Programación Concurrente

Clases abstractas vs Interfaces

Una interfaz puede considerarse similar a una clase completamente abstracta.

En muchos casos una interfaz se utiliza de la misma manera que una clase abstracta.

De una interfaz o de una clase abstracta no se pueden crear objetos.

¿Y la diferencia?

Como Java no permite herencia múltiple – una clase sólo puede extender una superclase – esto dificulta que una clase se adecue a más de un comportamiento.

Con el uso de interfaces se mejora esta situación levemente, dado que se permite que una clase implemente una o más interfaces para resolver el problema de mezclar diversos comportamientos en un mismo tipo de objeto.

Una clase abstracta ofrece comportamientos comunes a objetos del mismo tipo a través del mecanismo de la herencia.

La implementación de una interfaz permite a un objeto comportamientos que no dependen de su jerarquía de clases.

Una interfaz se diferencia de una clase abstracta porque una interfaz sólo puede contener constantes y métodos abstractos. Una clase abstracta puede contener métodos concretos, una interfaz no.

En una interfaz todos los atributos son por defecto **public final static** (constantes) y todos los métodos son **public abstract.** Esto quiere decir que una clase abstracta puede contener atributos variables pero una interfaz no.

CLASES ABSTRACTAS

Una clase abstracta representa una entidad, en la jerarquía de clases, que usualmente no está completamente definida para ser útil por sí misma. Su propósito es ofrecer descripciones parciales que puedan ser heredadas por otras clases que se encargan de concretar esas especificaciones.

Programación Concurrente

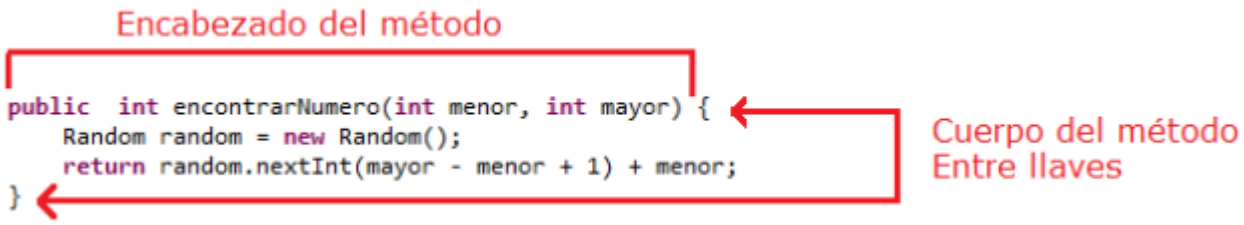
Una clase abstracta es una clase declarada ***absract*** y puede contener o no métodos abstractos. Las clases abstractas no se pueden instanciar pero sirven de superclases para factorizar el comportamiento común que otras clases puedan heredar de ellas.

Cualquier clase que contenga uno o más métodos abstractos se debe declarar abstracta.

MÉTODOS ABSTRACTOS

En Programación Orientada a Objetos (**POO**) las clases son los "*moldes*" a partir de los cuales se fabrican los objetos (instancias). Las clases se componen de atributos (datos) y métodos (funcionalidad).

Un método se compone de dos (2) partes básicas: encabezado (declaración) y cuerpo.



En la declaración se encuentran:

- La visibilidad (qué otras clases y objetos pueden invocar al método) *private*, *public*, *protected*. (por defecto es de paquete/package)
- El tipo de retorno, por ejemplo, *void*, *int*, *String*.
- Nombre del método (identificador) *encontrarNumero*
- La cantidad y el tipo de parámetros que acepta el método: *(int menor, int mayor)*

El cuerpo de un método va encerrado entre llaves *{ ... }* y contiene las sentencias algorítmicas que definen su funcionalidad.

Un método abstracto es un método sin cuerpo.

```
public abstract class ClaseAbstracta {

    public abstract int encontrarNumero(int menor, int mayor);

}
```

Ahora el método *encontrarNumero* es abstracto y pertenece a una clase abstracta. Nótese que el método termina en punto y coma (;) y no tiene las llaves del cuerpo {}.

Las clases que hereden de ***ClaseAbstracta*** deben implementar el método ***encontrarNumero*** o en su defecto ser declaradas clases abstractas.

```
1  import java.util.Date;
2  import java.util.Random;
3
4
5  public class ClaseInstanciable extends ClaseAbstracta {
6
7      @Override
8      public int encontrarNumero(int menor, int mayor) {
9          Date fechaHoy = new Date();
10         Random random = new Random(fechaHoy.getTime());
11         return random.nextInt(mayor - menor + 1) + menor;
12     }
13
14 }
15
```

La clase ***ClaseInstanciable*** hereda de clase ***ClaseAbstracta*** y debe ofrecer la implementación del método ***encontrarNumero***. Nótese que en esta clase el método tiene cuerpo.

EJEMPLO INTERFAZ

A continuación se muestra un ejemplo (incompleto) del uso de una interfaz.

Se define una interfaz “TipoElemento”, que se utilizará para la implementación de una clase “Pila”. De esta forma se logra que objetos de cualquier clase que implemente la interfaz TipoElemento puedan utilizarse como elemento para la pila.

Programación Concurrente

```
public interface TipoElemento {
    public boolean menor(TipoElemento elem);
    public boolean equals (TipoElemento elem);
    public String aCadena ();
}
```

```
public class Pila {
    // … ;
    public Pila() {
        //...
    }
    // ...
    public void apilar (TipoElemento elem){
        // ...
        }
    public TipoElemento tope(){
        // ...
     }
    // …

}
```

Se define la clase Empleado, que será utilizada para trabajar con una pila de empleados. Para ello la clase Empleado debe implementar la interfaz, o sea dar una implementación para cada método indicado en la interfaz, y además puede agregar otros métodos propios.

```
public class Empleado implements TipoElemento {
    String nombre;
    // …

    public Empleado(String cad){
        nombre = cad;
    }

    public boolean menor(TipoElemento elem){
      return (nombre.compareTo(((Empleado) elem).nombre))<0;
    };
```

```
    public boolean equals (TipoElemento elem){
       // …
    };

    public String aCadena (){
      // ...
      return nombre;
    };

}
```

Dado que lo que devuelve la operación “tope()” es TipoElemento es necesario hacer un casteo a “Empleado” al aplicarla.

```
 public class TestInterfaz

    public static void main(String[] args) {

      Pila laPila = new Pila();

        //...
      Empleado unEmp = new Empleado("Juan");
      laPila.apilar(unEmp);

       System.out.println(((Empleado)laPila.tope()).aCadena());
    }

}
```