

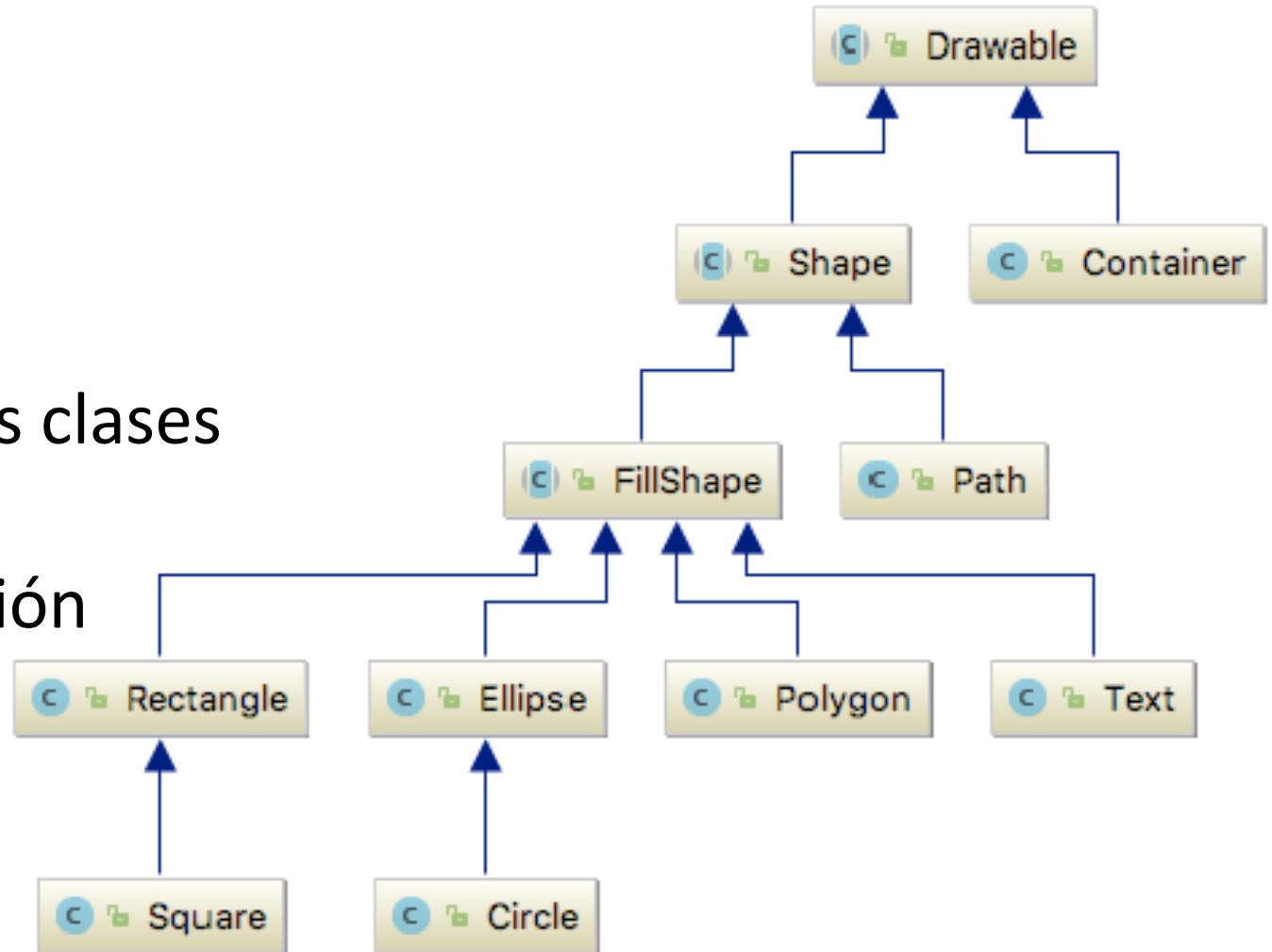
# Herencia e Interfaces

# Contenido

- Herencia
- Constructores y Herencia
- Polimorfismo
- Vinculación dinámica
- Clases abstractas
- Interfaces
  - Lambdas
- Abstracción funcional
- Datos algebraicos

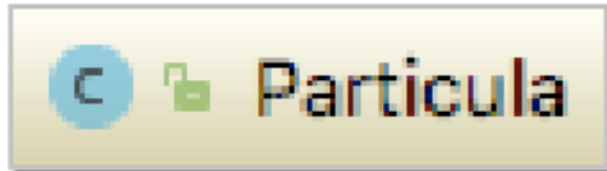
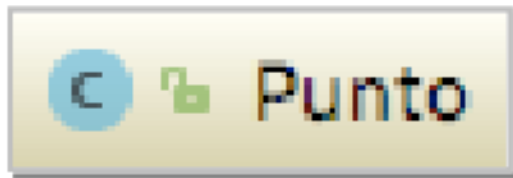
# Herencia

- Nueva posibilidad para **reutilizar** código
- Algo más que:
  - incluir ficheros, o
  - importar módulos
- Permite clasificar las clases en una jerarquía
- Responde a la relación **“es un”**



# Herencia

Padres / Ascendientes /  
Superclase



Hijos / Descendientes /  
Subclase

• Una subclase **dispone** de las **variables** y **métodos** de la superclase, y **puede añadir** otros nuevos.

La subclase puede **modificar** el comportamiento heredado (por ejemplo, redefiniendo algún método heredado) .

La herencia es transitiva.

Los objetos de una clase que hereda de otra **pueden verse** como objetos de esta última.

# Subclases

- En Java se pueden definir *subclases* o clases que *heredan* estado y comportamiento de otra clase (la *superclase*) a la que amplían, en la forma:

```
class MiClase extends Superclase {  
    . . .  
}
```

- En Java sólo se permite *herencia simple*, por lo que pueden establecerse jerarquías de clases.
- Todas las jerarquías confluyen en la clase **Object** de **java.lang** que recoge los comportamientos básicos que debe presentar cualquier clase.

# La clase Particula

```
public class Particula extends Punto {  
    private double masa;  
    final static double  $G = 6.67e-11$ ;  
  
    ...  
  
    public void masa(double m) { masa = m; }  
    public double masa() { return masa; }  
    public double atraccion(Particula part) {  
        double d = this.distancia(part);  
        return  $G * masa * part.masa() / (d * d)$ ;  
    }  
}
```

# Herencia y constructores

- Los constructores **no** se heredan.
- Cuando se define un constructor en herencia se debe proceder de alguna de las tres formas siguientes:

- Invocar a un constructor de la misma clase (con distintos argumentos) mediante this:

- Por ejemplo:

```
public Punto() {  
    this(0, 0);  
}
```

La llamada a this debe estar en la primera línea

- Invocar algún constructor de la superclase mediante super:

- Por ejemplo:

```
public Particula(double a, double b, double m) {  
    super(a, b);  
    masa = m;  
}
```

La llamada a super debe estar en la primera línea

- De no ser así, se invoca por defecto al constructor sin argumentos de la superclase.

- Por ejemplo:

```
public Particula() {  
    // Se invoca el constructor por defecto Punto()  
    masa = 0;  
}
```

# La clase Particula

```
public class Particula extends Punto {  
    private double masa;  
    final static double G = 6.67e-11;
```

```
    public Particula(double m) {  
        this(0, 0, m);  
    }
```

Se refiere a  
Particula(double, double, double)

```
    public Particula(double a, double b, double m) {  
        super(a, b);  
        masa = m;  
    }
```

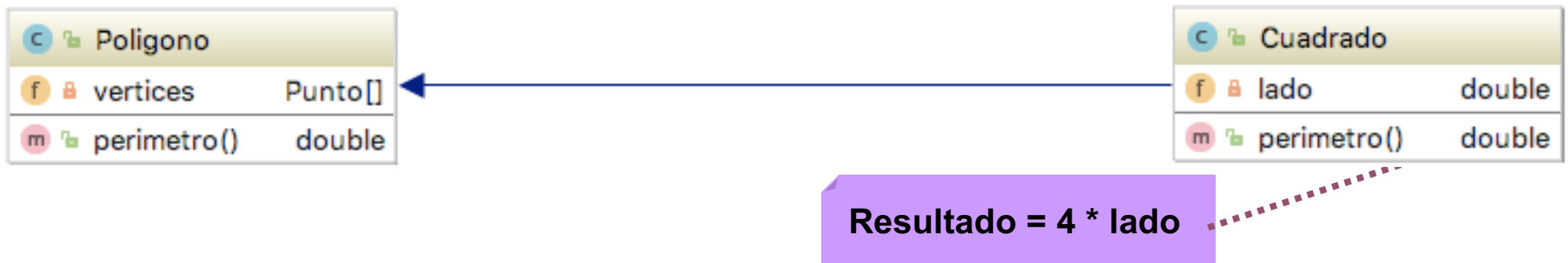
Se refiere a  
Punto(double, double)

```
    public void masa(double m) { masa = m; }  
    public double masa() { return masa; }  
    public double atraccion(Particula part) {  
        double d = this.distancia(part);  
        return G * masa * part.masa() / (d * d);  
    }  
}
```



# Redefinición del comportamiento

- Es muy corriente la redefinición de un método en la subclase.



- Es conveniente utilizar la anotación **@Override** para indicar que el método está sobrescrito.
- La redefinición puede impedirse mediante el uso del cualificador **final**.

