DELEGACIÓN

CASO PRÁCTICO: Implementación de una orquesta

Versión inicial: Orquesta v0.1

Criticar la siguiente implementación de una orquesta.

> **Pista**:
> Minimizar el *acoplamiento* y maximizar la *cohesión*

```
abstract class Instrumento {
  public void tocar() { }
  public static void afinarInstrumento(Instrumento i)
  {
    // Afinar en funcion del tipo de i
    if (i instanceof Viento)
      afinarViento(i);
    else if (i instanceof Cuerda)
      afinarCuerda(i);
    // Probar que esta afinado
    i.tocar();
  }
  public static void afinarViento(Viento i)
  { System.out.println("afinal soplido"); }

  public static void afinarCuerda(Cuerda i)
  { System.out.println("afinar rasgado"); }
}
```

Orquesta v0.1 (cont.)

```
class Viento extends Instrumento {
  public void tocar()
  { soplar(); }

  public void afinar()
  { System.out.println("afinar soplido");}

  public void soplar()
  { System.out.println("soplar"); }
}
```

```
class Cuerda extends Instrumento {
  public void tocar()
  { rasgar(); }

  public void afinar()
  { System.out.println("afinar rasgado"); }

  public void rasgar()
  { System.out.println("rasgar"); }
}
```

Orquesta v0.1 (cont.)

```
import java.util.ArrayList;

public class Orquesta {
  ArrayList<Instrumento> instrumentos;
  public Orquesta() {
    instrumentos = new ArrayList<Instrumento>(3); }
  public void tocar() {
      for (int i=0; i<instrumentos.size(); i++)
        instrumentos.get(i).tocar();
  }
  public static void main(String[] args) {
    instrumentos.add(new Viento());
    instrumentos.add(new Cuerda());
    for (int i=0; i<instrumentos.size(); i++)
        Instrumento.afinarInstrumento(
              instrumentos.get(i));
    tocar();
  }
}
```

Críticas a la Orquesta v0.1

- **Acoplamiento**: el abuso de métodos `static` obliga a numerosas dependencias (`Instrumento` ⇢ `Orquesta`, `Viento`, `Cuerda`) y debería ser al revés.
- **Cohesión**: la ubicación de `main` en `Orquesta` mezcla responsabilidades de la orquesta y el cliente de prueba.

Cambio propuesto

- Usar **polimorfismo** en lugar de métodos `static` y `instanceof`

Implementación alternativa: Orquesta v0.2

Seguir criticando la implementación...

> **Pista**:
>
> Visibilidad de la implementación

```
import java.util.ArrayList;

class Orquesta {
  ArrayList<Instrumento> instrumentos;
  public Orquesta() {
      instrumentos = new ArrayList<Instrumento>(3);
  }
  public void tocar() {
      for (int i=0; i<instrumentos.size(); i++)
        instrumentos.get(i).tocar();
  }
  public void afinar(Instrumento i) {
    i.afinar();  // Metodo polimorfico
    i.tocar();   // Prueba de que esta afinado
  }
}
```

Orquesta v0.2 (cont.)

```
public class PruebaOrquesta {
  public static void main(String[] args) {
    Orquesta orquesta = new Orquesta();
    orquesta.instrumentos.add(new Viento());
    orquesta.instrumentos.add(new Cuerda());
    orquesta.instrumentos.add(new Percusion());
    for (int i=0; i<instrumentos.size(); i++)
        orquesta.afinar(orquesta.instrumentos.get(i));
    orquesta.tocar();
  }
}
```

Orquesta v0.2 (cont.)

```
abstract class Instrumento {
  public void tocar() { };
  public void afinar() { };
}

class Viento extends Instrumento {
  public void tocar() { soplar(); }
  public void afinar() { System.out.println("afinar soplido"); }
  public void soplar() { System.out.println("soplar"); }
}

class Cuerda extends Instrumento {
  public void tocar() { rasgar(); }
  public void afinar() { System.out.println("afinar rasgado"); }
  public void rasgar() { System.out.println("rasgar"); }
}

class Percusion extends Instrumento {
  public void tocar() { golpear(); }
  public void afinar() { System.out.println("afinar golpeado"); }
  public void golpear() { System.out.println("golpear"); }
}
```

