Programación Orientada a Objetos

TEMA 3

Herencia: Conceptos básicos

Facultad de Informática

Universidad de Murcia

Índice

1.- Introducción

2.- Poligono y Rectangulo

- Herencia y Ocultamiento de Información
- Redefinición de características: refinamiento vs. reemplazo
- Herencia y creación

3.- **Polimorfismo**

4.- Herencia y Sistema de tipos

5.- **Genericidad**

- Estructuras de datos polimórficas
- Genericidad restringida

6.- **Ligadura dinámica**

7.- **Clases abstractas**

- Clases comportamiento: Iteradores

1.-Introducción

Las clases no son suficientes para conseguir los objetivos de:

(A) **REUTILIZACIÓN**

Necesidad de mecanismos para generar **código genérico**:

- Capturar aspectos comunes en grupos de estructuras similares
- Independencia de la representación
- Variación en estructuras de datos y algoritmos

(B) **EXTENSIBILIDAD**

Necesidad de mecanismos para favorecer:

- "Principio abierto-cerrado" y "Principio Elección Única"
- Estructuras de datos polimórficas.

Introducción

- Entre algunas clases pueden existir relaciones conceptuales:

  Extensión, Especialización, Combinación

EJEMPLO:

"Libros y Revistas tienen propiedades comunes"
"Una pila puede **definirse a partir** de una cola o viceversa"
"Un rectángulo es una **especialización** de polígono"
"Una ventana de texto **es un** rectángulo dónde se manipula texto"

¿Tiene sentido crear una clase a partir de otra?

Herencia { soporte para registrar y utilizar estas relaciones
posibilita la definición de una clase a partir de otra

Introducción. Jerarquías de clases

La herencia organiza las clases en una estructura jerárquica:

Jerarquías de clases

Ejemplos:

- No es tan solo un mecanismo para compartir código.
- Consistente con el sistema de tipos del lenguaje

Introducción

- Puede existir una clase "raíz" en la jerarquía de la cual heredan las demás directa o indirectamente.
- Incluye todas las características comunes a todas las clases
- En Java esta clase es la clase `Object`
  - `equals`
  - `clone`
  - `toString`

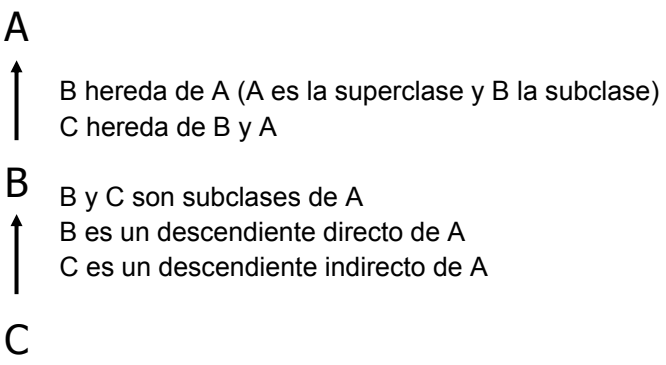
Introducción

Si *B* hereda de *A* entonces *B* incorpora la **estructura** (atributos) y **comportamiento** (métodos) de la clase *A*, pero puede incluir **adaptaciones**:

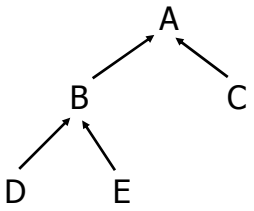
- -B puede **añadir** nuevos **atributos**
- -B puede **añadir** nuevos **métodos**
- -B puede **REDEFINIR métodos**
  - Refinar: Extender el uso original
  - Reemplazar: Mejorar la implementación
- -B puede implementar un método abstracto en A
- …

Adaptaciones dependientes del lenguaje

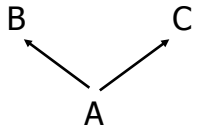
El proceso de la herencia es transitivo



Tipos de herencia



- **Herencia simple**
  - Una clase puede heredar de una única clase.
  - Ejemplo: Java, C#



- **Herencia múltiple**
  - Una clase puede heredar de varias clases.
  - Clases forman un grafo dirigido acíclico
  - Ejemplos: Eiffel, C++

¿Cómo detectar la herencia durante el diseño?