# Herencia vs. composición

- Mientras que la herencia establece una relación de tipo *“es-un”*, la composición responde a una relación de tipo *“tiene”* o *“está compuesto por”*.
- Así, por ejemplo, una partícula **es un** punto (con masa), mientras que un segmento **tiene** dos puntos (origen y extremo)

## Herencia

Particula		
f	masa	double
f	G	double
m	Particula(double)	
m	Particula(double, double, double)	
m	masa(double)	void
m	masa()	double
m	atraccion(Particula)	double
m	toString()	String



Punto		
f	x	double
f	y	double
m	Punto()	
m	Punto(double, double)	
m	abscisa()	double
m	ordenada()	double
m	abscisa(double)	void
m	ordenada(double)	void
m	trasladar(double, double)	void
m	distancia(Punto)	double
m	toString()	String

## Composición







Segmento		
f	origen	Punto
f	extremo	Punto
m	Segmento(Punto, Punto)	
m	trasladar(double, double)	void
m	longitud()	double



# Control de la visibilidad

Existen cuatro niveles de visibilidad:

- **private** – visibilidad dentro de la propia clase
- **protected** – visibilidad dentro del paquete y de las clases herederas
- **public** – visibilidad desde cualquier paquete
- Por omisión – visibilidad dentro del propio paquete (package)

			Mismo paquete		Otro paquete	
			Subclase	Otra	Subclase	Otra
	-	<b>private</b>	NO	NO	NO	NO
	#	<b>protected</b>	SÍ	SÍ	SÍ	NO
	+	<b>public</b>	SÍ	SÍ	SÍ	SÍ
	~	<b>package</b>	SÍ	SÍ	NO	NO

# Polimorfismo sobre los datos

- Un lenguaje tiene **capacidad polimórfica** sobre los datos cuando
  - una variable declarada de un tipo (o clase) –*tipo estático*– determinado
  - puede hacer referencia en tiempo de ejecución a valores (objetos) de tipo (clase) distinto –*tipo dinámico*–.
- La capacidad polimórfica de un lenguaje no suele ser ilimitada, y en los LOOs está habitualmente restringida por la relación de herencia:
  - El *tipo dinámico* debe ser **descendiente** del *tipo estático*.

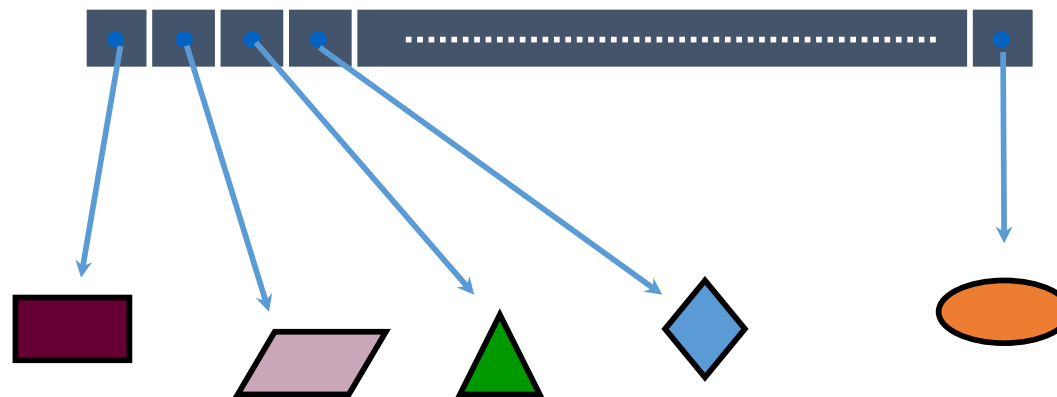
```
Punto pto = new Particula(3, 5, 22);
```

Tipo estático  
de pto

Tipo dinámico  
de pto

# Polimorfismo sobre los datos

- Una variable puede referirse a objetos de clases distintas de la que se ha declarado. Esto afecta a:
  - asignaciones explícitas entre objetos,
  - paso de parámetros,
  - devolución del resultado en una función.
- La restricción dada por la herencia permite construir estructuras con elementos de naturaleza distinta, pero con un comportamiento común:



# Vinculación dinámica

- La **vinculación dinámica** resulta el complemento indispensable del polimorfismo sobre los datos, y consiste en que:
  - La **invocación del método** que ha de resolver un mensaje **se retrasa al tiempo de ejecución**, y se hace depender del **tipo dinámico** del objeto receptor.

C	Poligono
f	vertices Punto[]
m	perimetro() double

Suma de distancias  
entre vertices

C	Cuadrado
f	lado double
m	perimetro() double

```
@Override
public double perimetro() {
    return lado*4;
}
```

Tipo estático de p

Poligono p = new Cuadrado(...);  
p.perimetro();

??

Tipo dinámico de p

# Herencia, variables y métodos

- Métodos de instancia:

- Un método de instancia de una clase puede redefinirse en una subclase.
  - Salvo si el método está declarado como `final` (o la clase).
- La **resolución de los métodos de instancia** se **realiza por vinculación dinámica**.
- Una redefinición puede ampliar la visibilidad de un método.
  - La anotación `@Override` asegura que es una redefinición.
- El método redefinido queda oculto en la subclase por el nuevo método.
  - Si se desea acceder al redefinido, se debe utilizar la sintaxis

`super.<nombre del método>(<argumentos>)`

- La resolución de una llamada a `super` se hace comenzando en la clase padre de la que aparece la palabra `super`.
- Métodos de clase y variables de instancia o de clase:
  - Se **resuelven por vinculación estática**.

# Herencia, y resolución del método a ejecutar

Dos fases:

- Compilación: **Atiende al tipo estático.**
  - El tipo estático tiene que ser capaz de responder al mensaje con un método suyo o de sus clases superiores.
  - Si no es así se produce un error de compilación
- Ejecución: **Atiende al tipo dinámico.**
  - El método a ejecutar comienza a buscarse en la clase del tipo dinámico y se sigue buscando de forma ascendente por las clases superiores.



## Compilación y vinculación dinámica

```
Punto pto = new Particula(3, 5, 22);
```

Tipo estático  
de pto

Tipo dinámico  
de pto

```
pto.trasladar(4, 6);
```

- **Compila** porque el tipo estático **sabe** responder a ese mensaje.
- Al ejecutar **se busca** en el **tipo dinámico**. Si no se encuentra, se sube por la herencia hasta encontrarlo.
  - **Es seguro** que se encuentra porque ha compilado

```
pto.atraccion(new Particula(3, 4, 6));
```

- **No compila** porque el tipo estático **no sabe** responder a ese mensaje.

## Prohibiendo subclases

- Por razones de seguridad o de diseño, se puede prohibir la definición de subclases para una clase etiquetándola con **final**.
  - Recordad que una subclase puede sustituir a su superclase donde ésta sea necesaria y tener comportamientos muy distintos
- El compilador rechazará cualquier intento de definir una subclase para una clase etiquetada con **final**.
- También se pueden etiquetar con **final**:
  - métodos, para evitar su redefinición en alguna posible subclase, y
  - variables, para mantener constantes sus valores o referencias.

# Clases abstractas

- Clases de la que **no se pueden** crear instancias
  - Pueden declarar métodos sin implementar
    - Métodos abstractos
  - Las subclases están obligadas a implementarlas
- Se pueden declarar variables cuyo tipo estático sea una clase abstracta que puedan referirse a objetos de diversas clases descendientes

# Clases abstractas

- Las clases abstractas definen un protocolo común en una jerarquía de clases.
- Obligan a sus subclases a implementar los métodos que se declararon como abstractos.
  - De lo contrario, esas subclases se siguen considerando abstractas.
- En Java, además de clases abstractas se pueden definir *interfaces* (que se pueden considerar clases “completamente” abstractas).

# Clases abstractas

```
abstract public class Poligono {  
    private Punto vertices[];  
    public void trasladar(double a, double b){  
        for (int i = 0; i < vertices.length; i++)  
            vertices[i].trasladar(a, b);  
    }  
    public double perimetro() {  
        double per = 0;  
        for (int i = 1; i < vertices.length; i++)  
            per = per + vertices[i - 1].distancia(vertices[i]);  
        return per  
            + vertices[0].distancia(vertices[vertices.length-1]);  
    }  
    abstract public double area();  
}
```

**MÉTODO ABSTRACTO**

```
Poligono pol = new Poligono() ;
```

# Solución 1. Clases abstractas: Poligono

```
public abstract class Poligono {
    protected Punto[] vert;

    public Poligono(Punto[] vs) {
        vert = vs;
    }
    public void trasladar(double dx, double dy) {
        for (Punto pto : vert) {
            pto.trasladar(dx, dy);
        }
    }
    public double perimetro() {
        Punto ant = vert[vert.length - 1];
        double res = 0;
        for (Punto pto : vert) {
            res += pto.distancia(ant);
            ant = pto;
        }
        return res;
    }
}
```

```
public class Rectangulo extends Poligono {

    public Rectangulo(...) {...}
    public double area() {
        return base() * altura();
    }
    public double base() {
        return vert[0].distancia(vert[1]);
    }
    public double altura() {
        return vert[1].distancia(vert[2]);
    }

    public String toString() {...}
}
```

```
public class Cuadrado extends Poligono {

    public Cuadrado(...) {...}
    public double area() {
        return lado() * lado();
    }
    public double lado() {
        return vert[0].distancia(vert[1]);
    }
    public String toString() {...}
}
```

```
; // No sabemos calcularla
```

