

Herencia. Interfaces



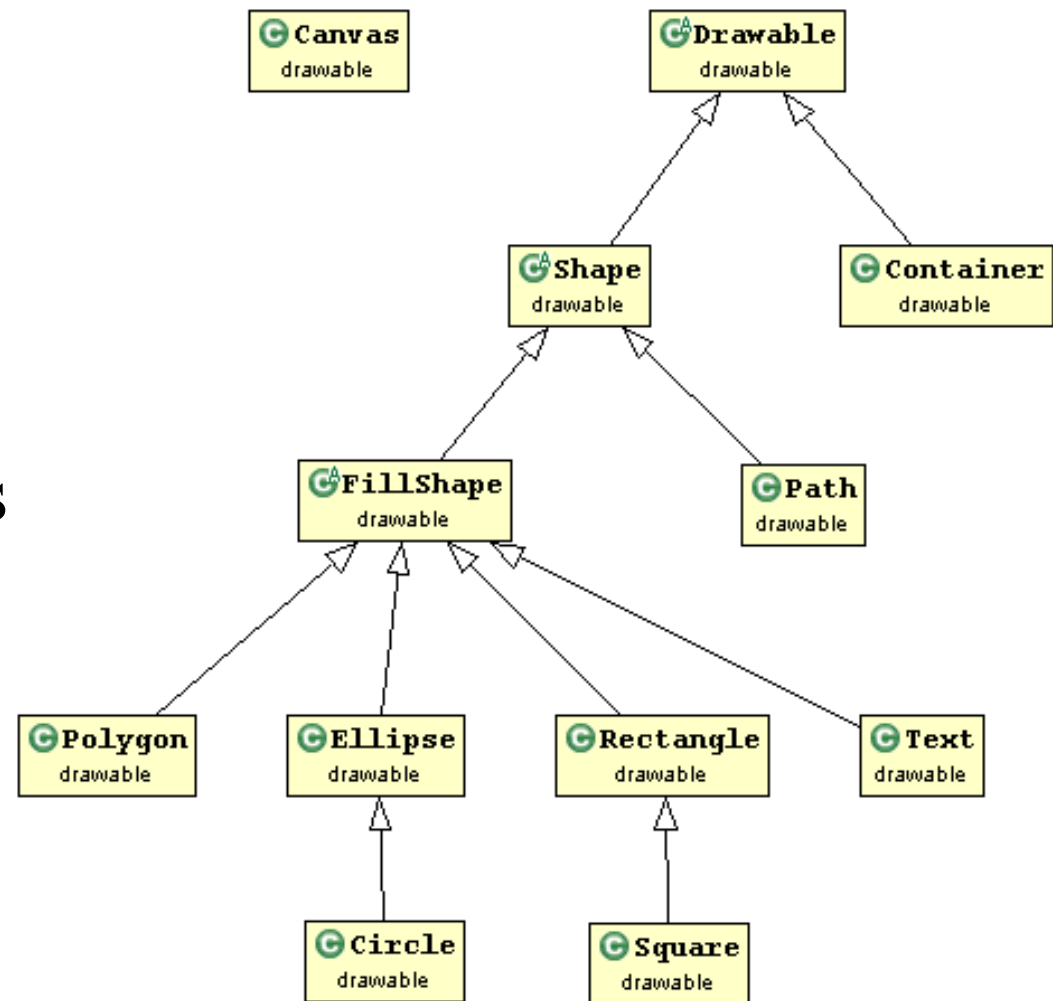
Samsung
TECH INSTITUTE

Contenido

- Herencia
- Constructores y Herencia
- Polimorfismo
- Vinculación dinámica
- Clases abstractas
- Interfaces

