


Tema 2

Introducción a Java

Contenido

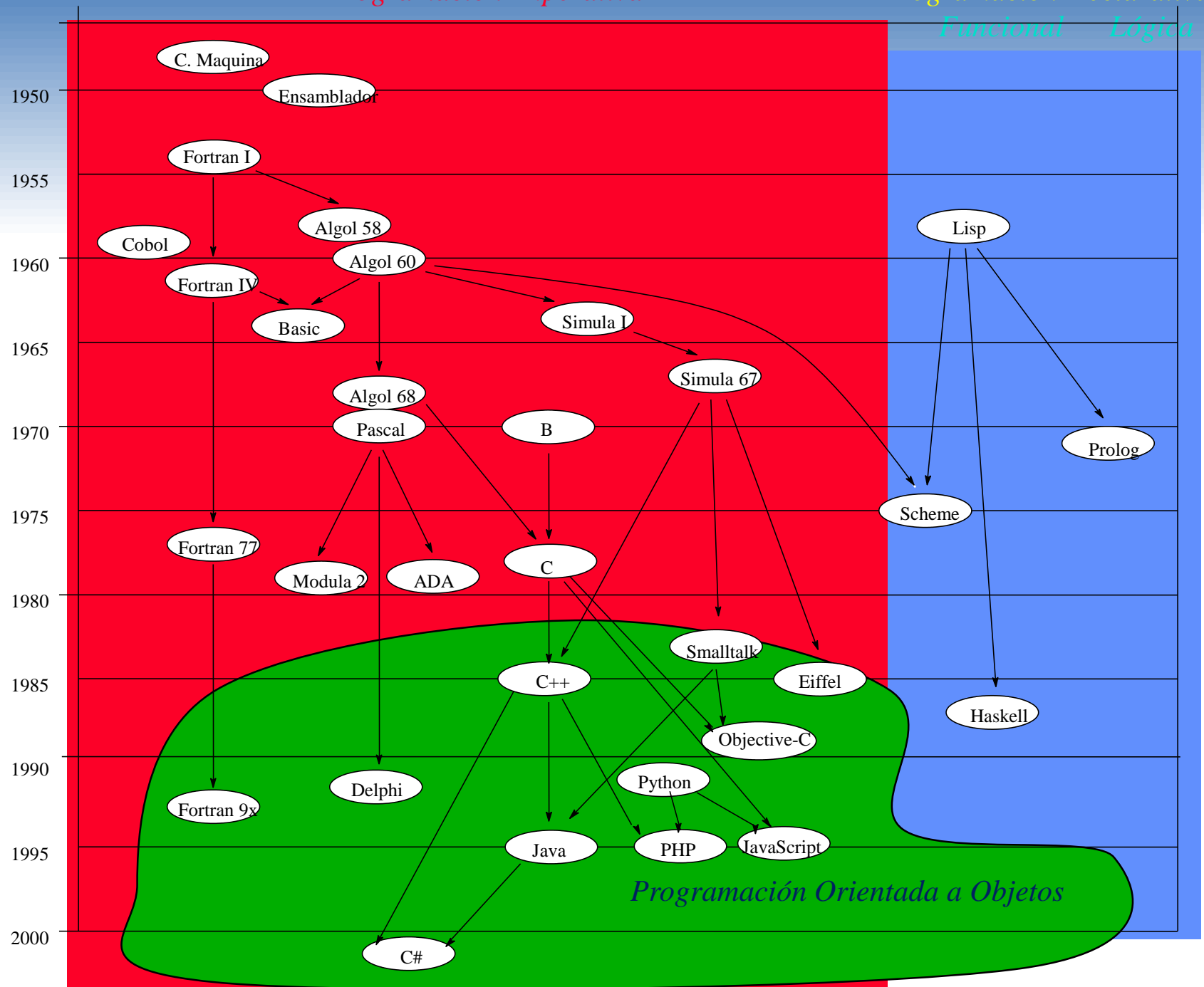
- Introducción histórica 
- Programas y Paquetes
- Clases y Objetos
- Elementos del Lenguaje:
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones
- Cadenas de caracteres
- Arrays
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces

Programación Imperativa

Programación Declarativa

Funcional


Lógica



Introducción a Java

- Desarrollado por Sun. Aparece en 1995
- Basado en C++ (y algo en Smalltalk) eliminando
 - definiciones de tipos de valores y macros,
 - punteros y aritmética de punteros,
 - necesidad de liberar memoria.
- Fiable y Seguro:
 - memoria dinámica automática (no punteros explícitos)
 - comprobación automática de tamaño de variables
- Orientado a Objetos con:
 - herencia simple y polimorfismo de datos,
 - redefinición de métodos y vinculación dinámica.
 - concurrencia integrada en el lenguaje
 - interfaz gráfica integrada en el lenguaje

Contenido

- Introducción histórica
- Programas y Paquetes 
- Clases y Objetos
- Elementos del Lenguaje:
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones
- Cadenas de caracteres
- Arrays
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces

Programa en Java

- Formado por una o varias clases diseñadas para resolver un determinado problema.
- Existe una clase (pública) “distinguida” que contiene un método de clase especial:

```
public static void main(String[] args)
```

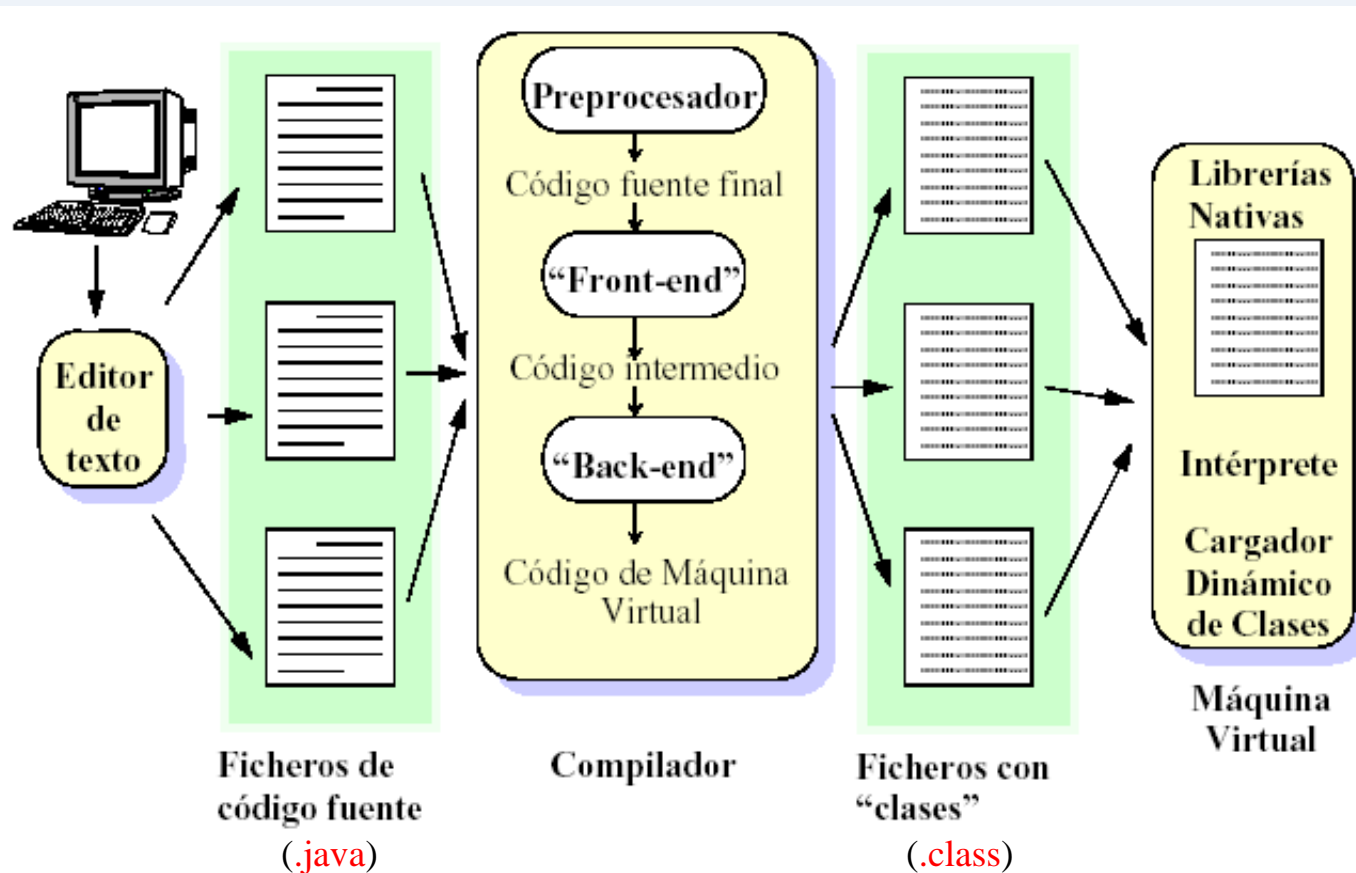
que desencadena la ejecución del programa.

- Esta clase distinguida puede contener más métodos.
- Las demás clases pueden estar definidas *ad hoc* o pertenecer a una biblioteca de clases.
- Cuando se trabaja con un IDE (como Eclipse), normalmente se crea un Proyecto para alojar el programa. Y el Proyecto estará dentro de un Espacio de Trabajo (Workspace)

Programa en Java

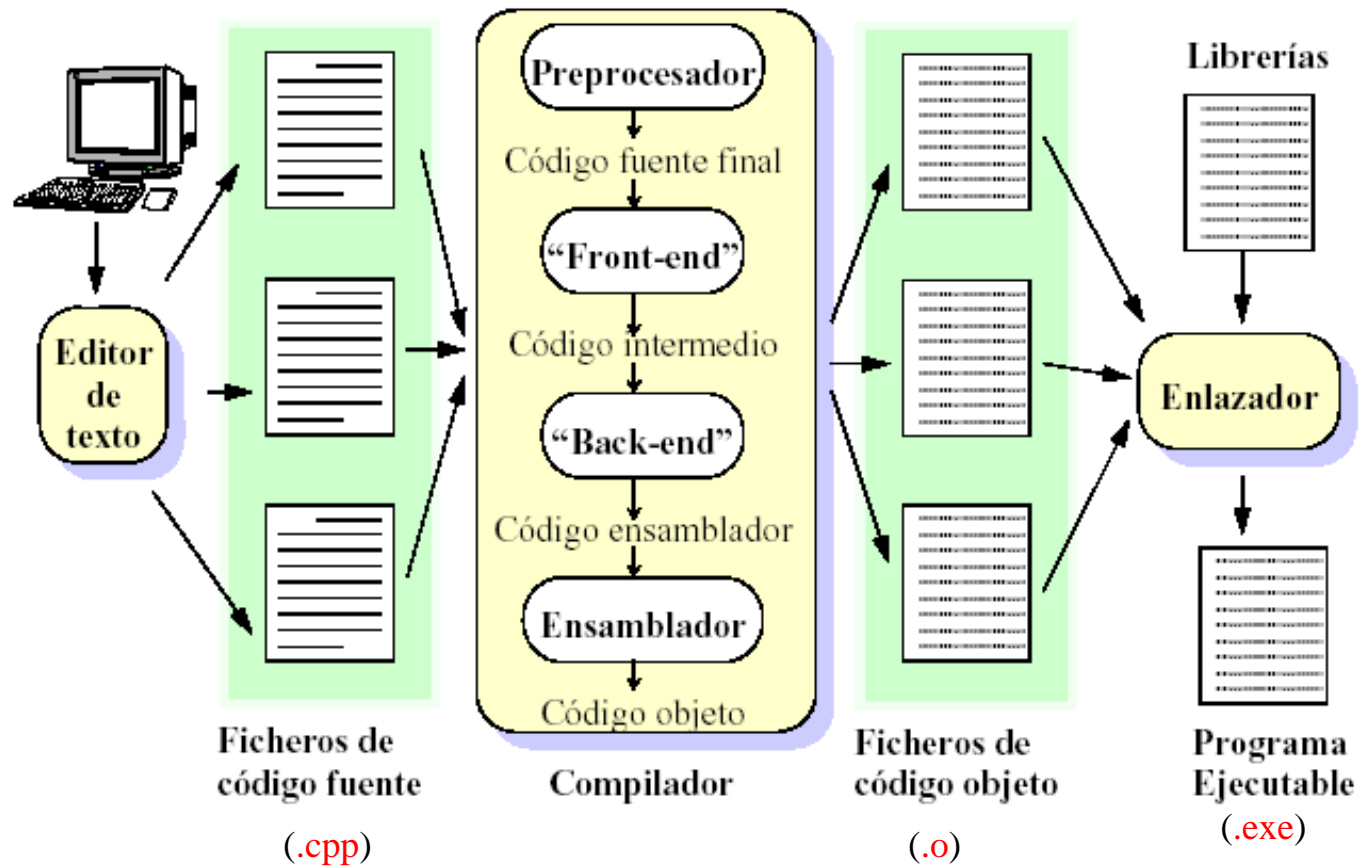
- Cada clase declarada como pública debe estar en un fichero **.java** con su mismo nombre.
- Cada fichero **.java** puede contener varias clases pero sólo una podrá ser pública. En este curso sólo pondremos una clase por fichero.
- Cada fichero **.java** debe compilarse generando un fichero **.class** (en *bytecode*) por cada clase contenida en él.
- El programa se ejecuta pasando el fichero **.class** de la clase distinguida al intérprete (**Máquina Virtual de Java**)

Compilación e Interpretación en Java



Compilación e Interpretación (Java)

Diferencia con otros lenguajes



Compilación y Enlazado (Ej. C++)

Ejemplo1 (Clase Distinguida)

“Hola.java”

```
public class Hola {  
    public static void main(String[] args) {  
        System.out.println(“Hola Clase”);  
    }  
}
```

Ejemplo2 (Clase Punto)

“Punto.java”

```
public class Punto {  
    private double x, y;  
    public Punto() { this(0,0); }  
    public Punto(double a, double b) { x = a; y = b; }  
    public double abscisa() {return x;}  
    public double ordenada() {return y;}  
    public void abscisa(double a){ x = a; }  
    public void ordenada(double b){ y = b; }  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public double distancia(Punto pto) {  
        return Math.sqrt(Math.pow(x - pto.x, 2) +  
                           Math.pow(y - pto.y, 2));  
    }  
}
```

Ejemplo2 (Clase Distinguida)

“TestPunto.java”

```
public class TestPunto {  
    public static void main(String[] args) {  
        Punto p1, p2;  
        p1 = new Punto(1,1);  
        p2 = new Punto(0,0);  
        System.out.println("La distancia entre los puntos es: "  
                           + p1.distancia(p2));  
        System.out.println("Trasladamos el primer punto (+2,+3)");  
        p1.trasladar(2, 3);  
        System.out.println("Ahora La distancia entre los puntos es: "  
                           + p1.distancia(p2));  
    }  
}
```

Paquetes

- En Java las clases se organizan en ***paquetes*** (**package**):
 - Un paquete es un **mecanismo lógico de organización** para agrupar clases relacionadas.
 - Por convención, el **nombre de un paquete** estará formado por letras minúsculas, con la posibilidad de usar dígitos y el símbolo _
 - Todas **las clases de un determinado paquete** deben estar **localizadas físicamente** en una misma **carpeta** con el nombre establecido para el paquete.

Paquetes

- En Java las clases se organizan en ***paquetes*** (**package**):
 - En el entorno que usaremos (**Eclipse**) las clases de **un determinado paquete**, dentro de un proyecto, estarán **localizadas físicamente** en una **carpeta** con el nombre del paquete **dentro de la carpeta src**, que está situada en la carpeta principal del proyecto.
 - Si **una clase** del proyecto **no especifica el paquete** al que pertenece, formará parte del paquete “por defecto” (***default package***). Esa clase estará situada directamente **en la carpeta src**. Normalmente será una clase “distinguida”.

Paquetes

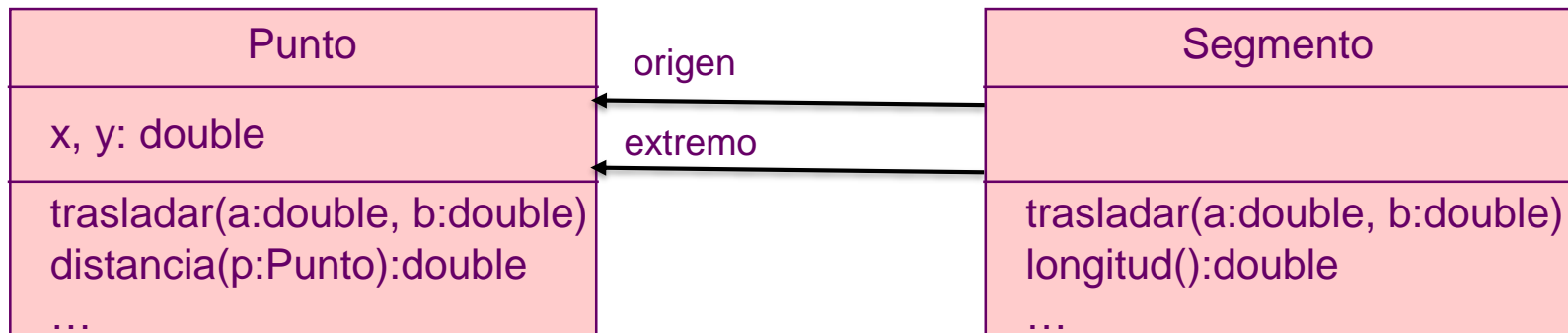
- En Java las clases se organizan en ***paquetes*** (**package**):
 - Las clases de un paquete sólo se pueden acceder por sus nombres desde otra clase dentro del mismo paquete
 - Para acceder a ellas desde otro paquete hay que hacerlo precediéndolas con el nombre del paquete o utilizando la construcción **import**

Paquetes

Ejemplo:

Recordemos, tal y como vimos en el tema 1, que aquí estamos usando el concepto de “*Composición*”:

un Segmento “*tiene*” o “*está compuesto/a por*” dos puntos



Paquetes

```
package paq1;

public class Punto {
    private double x, y;

    public Punto() { this(0,0); }
    public Punto(double a, double b) { x = a; y = b; }
    public double abscisa() { return x; }
    public double ordenada() { return y; }
    ...
}
```

Punto.java

Paquetes

`package paq1;` — mismo paquete

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new Punto(x1, y1);  
        extremo = new Punto(x2, y2);  
    }  
    ...  
}
```

correcto

Segmento.java

Paquetes

`package paq2;` — distinto paquete

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new Punto(x1, y1);  
        extremo = new Punto(x2, y2);  
    }  
    ...  
}
```

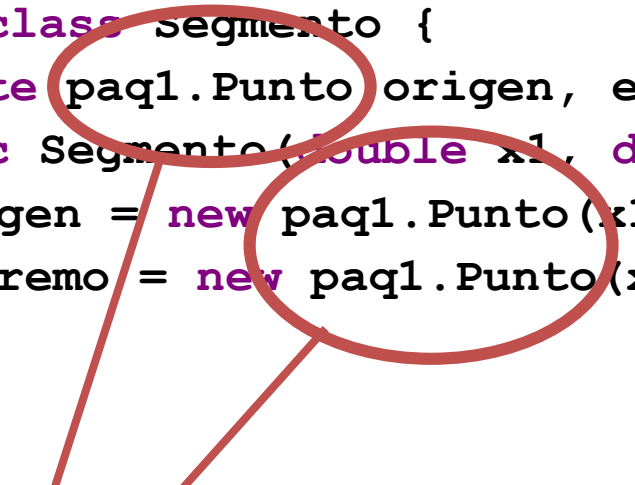
incorrecto

Segmento.java

Paquetes

```
package paq2;
```

```
public class segmento {  
    private paq1.Punto origen, extremo;  
    public segmento(double x1, double y1, double x2, double y2) {  
        origen = new paq1.Punto(x1, y1);  
        extremo = new paq1.Punto(x2, y2);  
    }  
    ...  
}
```



correcto

Segmento.java

Paquetes

```
package paq2;
```

```
import paq1.Punto; // import paq1.*;
```

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new Punto(x1, y1);  
        extremo = new Punto(x2, y2);  
    }  
    ...  
}
```

correcto

Segmento.java

Paquetes básicos del sistema (Bibliotecas de Java)

- `java.lang`: para funciones del lenguaje
- `java.util`: para utilidades adicionales
- `java.io`: para entrada y salida
- `java.text`: para formato especializado
- `java.awt`: para diseño gráfico e interfaz de usuario
- `java.awt.event`: para gestionar eventos
- `javax.swing`: nuevo diseño de GUI
- `java.net`: para comunicaciones
- ...

Acceso a las bibliotecas de Java

- A las clases incluidas en `java.lang` se puede acceder simplemente por sus nombres sin necesidad de anteponer (o importar) el nombre del paquete, p.e.: **System** o **Math**.
- Para el resto de clases de las bibliotecas sí hay que especificar el nombre del paquete

Ejemplo

- Programa para calcular el valor medio de un millón de números generados aleatoriamente, usando las clases **Random** del paquete **java.util** y **System** del paquete **java.lang**.

