

Programación Orientada a Objetos

Polimorfismo y clases abstractas

Dynamic dispatch, casting, métodos abstractos

Gerson Garrido

Semana 6

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Sustituir: una variable de la superclase, objetos de muchas subclases
2. Tipo estático y tipo dinámico
3. Dynamic dispatch: qué decide `javac` y qué decide la JVM
4. Overloading frente a override; casting e `instanceof`
5. Lo que no es polimórfico, y una trampa en los constructores
6. Clases y métodos abstractos; el patrón Template Method

> **Lectura**
>
> Eck, *Introduction to Programming Using Java* (v. 9): subsecs. 5.5.3, 5.5.4 y 5.5.5. JLS 25, §15.12.4 (la regla exacta).

SECCIÓN 01

Polimorfismo

01

Donde quedamos

semana 5

```
1  BankAccount a = new SavingsAccount("A-001",
       2);
2  a.withdraw(1000);
```

- `SavingsAccount` hace override de `withdraw`.

Donde quedamos

semana 5

```
1  BankAccount a = new SavingsAccount("A-001",
       2);
2  a.withdraw(1000);
```

- `SavingsAccount` hace override de `withdraw`.
- ¿Cuál se ejecuta: el de la variable o el del objeto?

Donde quedamos

semana 5

```
1  BankAccount a = new SavingsAccount("A-001",
       2);
2  a.withdraw(1000);
```

- `SavingsAccount` hace override de `withdraw`.
- ¿Cuál se ejecuta: el de la variable o el del objeto?
- ¿Quién lo decide: el compilador o la JVM?

Donde quedamos

semana 5

```
1  BankAccount a = new SavingsAccount("A-001",
       2);
2  a.withdraw(1000);
```

- `SavingsAccount` hace override de `withdraw`.
- ¿Cuál se ejecuta: el de la variable o el del objeto?
- ¿Quién lo decide: el compilador o la JVM?

Polimorfismo ***(polymorphism)***

El mismo mensaje, enviado a objetos de clases distintas, hace en cada uno lo que corresponde. Y el código que lo envía no necesita saber de qué clase es cada objeto.

Una jerarquía de avisos

Notification.java

```java
public class Notification {
    private final String recipient;
    private final String text;
    // constructor, recipient(), text()

    public String send() {
        return "[aviso] " + recipient
                + ": " + text;
    }
    public long cost() {   // en céntimos
        return 1;
    }
}
```

- La bodega avisa por correo, por SMS y en la pantalla de la aplicación.

Una jerarquía de avisos

Notification.java

```java
public class Notification {
    private final String recipient;
    private final String text;
    // constructor, recipient(), text()

    public String send() {
        return "[aviso] " + recipient
                + ": " + text;
    }
    public long cost() {   // en céntimos
        return 1;
    }
}
```

- La bodega avisa por correo, por SMS y en la pantalla de la aplicación.
- Todo aviso tiene destinatario y texto, y se puede `send()`.

Una jerarquía de avisos

Notification.java

```java
public class Notification {
    private final String recipient;
    private final String text;
    // constructor, recipient(), text()

    public String send() {
        return "[aviso] " + recipient
                + ": " + text;
    }
    public long cost() {   // en céntimos
        return 1;
    }
}
```

- La bodega avisa por correo, por SMS y en la pantalla de la aplicación.
- Todo aviso tiene destinatario y texto, y se puede `send()`.
- `send` devuelve el texto en lugar de imprimirlo: así se puede probar.

Tres subclases

SmsNotification.java

```java
public class SmsNotification
        extends Notification {
    // constructor: super(phone, text)
    public String phoneNumber() {  // solo aquí
        return recipient();
    }
    @Override
    public String send() {
        return "[SMS] al " + phoneNumber()
                + ": " + text();
    }
    @Override
    public long cost() { return 5; }
}
```

- `EmailNotification`: `send()` y `cost()` propios (costo 0).
- `SmsNotification`: los suyos, y además `phoneNumber()`.

Tres subclases

SmsNotification.java

```java
public class SmsNotification
        extends Notification {
    // constructor: super(phone, text)
    public String phoneNumber() {  // solo aquí
        return recipient();
    }
    @Override
    public String send() {
        return "[SMS] al " + phoneNumber()
                + ": " + text();
    }
    @Override
    public long cost() { return 5; }
}
```

- `EmailNotification`: `send()` y `cost()` propios (costo 0).
- `SmsNotification`: los suyos, y además `phoneNumber()`.
- `PushNotification`: no hace override de nada. Usa lo que hereda.

Sustituir: upcasting

upcasting

```
1  Notification n = new SmsNotification("987654321", "Código 4821");
```

- Un `SmsNotification` es un `Notification`: se guarda en una variable, se pasa como parámetro o se mete en un `ArrayList<Notification>`.

Sustituir: upcasting

upcasting

```
1  Notification n = new SmsNotification("987654321", "Código 4821");
```

- Un `SmsNotification` es un `Notification`: se guarda en una variable, se pasa como parámetro o se mete en un `ArrayList<Notification>`.
- De subclase a superclase: **upcasting**. Implícito y siempre seguro: todo lo que se le pide a un `Notification` se le puede pedir a un SMS.

Sustituir: upcasting

upcasting

```
1  Notification n = new SmsNotification("987654321", "Código 4821");
```

- Un `SmsNotification` es un `Notification`: se guarda en una variable, se pasa como parámetro o se mete en un `ArrayList<Notification>`.
- De subclase a superclase: **upcasting**. Implícito y siempre seguro: todo lo que se le pide a un `Notification` se le puede pedir a un SMS.
- **Sustitución**: donde el programa espera un objeto de la superclase, sirve uno de cualquier subclase.

Dos tipos en una misma línea

Tipo estático

El de la variable: `Notification`. Está escrito en el código y no cambia.

Lo usa el **compilador** para decidir qué se puede llamar.

Dos tipos en una misma línea

Tipo estático

El de la variable: `Notification`. Está escrito en el código y no cambia.

Lo usa el **compilador** para decidir qué se puede llamar.

Tipo dinámico

La clase del objeto: `SmsNotification`. Se fija en el `new`; la variable puede apuntar luego a otro.