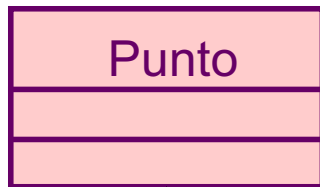
Herencia

- 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 {  
    protected 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 {  
    protected 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;  
    }
```

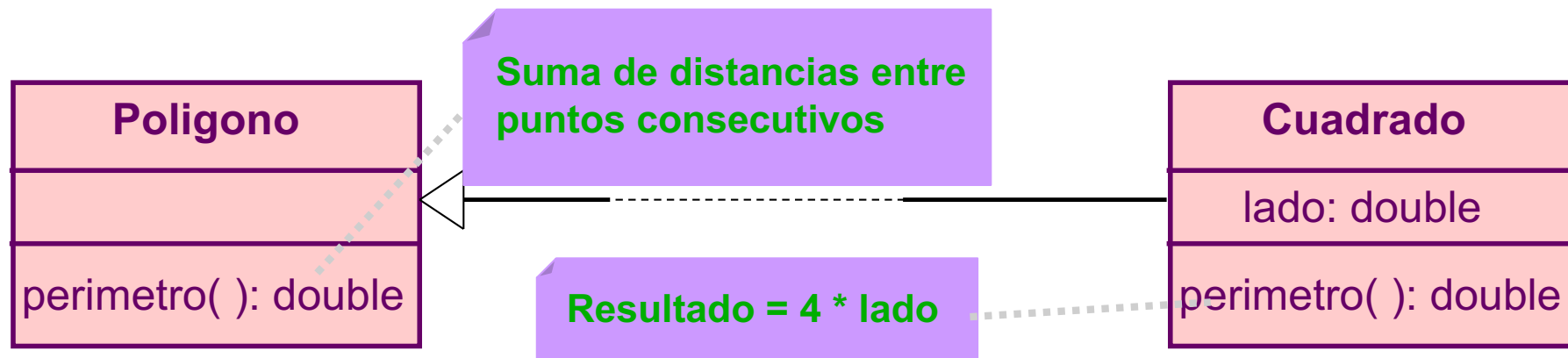
Se refiere a
Punto(double, double)

```
    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

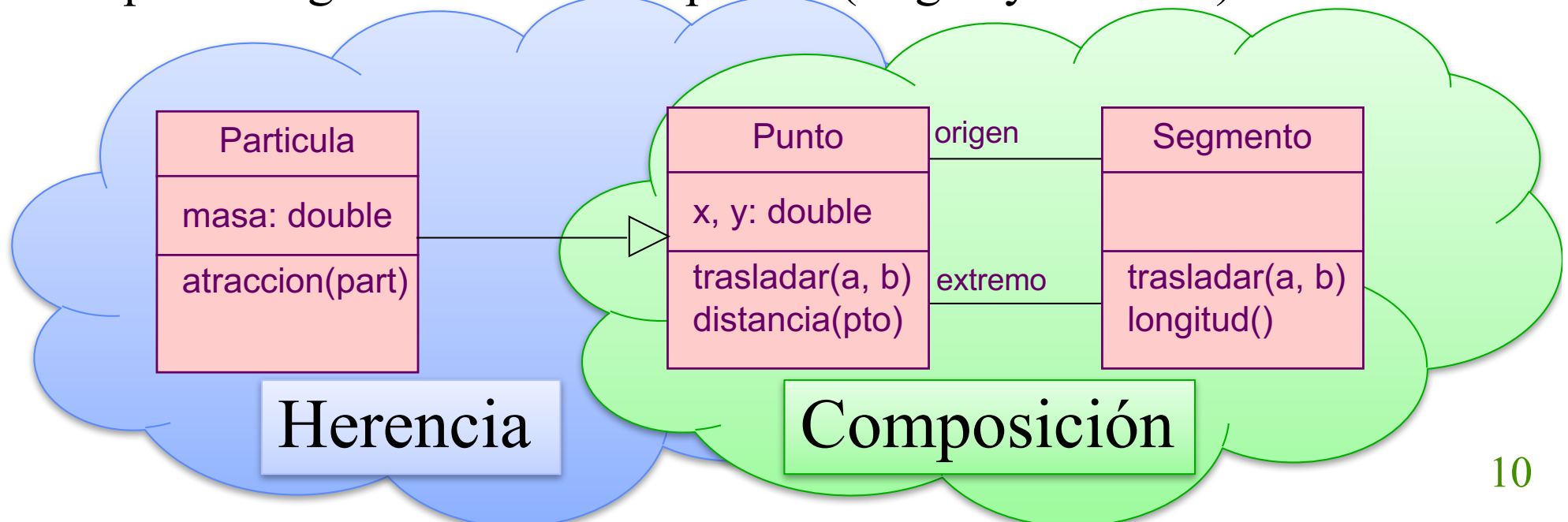
- Es muy corriente la redefinición de un método en la subclase.