Críticas a la Orquesta v0.2

- **Encapsulación**: visibilidad de `Orquesta::instrumentos` (en C++ sería `friend`)
- **Encapsulación**: el método `add` de `orquesta.instrumentos` expone la implementación de la colección (un `ArrayList`)

Cambio propuesto

- Proteger la forma de hacer altas/bajas de `Instrumento` en la colección de `Orquesta`

Implementación alternativa: Orquesta v0.3

Seguir criticando la implementación...

```
class Orquesta {
  protected ArrayList<Instrumento> instrumentos;

  public Orquesta() {
      instrumentos = new ArrayList<Instrumento>(3);
  }
  public boolean addInstrumento(Instrumento i) {
      return instrumentos.add(i);
  }
  public boolean removeInstrumento(Instrumento i) {
      return instrumentos.remove(i);
  }
  public void tocar() {
      for (int i=0; i<instrumentos.size(); i++)
        instrumentos.get(i).tocar();
  }
  public void afinar(Instrumento i) {
    i.afinar();
    i.tocar(); // Prueba de que esta afinado
  }
}
```

Orquesta v0.3 (cont.)

```
public class PruebaOrquesta {
  public static void main(String[] args) {
    Orquesta orquesta = new Orquesta();
    orquesta.addInstrumento(new Viento());
    orquesta.addInstrumento(new Cuerda());
    orquesta.addInstrumento(new Percusion());
    for (int i=0; i<orquesta.instrumentos.size(); i++)
        orquesta.afinar(orquesta.instrumentos.get(i));
    orquesta.tocar();
  }
}
```

Críticas a la Orquesta v0.3

- **Variabilidad**: ¿La colección de instrumentos será siempre lineal?
- **Flexibilidad**: la implementación `Orquesta::instrumentos` puede variar, pero...
- **Acoplamiento**: `PruebaOrquesta` sigue conociendo la implementación basada en un `ArrayList` de la colección de instrumentos de la orquesta.

Cambio propuesto

- Definir una **interfaz** para iterar en la colección de instrumentos

Implementación alternativa: Orquesta v0.4

```
import java.util.List;

class Orquesta {
  protected List<Instrumento> instrumentos;
  public Orquesta() {
      instrumentos = new ArrayList<Instrumento>(3);
  }
  public boolean addInstrumento(Instrumento i) {
      return instrumentos.add(i);
  }
  public boolean removeInstrumento(Instrumento i) {
      return instrumentos.remove(i);
  }
  public void tocar() {
    for (Iterator<Instrumento> i =
          instrumentos.iterator();
          i.hasNext(); )
      i.next().tocar();
}
  public void afinar(Instrumento i) {
      i.afinar();
      i.tocar(); // Prueba de que esta afinado
  }
}
```

```
public class PruebaOrquesta {
  public static void main(String[] args) {
    Orquesta orquesta = new Orquesta();
    orquesta.addInstrumento(new Viento());
    orquesta.addInstrumento(new Cuerda());
    orquesta.addInstrumento(new Percusion());
    for (Iterator<Instrumento> i =
          orquesta.instrumentos.iterator();
          i.hasNext(); )
        orquesta.afinar(i.next());
    orquesta.tocar();
  }
}
```

Seguir criticando la implementación...