Lo usa la **JVM** para decidir qué código se ejecuta.

El compilador solo mira el tipo estático

Wrong.java

```java
Notification n = new SmsNotification("987654321", "Código 4821");
IO.println(n.phoneNumber());
```

bash

```
Wrong.java:4: error: cannot find symbol
        IO.println(n.phoneNumber());
                   ^
  symbol:   method phoneNumber()
  location: variable n of type Notification
```

El compilador solo mira el tipo estático

Wrong.java

```java
Notification n = new SmsNotification("987654321", "Código 4821");
IO.println(n.phoneNumber());
```

bash

```
Wrong.java:4: error: cannot find symbol
        IO.println(n.phoneNumber());
                   ^
  symbol:   method phoneNumber()
  location: variable n of type Notification
```

El objeto *sí* tiene `phoneNumber()`. Pero el compilador solo sabe que `n` apunta a algún `Notification`, y lo busca ahí.

Una línea, tres métodos distintos

NotificationDemo.java

```java
ArrayList<Notification> inbox = new ArrayList<>();
inbox.add(new EmailNotification("alice@example.com", "Tu pedido salió"));
inbox.add(new SmsNotification("987654321", "Código 4821"));
inbox.add(new PushNotification("bob", "Stock bajo: arroz"));
for (Notification n : inbox) {
    IO.println(n.send());
}
```

bash

```
[correo] para alice@example.com: Tu pedido salió
[SMS] al 987654321: Código 4821
[aviso] bob: Stock bajo: arroz
```

Qué deja `javac` en el bytecode

javap -c -cp build NotificationDemo · recortado

```
  86: aload          5
  88: invokevirtual #53     // Method Notification.send:()Ljava/lang/String;
```

- **Dynamic dispatch**: el método que se ejecuta se elige al ejecutar, según la clase del objeto.

Qué deja `javac` en el bytecode

javap -c -cp build NotificationDemo · recortado

```
86: aload          5
88: invokevirtual #53     // Method Notification.send:()Ljava/lang/String;
```

- **Dynamic dispatch**: el método que se ejecuta se elige al ejecutar, según la clase del objeto.
- `invokevirtual Notification.send`: el compilador solo conoce el tipo estático. Queda fija la firma: `send`, sin parámetros, devuelve `String`.

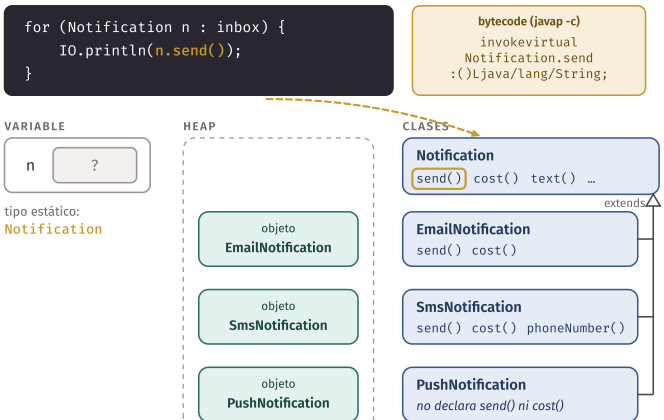
Qué deja `javac` en el bytecode

```
javap -c -cp build NotificationDemo · recortado

  86: aload          5
  88: invokevirtual #53     // Method Notification.send:()Ljava/lang/String;
```

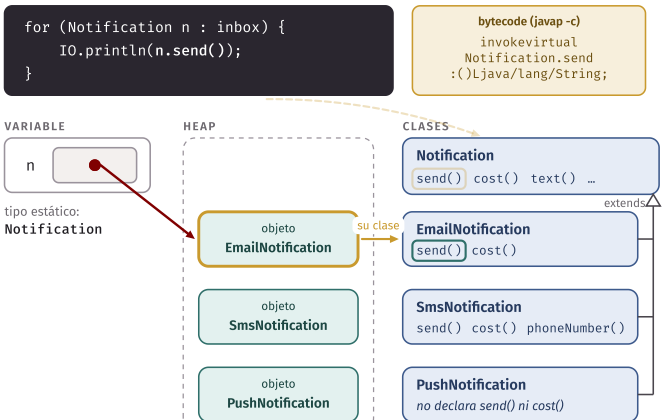
- **Dynamic dispatch**: el método que se ejecuta se elige al ejecutar, según la clase del objeto.
- `invokevirtual Notification.send`: el compilador solo conoce el tipo estático. Queda fija la firma: `send`, sin parámetros, devuelve `String`.
- No aparece `EmailNotification` ni `SmsNotification`: eso lo resuelve la JVM.

Dynamic dispatch, paso a paso



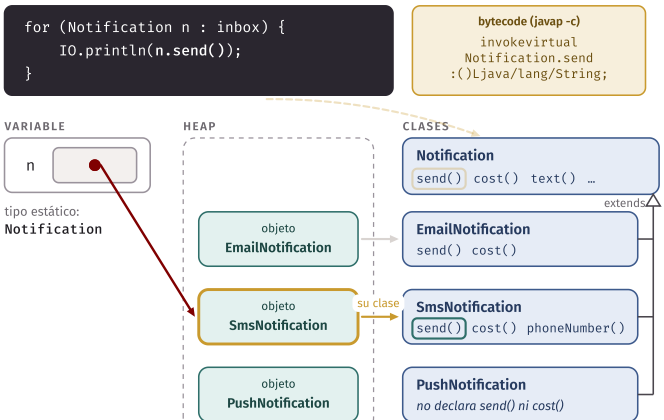
1. Al compilar solo cuenta el tipo estático: Notification tiene send(), así que compila.
javac deja una llamada a Notification.send; todavía no se sabe qué objeto habrá.