- La redefinición puede impedirse mediante el uso del calificador **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)



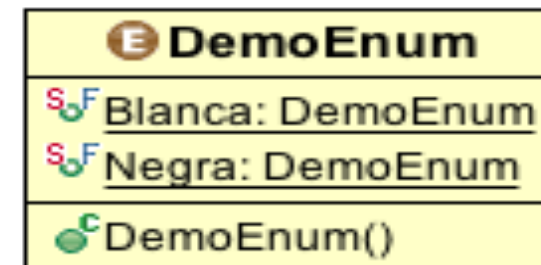
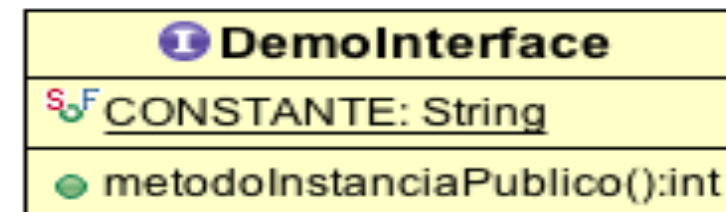
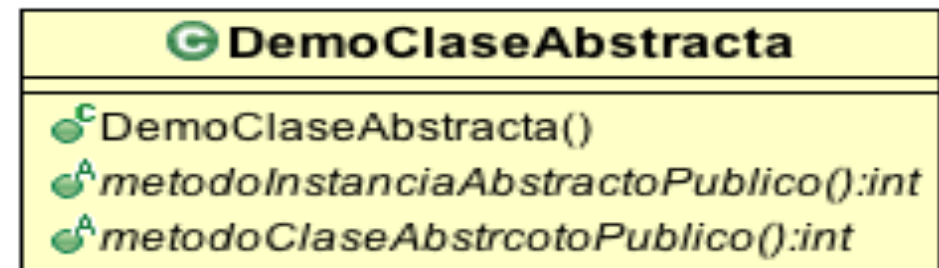
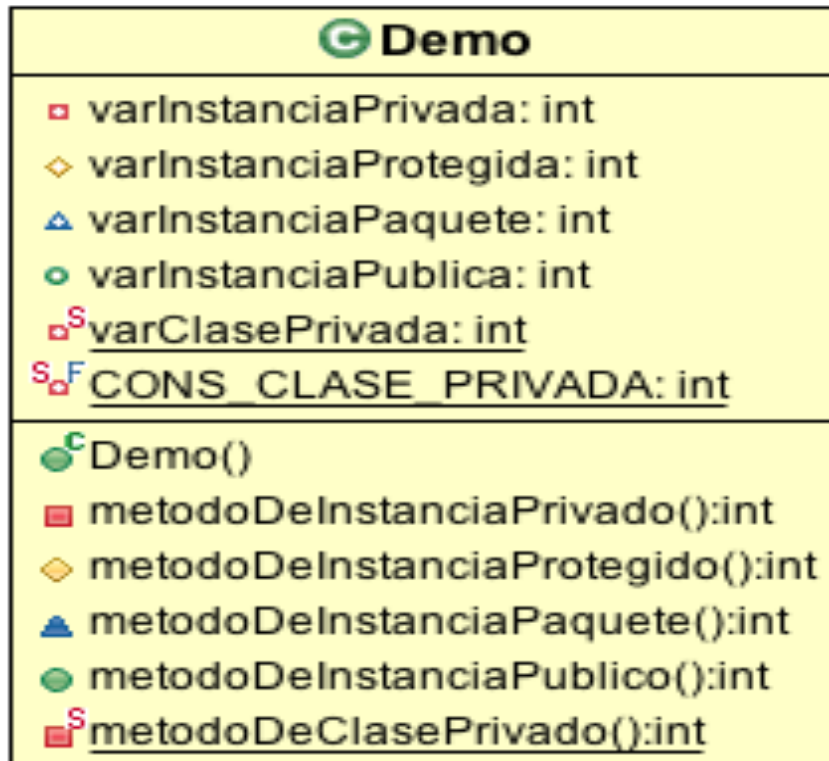
Control de la visibilidad

