Tema de la clase ¿De qué hablo?

- Relacionado con la asignatura `Estructuras de Datos`
- No cubierto por la docencia reglada de la asignatura
- Que sea interesante para vuestro currículum
- Que se esté usando ya o se prevea usar en un futuro próximo
- Que me guste a mí 😜

Consultando el temario

- Introducción a los tipos abstractos
    - Tipos algebraicos y patrones. Datos `mutables` e `inmutables`

Antecedentes

Hasta el año pasado en esta asignatura se estudiaban las `estructuras de datos`

- Mutables (estilo imperativo en ***Java***)
- Inmutables (estilo funcional en ***Haskell***)

Este año la visión funcional no se verá (no se estudia ***Haskell***)

- ***Java*** ha ido introduciendo las características de la programación funcional poco a poco
    - Ya se dispone de una versión muy completa en ***Java 22***

En cualquier caso, la programación funcional se verá en la asignatura `Programación Avanzada II`

Ventajas de la Programación Funcional

- Programas en un estilo conciso y claro
- El paralelismo es gratis
  - Los programas funcionales son intrínsicamente paralelos
    - Ya hemos mencionado que se verá en `Programación Avanzada II`
- Es posible realizar demostraciones sobre propiedades de los programas
  - No es testing ¡es matemáticas! 😃

Intención de esta clase

Mostrar cómo ha evolucionado el lenguaje ***Java*** para incorporar las características básicas de la programación funcional modelando:

- ***Comportamiento***: Lambdas, funciones de orden superior y composición funcional
- ***Datos***: Tipos algebraicos y concordancia de patrones

Ya desde ***Java 8*** (marzo 2014) se incluyó el modelado del comportamiento (***Presente***):

- Las utilizasteis el año pasado en *Programación Avanzada I*
  - para implementar abstracciones funcionales
  - para construir de forma flexible comparadores
  - en métodos de librerías (por ejemplo, `computeIfAbsent` de la interfaz `Map`)
- Análisis de datos (`Stream`)

Aquí hablaré de cómo ***Java*** ha modelado los datos funcionales: tipos algebraicos y concordancia de patrones (***Futuro***)

Índice

```
public final class PacoG {

    public int     tiposAlgebraicos() { return 6;}

    public PuntoF  tipoAlgebraicoPuntoPF() { var x = 9; return new PuntoF();}

    public PuntoJ  tipoAlgebraicoPuntoJava() { var x = 11; return new PuntoJ();}

    public int     comparativa(PuntoF p1, PuntoJ p2){ return 29;}

    public Stack   tipoAlgebraicoStack() { var x = 30; return new Stack();}

    public int     evaluacionNotacionPolacaInversa(Stack st) { return 36;}

    public boolean esConvenienteInmutabilidad() { var x = 40; return true;}

    public int     programacionOrientadaADatos() { return 41;}
}
```

Tipos algebraicos

- Introducidos en ***Hope*** (1970)
- Tipos compuestos formado por datos inmutables
- Se construyen por medio de ***tipos Sumas*** y ***tipos Productos*** posiblemente combinados
    - Aunque suene raro, esto ya lo habéis utilizado sin ponerle este nombre tan llamativo

Tipos algebraicos: Suma

Cuando los datos de un tipo se pueden expresar con diferentes alternativas o casos

- Los datos del tipo son la suma de todos los casos

```
enum Direccion {Norte, Sur, Este, Oeste}
```

- Podemos usar patrones en una expresión `switch` [1] para este tipo:

```
public Direccion gira(Direccion d) {
    return switch(d) {
        case Norte  ->  Este;
        case Este   ->  Sur;
        case Sur    ->  Oeste;
        case Oeste  ->  Norte;
    };
}
```

(1): las expresiones `switch` se introdujeron en ***Java 14***

Tipos algebraicos: Producto

Cuando declaramos una clase con variables de instancia, estamos contruyendo un tipo producto

```
class Persona {
    private String nombre;
    private int edad;
    ...
}
```

- Las instancias de esta clase son *asimilables* a los datos del producto `String x Int`

  Persona("juan", 17) $\equiv$ ("juan",17)

- Este *asimilable* se convierte en *equivalencia* cuando se trata de `datos inmutables`

