracto es también abstracta y debe ser declarada como tal.
- Se pueden crear objetos de una subclase de una clase abstracta solo si sobreescribe todos y cada uno de los métodos de la superclase y proporciona la implementación para todos ellos.
- Si una subclase de una clase abstracta no implementa todos los métodos abstractos que hereda, esta subclase es también abstracta.
- Métodos `static`, `private` y `final` no pueden ser abstractos ya que estos métodos no pueden ser sobreescritos por las subclases. Una clase declarada como `final` no puede ser abstracta ya que una clase `final` no se puede extender.
- Una clase puede ser declarada abstracta incluso si no define ningún método abstracto. Haciendo esto se indica que la implementación es incompleta y sirve como una superclase de una o más subclases que completarán la implementación. No es posible crear objetos de esta clase.

A diferencia de las interfaces, una clase abstracta puede tener un constructor ya que una parte de sus métodos pueden tener la implementación.

A continuación se muestra un ejemplo de clase abstracta y de una subclase.

```
public abstract class ClaseA{
   private int x;

   public ClaseA(int param){
      x = param;
   }

   public abstract void getX();

}

public class ClaseB extentds ClaseA{
   public ClaseB(int param){
      super(param)
   }

   public void getX(){
      return x;
   }
}
```

3. Collections Framework

Hay situaciones en las que es necesario almacenar una gran cantidad de datos. La elección del tipo de estructura que se utilice dependerá del problema que se esté intentando resolver. ¿Se necesita buscar de forma sencilla entre los datos? ¿Es necesario mantener la estructura siempre ordenada? ¿Se necesita una estructura que permita acceder a los datos de forma aleatoria? Java ofrece una serie de interfaces y clases que permiten trabajar con diferentes contenedores de datos en lo que se conoce como *Collections Framework*.

En la segunda práctica vimos dos de estas clases: `LinkedList` y `Hashtable`. En esta práctica veremos algunas de sus interfaces y veremos también otras clases.

3.1. Collection

```
public interface Collection
```

La interfaz fundamental de todas las clases de colección es `Collection`. Comentaremos dos de los métodos que define (el resto se pueden consultar en el API):

- `public boolean add(Object obj)` que añade un objeto a la colección si se puede. Por ejemplo hay algunos contenedores de datos que no permiten que se añadan dos objetos iguales, en cuyo caso el método devuelve false.
- `public Iterator interator()` devuelve un objeto que implementa la interfaz `Iterator`. Un objeto de este tipo se puede utilizar para recorrer los elementos del contenedor uno a uno.

3.2. Iterator

```
public interface Iterator
```

La interfaz `Iterator` dispone de tres métodos:

- `public Object next()` que devuelve el siguiente objeto y avanza.
- `public boolean hasNext()` que devuelve `true` si existen más objetos o `false` si no.
- `public void remove()` que borra el objeto que ha sido devuelto por la llamada a `next()`. Para utilizar `remove()` hay que haber utilizado antes `next()`.

3.3. AbstractCollection

```
public abstract class AbstractCollection extends Object implements Collection
```

Con el fin de facilitar realizar nuevas clases que implementen `Collection` hay una clase abstracta intermedia `AbstractCollection` que implementa alguno de sus métodos pero deja otros sin implementar para que lo hagan sus subclases. Para implementar una colección no modificable, el programador necesita extender esta clase y proporcionar la implementación a los métodos

- `public Iterator iterator()`
- `public int size()`

si por el contrario se desea que la colección sea modificable, el programador debe sobreescribir el método `public boolean add(Object o)`.

Se recomienda además que se proporcione un Constructor vacío y un constructor que reciba un objeto `Collection` de donde se obtengan los datos.

3.4. AbstractSet

```
public abstract class AbstractSet extends AbstractCollection implements Set
```

Esta clase proporciona un esqueleto de implementación de la interfaz `Set` para facilitar la implementación de esta interfaz. Un `Set` es básicamente idéntico a `Collection` salvo que `Set` no puede tener elementos duplicados.

El proceso de implementar `Set` utilizando `AbstractSet` es idéntico al de implementar `Collection` extendiendo `AbstractCollection`, excepto que todos los métodos y constructores en las subclases de esta clase deben cumplir la restricción de que no existan elementos duplicados.

3.5. TreeSet

```
public class TreeSet extends AbstractSet implements SortedSet, Cloneable, Serializable
```

Esta clase implementa el interfaz `Set`. Esta clase garantiza que el conjunto de datos estará ordenado en orden creciente de acuerdo al modo en el que se haya definido la comparación entre los objetos que se van a almacenar (utilizando un objeto que implemente la interfaz `Comparator` o haciendo que los objetos a almacenar implementen la interfaz `Comparable`).