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
△	-	private	NO	NO	NO	NO
△	#	protected	SÍ	SÍ	SÍ	NO
△	+	public	SÍ	SÍ	SÍ	SÍ
△	~	package	SÍ	SÍ	NO	NO


Símbolos en UML




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);
```



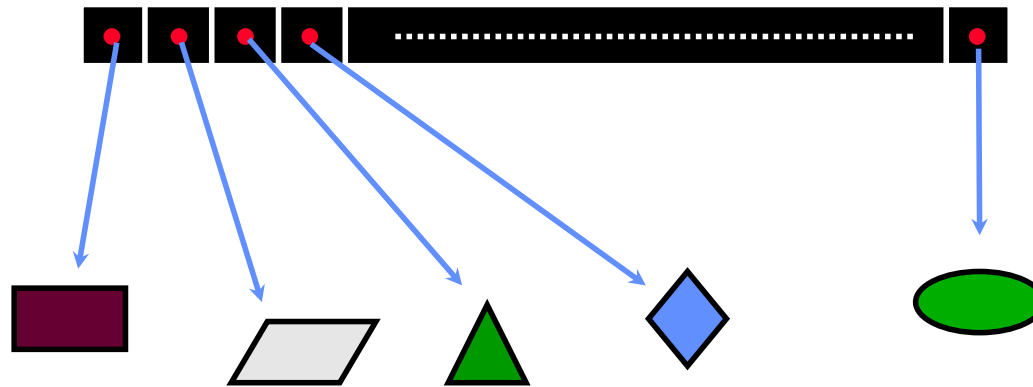
Tipo estático
de **pto**



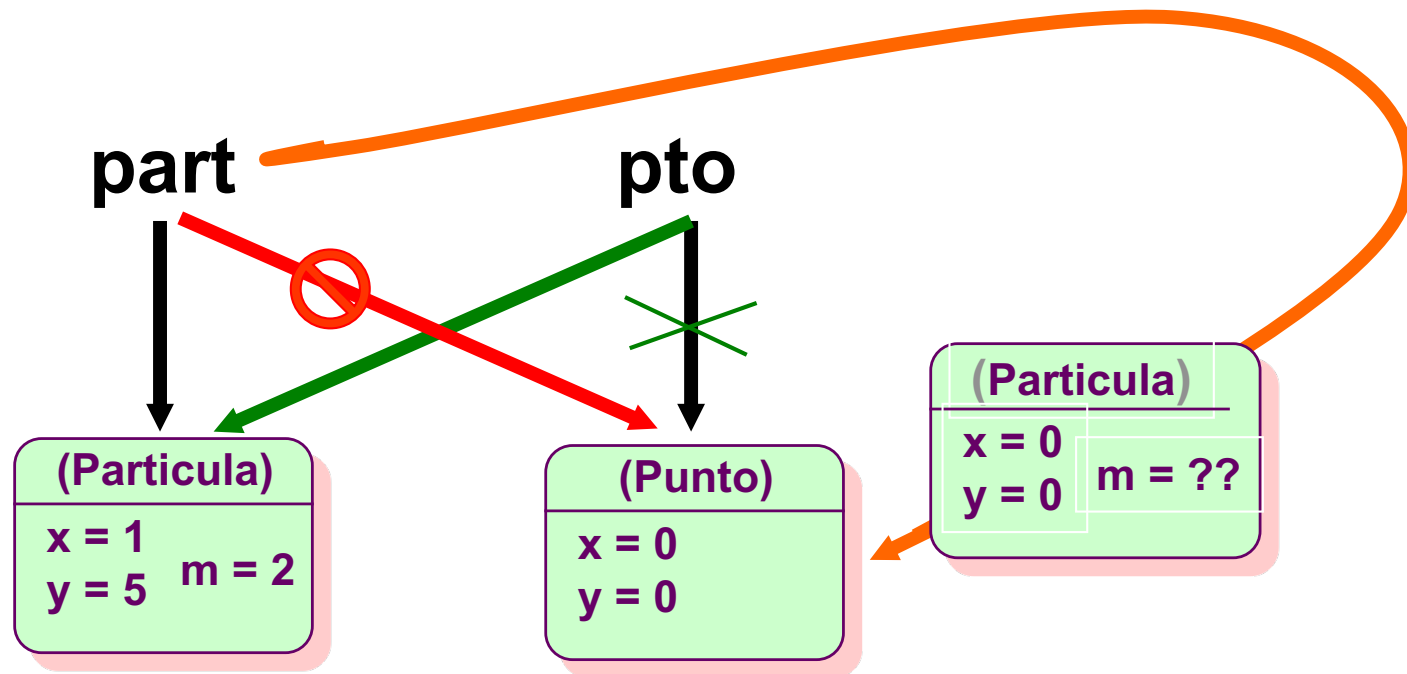
Tipo dinámico
de **pto**

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:

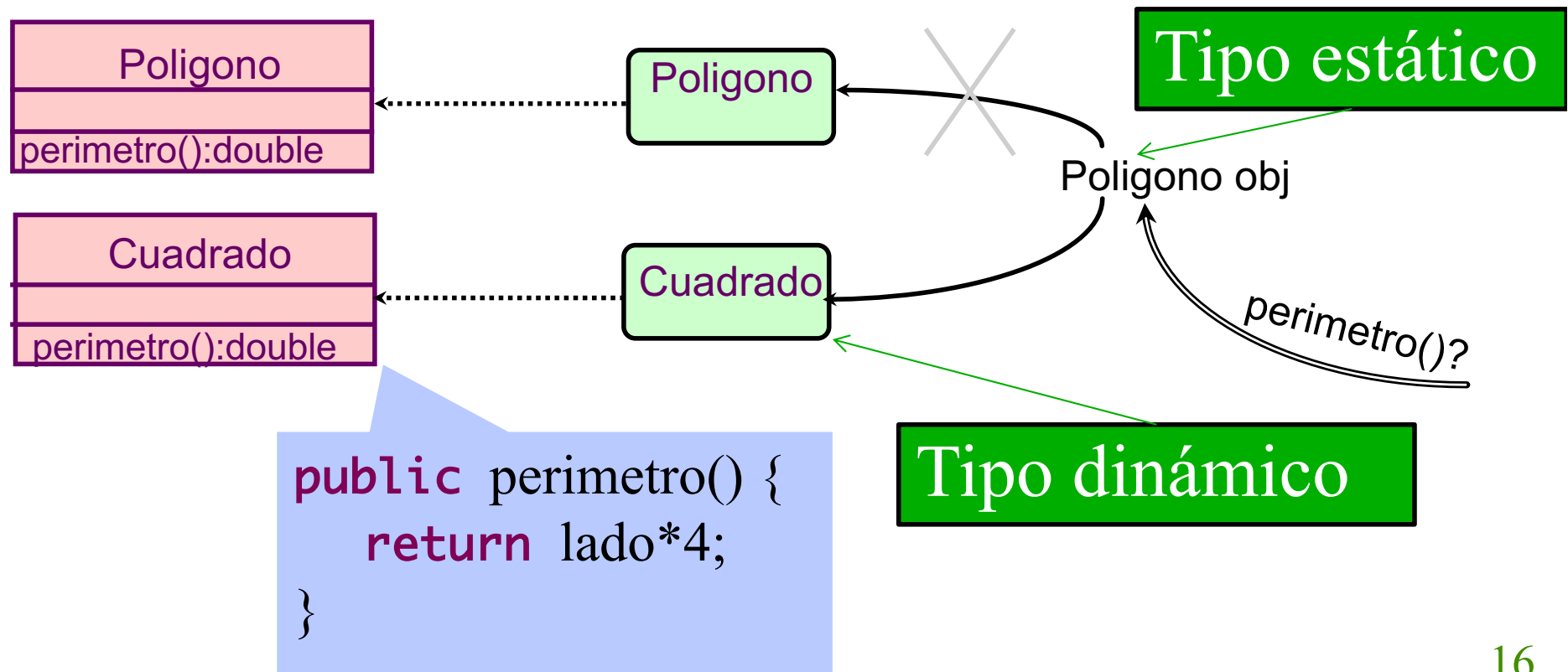


