


Programación Orientada a Objetos

# Interfaces y composición

## Composición frente a herencia · Sustitución de Liskov

**Gerson Garrido**

Semana 7

UNIVERSIDAD NACIONAL DE INGENIERÍA

# Hoy

1. Interfaces: un contrato sin estado; `implements`; la interfaz como tipo
2. Métodos `default` y `static`; `Comparable<T>`
3. ¿Interfaz o clase abstracta?
4. Composición frente a herencia: tener en lugar de ser; la superclase frágil
5. El principio de sustitución de Liskov
6. Test doubles: stub y spy

> **Lectura**
>
> Eck, *Introduction to Programming Using Java* (v. 9): sec. 5.7. Liskov y Wing (1994), introducción y sec. 2. Fowler: *Replace Superclass with Delegate* y *Test Double*.

# Donde quedamos

**Semana 6**

- Tipo estático: qué se puede llamar. Tipo dinámico: qué versión se ejecuta.
- `Employee` abstracta: fija el *qué*; cada subclase da el *cómo*.

**Dos preguntas para hoy**

- ¿Y si las clases que se sustituyen no son parientes?
- ¿Cuándo no conviene heredar?

SECCIÓN 01

# Interfaces

01



# Una factura y un empleado

- A fin de mes, la Bodega paga facturas de proveedores y sueldos.
- Queremos un método que sume todo lo que se debe.
- ¿Una superclase común `ThingToPay`? Forzado: `Employee` ya tiene su jerarquía, y una factura no *es un* empleado.
- Lo único que comparten es una capacidad: se pueden pagar.

> **Interfaz *(interface)***
>
> Un conjunto de métodos que una clase promete ofrecer, sin decir cómo y sin guardar estado.

# Una interfaz

Payable.java

```java
public interface Payable {

    // en céntimos
    long amountDue();
}
```

bash

```
$ javap -cp build Payable
public interface Payable {
  public abstract long amountDue();
  ...
}
```

- Sin cuerpo: método abstracto.
- No hace falta escribir `public` ni `abstract`: en una interfaz lo son.

# implements

Invoice.java

```java
public class Invoice
        implements Payable {
    private final String number;
    private final long amount;
    // constructor: valida amount > 0

    @Override
    public long amountDue() {
        return amount;
    }
}
```

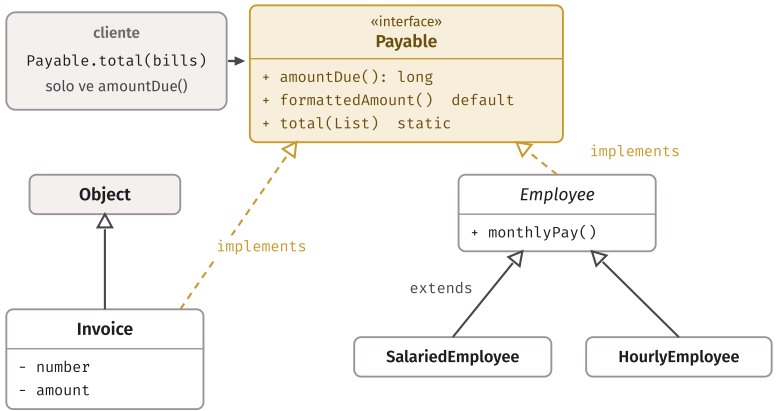
Employee.java · semana 6

```java
public abstract class Employee
        implements Payable {
    // name, monthlyPay()

    @Override
    public long amountDue() {
        return monthlyPay();
    }
}
```

`SalariedEmployee` y `HourlyEmployee` no cambian: heredan `amountDue()`.

# Dos jerarquías, una interfaz




Invoice y Employee no tienen ninguna superclase en común, salvo Object

# Error común: olvidar `public`

Receipt.java

```java
public class Receipt
        implements Payable {
    long amountDue() {
        return 100;
    }
}
```

bash

```
Receipt.java:2: error: amountDue() in
    Receipt
  cannot implement amountDue() in Payable
  attempting to assign weaker access
  privileges; was public
```

**Quien tiene un `Payable` tiene derecho a llamar a `amountDue()`**

Por eso no se puede ofrecer con menos acceso. Y si falta el método: `Gift is not abstract and does not override...`

# Una superclase, varias interfaces

- `class Employee extends Person implements Payable, Comparable<Employee>`
- ¿Por qué no `extends A, B`? Si los dos tienen un campo `id` y un método `describe()`, ¿cuántos `id`? ¿Qué `describe()`? El problema del diamante.
- Java lo evita: el estado se hereda de una sola superclase. Una interfaz no tiene campos de instancia.