## Solución 1. Clases abstractas: Punto

```
public abstract class Punto {  
    abstract public double x();  
    abstract public double y();  
    abstract public double modulo();  
    abstract public double angulo();  
    public double distancia(Punto pto) {  
        return Math.sqrt(Math.pow(x()-pto.x(),2) +  
            Math.pow(y()-pto.y(),2));  
    }  
}
```

```
public record Cartesiano(double x, double y)  
    extends Punto {  
    public double modulo() {  
        return Math.sqrt(Math.pow(x(),2)+  
            Math.pow(y(),2));  
    }  
    public double angulo() {  
        return Math.atan2(y(),x());  
    }  
}
```

```
public record Polar(double modulo, double angulo)  
    extends Punto {  
    public double x() {  
        return modulo()*Math.cos(angulo());  
    }  
    public double y() {  
        return modulo()*Math.sin(angulo());  
    }  
}
```

## Solución 1. Uso de puntos

```
public class MainPuntos {  
    public static void main(String[] args) {  
        Punto p1 = new Cartesiano(5,4);  
        Punto p2 = new Polar(6,Math.PI/6);  
        System.out.println(p1.modulo());  
        System.out.println(p2.x());  
        System.out.println(p1.distancia(p2));  
    }  
}
```



# Interfaces

- Define un **protocolo de comportamiento**, es decir un **contrato** que las clases deberán respetar.
  - Las clases pueden implementar la interfaz respetando el contrato.
  - Se utilizarán cuando se demande el contrato.
- Una interfaz **sólo puede ser extendida** por otras interfaces.
- Una interfaz **puede heredar** de varias interfaces.
- Una clase **puede implementar** varias interfaces.

# Definición de interfaces

- En una interfaz sólo se permiten constantes, métodos abstractos y **métodos por defecto**.

```
public static final package, en caso de omisión
public interface miInterfaz
    extends interfaz1, interfaz2 {
    String CAD1 = "SUN";
    String CAD2 = "PC";
    void valorCambiado(String producto, int val);
    default ... // método por defecto
    ...
}
```

Diagram illustrating the definition of an interface in Java. The code is annotated with boxes and arrows:

- public static final**: Boxed in green, with an arrow pointing to the `public` keyword.
- package, en caso de omisión**: Boxed in green, with an arrow pointing to the `package` keyword.
- public abstract**: Boxed in green, with an arrow pointing to the `public` keyword.

# Implementación de interfaces

- Cuando una clase implementa una interfaz,
  - **Se adhiere al contrato** definido en la interfaz y en sus superinterfaces,
  - **Hereda todas las constantes** definidas en la jerarquía,
- *Adherirse al contrato quiere decir que debe implementar todos los métodos abstractos*  
(salvo que sea una clase que se quiera mantener abstracta en cuyo caso, los métodos no implementados aparecerán como **abstract**).
- *Si una clase redefine un método por defecto se usará el redefinido. En otro caso se utilizará el definido en la interfaz.*

```
public class MiClase  
    extends Superclase1  
    implements Interfaz1, Interfaz2, ... {  
  
}
```

## Solución 2. Interfaz Punto

```
public interface Punto {  
    double x();  
    double y();  
    double modulo();  
    double angulo();  
    double distancia(Punto pto);  
}
```

## Solución 2. Punto en coordenadas cartesianas

```
public record Cartesiano(double x, double y) implements Punto {  
    public double modulo() {  
        return Math.sqrt(Math.pow(x(),2)+Math.pow(y(),2));  
    }  
  
    public double angulo() {  
        return Math.atan2(y(),x());  
    }  
    public double distancia(Punto pto) {  
        return  
            Math.sqrt(Math.pow(x()-pto.x(),2)+ Math.pow(y()-pto.y(),2));  
    }  
}
```

## Solución 2. Punto en coordenadas polares

```
public record Polar(double modulo, double angulo) implements Punto {  
    public double x() {  
        return modulo()*Math.cos(angulo());  
    }  
  
    public double y() {  
        return modulo()*Math.sin(angulo());  
    }  
  
    public double distancia(Punto pto) {  
        return  
            Math.sqrt(Math.pow(x()-pto.x(),2)+ Math.pow(y()-pto.y(),2));  
    }  
}
```

## Solución 2. Uso de puntos

```
public class MainPuntos {  
    public static void main(String[] args) {  
        Punto p1 = new Cartesiano(5,4);  
        Punto p2 = new Polar(6,Math.PI/6);  
        System.out.println(p1.modulo());  
        System.out.println(p2.x());  
        System.out.println(p1.distancia(p2));  
    }  
}
```

# Métodos por defecto y de clase en interfaces

- Las interfaces pueden incorporar:
  - Métodos por defecto.
    - Se declaran con **default**
    - Métodos con cuerpo.
    - Se utilizará si la clase no lo implementa.
  - Métodos de clase.
    - Se declaran con **static**
    - Métodos con cuerpo.
    - Podrá usarse con el nombre de la clase o de la interfaz.



## Solución 3: Métodos por defecto para la interfaz Punto y los registros Cartesiano y Polar

```
public record Polar(double modulo, double angulo) implements Punto {}
public record Cartesiano(double x, double y) implements Punto {}

public interface Punto {
    default double x() {
        return modulo()*Math.cos(angulo());
    }
    default double y() {
        return modulo()*Math.sin(angulo());
    }
    default double modulo() {
        return Math.sqrt(Math.pow(x(),2)+Math.pow(y(),2));
    }
    default double angulo() {
        return Math.atan2(y(),x());
    }
    default double distancia(Punto pto) {
        return Math.sqrt(Math.pow(x()-pto.x(),2) + Math.pow(y()-pto.y(),2));
    }
}
```

## Solución 3: Métodos static para la interfaz Punto

```
public record Polar(double modulo, double angulo) implements Punto {}
public record Cartesiano(double x, double y) implements Punto {}
public interface Punto {
    default double x() {
        return modulo()*Math.cos(angulo());
    }
    default double y() {
        return modulo()*Math.sin(angulo());
    }
    default double modulo() {
        return Math.sqrt(Math.pow(x(),2)+Math.pow(y(),2));
    }
    default double angulo() {
        return Math.atan2(y(),x());
    }
    default double distancia(Punto pto) {
        return Math.sqrt(Math.pow(x()-pto.x(),2) + Math.pow(y()-pto.y(),2));
    }
    static Punto origen() {
        return new Cartesiano(0,0);
    }
}
```

# Listas

- Alternativa al array mucho más fácil de manipular:
  - Array
    - Hay que crearlo de un tamaño dado. Vigilar siempre si hay espacio para añadir un elemento más.
    - Hay que llevar un contador para saber cuántos elementos tenemos.
    - Si queremos insertar o borrar un elemento hay que mover todos desde esa posición hasta el final.
  - Las listas hacen todo eso por nosotros

# Listas

- Para declarar una variable que referencia a una lista hacemos
  - `List<String> lista;`
  - `List` es una interfaz
    - Entre `<>` indicamos qué datos guardar (clase o interfaz pero no tipos básicos).
  - Hay dos clases que se adhieren al contrato de esta interfaz:
    - `ArrayList`
    - `LinkedList`
- 
- `List<String> lista1 = new ArrayList<>();`
  - `List<String> lista2 = new LinkedList<>();`

# Listas

- Hay que importarlas para poder usarlas:

```
import java.util.List  
import java.util.ArrayList;
```

- Crear una lista y añadirle elementos:

```
List<String> lista = new ArrayList<>();
```

```
lista.add("hola");
```

```
lista.add("que");
```

```
lista.add("tal");
```

```
lista.add(1, "amigo");
```

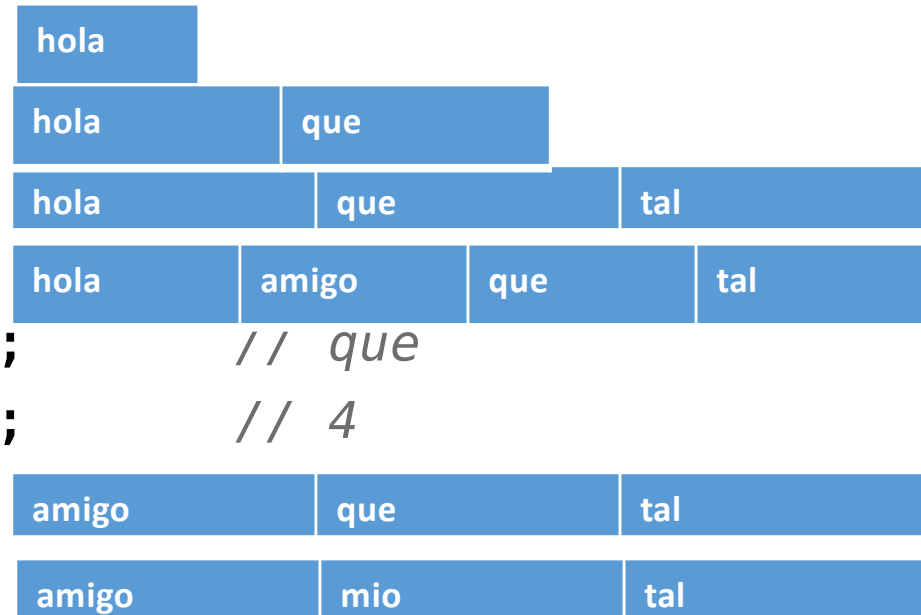
```
System.out.println(lista.get(2));           // que
```

```
System.out.println(lista.size());           // 4
```

```
lista.remove(0);
```

```
lista.set(1, "mio");
```

```
System.out.println(lista);                  // ["amigo", "mio", "tal"]
```