```
public class TestAleatorio {  
    public static void main(String[] args) {  
        java.util.Random rnd = new java.util.Random();  
        double sum = 0.0;  
        for (int i = 0; i < 1000000; i++) {  
            sum += rnd.nextDouble();  
        }  
        System.out.println("media = " + sum / 1000000.0);  
    }  
}
```


Ejemplo

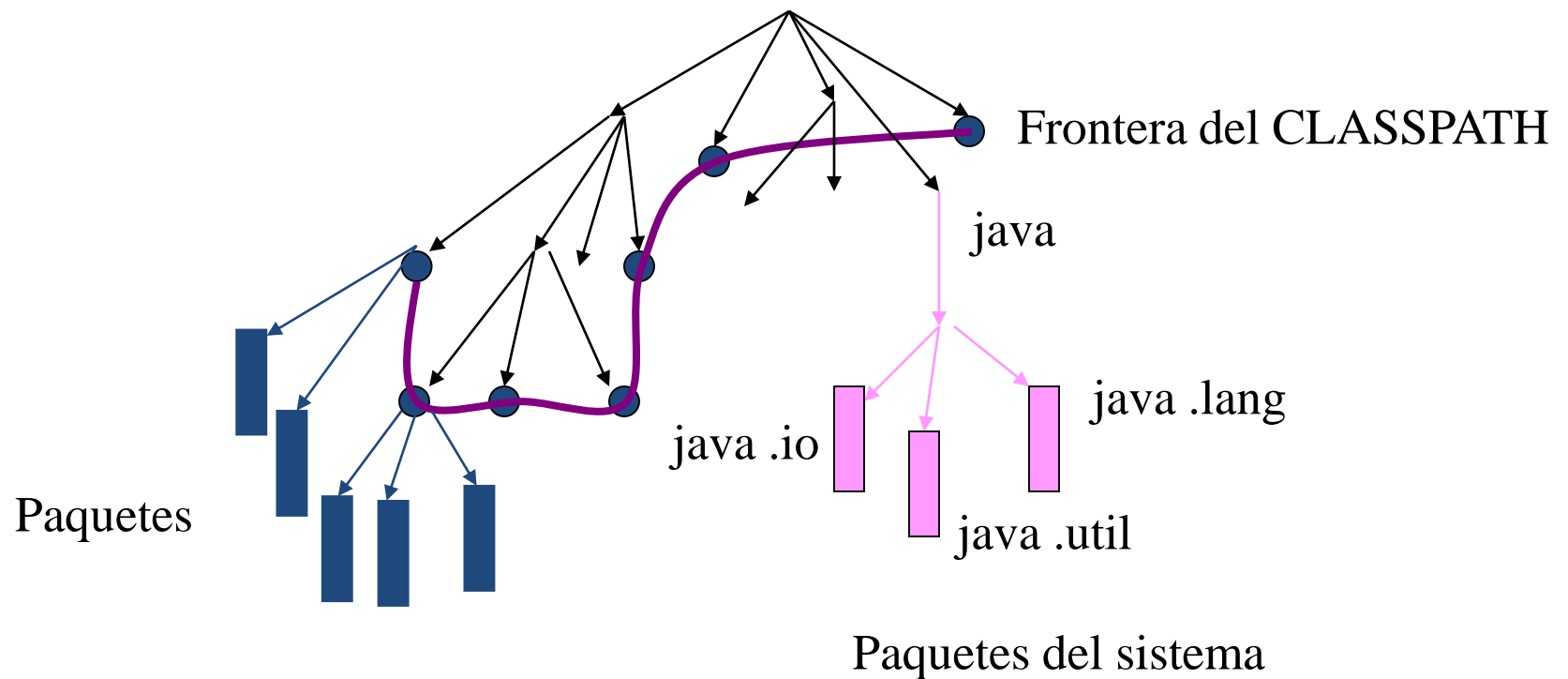
- Programa para calcular el valor medio de un millón de números generados aleatoriamente, usando las clases **Random** del paquete **java.util** y **System** del paquete **java.lang**.

```
import java.util.Random;
public class TestAleatorio {
    public static void main(String[] args) {
        Random rnd = new Random();
        double sum = 0.0;
        for (int i = 0; i < 1000000; i++) {
            sum += rnd.nextDouble();
        }
        System.out.println("media = " + sum / 1000000.0);
    }
}
```


Paquetes

- Los paquetes del sistema cuelgan de varios subdirectorios específicos:
 - `.../java`
 - `.../javax`
- Por defecto el compilador busca un paquete necesario entre los del sistema y en el directorio raíz del Proyecto creado para alojar un programa.
- Para que busque paquetes en otro sitio hay que indicárselo en la variable del entorno **CLASSPATH**
- En las sesiones de prácticas, si fuera necesario, ya se explicará cómo hacer esto en el IDE utilizado

Paquetes



Contenido

- Introducción histórica
- Programas y Paquetes
- Clases y Objetos 
- Elementos del Lenguaje:
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones
- Cadenas de caracteres
- Arrays
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces

Clases en Java

“Punto.java”

```
public class Punto {  
    private double x, y;  
    public Punto() { this(0,0); }  
    public Punto(double a, double b) { x = a; y = b; }  
    public double abscisa() {return x;}  
    public double ordenada() {return y;}  
    public void abscisa(double a) { x = a; }  
    public void ordenada(double b) { y = b; }  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public double distancia(Punto pto) {  
        return Math.sqrt(Math.pow(x - pto.x, 2) +  
            Math.pow(y - pto.y, 2));  
    }  
}
```

VARIABLES DE INSTANCIA

CONSTRUCTORES

MÉTODOS DE INSTANCIA

Clases para datos enumerados: **enum**

```
enum Semana    {Lun,Mar,Mie,Jue,Vie,Sab,Dom};

public class Ej {
    public static void main(String [] args) {
        Semana d = Semana.Lun; // acceso a un valor de Semana

        int ord = d.ordinal(); // posición de un valor de Semana

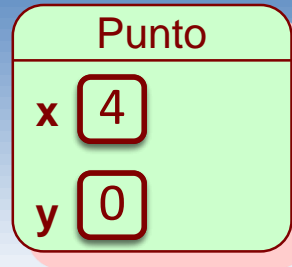
        // recorrido de los valores con un for-each
        for(Semana dia : Semana.values()) {
            System.out.print(dia + "  ");
        }

    }
}
```

Lun Mar Mie Jue Vie Sab Dom

Objetos en Java

```
public class Punto {  
    private double x, y;
```



pto

```
    public Punto(double a, double b) {  
        x = a; y = b;  
    }  
    ...  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public double distancia(Punto p) { ... }  
}
```

trasladar(3,-1)

```
Punto pto = new Punto(1,1);  
pto.trasladar(3,-1);
```

Variables y Métodos *de Instancia*

- Cada instancia (objeto) de una clase tiene sus propias variables de instancia.
 - Ej. cada punto tiene dos variables de instancia : `x`, `y`
- Las variables de instancia (o atributos) se acceden etiquetándolas con el nombre o la referencia de la instancia (**this** para el objeto implicado, aunque se puede suprimir si no hay conflicto de nombres).
 - Ej. `pto.x`
 - Ej. `this.x` (o directamente `x`)
- Una variable de instancia con la etiqueta **final** es una constante

Desde fuera de una clase, habrá que tener en cuenta el control de visibilidad (se verá más adelante)

Variables y Métodos *de Instancia*

- Todas las instancias (objetos) de una clase comparten los métodos de instancia
 - Ej. todos los puntos tienen el método: `trasladar`
- Los métodos de instancia se invocan etiquetándolos con el nombre o la referencia de la instancia (**this** para el objeto implicado, aunque se puede suprimir si no hay conflicto de nombres).
 - Ej. `p1.distancia(p2)`
 - Ej. `this.distancia(part)` (o directamente `distancia(part)`)
- Un método de instancia con la etiqueta **final** no se puede redefinir (herencia)

Desde fuera de una clase, habrá que tener en cuenta el control de visibilidad (se verá más adelante)

Variables y Métodos *de Clase*

- Las **variables de clase**
 - Existen aunque no se hayan creado objetos de la clase.
 - Son **comunes a todos los objetos** que se creen de la clase.
 - Se declaran como **static**.
 - Se acceden etiquetando sus nombres con el nombre de la clase (aunque también se pueden etiquetar con el nombre de alguna instancia).
 - Dentro de la propia clase se acceden directamente (o etiquetando sus nombres con **this**, si es necesario)
 - Con la etiqueta **final** son constantes de clase

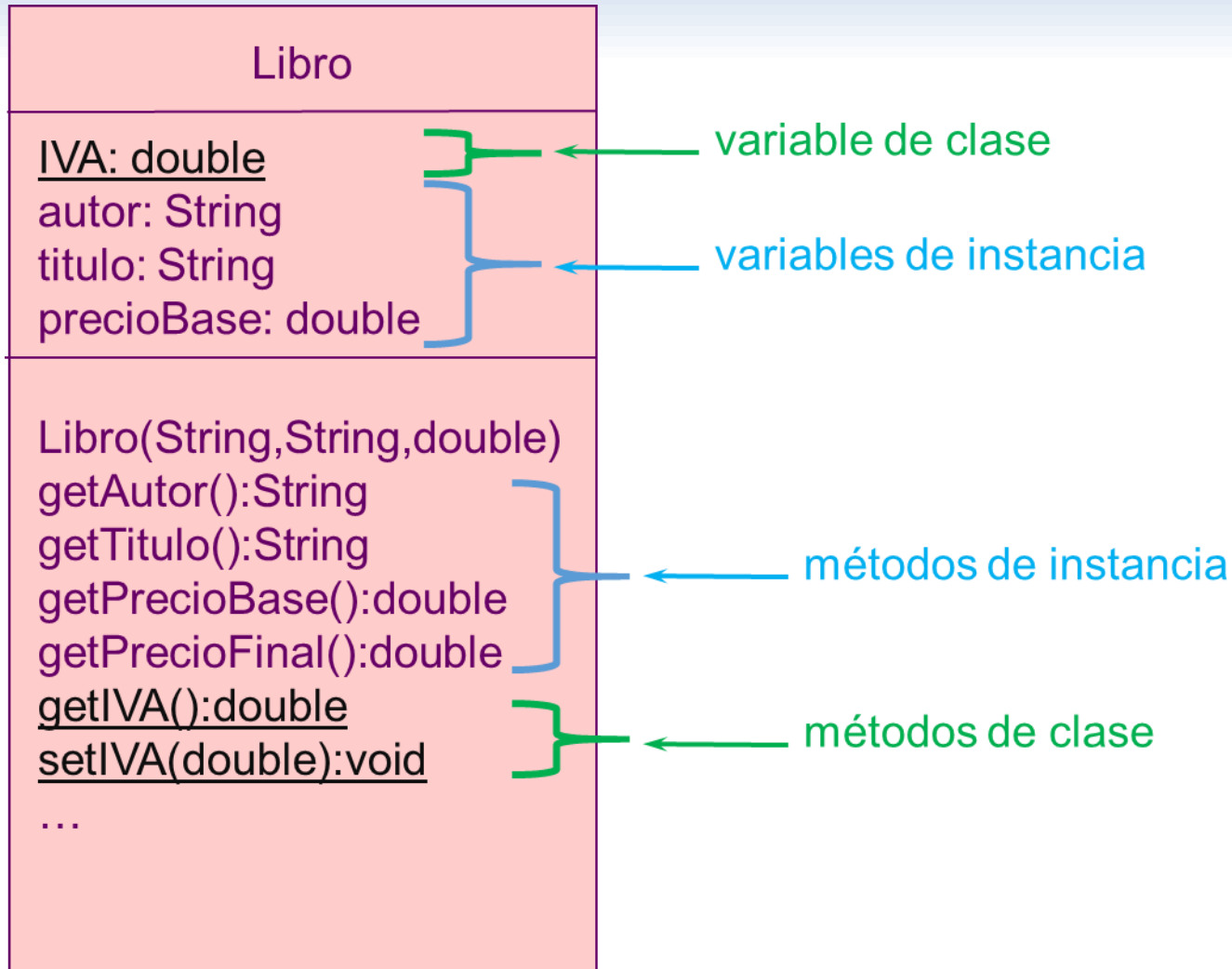
Desde fuera de una clase, habrá que tener en cuenta el control de **visibilidad** (se verá más adelante)

Variables y Métodos de Clase

- Los **métodos de clase**
 - Existen aunque no se hayan creado objetos de la clase.
 - Son comunes a todos los objetos que se creen de la clase
 - Se declaran como **static**.
 - Se invocan etiquetando sus nombres con el nombre de la clase (aunque también se pueden etiquetar también con el nombre de alguna instancia).
 - Dentro de la propia clase se invocan directamente (o etiquetando sus nombres con **this**, si es necesario)
 - Con la etiqueta **final** no se pueden redefinir (herencia)

Desde fuera de una clase, habrá que tener en cuenta el control de visibilidad (se verá más adelante)

Ejemplos de Variables y Métodos de Clase (I)



Ejemplos de Variables y Métodos de Clase

(II)

- `sqrt` y `pow` son métodos de clase de la clase **Math** y se invocan, en la clase **Punto**, precedidos por **Math**.

```
public double distancia(Punto pto) {  
    // uso de Métodos de clase  
    return Math.sqrt(Math.pow(x - pto.x, 2)  
        + Math.pow(y - pto.y, 2));  
}
```

- Otro ejemplo

```
public class Ejemplo {  
    // declaración de Método de clase  
    public static void main(String [] args) {  
        // uso de Variable de clase  
        System.out.println("Hola");  
        // uso de Método de clase  
        long time = System.currentTimeMillis();  
        ...  
    }  
}
```

Ejemplos de Variables y Métodos de Clase (III)

```
public class Vuelo {  
  
    private static int sigVuelo = 1; // Variable de Clase  
    private int localizadorVuelo;    // Variable de Instancia  
  
    public Vuelo() { // constructor  
        localizadorVuelo = sigVuelo;  
        sigVuelo++;  
    }  
    ...  
}  
  
Vuelo v1 = new Vuelo(); // localizador = 1  
Vuelo v2 = new Vuelo(); // localizador = 2  
Vuelo v3 = new Vuelo(); // localizador = 3
```

Ejemplos de Variables y Métodos de Clase (IV)

```
public class Saludo {  
  
    public static void main(String [] args) { // Método de Clase  
        saludar();  
    }  
  
    private static void saludar() { // Método de Clase  
        System.out.println("Hola");  
    }  
  
}
```

Variables y Métodos

- Los **métodos de instancia**
 - Pueden acceder tanto a variables de instancia como a variables de clase
 - Pueden invocar tanto a métodos de instancia como a métodos de clase
- Los **métodos de clase**
 - Sólo pueden acceder a variables de clase
 - Sólo pueden invocar a métodos de clase

Métodos (parámetros)

- En Java todos los parámetros son por valor
- No existe el paso por referencia (como existe por ejemplo en C++)

Control de la visibilidad

Existen cuatro niveles de visibilidad:

No aplicable a clases
(excepto anidadas)

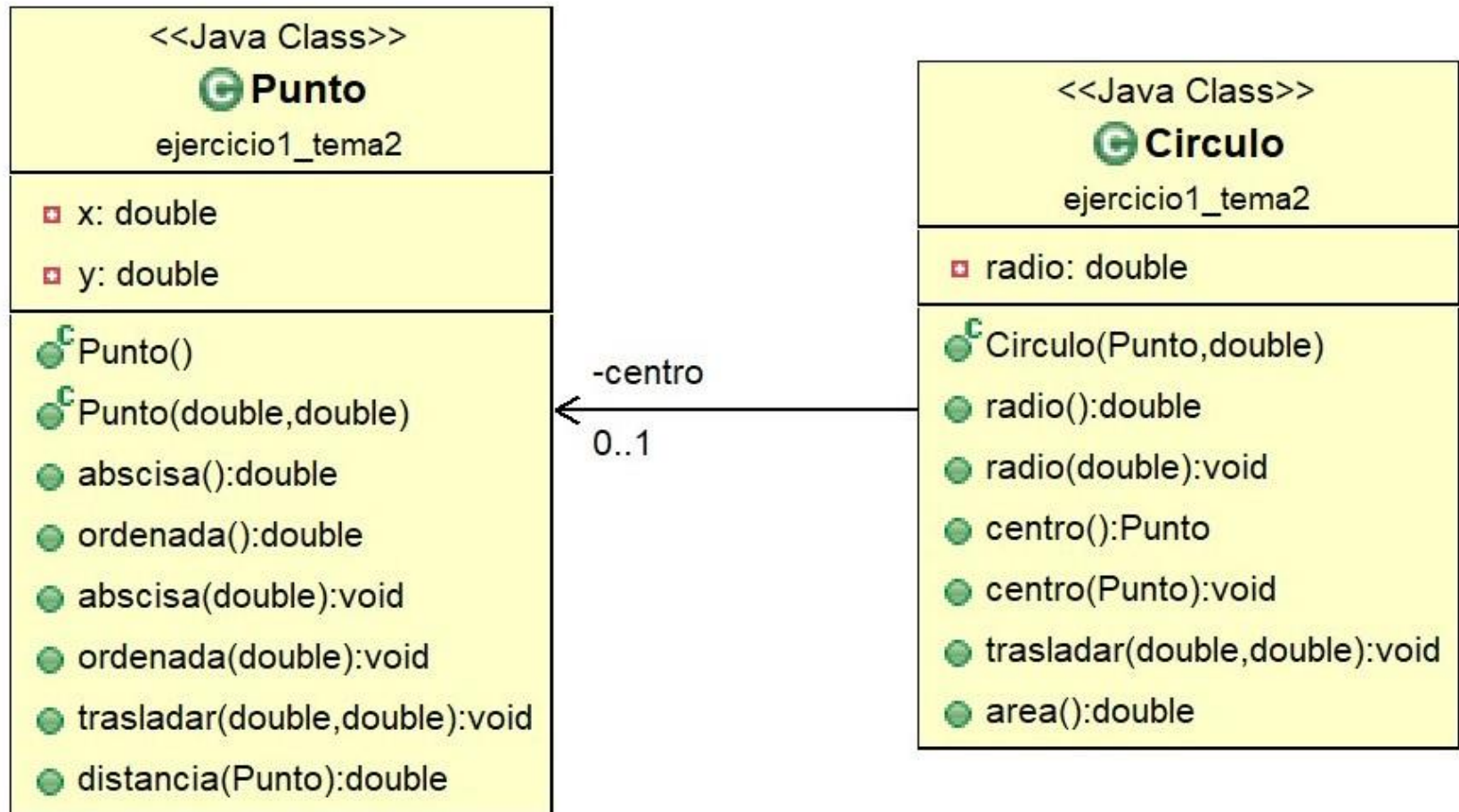
- **private** – visibilidad dentro de la propia clase
- **protected** – visibilidad dentro del paquete y de las subclases
- **public** – visibilidad desde cualquier paquete
- Por omisión – visibilidad dentro del propio paquete (package)

		Mismo paquete		Otro paquete	
		Subclase	Otra	Subclase	Otra
-	private	NO	NO	NO	NO
#	protected	SÍ	SÍ	SÍ	NO
+	public	SÍ	SÍ	SÍ	SÍ
~	package	SÍ	SÍ	NO	NO

Símbolos en diagramas UML (generados en Eclipse)



Símbolos en diagramas UML (generados en Eclipse)



La vida de los objetos

- Los objetos son siempre instancias de alguna clase.
- Durante la ejecución de un programa
 - Se crean objetos
 - Interactúan entre sí (enviándose mensajes)
 - Se eliminan los objetos no necesarios
 - La eliminación es automática (recolección de basura automática)

Creación de objetos

- Para crear un objeto (operación **new**)
 - Se debe utilizar un constructor (operación definida en la clase con su mismo nombre):
`new <constructor>(<lista args>)`
 - El constructor reserva espacio de memoria para el objeto
 - El constructor asigna unos valores iniciales a sus variables de instancia
- **new** devuelve una referencia al objeto que crea.
 - Puede asignarse a una variable
`pto = new Punto(3, 4);`
 - Puede usarse en una expresión
`pto.distancia(new Punto(2, 3));`

Constructores de objetos

- Una clase puede definir varios constructores
 - Con distinto número de argumentos o
 - Con argumentos de distintos tipos
- Si no hay ningún constructor entonces el sistema proporciona uno “por defecto” (asigna valores iniciales por defecto a las variables (se verá más adelante))
- Si hay constructores, entonces el “por defecto” no se crea

Variables de objeto

- Las variables de objeto se declaran de una determinada clase (o interfaz, como se verá más adelante)

```
Punto pto;
```

`pto`

- Almacenará una referencia a un objeto, NO un objeto
 - Con esa declaración, todavía no almacena ninguna referencia, por lo que todavía no puede recibir mensajes.
- Por medio de la asignación puede tomar una referencia a un objeto

```
pto = new Punto(3, 4);
```

- Ya puede recibir mensajes.