**Idea clave**

La herencia de clases dice qué es un objeto: una sola. Una interfaz dice qué sabe hacer: varias.

Por eso se llaman como capacidades: `Payable`, `Comparable`, `Runnable`, `Iterable`.

# La interfaz como tipo

PayableDemo.java

```java
List<Payable> bills = new ArrayList<>();
bills.add(new Invoice("F001-120", 45_000));
bills.add(new SalariedEmployee("alice", 320_000));
bills.add(new HourlyEmployee("bob", 1_250, 80));
for (Payable p : bills) IO.println(p + " -> " + p.formattedAmount());
IO.println("Total: " + Payable.total(bills) + " céntimos");
```

bash · java -cp build PayableDemo

```
Invoice[F001-120] -> S/ 450.00
SalariedEmployee[alice] -> S/ 3200.00
HourlyEmployee[bob] -> S/ 1000.00
Total: 465000 céntimos
```

# El mismo dynamic dispatch

bash · salida recortada

```
$ javap -c -cp build Payable
...
       1: invokeinterface #1,  1            // InterfaceMethod amountDue:()J
```

- `new Payable()` no compila; como tipo, sí. Y `p.monthlyPay()` tampoco: el tipo estático es `Payable`.
- La semana pasada: `invokevirtual` para un método de una clase.
- Hoy: `invokeinterface` para un método de una interfaz.
- La regla es la misma: la JVM busca el método empezando por la clase real del objeto.

# Métodos `default` y `static`

Payable.java

```java
public interface Payable {
    long amountDue();
    default String formattedAmount() {
        long cents = amountDue();
        return String.format("S/ %d.%02d",
                cents / 100, cents % 100);
    }
    static long total(List<Payable> items) {
        long sum = 0;
        for (Payable p : items)
            sum += p.amountDue();
        return sum;
    }
}
```

- `default`: método con cuerpo que las clases heredan. Escrito con el método abstracto.
- El Template Method de la semana 6, ahora en una interfaz.
- `static`: de la interfaz, no del objeto. `Payable.total(bills)`, como `List.of(...)`.

# ¿Para qué `default`?

- Hasta Java 7, agregar un método a una interfaz publicada rompía todas las clases que la implementaban.
- Con `default` (Java 8), las clases existentes reciben el método ya hecho.
- Constantes: todo campo de una interfaz es `public static final`. Úsenlo poco; para constantes, una clase o un `enum`.

**Dos interfaces con el mismo `default`**

El compilador no elige: la clase tiene que hacer override del método, y puede llamar a `Payable.super.formattedAmount()`. Y un `default` no puede guardar estado: si lo necesita, hace falta una clase.

# Una interfaz de la API: `Comparable<T>`

Song.java

```java
public record Song(String title, String artist,
        int seconds) implements Comparable<Song>
    {
    @Override
    public int compareTo(Song other) {
        return Integer.compare(seconds,
                other.seconds);
    }
}
```

En `PlaylistDemo`: `Collections.sort(sorted)` las ordena por duración.

- `a.compareTo(b)`: negativo si `a` va antes, 0 si empatan, positivo si va después.
- `Collections.sort` no sabe de canciones: solo llama a `compareTo`.
- No `seconds - other.seconds`: con valores grandes hace overflow.
- Orden y comparadores: semana 11.

# ¿Interfaz o clase abstracta?

| | **Interfaz** | **Clase abstracta** |
|---|---|---|
| Campos de instancia (estado) | no | sí |
| Constructores | no | sí |
| Métodos con cuerpo | `default`, `static`, `private` | cualquiera |
| Cuántas puede tener una clase | varias | una |
| Qué expresa | una capacidad | un parentesco |

Capacidad que tienen clases distintas: **interfaz**. Estado y código entre parientes: **clase abstracta**.
Y se combinan: `Employee` es abstracta *e* implementa `Payable`.