Dynamic dispatch, paso a paso



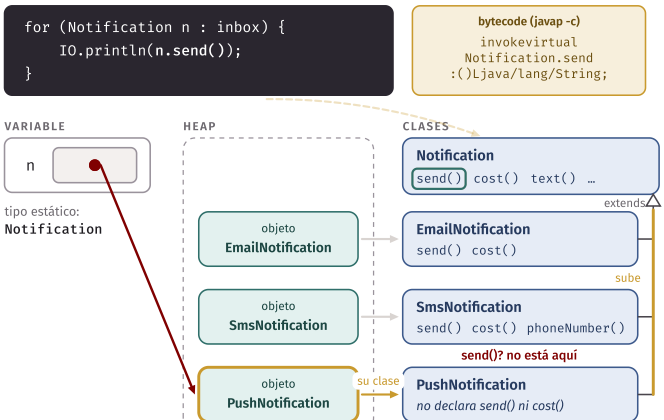
2. Al ejecutar, la JVM sigue la referencia hasta el objeto y mira su clase:
EmailNotification declara send(), y se ejecuta EmailNotification.send().

Dynamic dispatch, paso a paso



3. La misma instrucción, otro objeto: SmsNotification declara send(),
y se ejecuta SmsNotification.send().

Dynamic dispatch, paso a paso



4. PushNotification no declara send(): la búsqueda sube a la superclase y se ejecuta Notification.send(), la versión heredada.

La regla

Al ejecutar `invokevirtual`, la JVM:

La regla

Al ejecutar `invokevirtual`, la JVM:

1. sigue la referencia hasta el objeto;

La regla

Al ejecutar `invokevirtual`, la JVM:

1. sigue la referencia hasta el objeto;
2. mira la clase del objeto (todo objeto sabe de qué clase es);

La regla

Al ejecutar `invokevirtual`, la JVM:

1. sigue la referencia hasta el objeto;
2. mira la clase del objeto (todo objeto sabe de qué clase es);
3. busca la firma en esa clase; si no está, sube a la superclase, y así hasta encontrarla.

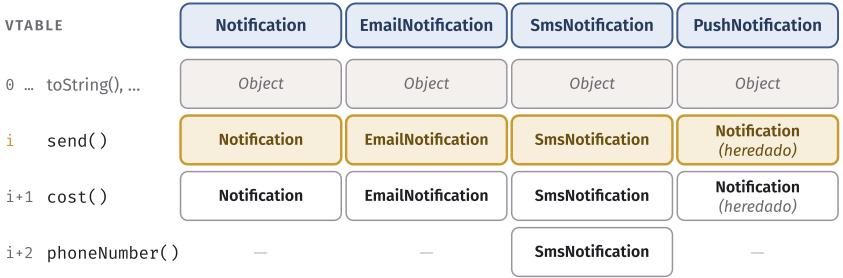
La regla

Al ejecutar `invokevirtual`, la JVM:

1. sigue la referencia hasta el objeto;
2. mira la clase del objeto (todo objeto sabe de qué clase es);
3. busca la firma en esa clase; si no está, sube a la superclase, y así hasta encontrarla.

> **JLS 25, §15.12.4.4 *Locate Method to Invoke***
>
> La regla del lenguaje es esa búsqueda hacia arriba. `PushNotification` no declara `send()`: se ejecuta el de `Notification`.

Cómo lo hace rápido HotSpot: la vtable



Es la forma en que HotSpot implementa la regla del lenguaje (buscar desde la clase del objeto hacia arriba).

invokevirtual Notification.send = "la fila de send() en la tabla de la clase del objeto":
una lectura y un salto, sin recorrer la jerarquía en cada llamada.

Cada clase copia la tabla de su superclase, reemplaza las entradas a las que hace override y agrega lo suyo al final.

Por qué importa

WhatsAppNotification.java · una clase nueva

```java
public class WhatsAppNotification extends Notification {
    // constructor
    @Override public String send() { return "[WhatsApp] ..."; }
    @Override public long cost() { return 2; }
}
```

Por qué importa

WhatsAppNotification.java · una clase nueva

```java
public class WhatsAppNotification extends Notification {
    // constructor
    @Override public String send() { return "[WhatsApp] ..."; }
    @Override public long cost() { return 2; }
}
```

- El `for` de `NotificationDemo` no cambia y no se recompila.

Por qué importa

WhatsAppNotification.java · una clase nueva

```java
public class WhatsAppNotification extends Notification {
    // constructor
    @Override public String send() { return "[WhatsApp] ..."; }
    @Override public long cost() { return 2; }
}
```

- El `for` de `NotificationDemo` no cambia y no se recompila.
- El `invokevirtual Notification.send` que ya está en el bytecode encuentra el método nuevo.

Por qué importa

WhatsAppNotification.java · una clase nueva

```java
public class WhatsAppNotification extends Notification {
    // constructor
    @Override public String send() { return "[WhatsApp] ..."; }
    @Override public long cost() { return 2; }
}
```

- El `for` de `NotificationDemo` no cambia y no se recompila.
- El `invokevirtual Notification.send` que ya está en el bytecode encuentra el método nuevo.
- Es lo que pide P2: agregar un tipo de producto sin tocar el código que vende.

SECCIÓN 02

Overloading y override

02



¿Cuándo se resuelve cada uno?

Overloading

Mismo nombre, distintos parámetros.

Se resuelve **al compilar**, con los tipos estáticos de los argumentos. La firma queda escrita en el bytecode.

¿Cuándo se resuelve cada uno?

Overloading

Mismo nombre, distintos parámetros.

Se resuelve **al compilar**, con los tipos estáticos de los argumentos. La firma queda escrita en el bytecode.

Override

Misma firma, en una subclase.

Se resuelve **al ejecutar**, con la clase del objeto que recibe el mensaje.

Traza: predigan la salida

Greeter.java · LoudGreeter.java

```java
class Greeter {
    String greet(Object o) { ... }
    String greet(String s) { ... }
}
class LoudGreeter extends Greeter {
    @Override
    String greet(Object o) { ... }
}
```

TraceDemo.java

```java
Greeter g = new LoudGreeter();
String s = "hola";
Object x = s;
IO.println("1. " + g.greet(s));
IO.println("2. " + g.greet(x));
IO.println("3. " + g.greet((String) x));
Greeter plain = new Greeter();
IO.println("4. " + plain.greet(x));
```

Cada `greet` devuelve su propio nombre.