```
pto.trasladar(2,2);
```

- La declaración y la asignación se pueden realizar simultáneamente:

```
Punto pto = new Punto(3, 4);
```


Asignación o Copia

- Una variable que referencia a un objeto se puede **asignar** a otra de su misma clase. En tal caso se copia la referencia y ambas compartirán el mismo objeto.
- Para duplicar un objeto se debe crear otro de la misma clase y **copiar** sus variables de estado.

Asignación

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(Punto pt1, Punto pt2) {  
        origen = pt1;  
        extremo = pt2;  
    }  
    ... // Otros métodos  
    public double longitud() {  
        return origen.distancia(extremo);  
    }  
}  
  
// en otra clase hacemos  
Punto pt1 = new Punto(1,3);  
Punto pt2 = new Punto(2,5);  
Segmento sg = new Segmento(pt1,pt2);
```

Asignación

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(Punto pt1, Punto pt2) {  
        origen = pt1;  
        extremo = pt2;  
    }  
    ... //  
    public  
    return  
}
```

The diagram shows a Segmento object 'sg' with two attributes: 'origen' and 'destino'. 'origen' points to a Punto object with x=1 and y=3, which is also pointed to by 'pt1'. 'destino' points to a Punto object with x=2 and y=5, which is also pointed to by 'pt2'.

// en otra clase hacemos

```
Punto pt1 = new Punto(1,3);
```

```
Punto pt2 = new Punto(2,5);
```

```
Segmento sg = new Segmento(pt1,pt2);
```

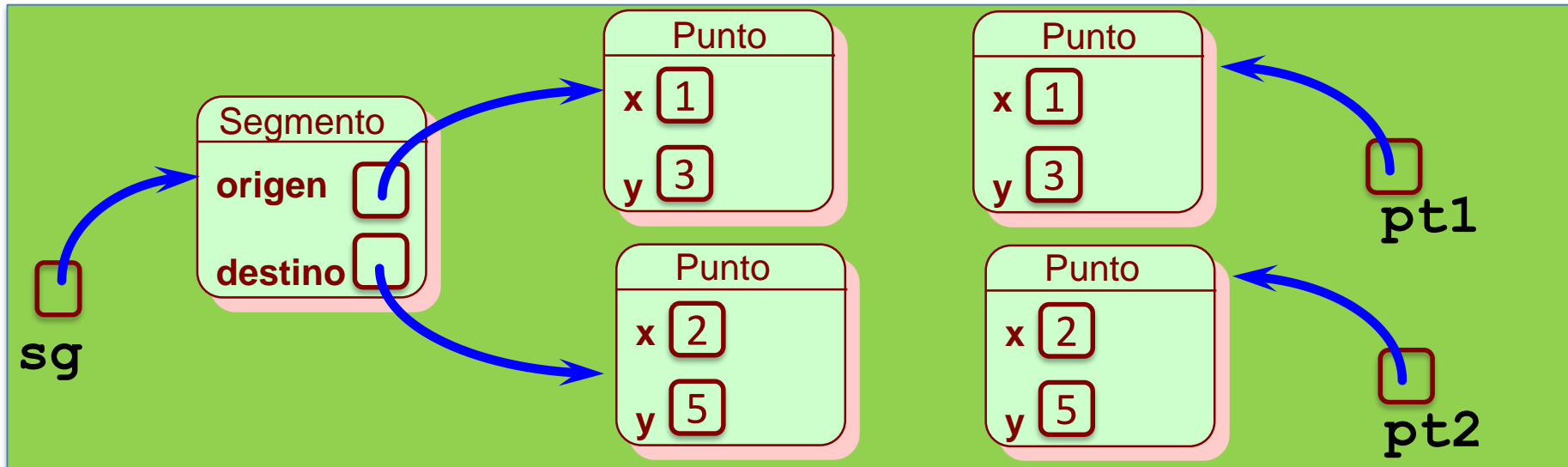
Copia

```
public class Segmento {
    private Punto origen, extremo;
    public Segmento(Punto pto1, Punto pto2) {
        origen = new Punto(pto1.abscisa(),
                             pto1.ordenada());
        extremo = new Punto(pto2.abscisa(),
                              pto2.ordenada());
    }
    ... // Otros métodos
    public double longitud() {
        return origen.distancia(extremo);
    }
}

// en otra clase hacemos
Punto pt1 = new Punto(1,3);
Punto pt2 = new Punto(2,5);
Segmento sg = new Segmento(pt1,pt2);
```

Copia

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(Punto pto1, Punto pto2) {  
        origen = new Punto(pto1.abscisa(),  
                             pto1.ordenada());  
        extremo = new Punto(pto2.abscisa(),  
                              pto2.ordenada());  
    }  
}
```




```
Punto pt1 = new Punto(1,3);  
Punto pt2 = new Punto(2,5);  
Segmento sg = new Segmento(pt1,pt2);
```

Eliminación de objetos

- La eliminación de objetos en Java se realiza de forma automática cuando se pierden las referencias a dichos objetos.
- Se puede “ayudar” a la eliminación de un objeto anulando su referencia. Ej: `pto = null;`
(en el supuesto de que no exista otra referencia a él).
- Se puede activar la eliminación automática
 - invocando el método de clase `gc ()` de la clase **System**.

Contenido

- Introducción histórica
- Programas y Paquetes
- Clases y Objetos
- Elementos del Lenguaje: 
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones
- Cadenas de caracteres
- Arrays
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces

Tipos básicos

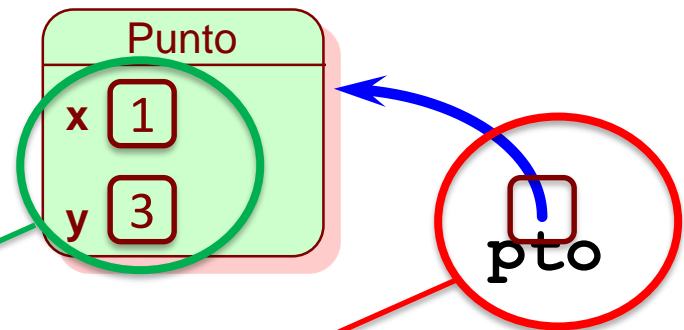
- En Java también hay *tipos básicos*.
- Sólo existen los siguientes tipos básicos:

byte (entero de 8 bits)	short (entero de 16 bits)
int (entero de 32 bits)	long (entero de 64 bits)
float (real de 32 bits)	double (real de 64 bits)
char (Unicode de 16 bits)	boolean (true , false)
- El número de bits dedicado es independiente de las plataformas sobre las que se ejecuten los programas.
- No se pueden definir más tipos básicos.

Tipos básicos frente a Clases

- Variables de tipos básicos
 - Almacenan el valor
- Variables de objetos
 - Almacenan la referencia al objeto

```
Punto pto = new Punto(1,3);
```



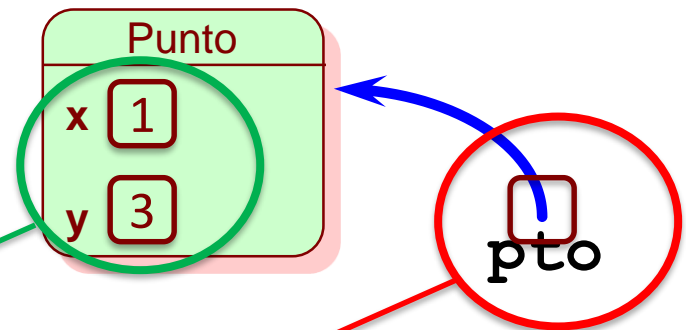
Variables de tipos básicos

Variable de objeto

Tipos básicos frente a Clases

- Esto tiene consecuencias en la manipulación de referencias y valores (por ejemplo en la asignación, comparación y el paso de parámetros).

```
Punto pto = new Punto(1,3);
```



Variables de tipos básicos

Variable de objeto

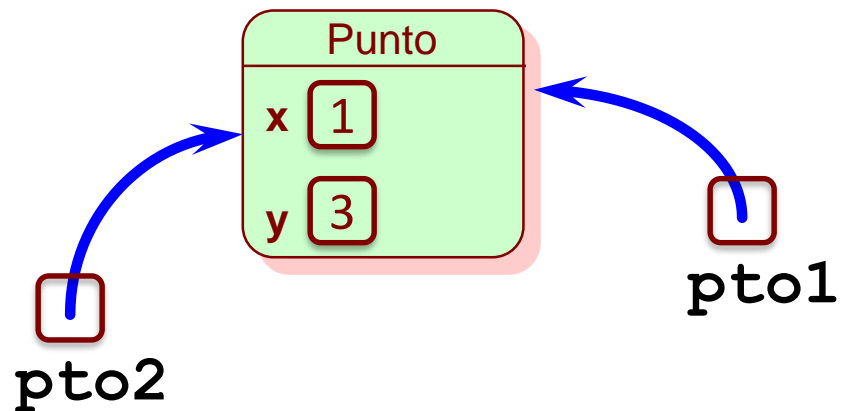
Tipos básicos frente a Clases

asignación

```
int a, b;  
b = 3;  
a = b;
```





```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = pto1;
```



Tipos básicos frente a Clases

comparación

```
int a, b;  
b = 3;  
a = b;
```

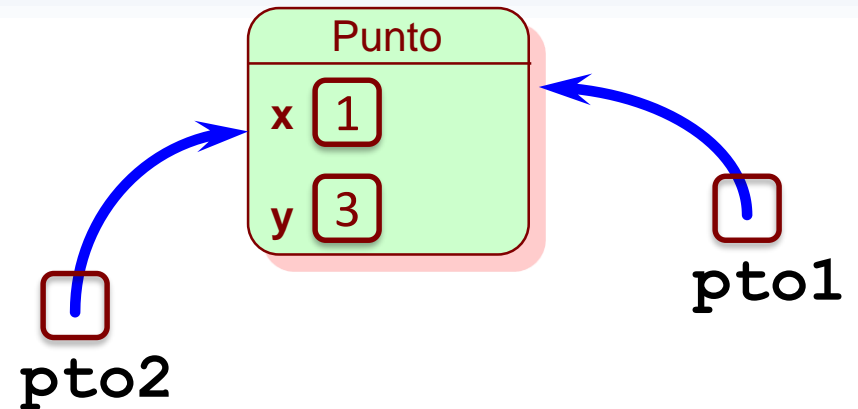
b 
a 

```
if (a == b) { // true  
    ...  
}
```

Tipos básicos frente a Clases

comparación

```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = pto1;
```



```
if (pto1 == pto2) { // true  
    ...  
}
```

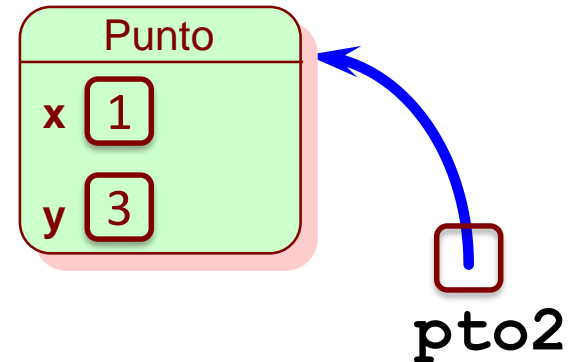
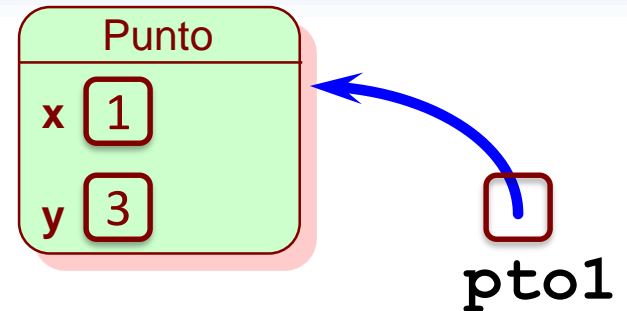
Tipos básicos frente a Clases

comparación

```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = new Punto(1,3);
```

```
if (pto1 == pto2) { // false  
    ...  
}
```

Compara referencias



Tipos básicos frente a Clases

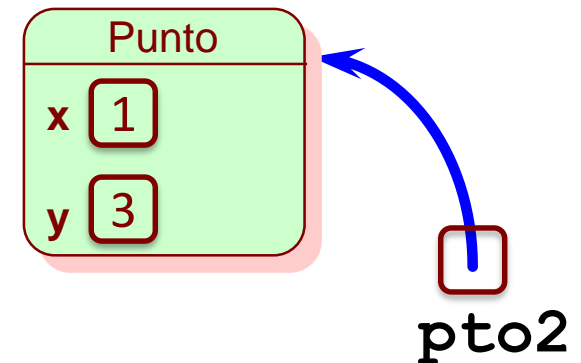
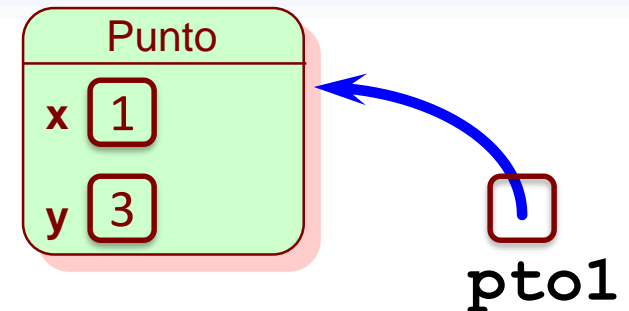
comparación

```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = new Punto(1,3);
```

Aunque sería mejor usar
`equals` (tema 4)

```
if (pto1.abscisa() == pto2.abscisa()  
    && (pto1.ordenada() == pto2.ordenada())) { // true  
    ...  
}
```

Compara contenido



Tipos básicos frente a Clases

paso de parámetros

- Recordemos que el paso de parámetros en Java es “Por Valor”, esto es, se realiza una copia del valor del parámetro real en el parámetro formal correspondiente

Tipos básicos frente a Clases

paso de parámetros

- Con los tipos básicos esto implica una copia del valor almacenado

```
int x = 3;  
...  
metodo(x);
```

3

x

```
private static void metodo(int y) {  
    ...  
}
```

3

y

Tipos básicos frente a Clases

paso de parámetros

- Cualquier modificación en el parámetro formal dentro del método no afecta al parámetro real

```
int x = 3;  
...  
metodo(x);
```

3

x

```
private static void metodo(int y) {  
    ...  
}
```

4

y

y = 4;

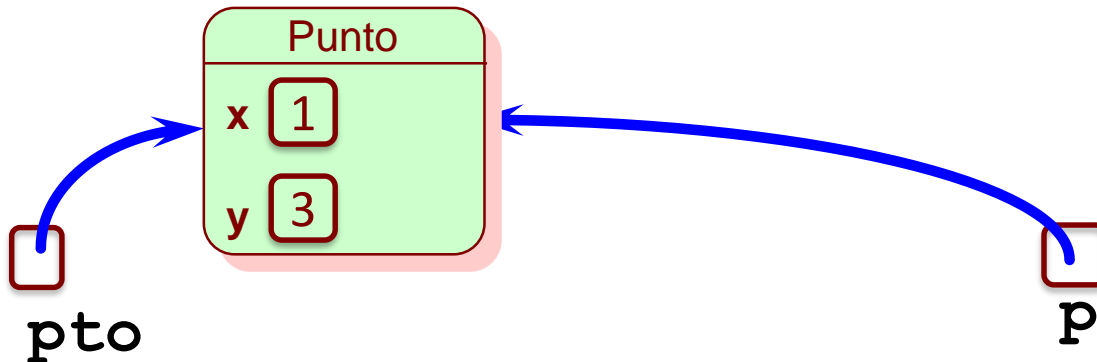
Tipos básicos frente a Clases

paso de parámetros

- En cambio, cuando tratamos con objetos, la copia es de la referencia, no del objeto referenciado

```
Punto pto =  
    new Punto(1,3) ;  
...  
metodo (pto) ;
```

```
private static void metodo(Punto p) {  
    ...  
}
```



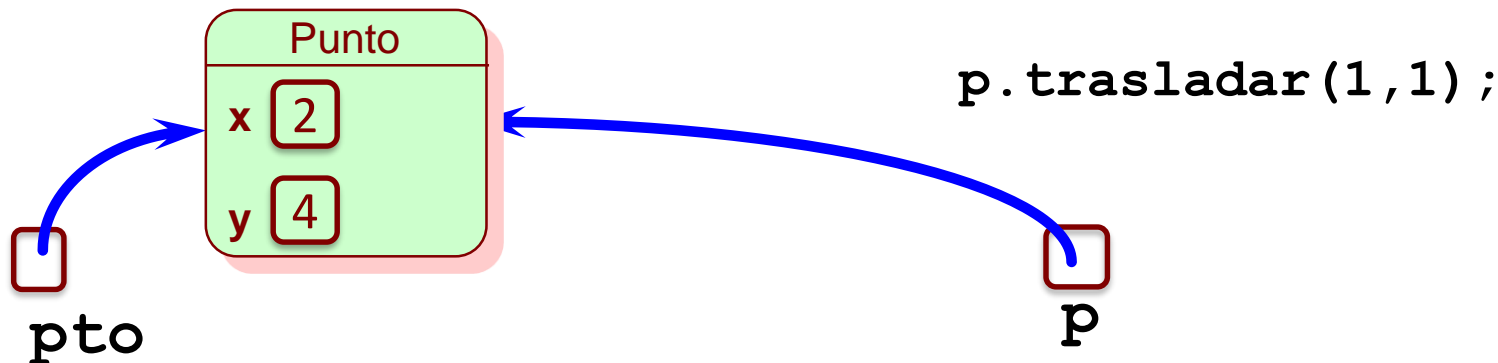
Tipos básicos frente a Clases

paso de parámetros

- Cualquier modificación en el parámetro formal dentro del método sí afecta al objeto del parámetro real

```
Punto pto =  
    new Punto(1,3);  
...  
metodo(pto);
```

```
private static void metodo(Punto p){  
    ...  
}
```



Conversiones de tipos y clases

- Se producen conversiones de tipo o de clase de forma **implícita** en ciertos contextos.

