


Programación Orientada a Objetos

# Tipos sealed y pattern matching

## Repaso para el examen parcial

**Gerson Garrido**

Semana 8

UNIVERSIDAD NACIONAL DE INGENIERÍA

# Hoy

1. Jerarquías cerradas: `sealed` y `permits`
2. Pattern matching con `instanceof` y en `switch`
3. Exhaustividad: el compilador avisa cuando falta un caso
4. Ejemplo: un evaluador de expresiones
5. Dos estilos: override o `switch`
6. Repaso para el parcial

> **Lectura**
>
> Eck, subsec. 3.6.5 (*Switch Expressions*). Java 25, *Java Language Updates*: *Sealed Classes*, *Pattern Matching* y *Pattern Matching with switch*. dev.java: *Using Pattern Matching*.

SECCIÓN 01

# Jerarquías cerradas

01

# Recuerdo de la semana 6: la cadena de `instanceof`

Shapes.java

```java
static double areaWithInstanceof(Shape s) {
    if (s instanceof Circle c) {
        return Math.PI * c.radius() * c.radius();
    } else if (s instanceof Square sq) {
        return sq.side() * sq.side();
    }
    throw new IllegalArgumentException(
            "figura desconocida: " + s);
}
```

- Olvidó los triángulos. Compila igual.
- El error aparece recién al ejecutar: `figura desconocida: Triangle[...]`.
- El compilador no puede avisar: la interfaz está abierta y nunca sabe cuáles son *todos* los subtipos.

# `sealed` y `permits`

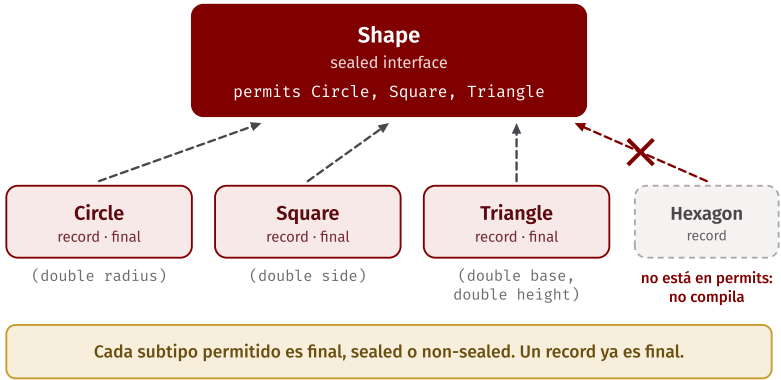
Shape.java, Circle.java, ...

```java
public sealed interface Shape
        permits Circle, Square, Triangle {
}

public record Circle(double radius)
        implements Shape { ... }
public record Square(double side)
        implements Shape { ... }
public record Triangle(double base,
        double height) implements Shape { ... }
```

- Una interfaz **sealed** nombra sus subtipos directos.
- Ningún otro puede implementarla: la jerarquía queda cerrada.
- Records de la semana 3: datos inmutables + jerarquía cerrada.
- No cambia lo que el programa hace: cambia lo que el compilador puede verificar.

# Una jerarquía sealed



# Fuera de la lista: no compila

broken/Hexagon.java

```java
public record Hexagon(double side) implements Shape {
}
```

bash · make errores

```
broken/Hexagon.java:2: error: class is not allowed to extend sealed class:
    Shape (as it is not listed in its 'permits' clause)
public record Hexagon(double side) implements Shape {
       ^
```

Los subtipos permitidos van en el mismo paquete. Si están en el mismo archivo, `permits` se puede omitir. Desde Java 17 *(JEP 409)*.

# `final`, `sealed` o `non-sealed`

| Modificador | Significa |
|---|---|
| `final` | Nadie lo extiende. |
| `sealed` | Se cierra otra vez. |
| `non-sealed` | Se abre a propósito. |

- Cada subtipo permitido tiene que elegir uno.
- Un `record` ya es `final`.

