El futuro digital es de todos

MinTIC

Universidad Pontificia Bolivariana

Vigilada Mineducación

Mision TIC 2022

Interfaces

Interfaces

Una interface comprende un grupo de prototipos de métodos públicos sin implementar código, de similar forma que los prototipos de métodos de las clases abstractas. Además, se puede declarar constantes con visibilidad public, static y final con su respectiva inicialización de datos. Los métodos de una interfaz representan un comportamiento común de un grupo de objetos de varios tipos de clases y que no necesariamente forman parte de una misma jerarquía. A continuación la sintaxis de interface:

```
public interface Nombre_Interface {…}
```

En una primer mirada, las interfaces son similares a las clases, la diferencia más obvia radica en que sus definiciones de métodos no incluyen cuerpos. Por lo tanto, se parecen a las clases abstractas en las que todos sus métodos son abstractos.

Interfaces

Las interfaces en Java tienen una cantidad de características importantes:

- ✓ En el encabezado de la declaración se usa la palabra clave **interface** en lugar de **class.**
- ✓ Todos los métodos de una interfaz son abstractos; no se permiten métodos con cuerpo. No es necesaria la palabra clave abstract.
- ✓ Las interfaces no contienen ningún constructor.
- ✓ Todos los métodos de una interfaz tienen visibilidad pública. No es necesario declarar la visibilidad: por ejemplo, no es necesario que cada método contenga la palabra clave public.

En una interfaz, sólo se permiten los campos constantes (campo público, estático y final). Pueden omitirse las palabras clave public, static y final pero todos los campos, igualmente, serán tratados como públicos, estáticos y finales. Una clase puede derivar de una interfaz de la misma manera en que deriva de una clase. Sin embargo, Java utiliza una palabra clave diferente, **implements**, para la herencia a partir de interfaces.

Interfaces

Se dice que una clase implementa una interfaz si incluye una cláusula **implements** en su encabezado. Por ejemplo:

```
public class Zorro extends Animal implements Dibujable
{
   ...
}
```

Como en este caso, si una clase extiende a una clase e implementa una interfaz, entonces la cláusula extends debe escribirse primero en el encabezado de la clase.

Interfaces

Las clases que implementan una interface deben definir el código de los métodos de dicha interface, asumiendo de esta manera una conducta determinada. Una clase, a diferencia de la herencia, puede implementar una o varias interfaces separadas por una coma "," y se ubica a la derecha del nombre de clase o en caso de derivación, debe ser a la derecha del nombre de la superclase.

```
public class Circulo extends Figura implements Dibuja, Mueve {
  //...
}
```

Diferencias entre una interface y una clase abstracta

•Una de las diferencias entre clase abstracta e interface es que todos los métodos de la interface no tienen definición de su código, mientras que en la clase abstracta al menos un método es abstracto sin definición de código; eso implica que otros métodos de la clase sí implementan código.

• Los métodos de una interface son públicos y abstractos por defecto, en el caso de no indicarse explícitamente.

• Los datos sólo pueden ser constantes públicas, estáticas y finales.

• Las interfaces no declaran métodos constructores, debido a que sólo se declaran constantes.

• Las interfaces se usan para publicar el comportamiento común de clases de distinta jerarquía.

• También es posible aplicar el polimorfismo de forma similar a como se aplica en herencia de clases.

Una interface public requiere definirse en un archivo *.java con el mismo nombre. Las interfaces package son cuando no se especifica ningún modificador, la visibilidad es a nivel del paquete y puede colocarse en el mismo archivo de una clase o interfaz public.

¿Clase abstracta o interfaz? En algunas situaciones se tiene que elegir entre usar una clase abstracta o una interfaz. Algunas veces la elección es fácil: cuando se pretende que la clase contenga implementaciones para algunos métodos necesitamos usar una clase abstracta.

En otros casos, tanto la clase abstracta como la interfaz pueden hacer el mismo trabajo. Si tenemos que elegir, es preferible usar interfaces. Si proveemos un tipo mediante una clase abstracta, las subclases no pueden extender ninguna otra clase; dado que las interfaces permiten la herencia múltiple, el uso de una interfaz no crea tal restricción. Por lo tanto, el uso de interfaces da por resultado una estructura más flexible y más extensible.

Interfaces como tipos

Cuando una clase implementa una interfaz no hereda ninguna implementación de ella, pues las interfaces no pueden contener cuerpos de métodos. Entonces, la pregunta que cabe es: ¿qué ganamos realmente al implementar interfaces?

Cuando presentamos la herencia pusimos énfasis en dos grandes beneficios de la herencia:

La subclase hereda el código (la implementación de métodos y campos) de la superclase. Esto permite la reutilización de código existente y evita la duplicación de código.