# Error común: una interfaz por cada clase

- `ProductService` con una sola implementación, `ProductServiceImpl`, que nadie más va a escribir.
- Cada firma duplicada en dos archivos. No gana nada.

**Una interfaz se justifica si...**

- hay, o habrá con certeza, más de una implementación, o
- una prueba necesita reemplazar la real por un test double (al final de hoy).

SECCIÓN 02

# Composición frente a herencia

02



# Tener en lugar de ser

Playlist.java

```java
public class Playlist {
    private final int maxSeconds;
    private final ArrayList<Song> songs = ...;
    private int totalSeconds = 0;
    public boolean add(Song song) {
        if (totalSeconds + song.seconds()
                > maxSeconds)
            return false;
        songs.add(song);   // delegación
        totalSeconds += song.seconds();
        return true;
    }   // size(): return songs.size();
}
```

**Composición**
***(composition)***

Un objeto tiene a otro como campo y le delega *(delegation)* trabajo.

- Invariante: el total nunca pasa de `maxSeconds`.
- `songs()` devuelve una copia.

# La tentación: `extends ArrayList<Song>`

InheritedPlaylist.java

```java
@SuppressWarnings("serial")
public class InheritedPlaylist
        extends ArrayList<Song> {
    private final int maxSeconds;
    @Override
    public boolean add(Song song) {
        if (totalSeconds() + song.seconds()
                > maxSeconds)
            return false;
        return super.add(song);
    }   // constructor y totalSeconds()
}
```

- "Una lista de reproducción *es* una lista." Menos código.
- Sin la anotación, `-Xlint:all` avisa: `[serial] serializable class ... has no definition of serialVersionUID`.
- Primera pista: heredamos más de lo que queríamos.

# Las otras puertas

PlaylistDemo.java · máximo de 600 s

```java
bad.add(a); bad.add(b); bad.add(c);   // la tercera no cabe
bad.add(0, c);                        // heredado: no pasa por el add que valida
bad.addAll(List.of(c, c));            // heredado
```

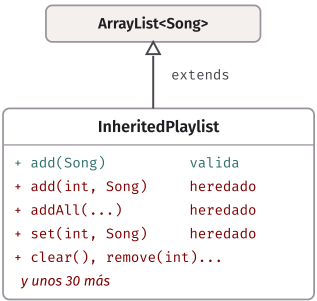
bash

```
$ java -cp build PlaylistDemo
composición: true true false -> 395 s
herencia:    true true false -> 395 s
después de add(0, c) y addAll: 1175 s de 600
```

- `ArrayList` tiene 38 nombres de métodos públicos (sin contar los de `Object`).
- La subclase los hereda todos, y casi ninguno pasa por nuestro `add(Song)`.

# Quién decide la interfaz pública

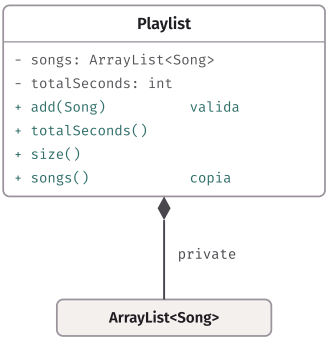
HERENCIA · ES UN ARRAYLIST




cualquiera de ellos rompe el máximo

la interfaz pública la decide ArrayList

COMPOSICIÓN · TIENE UN ARRAYLIST




add y size delegan en la lista

la interfaz pública la decide Playlist

# La superclase frágil

CountingSet.java

```java
public class CountingSet
        extends HashSet<String> {
    private int added = 0;
    @Override public boolean add(String s) {
        added++;
        return super.add(s);
    }
    @Override public boolean addAll(
            Collection<? extends String> c) {
        added += c.size();
        return super.addAll(c);
    }
}
```