Críticas a la Orquesta v0.4

- **Ocultación**: el atributo `instrumentos` sigue sin ser privado

Cambio propuesto

Usar delegación, interfaces y el nuevo `for` (disponible desde el JDK 1.5), que permite iterar sobre una colección que implemente la interfaz `Iterable`

Implementación alternativa: Orquesta v0.5

```
class Orquesta {
  private List<Instrumento> instrumentos;
  public Orquesta() {
      instrumentos = new ArrayList<Instrumento>(3);
  }
  public boolean addInstrumento(Instrumento i) {
      return instrumentos.add(i);
  }
  public boolean removeInstrumento(Instrumento i) {
      return instrumentos.remove(i);
  }
  public List<Instrumento> instrumentos() {
      return instrumentos;
  }
  public void tocar() {
      for (Instrumento i: instrumentos)
        i.tocar();
  }
  public void afinar(Instrumento i) {
      i.afinar();
      i.tocar(); // Prueba de que esta afinado
  }
}
```

```
public class PruebaOrquesta {
  public static void main(String[] args) {
    Orquesta orquesta = new Orquesta();
    orquesta.addInstrumento(new Viento());
    orquesta.addInstrumento(new Cuerda());
    orquesta.addInstrumento(new Percusion());
    for (Instrumento i: orquesta.instrumentos())
        orquesta.afinar(i);
    orquesta.tocar();
  }
}
```

Seguir criticando la implementación...

Críticas a la Orquesta v0.5

- **Ocultación**: la interfaz del método `instrumentos()` sigue expuesta: el cliente sabe que devuelve una `List`.
- Hemos ocultado un poco la implementación de `instrumentos` (que es una `List`), pero ¿conviene saber que es una `List`? Quizá no hemos ocultado lo suficiente.

Cambio propuesto

- Nos quedamos sólo con lo que nos interesa de `Orquesta`: que es una colección iterable.
- Eliminamos lo que no nos interesa: el resto de elementos de la interfaz `List` que implementan la forma **lineal** de almacenar los instrumentos.

Implementación alternativa: Orquesta v0.6

```
class Orquesta implements Iterable<Instrumento> {
  private List<Instrumento> instrumentos;
  public Orquesta() {
      instrumentos = new ArrayList<Instrumento>(3);
  }
  public boolean addInstrumento(Instrumento i) {
      return instrumentos.add(i);
  }
  public boolean removeInstrumento(Instrumento i) {
      return instrumentos.remove(i);
  }
  public Iterator<Instrumento> iterator() {
      return instrumentos.iterator();
  }
  public void tocar() {
      for (Instrumento i: this)
        i.tocar();
  }
  public void afinar(Instrumento i) {
    i.afinar();
    i.tocar(); // Prueba de que esta afinado
  }
}
```

```
  public class PruebaOrquesta {
     public static void main(String[] args) {
        Orquesta orquesta = new Orquesta();
        orquesta.addInstrumento(new Viento());
        orquesta.addInstrumento(new Cuerda());
        orquesta.addInstrumento(new Percusion());
        for (Instrumento i: orquesta)
           orquesta.afinar(i);
        orquesta.tocar();
     }
  }
```

Cambio de requisitos

- Supongamos que queremos poder iterar solo sobre un grupo de instrumentos de un mismo tipo (viento, cuerda, percusión). Hacerlo sobre una colección lineal es ineficiente.
- Proponemos sustituir la implementación actual (basada en una `List`) por otra (quizá más eficiente) basada en un `Map`
- Consultamos la interfaz de `Map`: `java.util.Map` o `java.util.Map<K,V>`...

  ¡Sorpresa!... `Map` no implementa `Iterable`

Pegas:

- Tenemos que implementar la interfaz `Iterable` en `Orquesta` para que el cliente siga funcionando sin cambios.
- El método `Orquesta::iterator()` queda un poco ineficiente al tener que iterar sobre todos los valores de un `Map`.

A pesar de esto, ¿construimos un `Map` en lugar de una `List` para almacenar los instrumentos?

Más de lo que necesitamos

`Map<K,V>` ofrece más de lo que necesitamos: ¡hay un `clear()` en el `Map`!