– Siempre a **tipos más amplios** siguiendo la ordenación:

byte → short → int → long → float → double
char → int

o a **clases ascendentes** en la línea de la herencia.

- Se permiten conversiones **explícitas** en sentido contrario mediante la construcción:

(<tipo/clase>) <expresión>

Sólo se comprueban durante la ejecución.

Conversiones implícitas: contextos

- La conversión implícita se produce en los siguientes contextos:
 - **Asignaciones** (el tipo de la expresión se promociona al tipo de la variable de destino)
 - **Invocaciones de métodos** (los tipos de los parámetros reales se promocionan a los tipos de los parámetros formales)
 - **Evaluación de expresiones aritméticas** (los tipos de los operandos se promocionan al del operando con el tipo más general y, como mínimo se promocionan a `int`)
 - **Concatenación de cadenas** (los valores de los argumentos se convierten en cadenas)

Variables

- Tendremos variables de tipos básicos y variables de objetos.
- Antes de usar una variable se requiere una declaración:

<tipo> <identificador>

int contador;

- Las variables se pueden inicializar mediante una sentencia de asignación:

contador = 0;

- Declaración e inicialización pueden hacerse al mismo tiempo:

int contador = 0;

Constantes

- Una variable se puede declarar como constante precediendo su declaración con la etiqueta **final**:
final int MAXIMO = 0;
- La inicialización de una constante se puede hacer en cualquier momento posterior a su declaración (salvo si son constantes de clase (**static**)).

final int MAXIMO;

...

MAXIMO = 0;

- Cualquier intento de cambiar el valor de una variable **final** después de su inicialización produce un error en tiempo de compilación.

Identificadores

- Un *identificador* (nombre) es una secuencia arbitraria de caracteres Unicode: letras, dígitos, subrayado,... . No debe comenzar por dígito ni coincidir con una palabra reservada (`class`, `private`, `int`, ...).
`int numero;`
- Los identificadores dan nombre a:
variables, métodos, clases e interfaces.
- Por convenio:
 - Nombres de variables y métodos en minúsculas. Si son compuestos, las palabras no se separan y comienzan con mayúscula a partir de la segunda.
`long valorMaximo`
 - Nombres de clase igual, pero comenzando con mayúscula.
`class ConjuntoEnteros`
 - Nombres de constantes todo en mayúsculas. Si son compuestos, las palabras se separan con subrayados.
`final int CTE_GRAVITACION`

Ámbito de una variable

- Un identificador debe ser único dentro de su ámbito.
- El **ámbito** de una variable es la zona de código donde se puede usar su identificador sin calificar.
- El ámbito determina cuándo se crea y cuándo se destruye espacio de memoria para la variable.
- Las variables, según su ámbito, se clasifican en las siguientes categorías:
 - Variable de clase o de instancia
 - Variable local
 - Parámetro de método
 - Parámetro de gestor de excepciones (catch)

Ámbitos

```
public class MiClase { ...
```

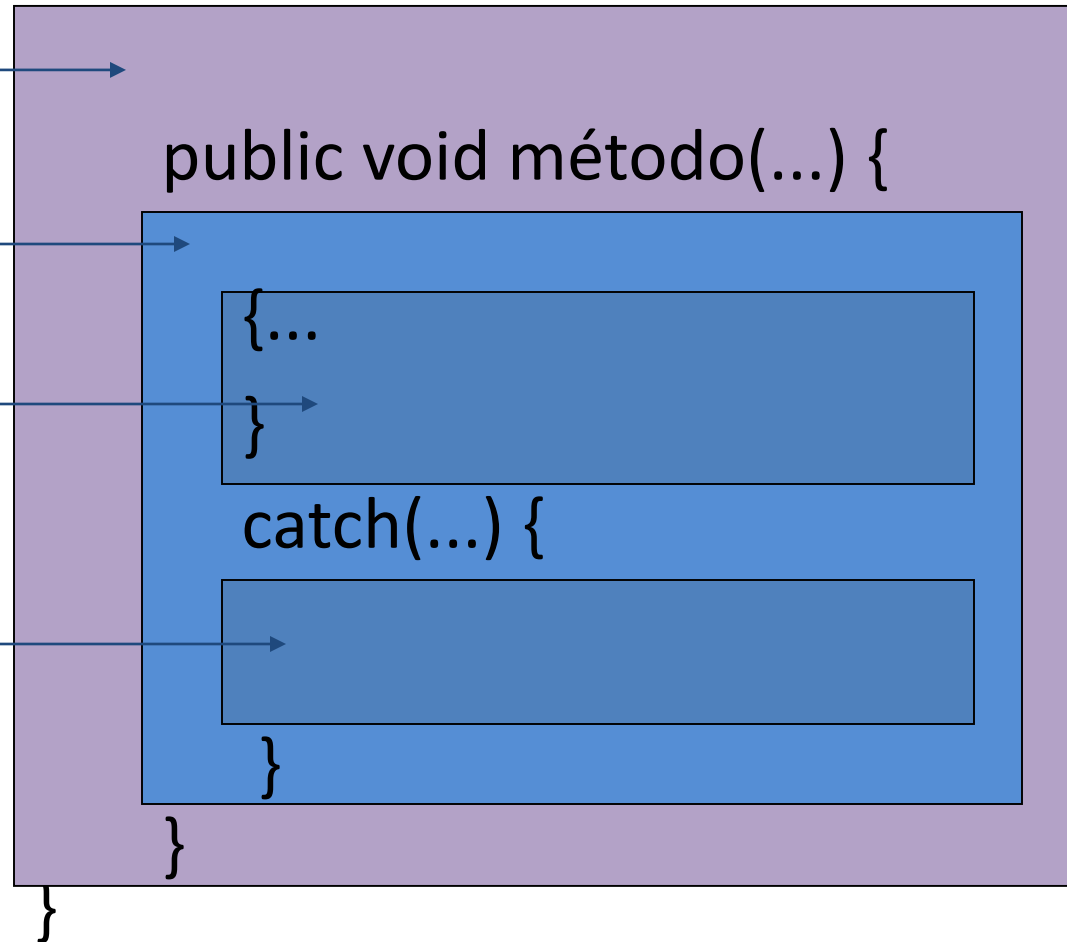
Variables de clase →

Parámetros,
variables de instancia
y variables locales

Variables locales →

Parámetros →

catch →



Ámbitos

```
public class MiClase { ...
```

Variables de clase

Parámetros,

~~variables de instancia~~
y variables locales

Variables locales

Parámetros

catch

```
static public void método(...) {
```

```
{...
```

```
}
```

```
catch(...) {
```

```
}
```

```
}
```

```
}
```

Inicialización de variables

- Cuando no se les asigna un valor explícitamente
 - Las **variables de clase** se inicializan automáticamente al cargar la clase.
 - Las **variables de instancia** se inicializan automáticamente cada vez que se crea una instancia.
 - Las **variables locales** no se inicializan de forma automática y el compilador produce un error.

- Valores de inicialización automática:

```
boolean (false) char ('\u0000') int (0)
double (+0.0D) objetos (null)
```

Expresiones

- Una *expresión* es una combinación de
 - literales,
 - variables,
 - operadores y
 - mensajes,

acorde con la sintaxis del lenguaje, que *se evalúa*

- a un valor simple o
- a una referencia a un objeto y

devuelve el resultado obtenido.

Operadores (I)

- Un *operador* es una función de uno, dos o tres argumentos.
- Existen operadores
 - aritméticos $(+_ , -_ , _{++} , _{--} , ++_ , --_)$
 $(+_ , -_ , *_ , _/_ , _\%_)$
 - de relación/comparación $(> , >= , < , <= , == , !=)$
 - lógicos $(\&\& , || , !)$
 - de asignación $(= , += , -= , *= , /= , \% =)$
 - para la manipulación de bits
 - Otros operadores: $(_?_:_)$ new instanceof

Operadores (II)

- Con un operador y sus argumentos se construyen expresiones simples.

`3 * 5`

`x += 7.3`

`'a' <= 45`

- Las expresiones simples se pueden combinar dando lugar a expresiones compuestas.


`3 * 5 + x / 7.3`

`y *= x / 7.3`

- El orden de evaluación de las expresiones compuestas depende de la precedencia y de la asociatividad de los operadores que aparezcan

Precedencia de operadores

- Precedencia (en sentido decreciente)



```
var++  var--  
++var  --var  !  
new    (tipo)exp  
*      /      %  
+      -  
<      >      <=     >=  instanceof  
==     !=  
&&  
||  
=  +=  -=  *=  ...
```

Asociatividad de operadores

- **Asociatividad**

- Todos los operadores binarios (excepto la asignación) a igualdad de precedencia, asocian por la izquierda.
- La asignación asocia por la derecha

Bloques

- Un bloque es un grupo de cero o más sentencias, encerradas entre llaves, dando lugar a una sentencia compuesta.

```
{  
    <sentencia1>  
    . . .  
    <sentenciaN>  
}
```

- Un bloque se puede usar en cualquier parte donde se necesite una sentencia simple.

Instrucciones/sentencias

- Existen tres clases de instrucciones o sentencias:
 - Sentencias de expresión – Se obtienen terminando en ‘;’ alguna de las expresiones siguientes:
 - asignaciones
 - incrementos/decrementos ++/--
 - mensajes
 - creaciones de objeto
 - Sentencias de declaración de variables
 - Sentencias de control

Sentencias de declaración

- Las sentencias de declaración de variables tienen la forma: <tipo/clase> <variable>
`int x;`
- Las declaraciones de variables del mismo tipo/clase pueden agruparse:
`int x, y, z;`
- Las sentencias de declaración pueden agruparse con las de asignación a las mismas variables:
`int x = 5, y = 12, z = 213;`

Sentencias de control

Las sentencias de control del flujo de ejecución se agrupan en:

- sentencias de iteración
- sentencias de selección
- sentencias para el control de excepciones
- sentencias de salto/ramificación

Sentencias de iteración

```
while (<exp. booleana>
    <sentencias>
```

```
do
```

```
    <sentencias>
```

```
while (<exp. booleana>);
```

```
for (<exp1>; <exp. bool>; <exp2>)
    <sentencias>
```

Existe una sintaxis de **for** especial para arrays y colecciones.

Sentencias de selección (I)

```
if (<exp. bool>) <sentencias>
```

```
if (<exp. bool>) <sentencias1>  
else <sentencias2>
```

```
if (<exp. bool1>) <sentencias1>  
else if (<exp. bool2>) <sentencias2>  
...  
else <sentenciasN>
```


```
<exp bool> ? <exp1> : <exp2>
```


Sentencias de selección (II)

```
switch (<exp>) {  
    case <altern1>: <sent1>; break;  
    case <altern2>: <sent2>; break;  
    ...  
    case <alternk>: <sentk>; break;  
    default: <sentD>; break;  
}
```

- <exp> debe ser de tipo `char`, `byte`, `short` o `int`
- A partir de Java 1.7 también se admiten cadenas de caracteres (`String`)

Contenido

- Introducción histórica
- Programas y Paquetes
- Clases y Objetos
- Elementos del Lenguaje:
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones 
- Cadenas de caracteres
- Arrays
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces

Control de errores, excepciones (I)

- Mecanismo de ayuda para la comunicación y el manejo de errores
- Cuando se produce un error en un método:
 1. Se crea un objeto (el sistema o el programador con `new`) de la clase **Exception**, **RuntimeException** (de `java.lang`) o de alguna clase heredera de ellas, con información sobre el error (normalmente una cadena de caracteres como parámetro)
 2. Se lanza (el sistema o el programador mediante la instrucción `throw`) dicha excepción (objeto)

Por ejemplo:

```
throw new RuntimeException("Error...") ;
```

3. Se interrumpe el flujo normal de ejecución
4. El entorno de ejecución intenta encontrar un tratamiento (captura, **catch**) para dicho objeto.
 1. dentro del propio método o
 2. en alguno de los anteriores en las sucesivas invocaciones
5. Si no se encuentra un tratamiento, el programa finaliza con error

Control de errores, excepciones (II)

Existen tres sentencias relacionadas con el tratamiento de excepciones:

- **try**

delimita un bloque de instrucciones donde se puede producir una excepción,

- **catch**

identifica un bloque de código asociado a un bloque **try** donde se trata un tipo particular de excepción,

- **finally**

identifica un bloque de código que se ejecutará después de un bloque **try** con independencia de que se produzcan o no excepciones.

Control de errores, excepciones (III)

El aspecto normal de un segmento de código con control de excepciones sería el siguiente:


```
try {  
  <sentencia/s>  
} catch (<tipoexcepción> <identif>) {  
  <sentencia/s>  
}  
...  
} catch (<tipoexcepción> <identif>) {  
  <sentencia/s>  
} finally {  
  <sentencia/s>  
}
```

The diagram illustrates the components of a try-catch-finally block with callouts:

- A red callout labeled "Bloque vigilado" (Monitored block) points to the code inside the `try` block.
- A green callout labeled "Manejador" (Handler) points to the code inside the first `catch` block.
- Another green callout labeled "Manejador" (Handler) points to the code inside the second `catch` block.
- A blue callout labeled "Siempre se ejecuta" (Always executes) points to the code inside the `finally` block.

El tema 3 estará dedicado a Excepciones

Contenido

- Introducción histórica
- Programas y Paquetes
- Clases y Objetos
- Elementos del Lenguaje:
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones
- Cadenas de caracteres 
- Arrays
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces

Cadenas de caracteres

- Las cadenas de caracteres literales constantes se representan en Java como secuencias de caracteres Unicode encerradas entre comillas:

`"Ejemplo de cadena de caracteres"`

- La clase **String** (de `java.lang`) dispone de constructores y métodos para crear y manipular cadenas
- Los objetos de esta clase se pueden inicializar...

- de la forma normal:

`String str = new String("¡Hola!");`

- de la forma simplificada:

`String str = "¡Hola!";`

Cadenas de caracteres

- Los métodos más básicos de String son:
 - length() – devuelve el número de caracteres de la cadena
 - charAt(i) – devuelve el carácter de la posición i en la cadena (el primer carácter ocupa la posición 0)
- Los objetos de la clase **String** en Java son **objetos inmutables** (**En el tema 4 se verá otra clase para manipular cadenas como objetos mutables**)
- Si se intenta acceder a una posición no válida el sistema lanza una excepción:
 - IndexOutOfBoundsException
 - StringIndexOutOfBoundsException

Cadenas de caracteres

- En la concatenación pueden intervenir otros tipos de datos

```
int i = 42;  
System.out.println("i es " + i);
```

SALIDA: i es 42

- Si no son tipos básicos ...

```
Punto p = new Punto(3,4);  
  
System.out.println("p es " + p);
```

SALIDA: p es Punto@119c0982

Cadenas de caracteres

- Para evitar esto hay que incluir en las clases el método `toString()`

```
class Punto {  
    ...  
    @Override  
    public String toString() {  
        return "(" + x + "," + y + ")";  
    }  
}
```

- Ahora

```
Punto p = new Punto(3,4);
```

```
System.out.println("p es " + p);
```

```
SALIDA:                p es (3,4)
```

Cadenas de caracteres


- Comparación de cadenas:
 - `c1.equals(c2)` – devuelve `true` si `c1` y `c2` son iguales y `false` en otro caso.
 - `c1.equalsIgnoreCase(c2)` – Igual que la anterior, pero sin tener en cuenta las diferencias por mayúsculas y minúsculas.
 - `c1.compareTo(c2)` – devuelve un entero menor, igual o mayor que cero cuando `c1` es menor, igual o mayor que `c2`.
 - `c1.compareToIgnoreCase(c2)` – Igual que la anterior, pero sin tener en cuenta las diferencias por mayúsculas y minúsculas.
- ¡ojo!
 - `c1 == c2`, `c1 != c2`, ... (operadores relacionales) comparan variables referencia

Cadenas de caracteres

- Otros métodos:
 - `indexOf(...)` – primera posición de ... (carácter o cadena)
 - `lastIndexOf(...)` – última posición de ... (carácter o cadena)
 - `toLowerCase()` – nuevo String con todo minúsculas
 - `toUpperCase()` – nuevo String con todo mayúsculas
 - `substring(inicio,fin)` – nuevo String con caracteres a partir de posición inicio y hasta posición fin (sin incluirlo)
 - ...

En el Tema 4 veremos más

Contenido

- Introducción histórica
- Programas y Paquetes
- Clases y Objetos
- Elementos del Lenguaje:
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones
- Cadenas de caracteres
- Arrays 
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces

Arrays

- Un array es un objeto que almacena una estructura de tamaño fijo con componentes de un mismo tipo o clase.
- Con una sintaxis particular:

- Declaración

```
int [] listaEnteros;
```

```
Punto [] listaPuntos;
```

- Inicialización

```
listaEnteros = new int[10];
```

```
listaPuntos = new Punto[t];
```

```
char[] vocales = {'a', 'e', 'i', 'o', 'u'};
```

```
Punto p = new Punto(1, 1);
```

```
Punto[] ap = {new Punto(2, 2), p, null};
```

- El tamaño del array se guarda en una variable de instancia **length** que sólo se puede consultar.
- Los componentes de un array se inicializan con los valores por defecto (excepto al usar arrays literales)

Arrays

componentes

array

nombre variable

- Un array es un objeto que almacena una estructura de tamaño fijo con componentes de un mismo tipo o clase.
- Con una sintaxis particular:

➤ Declaración

`int [] listaEnteros;`

`Punto [] listaPuntos;`

➤ Inicialización

`listaEnteros = new int[10];`

`listaPuntos = new Punto[t];`

`char[] vocales = {'a', 'e', 'i', 'o', 'u'};`

`Punto p = new Punto(1, 1);`

`Punto[] ap = {new Punto(2, 2),`

componentes

tamaño

array literal

- El tamaño del array se guarda en una variable de instancia **length** que sólo se puede consultar.
- Los componentes de un array se inicializan con los valores por defecto (excepto al usar arrays literales)

Arrays

- Los arrays siempre comienzan por la posición 0

```
for (int i = 0; i < listaEnteros.length; i++){  
    listaEnteros[i] = i;  
}
```

```
for (int i = 0; i < listaPuntos.length; i++){  
    listaPuntos[i] = new Punto(i, i);  
}
```

```
String[] cadenas = {"CAD1", "CAD2", "CAD3"};  
for (int i = 0; i < cadenas.length; i++) {  
    System.out.println(cadenas[i].toLowerCase());  
}
```

- Si se intenta acceder a una posición no válida el sistema lanza una excepción:
 - `ArrayIndexOutOfBoundsException`

Arrays

- Para recorrer un array se pueden utilizar las estructuras iterativas ya conocidas :
 - `while`
 - `do while`
 - `for`
- Además, es posible utilizar una sintaxis alternativa (para acceder a todos los elementos), denominada “`for-each`”

```
for (int e : listaEnteros) {  
    System.out.println(e);  
}  
// e va tomando una copia de cada elemento del array
```

Arrays

- Para mostrar el contenido de un array por pantalla se puede utilizar el método de clase **toString** de la clase **Arrays** (de java.util)

```
public static  
    String toString(TipoBase[] arrayOriginal)
```

Ejemplo:

El siguiente segmento de código

```
int [] ar = {1,3,5,7,9};  
System.out.println(Arrays.toString(ar));
```

mostrará en la salida estándar

```
[1, 3, 5, 7, 9]
```

Arrays

- La clase **Arrays** (de java.util) tiene un método para comprobar si dos arrays son iguales:

```
boolean equals(Tipo[] a1, Tipo[] a2)
```

- Tiene otro método para comprobar si un array (ordenado) contiene un determinado elemento:

```
int binarySearch(Tipo[] a, Tipo valor)
```