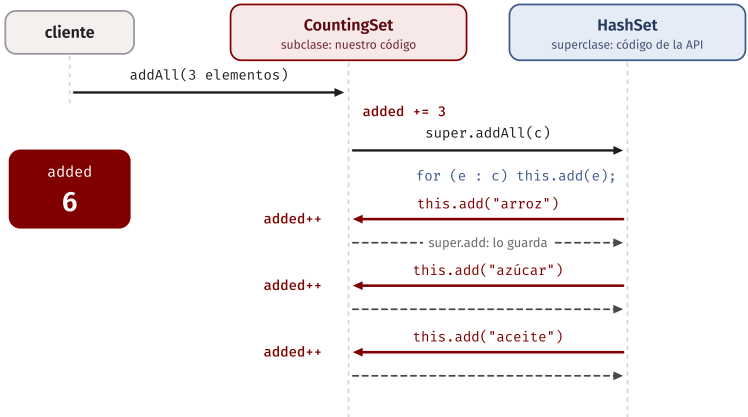
Cuenta cuántos elementos se intentó agregar. Cada método cuenta lo suyo y delega en `super`.

java FragileDemo

```
herencia:    added() = 6
composición: added() = 3
```

¿Seis, con tres elementos?

# ¿De dónde salen los otros tres?




**added() devuelve 6, pero se agregaron 3 elementos**

# La superclase frágil

- `HashSet` hereda `addAll` de `AbstractCollection`, que llama a `this.add(e)` por cada elemento.
- `this` es un `CountingSet`: el dynamic dispatch lleva cada llamada de vuelta a nuestro `add`.
- La documentación no promete que `addAll` use `add`: es un detalle de implementación.
- Si la biblioteca cambia, la subclase se rompe (o se arregla) sin que nadie toque su código.

**Superclase frágil** *(fragile base class)*

La subclase depende de *cómo* hace las cosas la superclase, no solo de *qué* hace.

# La misma cuenta, por composición

DelegatingCountingSet.java

```java
public class DelegatingCountingSet {
    private final Set<String> items
            = new HashSet<>();
    private int added = 0;
    public boolean add(String s) {
        added++;
        return items.add(s);
    }
    public boolean addAll(Collection<String> c) {
        added += c.size();
        return items.addAll(c);
    }   // contains, size, added
}
```

- `items.addAll` llama al `add` de `HashSet`: `items` es un `HashSet`.
- Cuenta 3, aunque `HashSet` cambie por dentro.
- Bloch, *Effective Java*, ítem 18: *favor composition over inheritance*.

# Composición: idea clave

**Con herencia**

La subclase es también todo lo que es la superclase: sus métodos públicos y sus detalles internos.

**Con composición**

La clase elige qué ofrecer; el objeto de adentro queda encapsulado. El precio: unas líneas de delegación.

Heredar de una clase diseñada y documentada para eso está bien. Heredar de una clase concreta cualquiera para reutilizar su código es frágil. Por eso conviene marcar `final` las clases que no se pensaron para extenderse.

SECCIÓN 03

# Sustitución de Liskov

03

# "Es un", también en el comportamiento

- La pregunta no es si "B es un A" en el diccionario.
- Es: ¿puedo usar un B en cualquier lugar donde el programa espera un A, sin que nada se rompa?
- Barbara Liskov la planteó en 1987; con Jeannette Wing la formalizó en 1994.

**Principio de sustitución de Liskov** ***(LSP)***

Todo lo que vale para los objetos del supertipo tiene que valer para los del subtipo. Un cliente escrito para el supertipo funciona, sin cambios, con cualquier subtipo.



Barbara Liskov, 2010. Kenneth C. Zirkel, CC BY-SA 3.0, Wikimedia Commons.

# Tres preguntas por cada override

1. **Precondiciones: no se refuerzan.** El subtipo no exige más que el supertipo.
2. **Postcondiciones: no se debilitan.** El subtipo garantiza al menos lo mismo.
3. **Invariantes: se mantienen.**

**Un caso que ya escribimos**

La `SavingsAccount` de la semana 5 limita a 3 retiros por mes. Un cliente de `BankAccount` no sabe de ese límite: la subclase refuerza la precondición de `withdraw`.

# El cuadrado y el rectángulo

Rectangle.java

```java
public class Rectangle {
    private int width, height;
    // Post: el alto no cambia.
    public void setWidth(int w) {
        this.width = w;
    }
    // setHeight: igual, al revés
    public int area() {
        return width * height;
    }
}
```

Square.java

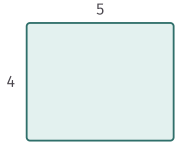
```java
public class Square
        extends Rectangle {
    public Square(int side) {
        super(side, side);
    }
    @Override
    public void setWidth(int w) {
        super.setWidth(w);
        super.setHeight(w);
    }   // setHeight: igual
}
```

En geometría, un cuadrado es un rectángulo. Cada clase, sola, es correcta.

