APRENDERAPROGRAMAR.COM

PARA QUÉ SIRVEN LAS INTERFACES JAVA. IMPLEMENTAR UNA INTERFAZ DEL API. VENTAJAS. EJEMPLOS BÁSICOS. (CU00697B)

Sección: Cursos
Categoría: Curso “Aprender programación Java desde cero”

Fecha revisión: 2029

Resumen: Entrega nº97 curso Aprender programación Java desde cero.

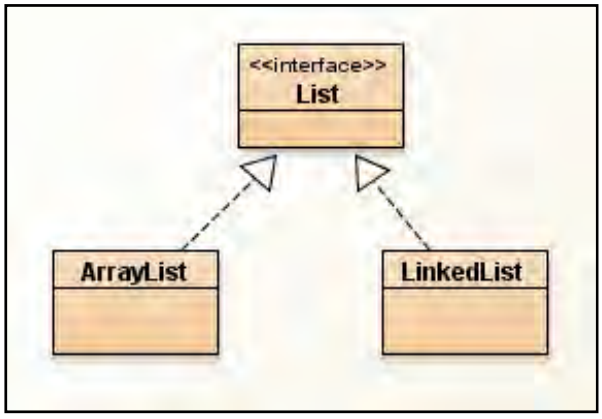
Autor: Alex Rodríguez

PARA QUÉ SIRVEN LAS INTERFACES EN JAVA

Si una interfaz define un tipo (al igual que una clase define un tipo) pero ese tipo no provee de ningún método podemos preguntarnos: ¿qué se gana con las interfaces en Java? La implementación (herencia) de una interfaz no podemos decir que evite la duplicidad de código o que favorezca la reutilización de código puesto que realmente no proveen código.

En cambio sí podemos decir que reúne las otras dos ventajas de la herencia: favorecer el mantenimiento y la extensión de las aplicaciones. ¿Por qué? Porque **al definir interfaces permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos.** En el diagrama que vimos anteriormente tanto árboles como arbustos, vehículos y personas son de tipo Actor, de modo que podemos generar código que haga un tratamiento en común de todo lo que son actores. Por ejemplo, podemos necesitar una lista de Actores. Podemos declarar una variable como de tipo Actor (aunque no puedan existir instancias de Actor) que permita referenciar alternativamente a objetos de las distintas subclases de la interfaz.

Un aspecto fundamental de las interfaces en Java es hacer lo que ya hemos dicho que hace una interfaz de forma genérica: **separar la especificación de una clase (qué hace) de la implementación (cómo lo hace).** Esto se ha comprobado que da lugar a programas más robustos y con menos errores. Pensemos en el API de Java. Por ejemplo, disponemos de la interfaz List que es implementada por las clases ArrayList y LinkedList (y también por otras varias clases).



El hecho de declarar una variable de tipo lista, por ejemplo List <String> miLista; nos dice que miLista va a ser una implementación de List, pero todavía no hemos definido cuál de las posibles implementaciones va a ser. De hecho, el código podría definir que se implementara de una u otra manera en función de las circunstancias usando condicionales. O a nivel de programación, mantendríamos la definición como List y nos permitiría comprobar el rendimiento de distintas

configuraciones (hacer funcionar miLista bien como ArrayList bien como LinkedList viendo su rendimiento). La variable declarada se crea cuando escribimos miLista = new LinkedList <String> (); o también se puede usar la sintaxis: List <String> miLista = new LinkedList <String> ();

Usar una u otra implementación puede dar lugar a diferentes rendimientos de un programa. ArrayList responde muy bien para la búsqueda de elementos situados en posiciones intermedias pero la inserción o eliminación de elementos puede ser más rápida con una LinkedList. Declarando las variables simplemente como List tendremos la posibilidad de que nuestro programa pase de usar un tipo de lista a otro tipo.

Como List es un tipo, podemos especificar los métodos para que requieran List y después enviarles como parámetro bien un ArrayList bien un LinkedList sin tener que preocuparnos de hacer cambios en el código en caso de que usáramos uno u otro tipo de lista. En esencia, usando siempre List, el único sitio donde habría que especificar la clase concreta sería donde se declara la creación de la variable, con lo cual todo nuestro código (excepto el lugar puntual donde se crea la variable) es independiente del tipo de lista que usemos y esto resulta ventajoso porque pasar de usar un tipo de lista a usar otro resultará muy sencillo.