Queda ver ***cómo combinar*** los tipos sumas con los tipos productos

Ejemplo `Punto` en un pseudo-lenguaje funcional 1/2

Queremos representar puntos del plano que puedan venir dados en forma cartesiana o en forma polar

- ***Cartesiana***: tendrán dos atributos, la abscisa y la ordenada (tipo producto)

```
Car(Double x, Double y)
```

- ***Polar***: tendrán dos atributos, el módulo y el ángulo (tipo producto)

```
Pol(Double modulo, Double angulo)
```

- Un punto puede ser de una forma o de otra (suma de tipos productos)

```
data Punto = Car(Double x, Double y) | Pol(Double modulo, Double angulo)
```

Nada más puede ser un `Punto`

Ejemplo `Punto` en un pseudo-lenguaje funcional 2/2

```
data Punto = Car(Double x, Double y) | Pol(Double modulo, Double angulo)
```

Vamos a crear una función que determine si un punto está situado en el primer cuadrante

- Se debe poder determinar si el punto viene dado de una forma o de otra (*concordancia de patrones*)

  > Descomponer el dato para ver su forma y contenido y actuar en consecuencia

```
primerCuadrante : Punto -> Bool
primerCuadrante(Car(x, y)) = x >= 0 && y >= 0
primerCuadrante(Pol(_, a)) = a >= 0 && a <= PI/2

p1 = Car(3,5)
p2 = Pol(3,PI)

primerCuadrante(p1)     -- True
primerCuadrante(p2)     -- False
```

Objetivo: `Punto` en *Java*

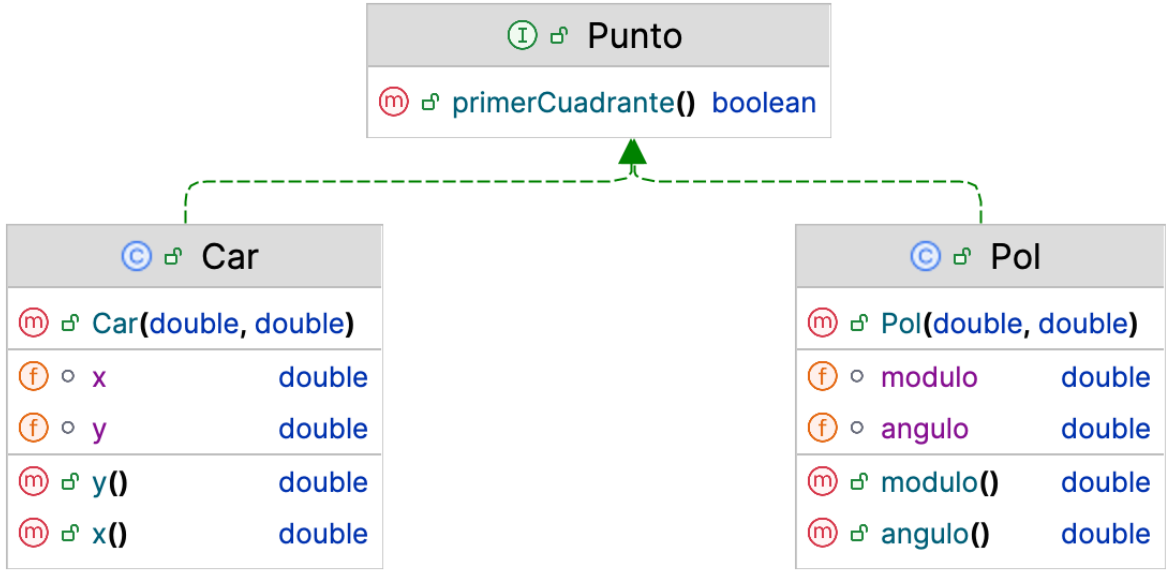
- Construir el tipo algebraico `Punto` en ***Java***
- Definir el método `primerCuadrante` usando concordancia de patrones

Lo haremos avanzado sobre la versiones de ***Java*** para mostrar cómo se han introducido poco a poco los conceptos necesarios:

1. ***Java*** anterior a 16,
2. ***Java*** 16,
3. ***Java*** 17,
4. ***Java*** 21 y
5. ***Java*** 22