- **Generalización (Factorización)**

  Se detectan clases con un comportamiento común (p.e. `Libro` y `Revista`)

- **Especialización (*Abstracción*)**

  Se detecta que una clase es un caso especial de otra (p.e. `Rectangulo` de `Poligono`)

No hay receta mágica para crear buenas jerarquías

Problemas con la evolución de la jerarquía

Significado de la herencia

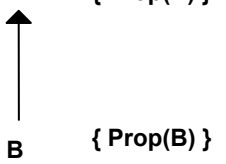
Clases	Herencia	
Modulo	Mecanismo de extensión	Reutilizar características
Tipo	Mecanismo de especialización	Clasificación de tipos

¿relación es-un?

Sean:

Prop(X) : Propiedades (atributos y métodos) de la clase X

dom(C) : Conjunto de instancias de una clase C



- **B extiende la clase A => Prop (A) ⊂ Prop (B)**
- Cualquier objeto de ***B*** puede ser considerado objeto de ***A***
- Siempre que se espera un objeto de ***A*** podemos recibir un objeto de ***B***, puesto que aceptaría todos sus mensajes
- **dom (B) ⊂ dom (A) =>B es un subtipo de A**

El significado de los subtipos

- **El principio de sustitución de Liskov** [B. Liskov 88]:

  "Lo que se quiere aquí es algo como la siguiente propiedad de sustitución: si para cada objeto `o1` de tipo `S` hay un objeto `o2` de tipo `T` tal que para todos los programas `P` definidos en términos de `T`, el comportamiento de `P` no varía cuando se sustituye `o1` por `o2` entonces ***S en un subtipo de T***"

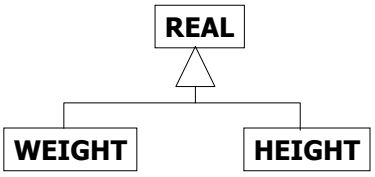
- Funciones que utilizan referencias a superclases deben ser capaces de utilizar objetos de subclases sin saberlo.

Herencia de Implementación vs. Herencia de Tipos

- No siempre se corresponden las clases con tipos
- Dos consideraciones:
  - ¿Cómo relacionamos los tipos?
    - **HERENCIA DE TIPOS o DE COMPORTAMIENTO**
    - Da lugar a jerarquías basadas en aspectos comunes
  - ¿Cómo organizamos las clases para reutilizar código?
    - **HERENCIA DE IMPLEMENTACIÓN o CÓDIGO**
    - Clases no relacionadas pero que tienen código similar
    - Las clases podrían parecer repositorios de código

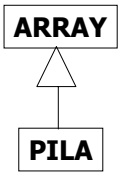
Ejemplos. Herencia de Tipos e Implementación

1) Coincide **herencia de tipos e implementación**

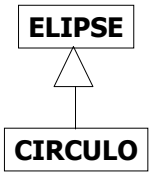


`Weight` y `Height` son tipos de medidas que tienen propiedades en común con los números reales

2) **Herencia de implementación**: todas las propiedades del padre pueden no aplicarse al hijo



3) **Herencia de comportamiento** (especializar un tipo)



2.- Polígonos y Rectángulos

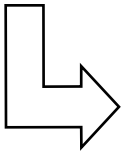
- Tenemos la clase **`Poligono`** y necesitamos representar rectángulos:

¿Debemos crear la clase **`Rectangulo`** partiendo de cero?