## Creación de listas inmutables

- Es posible crear una lista de una manera rápida pero inmutable:

```
List<String> lista = List.of("hola", "que", "tal");
```

```
lista.add(1, "amigo" ); // error
```

- Si queremos crear ahora una mutable a partir de ella:

```
List<String> listaMut = new ArrayList<>(lista);
```

```
listaMut.add(1, "amigo" ); // correcto
```

# Listas. Recorrido

- Dos formas de recorrerla completa:

```
List<String> lista = List.of("hola", "que", "tal");
```

```
for (int i = 0; i < lista.size(); i++) {  
    System.out.println(lista.get(i));  
}
```



```
// Mejor con un foreach  
for(String s : lista) {  
    System.out.println(s);  
}
```



# Listas. Remove

- En realidad, hay dos métodos remove:

```
T remove(int index)
```

```
boolean remove(Object obj)
```

```
List<String> lista = new ArrayList<>();
```

```
lista.add("hola");
```

```
lista.add("que");
```

```
lista.add("tal");
```

```
lista.remove("que"); // es lo mismo que lista.remove(1);
```

- ¿Qué ocurre si la lista es de enteros?

```
List<Integer> lista = new ArrayList<>();
```

```
lista.add(9);
```

```
lista.add(7);
```

```
lista.add(24);
```

```
lista.remove(1); // elimina el elemento de la posición 1
```

```
lista.remove((Integer)24); // elimina el elemento 24
```



# Solución a problemas con Interfaces

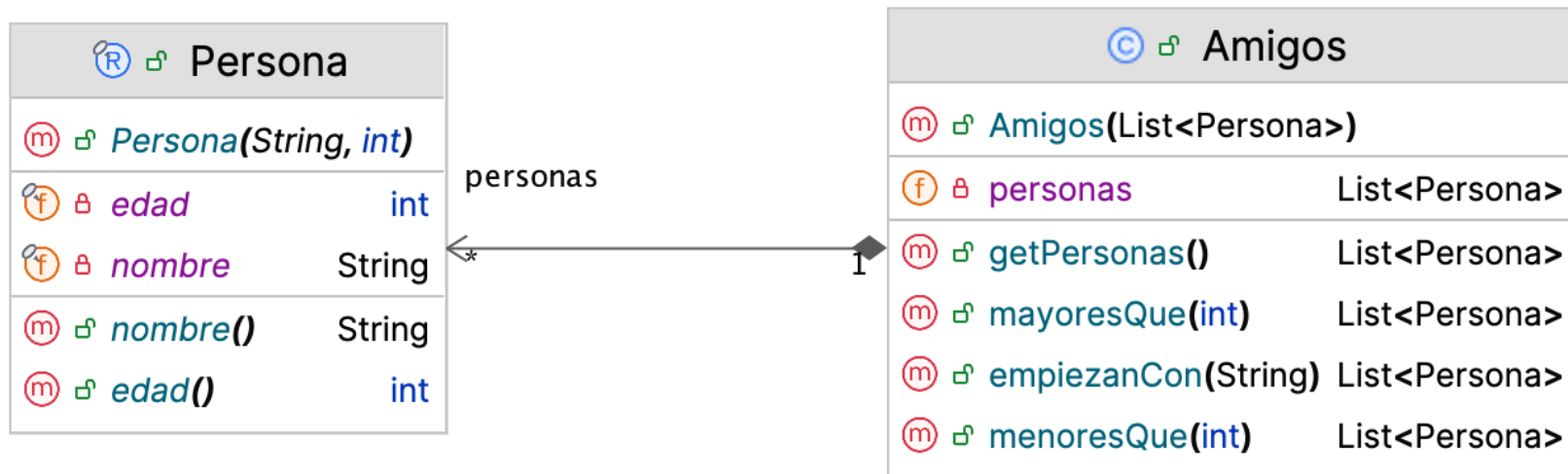
- Los interfaces proporcionan soluciones elegantes a problemas donde se desea aplicar diferentes funcionalidades a unos mismos datos (**abstracción funcional**).
- Veamos un ejemplo solucionado sin interfaces y después con interfaces aplicando abstracción funcional.
  - Tenemos dos clases
    - Clase **Persona** con información de una persona (nombre y edad)
    - Clase **Amigos** con información de muchas personas (un array de personas)
  - Queremos coleccionar las personas de Amigos:
    - Que son menores de una edad dada.
    - Que son mayores de una edad dada.
    - Cuyo nombre empieza por una cadena dada.
    - ...

# Caso 1: Solución sin interfaces

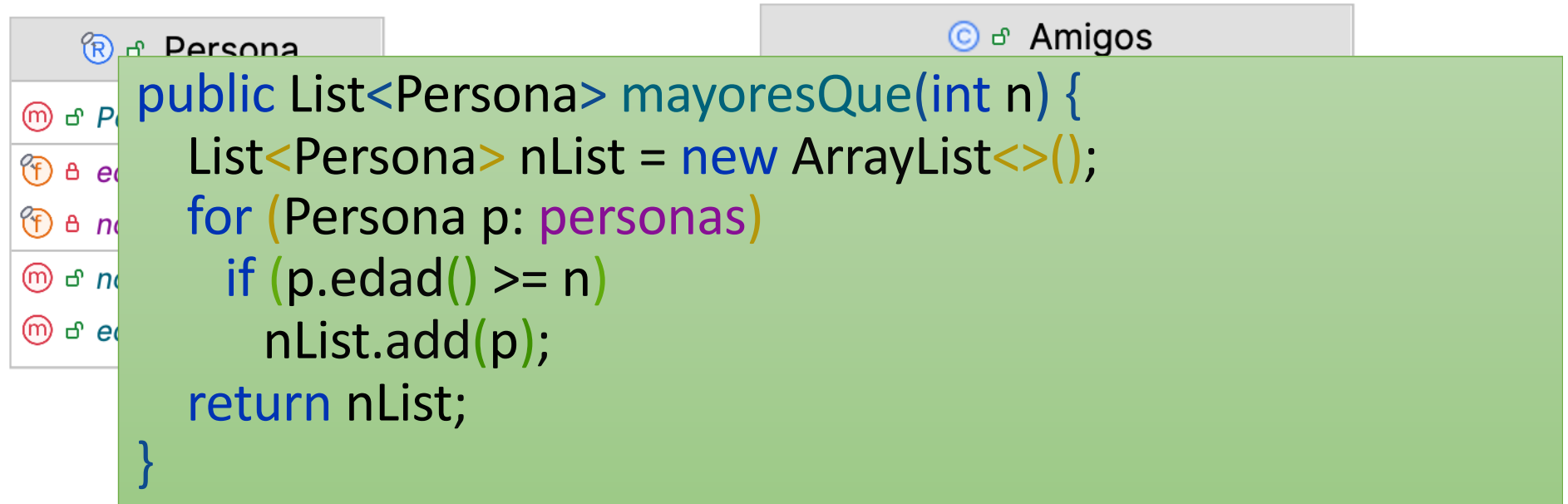
## Solución sin interfaces

- Se crea un método en la clase **Amigos** que resuelva cada una de las funcionalidades requeridas
  - Las personas menores de una edad dada.  
`List<Persona> menoresQue(int n)`
  - Las personas mayores de una edad dada.  
`List<Persona> mayoresQue(int n)`
  - Las personas que empiezan por una cadena dada.  
`List<Persona> empiezaCon(String s)`
  - ...
- El cuerpo de estos métodos será muy parecido, solo cambiarán la manera de seleccionar las personas.

# Solución sin interfaces



# Solución sin interfaces



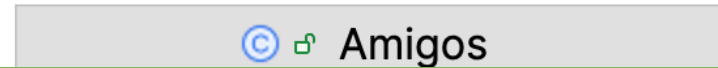
The image shows a screenshot of a Java IDE. At the top, there are two tabs: 'Persona' and 'Amigos'. Below the 'Persona' tab, there is a list of code elements with icons: a method 'mayoresQue', a field 'nList', a loop 'for', an if-statement 'if', and an 'add' method call. A green rectangular box is overlaid on the code, containing the following Java code snippet:

```
public List<Persona> mayoresQue(int n) {  
    List<Persona> nList = new ArrayList<>();  
    for (Persona p: personas)  
        if (p.edad() >= n)  
            nList.add(p);  
    return nList;  
}
```

# Solución sin interfaces

```
Persona Amigos  
public List<Persona> mayoresQue(int n) {  
    List<Persona> nList = new ArrayList<>();  
    for (Persona p: personas)  
        if (p.edad() > n)  
            nList.add(p);  
    return nList;  
}  
  
public List<Persona> menoresQue(int n) {  
    List<Persona> nList = new ArrayList<>();  
    for (Persona p: personas)  
        if (p.edad() <= n)  
            nList.add(p);  
    return nList;  
}
```

# Solución sin interfaces



```
public List<Persona> mayoresQue(int n) {  
    List<Persona> nList = new ArrayList<>();
```

```
public List<Persona> menoresQue(int n) {  
    List<Persona> nList = new ArrayList<>();  
    for (Persona p: personas)
```

```
public List<Persona> empiezanCon(String s) {  
    List<Persona> nList = new ArrayList<>();  
    for (Persona p: personas)  
        if (p.nombre().startsWith(s))  
            nList.add(p);  
    return nList;  
}
```

```
public class Main1 {  
    public static void main(String [] args) {  
        List<Persona> personas = List.of(  
            new Persona("juan", 25), new Persona("maria", 32), new Persona("marta", 28),  
            new Persona("julio", 33), new Persona("manuel", 29), new Persona("justino", 25));  
  
        Amigos amigos = new Amigos(personas);  
  
        System.out.println("Empiezan con ma");  
        List<Persona> ps1 = amigos.empiezanCon("ma");  
        for (Persona p : ps1)  
            System.out.println(p);  
  
        System.out.println("Mayores de 28");  
        List<Persona> ps2 = amigos.mayoresQue(28);  
        for (Persona p : ps2)  
            System.out.println(p);  
  
        System.out.println("Menores de 27");  
        List<Persona> ps3 = amigos.menoresQue(27);  
        for (Persona p : ps3)  
            System.out.println(p);  
    }  
}
```

## Solución sin interfaces

Para usarlos  
simplemente se  
invoca al método  
adecuado



# Solución sin interfaces

¿Cómo añadir una nueva funcionalidad?