```
Punto pto = new Punto();  
Particula part = new Particula(1,5,2);  
pto = part; // Asignación correcta  
part = pto; // Asignación incorrecta  
part = (Particula) pto; // Peligroso
```



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.



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 subclasses a implementar los métodos que se declararon como abstractos.
 - De lo contrario, esas subclasses 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

Polígono

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

    public Poligono(Punto[] v) {
        vert = v;
    }
    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() {...}
}
```

```
abstract public double area(); // No sabemos calcularla

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() {...}
}
```


Solución 1. Clases abstractas

SecuenciaDeEnteros

```
public abstract class SecuenciaEnteros {
    protected númElementos = 0;

    abstract public void insertar(int pos, int elem);
    abstract public void eliminar(int pos);
    abstract public int maximo();
    abstract public boolean pertenece(int elem);
    public boolean esVacia() { return númElementos == 0; }
    public int tamaño() { return númElementos; }
}
```

```
public class SecuenciaEntEstatica
    extends SecuenciaEnteros {

    private int[] secuencia;

    public SecuenciaEntEstatica(int tam)
        secuencia = new int[tam];
    ...
}

public void insertar(int p, int e) {...}
public void eliminar(int p) {...}
public int maximo() {...}
public boolean pertenece(int e) {...}
public String toString() {...}
}
```

```
public class SecuenciaEntDinamica
    extends SecuenciaEnteros {

    static protected class Nodo {
        int dato;
        Nodo siguiente;
        ...
    }

    private Nodo primero;

    public SecuenciaEntDinamica() {...}

    public void insertar(int p, int e) {...}
    public void eliminar(int p) {...}
    public int maximo() {...}
    public boolean pertenece(int e) {...}
    public String toString() {...}
    public void añadir(SecuenciaEnteros sec) {...}
}
```

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 otra interfaz.
- 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`

`...`

`public abstract`

Métodos por defecto y de clase en interfaces

- Las interfaces se han enriquecido en la versión 1.8 de java incorporando:
 - Métodos por defecto.
 - Métodos de clase.

Métodos por defecto

- Se declaran con **default**.
 - Tienen cuerpo.
- Si una clase implementa la interfaz o una subinterfaz
 - Dispone ya del método por defecto.
 - Si lo desea puede redefinirlo.
 - En este caso, el redefinido prevalece.