Podemos aprovechar la existencia de similitudes y particularidades entre ambas clases

Polígonos y Rectángulos

- Un rectángulo tiene muchas de las características de un polígono (*rotar*, *trasladar*, *vértices*,..)
- Pero tiene características especiales (*diagonal*) y propiedades especiales (*4 lados, ángulos rectos*)
- Algunas características de polígono pueden implementarse más eficientemente *(perímetro)*



```
class Rectangulo extends Poligono{

      ...Características específicas

}
```

Clase Polígono 1/3

```
public class Poligono {
   //Un polígono se implementa como una lista de //puntos
   sucesivos
   private List<Punto> vertices;
   private int numVertices;

   public Poligono(…) {
        vertices = new LinkedList();
   … }

   public int getNumVertices(){
        return numVertices;
   }

   public void rotar (Punto centro,
                              double angulo){…}
   public void trasladar (double a, double b){…}
   public void visualizar(){…}
   public double perimetro(){…}
   …

}
```

Clase Polígono 2/3

```
/**
 * Desplaza a horizontalmente y b verticalmente
 */
public void trasladar (double a, double b){
   for (Punto pto : vertices)
        pto.trasladar(a,b);
}


/**
 * Rota el ángulo alrededor del centro
 */
public void rotar (Punto centro, double angulo){
   for (Punto pto : vertices)
        pto.rotar(centro, angulo);
}
```

Clase Polígono 3/3

```
/**
 * Suma las longitudes de los lados
 */

public double perimetro(){
   double total = 0;
   Punto anterior;
   Punto actual = vertices.get(0);

   for (int index = 1; index < numVertices; index++){
       anterior = actual;
       actual = vertices.get(index);
       total = total + actual.distancia(anterior);
   }

   total=total+actual.distancia(vertices.get(0));

   return total;
}
```


Clase Rectángulo

```
public class Rectangulo extends Poligono{
    private double lado1, lado2;     //Nuevos atributos
    private double diagonal;

    public Rectangulo(double lado1, double lado2){
          super(…);
          this.lado1 = lado1;
          this.lado2 = lado2;
   }
    @Override //Redefine perímetro
    public double perimetro(){
          return (2*(lado1 + lado2 ));
    }
}
```

Todas las características de **Poligono** están disponibles automáticamente para la clase **Rectangulo**, no es necesario que se repitan.

Acceso protegido

- Una subclase no puede acceder a los campos privados de la superclase
- Para permitir que un método de la subclase pueda acceder a un campo de la superclase, éste tiene que declararse como **`protected`**
- **`protected:`** miembros visibles a las subclases y al resto de clases del paquete
- Resumen de modificadores de acceso:
  - `private` –visible sólo en la clase
  - `public` –visible a todas las clases
  - `protected` –visible a las subclases y en el paquete
  - Sin modificador –visible en el paquete

Herencia en Java

- Toda clase hereda directa o indirectamente de la clase **`Object`**
- Una subclase **hereda** de su superclase **métodos** y **variables** tanto de clase como de instancia.
- Una subclase puede **añadir** nuevos métodos y variables.
- Una subclase puede **redefinir** métodos heredados.

Redefinición

- La redefinición reconcilia la **reutilización** con la **extensibilidad**:
  "Es raro reutilizar un componente software sin necesidad de cambios"
- Los **atributos** no se pueden redefinir, sólo se **OCULTAN**
  - el campo de la superclase todavía existe pero no se puede acceder
- Un **método** de la subclase con la **misma signatura** y valor de retorno que un método de la superclase lo está **REDEFINIENDO**.
  - Si se cambia el tipo de los parámetros se está sobrecargando el método original

Redefinición de métodos

- Una clase hija puede redefinir un método de la clase padre por dos motivos:
  - **Reemplazo:**
    - Mejorar implementación.
      - Ej: redefinir `perímetro` en la clase `Rectangulo`.
    - Otra diferente (aunque con la misma semántica).
      - Ej: el método `dibujar` en la jerarquía de `Figura`.
  - **Refinamiento:**
    - Método del padre + acciones específicas.