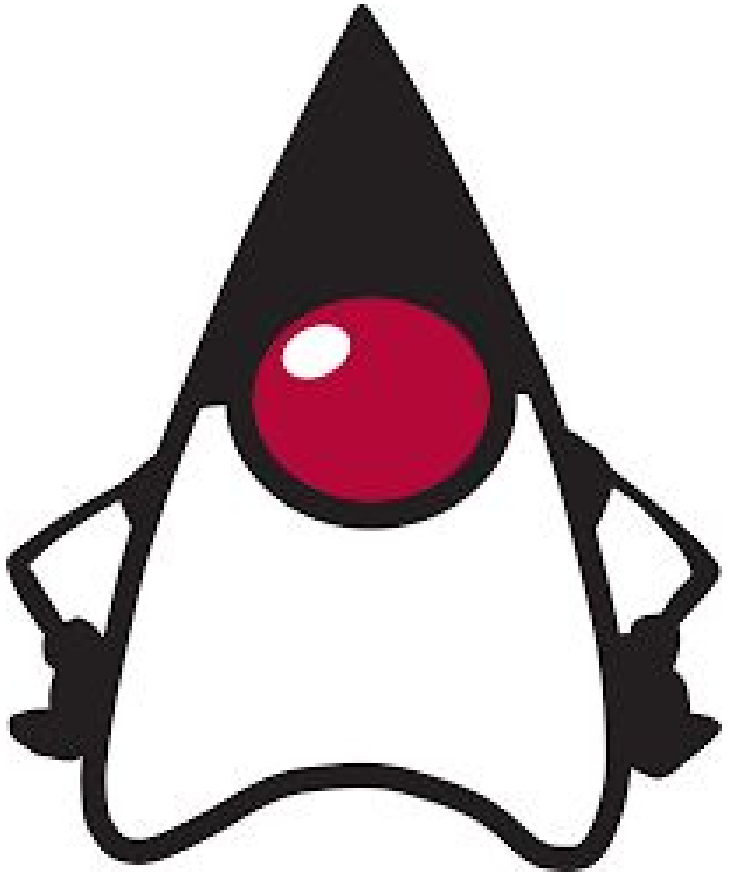
Planteamiento del problema. `Punto` en *Java*

- Crearemos una interfaz `Punto`
  - Le añadiremos el método por defecto `primerCuadrante`
- Crearemos una clase inmutable `Car` que implemente `Punto` para describir un punto en coordenadas cartesianas
- Crearemos una clase inmutable `Pol` que implemente `Punto` para describir un punto en coordenadas polares



Java anterior a la versión 16

No se ha tomado ninguna medida para incluir tipos algebraicos y concordancia de patrones



Implementación de un `Punto` en forma cartesiana

```
public final class Car implements Punto {
    final private double x, y;
    public Car(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double x() {
        return x;
    }
    public double y() {
        return y;
    }
}
```

Implementación de un `Punto` en forma polar

```
public final class Pol implements Punto {
    private final double modulo, angulo;
    public Pol(double m, double a) {
        modulo = m;
        angulo = a;
    }
    public double modulo() {
        return modulo;
    }
    public double angulo() {
        return angulo;
    }
}
```

Implementación de la interfaz `Punto`

`primerCuadrante` lo implementamos como método por defecto para separar datos y código

```
public interface Punto {
    default boolean primerCuadrante() {
        boolean res = false;
        if (this instanceof Car) {
            Car c = (Car) this;
            res = c.x() >= 0 && c.y() >= 0;
        } else if (this instanceof Pol) {
            Pol p = (Pol) this;
            res = p.angulo() >= 0 && p.angulo() <= Math.PI/2;
        }
        return res;
    }
}
```

Ejemplo de uso de `Punto`

```
public class Main {
    public static void main(String[] args) {
        Punto p1 = new Car(3.0, 5.0);
        Punto p2 = new Pol(3.0, Math.PI);
        System.out.println(p1.primerCuadrante());
        System.out.println(p2.primerCuadrante());
    }
}

// true
// false
```

Java 16 (Marzo 2021)

- Registros
- Concordancia de patrones simples en `instanceof`

Record 1/2

Java 16 Introduce:

- Registros `record` para describir cómoda y rápidamente clases inmutables
  - Clases simples inmutables con constructor, métodos de acceso a sus variables, `equals`, `hashCode` y `toString` implícitos
- Una forma limitada de concordancia de patrones en `instanceof` con *variable ligada*

Record 2/2

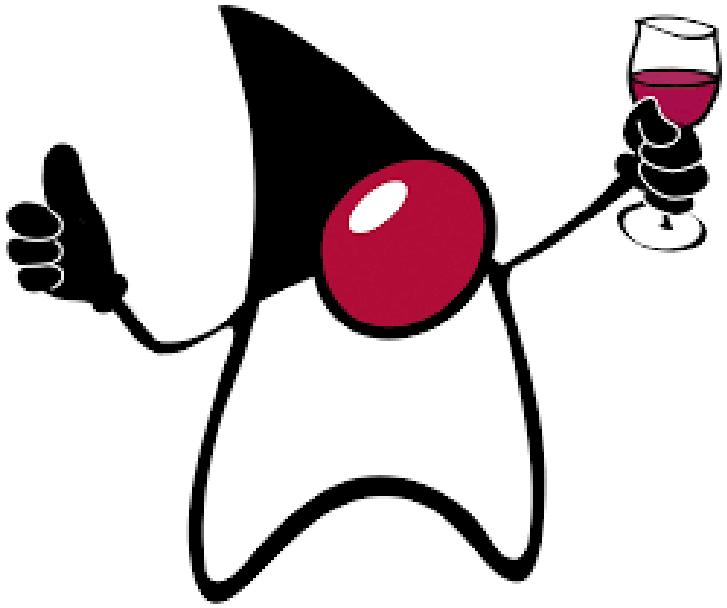
```java
public record Car(double x, double y) implements Punto{}
public record Pol(double modulo, double angulo) implements Punto{}

public interface Punto {
    default boolean primerCuadrante() {
        boolean res = false;
        if (this instanceof Car c)          // c es la variable ligada
            res = c.x() >= 0 && c.y() >= 0;
        else if (this instanceof Pol p)     // p es la variable ligada
            res = p.angulo() >= 0 && p.angulo() <= Math.PI/2;
        return res;
    }
}
```

tipo algebraico incompleto

- los registros son equivalentes a tipos productos ✅
- la interfaz es equivalente a la suma de tipos productos (pero suma abierta) ❌
    - Cualquier clase puede implementar la interfaz `Punto` 😬

Java 17 (Septiembre 2021)

- Clases e interfaces selladas



Restricciones a la herencia e implementación de interfaces

La herencia es, o bien universal, o está prohibida

- Si una clase se declara `final`, nadie puede heredar de ella (***herencia prohibida***)
- Si no se declara `final`, cualquiera puede heredar de ella (***herencia universal***)

La implementación de interfaces es universal

- Cualquier clase puede implementar una interfaz (***implementación universal***)

Java 17 introduce el concepto de **herencia selectiva**

- Una clase puede declarar ***quién*** va a heredar de ella
- Una interfaz puede declarar ***quién*** la debe implementar

La interfaz `Punto` sellada

```
public record Car(double x, double y) implements Punto{}
public record Pol(double modulo, double angulo) implements Punto{}

public sealed interface Punto permits Car, Pol {
    default boolean primerCuadrante() {
        boolean res = false;
        if (this instanceof Car c)
            res = c.x() >= 0 && c.y() >= 0;
        else if (this instanceof Pol p)
            res = p.angulo() >= 0 && p.angulo() <= Math.PI / 2;
        return res;
    }
}
```

tipo algebraico

- Ya se ha cerrado la suma reflejando todas las alternativas ✅

Solo queda mejorar la concordancia de patrones

Java 21 (Septiembre 2023)

- Concodancia de patrones en `instanceof`
- Corcondacia de patrones en `switch`

Concordancia de patrones con `instanceof`

Permite descomponer ***los registros*** según su constructor

- Podemos usarlo en `instanceof` en lugar de la clase y la variable ligada

```
public record Car(double x, double y) implements Punto{}
public record Pol(double modulo, double angulo) implements Punto{}

public sealed interface Punto permits Car, Pol {
    default boolean primerCuadrante() {
        boolean res = false;
        if (this instanceof Car(double x, double y))
            res = x >= 0 && y >= 0;
        else if (this instanceof Pol(double m, double a))
            res = a >= 0 && a <= Math.PI/2;
        return res;
    }
}
```