- Por ejemplo : queremos coleccionar las personas que contienen un determinado texto en su nombre
- Habrá que modificar la clase **Amigos** y añadir un nuevo método:

```
public List<Persona> contieneA(String s) {  
    List<Persona> nList = new ArrayList<>();  
    for (Persona p: personas)  
        if (p.nombre().contains(s))  
            nList.add(p);  
    return nList;  
}
```

Los métodos son todos  
muy parecidos

!!! Hay que recompilar  
la clase Amigos !!!

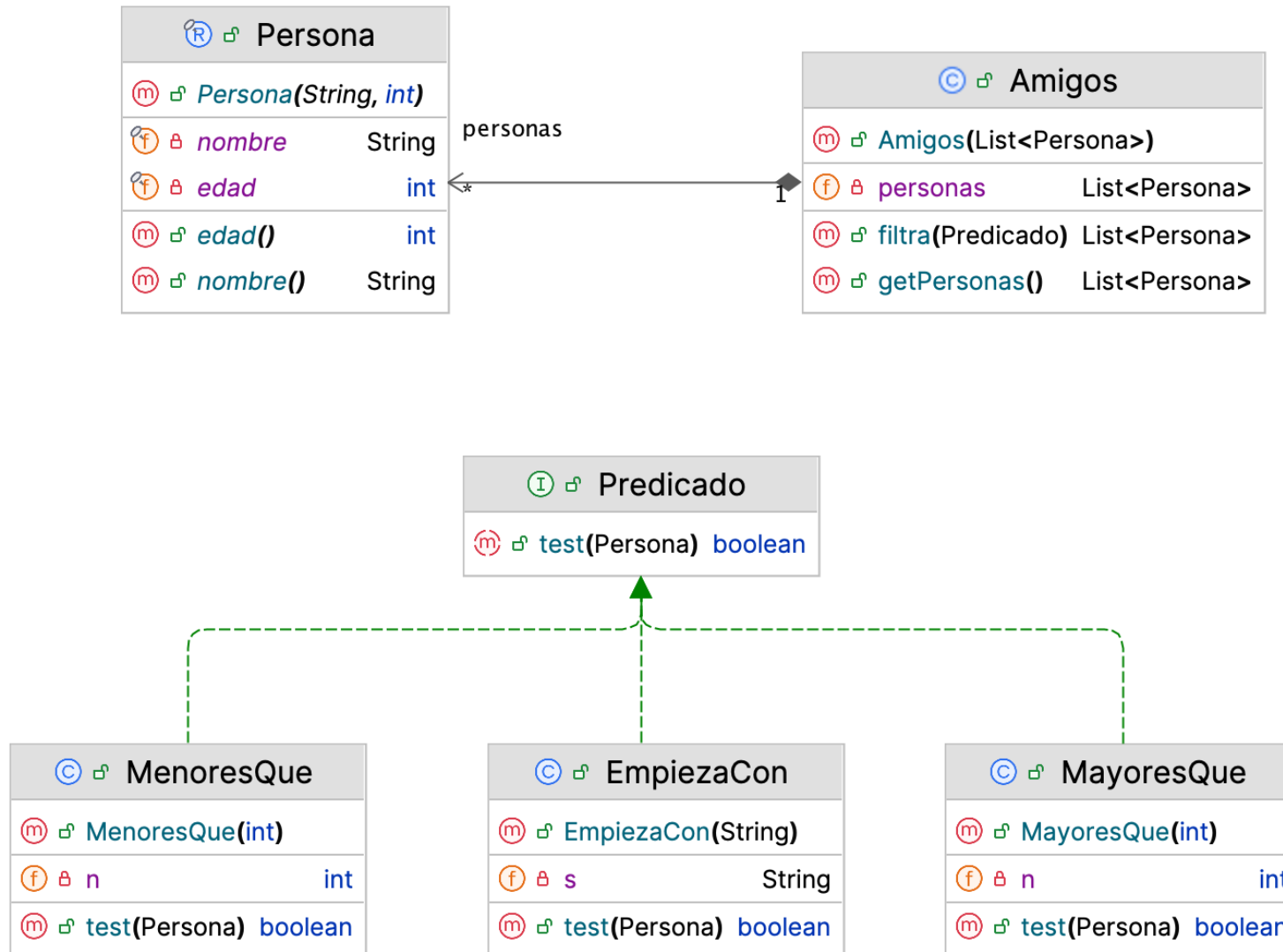
## Caso 2: Solución con interfaces

# Solución con interfaces (Abstracción funcional)

- Se crea una interfaz que defina la signature de un método que determinará la selección:
  - En el ejemplo la interfaz la llamaremos **Predicado**.
    - Incluirá la signature **boolean test(Persona p)**  
que será la que defina qué personas coleccionar según si pasan el test o no.
- Cada funcionalidad se define en una clase que implementa la interfaz
  - En el ejemplo, las clases serán **MenoresQue**, **MayoresQue**, y **EmpiezaCon** que implementan la interfaz **Predicado** y por tanto definen el método **test**, cada una a su manera.
- La clase contenedora dispone de un método que toma como argumento un objeto que implementa la interfaz y realiza la selección.
  - En el ejemplo, la clase **Amigos** implementa el método

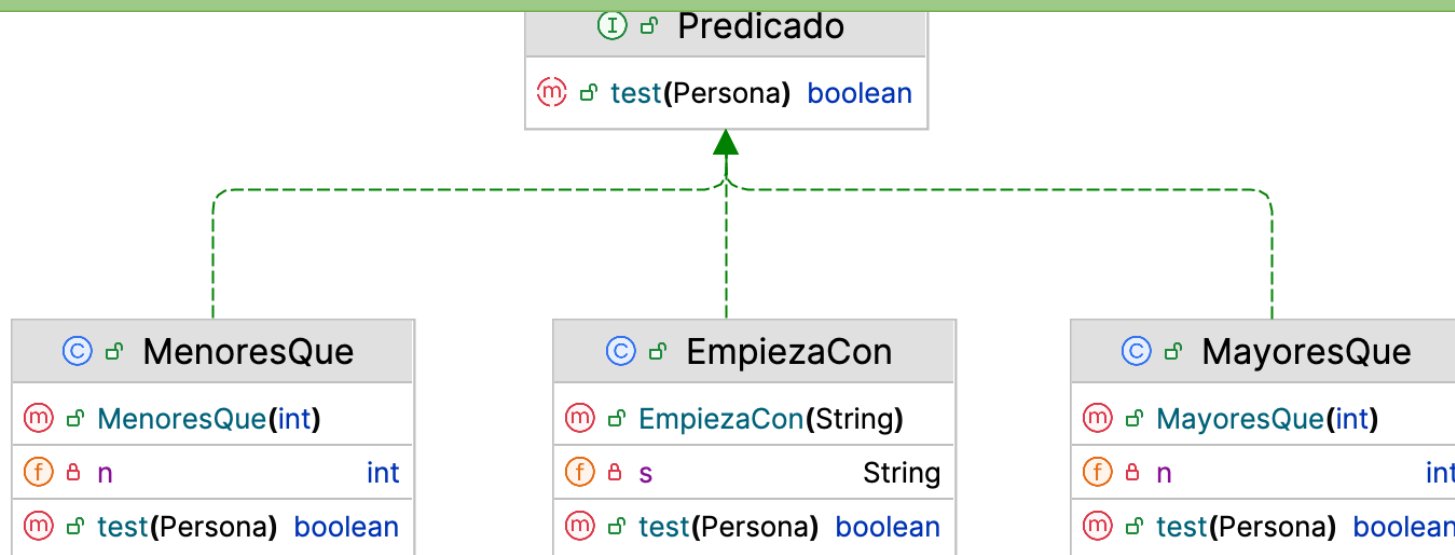
**List<Persona> filtra(Predicado pred)**

# Solución con interfaces



# Solución con interfaces

```
public List<Persona> filtra(Predicado pred) {
    List<Persona> nList = new ArrayList<>();
    for (Persona p: personas)
        if (pred.test(p))
            nList.add(p);
    return nList;
}
```



# Solución con interfaces

```

public List<Persona> filtra(Predicado pred) {
    List<Persona> resultado = new ArrayList<>();
    for (Persona p : personas) {
        if (pred.test(p)) {
            resultado.add(p);
        }
    }
    return resultado;
}

public class MayoresQue implements Predicado {
    private int n;

    public MayoresQue(int n) {
        this.n = n;
    }

    public boolean test(Persona p) {
        return p.edad() >= n;
    }
}
  