Tensión de frontera

Existe una cierta tensión proveedor-cliente en la **frontera** de una interfaz

- Los proveedores de packages y frameworks quieren ampliar aplicabilidad
- Los clientes quieren una interfaz centrada en sus necesidades particulares

Es mejor ocultar lo que no necesitamos:

- Ocultar la implementación en una interfaz
- Filtrar los métodos que no nos sirven
- Más fácil de hacer evolucionar sin impacto en el resto de la aplicación

Implementación alternativa: Orquesta v0.7

```
class Orquesta implements Iterable<Instrumento> {
  private Instrumentos instrumentos;
  public Orquesta() {
      instrumentos = new Instrumentos(3);
  }
  public boolean addInstrumento(Instrumento i) {
      return instrumentos.addInstrument(i);
  }
  public boolean removeInstrumento(Instrumento i) {
      return instrumentos.removeInstrument(i);
  }
  public Iterator<Instrumento> iterator() {
      return instrumentos.iterator();
  }
  public void tocar() {
      for (Instrumento i: instrumentos)
        i.tocar();
  }
  public void afinar(Instrumento i) {
    i.afinar();
    i.tocar(); // Prueba de que esta afinado
  }
}
```

```
public class PruebaOrquesta {
    public static void main(String[] args) {
      Orquesta orquesta = new Orquesta();
      orquesta.addInstrumento(new Viento());
      orquesta.addInstrumento(new Cuerda());
      orquesta.addInstrumento(new Percusion());
      for (Instrumento i: orquesta)
          orquesta.afinar(i);
      orquesta.tocar();
    }
}
```

Añadimos método para saber el tipo:

```
abstract class Instrumento {
  ...
  public String tipo() {
    return getClass().getSimpleName().toLowerCase();
  }
}
```

```
public class Instrumentos
       implements Iterable<Instrumento> {
  private Map<String,List<Instrumento>> instrumentos;

  public Instrumentos(int numero) {
    instrumentos =
        new LinkedHashMap<String,List<Instrumento>>(numero);
  }

  public Iterator<Instrumento> iterator() {
    List<Instrumento> todos =
        new ArrayList<Instrumento>();
    for (List<Instrumento> grupo: instrumentos.values())
      todos.addAll(grupo);
    return todos.iterator();
  }
```

```
  public boolean addInstrument(Instrumento i) {
    String tipo = i.tipo();
    List<Instrumento> grupo =
          instrumentos.get(tipo);
    if (grupo == null) {
      grupo = new ArrayList<Instrumento>();
      instrumentos.put(tipo, grupo);
    }
    return grupo.add(i);
  }
  public boolean removeInstrument(Instrumento i) {
    String tipo = i.tipo();
    List<Instrumento> grupo =
          instrumentos.get(tipo);
    if (grupo == null)
      return false;
    boolean removed = grupo.remove(i);
    if (grupo.isEmpty())
      instrumentos.remove(tipo);
    return removed;
  }
}
```

Esta implementación desacopla `Orquesta` de la estructura concreta de almacenamiento, pues la responsabilidad de ser iterable queda confinada en `Instrumentos`, que encapsula y filtra la implementación elegida (`List`, `Map`, etc.) para la colección de instrumentos.

Esto ya es más re-diseño que implementación (separación de responsabilidades)...

Resumen (sin versiones intermedias)

- Reducir acoplamiento: Evitar tipos `static` y uso de `instanceof`) en `afinarInstrumento(...)` → usar polimorfismo en `Instrumento::afinar()`
- Mejorar cohesión: `main` no pertenece a `Orquesta` → cliente de prueba aparte.
- Encapsular la colección `instrumentos` para no exponer su implementación → altas/bajas sólo vía `addInstrumento` y `removeInstrumento`
- Reducir acoplamiento a la implementación de la colección → el cliente no debe conocer `ArrayList` o incluso `List`
- Ocultar las capacidades innecesarias → `Orquesta` se ve como `Iterable<Instrumento>` (con saber cómo iterar es suficiente)
- Anticipar variabilidad real de la colección (v.g. si `List` cambia a `Map`) → introducir un agregado `Instrumentos` que encapsula la estructura y filtra operaciones