Traza: la salida

bash

```
$ java -cp build TraceDemo
1. Greeter.greet(String)
2. LoudGreeter.greet(Object)
3. Greeter.greet(String)
4. Greeter.greet(Object)
```

Traza: la salida

```bash
$ java -cp build TraceDemo
1. Greeter.greet(String)
2. LoudGreeter.greet(Object)
3. Greeter.greet(String)
4. Greeter.greet(Object)
```

- 1: firma `(String)`; `LoudGreeter` no le hace override: se sube a `Greeter`.

Traza: la salida

```bash
$ java -cp build TraceDemo
1. Greeter.greet(String)
2. LoudGreeter.greet(Object)
3. Greeter.greet(String)
4. Greeter.greet(Object)
```

- 1: firma `(String)`; `LoudGreeter` no le hace override: se sube a `Greeter`.
- 2: `x` es `Object` para el compilador: firma `(Object)`; el objeto es `LoudGreeter`.

Traza: la salida

```bash
$ java -cp build TraceDemo
1. Greeter.greet(String)
2. LoudGreeter.greet(Object)
3. Greeter.greet(String)
4. Greeter.greet(Object)
```

- 1: firma `(String)`; `LoudGreeter` no le hace override: se sube a `Greeter`.
- 2: `x` es `Object` para el compilador: firma `(Object)`; el objeto es `LoudGreeter`.
- 3: el casting cambia el tipo estático: otra vez `(String)`.

Traza: la salida

```bash
$ java -cp build TraceDemo
1. Greeter.greet(String)
2. LoudGreeter.greet(Object)
3. Greeter.greet(String)
4. Greeter.greet(Object)
```

- 1: firma `(String)`; `LoudGreeter` no le hace override: se sube a `Greeter`.
- 2: `x` es `Object` para el compilador: firma `(Object)`; el objeto es `LoudGreeter`.
- 3: el casting cambia el tipo estático: otra vez `(String)`.
- El tipo dinámico del argumento no participa; solo el del receptor.

Dos decisiones, en dos momentos

```
Greeter g = new LoudGreeter();   Object x = "hola";
g.greet(x)
```

1 · AL COMPILAR (JAVAC)

tipos estáticos: g es Greeter, x es Object

candidatos en Greeter:

`greet(Object)` — elige la firma

~~`greet(String)`~~ — un Object no es necesariamente String

`invokevirtual Greeter.greet:(Object)`

2 · AL EJECUTAR (JVM)

La firma ya quedó fija: greet(Object). El overloading se resolvió al compilar.

Dos decisiones, en dos momentos

```
Greeter g = new LoudGreeter();   Object x = "hola";
g.greet(x)
```

1 · AL COMPILAR (JAVAC)

tipos estáticos: g es Greeter, x es Object

candidatos en Greeter:

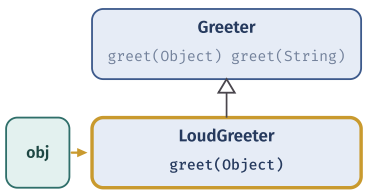
`greet(Object)` — elige la firma

~~`greet(String)`~~ — un Object no es necesariamente String

`invokevirtual Greeter.greet:(Object)`

2 · AL EJECUTAR (JVM)

el objeto de g es un LoudGreeter



Se ejecuta LoudGreeter.greet(Object): el override se resuelve al ejecutar.

Error común: `equals(Fraction)` otra vez

Fraction.java · mal

```
1  public boolean equals(Fraction f) {
2      return num == f.num && den == f.
       den;
3  }
```

- Es un overload de `equals(Object)`, no un override.

Error común: `equals(Fraction)` otra vez

Fraction.java · mal

```java
public boolean equals(Fraction f) {
    return num == f.num && den == f.
     den;
}
```

- Es un overload de `equals(Object)`, no un override.
- `ArrayList.contains` llama a `equals` con un argumento de tipo estático `Object`: su firma quedó fija en `equals(Object)`.

Error común: `equals(Fraction)` otra vez

Fraction.java · mal

```java
public boolean equals(Fraction f) {
    return num == f.num && den == f.
     den;
}
```

- Es un overload de `equals(Object)`, no un override.
- `ArrayList.contains` llama a `equals` con un argumento de tipo estático `Object`: su firma quedó fija en `equals(Object)`.
- El método de `Fraction` nunca se ejecuta.

Error común: `equals(Fraction)` otra vez

Fraction.java · mal

```
1  public boolean equals(Fraction f) {
2      return num == f.num && den == f.
        den;
3  }
```

- Es un overload de `equals(Object)`, no un override.
- `ArrayList.contains` llama a `equals` con un argumento de tipo estático `Object`: su firma quedó fija en `equals(Object)`.
- El método de `Fraction` nunca se ejecuta.

`@Override` siempre

Con `@Override`, ese error aparece al compilar.

SECCIÓN 03

Casting e `instanceof`

03



Downcasting

CastDemo.java

```java
Notification n = new SmsNotification("987654321", "Código 4821");
if (n instanceof SmsNotification sms) {
    IO.println("Es un SMS para " + sms.phoneNumber());
}
Notification other = new EmailNotification("alice@example.com", "Hola");
SmsNotification wrong = (SmsNotification) other;
```

bash · salida recortada

```
Es un SMS para 987654321
ClassCastException: class EmailNotification cannot be cast to class
    SmsNotification ...
```

Downcasting

CastDemo.java

```java
Notification n = new SmsNotification("987654321", "Código 4821");
if (n instanceof SmsNotification sms) {
    IO.println("Es un SMS para " + sms.phoneNumber());
}
Notification other = new EmailNotification("alice@example.com", "Hola");
SmsNotification wrong = (SmsNotification) other;
```

bash · salida recortada

```
Es un SMS para 987654321
ClassCastException: class EmailNotification cannot be cast to class
    SmsNotification ...
```

`instanceof` mira el tipo dinámico; el casting se verifica al ejecutar.

Error común: una cadena de `instanceof`

CostByInstanceof.java

```java
static long cost(Notification n) {
    if (n instanceof EmailNotification) {
        return 0;
    } else if (n instanceof
     SmsNotification) {
        return 5;
    } else {
        return 1;
    }
}
```

- Funciona: da lo mismo que `n.cost()`.