- También tiene un método para ordenar un array:

```
void sort(Tipo[] a)
```

- Uso:

- `if (Arrays.equals(array1,array2)) {...}`
- `int indice=Arrays.binarySearch(array,v) ;`
- `Arrays.sort(array) ;`

Copia de un array a otro

- Para la copia eficiente de componentes de un array a otro Java tiene el método de clase **arraycopy** de la clase **System** (de java.lang)
- El array de destino debe existir previamente

```
public static  
    void arraycopy(Object arrayOrigen,  
                   int posOrigen,  
                   Object arrayDestino,  
                   int posDestino,  
                   int numCompCopia)
```

Ejemplo

```
char[] arrayOrigen =  
    { 'd', 'e', 's', 'c', 'a', 'f', 'e', 'i', 'n', 'a', 'd', 'o' };  
char[] arrayDestino = new char[7];  
  
System.arraycopy(arrayOrigen, 3, arrayDestino, 0, 7);  
  
arrayDestino contendrá: { 'c', 'a', 'f', 'e', 'i', 'n', 'a' }
```

Duplicación de un array

- Se puede utilizar el método de clase **copyOf** de la clase **Arrays** (de java.util) para crear un nuevo array a partir de otro.
- Si la nueva longitud especificada para el duplicado es diferente de la del array original:
 - Elimina elementos si *nuevaLongitud* es menor
 - Añade valores por defecto si *nuevaLongitud* es mayor

```
public static  
    TipoBase[] copyOf(TipoBase[] arrayOriginal,  
                      int nuevaLongitud)
```

Ejemplo

```
int[] arrayOrigen = {12,4,8,3,5,203,28};
```

1) `int[] arrayDestino = Arrays.copyOf(arrayOrigen,7);`
arrayDestino contendrá: {12,4,8,3,5,203,28}

2) `int[] arrayDestino = Arrays.copyOf(arrayOrigen,3);`
arrayDestino contendrá: {12,4,8}

3) `int[] arrayDestino = Arrays.copyOf(arrayOrigen,10);`
arrayDestino contendrá: {12,4,8,3,5,203,28,0,0,0}

Duplicación de un array

- ¿Qué ocurre en el siguiente código?

```
int[] array = {12,4,8,3,5,203,28};  
...  
array = Arrays.copyOf(array,10);
```


Duplicación de un array

- Existe otro método de clase **copyOfRange** de la clase **Arrays** (de java.util) para crear un nuevo array a partir de los valores de un determinado subrango de otro.

```
public static  
TipoBase[] copyOfRange(TipoBase[] arrayOriginal,  
                        int desde, int hasta)
```

Ejemplo

```
int[] arrayOrigen = {12,4,8,3,5,203,28};
```

```
int[] arrayDestino =  
    Arrays.copyOfRange(arrayOrigen,2,6);
```

arrayDestino contendrá: {8,3,5,203}

Arrays

- El método **main** de la Clase Distinguida recibe como argumento un array de String:

```
public static void main(String[] args)
```

- ¿Cómo se pasan argumentos en Eclipse?

```
public class Saludo {  
    public static void main(String[] args) {  
        System.out.println("Hola " + args[0] +  
                           " y " + args[1]);  
        ...  
    }  
}
```


Arrays

- El método **main** de la Clase Distinguida recibe como argumento un array de String:

```
public static void main(String[] args)
```

- ¿Cómo se pasan argumentos en Eclipse?
- ¿Cómo se convierten a int, double, ...?

```
public class PruebaJarra {  
    public static void main(String[] args) {  
        Jarra j1 = new Jarra(Integer.parseInt(args[0]));  
        Jarra j2 = new Jarra(Integer.parseInt(args[1]));  
        ...  
    }  
}
```



Ya se verán de forma detallada en el tema 4

Array Multidimensionales

- Las componentes de un array pueden ser arrays.
- Podemos tener arrays de distintas dimensiones.

```
double [][] a = new double [3][5];
```

```
double [][][] b = new double [3][5][4];
```

- El uso más común de arrays multidimensionales es el array de dos dimensiones

Listas o Secuencias de elementos

- Los **arrays** nos permiten manipular listas o secuencias de elementos del mismo tipo, pero tienen algunos **inconvenientes**:
 - Si el número de elementos no es fijo, hay que controlar en todo momento cuántas celdas están ocupadas (contador).
 - Si el array se llena y deseamos almacenar más elementos, hay que gestionar una “duplicación” del array.
 - Si deseamos insertar un elemento en una determinada posición de la lista, manteniendo el orden de los elementos, hay que gestionar un desplazamiento de elementos.
 - Si deseamos borrar un elemento en una determinada posición de la lista, manteniendo el orden de los elementos, hay que gestionar un desplazamiento de elementos.
 - Debemos diseñar operaciones de búsqueda.
 -

Listas o Secuencias de elementos

- Una buena **alternativa** al uso de arrays para manipular listas o secuencias de elementos del mismo tipo son las propias **Listas** que nos ofrece el lenguaje Java.
- En el Tema 6 se profundizará más, no sólo en Listas, sino en otros tipos de Colecciones (Conjuntos, Colas) y Correspondencias.
- En este Tema 2 adelantaremos el uso de un tipo concreto de Listas: **ArrayList**. Veremos algunas operaciones que ofrecen, dejando otras para más adelante.

ArrayList<T>

- Hay que importar la clase:

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

- Para declarar una variable lista:

```
ArrayList<T> lista;
```

donde T es el tipo de los elementos de la lista

- T no puede ser un tipo básico (int, double, char,...). Para poder almacenar elementos de tipos básicos (enteros, reales, caracteres, ...), hay que usar las clases “envoltorios” correspondientes (Integer, Double, Character, ...), que se verán más exhaustivamente en el Tema 4.

ArrayList<T>

- Ejemplos de declaraciones de listas:

```
ArrayList<String> listaCadenas;
```

```
ArrayList<Integer> listaEnteros;
```

```
ArrayList<Punto> listaPuntos;
```

ArrayList<T>

- Para crear una lista (con un tamaño inicial predefinido):

```
lista = new ArrayList<T>();
```

- Para crear una lista (con un tamaño inicial deseado):

```
lista = new ArrayList<T>(tam);
```

- En ambos casos, se puede omitir el tipo de los elementos (ya se especificó al declarar la variable):

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

```
lista = new ArrayList<>(tam);
```

- Una lista se puede crear al mismo tiempo que se declara la variable:

```
ArrayList<T> lista = new ArrayList<>();
```

ArrayList<T>

- Ejemplos de creaciones de listas:

```
listaCadenas = new ArrayList<String>();  
listaEnteros = new ArrayList<Integer>();  
listaPunto = new ArrayList<Punto>();
```

O bien:

```
listaCadenas = new ArrayList<>();  
listaEnteros = new ArrayList<>();  
listaPunto = new ArrayList<>();
```

ArrayList<T>

- Operaciones/métodos de ArrayList<T>:

int size();

boolean isEmpty();

void clear();

boolean add(T element);

T get(**int** index);

T set(**int** index, T element);

void add(**int** index, T element);

T remove(**int** index);

ArrayList<T>

- Operaciones/métodos de ArrayList<T>:

int indexOf(T element);

boolean contains(T element);

- Estas 2 operaciones de búsqueda sólo las podremos usar por ahora con listas de elementos de tipos básicos (enteros, reales, caracteres, ...) y de cadenas.
- A partir del Tema 4 (cuando veamos el método “equals”) ya sí se podrán usar para elementos de cualquier tipo (básicos y objetos).

ArrayList<T>

- Operaciones/métodos de ArrayList<T>:

boolean remove(T element);

- Esta operación de eliminación sólo la podremos usar por ahora con listas de elementos de tipos básicos (enteros, reales, caracteres, ...) y de cadenas.
- A partir del Tema 4 (cuando veamos el método “equals”) ya sí se podrán usar para elementos de cualquier tipo (básicos y objetos).

ArrayList<T>

- Operaciones/métodos de ArrayList<T>:

boolean remove(T element);

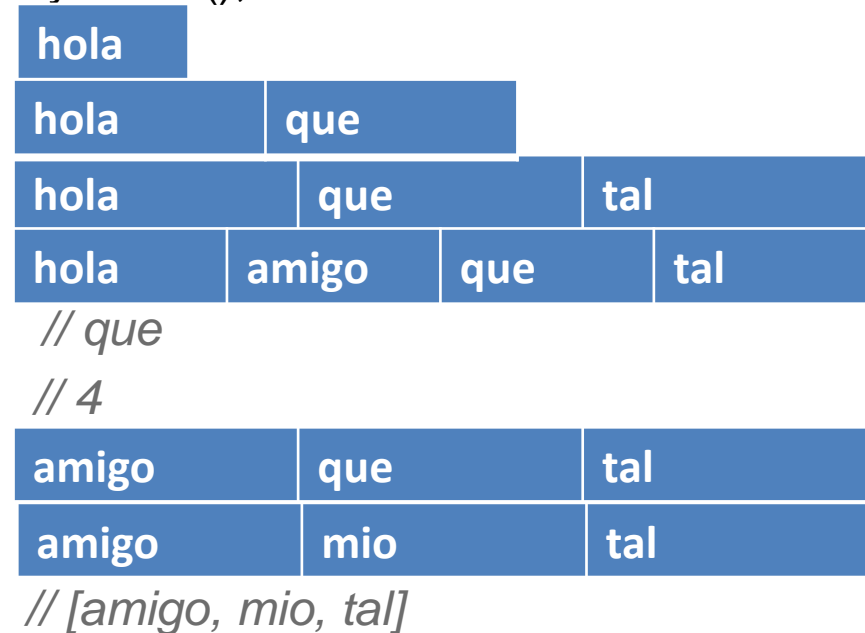
- Además, para poder usarla con listas de números enteros, el número entero que se especifique en el parámetro hay que convertirlo a objeto de la clase Integer. Con el resto de tipos básicos y con los objetos en general, no hay que hacer nada.

Ejemplo: `listaEnteros.remove((Integer) 3);`

ArrayList<T>

- Ejemplo:

```
ArrayList<String> listaCadenas = new ArrayList<>();  
listaCadenas.add("hola");  
listaCadenas.add("que");  
listaCadenas.add("tal");  
listaCadenas.add(1, "amigo");  
System.out.println(listaCadenas.get(2));  
System.out.println(listaCadenas.size());  
listaCadenas.remove(0);  
listaCadenas.set(1, "mio");  
System.out.println(listaCadenas);
```



La clase ArrayList<T> tiene implementado toString()

ArrayList<T>

- Al igual que ocurre con un array:
 - ✓ Una lista se puede recorrer utilizando las estructuras iterativas:
 - **while**
 - **do while**
 - **for**

```
int suma = 0;
for (int i = 0; i < listaEnteros.size(); i++) {
    suma += listaEnteros.get(i);
}
```

ArrayList<T>

- Al igual que ocurre con un array:
 - ✓ Una lista se puede recorrer utilizando “for-each”

```
int suma = 0;
for (int e : listaEnteros) {
    suma += e;
}
```

Contenido

- Introducción histórica
- Programas y Paquetes
- Clases y Objetos
- Elementos del Lenguaje:
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones
- Cadenas de caracteres
- Arrays
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces

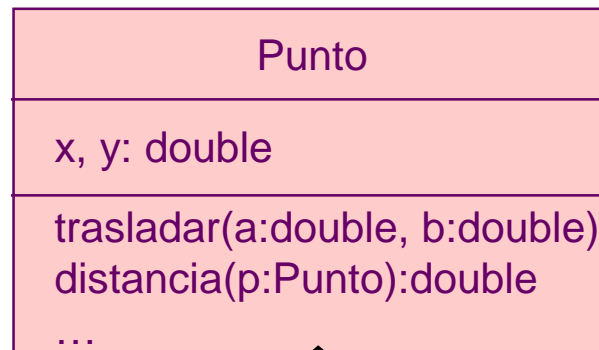


Herencia

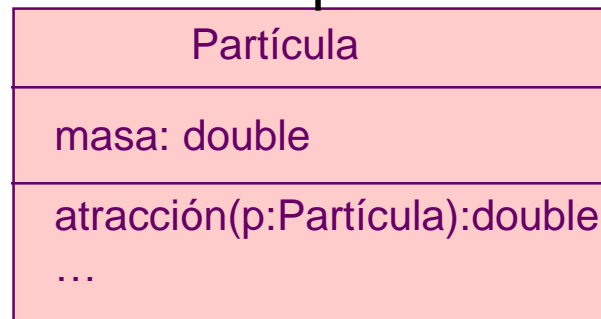
- Se puede definir una clase (*subclase*) que *hereda* estado y comportamiento de otra clase (*superclase*) a la que amplía (añade estado y comportamiento), en la forma:

```
class Subclase extends Superclase {
```

```
...  
}
```



Superclase



↑ Otra forma

Subclase

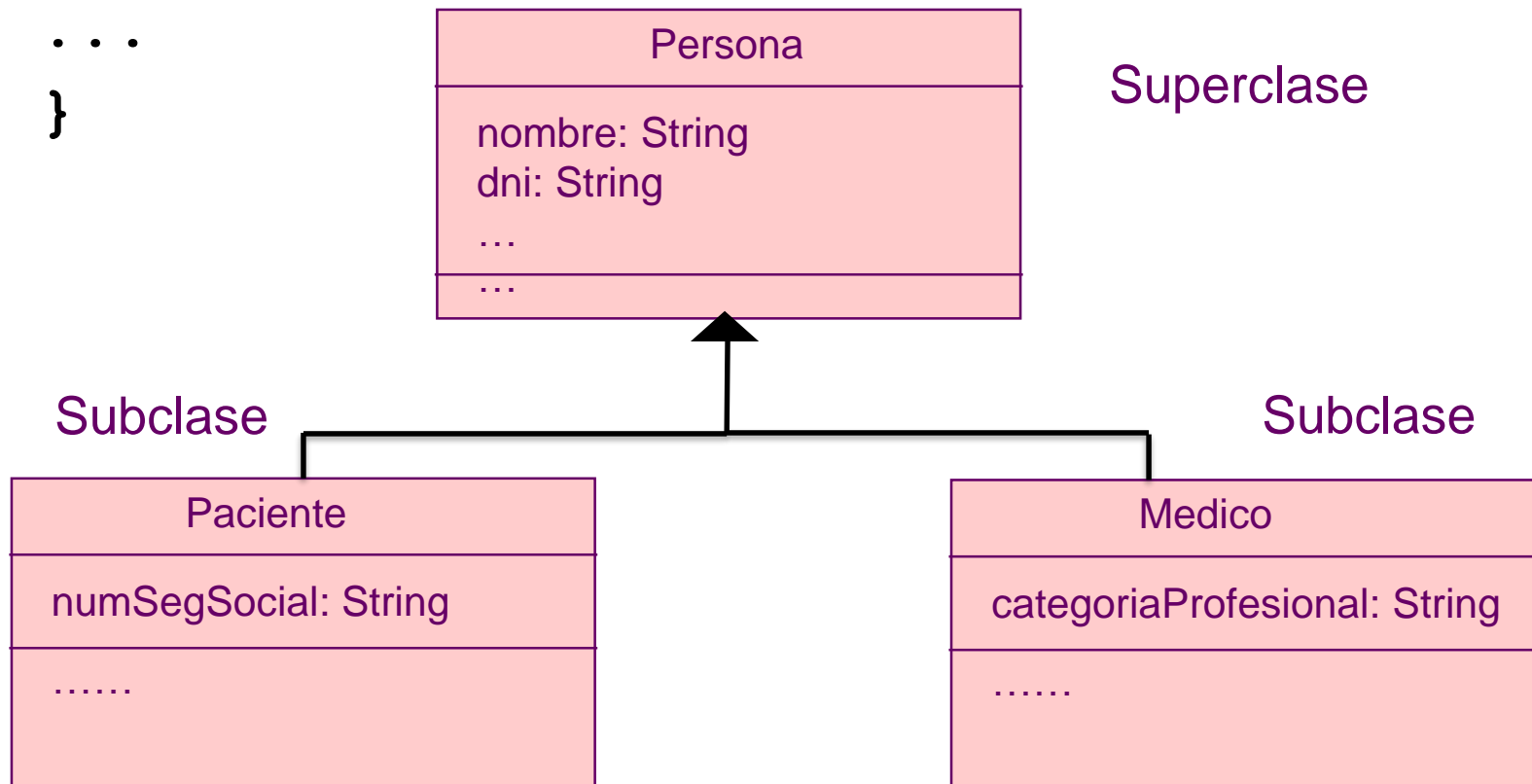
Herencia

- Se puede definir una clase (*subclase*) que *hereda* estado y comportamiento de otra clase (*superclase*) a la que amplía (añade estado y comportamiento), en la forma:

```
class Subclase extends Superclase {
```

```
...
```

```
}
```



Herencia

Estado

```
public class Punto {  
    private double x, y;  
    public Punto() { this(0,0); }  
    public Punto(double a, double b) {  
        x = a; y = b;  
    }  
    public double abscisa() { return x; }  
    public double ordenada() { return y; }  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public void abscisa(double a) { x = a; }  
    public void ordenada(double b) { y = b; }  
    public double distancia(Punto pto) {  
        return Math.sqrt(Math.pow(x - pto.x, 2)  
            + Math.pow(y - pto.y, 2));  
    }  
}
```

Comportamiento

Herencia

```
public class Partícula extends Punto {  
    final static double G = 6.67e-11;  
    private double masa;  
    public Partícula(double m) {  
        this(0,0,m);  
    }  
    public Partícula(double a, double b, double m) {  
        super(a, b);  
        masa = m;  
    }  
    public void masa(double m) { masa = m; }  
    public double masa() { return masa; }  
    public double atracción(Partícula part) {  
        return G * this.masa * part.masa /  
            Math.pow(this.distancia(part), 2);  
    }  
}
```

Estado
(+ Estado
Punto)

Se refiere a
Partícula(double, double,
double)

Se refiere a
Punto(double, double)

Comportamiento
(+ Comportamiento
Punto)

Heredada de
Punto

Herencia

```
public class Partícula extends Punto {  
    final static double G = 6.67e-11;  
    private double masa;  
    public Partícula(double m) {  
        this(0,0,m);  
    }  
    public Partícula(double a, double b, double m) {  
        super(a, b);  
        masa = m;  
    }  
    public void masa(double m) { masa = m; }  
    public double masa() { return masa; }  
    public double atracción(Partícula part) {  
        return G * masa * part.masa /  
            Math.pow(distancia(part), 2);  
    }  
}
```

Estado
(+ Estado
Punto)

Se refiere a
Partícula(double, double,
double)