Refinamiento: super

- Si un método redefinido refina el comportamiento del método original puede necesitar hacer referencia a este comportamiento

- **super** se utiliza para invocar a un método de la clase padre:
  - `super.met();`
  - Se "rompe" la ligadura dinámica
  - No sólo se utiliza en los métodos redefinidos

Ejemplo super

```
public class CuentaAhorro extends Cuenta{
   …
   @Override //Refina el comportamiento heredado

   public void ingresar(double cantidad){
       //Hace lo mismo que el método de la clase padre
       super.ingresar(cantidad);
       //Además hace cosas propias de la CuentaAhorro
       beneficio = cantidad * PORCENTAJE_BENEFICIO;
   }
}
```

¿Qué se puede cambiar en la redefinición?

- Puede **cambiar el nivel de acceso**
  - siempre que lo relaje
  - Puedo pasar de `protected` a `public`
    - Por ejemplo, método `clone` en `Object` es `protected`
  - No se puede pasar de `public` a `private`
    - Hacer un miembro más restrictivo no tiene sentido puesto que podrías saltarte la restricción a través de las asignaciones polimórficas.
- El **tipo de retorno** (tipo covariante)
  - Siempre que el tipo de retorno del método redefinido sea compatible don el tipo de retorno del método original.

Cambiar el nivel de acceso

- Un método público no se puede ocultar a los clientes en las subclases (no puede redefinirse como privado)
- Se puede redefinir lanzando una excepción para indicar que la operación no se soporta en la subclase.
  - El mecanismo de excepciones lo veremos en el tema 4.
- El compilador no avisará de que es método no se puede ejecutar.
- Los errores ocurrirán en tiempo de ejecución si se intenta invocar a un método no soportado.

Ejemplo: Operación no soportada

```
public class Pila<T> extends LinkedList<T>{

   //Operaciones propias de Pila

   public void push(T item){ … }
   public T pop(){ … }
   public T top() { … }

   //Operaciones de Lista con accesos aleatorios

   public void add (int index, T objeto){
       throw new NotSupportedOperationException();
   }
   …
}
```

Puedo utilizar add en Pila sin que me de error??

Tipos de retorno covariante desde JDK 1.5

- Es posible que una subclase modifique el tipo de retorno de una función y cambiarlo por un subtipo del original.

- **Ejemplos:**
  a) Redefinición del método clone:
     - `Object >> protected Object clone() …`
     - `Punto3D >> public Punto3D clone() …`
  b) Jerarquía de Empleado
     - `Empleado >> public Empleado getColega() …`
     - `Jefe >> public Jefe getColega() …`

Herencia y creación en Java

- **La primera** sentencia del constructor de la clase hija **SIEMPRE** es una llamada al constructor de la clase padre.
  - El constructor de la clase hija refina el comportamiento del padre
- Se puede incluir una llamada **explícita:**
  - `super();`
  - `super(a,b);`

  Dependiendo de si el constructor al que invocamos tiene o no argumentos
- Si se omite, se llamará **implícitamente** al constructor por defecto
  - Equivale a poner como primera sentencia `super();`
  - Si no existe el constructor por defecto en la clase padre dará un error

3.- Polimorfismo

- El término **polimorfismo** significa que hay un nombre (variable, función o clase) y muchos significados diferentes (distintas definiciones).
- Formas de polimorfismo [Budd'02]:
  - Polimorfismo de asignación (*variables polimorfas*)
  - Polimorfismo puro (*función polimorfa*)
  - Polimorfismo ad hoc (*sobrecarga*)
  - Polimorfismo de inclusión (*redefinición*)
  - Polimorfismo paramétrico (*genericidad*)

Polimorfismo de asignación y puro

Capacidad de una entidad de referenciar en tiempo de ejecución a instancias de diferentes clases.