Error común: una cadena de `instanceof`

CostByInstanceof.java

```java
static long cost(Notification n) {
    if (n instanceof EmailNotification) {
        return 0;
    } else if (n instanceof
     SmsNotification) {
        return 5;
    } else {
        return 1;
    }
}
```

- Funciona: da lo mismo que `n.cost()`.
- Pero cada decisión por tipo (costo, ícono, reporte) tiene su propia cadena.

Error común: una cadena de `instanceof`

CostByInstanceof.java

```java
static long cost(Notification n) {
    if (n instanceof EmailNotification) {
        return 0;
    } else if (n instanceof
     SmsNotification) {
        return 5;
    } else {
        return 1;
    }
}
```

- Funciona: da lo mismo que `n.cost()`.
- Pero cada decisión por tipo (costo, ícono, reporte) tiene su propia cadena.
- Un tipo nuevo cae en el `else`: cuesta 1 céntimo sin que nadie lo decida.

Error común: una cadena de `instanceof`

CostByInstanceof.java

```java
static long cost(Notification n) {
    if (n instanceof EmailNotification) {
        return 0;
    } else if (n instanceof
     SmsNotification) {
        return 5;
    } else {
        return 1;
    }
}
```

- Funciona: da lo mismo que `n.cost()`.
- Pero cada decisión por tipo (costo, ícono, reporte) tiene su propia cadena.
- Un tipo nuevo cae en el `else`: cuesta 1 céntimo sin que nadie lo decida.
- Y nadie avisa.

Agregar un tipo de aviso

CON CADENAS DE INSTANCEOF

```
cost(n)
if (n instanceof Email…)
else if (n instanceof Sms…)
else …
```

```
report(n)
if (n instanceof Email…) …
```

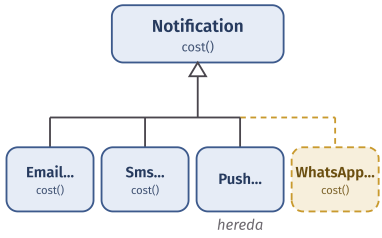
```
icon(n)
if (n instanceof Sms…) …
```

WhatsAppNotification
clase nueva

hay que encontrar y cambiar cada cadena;
si se olvida una, nadie avisa

CON OVERRIDE EN CADA SUBCLASE

```
total += n.cost();          no cambia
```



solo se escribe la clase nueva,
con su propio cost()

¿Cuándo es razonable un `instanceof`?

- Cuando la jerarquía no es nuestra y no podemos agregarle un método.

¿Cuándo es razonable un `instanceof`?

- Cuando la jerarquía no es nuestra y no podemos agregarle un método.
- En `equals`: hay que preguntar si el otro objeto es comparable.

¿Cuándo es razonable un `instanceof`?

- Cuando la jerarquía no es nuestra y no podemos agregarle un método.
- En `equals`: hay que preguntar si el otro objeto es comparable.
- Cuando la operación de verdad no le pertenece a la clase.

¿Cuándo es razonable un `instanceof`?

- Cuando la jerarquía no es nuestra y no podemos agregarle un método.
- En `equals`: hay que preguntar si el otro objeto es comparable.
- Cuando la operación de verdad no le pertenece a la clase.

> **Semana 8**
> Con tipos `sealed` y `switch`, el compilador sí avisa si falta un caso.

SECCIÓN 04

Lo que no es polimórfico

04

`static` y campos: decide el tipo estático

StaticDemo.java · recortado

```java
class Parent {
    String label = "campo de Parent";
    static String kind() { return "Parent.kind()"; }
    String describe() { ... }
}
class Child extends Parent {
    String label = "campo de Child";
    static String kind() { return "Child.kind()"; }
    @Override String describe() { ... }
}
```

main

```java
Parent p = new Child();
IO.println(p.describe());
IO.println(p.kind());
IO.println(p.label);
```

bash

```
Child.describe()
Parent.kind()
campo de Parent
```

`static` y campos: decide el tipo estático

StaticDemo.java · recortado

```java
class Parent {
    String label = "campo de Parent";
    static String kind() { return "Parent.kind()"; }
    String describe() { ... }
}
class Child extends Parent {
    String label = "campo de Child";
    static String kind() { return "Child.kind()"; }
    @Override String describe() { ... }
}
```

main

```java
Parent p = new Child();
IO.println(p.describe());
IO.println(p.kind());
IO.println(p.label);
```

bash

```
Child.describe()
Parent.kind()
campo de Parent
```

`static` y campos: decide el tipo de `p`. En la subclase se ocultan *(hiding)*.

`javac` lo marca

bash

```
StaticDemo.java:34: warning: [static] static method should be qualified by type
     name,
Parent, instead of by an expression
        IO.println(p.kind());       // static: decide el tipo de p
                   ^
```

- `p.kind()` parece una llamada a través del objeto, y no lo es.

`javac` lo marca

```
bash
StaticDemo.java:34: warning: [static] static method should be qualified by type
     name,
Parent, instead of by an expression
        IO.println(p.kind());       // static: decide el tipo de p
                   ^
```

- `p.kind()` parece una llamada a través del objeto, y no lo es.
- Regla: `static` con el nombre de la clase, `Parent.kind()`.

`javac` lo marca

```
bash
StaticDemo.java:34: warning: [static] static method should be qualified by type
     name,
Parent, instead of by an expression
        IO.println(p.kind());       // static: decide el tipo de p
                   ^
```

- `p.kind()` parece una llamada a través del objeto, y no lo es.
- Regla: `static` con el nombre de la clase, `Parent.kind()`.
- Y no repetir en una subclase el nombre de un campo de la superclase.

Una trampa: el constructor que llama a un método

Tag.java

```java
public class Tag {
    public Tag() {
        IO.println("Tag(): "
                + describe());
    }
    public String describe() {
        return "etiqueta";
    }
}
```

LabeledTag.java

```java
public final class LabeledTag extends Tag {
    private final String label;
    public LabeledTag(String label) {
        super();
        this.label = label;
        IO.println("LabeledTag(): "
                + describe());
    }
    @Override public String describe() {
        return "etiqueta " + label;
    }
}
```

¿De dónde salió el `null`?

bash

```
$ java -cp build ConstructorTrap
Tag(): etiqueta null
LabeledTag(): etiqueta A-7
```

¿De dónde salió el `null`?

bash

```
$ java -cp build ConstructorTrap
Tag(): etiqueta null
LabeledTag(): etiqueta A-7
```

1. `new LabeledTag("A-7")` reserva el objeto: `label` empieza en `null`.