Vehicle.java

```java
sealed interface Vehicle permits Car,
    Truck { }
class Car implements Vehicle { }
final class Truck implements Vehicle { }
```

bash

```
Vehicle.java:4: error: sealed, non-
    sealed or final modifiers expected
class Car implements Vehicle {
^
```

SECCIÓN 02

# Pattern matching

02



# `instanceof` con patrón

antes de Java 16

```
1  if (o instanceof Fraction) {
2      Fraction f = (Fraction) o;
3      ...
4  }
```

como en equals, semana 5

```
1  if (o instanceof Fraction f) {
2      return num == f.num && den == f.
        den;
3  }
```

- `Fraction f` es un **patrón de tipo**: prueba y declara en un paso, sin casting.
- `f` existe solo donde la prueba es verdadera.

alcance

```
1  if (!(o instanceof String s)) {
2      return 0;
3  }
4  return s.length();        //
       compila
```

- Con `&&`, sí; con `||`, no.

# `switch` con patrones de tipo

Java 21+

```java
double area = switch (s) {
    case Circle c -> Math.PI * c.radius() * c.radius();
    case Square sq -> sq.side() * sq.side();
    case Triangle t -> t.base() * t.height() / 2;
};
```

- Cada `case` prueba el tipo y declara la variable.
- Con `->`, el `switch` es una **expresión**: produce un valor (Eck, 3.6.5).
- Es la cadena de `instanceof`… pero el compilador va a poder revisarla.

# Patrones de record: desarmar en el `case`

Shapes.java

```java
public static double area(Shape s) {
    return switch (s) {
        case Circle(double r) -> Math.PI * r * r;
        case Square(double side) -> side * side;
        case Triangle(double b, double h) -> b * h / 2;
    };
}
```

- `Circle(double r)`: "si es un `Circle`, llama `r` a su radio".
- Orden y tipo como en la declaración; el nombre lo eligen ustedes.

- `var` sirve: `Triangle(var b, var h)`.
- Se anidan: `Mul(Num(int v), Expr r)`.
- `_` para lo que no se usa *(Java 22)*.

# Guards con `when` y `case null`

Shapes.java · describe

```java
return switch (s) {
    case null -> "sin figura";
    case Circle c when c.radius() >= 10 -> "círculo grande";
    case Circle c -> "círculo de radio " + c.radius();
    case Square(double side) when side == 1 -> "cuadrado unitario";
    case Square sq -> "cuadrado de lado " + sq.side();
    case Triangle t -> "triángulo de base " + t.base();
};
```

- **Guard**: el patrón coincide y la condición es verdadera.
- Se prueba en orden y gana el primero que coincide: el guard va *antes* del caso general.
- Sin `case null`, un `null` lanza `NullPointerException`.

# Un caso que nunca se alcanza: no compila

broken/Dominated.java

```java
return switch (s) {
    case Shape any -> "una figura";
    case Circle c -> "un círculo";
};
```

bash · make errores

```
broken/Dominated.java:6: error: this case label is dominated by a preceding
    case label
            case Circle c -> "un círculo";
```

Caso **dominado** *(dominated)*: un caso anterior ya captura todo lo que captura este.

# Todo junto: ShapeDemo

Para cada figura de la lista: `s`, su área y `Shapes.describe(s)`.

bash

```
$ java -cp build ShapeDemo
Circle[radius=1.0] -> área 3.14, círculo de radio 1.0
Square[side=2.0] -> área 4.00, cuadrado de lado 2.0
Triangle[base=3.0, height=4.0] -> área 6.00, triángulo de base 3.0
Circle[radius=10.0] -> área 314.16, círculo grande
sin figura
```

- `Circle(10)` entra por el guard y nunca llega al caso general.
- La primera parte de cada línea es el `toString` que generó el record.

SECCIÓN 03

# Exhaustividad

03

# Sin `default`: el compilador revisa

- `area` trata los tres subtipos: es **exhaustivo**.
- Antes había solo `Circle` y `Square`. Se agrega `Triangle`...
- ...y cada `switch` incompleto deja de compilar.

broken/AreaMissingCase.java

```
1  return switch (s) {
2      case Circle(double r) -> Math.PI * r *
         r;
3      case Square(double side) -> side *
       side;
4  };
```

bash · make errores

```
broken/AreaMissingCase.java:4: error: the switch expression does not cover all
    possible input values
```