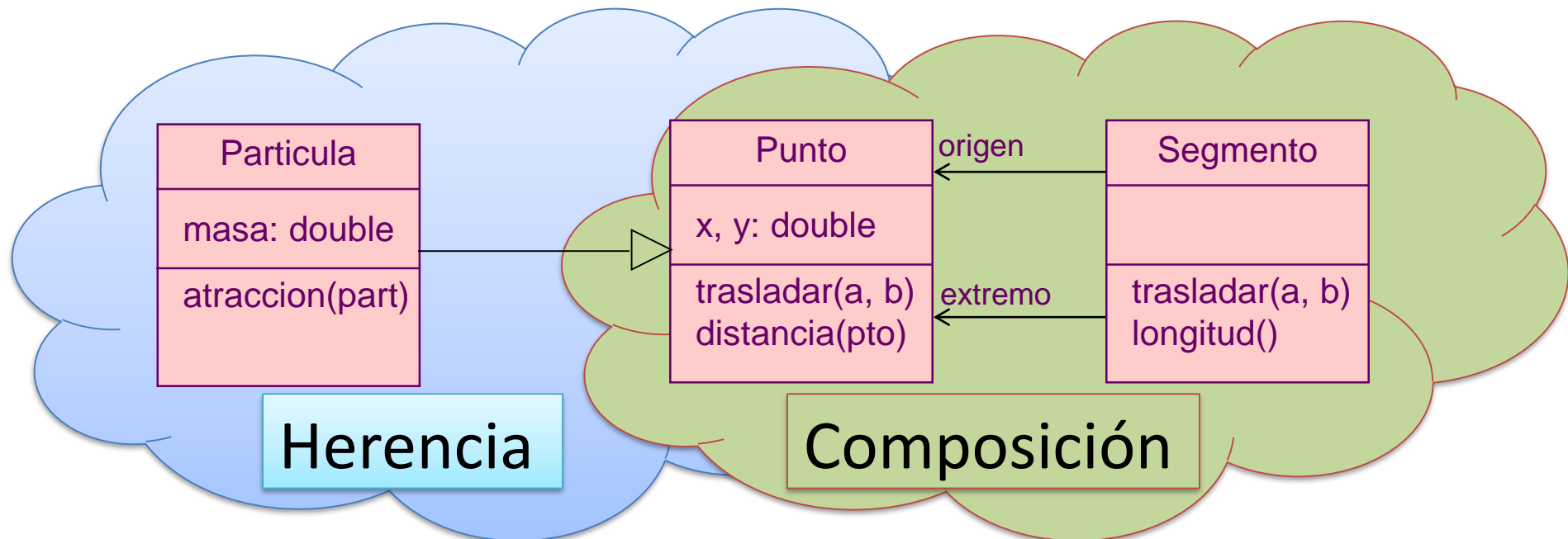
Se refiere a
Punto(double, double)

Heredada de
Punto

Comportamiento
(+ Comportamiento
Punto)

Herencia vs. composición

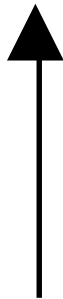
- Mientras que la herencia establece una relación de tipo “*es un/a*”, la composición responde a una relación de tipo “*tiene*” o “*está compuesto/a por*”.
- Así, por ejemplo, una partícula **es un** punto (con masa), mientras que un segmento **tiene** dos puntos (origen y extremo)



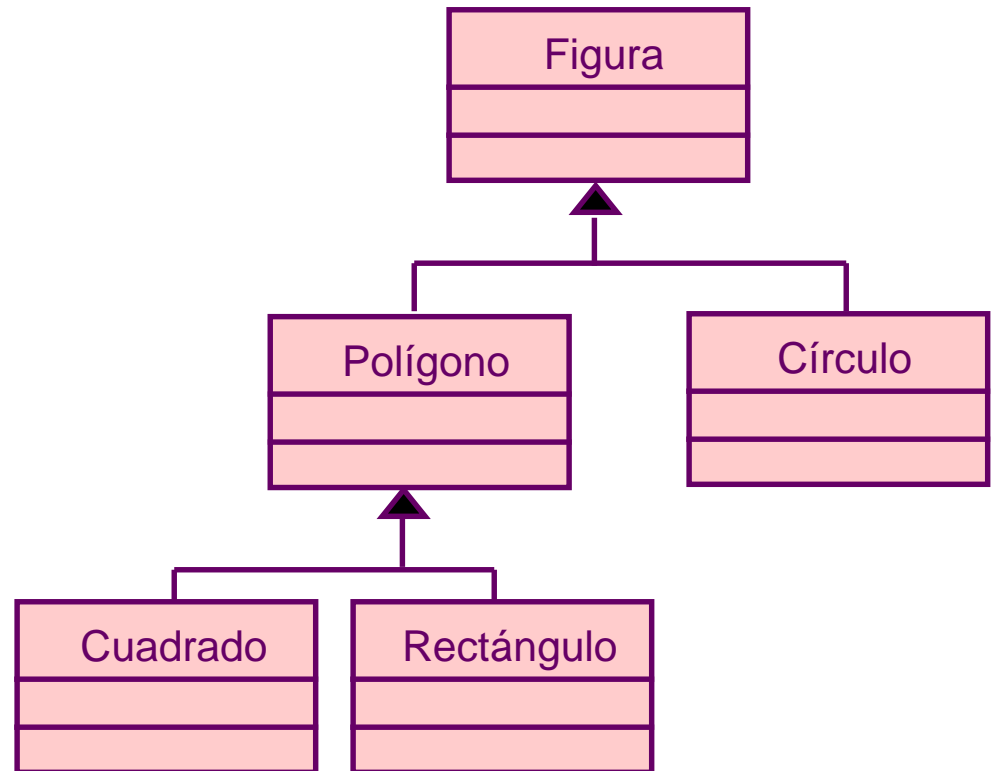
Herencia (jerarquía de clases)

- En Java sólo se permite *herencia simple*, por lo que pueden establecerse jerarquías de clases (una clase sólo puede heredar de una única clase, pero puede tener varias clases herederas)

Padres / Ascendientes /
Superclase



Hijos / Descendientes /
Subclase



- Todas las jerarquías confluyen en la clase **Object** de **java.lang** que recoge los comportamientos básicos que debe presentar cualquier clase.

Ya se verá en el tema 4

Herencia y constructores

- Los constructores **no** se heredan.
- Cuando se define un constructor en una subclase 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 Partícula(double m) {  
    this(0,0,m) ;  
}
```

- La llamada a **this** debe estar en la primera línea (si existen más)
 - Esto se puede hacer en cualquier clase (sea o no subclase de otra)

Herencia y constructores

- Los constructores **no** se heredan.
- Cuando se define un constructor en una subclase se debe proceder de alguna de las tres formas siguientes:
 - Invocar a algún constructor de la superclase mediante **super**:

- Por ejemplo:

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

- La llamada a **super** debe estar en la primera línea (si existen más)

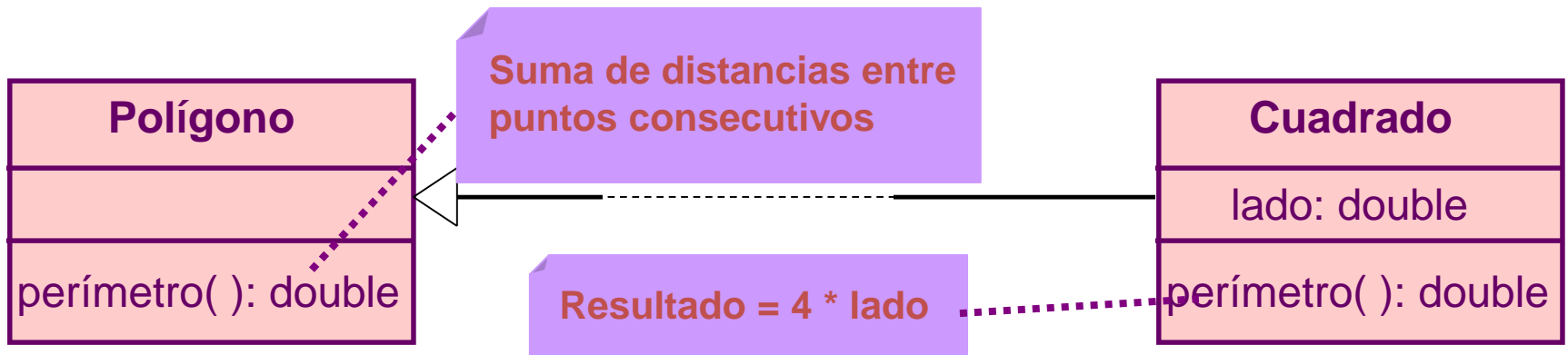
Herencia y constructores

- Los constructores **no** se heredan.
- Cuando se define un constructor en una subclase se debe proceder de alguna de las tres formas siguientes:
 - En otro caso, se invoca por defecto al constructor sin argumentos (constructor por defecto) de la superclase:
 - Por ejemplo:

```
public Partícula(double m) {  
    // Se invoca el constructor por defecto Punto()  
    masa = m;  
}
```
 - Se producirá un error de compilación si dicho constructor por defecto no existe en la superclase

Herencia y Redefinición del comportamiento

- Una subclase puede modificar el comportamiento heredado de la superclase redefiniendo algún método heredado (salvo que esté declarado **final**)



Herencia y Redefinición del comportamiento

```
public class Polígono {
    private Punto[] vért;

    public Polígono(Punto[] vs) {
        vért = vs;
    }

    public double perímetro() {
        Punto ant = vért[vért.length-1];
        double res = 0;
        for(Punto pto : vért) {
            res += pto.distancia(ant);
            ant = pto;
        }
        return res;
    }
    ...
}

public class Cuadrado extends Polígono {
    private double lado;

    public Cuadrado(Punto[] vs, double lado){
        super(vs);
        this.lado = lado;
    }
    ...

    Cuadrado cuad = new Cuadrado(...);

    per = cuad.perímetro(); // método de Polígono
```

Herencia y Redefinición del comportamiento

```
public class Polígono {
    private Punto[] vért;

    public Polígono(Punto[] vs) {
        vért = vs;
    }

    public double perímetro() {
        Punto ant = vért[vért.length-1];
        double res = 0;
        for(Punto pto : vért) {
            res += pto.distancia(ant);
            ant = pto;
        }
        return res;
    }
    ...
}
```

```
public class Cuadrado extends Polígono {
    private double lado;

    public Cuadrado(Punto[] vs, double lado){
        super(vs);
        this.lado = lado;
    }

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

    ...
}
```

```
Cuadrado cuad = new Cuadrado(...);
```

```
per = cuad.perímetro(); // método de Cuadrado
```


Herencia y Redefinición del comportamiento

```
public class Polígono {  
    private Punto[] vért;  
  
    public Polígono(Punto[] vs) {  
        vért = vs;  
    }  
  
    public double perímetro() {  
        Punto ant = vért[vért.length-1];  
        double res = 0;  
        for(Punto pto : vért) {  
            res += pto.distancia(ant);  
            ant = pto;  
        }  
        return res;  
    }  
}
```

```
public class Cuadrado extends Polígono {  
    private double lado;  
  
    public Cuadrado(Punto[] vs, double lado){  
        super(vs);  
        this.lado = lado;  
    }  
  
    @Override  
    public double perímetro() {  
        return 4 * lado;  
    }  
  
    ...  
}
```

@Override

... La anotación `@Override` indica al compilador que el método pretende redefinir el comportamiento de un método de la superclase u otra ascendiente en la jerarquía, de tal forma que si el compilador no encuentra ese método en ninguna clase ascendiente, avisa del error. (Como se verá más adelante, también se utiliza para los métodos de interfaces implementados por alguna clase)

Herencia y Redefinición del comportamiento

```
public class Polígono {  
    protected Punto[] vért;  
  
    public Polígono(Punto[] vs) {  
        vért = vs;  
    }  
  
    public double perímetro() {  
        Punto ant = vért[vért.length-1];  
        double res = 0;  
        for(Punto pto : vért) {  
            res += pto.distancia(ant);  
            ant = pto;  
        }  
        return res;  
    }  
    ...  
}
```

```
public class Cuadrado extends Polígono {  
  
    public Cuadrado(Punto[] vs) {  
        super(vs);  
    }  
  
    @Override  
    public double perímetro() {  
        return 4 * lado();  
    }  
  
    public double lado() {  
        return vért[0].distancia(vért[1]);  
    }  
    ...  
}
```

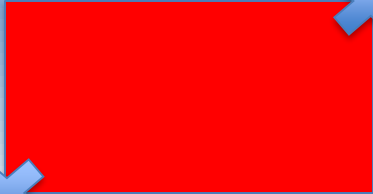
Si en el constructor de la clase Cuadrado no se recibe el lado, éste se puede calcular internamente usando el array vért. Pero para ello, el nivel de visibilidad de vért debe cambiarse en la clase Polígono (protected en lugar de private).

Herencia y Redefinición del comportamiento

- El método heredado que se redefine queda oculto en la subclase por el nuevo método.
 - Si se desea acceder al método heredado (dentro de algún método de la subclase, incluido el que está redefiniendo al heredado), se usa:

super.<nombre del método>(argumentos)

```
public class PuntoAcotado extends Punto {  
    private Punto esquinaI, esquinaD;  
  
    public PuntoAcotado() { ... }  
    public PuntoAcotado(Punto eI, Punto eD) { ... }  
    public double ancho() { ... }  
    public double alto() { ... }
```




@Override

```
public void trasladar(double a, double b) {  
    super.trasladar(a, b);  
    if (abscisa() < esquinaI.abscisa())  
        abscisa(esquinaI.abscisa())  
    if (abscisa() > esquinaD.abscisa())  
        abscisa(esquinaD.abscisa())  
    if (ordenada() < esquinaI.ordenada())  
        ordenada(esquinaI.ordenada())  
    if (ordenada() > esquinaD.ordenada())  
        ordenada(esquinaD.ordenada())  
}
```


Herencia y Polimorfismo

- Un lenguaje tiene **capacidad polimórfica** sobre los datos cuando
 - una variable declarada de un tipo (o clase) determinado –*tipo estático*– 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 Partícula(3, 5, 22);
```



Tipo estático
de **pto**



Tipo dinámico
de **pto**

Herencia y Polimorfismo

- Los objetos de una clase descendiente se pueden considerar también como objetos de una clase ascendiente (ej. toda partícula es un punto; todo cuadrado es una figura)
- Es legal que una variable del tipo de una clase ascendiente (tipo estático) haga referencia a un objeto de una clase descendiente (tipo dinámico)

```
Punto pto = new Partícula(...);
```

- Pero esa variable sólo podrá acceder variables y métodos definidos en la clase ascendiente (tipo estático).

```
pto.trasladar(...);      OK  
pto.masa(...);           INCORRECTO
```

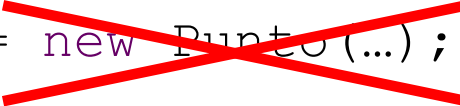
- Aunque podemos hacer una conversión de tipos (hay que estar muy seguros)

```
((Partícula) pto).masa(...);      OK
```

Herencia y Polimorfismo

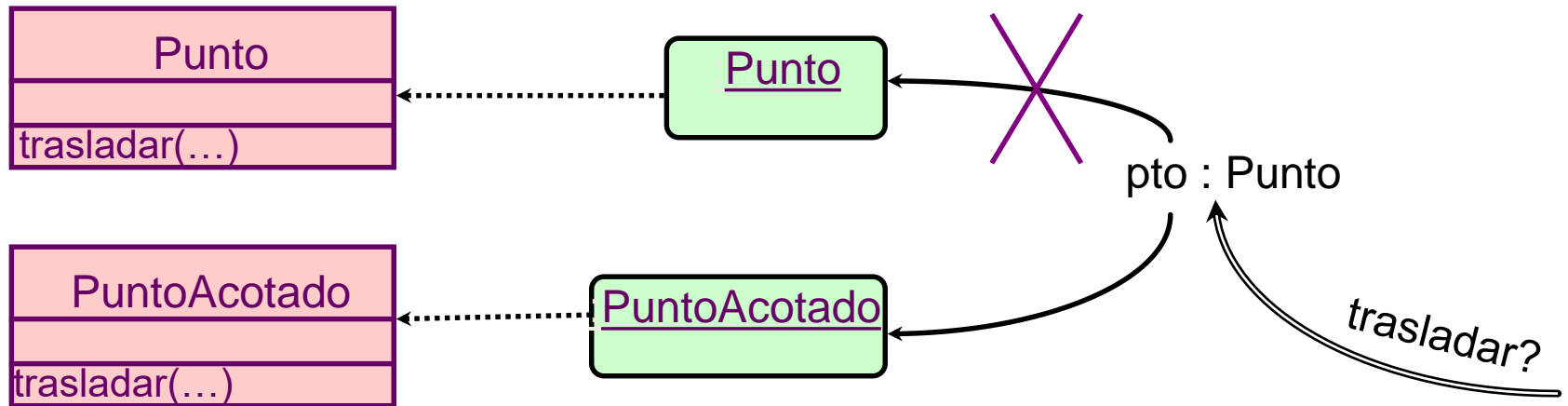
- Esto nos permite construir estructuras de datos con elementos de naturaleza distinta, pero que tienen algún comportamiento común.
 - Ejemplo: un array de puntos que almacene puntos y partículas y realizar una operación de traslación de todos.
- Los objetos de una clase ascendiente NO son objetos de una clase descendiente.

```
Partícula part;  
...  
part = new Punto(...);
```



Polimorfismo y 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



- El compilador admitirá la expresión
`pto.trasladar()` ;
si el **tipo estático** de `pto` (es decir la clase **Punto**) **acepta el mensaje** `trasladar()`, aunque para resolver utilice **vinculación dinámica** (es decir, el método `trasladar()` de la clase **PuntoAcotado**)

Polimorfismo y Vinculación Dinámica (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 sigue buscando de forma ascendente por las clases superiores.
 - Si ha compilado, seguro que hay un método

Polimorfismo y Vinculación Dinámica (resolución del método a ejecutar)

```
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.

Polimorfismo y Vinculación Dinámica (resolución del método a ejecutar)

```
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


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

- **Si compila** porque se ha realizado una **conversión** de Punto a Particula (hay que asegurarse previamente que esto es correcto (más adelante se verá cómo)).

Prohibiendo subclases

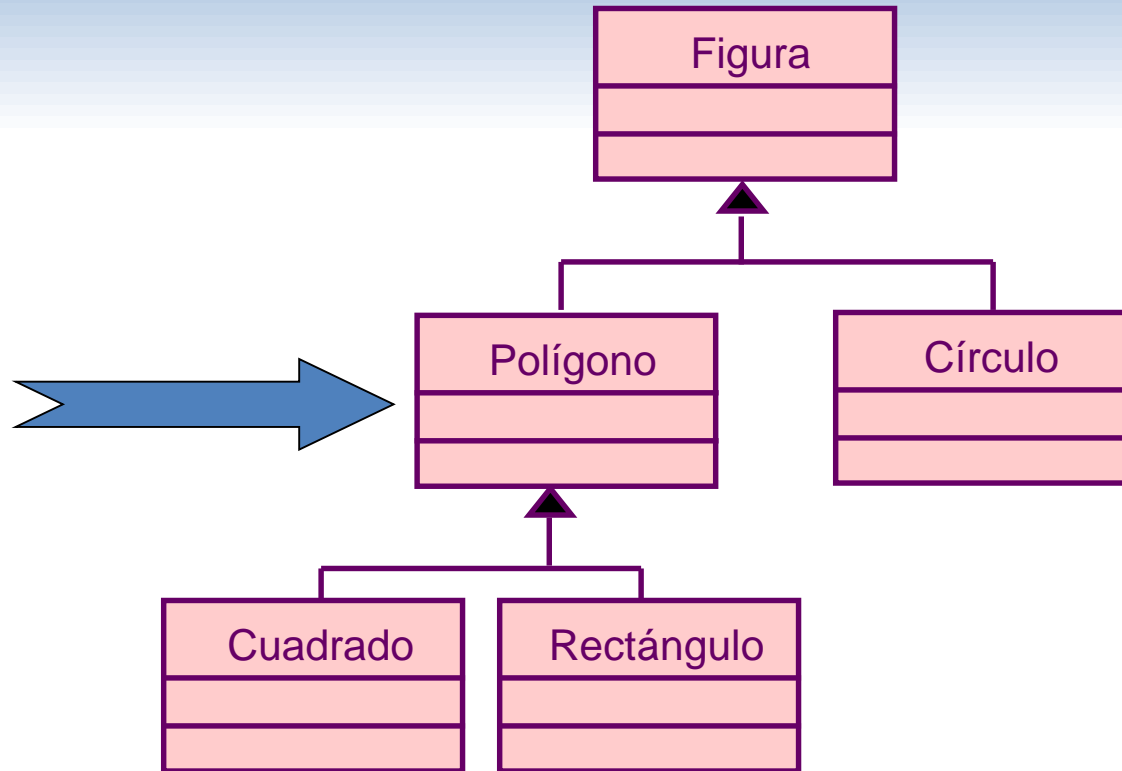
- Por razones de seguridad o de diseño, se puede prohibir la definición de subclases para una clase etiquetándola con **final**.
 - Recordad que una subclase puede sustituir a su superclase donde ésta sea necesaria y tener comportamientos muy distintos
- El compilador rechazará cualquier intento de definir 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.