¿De dónde salió el `null`?

bash

```
$ java -cp build ConstructorTrap
Tag(): etiqueta null
LabeledTag(): etiqueta A-7
```

1. `new LabeledTag("A-7")` reserva el objeto: `label` empieza en `null`.
2. Lo primero es `super()`: corre `Tag()`, que llama a `describe()`.

¿De dónde salió el `null`?

bash

```
$ java -cp build ConstructorTrap
Tag(): etiqueta null
LabeledTag(): etiqueta A-7
```

1. `new LabeledTag("A-7")` reserva el objeto: `label` empieza en `null`.
2. Lo primero es `super()`: corre `Tag()`, que llama a `describe()`.
3. El objeto ya es un `LabeledTag`: se ejecuta la versión de la subclase… antes de `this.label = label`.

¿De dónde salió el `null`?

```bash
$ java -cp build ConstructorTrap
Tag(): etiqueta null
LabeledTag(): etiqueta A-7
```

1. `new LabeledTag("A-7")` reserva el objeto: `label` empieza en `null`.
2. Lo primero es `super()`: corre `Tag()`, que llama a `describe()`.
3. El objeto ya es un `LabeledTag`: se ejecuta la versión de la subclase… antes de `this.label = label`.

Un campo `final` se ve con dos valores distintos.

[this-escape]

bash

```
Tag.java:10: warning: [this-escape] possible 'this' escape before subclass is
    fully initialized
        IO.println("Tag(): " + describe());
                                  ^
```

- Desde el JDK 21, `javac -Xlint:all` detecta el patrón.

[this-escape]

bash

```
Tag.java:10: warning: [this-escape] possible 'this' escape before subclass is
    fully initialized
        IO.println("Tag(): " + describe());
                                  ^
```

- Desde el JDK 21, `javac -Xlint:all` detecta el patrón.
- `this` escapa del constructor antes de que el objeto esté completo.

[this-escape]

```bash
Tag.java:10: warning: [this-escape] possible 'this' escape before subclass is
    fully initialized
        IO.println("Tag(): " + describe());
                                   ^
```

- Desde el JDK 21, `javac -Xlint:all` detecta el patrón.
- `this` escapa del constructor antes de que el objeto esté completo.

Regla práctica

Un constructor solo llama a métodos `private`, `static` o `final` de su propia clase.

SECCIÓN 05

Clases abstractas

05



¿Cuánto gana un empleado “en general”?

- La planilla de la bodega: empleados con sueldo fijo y empleados por hora.

¿Cuánto gana un empleado “en general”?

- La planilla de la bodega: empleados con sueldo fijo y empleados por hora.
- Los dos tienen nombre y boleta de pago. El pago se calcula distinto.

¿Cuánto gana un empleado “en general”?

- La planilla de la bodega: empleados con sueldo fijo y empleados por hora.
- Los dos tienen nombre y boleta de pago. El pago se calcula distinto.
- Superclase común `Employee`. Pero ¿qué devuelve `monthlyPay()` de un empleado en general?

¿Cuánto gana un empleado “en general”?

- La planilla de la bodega: empleados con sueldo fijo y empleados por hora.
- Los dos tienen nombre y boleta de pago. El pago se calcula distinto.
- Superclase común `Employee`. Pero ¿qué devuelve `monthlyPay()` de un empleado en general?
- No hay respuesta sensata: no existen empleados en general.

¿Cuánto gana un empleado “en general”?

- La planilla de la bodega: empleados con sueldo fijo y empleados por hora.
- Los dos tienen nombre y boleta de pago. El pago se calcula distinto.
- Superclase común `Employee`. Pero ¿qué devuelve `monthlyPay()` de un empleado en general?
- No hay respuesta sensata: no existen empleados en general.

Clase abstracta ***(abstract class)***

Una clase que existe para ser extendida. Sirve como tipo y como superclase, pero no para hacer `new`.

abstract

Employee.java

```java
public abstract class Employee {
    private final String name;
    protected Employee(String name) {
        this.name = name;
    }
    public String name() { return name; }
    public abstract long monthlyPay();
    public String payslip() {
        return name + ": " + soles(monthlyPay()
    );
    }
}
```

- `abstract class`: no se instancia.

abstract

Employee.java

```java
public abstract class Employee {
    private final String name;
    protected Employee(String name) {
        this.name = name;
    }
    public String name() { return name; }
    public abstract long monthlyPay();
    public String payslip() {
        return name + ": " + soles(monthlyPay()
    );
    }
}
```

- `abstract class`: no se instancia.
- `abstract long monthlyPay();` sin cuerpo: la clase promete la operación.

abstract

Employee.java

```java
public abstract class Employee {
    private final String name;
    protected Employee(String name) {
        this.name = name;
    }
    public String name() { return name; }
    public abstract long monthlyPay();
    public String payslip() {
        return name + ": " + soles(monthlyPay()
    );
    }
}
```

- `abstract class`: no se instancia.
- `abstract long monthlyPay();` sin cuerpo: la clase promete la operación.
- Con un método abstracto, la clase tiene que ser abstracta.

abstract

Employee.java

```java
public abstract class Employee {
    private final String name;
    protected Employee(String name) {
        this.name = name;
    }
    public String name() { return name; }
    public abstract long monthlyPay();
    public String payslip() {
        return name + ": " + soles(monthlyPay()
    );
    }
}
```

- `abstract class`: no se instancia.
- `abstract long monthlyPay();` sin cuerpo: la clase promete la operación.
- Con un método abstracto, la clase tiene que ser abstracta.
- Campos, constructor y métodos concretos: sí puede tenerlos.

No se instancia

Wrong.java

```java
Employee e = new Employee("carol");
```

bash

```
Wrong.java:5: error: Employee is abstract; cannot be instantiated
        Employee e = new Employee("carol");
                     ^
```

Sirve como tipo: `Employee e = new SalariedEmployee(...)` sí compila.