`instanceof` no refleja claramente que el tipo algebraico es completo

Concordancia de patrones con `switch`

Permite descomponer los registros según su constructor

- Podemos usarlo en los `case` de la sentencia y la expresión `switch`

```
public record Car(double x, double y) implements Punto{}
public record Pol(double modulo, double angulo) implements Punto{}

public sealed interface Punto permits Car, Pol {
    default boolean primerCuadrante() {
        return switch(this) {
            case Car(double x, double y) -> x >= 0 && y >= 0;
            case Pol(double m, double a) -> a >= 0 && a <= Math.PI/2;
        };
    }
}
```

- Como se ve, `switch` no necesita cláusula `default` gracias a que `this` es `Punto` que es una interfaz sellada (¡no puede haber otra alternativa!) 😃

Java 22 (Marzo 2024)

- Variables anónimas

Variables anónimas

Las variables que no se utilizan pueden sustituirse por el símbolo `_`

```
public record Car(double x, double y) implements Punto{}
public record Pol(double modulo, double angulo) implements Punto{}

public sealed interface Punto permits Car, Pol {
    default boolean primerCuadrante() {
        return switch(this) {
            case Car(double x, double y) -> x >= 0 && y >= 0;
            case Pol(_       , double a) -> a >= 0 && a <= Math.PI/2;
        };
    }
}
```

Con todo esto se consigue implementar la concordancia de patrones 👏

Comparativa pseudo-lenguaje funcional con *Java*

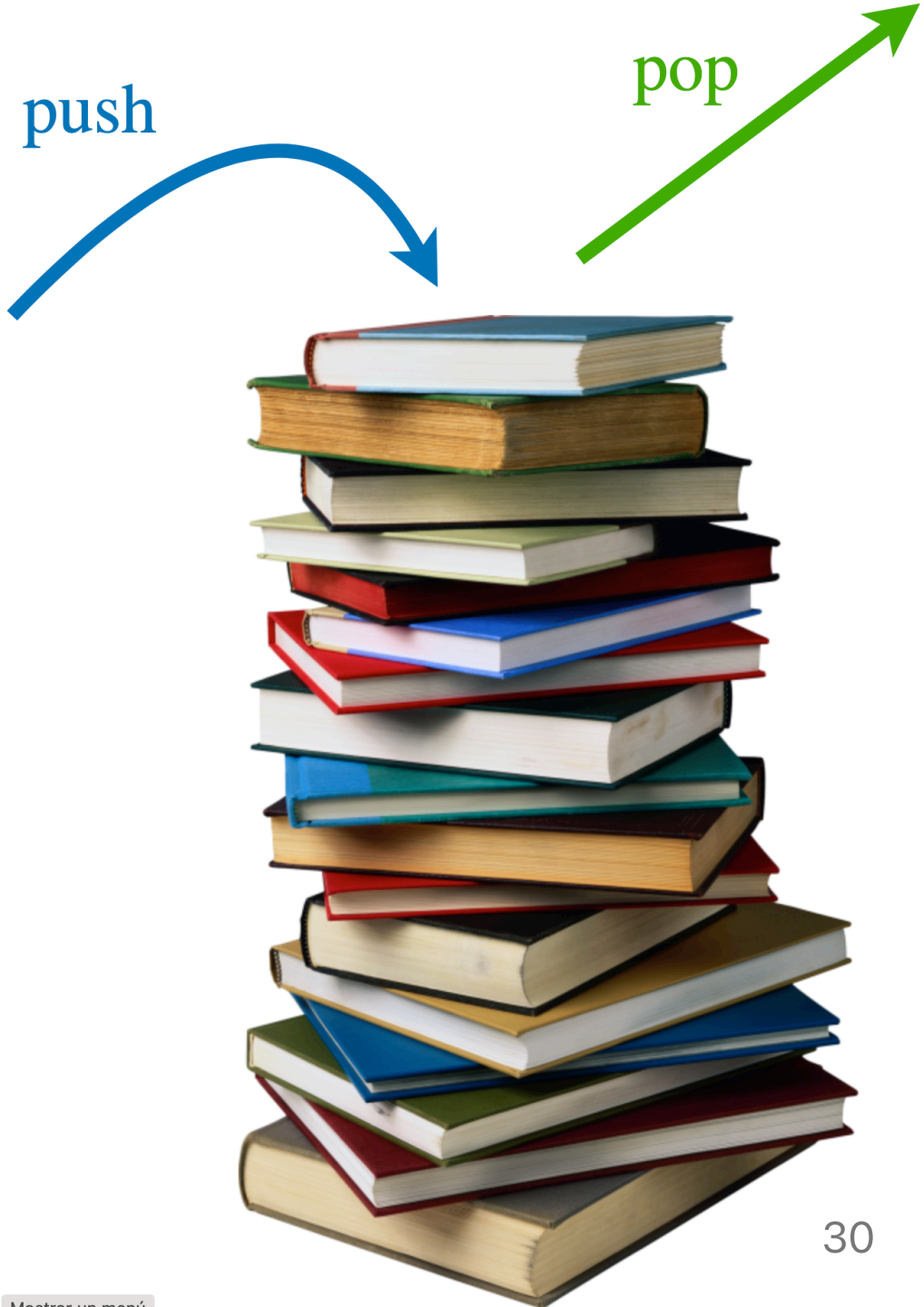
```
data Punto = Car(Double x, Double y) | Pol(Double modulo, Double angulo)

primerCuadrante : Punto -> Bool
primerCuadrante(Car(x, y)) = x >= 0 && y >= 0
primerCuadrante(Pol(_, a)) = a >= 0 && a <= PI/2
```

```
public record Car(double x, double y) implements Punto{}
public record Pol(double modulo, double angulo) implements Punto{}

public sealed interface Punto permits Car, Pol {
    default boolean primerCuadrante() {
        return switch(this) {
            case Car(double x, double y) -> x >= 0 && y >= 0;
            case Pol(_       , double a) -> a >= 0 && a <= Math.PI/2;
        };
    }
}
```