Contenido

- Introducción histórica
- Programas y Paquetes
- Clases y Objetos
- Elementos del Lenguaje:
 - Tipos, constantes, variables
 - Expresiones
 - Operadores
 - Instrucciones
 - Bloques
- Control de errores, excepciones
- Cadenas de caracteres
- Arrays
- Herencia y Redefinición del Comportamiento
- Polimorfismo y Vinculación Dinámica
- Clases Abstractas e Interfaces 

Clases abstractas

- Se puede definir una clase como resultado de una abstracción sobre otras clases recogiendo un estado básico común y un comportamiento básico común.
- Estas clases se etiquetan como **abstract** y tienen métodos sin implementación, también etiquetados como **abstract**.
- Se utilizan para formar jerarquías.
- Se pueden utilizar como tipos, pero no se pueden crear objetos de ellas.
- Deben tener subclases que no sean abstractas para crear objetos.



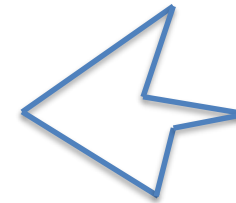
```

public abstract class Polígono {
    protected Punto[] vért;

    public Polígono(Punto[] vs) {
        vért = vs;
    }
    public void trasladar(double a, double b) {
        for(Punto pto : vért) { pto.trasladar(a,b); }
    }
    public double perímetro() {
        Punto ant = vért[vért.length-1];
        double res = 0;
        for(Punto pto : vért) {
            res += pto.distancia(ant);
            ant = pto;
        }
        return res;
    }

    public abstract double área(); // No sabemos calcularla
}

```



```

public class Cuadrado extends Polígono {

    public Cuadrado(...) {...}
    @Override
    public double área() {
        double l = lado();
        return l*l;
    }
    public double lado() {
        return vért[0].distancia(vért[1]);
    }
    @Override
    public String toString() {...}
}

```



```

public class Rectangulo extends Polígono {

    public Rectangulo(...) {...}
    @Override
    public double área() {
        return base()*altura();
    }
    public double base() {
        return vért[0].distancia(vért[1]);
    }
    public double altura() {
        return vért[1].distancia(vért[2]);
    }
    @Override
    public String toString() {...}
}

```




```
public abstract class Polígono {  
    protected Punto[] vért;  
    ...  
    public void trasladar(double a, double b) {  
        for (Punto pto : vért) {  
            vért.trasladar(a, b);  
        }  
    }  
    ...  
    public abstract double área(); // No sabemos calcularla  
};
```

~~Polígono pol = new Polígono(...);~~

Polígono pol = new Cuadrado(...);

Interfaces

- Una interfaz define *un protocolo de comportamiento* que debe ser *implementado* por cualquier clase que pretenda ofrecer ese comportamiento.
- Una clase puede *implementar* varias interfaces.
- Una interfaz sólo puede ser *extendida* por otra interfaz.
- Una interfaz puede heredar de varias interfaces.

Definición de interfaces

- En una interfaz sólo se permiten constantes, métodos abstractos y métodos por defecto (pueden redefinirse).

`public static final`

```
public interface Interfaz
    extends Interfaz1, Interfaz2 {
    int TAM = 10;
    default void metodoPorDefecto(...) {
        ... // implementación en función del resto de métodos
    }
    void metodoAbstracto(int val);
    ...
}
```

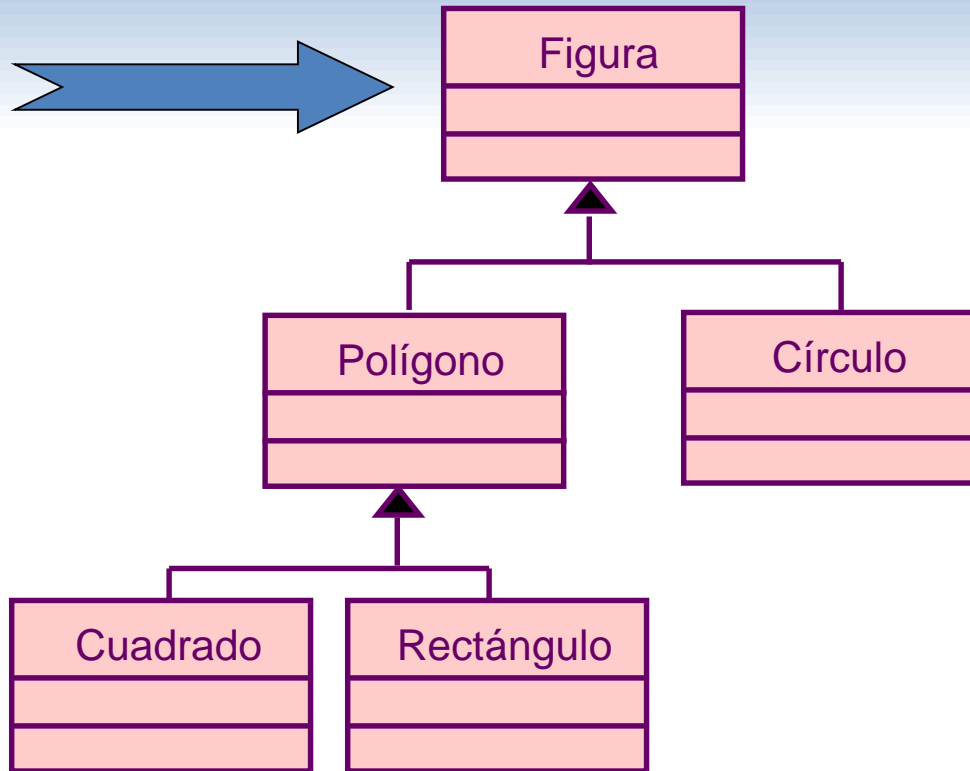
The diagram illustrates the mapping of Java keywords to interface syntax. An arrow points from the text `public static final` to the `public interface` line. Another arrow points from the text `public abstract` to the `void metodoAbstracto` line.

`public abstract`

Implementación de interfaces

- Cuando una clase implementa una interfaz:
 - *Hereda* todas las constantes definidas en la interfaz y en sus superinterfaces,
 - *Se adhiere al contrato* definido en la interfaz y en sus superinterfaces,
 - *Adherirse al contrato quiere decir que debe implementar* todos los métodos
(salvo que sea una clase que se quiera mantener abstracta, en cuyo caso los métodos no implementados aparecerán como **abstract**).

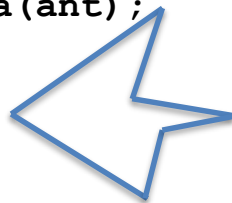
```
public class Clase
    implements Interfaz {
    ...
}
```



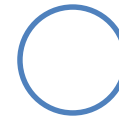
```
public interface Figura {  
    double perímetro();  
    double área();  
}
```



```
public abstract class Polígono  
    implements Figura{  
  
    protected Punto[] vért;  
  
    public Polígono(Punto[] vs) {  
        vért = vs;  
    }  
  
    @Override  
    public double perímetro() {  
        Punto ant = vért[vért.length-1];  
        double res = 0;  
        for(Punto pto : vért) {  
            res += pto.distancia(ant);  
            ant = pto;  
        }  
        return res;  
    }  
  
    @Override  
    public abstract double área();  
}
```



```
public class Círculo implements Figura{  
  
    private double radio;  
  
    public Círculo(double r) {  
        radio = r;  
    }  
    @Override  
    public double perímetro() {  
        return 2.0 * Math.PI * radio;  
    }  
    @Override  
    public double área() {  
        return Math.PI * radio * radio;  
    }  
    @Override  
    public String toString() {...}  
}
```



```

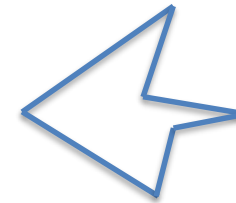
public abstract class Polígono implements Figura{
    protected Punto[] vért;

    public Polígono(Punto[] vs) {
        vért = vs;
    }

    @Override
    public double perímetro() {
        Punto ant = vért[vért.length-1];
        double res = 0;
        for(Punto pto : vért) {
            res += pto.distancia(ant);
            ant = pto;
        }
        return res;
    }

    @Override
    public abstract double área();
}

```



```

public class Cuadrado extends Polígono {

    public Cuadrado(...) {...}
    @Override
    public double área() {
        double l = lado();
        return l*l;
    }
    public double lado() {
        return vért[0].distancia(vért[1]);
    }
    @Override
    public String toString() {...}
}

```



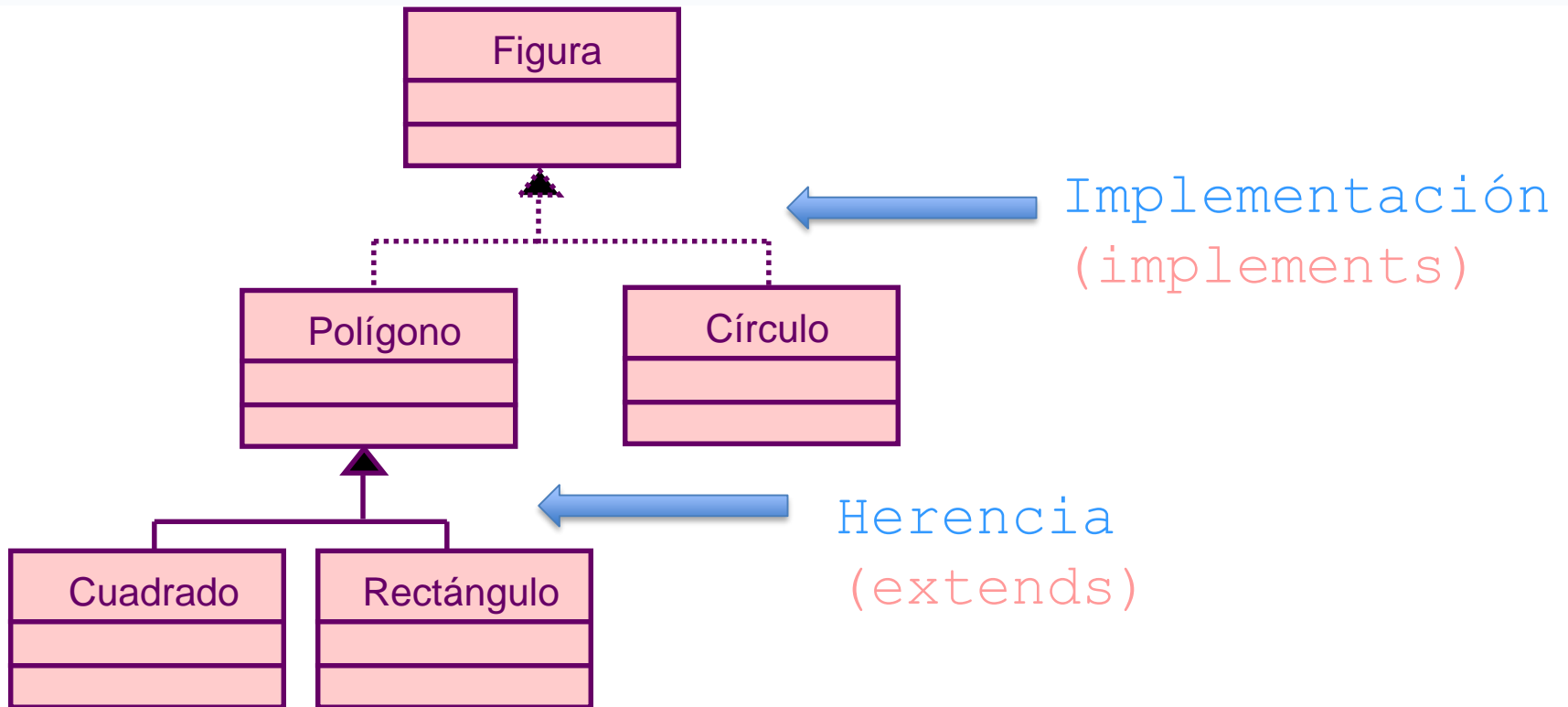
```

public class Rectangulo extends Polígono {

    public Rectangulo(...) {...}
    @Override
    public double área() {
        return base()*altura();
    }
    public double base() {
        return vért[0].distancia(vért[1]);
    }
    public double altura() {
        return vért[1].distancia(vért[2]);
    }
    @Override
    public String toString() {...}
}

```





Uso de interfaces

- *No se pueden crear instancias* de una interfaz.
- Podemos declarar variables del tipo definido por una interfaz, que pueden hacer referencia a cualquier objeto de una clase que implementa la interfaz (y de las herederas de ella: Polimorfismo)

```
Figura fig;  
...  
fig = new Cuadrado(...);
```

```
Figura [] figs = new Figura [10];  
...  
figs[0] = new Circulo(...);  
figs[1] = new Cuadrado(...);  
...
```

Ejemplo.

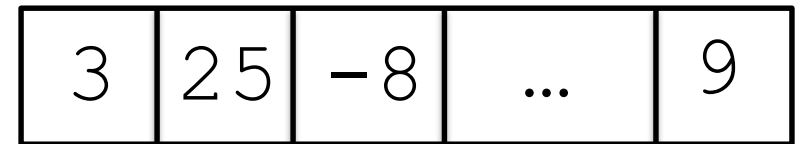
Interfaz *ColeccionEnteros*

```
public interface ColeccionEnteros {  
    void añadir(int elem);  
    void eliminar(int elem);  
    int maximo();  
    boolean pertenece(int elem);  
    default boolean esVacia() {  
        return tamaño() == 0;  
    }  
    int tamaño();  
}
```

Ejemplo.

Interfaz *ColeccionEnteros*

```
public interface ColeccionEnteros {  
    void añadir(int elem);  
    void eliminar(int elem);  
    int maximo();  
    boolean pertenece(int elem);  
    default boolean esVacia() {  
        return tamaño() == 0;  
    }  
    int tamaño();  
}
```



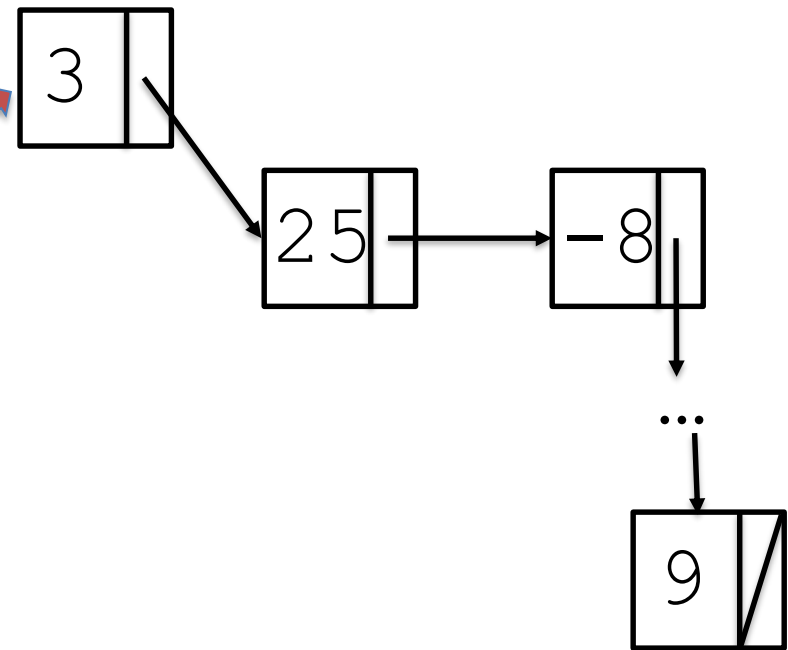
```
public class ColeccionEntArray  
    implements ColeccionEnteros {  
  
    private int numElementos;  
    private int[] secuencia;  
  
    public ColeccionEntArray(int tam) {  
        secuencia = new int[tam];  
        numElementos = 0;  
    }  
  
    public void añadir(int e) {...}  
    public void eliminar(int e) {...}  
    public int máximo() {...}  
    public boolean pertenece(int e) {...}  
    public int tamano() {...}  
    public String toString() {...}  
}
```

Ejemplo.

Interfaz *ColeccionEnteros*

```
public interface ColeccionEnteros {  
    void añadir(int elem);  
    void eliminar(int elem);  
    int maximo();  
    boolean pertenece(int elem);  
    default boolean esVacia() {  
        return tamaño() == 0;  
    }  
    int tamaño();  
}
```

```
public class ColeccionEntListaEnlazada  
    implements ColeccionEnteros {  
  
    private static class Nodo {  
        int dato;  
        Nodo siguiente;  
    }  
  
    private Nodo primero;  
  
    public ColeccionEntListaEnlazada() {...}  
  
    public void añadir(int e) {...}  
    public void eliminar(int e) {...}  
    public int maximo() {...}  
    public boolean pertenece(int e) {...}  
    public int tamaño() {...}  
    public String toString() {...}  
}
```



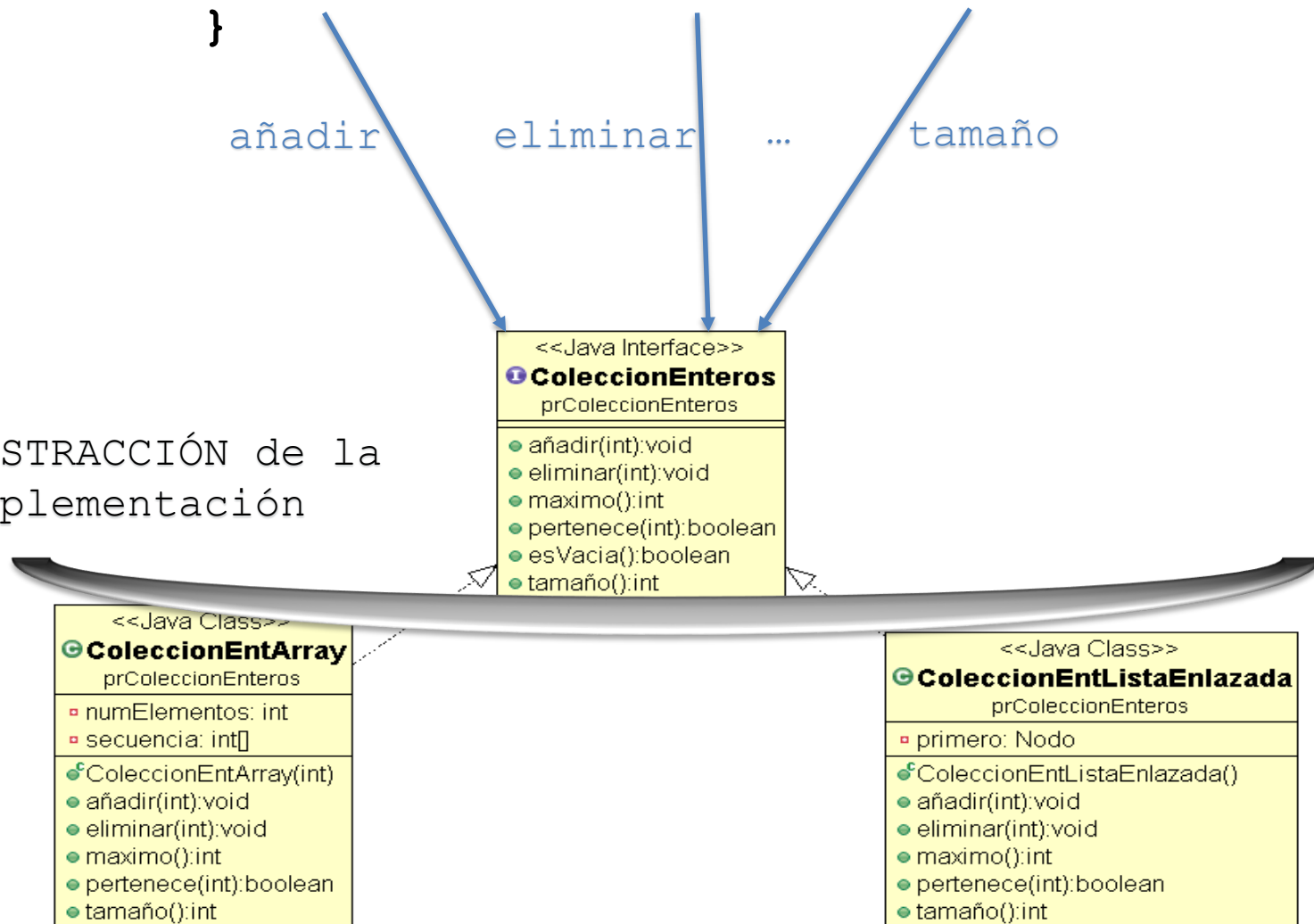
Ejemplo.