# Un cliente de `Rectangle`

```
static void resize(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    // espera: r.area() == 20
}
```

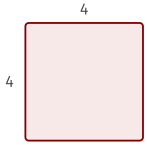
new Rectangle(2, 3)




width 5, height 4

área 20: lo esperado

new Square(2)




width 4, height 4

área 16: se esperaba 20

# La prueba que lo detecta

RectangleTest.java

```java
static int areaAfterResize(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    return r.area();
}
@Test void rectangleKeepsSidesIndependent() {
    Rectangle r = new Rectangle(2, 3);
    assertEquals(20, areaAfterResize(r));
}
@Test void squareBreaksTheExpectation() {
    Rectangle s = new Square(2);  // documenta
    assertNotEquals(20, areaAfterResize(s));
}
```

- Las pruebas del supertipo, para cualquier `Rectangle`, con objetos de cada subtipo.
- `Square.setHeight` debilitó la postcondición: el ancho cambió.
- Arreglo: figuras inmutables, o sin herencia y con una interfaz `Shape`.

# En la biblioteca estándar: `java.util.Stack`

StackDemo.java

```java
Stack<String> pages = new Stack<>();
pages.push("inicio");
pages.push("catálogo");
pages.push("carrito");
pages.add(0, "intrusa");      // heredado
String middle = pages.get(2); // heredado
pages.remove(1);              // heredado
```

bash

```
stack: [inicio, catálogo, carrito]
después: [intrusa, catálogo, carrito]
```

- `Stack extends Vector` (Java 1.0): hereda todo lo de una lista.
- `push`, `pop` y `peek` respetan LIFO, pero un cliente puede insertar, leer o borrar en cualquier posición.
- La documentación recomienda `Deque` y `ArrayDeque`.
- No se puede corregir: millones de líneas la usan. Un error de herencia en una clase pública queda para siempre.

# `ColorPoint` otra vez

ColorPoint.java

```java
public final class ColorPoint {
    private final Point point;
    private final String color;
    public Point asPoint() {
        return point;
    }
    @Override
    public boolean equals(Object o) {
        return o instanceof ColorPoint cp
                && point.equals(cp.point)
                && color.equals(cp.color);
    }
    // constructor, hashCode, toString
}
```

Semana 5: con herencia, `equals` no era simétrico, o el punto con color dejaba de ser sustituible.

bash

```
p.equals(red)   = false
red.equals(p)   = false
p.equals(red.asPoint()) =
    true
```

- Con composición, la simetría vuelve.
- Quien necesita un `Point` lo pide: `asPoint()`.

# Regla práctica

**Herencia solo si "es un" se cumple en el comportamiento**

Cada método del supertipo, con sus precondiciones, postcondiciones e invariantes, sigue valiendo en el subtipo.

**Si no: composición + una interfaz común**

Fowler, *Replace Superclass with Delegate*: la subclase pasa a tener un campo del tipo de su antigua superclase.

**Error común: decidir por la frase del diccionario**

Un pingüino es un ave, pero si `Bird` tiene `fly()`, `Penguin` no es un buen subtipo.

SECCIÓN 04

# Test doubles

04



# Una dependencia difícil de controlar

StockMonitor.java

```java
public class StockMonitor {
    private final StockLevels levels;
    private final Notifier notifier;
    private final int threshold;
    // constructor: recibe las tres cosas
    public void check(String product) {
        int units = levels.unitsOf(product);
        if (units < threshold)
            notifier.send("Stock bajo: "
                + product + " (" + units + ")");
    }
}
```

- Con correo e inventario reales, cada prueba mandaría correos y armaría un inventario.
- `StockLevels` y `Notifier` son interfaces y llegan por el constructor.
- En la prueba: test doubles.