La implementación que proporcionan garantiza un coste $\log(n)$ para las operaciones básicas (`add`, `remove` y contains).

3.6. Almacenes utilizados por estas clases

Un aspecto importante es que casi todos los tipos de colecciones utilizan algún almacén de datos oculto como por ejemplo un array. El almacén de datos crea una representación interna que expuesta al exterior mediante los diferentes métodos de la colección. Algunas colecciones utilizan otros objetos `Colection` para almacenar los datos. Por ejemplo `TreeSet` utiliza un `TreeMap` para almacenar los datos (en la primera línea de la documentación de `TreeSet` se puede leer lo siguiente *This class implements the Set interface, backed by a TreeMap instance...*).

3.7. Collections

Esta clase tiene varios métodos estáticos que reciben o devuelven objetos del tipo `Collection`. Entre otros, hay métodos para encontrar el máximo y el mínimo, ordenar listas, buscar elementos en listas, etc.

3.8. Más información

Además del API donde se puede obtener información detallada de todos los métodos que definen estas clases e interfaces, en el fichero `docs\guide\collections\reference.html` se pueden ver todas las clases que forman parte del *Collections Framework*.

4. Tareas

Se ha desarrollado un paquete con una serie de clases que pueden ser utilizadas como base para crear una aplicación en la que se puedan almacenar curvas como una serie de puntos. Además define algunos métodos como por ejemplo la interpolación lineal. Este paquete define las siguientes clases e interfaces

```
/**  Clase con la que se representara un punto en el plano.
 *   Deseamos que la clase implemente Comparable (de java.util) ya
```

```
 *   que los puntos se almacenaran ordenados (de menor a mayor x).
 * @author Lenguajes de Programacion
 *
 */
public abstract class Punto2D implements Comparable{
    protected double x;
    protected double y;


    /** Constructor de Punto2D
     * @param vx coordenada x
     * @param vy coordenada y
     */
    public Punto2D(double vx, double vy){
        x = vx;
        y = vy;
    }

    /** Devuelve la coordenada x
     * @return coordenada x
     */
    public double getX(){
        return x;
    }

    public double getY(){
        return y;
    }

    /** Puesto que vamos a utilizar un AbstractSet para almacenar
     * los puntos, y esta estructura no admite dos objetos iguales,
     * exigimos que las clases que extiendan a estas
     * especifiquen si dos objetos son iguales.
     */
    abstract public boolean equals(Object o);

}
```

```
/** Esta interface define los metodos que debe tener un
 *  una clase que la implemente. Asi nos aseguramos que
 *  podemos utilizar estos metodos en la clase Curva.
 * @author Lenguajes de Programacion
 *
 */
public interface DatosCurva{
    /** Devuelve el Punto cuya abscisa tiene el valor minimo*/
    public Punto2D minimoValorX();

    /** Devuelve el Punto cuya abscisa tiene el valor maximo*/
    public Punto2D maximoValorX();

    /** Devuelve el Punto cuya ordenada es minima*/
    public Punto2D minimoValorY();

    /** Devuelve el Punto cuya ordenada es maxima*/
    public Punto2D maximoValorY();

    /** Devuelve la pendiente dados dos puntos*/
    public double pendiente(Punto2D p1, Punto2D p2);
}
```

```
import java.util.AbstractSet;
import java.util.Iterator;

/** Clase que sirve para tener una representacion apropiada de una curva
 *  como un conjunto de pares de puntos.Es una clase intermedia que ofrece
 *  ciertas funciones para simplificar el trabajo de sus subclases.
```

```
 *  El hecho de que sea un AbstractSet implica que no se podran insertar
 *  dos objetos iguales.
 *  @author Lenguajes de Programacion
 */
public abstract class Curva extends AbstractSet implements DatosCurva{

    /** Comprueba si un punto x esta dentro del rango de valores x de la curva
     *  Aqui se utiliza el hecho de que se pueden utilizar los metodos
     *  minimoValorX y maximoValorX ya que esta clase implementa a DatosCurva.
     *  Como en esta clase no se implementan, lo tendran que realizar las
     *  subclases.
     *
     * @param xVal valor x a comprobar
     * @return true si esta dentro false si no
     */
    public boolean valorEnRango(double xVal){
        boolean condicion = true;

        Punto2D p1 = minimoValorX();
        Punto2D p2 = maximoValorX();

        if ((xVal<p1.getX()) || (xVal>p2.getX()))
           condicion = false;
        return condicion;
    }


    /** Metodo que se utiliza para encontrar el ultimo punto
     *  para el que se cumple que xVal > punto.getX()
     *  Puesto que la clase extiende a un AbstractSet se tendra
     *  acceso a un iterador.
     *  Aqui se asume que los datos estan ordenados.
     *  Por esto se ha elegido el AbstractSet.
     *  @param xVal la coordenada x
     *  @return el punto
     */
    private Punto2D ultimoPuntoMenorQue(double xVal){
        Punto2D p, pMenor = null;
      boolean encontrado = false;

        if (valorEnRango(xVal)){
            Iterator i = iterator();
            while ((i.hasNext()) && (!encontrado)){
               p = (Punto2D)i.next();
               if (p.getX()<xVal)
                  pMenor = p;
               else
                  encontrado = true;
            }
        }

        return pMenor;

    }

    /** Metodo que se utiliza para encontrar el primer punto
     *  para el que se cumple que xVal < punto.getX()
     *  Puesto que la clase extiende a un AbstractSet se tendra
     *  acceso a un iterador.
     *  Aqui se asume que los datos estan ordenados.
     *  Por esto se ha elegido el AbstractSet.
     * @param xVal la coordenada x
     * @return el punto
     */
    private Punto2D primerPuntoMayorQue(double xVal){
        Punto2D p, pMayor = null;
        boolean encontrado = false;

        if (valorEnRango(xVal)){
            Iterator i = iterator();
```

```
            while ((i.hasNext()) && (!encontrado)){
                p = (Punto2D)i.next();
                if (p.getX()>xVal){
                    pMayor = p;
                    encontrado = true;
                }
            }
        }

        return pMayor;

    }

    /** Funcion que sirve para encontrar el valor interpolado dada la
     *  coordenada x. Se realiza una interpolacion lineal. Utilizamos
     *  el metodo pendiente(.) ya que la clase implementa a DatosCurva.
     *  Como en esta clase no se implementa lo deberá hacer la subclase.
     * @param xVal coordenada x
     * @return el valor y correspondiente
     * @throws Exception si la coordenada x no esta en el dominio de los puntos.
     */
    public double interpola(double xVal) throws Exception{
        Punto2D p1, p2;
        double yInterp=0;

        if (valorEnRango(xVal)){
            p1 = ultimoPuntoMenorQue(xVal);
            p2 = primerPuntoMayorQue(xVal);
            yInterp =  p1.getY() + pendiente(p1,p2)* (xVal-p1.getX());
       }
       else
          throw new Exception("El valor de x no esta en el dominio de los puntos.");
        return yInterp;
    }

    /** En este metodo utilizamos los metodos definidos en la interface
     *  DatosCurva. Como en esta clase no se implementan, lo tendra que
     *  hacer la subclase.
     *
     */
    public void datosSobreCurva(){
        System.out.println("Punto con minima abscisa: " + minimoValorX());
        System.out.println("Punto con maxima abscisa: " + maximoValorX());
        System.out.println("Punto con minima Ordenada: " + minimoValorY());
        System.out.println("Punto con maxima Ordenada: " + maximoValorY());
    }
}
```