# Agregar un subtipo

```
sealed interface Shape permits Circle, Square, Triangle
```

**area(s): switch sin default**

```
case Circle(double r) ->
case Square(double side) ->
case Triangle(double b, double h) ->
```

compila

**describe(s): switch sin default**

```
case Circle c ->
case Square sq ->
case Triangle t ->
```

compila

**switch con default**

```
case Circle(double r) ->
case Square(double side) ->
default -> 0;
```

compila: Triangle da 0

**cadena de instanceof**

```
if (s instanceof Circle c) ...
else if (s instanceof Square sq) ...
throw ...;
```

compila: falla al ejecutar

3. Se completan esos dos switch. Los otros dos siguen con el error escondido.

# Error común: el `default` “por seguridad”

broken/ChainNoWarning.java

```java
return switch (s) {
    case Circle(double r) -> Math.PI * r *
      r;
    case Square(double side) -> side *
     side;
    default -> 0;
};
```

bash · make errores

```
== broken/ChainNoWarning.java
(compila sin errores)
```

- Compila. El área de un triángulo es 0, en silencio.
- `default` dice “todo lo demás va aquí”: el compilador ya no tiene nada que avisar.
- Sobre un tipo sealed, no escriban `default`.
- Si solo se recompila `Shape`, el `switch` lanza `MatchException` al ejecutar: tarde, pero no en silencio.

SECCIÓN 04

# Evaluador de expresiones

04

# Expresiones: una jerarquía cerrada y recursiva

Expr.java, Num.java, Add.java, Mul.java

```java
public sealed interface Expr
        permits Num, Add, Mul {
}
public record Num(int value) implements Expr {
      }
public record Add(Expr left, Expr right)
        implements Expr { }
public record Mul(Expr left, Expr right)
        implements Expr { }
```

- Una expresión es un número, una suma o un producto. Nada más.
- Recursiva: los lados de `Add` y `Mul` son expresiones.
- `2 + 3 * 4` es `new Add(new Num(2), new Mul(new Num(3), new Num(4)))`.

# Cada operación, un `switch`

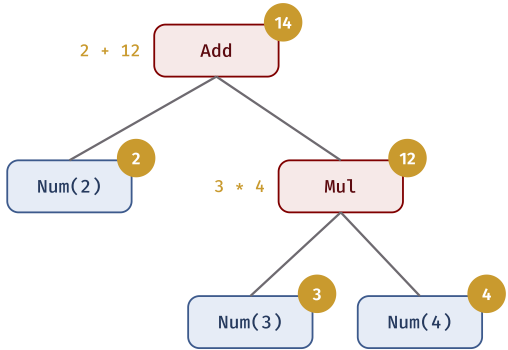
Calc.java

```java
public static int eval(Expr e) {
    return switch (e) {
        case Num(int v) -> v;
        case Add(Expr l, Expr r) -> eval(l) + eval(r);
        case Mul(Expr l, Expr r) -> eval(l) * eval(r);
    };
}
```

- La recursión sigue la forma de los datos.
- `show` es otro `switch` igual, que arma el texto: `(2 + 3 * 4)`.
- Sin `default`: si se agrega un tipo de expresión, el compilador marca `eval` y `show`.

# `eval`, paso a paso

EL ÁRBOL DE LA EXPRESIÓN




CALC.EVAL

```
return switch (e) {
  case Num(int v)
    -> v;
  case Add(Expr l, Expr r)
    -> eval(l) + eval(r);
  case Mul(Expr l, Expr r)
    -> eval(l) * eval(r);
};
```

3. De vuelta en eval(Add): 2 + 12 = 14.

# Patrones anidados: `simplify`

Calc.java

```
return switch (e) {
    case Mul(Num(int v), Expr r) when v == 1 -> simplify(r);
    case Mul(Expr l, Num(int v)) when v == 1 -> simplify(l);
    case Add(Num(int v), Expr r) when v == 0 -> simplify(r);
    case Add(Expr l, Num(int v)) when v == 0 -> simplify(l);
    case Add(Expr l, Expr r) -> new Add(simplify(l), simplify(r));
    case Mul(Expr l, Expr r) -> new Mul(simplify(l), simplify(r));
    case Num n -> n;
};
```

- Los casos con guard no cuentan para la exhaustividad.
- Los tres últimos, sin guard, cubren los tres tipos: exhaustivo.

## ExprDemo

bash

```
$ java -cp build ExprDemo
Add[left=Num[value=2], right=Mul[left=Num[value=3], right=Num[value=4]]]
(2 + 3 * 4) = 14
(2 + 3) * 4 = 20
(1 * 7 + 0)  se simplifica a  7
```

- La primera línea es el `toString` de los records: el árbol tal como está en memoria.
- El árbol, no el texto, decide el orden de las operaciones.