# Stub y spy, escritos a mano

FixedStockLevels.java · stub

```java
public class FixedStockLevels
        implements StockLevels {
    private final int units;
    // constructor
    @Override
    public int unitsOf(String p) {
        return units;
    }
}
```

**Stub**: responde algo fijo.

SpyNotifier.java · spy

```java
public class SpyNotifier
        implements Notifier {
    private final List<String> sent
            = new ArrayList<>();
    @Override
    public void send(String msg) {
        sent.add(msg);
    }
    // sent(): copia de la lista
}
```

**Spy**: anota lo que recibió.

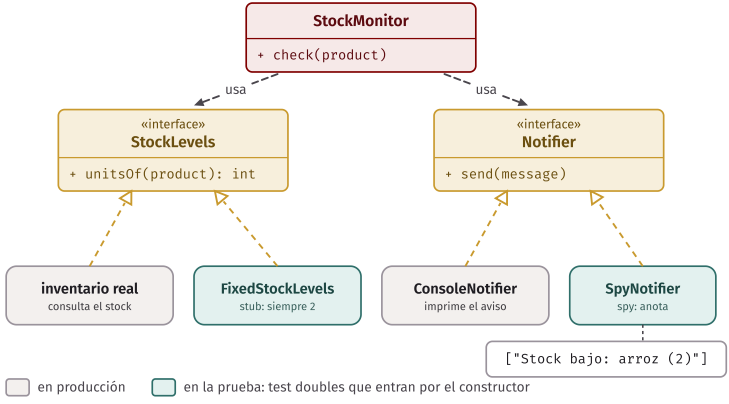
# La prueba

StockMonitorTest.java

```java
@Test
void notifiesWhenStockIsBelowThreshold() {
    SpyNotifier spy = new SpyNotifier();
    StockMonitor monitor = new StockMonitor(
            new FixedStockLevels(2), spy, 5);
    monitor.check("arroz");
    assertEquals(
        List.of("Stock bajo: arroz (2)"),
        spy.sent());
}
```

- La prueba arma exactamente la situación que quiere.
- Otra prueba, con `FixedStockLevels(5)`: sin aviso. Revisa el borde.
- Sin bibliotecas de mocks: un test double es solo otra implementación de la interfaz.

# En producción y en la prueba




El código de `StockMonitor` es el mismo: solo conoce las interfaces.

SECCIÓN 05

# Laboratorio

05

# Laboratorio: interfaces y test doubles

1. `Greeter` saluda según `LocalTime.now()`: hoy no se puede probar el saludo de las 21 h. Extraer una interfaz para "la hora actual".
2. Escribir a mano un stub (hora fija) y un spy (cuenta las consultas).
3. Probar `Greeter` con los test doubles, incluidos los bordes (11:59, 12:00…).
4. `UndoHistory extends Stack`: pasarla a composición, con `UndoHistoryTest` como red de seguridad.

**En `code/week-07`**

Los ejemplos de hoy y el código inicial: `Greeter`, `UndoHistory` y sus pruebas. `make test`: 20 pruebas.

**P2 en curso**

Las reglas de precio y descuento del catálogo, como interfaces. Entrega en la semana 9.

# Resumen

- Una **interfaz** es un contrato sin estado; una clase extiende una sola clase pero implementa varias interfaces.
- La interfaz es un **tipo**; el dynamic dispatch funciona igual (`invokeinterface`). `default` agrega comportamiento sin romper a nadie; `static`, utilidades.
- Interfaz: una **capacidad**. Clase abstracta: estado y código entre **parientes**.
- **Composición**: la clase decide su interfaz pública y protege su invariante. La herencia expone todo y depende de detalles internos: **superclase frágil**.
- **Liskov**: precondiciones que no se refuerzan, postcondiciones que no se debilitan, invariantes que se mantienen. `Square`/`Rectangle` y `Stack` lo violan.
- Dependencias como interfaces por el constructor: se prueban con **stubs** y **spies**.

# Para la próxima clase

- Semana 8: tipos `sealed` y pattern matching, y el examen parcial (semanas 1 a 8).
- Lecturas: Eck, subsec. 3.6.5 (*switch expressions*) y *Java Language Updates*: *Sealed Classes* y *Pattern Matching*.
- Repasar las preguntas de repaso de las semanas 1 a 7.
- P2 en curso: entrega en la semana 9.

**Pregunta para pensar**

Un override en cada subclase hace fácil agregar un *tipo* nuevo. ¿Qué pasa si lo que quieren agregar es una *operación* nueva sobre todos los tipos?