Los métodos de ArrayList en algunos casos definen los métodos abstractos de List, y en otros casos son específicos. Recordar que en List **todos los métodos son abstractos por ser una interfaz**, aunque no se indique específicamente en la documentación del API de Java. Recordar también que List, por ser **una interfaz no tiene constructores y no es instanciable**. Al ver la documentación del API nos puede parecer una clase, pero la ausencia de constructor (aparte del propio nombre en el encabezado) delata que no se trata de una clase.

java.util

Interface List<E>

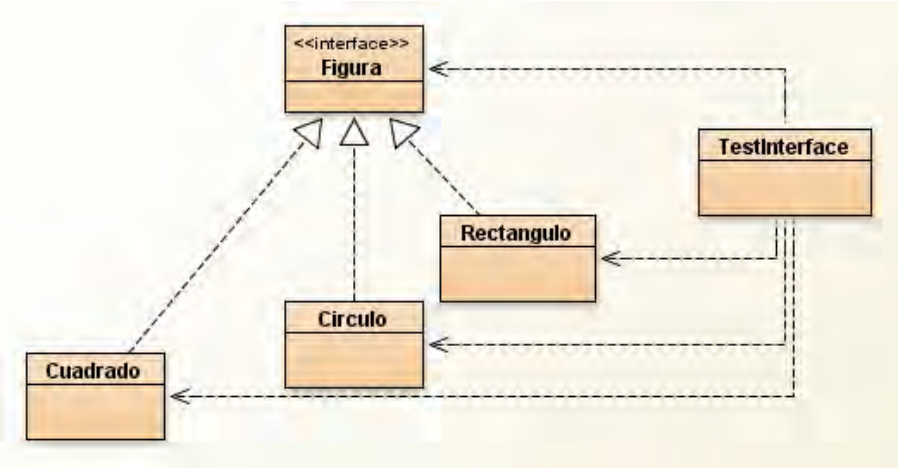
All Superinterfaces: Collection<E>, Iterable<E>

All Known Implementing Classes: AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

Otra ventaja clara de las interfaces es que nos permiten declarar constantes que van a estar disponibles para todas las clases que queramos (implementando esa interfaz). Nos ahorra código evitando tener que escribir las mismas declaraciones de constantes en diferentes clases.

EJEMPLO SENCILLO DE INTERFACE EN JAVA

Vamos a ver un ejemplo simple de definición y uso de interface en Java. Las clases que vamos a usar y sus relaciones se muestran en el esquema. Escribe el código y ejecútalo.




```
public interface Figura {  // Ejemplo aprenderaprogramar.com
float PI = 3.1416f;   // Por defecto public static  final. La f final indica que el número es float
float area();   // Por defecto abstract public
} //Cierre de la interface
```

```
public class Cuadrado implements Figura { // La clase implementa la interface Figura
private float lado;
public Cuadrado (float lado) { this.lado = lado; }
public float area() { return lado*lado; }
} //Cierre de la clase ejemplo aprenderaprogramar.com
```

```
public class Circulo implements Figura{ // La clase implementa la interface Figura
private float diametro;
public Circulo (float diametro) { this.diametro = diametro; }
public float area() { return (PI*diametro*diametro/4f); }
} //Cierre de la clase ejemplo aprenderaprogramar.com
```

```
public class Rectangulo implements Figura{ // La clase implementa la interface Figura
private float lado; private float altura;
public Rectangulo (float lado, float altura) { this.lado = lado; this.altura = altura; }
public float area() { return lado*altura; }
} //Cierre de la clase ejemplo aprenderaprogramar.com
```

```
import java.util.List; import java.util.ArrayList; //Test ejemplo aprenderaprogramar.com
public class TestInterface {
   public static void main (String [ ] Args) {
      Figura cuad1 = new Cuadrado (3.5f); Figura cuad2 = new Cuadrado (2.2f); Figura cuad3 = new Cuadrado (8.9f);
      Figura circ1 = new Circulo (3.5f); Figura circ2 = new Circulo (4f);
      Figura rect1 = new Rectangulo (2.25f, 2.55f); Figura rect2 = new Rectangulo (12f, 3f);
      List <Figura> serieDeFiguras = new ArrayList <Figura> ();
      serieDeFiguras.add (cuad1); serieDeFiguras.add (cuad2); serieDeFiguras.add (cuad3);
      serieDeFiguras.add (circ1); serieDeFiguras.add (circ2); serieDeFiguras.add (rect1); serieDeFiguras.add (rect2);
      float areaTotal = 0;
      for (Figura tmp: serieDeFiguras) {     areaTotal = areaTotal + tmp.area();    }
      System.out.println ("Tenemos un total de " + serieDeFiguras.size() + " figuras y su área total es de "  +
      areaTotal + " uds cuadradas")  } } //Cierre del main y de la clase
```