Resultan implementaciones muy similares 😍

El tipo abstracto de datos `Stack`

- Un **stack** (pila) es una colección que almacena elementos de manera que el último en entrar es el primero en salir (*LIFO*)
- **Operaciones**:
  - `push` : Añade un elemento en la *cima* del stack
  - `pop` : Elimina la *cima* del stack
  - `top` : Devuelve el elemento en la *cima* del stack sin eliminarlo
  - `isEmpty` : Chequea si el stack está vacío



Propiedades de los `Stack`

Propiedad 1: $\forall x \in \mathcal{T}, \forall s \in \mathcal{S}tack$ `s.push(x).top() == x`

Propiedad 2: $\forall x \in \mathcal{T}, \forall s \in \mathcal{S}tack$ `s.push(x).pop() == s`

Propiedad 3: `(new Empty()).isEmpty()`

Propiedad 4: $\forall x \in \mathcal{T}, \forall s \in \mathcal{S}tack$ `!s.push(x).isEmpty()`

Aplicaciones de los stacks

- **Llamada de funciones**: Para almacenar información de la llamada a una función (parámetros, resultados, dirección de retorno, etc.)
- **Evaluación de expresiones**: Para evaluar expresiones aritméticas (*Dijkstra's two-stack algorithm, RPN*)
- **Búsqueda en profundidad**: Para almacenar los nodos a visitar en algoritmos de búsquedas en profundidad
- **Mecanismo undo**: Para almacenar la historia de operaciones que pueden deshacerse
- **Análisis sintáctico**: Para hacer el análisis sintáctico en un programa
- **Expresar recursión**: Para simular recursión cuando el lenguaje no lo soporte

El `Stack` como tipo algebraico

- No la hacemos genérica por simplificación
  - Vamos a implementar una pila de enteros
- Una pila puede estar vacía o tener un elemento en la cima seguido de otra pila
- Será un tipo algebraico recursivo

```
data Stack = Empty | Node(Int top, Stack rest)

p1 = Empty
p2 = Node(4, Node (2, Empty))
p3 = Node(5, p2)

isEmpty : Stack -> Bool
isEmpty(Empty)      = True
isEmpty(Node(_,_)) = False

pop : Stack -> Stack
pop(Empty)          = error("empty stack")
pop(Node(_,stack)) = stack

push : (Int,Stack) -> Stack
push(x,stack) = Node(x,stack)

top   : Stack -> Int
top(Empty)      = error("empty stack")
top(Node(x,_)) = x
```