```

MenoresQue
MenoresQue(int)
n int
test(Persona) boolean

EmpiezaCon
EmpiezaCon(String)
s String
test(Persona) boolean

MayoresQue
MayoresQue(int)
n int
test(Persona) boolean

# Solución con interfaces

```

public List<Persona> filtra(Predicado pred) {
    List<Persona> resultado = new ArrayList<>();
    for (Persona p : personas) {
        if (pred.test(p)) {
            resultado.add(p);
        }
    }
    return resultado;
}

public class MayoresQue implements Predicado {
    private int n;

    public MayoresQue(int n) {
        this.n = n;
    }

    public boolean test(Persona p) {
        return p.edad() > n;
    }
}

public class MenoresQue implements Predicado {
    private int n;

    public MenoresQue(int n) {
        this.n = n;
    }

    public boolean test(Persona p) {
        return p.edad() <= n;
    }
}
  
```

## Solución con interfaces

```
public List<Persona> filtra(Predicado pred) {  
    List<Persona> lista = new ArrayList<>();  
    for (Persona p : personas) {  
        if (pred.test(p)) {  
            lista.add(p);  
        }  
    }  
    return lista;  
}  
  
public class MayoresQue implements Predicado {  
    private int n;  
    public MayoresQue(int n) {  
        this.n = n;  
    }  
    public boolean test(Persona p) {  
        return p.getEdad() > n;  
    }  
}  
  
public class MenoresQue implements Predicado {  
    private int n;  
    public MenoresQue(int n) {  
        this.n = n;  
    }  
    public boolean test(Persona p) {  
        return p.getEdad() < n;  
    }  
}  
  
public class EmpiezaCon implements Predicado {  
    private String s;  
    public EmpiezaCon(String s) {  
        this.s = s;  
    }  
    public boolean test(Persona p) {  
        return p.getNombre().startsWith(s);  
    }  
}
```



```
public class Main2 {  
    public static void main(String [] args) {  
        List<Persona> personas = List.of(  
            new Persona("juan", 25), new Persona("maria", 32), new Persona("marta", 28),  
            new Persona("julio", 33), new Persona("manuel", 29), new Persona("justino", 25));  
  
        Amigos amigos = new Amigos(personas);  
  
        System.out.println("Empiezan con ma");  
        List<Persona> ps1 = amigos.filtro(new EmpiezaCon("ma"));  
        for (Persona p : ps1)  
            System.out.println(p);  
  
        System.out.println("Mayores de 28");  
        List<Persona> ps2 = amigos.filtro(new MayoresQue(28));  
        for (Persona p : ps2)  
            System.out.println(p);  
  
        System.out.println("Menores de 27");  
        List<Persona> ps3 = amigos.filtro(new MenoresQue(27));  
        for (Persona p : ps3)  
            System.out.println(p);  
    }  
}
```

Se llama a  
filtro y se  
pasa como  
argumento  
el objeto  
que define  
el test

# Solución con interfaces

¿Cómo añadir la nueva funcionalidad ahora?

- Habrá que agregar otra clase que implemente la interfaz **Predicado**:

```
public class Contiene implements Predicado {  
    private String s;  
    public Contiene(String s) {  
        this.s = s;  
    }  
  
    public boolean test(Persona p) {  
        return p.nombre().contains(s);  
    }  
}
```

!!! No hay que recompilar nada !!!

## Resumen uso interfaces

Hemos creado una clase que implemente la interfaz `Predicado` y define el método `test` con el comportamiento deseado:

```
public class MenoresQue implements Predicado {  
    private int n;  
    public MenoresQue(int n) {  
        this.n = n;  
    }  
  
    public boolean test(Persona p) {  
        return p.edad() <= n;  
    }  
}
```

# Resumen uso interfaces

Luego creamos un objeto de esa clase y se lo pasamos como argumento al método `filtra` que requiere un `Predicado`:

```
public class Main2 {  
    public static void main(String [] args) {  
        List<Persona> personas = List.of(  
            new Persona("juan", 25), new Persona("maria", 32), new Persona("marta", 28),  
            new Persona("julio", 33), new Persona("manuel", 29), new Persona("justino", 25));  
  
        Amigos amigos = new Amigos(personas);  
  
        System.out.println("Mayores de 28");  
        List<Persona> ps2 = amigos.filtra(new MayoresQue(28));  
        for (Persona p : ps2)  
            System.out.println(p);  
    }  
}
```

## Programación Funcional

- Programa es un estilo claro y conciso
- El paralelismo es gratis
  - Los programas funcionales son intrínsecamente paralelos
- Es posible realizar demostraciones sobre propiedades de los programas.
  - No es testing, ¡es matemáticas!
- Vamos a ver cómo este estilo funcional modela:
  - Comportamiento
    - Lambdas, funciones de orden superior y composición funcional
  - Datos
    - Tipos algebraicos y concordancia de patrones

## Lambdas. Interfaces funcionales

- Un interfaz que solo tenga un método abstracto se dice que es un **Interface Funcional** o **SAM** (Single Abstract Method).
- La anotación `@FunctionalInterface` se asegura de que una interface sea realmente funcional.

```
@FunctionalInterface
public interface Predicado {
    boolean test(Persona p);
}
```

## Situación actual

- Hay que crear una clase que implemente la interfaz

```
public class MenoresQue implements Predicado {  
    private int n;  
    public MenoresQue(int n) {  
        this.n = n;  
    }  
  
    public boolean test(Persona p) {  
        return p.edad() <= n;  
    }  
}
```

- Y crear un objeto para usarla

```
Predicado pred = new MenoresQue(27);  
List<Persona> ps3 = amigos.filtrar(pred);
```

## Lambdas

- Una expresión lambda permite implementar de forma muy simple una **interfaz funcional**.

```
Predicado pred = (Persona p) -> p.edad() <= 27;  
List<Persona> ps3 = amigos.filtrar(pred);
```

- (Persona p) es el argumento del método test (único método de la interfaz)
- Lo que hay detrás de la flecha -> es el cuerpo del método test. No es necesario usar la palabra **return**



# Definición de lambda

(parámetros)  $\rightarrow$  expresión

```
(String s, int len) -> s.length() >= len
```

Define una función que:

**toma dos argumentos:** una cadena s y un entero len

**devuelve:** un booleano.

En concreto, devuelve true si la longitud de s es mayor o igual que len y false en otro caso.

```
(String s) -> s.toLowerCase()
```

Define una función que:

**toma un argumento:** una cadena s

**devuelve:** una cadena igual pero en minúsculas.

```
() -> System.out.println("Hola");
```

```
(int i) -> i%10;
```

```
(double b, double e) -> Math.pow(b, e);
```

## Tipo y uso de una lambda

- El tipo de una expresión lambda es el de la interfaz funcional a la que se asigna.

```
@FunctionalInterface
public interface MiFuncion {
    int aplica(int a, int b);
}
```

```
public class Main {
    public static void main(String [] args) {
        MiFuncion f1 = (int x, int y) -> x + y;
        MiFuncion f2 = (int x, int y) -> x - y;

        System.out.println(f1.aplica(2,3));
        pasaLambda(f2);
        pasaLambda((int x, int y) -> x + y);
    }

    public static void pasaLambda(MiFuncion f) {
        System.out.println(f.aplica(5,6));
    }
}
```

# Inferencia de tipos en lambda

- Si el tipo del parámetro se puede inferir, puede eliminarse

```
public class Main {  
    public static void main(String [] args) {  
        MiFuncion f1 = (x, y) -> x + y;  
        MiFuncion f2 = (x, y) -> x - y;  
        System.out.println(f1.aplica(2,3));  
        pasaLambda(f2);  
        pasaLambda((x, y) -> x + y);  
    }  
    public static void pasaLambda(MiFuncion f) {  
        System.out.println(f.aplica(5,6));  
    }  
}
```

```
@FunctionalInterface  
public interface MiFuncion {  
    int aplica(int a, int b);  
}
```

Si tiene un parámetro, los paréntesis pueden omitirse:  $x \rightarrow x + 1$

# Lambdas con bloques

(parámetros) -> { cuerpo-lambda }

- Podemos utilizar bloques en el cuerpo de una lambda.
  - Se usa **return** para devolver el resultado

```
@FunctionalInterface  
public interface MiFuncion {  
    int aplica(int a, int b);  
}
```

```
MiFuncion max = (x, y) -> {  
    int z = x;  
    if (x < y)  
        z = y;  
    return z;  
};
```

```
System.out.println(max.aplica(2,3));
```