Métodos estáticos

- Se declaran con **static**.
 - Tienen cuerpo.
- Puede utilizarse como método de clase desde la propia interfaz o desde la clase que la implemente.

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. Interfaces

SecuenciaDeEnteros

```
public interface SecuenciaEnteros {
    void insertar(int pos, int elem);
    void eliminar(int pos);
    int maximo();
    boolean pertenece(int elem);
    default boolean esVacia() {
        return tamano() == 0;
    }
    int tamano();
}
```

```
public class SecuenciaEntEstatica
    implements SecuenciaEnteros {

    private numElementos = 0;
    private int[] secuencia;

    public SecuenciaEntEstatica(int tam) {
        secuencia = new int[tam];
        numElementos = 0;
        ...
    }

    public void insertar(int p, int e) {...}
    public void eliminar(int p) {...}
    public int máximo() {...}
    public boolean pertenece(int e) {...}
    public int tamano() {...}
    public String toString() {...}
}
```

```
public class SecuenciaEntDinamica
    implements SecuenciaEnteros {

    static protected class Nodo {
        int dato;
        Nodo siguiente;
        ...
    }

    private Nodo primero;

    public SecuenciaEntDinamica() {...}

    public void insertar(int p, int e) {...}
    public void eliminar(int p) {...}
    public int maximo() {...}
    public boolean pertenece(int e) {...}
    public int tamano() {...}
    public void anadir(SecuenciaEnteros sec)
    {...}
    public String toString() {...}
}
```


Uso de interfaces

- Se pueden declarar variables y parámetros de tipo una interfaz.
- Se requieren instancias de clases que implementen la interfaz.

```
SecuenciaEnteros sec;  
sec = new SecuenciaEntEstatica(100);  
... // Inserciones múltiples sobre la secuencia  
  
if (sec.tamaño() == 100) {  
    SecuenciaEnterosDinamica sec1;  
    sec1 = new SecuenciaEnterosDinamica();  
    sec1.anadir(sec);  
    sec = sec1;  
}  
  
... // Nuevas inserciones
```

Interfaz

Objeto de clase
que
implementa la
interfaz

El método anadir tiene
como argumento formal la
interfaz

Clases anónimas e interfaces

- A veces es necesario crea un único objeto perteneciente a una clase y que implemente cierta interfaz. Para resolver el problema tenemos dos opciones:
- Opción 1ª
 - Crear una clase que implemente dicha interfaz:
 - Crear un objeto de esa clase.
- Opción 2ª
 - Crear un objeto de un clase anónima.
- Ejemplo. Sea la interfaz

```
public interface Ejecutable {  
    public void ejecuta();  
}
```

Clases anónimas e interfaces

Opción 1ª

- Crear una clase que implemente dicha interfaz.
- Crear un objeto de esa clase.

```
public class EjecutableImpl implements Ejecutable {  
    public void ejecuta() {  
        System.out.println("Ya está ejecutado");  
    }  
}
```

Clase que
implementa la
interfaz

Objeto único
de esa clase


```
public class Main {  
    public static void main(String [] args) {  
        // opcion 1  
        Ejecutable ej = new EjecutableImpl();  
        prueba(ej);  
    }  
  
    public static void prueba(Ejecutable ej) {  
        ej.ejecuta();  
    }  
}
```

Clases anónimas e interfaces

Opción 2ª

- Crear un objeto de una clase anónima.

```
public class Main {  
    public static void main(String [] args) {  
        // opcion 2  
        prueba(new Ejecutable() {  
            public void ejecuta() {  
                System.out.println("Ya está ejecutado");  
            }  
        })  
    }  
  
    public static void prueba(Ejecutable ej) {  
        ej.ejecuta();  
    }  
}
```



Objeto de una
clase anónima
que
implementa la
interfaz