- **Es restringido por la herencia**
- Importante para escribir código genérico
- Sea las declaraciones:

```
X ox; rutina1(Y oy)
```

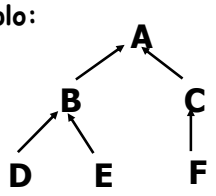
  - En un lenguaje con **monomorfismo** (Pascal, Ada, ..) en t.e. **ox** y **oy** denotarán valores de los tipos **X** e **Y**, respectivamente.
  - En un lenguaje con **polimorfismo** (Java, ..) en t.e. **ox** y **oy** podrán estar asociados a objetos de varios tipos diferentes:

tipo estático vs. tipo dinámico

Tipo estático y tipo dinámico

- **Tipo estático:**
  - Tipo asociado en la declaración
- **Tipo dinámico:**
  - Tipo correspondiente a la clase del objeto conectado a la entidad en tiempo de ejecución
- **Conjunto de tipos dinámicos:**
  - Conjunto de posibles tipos dinámicos de una entidad

Ejemplo:

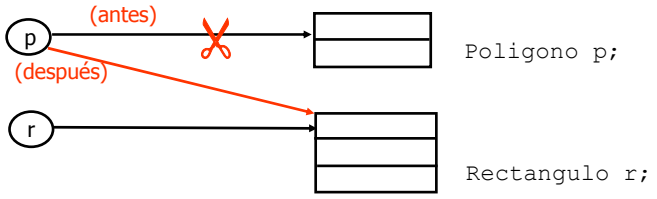



```
A oa; B ob; C oc;
```

te(oa) = A	ctd(oa) = {A,B,C,D,E,F}
te(ob) = B	ctd(ob) = {B, D, E}
te(oc) = C	ctd(oc) = {C,F}

Entidades y rutinas polimorfas

Conexión polimorfa: el origen y el destino tiene tipos diferentes



a) asignación:

```
p = r;
```

-- p es una **entidad polimorfa** ***(polimorfismo de asignación)***

b) paso de parámetros:

```
void f (Poligono p){
...
}
```

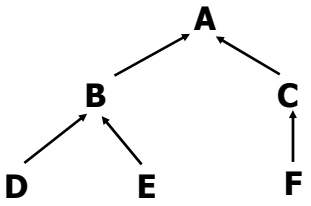
-- f es una **rutina polimorfa** ***(polimorfismo puro)***

- **Sólo** se permite para entidades destino de **tipo referencia**

Polimorfismo puro vs. Sobrecarga

- Funciones sobrecargadas ≠ funciones polimórficas
- Sobrecarga:
  - Dos o mas funciones comparten el nombre y distintos argumentos (nº y tipo). **El nombre es polimórfico**.
  - Distintas definiciones y tipos (distintos comportamientos)
  - Función correcta se determina en **tiempo de compilación** según la signatura.
- Funciones polimórficas:
  - Una única función que puede recibir una variedad de argumentos (comportamiento uniforme).
  - La ejecución correcta se determina dinámicamente en **tiempo de ejecución.**

4.- Herencia y sistema de tipos




```
A oa; B ob; C oc; D od;
```

¿Son legales las siguientes asignaciones?

```
oa = ob; oc = ob; oa = od
```

¿Es legal el mensaje `od.metodo1`?

Herencia y Sistema de Tipos

Un lenguaje OO tiene **comprobación estática de tipos** si está equipado con un cjto de **reglas de consistencia**, cuyo cumplimiento es controlado por los **compiladores**, y que si el código de un sistema las cumple se garantiza que ninguna ejecución de dicho sistema puede provocar una violación de tipos

- **Política pesimista**:

"al tratar de garantizar que ninguna operación fallará, el compilador puede rechazar código que tenga sentido en tiempo de ejecución"

Ejemplo: (Pascal) n:INTEGER; r: REAL ⇒ n:=r 💣

n:= 0.0	Podría funcionar
n:= -3.67	No funcionaría
n:= 3.67 - 3.67	Funcionaría

- **Beneficios esperados:** Fiabilidad, legibilidad y eficiencia

Reglas básicas

- La herencia es consistente con el sistema de tipos

Regla de llamada a una característica

En una llamada a una característica **`x.f`** donde el tipo de `x` se basa en una clase `C`, la característica `f` debe estar definida en uno de los antecesores de `C`.