# El paquete `java.util.function`

- Define interfaces funcionales de utilidad

`Predicate<T>`

`BiPredicate<T,U>`

`Function<T,R>`

`BiFunction<T,U,R>`

`XXXFunction<R>`

`Consumer<T>`

`Supplier<R>`

`UnaryOperator<T>`

`BinaryOperator<T>`

`ToXXXFunction<T>`

`XXXToYYYFunction`

Versiones para los tipos básicos

`boolean test(T)`

`boolean test(T,U)`

`R apply(T)`

`R apply(T,U)`

`R apply(xxx)`

`void accept(T)`

`R get()`

`T apply(T)`

`T apply(T,T)`

`xxx applyAsXXX(T)`

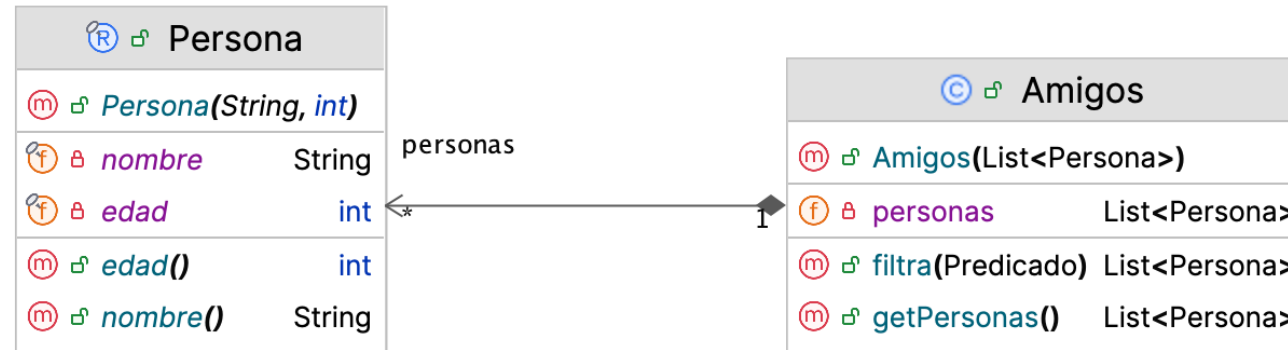
`yyy applyAsYYY(xxx)`

# Implementación de una interfaz funcional. Ejemplo

```
public interface Interfaz1 {  
    double miFun(int i);  
}  
  
class Main {  
    public static void main(String[] args) {  
        Interfaz1 i2 = x -> Math.sqrt(x);           // int -> double  
        System.out.println(i2.miFun(3));  
        Function<Integer, Double> f1 = x -> Math.sqrt(x); // Integer -> Double  
        System.out.println(f1.apply(3));  
        IntToDoubleFunction idf = x -> Math.sqrt(x);    // int -> double  
        System.out.println(idf.applyAsDouble(3));  
        ToDoubleFunction<Integer> td = x -> Math.sqrt(x); // Integer -> double  
        System.out.println(td.applyAsDouble(3));  
        Predicate<Integer> p = x -> Math.sqrt(x) > 5;   // Integer -> boolean  
        System.out.println(p.test(10));  
    }  
}
```

## Caso 3: Solución con interfaces y lambdas

# Solución con interfaces y lambdas

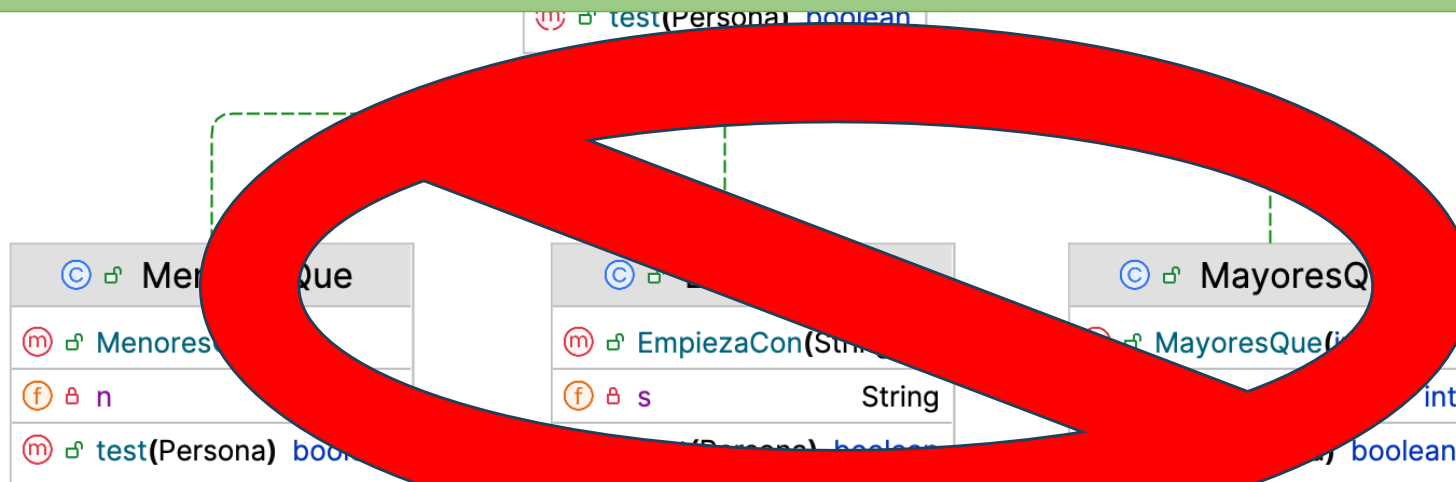




# Solución con interfaces y lambdas

Persona

```
public List<Persona> filtra(Predicate<Persona> pred) {
    List<Persona> nList = new ArrayList<>();
    for (Persona p: personas)
        if (pred.test(p))
            nList.add(p);
    return nList;
}
```



```
public class Main3 {  
    public static void main(String [] args) {  
        List<Persona> personas = List.of(  
            new Persona("juan", 25), new Persona("maria", 32), new Persona("marta", 28),  
            new Persona("julio", 33), new Persona("manuel", 29), new Persona("justino", 25));  
  
        Amigos amigos = new Amigos(personas);  
  
        System.out.println("Empiezan con ma");  
        List<Persona> ps1 = amigos.filtro(p -> p.nombre().startsWith("ma"));  
        for (Persona p : ps1)  
            System.out.println(p);  
  
        System.out.println("Mayores de 28");  
        List<Persona> ps2 = amigos.filtro(p -> p.edad() > 28);  
        for (Persona p : ps2)  
            System.out.println(p);  
  
        System.out.println("Menores de 27");  
        Predicate<Persona> pred = (Persona p) -> p.edad() < 27;  
        List<Persona> ps3 = amigos.filtro(pred);  
        for (Persona p : ps3)  
            System.out.println(p);  
    }  
}
```

Se llama a  
filtro y se  
pasa como  
la lambda  
que define  
el test

## Stream: Idea de flujo

- Un flujo es una propuesta de suministro de objetos o valores que se produce cuando haya una demanda de ellos.
- Por ejemplo:
  - Tenemos una lista de coches y creamos un flujo a partir de ellos y operamos así:
    - De cada coche obtenemos sus baterías,
    - De cada batería obtenemos su voltaje
    - Nos quedamos con los voltajes mayores que 13 voltios

Esto es como una cadena de montaje, pero aún no ha pasado ningún coche de la lista inicial por la cadena.

Para poner en marcha el flujo necesitamos pedirle algo

- Cuántos voltajes quedan con estas condiciones
- Cuál es el mínimo de los que quedan en estas condiciones
- Cuánto suman los voltajes que quedan en estas condiciones

```
public class Coche {
    Bateria getBateria() {
```

...

}

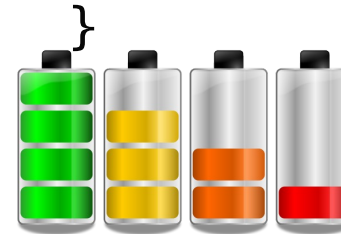
}



```
class Bateria {
    double getVoltaje() {
```

...

}



Extrae baterías

Extrae voltajes

18 15 8 4

Filtra  
>13

18 15

cuenta

2

```
class Main2 {
    public static void main(String[] args) {
        List<Coche> lc = List.of(new Coche(...),...);
        long cont =
            lc.stream()
                .map(coche -> coche.getBateria())
                .mapToDouble(bateria -> bateria.getVoltaje());
                .filter(voltaje -> voltaje > 13)
                .count();
        System.out.println(cont);
    }
}
```

// muchos coches

// flujo de coches  
 // flujo de baterías  
 // flujo de voltajes  
 // flujo de voltajes  
 // Terminador de flujo

# Generadores de flujo

Permiten crear flujos

- `Stream.of(...)`

- `stream()` sobre colecciones

- `chars()` sobre `String`

- `IntStream.range(int, int)`

- `InstStream.rangeClosed(int,int)`

## Dos tipos de operaciones con flujos

Intermedias

- Dado un flujo

- Generan como resultado otro flujo

Finales o terminadoras

- Extraen información de un flujo.