`Stack` en *Java* 1/2

```
public record Empty() implements Stack{}
public record Node(int top, Stack rest) implements Stack{}

public sealed interface Stack permits Empty, Node {
    default boolean isEmpty() {
        return switch(this) {
            case Empty()   -> true;
            case Node(_,_) -> false;
        };
    }

    default Stack push(int elem) {
        return new Node(elem, this);
    }
    ...
}
```

`Stack` en *Java* 2/2

```
public sealed interface Stack permits Empty, Node {
    ...
    default int top() {
        return switch(this) {
            case Empty() -> throw new NoSuchElementException("Empty Stack");
            case Node(int top, _) -> top;
        };
    }

    default Stack pop() {
        return switch(this) {
            case Empty() -> throw new NoSuchElementException("Empty Stack");
            case Node(_, Stack stack) -> stack;
        };
    }
}
```

Las propiedades de los stack ¡***pueden demostrarse*** con boli y papel! 😳

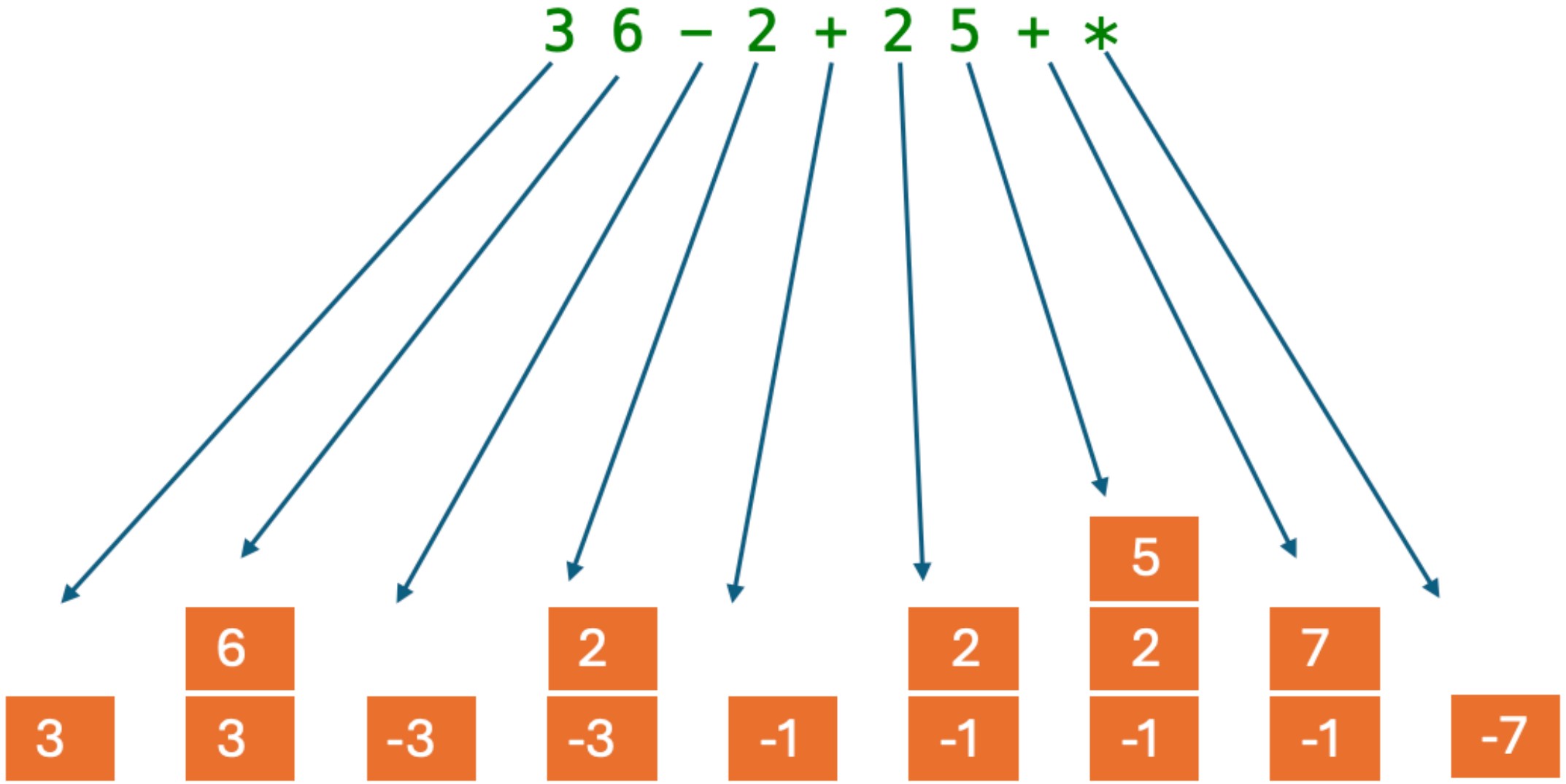
Uso de `Stack`. Notación Polaca Inversa (RPN)