Delegación

Delegación *en horizontal* hacia otras clases cuya interfaz es bien conocida

- Los objetos miembro **delegados** son cambiables en tiempo de ejecución sin afectar al código cliente ya existente
- Alternativa más flexible que la herencia. Ejemplo: `Cola extends ArrayList` implica que una cola va a implementarse como un `ArrayList` para toda la vida, sin posibilidad de cambio en ejecución

Composición vs. Herencia

- **Composición** (delegación *en horizontal*)
  - Útil cuando hacen falta las características de una clase existente dentro de una nueva, *pero no su interfaz*.
  - Los objetos miembro privados pueden cambiarse en tiempo de ejecución.
  - Los cambios en el objeto miembro no afectan al código del cliente.
- **Herencia** (delegación *en vertical*)
  - Útil para hacer una versión especial de una clase existente, reutilizando su interfaz.
  - La relación de herencia en los lenguajes de programación *suele ser* **estática** (definida en tiempo de compilación) y no **dinámica** (que pueda cambiarse en tiempo de ejecución).

CASO PRÁCTICO: Implementación de comparadores

Comparadores

- Un requisito habitual es implementar formas de **comparación** entre objetos de un mismo tipo.
- El criterio de comparación más habitual es usar un **identificador**
- A veces es necesario comparar los objetos en función de otros criterios (por ejemplo, para ordenar una colección)

¿Cómo se proporcionan esos criterios?

Cada lenguaje tiene sus mecanismos de implementación...

Comparadores: Implementación en Java

Implementación por herencia

`java.lang.Comparable` es una interfaz implementada por `String`, `File`, `Date`, etc. y todas las llamadas *clases de envoltura* del JDK (i.e. `Integer`, `Long`, etc.)

Métodos de la interfaz `Comparable`

```
// JDK 1.4
public interface Comparable {
  public int compareTo(Object o); //throws ClassCastException
}
```

```
// JDK 1.5
public interface Comparable<T> {
  public int compareTo(T o); //throws ClassCastException
}
```

Invariantes

- Anticonmutativa:

  `sgn(x.compareTo(y)) = -sgn(y.compareTo(x))`

- Transitividad:

  `(x.compareTo(y)>0 and y.compareTo(z)>0)` $\longrightarrow$ `x.compareTo(z)>0`

  `x.compareTo(y)=0` $\longrightarrow$ `sgn(x.compareTo(z))=sgn(y.compareTo(z))` $\forall$ `z`

- Consistencia con `equals` (no obligatoria):

  `(x.compareTo(y)=0)` $\longleftarrow$ `(x.equals(y))`

Identificador de BankAccount: Implementación en Java ≥ 1.5

- Utilizando *templates* (**polimorfismo paramétrico**)
- Delegar en `compareTo` y `equals` del tipo de id *envuelto* (e.g. `String`)

```java
import java.util.*;
import java.io.*;

public final class BankAccount implements Comparable<BankAccount> {
  private final String id;
  public BankAccount (String number)  {
    this.id = number;
  }
  public String getId() { return id; }
  @Override
  public int compareTo(BankAccount other) {
    if (this == other) return 0;
    assert this.equals(other) : "compareTo inconsistent with equals.";
    return this.id.compareTo(other.getId());
  }
  @Override
  public boolean equals(Object other) {
    if (this == other) return true;
    if (!(other instanceof BankAccount)) return false;
    BankAccount that = (BankAccount)other;
    return this.id.equals(that.getId());
   }
  @Override
  public String toString() {
    return id.toString();
  }
}
```