Interfaz *ColeccionEnteros*

```
void metodo (ColeccionEnteros c) {
```

```
    ...  
}  
  
añadir    eliminar    ...    tamaño
```

ABSTRACCIÓN de la
Implementación



Ejemplo.

Interfaz *ColeccionEnteros*

- Una interfaz se puede usar como tipo para: declarar variables, parámetros y valores devueltos por una función.
- Pero se requieren instancias de clases que implementen la interfaz para poder manipularlas

```
public class Prueba {  
  
    public static void main(String[] args) {  
        ColeccionEnteros col;  
  
        col = leer();  
        escribir(col);  
        if (!col.esVacia()) {  
            System.out.println("mayor = " + col.maximo());  
        }  
        ...  
    }  
    private static ColeccionEnteros leer() {  
  
        ColeccionEnteros c = new ColeccionEntArray(100);  
        ...  
    }  
    private static void escribir (ColeccionEnteros c) {  
        System.out.println("la colección es: " + c);  
    }  
}
```

Interfaz

Interfaz

Objeto de clase
que
implementa la
interfaz

Interfaz

Ejemplo.

Interfaz *ColeccionEnteros*

- Una interfaz se puede usar como tipo para: declarar variables, parámetros y valores devueltos por una función.
- Pero se requieren instancias de clases que implementen la interfaz para poder manipularlas

```
public class Prueba {  
  
    public static void main(String[] args) {  
        ColeccionEnteros col;  
  
        col = leer();  
        escribir(col);  
        if (!col.esVacia()) {  
            System.out.println("mayor = " + col.maximo());  
        }  
        ...  
    }  
    private static ColeccionEnteros leer() {  
  
        ColeccionEnteros c = new ColeccionEntListaEnlazada();  
        ...  
    }  
    private static void escribir (ColeccionEnteros c) {  
        System.out.println("la colección es: " + c);  
    }  
}
```

Interfaz

Interfaz

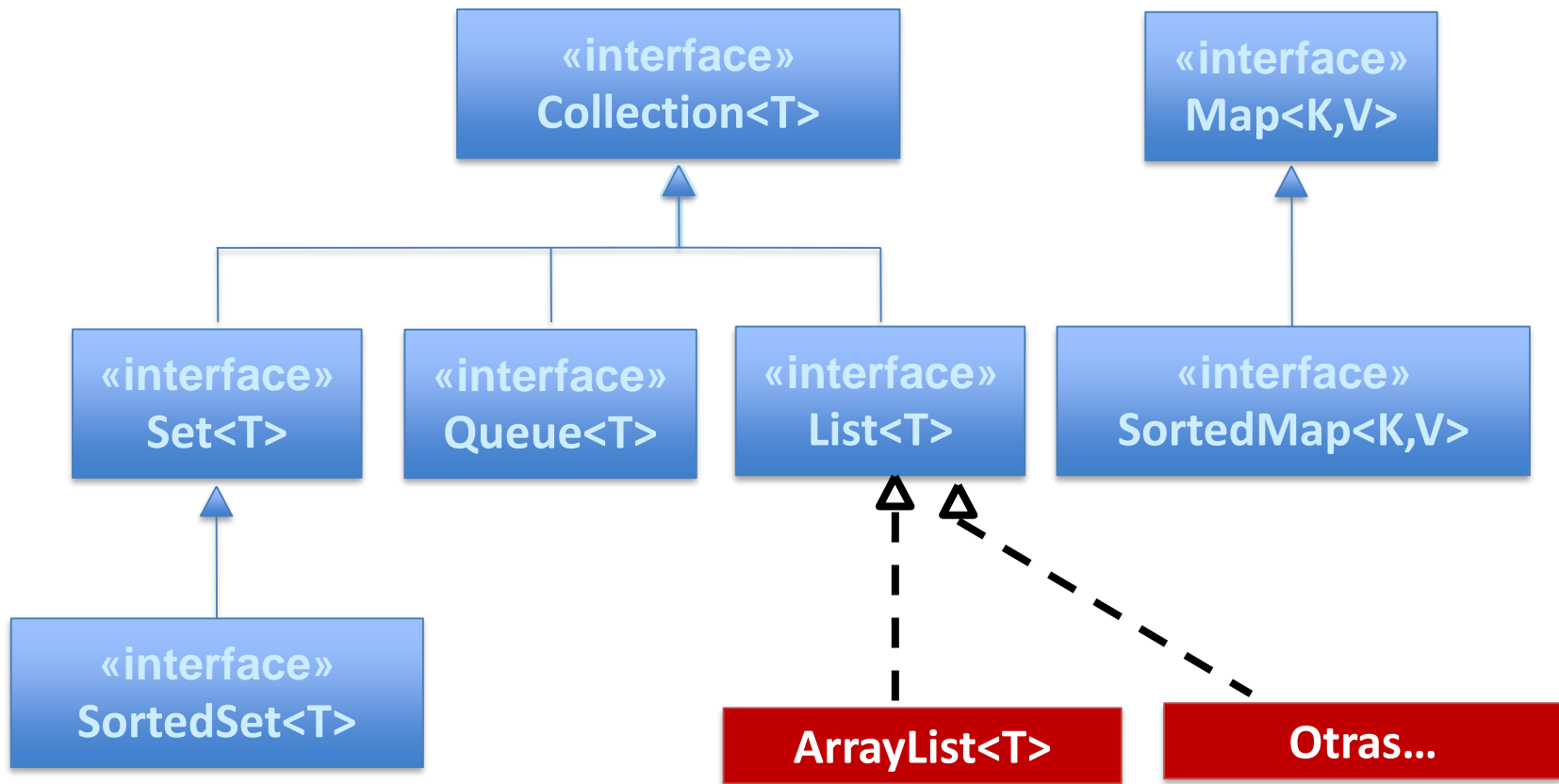
Objeto de clase
que
implementa la
interfaz

Interfaz

Interfaces básicas que ofrece Java

(java.util)

Las veremos en el Tema 6



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

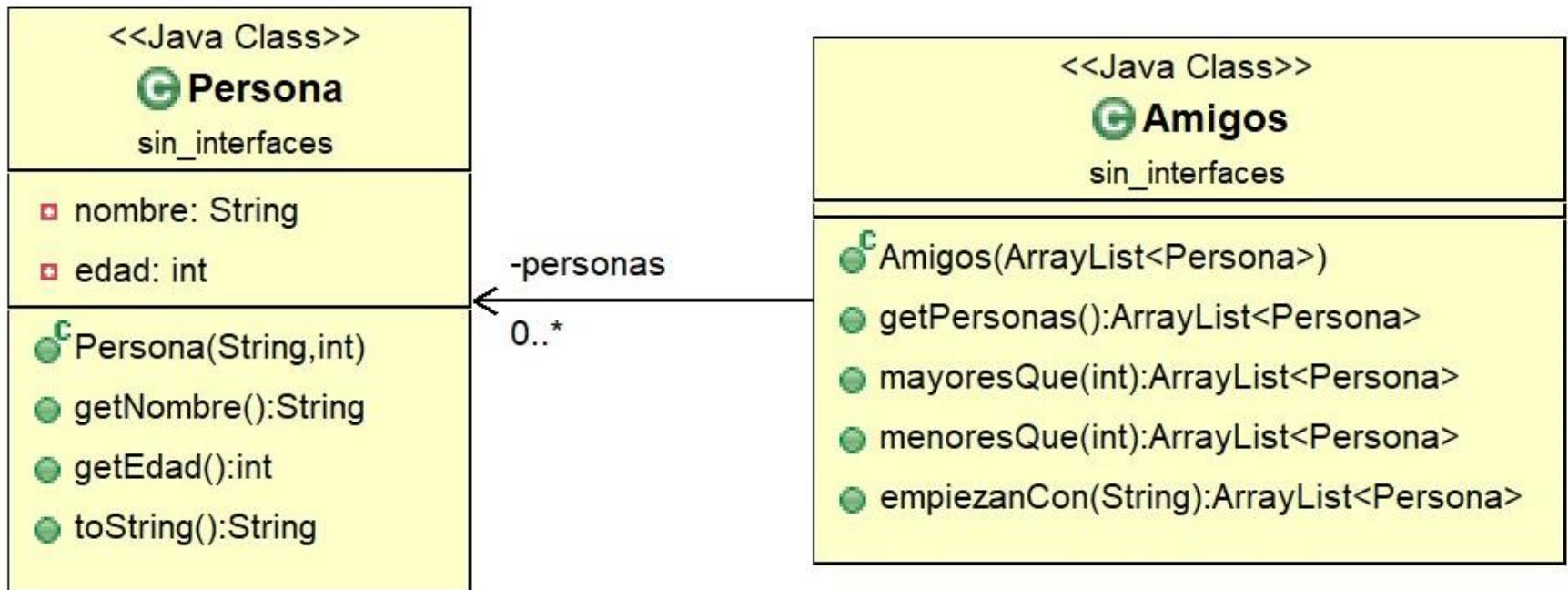

Interfaces como abstracción funcional

- Ya hemos visto cómo las interfaces nos proporcionan una *abstracción de implementaciones* concretas (ej. podemos usar una colección de enteros (ColeccionEnteros) sin conocer cómo está implementada).
- Las interfaces también proporcionan soluciones elegantes a problemas donde se desea aplicar diferentes funcionalidades a unos mismos datos (**abstracción funcional**).
- Para comprenderlo mejor, veamos un ejemplo solucionado sin interfaces y después con interfaces aplicando abstracción funcional.
 - Disponemos de dos clases
 - Clase **Persona** con información de una persona (nombre y edad)
 - Clase **Amigos** con información de muchas personas (una lista de personas)
 - Queremos operar con la clase Amigos para coleccionar las personas:
 - Que son menores de una edad dada.
 - Que son mayores de una edad dada.
 - Cuyos nombres empiezan por una cadena dada.
 - ...

Solución sin interfaces

- Lo natural es crear un método en la clase **Amigos** que resuelva cada una de las funcionalidades requeridas
 - Para las personas menores de una edad dada.
ArrayList<Persona> menoresQue(int n)
 - Para las personas mayores de una edad dada.
ArrayList<Persona> mayoresQue(int n)
 - Para las personas cuyos nombres empiezan por una cadena dada.
ArrayList<Persona> empiezanCon(String s)
 - ...
- El cuerpo de estos métodos **será muy parecido**, solo cambiarán la manera de **seleccionar** las personas.

Solución sin interfaces



Solución sin interfaces

```
public class Amigos {  
    private ArrayList<Persona> personas;  
    public Amigos(ArrayList<Persona> pers) {  
        personas = pers;  
    }  
    public ArrayList<Persona> getPersonas() {  
        return personas;  
    }  
    ...  
}
```

Solución sin interfaces

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

Solución sin interfaces

```
public ArrayList<Persona> mayoresQue(int n) {  
    ArrayList<Persona> res = new ArrayList<>();  
    for(Persona p : personas) {  
        if(p.getEdad() > n) {  
            res.add(p);  
        }  
    }  
}
```

```
public ArrayList<Persona> menoresQue(int n) {  
    ArrayList<Persona> res = new ArrayList<>();  
    for(Persona p : personas) {  
        if(p.getEdad() < n) {  
            res.add(p);  
        }  
    }  
    return res;  
}
```

Solución sin interfaces

```
public ArrayList<Persona> mayoresQue(int n) {  
    ArrayList<Persona> res = new ArrayList<>();  
    for(Persona p : personas) {  
        if(p.getEdad() > n) {  
            res.add(p);  
        }  
    }  
}
```

```
public ArrayList<Persona> menoresQue(int n) {  
    ArrayList<Persona> res = new ArrayList<>();  
    for(Persona p : personas) {  
        if(p.getEdad() < n) {  
            res.add(p);  
        }  
    }  
}
```

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

Solución sin interfaces

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

Para usarlos
simplemente se
invoca al método
adecuado

Solución sin interfaces

Empiezan con ma

(maria, 32)

(marta, 28)

(manuel, 29)

Mayores de 28

(maria, 32)

(julio, 33)

(manuel, 29)

Menores de 27

(juan, 25)

(justino, 25)

Solución sin interfaces

¿Cómo añadir una nueva funcionalidad?

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

```
public ArrayList<Persona> contiene(String s) {  
    ArrayList<Persona> res = new ArrayList<>();  
    for(Persona p : personas) {  
        if(p.getNombre().contains(s)) {  
            res.add(p);  
        }  
    }  
    return res;  
}
```

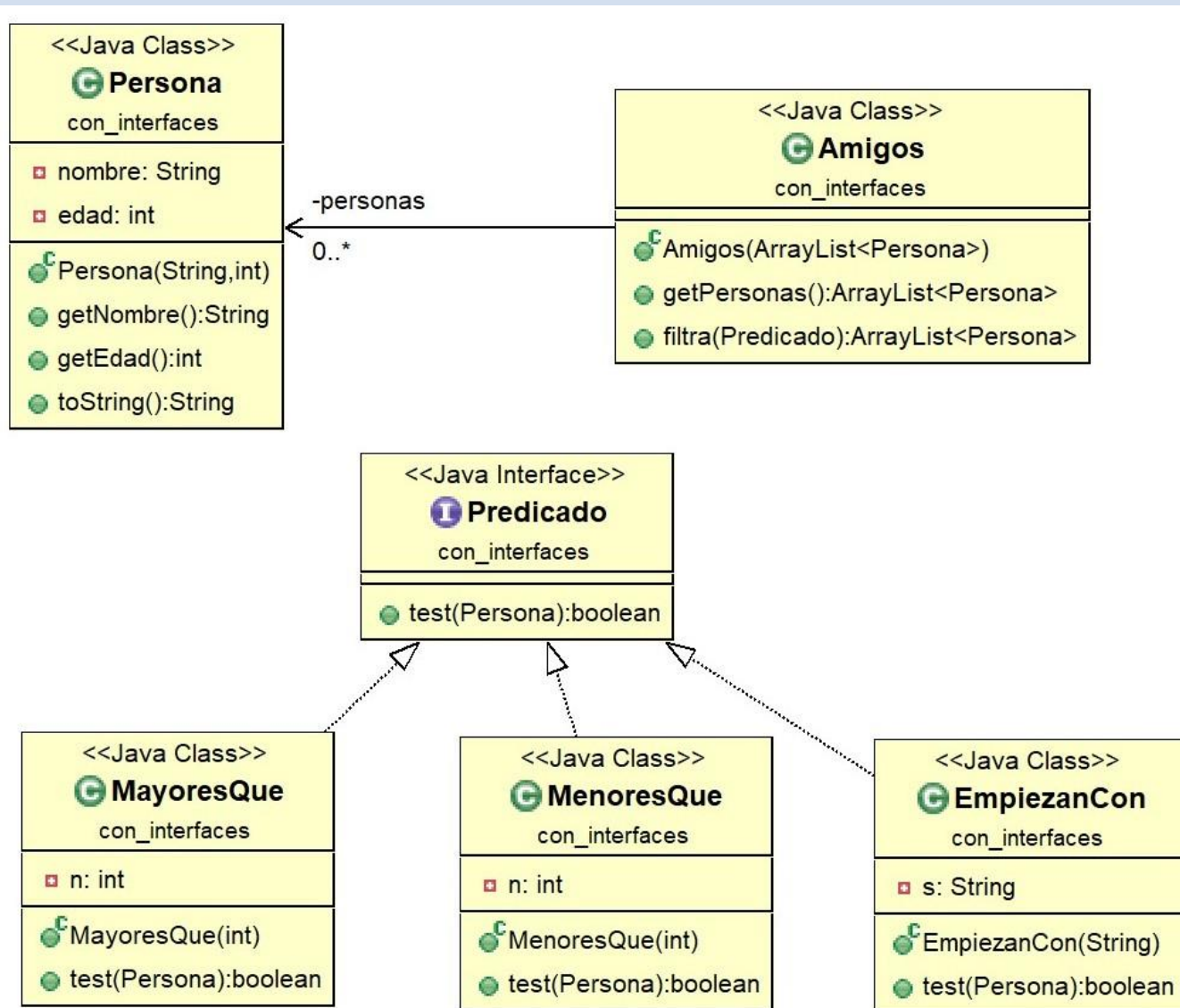
¿Podemos diseñar la clase **Amigos** sin necesidad de tener que conocer todas las funcionalidades que después se puedan necesitar para seleccionar personas?

Solución con interfaces

(Abstracción funcional)

- Se crea una interfaz que incluya un método que decide qué elementos seleccionar:
 - En el ejemplo la interfaz la llamaremos **Predicado**.
 - Incluirá el método **boolean test(Persona p)** que será el que defina qué personas seleccionar 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 **EmpiezanCon** que implementan la interfaz **Predicado** y por tanto definen el método **test**, cada una a su manera.
- La clase *contenedora* (que almacena los elementos sobre los que aplicar una determinada funcionalidad) dispondrá 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
ArrayList<Persona> filtra(Predicado pred)

Solución con interfaces (Abstracción funcional)



Solución con interfaces (Abstracción funcional)

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

Solución con interfaces (Abstracción funcional)

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

Solución con interfaces (Abstracción funcional)

@FunctionalInterface

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

```
public class MayoresQue implements Predicado {  
    private int n;  
    public MayoresQue(int n) {  
        this.n = n;  
    }  
    @Override  
    public boolean test(Persona p) {  
        return p.getEdad() > n;  
    }  
}
```

Solución con interfaces (Abstracción funcional)

@FunctionalInterface

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

```
public class MayoresQue implements Predicado {  
    private int n;
```

```
}
```

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


Solución con interfaces (Abstracción funcional)

@FunctionalInterface

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

```
public class MayoresQue implements Predicado {  
    private int n;  
    public MayoresQue(int n) {
```

```
public class MenoresQue implements Predicado {  
    private int n;
```

```
public class EmpiezanCon implements Predicado {  
    private String s;  
    public EmpiezanCon(String s) {  
        this.s = s;  
    }  
    @Override  
    public boolean test(Persona p) {  
        return p.getNombre().startsWith(s);  
    }  
}
```

Solución con interfaces (Abstracción funcional)

```
public class Main {  
    public static void main(String [] args) {  
        ArrayList<Persona> personas = new ArrayList<>();  
        personas.add(new Persona("juan", 25));  
        personas.add(new Persona("maria", 32));  
        personas.add(new Persona("marta", 28));  
        personas.add(new Persona("julio", 33));  
        personas.add(new Persona("manuel", 29));  
        personas.add(new Persona("justino", 25));  
  
        Amigos amigos = new Amigos(personas);  
  
        escribir("Empiezan con ma", amigos.filtra(new EmpiezanCon("ma")));  
  
        escribir("Mayores de 28", amigos.filtra(new MayoresQue(28)));  
  
        escribir("Menores de 27", amigos.filtra(new MenoresQue(27)));  
    }  
  
    private static void escribir(String msj, ArrayList<Persona> ps) {  
        System.out.println(msj);  
        for(Persona p : ps) {  
            System.out.println(p);  
        }  
    }  
}
```

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

Solución con interfaces (Abstracción funcional)

¿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;  
    }  
    @Override  
    public boolean test(Persona p) {  
        return p.getNombre().contains(s);  
    }  
}
```