- Evaluar expresiones como `((3-6)+2)*(2+5)` requiere análisis sintáctico
- Evaluar la expresión equivalente en polaca inversa `3 6 - 2 + 2 5 + *` no lo necesita
    - Solo necesita un `Stack`

Recorremos la expresión de izquierda a derecha:

- Si encontramos un número lo introducimos en el stack
- Si encontramos un operador
    - Sacamos la cima del stack (segundo argumento)
    - Sacamos la nueva cima del stack (primer argumento)
    - Aplicamos el operador con estos argumentos
    - Introducimos el resultado en el stack
- Al finalizar, la cima del stack tendrá el resultado

Evaluación en RPN



La clase `RPN` 1/2

```
public class RPN {
    private Stack stack;
    private RPN() {
        stack = new Empty();
    }
    private void data(int elem) {
        stack = stack.push(elem);
    }
    private void operate(BiFunction<Integer, Integer, Integer> operator) {
        int a2 = stack.top();                // Segundo argumento
        stack = stack.pop();
        int a1 = stack.top();                // Primer argumento
        stack = stack.pop();
        int res = operator.apply(a1, a2);    // Se aplica la función
        stack = stack.push(res);             // Se introduce el resultado
    }
    private int result() {
        return stack.top();
    }
    ...
}
```

La clase `RPN` 2/2

```
public class RPN {
    ...
    public static int execute(String expresion) {
        RPN rpn = new RPN();
        for(String e : expresion.split(" ")) {
            switch(e) {
                case "+": rpn.operate((x,y)->x+y); break;
                case "-": rpn.operate((x,y)->x-y); break;
                case "*": rpn.operate((x,y)->x*y); break;
                default : rpn.data(Integer.parseInt(e));
            }
        }
        return rpn.result();
    }
}

public class Main {
    public static void main(String[] args) {
        // ((3-6)+2)*(2+5)
        // 3 6 - 2 + 2 5 + *
        System.out.println(RPN.execute("3 6 - 2 + 2 5 + *")); // -7
    }
}
```

¿Es conveniente escribir clases inmutables?

```
Effective Java
Joshua Bloch
Third edition. 2018
Addison-Wesley
```

Proporciona 90 recomendaciones para escribir buen código en ***Java***

- ***Item 17: Minimize mutability***

  > Las clases deben ser inmutables a menos que haya una razón muy buena para hacerlas mutables ... Si una clase no puede ser de hecho inmutable, se debería limitar su mutabilidad tanto como sea posible.

Programación Orientada a Datos

> Estilo de programación que modela los datos como datos inmutables y los mantiene separados del código que incorpora la lógica de la aplicación

- Los tipos algebraicos y la concordancia de patrones (los registros, los tipos sellados y la concordancia de patrones en ***Java***) funcionan conjuntamente para facilitar la *programación orientada a datos* en ***Java***
- Programación Orientada a Objetos
  - es adecuada para modelar grandes sistemas complejos
- Programación Orientada a Datos
  - es adecuada para (sub)sistemas más sencillos o pequeños

Ambos estilos ***son compatibles y pueden utilizarse conjuntamente*** en cualquier proyecto software 😜

Agradecimientos

José Enrique Gallardo Ruíz

Pablo López Olivas

Muchas gracias por la atención