- El flujo se consume

# Algunos operadores intermedios de flujo

Para todos los flujos

`filter(Predicate<T> pred)`

`map(Function<T,R> mapper)`

`mapToYYY(XXToYYYFunction mapper)` // Flujo numérico

`limit(long n),`

`skip(long n)`

`takeWhile(Predicate<T> pred),`

`dropWhile(Predicate<T> pred)`

# Algunos terminadores de flujo

Para flujos numéricos

`sum()`, `max()`, `min()`, `average()`

Para todos los flujos

`count()`

`findFirst()`

`allMatch(Predicate<T> pred)`,

`anyMatch(Predicate<T> pred)`

`forEach(Consumer<T>)`

`toList()` // lista inmutable

`collect(Collections.toList())` // lista mutable

`collect(Collections.toSet())` // Conjunto mutable

```
class Bateria {  
    double getVoltaje() {  
        ...  
    }  
}
```

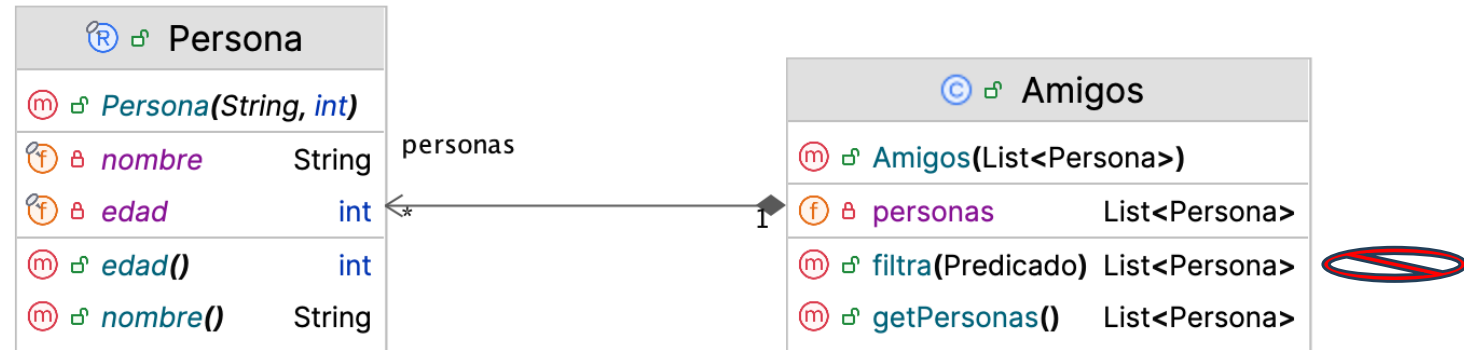
```
public class Coche {  
    Bateria getBateria() {  
        ...  
    }  
}
```

```
class Main2 {  
    public static void main(String[] args) {  
  
        List<Coche> lc = List.of(new Coche(...),...);           // muchos coches  
  
        long cont =  
            lc.stream()  
                .map(c -> c.getBateria())                       // flujo de coches  
                .mapToDouble((b -> b.getVoltaje()))              // flujo de baterías  
                .filter(v -> v > 13)                             // flujo de voltajes  
                .count();                                         // flujo de voltajes  
                                                                // Terminador de flujo  
  
        System.out.println(cont);  
    }  
}
```



## Caso 4: Solución con interfaces, lambdas y flujos

# Solución sin interfaces, lambdas y flujos



```
public class Main4 {  
    public static void main(String [] args) {  
        List<Persona> personas = List.of(  
            new Persona("juan", 25), new Persona("maria", 32), new Persona("marta", 28),  
            new Persona("julio", 33), new Persona("manuel", 29), new Persona("justino", 25));  
  
        Amigos amigos = new Amigos(personas);  
  
        System.out.println("Empiezan con ma");  
        List<Persona> ps1 = amigos.getPersonas()  
            .stream()  
            .filter(p -> p.nombre().startsWith("ma"))  
            .toList();  
        ps1.forEach(p -> System.out.println(p));  
  
        System.out.println("Mayores de 28");  
        List<Persona> ps2 = amigos.getPersonas()  
            .stream()  
            .filter(p -> p.edad() > 28)  
            .toList();  
        ps2.forEach(p -> System.out.println(p));  
    }  
}
```

Con flujos se  
pasa como  
argumento  
la lambda  
que define  
el test

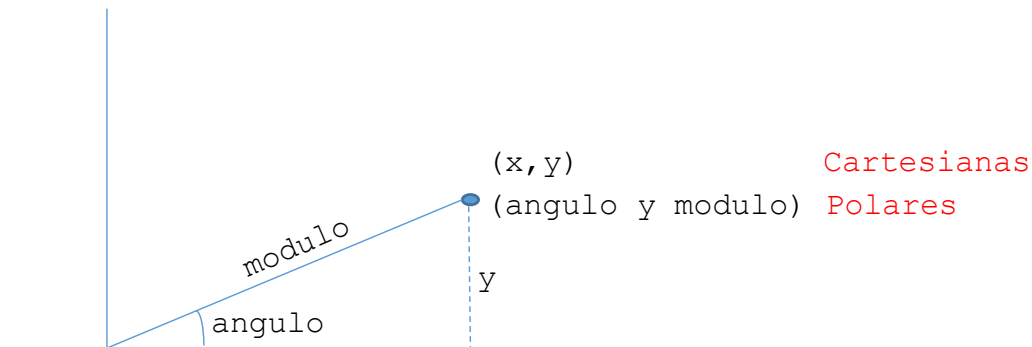
# Datos algebraicos. Clases selladas

- Modelan situaciones en las que un dato puede tener varias formas.
  - Un punto puede venir en coordenadas cartesianas o polares.
  - Una temperatura puede venir en grados Celsius o Fahrenheit.
- Puntos
  - Podemos crear la interfaz Punto y dos clases que la implementan
    - Cartesiana
    - Polar
- Temperaturas
  - Podemos crear la interfaz Temperatura y dos clases que la implementan
    - Celsius
    - Fahrenheit

No podemos prohibir que nadie  
más implemente la interfaz

Pero sí se puede prohibir  
con clases o interfaces selladas

# Clases e interfaces selladas. Puntos



$$modulo = \sqrt{x^2 + y^2}$$

$$angulo = \tan^{-1} \frac{y}{x}$$

$$x = modulo * \cos angulo$$

$$y = modulo * \sin angulo$$

```
public sealed interface Punto permits Cartesiano, Polar {
    default double x() {
        return modulo()*Math.cos(angulo());
    }
    default double y() {
        return modulo()*Math.sin(angulo());
    }
    default double modulo() {
        return Math.sqrt(Math.pow(x(),2)+Math.pow(y(),2));
    }
    default double angulo() {
        return Math.atan2(y(),x());
    }
    default double distancia(Punto pto) {
        return Math.sqrt(Math.pow(x()-pto.x(),2) + Math.pow(y()-pto.y(),2));
    }
    static Punto origen() {
        return new Cartesiano(0,0);
    }
}
```

# Interfaces selladas. Puntos

```
public record Cartesiana(double x, double y) implements Punto {}
```

```
public record Polar(double angulo, double modulo) implements Punto {}
```

# Concordancia de patrones

Descomposición de un registro para conocer sus componentes

Se descompone según su constructor.

Puede usarse en instanceof o en switch

```
public static boolean primerCuadrante(Punto pto) {  
    boolean res = false;  
    if (pto instanceof Cartesiano(double x, double y))  
        res = x >= 0 && y >= 0;  
    else if (pto instanceof Polar(double m, double a))  
        res = a >= 0 && a <= Math.PI/2;  
    return res;  
}  
  
public static boolean primerCuadrante(Punto pto) {  
    return switch(pto) {  
        case Cartesiano(double x, double y) -> x >= 0 && y >= 0;  
        case Polar(_, double a) -> a <= 0 && a <= Math.PI/2;  
    };  
}
```

## Java 22

## Interfaces selladas. Puntos

```
public class Main {  
    public static void main(String[] args) {  
        List<Punto> list = List.of(new Polar(0, 4),  
                                   new Cartesiana(3, 3),  
                                   new Polar(Math.PI / 4, 2));  
        for (Punto p : list)  
            System.out.println(p + " " + p.x() + " " + p.y() + " " +  
                               p.angulo() + " " + p.modulo());  
        Punto p1 = list.get(0);  
        Punto p2 = list.get(1);  
        System.out.println(p1.distancia(p2));  
        list.forEach(p -> System.out.println(primerCuadrante(p)));  
        double d =  
            list.stream()  
                .filter(p -> primerCuadrante(p))  
                .mapToDouble(p -> p.modulo())  
                .average();  
    }  
}
```



## Dato Algebraico: Degree

```
public record Celsius(double degrees) implements Degree {}
public record Fahrenheit(double degrees) implements Degree{}

public sealed interface Degree permits Celsius, Fahrenheit {
    default boolean isFrozen() {
        return switch(this) {
            case Celsius(double gc) -> gc < 0;
            case Fahrenheit(double gf) -> gf < 32;
        };
    }
    default Degree toCelsius() {
        return switch(this) {
            case Celsius _ -> this;
            case Fahrenheit(double gf)
                -> new Celsius((gf-32)/1.8);
        };
    }
    ...
}
```

# Dato Algebraico: Degree

```
public sealed interface Degree permits Celsius, Fahrenheit {  
    ...  
    default Degree toFahrenheit() {  
        return switch (this) {  
            case Celsius(double gc) -> new Fahrenheit(gc*1.8+32);  
            case Fahrenheit _ -> this;  
        };  
    }  
    // Métodos de Fábrica  
    static Degree fahrenheit(double degrees) {  
        return new Fahrenheit(degrees);  
    }  
    static Degree celsius(double degrees) {  
        return new Celsius(degrees);  
    }  
}
```

## Dato Algebraico: Degree

```
public class Main {  
    public static void main(String[] args) {  
        List<Degree> ld = List.of(  
            Degree.fahrenheit(37),  
            Degree.celsius(-7),  
            Degree.fahrenheit(29),  
            Degree.celsius(31)  
        );  
        ld.forEach(d -> System.out.println(d.toCelsius()));  
        ld.forEach(d -> System.out.println(d.isFrozen()));  
    }  
}
```

## Ejemplo

Dada una lista de puntos, cómo saber si todos los puntos cuya abscisa sea positiva se encuentran a menos de una unidad del origen de coordenadas:

```
List<Punto> list = List.of(  
    new Cartesiano(0.5,0.9),  
    new Cartesiano(-1.1,0.4),  
    new Cartesiano(1.0,0.1),  
    new Polar(2,Math.PI/3));  
  
Punto origen = new Cartesiano(0,0);  
  
boolean res = list  
    .stream()  
    .filter(p -> p.x() > 0)  
    .allMatch(p -> p.distancia(origen) < 1)
```