Identificador de BankAccount: Implementación en Java ≤ 1.4

- No hay plantillas (polimorfismo paramétrico).
- La genericidad se consigue con `Object`. Hay que hacer casting.
- Cuidado con `Boolean` que no implementa `Comparable` en JDK 1.4

```
import java.util.*;
import java.io.*;

public final class BankAccount implements Comparable {
  private final String id;
  public BankAccount (String number)  {
    this.id = number;
  }
  public String getId() { return id; }
  public int compareTo(Object other) {
    if (this == other) return 0;
    assert (other instanceof BankAccount) : "compareTo comparing objects of different type";
    BankAccount that = (BankAccount)other;
    assert this.equals(that) : "compareTo inconsistent with equals.";
    return this.id.compareTo(that.getId());
  }
  public boolean equals(Object other) {
    if (this == other) return true;
    if (!(other instanceof BankAccount)) return false;
    BankAccount that = (BankAccount)other;
    return this.id.equals(that.getId());
  }
  public String toString() {
      return id.toString();
  }
}
```

Implementación por composición/delegación

Cuando una clase hereda de una clase concreta que implementa `Comparable` y le añade un campo significativo para la comparación, no se puede construir una implementación correcta de `compareTo`. La única alternativa entonces es la composición en lugar de la herencia.

Una alternativa (no excluyente) a implementar `Comparable` es pasar un `Comparator` como parámetro (se prefiere **composición** frente a **herencia**):

- Si `BankAccount` implementa `Comparable`:

```
class BankAccountComparator implements java.util.Comparator<BankAccount> {
    public int compare(BankAccount o1, BankAccount o2) {
        return o1.compareTo(o2);
    }
}
```

- Si `BankAccount` no implementa `Comparable`:

```
class BankAccountComparator implements java.util.Comparator<BankAccount> {
    public int compare(BankAccount o1, BankAccount o2) {
        return compare(o1.getId(), o2.getId());
    }
}
```

Comparadores: Implementación en Scala

En Scala se puede implementar el equivalente a la interfaz `Comparable` de Java mediante *traits*:

```
object MiApp {
  def main(args: Array[String]) : Unit = {
    val f1 = new Fecha(12,4,2009)
    val f2 = new Fecha(12,4,2019)
    println(s"$f1 es posterior a $f2? ${f1>=f2}")
  }
}

trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

```
class Fecha(d: Int, m: Int, a: Int) extends Ord {
  def anno = a
  def mes = m
  def dia = d
  override def toString(): String = s"$dia-$mes-$anno"
  override def equals(that: Any): Boolean =
    that.isInstanceOf[Fecha] && {
      val o = that.asInstanceOf[Fecha]
      o.dia == dia && o.mes == mes && o.anno == anno
    }
  def <(that: Any): Boolean = {
    if (!that.isInstanceOf[Fecha])
      sys.error("no se puede comparar" + that + " y una fecha")
    val o = that.asInstanceOf[Fecha]
    (anno < o.anno) ||
    (anno == o.anno && (mes < o.mes ||
                        (mes == o.mes && dia < o.dia)))
  }
}
```

Mixins

Un **mixin** es un módulo/clase con métodos disponibles para otros módulos/clases *sin tener que usar la herencia*

- Los mixin son un mecanismo de **reutilización de código** sin herencia
- Es una **alternativa** a la herencia múltiple
- Incluye una **interfaz** con métodos ya implementados
- No se heredan sino que se **incluyen**
- Un mixin es una (sub)clase, luego define un comportamiento y un **estado**
- Es una forma de implementar la **inversión de dependencias**

¿Qué lenguajes tienen mixins?

Ruby modules

En Ruby los mixins se implementan mediante módulos (`module`).

- Un módulo no puede tener instancias (porque no es una clase)
- Un módulo puede incluirse (`include`) dentro de la definición de una clase

Comparadores: Implementación en Ruby

Una manera de implementar un `Comparable` en ruby mediante el **módulo** Comparable:

- La clase que incluye el módulo `Comparable` tiene que implementar:
  - el método `<=>`: es un método que incluye los siguientes operadores/métodos: `<, <=, ==, >, >=, between?`
  - el atributo-criterio de comparación
- En `x <=> y`, `x` es el receptor del mensaje/método e `y` es el argumento