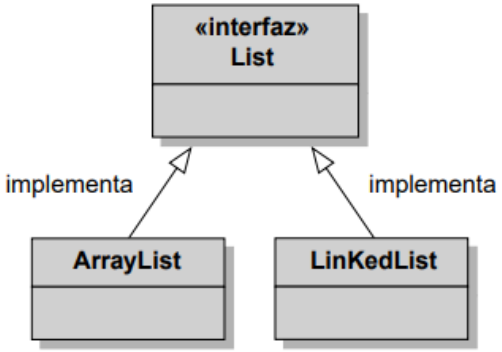
La subclase se convierte en un subtipo de la superclase. Esto permite la existencia de variables polimórficas y la invocación polimórfica de métodos. En otras palabras, permite que los casos especiales de objetos (instancias de subclases) se traten de manera uniforme (como instancias del supertipo).

Las interfaces no brindan el primer beneficio (ya que no contienen ninguna implementación), pero sí ofrecen el segundo. Una interfaz define un tipo tal como lo hace una clase. Esto quiere decir que las variables pueden ser declaradas del tipo de la interfaz, aun cuando no pueda existir ningún objeto de tal tipo (sólo de los subtipos).

Interfaces como especificaciones

En este capítulo hemos introducido las interfaces con el sentido de implementar herencia múltiple en Java. Este es un uso importante de las interfaces, pero existen otros. La característica más importante de las interfaces es que separan completamente la definición de la funcionalidad (la clase «interfaz» en el sentido más amplio de la palabra) de su implementación. U

Un buen ejemplo de cómo pueden usarse las interfaces en la práctica se puede encontrar en la jerarquía de las colecciones de Java. La jerarquía de colecciones define, entre otros tipos, la interfaz List y las clases ArrayList y LinkedList. La interfaz List especifica la funcionalidad total de una lista sin aportar ninguna implementación. Las subclases LinkedList y ArrayList proveen dos implementaciones diferentes para la misma interfaz.



Esto es interesante porque las dos implementaciones difieren enormemente en la eficiencia de algunas de sus funciones. Por ejemplo, el acceso aleatorio de elementos situados en el medio de una lista es mucho más rápido en el ArrayList, sin embargo la inserción o la eliminación de elementos puede ser mucho más rápida en la LinkedList.

La decisión de cuál de las implementaciones resulta mejor para una aplicación determinada puede ser difícil de juzgar anticipadamente, depende mucho de la frecuencia relativa con que se lleven a cabo ciertas operaciones y algunos otros factores.

En la práctica, la mejor forma de descubrir cuál es la mejor es probando: implementar la aplicación con ambas alternativas y medir el rendimiento. La existencia de la interfaz List facilita esta prueba. Si en lugar de usar un ArrayList o una LinkedList como tipo de variable y tipo de parámetro usamos siempre List, nuestra aplicación funcionará independientemente del tipo específico de lista que estemos usando realmente. Debemos usar el nombre específico de la implementación seleccionada sólo cuando creamos una nueva lista. Por ejemplo, podemos escribir private

```
private List<Tipo> miLista = new ArrayList<Tipo>();
```

Observe que el tipo del campo es justamente List de Tipo. De esta manera, podemos modificar toda la aplicación para que use una lista enlazada con sólo cambiar ArrayList por LinkedList en un único lugar: el lugar en el que se crea la lista.

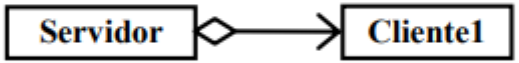
Beneficios de la interface

En una relación de agregación entre la clase A y la clase B, la clase A debe lanzar mensajes a la clase B, y para ello es necesario que a la clase A le pasen la referencia de la clase B. Una vez implementada la clase A, cualquier cambio en la clase B (por ejemplo, una mejora o cambio en la implementación de B) provoca que debiéramos cambiar todas las referencias de la clase A sobre la B.

Incluso puede que cuando se desarrolla la clase A, no se dispone de la implementación de la clase B. Para ello disponemos de dos mecanismos:

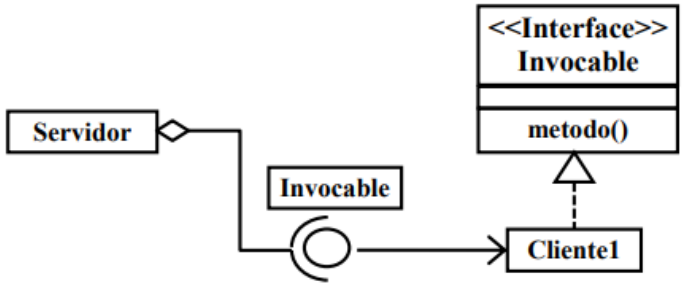
- Mediante herencia de clases. En este caso, la clase A referencia los métodos de una superclase B, pudiendo existir varias subclases de B que sirvan a tal fin.
- Mediante interfaces. Aquí, la clase A referencia los métodos de un interface, pudiendo existir un conjunto de clases que implementan ese interface.

En casos, cualquier cambio en las implementaciones de B no afectan al código de la clase A. Veamos un ejemplo concreto. En esta asociación existe una dependencia clara entre la clase Servidor y la clase Cliente.