El resultado de ejecución podría ser algo así:

```
Tenemos un total de 7 figuras y su área total es de 160.22504 uds cuadradas
```

En este ejemplo **comprobamos que la interface Figura define un tipo**. Podemos crear un ArrayList de figuras donde tenemos figuras de distintos tipos (cuadrados, círculos, rectángulos) aprovechándonos del polimorfismo. Esto nos permite darle un tratamiento común a todas las figuras. En concreto, usamos un bucle for-each para recorrer la lista de figuras y obtener un área total.

IMPLEMENTAR UNA INTERFACE DEL API JAVA. EJEMPLO.

El API de Java define interfaces que aparte de usarlas para definir tipos, nosotros podemos implementar en una clase propia en nuestro código. Esto tiene cierta similitud con hacer una redefinición de un método (ya hemos visto cómo redefinir métodos como toString()), pero no es exactamente lo mismo. Para empezar, algunos métodos como toString() están definidos en la clase Object. Estos métodos declarados en la clase Object los podemos redefinir en una clase propia sin necesidad de escribir nada en cabecera de la clase, puesto que por defecto todo objeto hereda de Object. Para utilizar interfaces, como la interfaz Comparable, habremos de escribir en cabecera de la clase:

```
public class NombreDeLaClase implements Comparable <NombreDeLaClase> { … }
```

Por ejemplo *public class Persona implements Comparable <Persona>.*

¿Qué interés tiene implementar una interface del API si no nos proporciona código ninguno? Tal y como dijimos en su momento, una interface puede verse en relación a la programación como una norma urbanística en una ciudad. Si lees la documentación de la interfaz, aunque no proporciona código, sí proporciona instrucciones respecto a características comunes para las clases que la implementen y define qué métodos han de incluirse para cumplir con la interfaz y para qué servirán esos métodos. Si implementamos la interface, lo que hacemos es ajustarnos a la norma. Y si todos los programadores se ajustan a la misma norma, cuando un programador tiene que continuar un programa iniciado por otro no tiene que preguntarse: ¿qué método podré usar para comparar varios objetos de este tipo y ponerlos en orden? Y no hay que preguntárselo porque en general los programadores se ciñen a lo establecido por el API de Java: para comparar varios objetos y ponerlos en orden (“orden natural”) se implementa la interfaz Comparable y su método compareTo(). Y además, ya sabemos qué tipo ha de devolver ese método y cómo ha de funcionar, porque así lo indica la documentación de la interface.

Muchas clases del API de Java ya tienen implementada la interface Comparable. Por ejemplo la clase Integer tiene implementada esta interfaz, lo que significa que el método compareTo() es un método disponible para cualquier objeto de tipo Integer.

No podemos conocer ni todas las clases ni todas las interfaces del API de Java. No obstante, a medida que vayamos realizando programas y adquiriendo práctica con Java, nos daremos cuenta de que algunas clases e interfaces son muy usadas. A base de usarlas, iremos memorizando poco a poco sus nombres y métodos. Otras clases o interfaces las usaremos ocasionalmente y recurriremos a la consulta de documentación del API cada vez que vayamos a usarlas. Y otras clases o interfaces quizá no lleguemos a usarlas nunca.

EJERCICIO

Se plantea desarrollar un programa Java que permita representar la siguiente situación. Una instalación deportiva es un recinto delimitado donde se practican deportes, en Java interesa disponer de un método int getTipoDeInstalacion(). Un edificio es una construcción cubierta y en Java interesa disponer de un método double getSuperficieEdificio(). Un polideportivo es al mismo tiempo una instalación deportiva y un edificio; en Java interesa conocer la superficie que tiene y el nombre que tiene. Un edificio de oficinas es un edificio; en Java interesa conocer el número de oficinas que tiene.

Definir dos interfaces y una clase que implemente ambas interfaces para representar la situación anterior. En una clase test con el método main, crear un ArrayList que contenga tres polideportivos y dos edificios de oficinas y utilizando un iterator, recorrer la colección y mostrar los atributos de cada elemento. ¿Entre qué clases existe una relación que se asemeja a la herencia múltiple?

Puedes comprobar si tu respuesta es correcta consultando en los foros aprenderaprogramar.com.

Próxima entrega: CU00698B

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:

http://www.aprenderaprogramar.com/index.php?option=com_content&view=category&id=68&Itemid=188