Toda subclase concreta tiene que completarlo

Contractor.java

```java
public class Contractor extends Employee {
    public Contractor(String name) { super(name); }
}
```

bash

```
Contractor.java:1: error: Contractor is not abstract and does not override
    abstract method
monthlyPay() in Employee
```

Dos salidas: implementar `monthlyPay()` o declarar `Contractor` también abstracta.

Las subclases concretas

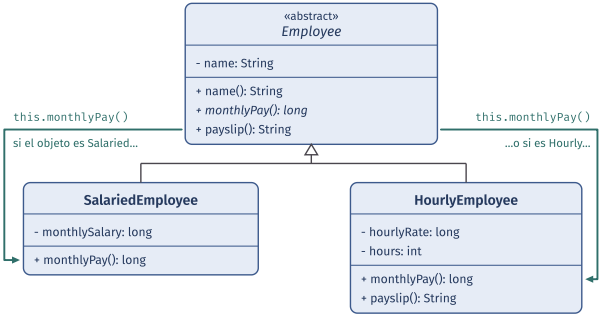
SalariedEmployee.java

```java
public class SalariedEmployee
        extends Employee {
    private final long monthlySalary
     ;
    // constructor: super(name)
    @Override
    public long monthlyPay() {
        return monthlySalary;
    }
}
```

HourlyEmployee.java

```java
public class HourlyEmployee
        extends Employee {
    private final long hourlyRate;
    private final int hours;
    // constructor: super(name)
    @Override
    public long monthlyPay() {
        return hourlyRate * hours;
    }
    // y payslip() con las horas
}
```

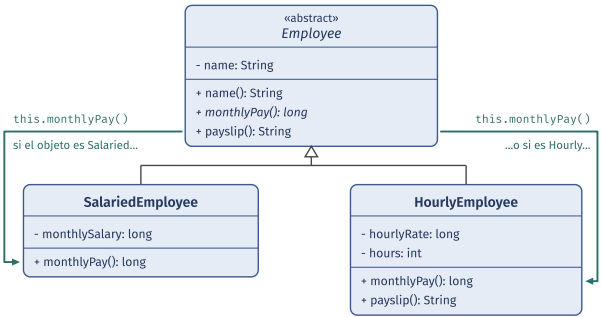
El patrón Template Method



payslip() se escribe una vez en Employee; cada subclase concreta dice cuánto se paga.

- `payslip()` se escribe una vez, en `Employee`.

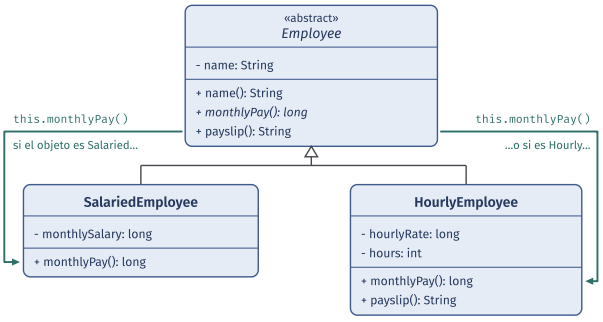
El patrón Template Method



payslip() se escribe una vez en Employee; cada subclase concreta dice cuánto se paga.

- `payslip()` se escribe una vez, en `Employee`.
- Su llamada a `monthlyPay()` va a la subclase del objeto.

El patrón Template Method



payslip() se escribe una vez en Employee; cada subclase concreta dice cuánto se paga.

- `payslip()` se escribe una vez, en `Employee`.
- Su llamada a `monthlyPay()` va a la subclase del objeto.
- **Template Method**: la superclase escribe el algoritmo; las subclases llenan los huecos.

Una planilla polimórfica

PayrollDemo.java · recortado

```java
ArrayList<Employee> staff = new ArrayList<>();
staff.add(new SalariedEmployee("alice", 350_000));
staff.add(new HourlyEmployee("bob", 1_250, 96));
for (Employee e : staff) {
    IO.println(e.payslip());
}
```

bash

```
alice: S/ 3500.00
bob: S/ 1200.00 (96 h)
```

Una planilla polimórfica

PayrollDemo.java · recortado

```java
ArrayList<Employee> staff = new ArrayList<>();
staff.add(new SalariedEmployee("alice", 350_000));
staff.add(new HourlyEmployee("bob", 1_250, 96));
for (Employee e : staff) {
    IO.println(e.payslip());
}
```

bash

```
alice: S/ 3500.00
bob: S/ 1200.00 (96 h)
```

Bob: 1 250 céntimos $\times$ 96 h = 120 000 céntimos.

Una planilla polimórfica

PayrollDemo.java · recortado

```java
ArrayList<Employee> staff = new ArrayList<>();
staff.add(new SalariedEmployee("alice", 350_000));
staff.add(new HourlyEmployee("bob", 1_250, 96));
for (Employee e : staff) {
    IO.println(e.payslip());
}
```

bash

```
alice: S/ 3500.00
bob: S/ 1200.00 (96 h)
```

Bob: 1 250 céntimos $\times$ 96 h = 120 000 céntimos. `super.payslip()` no despacha: llama directo a la versión de `Employee`.

Constructores y UML

¿Constructor sin `new`?

Sí: inicializa la parte del objeto que le corresponde (`name`). Cada subclase lo llama con `super(name)`. Va `protected`: solo lo usan las subclases.

Constructores y UML

¿Constructor sin `new`?

Sí: inicializa la parte del objeto que le corresponde (`name`). Cada subclase lo llama con `super(name)`. Va `protected`: solo lo usan las subclases.

En UML

Clase y método abstractos en *cursiva*, o «abstract» sobre el nombre.

Mermaid

```
class Employee {
    <<Abstract>>
    +monthlyPay() long*
    +payslip() String
}
```

Error común: un método “para que compile”

Employee.java · mal

```java
public long monthlyPay() {
    return 0;   // las subclases
                // le hacen
     override
}
```

- `Employee` ya no necesita ser abstracta…
  y el compilador deja de vigilar.

Error común: un método “para que compile”

Employee.java · mal

```java
public long monthlyPay() {
    return 0;   // las subclases
                // le hacen
     override
}
```

- `Employee` ya no necesita ser abstracta… y el compilador deja de vigilar.
- Una subclase que lo olvida compila y paga cero.