SECCIÓN 05

# Dos estilos

05

# El mismo evaluador, con override en cada tipo

ExprMethods.java

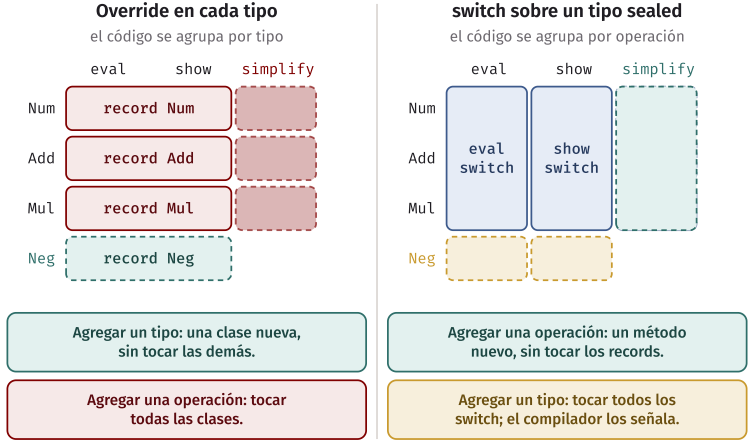
```java
interface Expr {
    int eval();
    String show();
}
record Add(Expr left, Expr right)
     implements Expr {
    public int eval() {
        return left.eval() + right.eval();
    }
    public String show() {
        return "(" + left.show() + " + "
                + right.show() + ")";
    }
}
```

bash

```
$ java -cp build ExprMethods
(2 + 3 * 4) = 14
```

- Mismo resultado, organización opuesta.
- Aquí, el código de cada **tipo** está junto.
- En `Calc`, el de cada **operación**.

# Tipos × operaciones



# ¿Cuál elegir?

| | **Override en cada tipo** | **`switch` sobre un tipo sealed** |
|---|---|---|
| Agregar un **tipo** | Fácil: una clase nueva | Tocar cada `switch`; el compilador los señala |
| Agregar una **operación** | Tocar cada clase | Fácil: un método nuevo |
| ¿Quién agrega tipos? | Cualquiera | Quien controla `permits` |
| Encaja con | Capacidades: `Payable` | Datos de forma fija |

**La pregunta de diseño**

¿Qué va a cambiar más: los tipos o las operaciones?

SECCIÓN 06

# Repaso para el parcial

06

# Qué entra: semanas 1 a 8

| Sem. | Saber hacer |
|---|---|
| 1–2 | Stack y heap línea por línea; alias, `null`; `==` frente a `equals`. |
| 2–3 | Qué constructor o qué overload de un método se elige; `this`, `static`. |
| 3 | Enunciar un invariante y encontrar quién lo rompe; record o clase; leer y escribir pruebas JUnit. |
| 4 | Arreglos y `ArrayList`; responsabilidades y diagrama de clases. |
| 5 | `super`, override; `toString`, `equals`, `hashCode` y sus contratos. |
| 6 | Tipo estático y dinámico; trazas con overloading y override; clases abstractas. |
| 7 | Interfaz, clase abstracta o composición; violaciones de Liskov. |
| 8 | Solo lectura: qué devuelve un `switch` con patrones o por qué no compila. |

# Cómo estudiar

1. Las *Preguntas de repaso* de las guías 1 a 8 y los ejercicios de los apuntes.
2. Predecir antes de ejecutar: escribir la salida de cada ejemplo de `code/week-NN/` y después comprobarla con `java` o `jshell`.
3. Rehacer el diseño de un dominio pequeño: tarjetas CRC, diagrama de clases, invariantes, pruebas.
4. Los cuestionarios de Eck, capítulos 5 y 7.

**En los apuntes**

La sección *Repaso para el examen parcial* tiene diez ejercicios tipo examen, sin respuestas. Se resuelven en papel, justificando cada paso.

# En clase: overloading y override

R2

```
class Account {
    String kind() { return "cuenta"; }
    String transferTo(Account other) { return "Account -> Account"; }
}
class SavingsAccount extends Account {
    @Override String kind() { return "ahorros"; }
    String transferTo(SavingsAccount other) { return "Savings -> Savings"; }
    @Override String transferTo(Account other) { return "Savings -> Account"; }
}
```

Dos preguntas por cada llamada. Al compilar: ¿qué firma se elige, según el tipo estático? Al ejecutar: ¿qué versión de esa firma, según el objeto?