El paquete está en el fichero practica4.jar.

Hay que realizar una serie de clases para completar una jerarquía con clases que implementen los métodos definidos por las interfaces y que implementen los métodos abstractos definidos por las clases abstractas. Para ello hay que realizar las siguientes clases:

- `public class Punto extends Punto2D`
- `public class Curva1 extends Curva`

La clase `Curva1` debe tener un atributo del tipo `TreeSet` para almacenar los puntos.

Hay que observar cuidadosamente cual es la jerarquía de clases e interfaces para determinar cuales son los métodos que deben ofrecer estas dos clases (si hay algún método que debería estar y no está el compilador lo avisará).

Se proporcionan además del paquete los siguientes ficheros:

- `puntos.txt` fichero con 629 puntos correspondientes a tomar 629 valores entre $[0, 2\pi]$ como abscisas y como ordenadas los valores de la función $f(x) = cos(x)$ en esos puntos. Este fichero

contiene líneas con los pares de coordenadas separadas por comas.

- `Aplicacion.java` la clase que contiene el `main()` y que debe ser utilizada sin modificación. En el `main()` se lee el fichero anterior almacenando los datos en un `LinkedList` A continuación se crea un objeto del tipo `Curva1` a partir de la LinkedList quedándonos con una referencia del tipo `Curva`. Utilizando esta referencia podemos obtener información sobre la curva o realizar una interpolación.