Error común: un método “para que compile”

Employee.java · mal

```java
public long monthlyPay() {
    return 0;   // las subclases
                // le hacen
     override
}
```

- `Employee` ya no necesita ser abstracta… y el compilador deja de vigilar.
- Una subclase que lo olvida compila y paga cero.
- Con `throw new UnsupportedOperationException()`: el error pasa de la compilación a la ejecución.

Error común: un método “para que compile”

Employee.java · mal

```java
public long monthlyPay() {
    return 0;   // las subclases
                // le hacen override
}
```

- `Employee` ya no necesita ser abstracta… y el compilador deja de vigilar.
- Una subclase que lo olvida compila y paga cero.
- Con `throw new UnsupportedOperationException()`: el error pasa de la compilación a la ejecución.

Si toda subclase concreta tiene que darlo, es `abstract`

Que el compilador haga el trabajo.

Probar una jerarquía

EmployeeTest.java

```java
@Test
void totalOverAListOfEmployees() {
    List<Employee> staff = List.of(
        new SalariedEmployee("alice", 350_000)
    ,
        new HourlyEmployee("bob", 1_250, 96));
    long total = 0;
    for (Employee e : staff) {
        total += e.monthlyPay();
    }
    assertEquals(470_000, total);
}
```

- Variables de tipo `Employee`: así usa la jerarquía el resto del programa.

Probar una jerarquía

EmployeeTest.java

```java
@Test
void totalOverAListOfEmployees() {
    List<Employee> staff = List.of(
        new SalariedEmployee("alice", 350_000)
    ,
        new HourlyEmployee("bob", 1_250, 96));
    long total = 0;
    for (Employee e : staff) {
        total += e.monthlyPay();
    }
    assertEquals(470_000, total);
}
```

- Variables de tipo `Employee`: así usa la jerarquía el resto del programa.
- Otra prueba fija que `payslip()` use el monto de la subclase.

Probar una jerarquía

EmployeeTest.java

```java
@Test
void totalOverAListOfEmployees() {
    List<Employee> staff = List.of(
        new SalariedEmployee("alice", 350_000)
    ,
        new HourlyEmployee("bob", 1_250, 96));
    long total = 0;
    for (Employee e : staff) {
        total += e.monthlyPay();
    }
    assertEquals(470_000, total);
}
```

- Variables de tipo `Employee`: así usa la jerarquía el resto del programa.
- Otra prueba fija que `payslip()` use el monto de la subclase.
- `List.of(...)`: lista no modificable, cómoda para pruebas (semana 11).

SECCIÓN 06

Laboratorio

06

Laboratorio: polimorfismo en el proyecto

1. **Traza en papel**: predecir `LabTrace`; luego `java -cp build LabTrace`.

```
bash
$ cd code/week-06
$ make test
...
== PayrollDemo
alice: S/ 3500.00
bob: S/ 1200.00 (96 h)
...
[        11 tests successful      ]
[         0 tests failed          ]
```

Salida recortada.

Laboratorio: polimorfismo en el proyecto

1. **Traza en papel**: predecir `LabTrace`; luego `java -cp build LabTrace`.
2. Quitar la cadena de `instanceof` de `ShippingCalculator`: un override de `cost` en `Letter`, `Parcel` y `Pallet`. Las pruebas no cambian.

```bash
$ cd code/week-06
$ make test
...
== PayrollDemo
alice: S/ 3500.00
bob: S/ 1200.00 (96 h)
...
[        11 tests successful      ]
[         0 tests failed          ]
```

Salida recortada.

Laboratorio: polimorfismo en el proyecto

1. **Traza en papel**: predecir `LabTrace`; luego `java -cp build LabTrace`.
2. Quitar la cadena de `instanceof` de `ShippingCalculator`: un override de `cost` en `Letter`, `Parcel` y `Pallet`. Las pruebas no cambian.
3. Una clase abstracta para los tipos de producto de Bodega, con pruebas sobre una lista de la superclase.

bash

```
$ cd code/week-06
$ make test
...
== PayrollDemo
alice: S/ 3500.00
bob: S/ 1200.00 (96 h)
...
[        11 tests successful      ]
[         0 tests failed          ]
```

Salida recortada.

Laboratorio: polimorfismo en el proyecto

1. **Traza en papel**: predecir `LabTrace`; luego `java -cp build LabTrace`.
2. Quitar la cadena de `instanceof` de `ShippingCalculator`: un override de `cost` en `Letter`, `Parcel` y `Pallet`. Las pruebas no cambian.
3. Una clase abstracta para los tipos de producto de Bodega, con pruebas sobre una lista de la superclase.
4. Diagrama de clases de la jerarquía de productos: arranque de **P2**.

bash

```
$ cd code/week-06
$ make test
...
== PayrollDemo
alice: S/ 3500.00
bob: S/ 1200.00 (96 h)
...
[        11 tests successful      ]
[         0 tests failed          ]
```

Salida recortada.

Resumen

- **Sustitución**: un objeto de la subclase sirve donde se espera la superclase (upcasting implícito).
- Tipo **estático**: qué se puede llamar (compilador). Tipo **dinámico**: qué se ejecuta (JVM).
- **Dynamic dispatch**: `invokevirtual` con la firma; la JVM busca desde la clase del objeto hacia arriba (en HotSpot, con una vtable).
- **Overloading** al compilar; **override** al ejecutar.
- Downcasting puede fallar (`ClassCastException`); una cadena de `instanceof` suele ser un override que falta.
- `static` y campos no son polimórficos. Un constructor no llama a métodos que admiten override.
- **Clase abstracta**: no se instancia. **Método abstracto**: obliga a cada subclase concreta. **Template Method**: lo concreto llama a lo abstracto.

Para la próxima clase

- Lecturas de la semana 7: Eck, subsecs. 5.7.1 a 5.7.3 (interfaces).
- Entregar **P1**. Leer el enunciado de **P2**: Catálogo.
- Terminar el laboratorio: `ShippingCalculator` sin `instanceof`.

Pregunta para pensar

Una factura y un empleado no tienen ninguna superclase en común, pero los dos se pueden pagar. ¿Cómo escribirían un `for` que recorra "todo lo que hay que pagar"?