Beneficios de la interface

Pero en este ejemplo tenemos un problema, ya que se pretende construir un servidor para que de servicio a muchos clientes y se debe romper esa dependencia. Obligar a los clientes a heredar de una clase no es una buena solución, ya que le limitamos en su diseño. La mejor solución es utilizar los interfaces. A continuación se presenta el nuevo diseño:



En este nuevo diseño, hemos roto la dependencia del servidor respecto a las diferentes implementaciones de los clientes.

Invocable.java

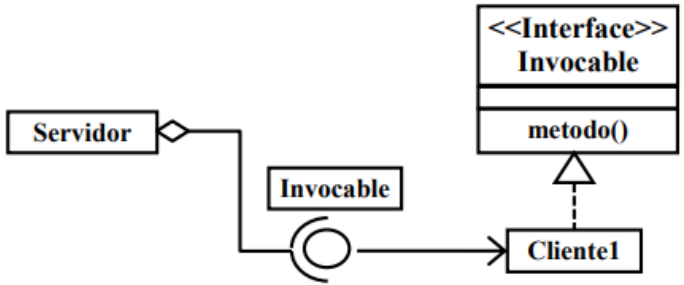
```java
public interface Invocable {
    public void newMsg(String msg);
}
```

Servidor.java

```java
public class Servidor {
   private Invocable cliente;

   public void add(Invocable cliente) {
      this.cliente = cliente;
   }

   public void remove() {
      this.cliente = null;
   }

   public void browcast() {
      if (this.cliente != null) cliente.newMsg("Nuevo mensaje");
   }
}
```

Cliente1.java

```java
public class Cliente1 implements Invocable {
   public Cliente1(Servidor servidor) {
      servidor.add(this);
   }

   @Override
   public void newMsg(String msg) {
      System.out.println("Llega un nuevo mensaje..." + msg);
   }
}
```



Polimorfismo en interfaces

De una interface no es posible crear instancias (objetos) pero si referencias a objetos de clases que implemente dicha interfaz, esto es posible gracias al polimorfismo.

```
Dibujable c= new CirculoGrafico( );
```

Ligaduras estática y dinámica

La ligadura tiene que ver con la invocación a los métodos de un objeto.

Ligadura estática

Cuando la llamada a un método se resuelve en el momento de la compilación, se efectúa un ligado estático. En el ejemplo que se muestra a continuación, se conoce en tiempo de compilación que el método a invocarse es p1.imprimir() de la clase Persona debido a que p1 es su instancia.

```
Persona p1;
p1=new Persona("0601312","Ana","Perez",12);
System.out.println(p1.Imprimir());
```

Ligadura dinámica

Cuando la llamada a un método se resuelve en el momento de la ejecución, se efectúa un ligado dinámico. En el ejemplo que se muestra a continuación, se conoce, en tiempo de ejecución, que el método a invocarse es p2.imprimir() de la clase Estudiante debido a que se invoca con una referencia de la clase padre Persona.

```
Persona p2;
P2=new Estudiante("0601322222","Pablo","Lopez",'M',"Quito",19,"Medicina", "Primero");
System.out.println(p2.Imprimir());
```

Paquetes

Los paquetes agrupan un conjunto de clases que trabajan conjuntamente sobre el mismo ámbito. Es una facilidad ofrecida por Java para agrupar sintácticamente clases que van juntas conceptualmente y definir un alto nivel de protección para los atributos y los métodos. La ventaja del uso de paquetes es que las clases quedan ordenadas y no hay colisión de nombres.

Si dos programadores llaman igual a sus clases y luego hay que juntar el código de ambos, basta explicitar a qué paquete nos referimos en cada caso. La forma de nombrar los paquetes es con palabras (normalmente en minúsculas) separadas por puntos, estableciendo una jerarquía de paquetes; la jerarquía de paquetes es independiente de la jerarquía de clases. Las clases deben almacenarse en una estructura de carpetas que coincidan con la estructura de paquetes.

Para que una clase se asocie a un paquete se realiza con la sentencia package, se debe situar antes de la clase. Algunos ejemplos de uso son:

```
public class Prueba ...

package p1.p2;
public class MiClase ...

package p1;
public class Clase2 ...
```

Para su distribución, se genera un fichero jar, con la estructura de carpetas y clases. Recordar que este fichero no hace falta descomprimirlo. Para su utilización, se debe referenciar mediante la variable de entorno CLASSPATH. Para referenciar cualquier clase en Java, se debe poner el paquete al que pertenece. Por ejemplo: java.lang.String, java.lang.Integer... Hasta ahora no ha sido necesario, porque el paquete java.lang se importa de forma automática. Para evitar poner siempre la jerarquía de paquetes, se puede utilizar la sentencia import, con ello se puede importar un paquete entero (pero no los paquetes inferiores) o una clase.

Se recomienda utilizar un IDE, como por ejemplo Eclipse. Para ejecutar una clase de un paquete desde la consola que no está referenciado mediante la variable de entorno CLASSPATH, nos debemos situar en la carpeta raíz de los paquetes y referenciar al fichero class mediante la ruta completa.