```ruby
class Student
  include Comparable
  attr_accessor :name, :score

  def initialize(name, score)
    @name = name
    @score = score
  end

  def <=>(other)
    @score <=> other.score
  end
end

s1 = Student.new("Peter", 100)
s2 = Student.new("Jason", 90)
s3 = Student.new("Maria", 95)

s1 > s2 #true
s1 <= s2 #false
s3.between?(s1,s2) #true
```

Scala Traits

Un **trait** es una forma de separar las dos principales responsabilidades de una clase: definir el **estado** de sus instancias y definir su **comportamiento**.

- Las clases y los objetos en Scala pueden extender un `trait`
- Los `trait` de Scala son similares a las `interface` de Java.
- Los `trait` no pueden instanciarse
- Los métodos definidos en una clase tienen precedencia sobre los de un `trait`
- Los `trait` no tienen estado propio, sino el del objeto o la instancia de la clase a la que se aplica

Ejemplo 2: Un iterador con Scala traits

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}

class IntIterator(to: Int) extends Iterator[Int] {
  private var current = 0
  override def hasNext: Boolean = current < to
  override def next(): Int =  {
    if (hasNext) {
      val t = current
      current += 1
      t
    } else 0
  }
}

val iterator = new IntIterator(10)
println(iterator.next())  // prints 0
println(iterator.next())  // prints 1
```

¿Un `trait` de Scala es un *mixin*?

Ejemplo: mezcla de traits con comportamiento

```
trait Fighter {
    def fight(): String   //abstract
}

trait Flyer {
    def startFlying(): Unit = println("start flying")
    def stopFlying(): Unit = println("stop flying")
}

trait Swimmer {
    def startSwimming(): Unit = println("start swimming")
    def stopSwimming(): Unit = println("stop swimming")
}

class Hero(name: String) extends Fighter with Flyer {
    def fight(): String = "thump!"
}

class AmphibiousHero extends Fighter with Flyer with Swimmer {
    def fight(): String = "splash!"
}
```

```
object Test {
  def main(args: Array[String]): Unit = {
    val superman = new Hero("Superman")
    val aquawoman = new AmphibiousHero

    println( superman.fight() )
    superman.startFlying()
    println( aquawoman.fight() )
    aquawoman.startSwimming()

    val aquaman = new Hero("Aquaman") with Swimmer
    aquaman.startSwimming()
  }
}
```

Scala traits como mixins

Los traits de Scala tienen una interfaz que las clases heredan (`extends`)

Entonces... una clase que extiende un trait con un comportamiento, ¿va contra el principio general de que la herencia de comportamiento es una mala idea?

- Odersky llama **mixin traits** a los traits con comportamiento
- Para ser un mixin genuino, un trait debería mezclar comportamiento y no interfaces heredadas

Lectura recomendada: Scala Mixins: The right way

¿Y en Java no hay *traits*?

Java default methods

- Desde Java 8, las interfaces pueden incorporar métodos por defecto que hacen que las interfaces de Java se comporten más como un trait.
- Sirven para implementar herencia múltiple

Ejemplo de métodos `default`

Resolver la ambigüedad en la herencia múltiple con métodos `default`

```
interface Volador {
  default void mover() {
    System.out.println("Moviendo por aire");
  }
}

interface Nadador {
  default void mover() {
    System.out.println("Moviendo por agua");
  }
}
```

```
class Pato implements Volador, Nadador {
  @Override
  public void mover() {
    // Obligatorio resolver el conflicto
    Volador.super.mover(); // o Nadador.super.mover()
    System.out.println("... como un pato");
  }
}

// Válido desde JDK 25...
void main() {
  var pato = new Pato();
  pato.mover();
}
```

¿Qué ventajas tienen las implementaciones basadas en **Composición** frente a las basadas en **Herencia** (estática)?

La respuesta está en la **inyección de dependencias**...