# En clase: ¿qué imprime?

R2

```
1  Account a = new SavingsAccount();
2  SavingsAccount s = new SavingsAccount();
3  IO.println(a.kind());
4  IO.println(a.transferTo(s));
5  IO.println(s.transferTo(s));
6  IO.println(s.transferTo(a));
7  IO.println(((Account) s).transferTo(s));
```

- Primero, la firma: `transferTo(Account)` o `transferTo(SavingsAccount)`.
- Después, la versión: la de `Account` o la de `SavingsAccount`.
- Un casting cambia el tipo estático, no el objeto.

# En clase: un constructor que llama a un método con override

R5

```
class Base {
    Base() { IO.println("Base: " + describe()); }
    String describe() { return "base"; }
}
class Derived extends Base {
    private final String name;
    Derived(String n) { super(); name = n; }
    @Override String describe() { return "derived " + name; }
}
```

- ¿Qué imprime `new Derived("x")`?
- ¿En qué orden se ejecuta cada línea?
- ¿Qué vale `name` cuando se llama a `describe()`?
- ¿Cómo lo evitarían?

# En clase: Liskov

R7

```java
class ReadOnlyPlaylist extends Playlist {
    @Override
    public void add(Song s) {
        throw new UnsupportedOperationException("solo lectura");
    }
}
```

- ¿Qué expectativa de `Playlist` se rompe?
- Escriban una prueba que pase con `Playlist` y falle con `ReadOnlyPlaylist`, recibiendo el objeto como `Playlist`.
- ¿Qué diseño, con interfaz o composición, evita el problema?

# En clase: leer un `switch` con patrones

R10

```java
static String toll(Vehicle v) {
    return switch (v) {
        case Car(String p, int seats) when seats > 5 -> "minivan " + p;
        case Car c -> "auto " + c.plate();
        case Bike(boolean e) when e -> "bici eléctrica";
        case Bike b -> "bici";
    };
}
```

- ¿Qué devuelve para `new Car("ABC-123", 7)`? ¿Y para `new Bike(false)`?
- ¿Compila si se borra `case Bike b`? ¿Y si `case Car c` va primero?
- Se agrega `Truck` a `permits`: ¿qué pasa con `toll`? ¿Y si tuviera `default`?

SECCIÓN 07

# Laboratorio

07

# Laboratorio: repaso con ejercicios

1. Reescribir `PaymentFees` con un tipo sealed y `switch`; agregar un tipo de pago y ver qué marca el compilador. `PaymentFeesTest` tiene que seguir pasando.
2. Ejercicios tipo examen en parejas.
3. Leer pruebas JUnit ajenas: qué fijan y qué caso falta.

PaymentFees.java · código inicial

```java
if (p instanceof Cash) {
    return 0;
} else if (p instanceof Card c) {
    return c.credit() ? c.cents() * 3 / 100 : c.cents() / 100;
} else if (p instanceof Transfer) {
    return 150;
}
throw new IllegalArgumentException("forma de pago desconocida: " + p);
```

# Resumen

- `sealed ... permits ...` cierra la jerarquía. Cada subtipo es `final`, `sealed` o `non-sealed`; un record ya es `final`.
- `x instanceof Circle c`: prueba y declara, sin casting.
- `switch` con patrones de tipo y de **record** (que se anidan), guards `when` y `case null`. Los casos se prueban en orden; un caso dominado no compila.
- Sobre un tipo sealed, un `switch` sin `default` es **exhaustivo**: si se agrega un subtipo, cada `switch` incompleto deja de compilar.
- Override en cada tipo: fácil agregar tipos. `switch` sobre un tipo sealed: fácil agregar operaciones.

# Para la próxima clase

- **Sesión B: examen parcial**, semanas 1 a 8. Formato en `exams/README.md`.
- Repaso: la sección final de los apuntes de esta semana y las *Preguntas de repaso* de cada guía.
- Semana 9, excepciones: Eck, secs. 3.7 (3.7.1 y 3.7.2) y 8.3 (8.3.1 a 8.3.3).
- **P2** (Catálogo) se entrega la semana 9.

**Pregunta para pensar**

En P2, los productos se venden por unidad, por peso o vencen. Si el precio, la boleta y la etiqueta dependen del tipo, ¿override en cada tipo o un tipo sealed con `switch`? ¿Qué va a cambiar más?