Luego, sean las declaraciones

```
Poligono p ; Rectangulo r;
```

Mensajes legales:	Mensajes ilegales:

`p.perimetro(); p.getVertices() ;`
`p.trasladar(..); p.rotar (..);`
`r.getDiagonal();r.getLado1();`
`r.getLado2();`
`r.getVertices(); r.trasladar(..);`
`r.rotar (..); r.perimetro();`
	`p.getLado1();`
	`p.getLado2();`
	`p.getDiagonal();`

Reglas básicas

- La herencia regula que conexiones polimorfas son permitidas

Definición: compatibilidad o conformidad de tipos

Un tipo **`U`** es compatible o conforme con un tipo **`T`** sólo si la clase base de `U` es un descendiente de la clase base de `T`, además, para los tipos derivados genéricamente, todo parámetro real de `U` debe (recursivamente) ser compatible con el correspondiente parámetro formal en `T`.

- Por ejemplo, `Rectangulo` es compatible con `Poligono`

Reglas básicas

Regla de compatibilidad de tipos

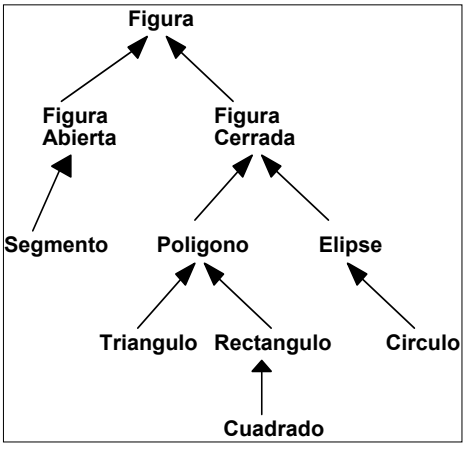
Una conexión con origen **y** y destino **x** (esto es, una asignación **x=y**, o invocación **r(..,y,..)** a una rutina **r(.., T x,..)** es válido solamente si el tipo de **y** es compatible con el tipo de **x**.

Regla de validez de un mensaje

Un mensaje **ox.rut (y),** supuesta la declaración **X ox;** será legal si i) **X** incluye una propiedad con nombre final **rut**, ii) los argumentos son compatibles con los parámetros y coinciden en número, y iii) **rut** está disponible para la clase que incluye el mensaje.

Ejemplo

```
Poligono p; Rectangulo r; Triangulo t;... double x;
```



SERÍA CORRECTO EL CODIGO

```
x = p.perimetro();
x = r.perimetro();
x = r.getDiagonal();
if (test) p = r
else p = t
x = p.perimetro();
```

SERÍA INCORRECTO

```
x = p.getDiagonal();
r = p;
```

¿Están las restricciones justificadas?

- ¿Tiene sentido que el compilador rechace el siguiente código:

  c1) `p = r; r = p`
  c2) `p = r; x = p.getDiagonal()`

  ?

- Situaciones poco probables (sin sentido).
- Asignaciones como **`p = r`** son normalmente ejecutadas como parte de alguna estructura de control que dependa de condiciones "run-time", tal como la entrada de un usuario.
- Lo normal es encontrar situaciones en las que se desconoce el tipo exacto de una entidad.

A)
```
Poligono p;
   Rectangulo r = new Rectangulo(…);
  Triangulo t = new Triangulo (…); ...
   pantalla.visualizarIconos();
   pantalla.esperarClickRaton();
   double x = pantalla.posicionRaton();
   iconoElegido = pantalla.getIconoEn(x);
   if (iconoElegido == iconoRectangulo) p = r
   else if (iconoElegido == iconoTriangulo) p = t
        else if  … //tantos if como tipos de figuras
       ...
   //Aplicar operaciones de Poligono
   p.visualizar(); p.rotar(...);
```

B)
```
void unMetodo(Poligono p){
      … p.met() //siendo met un método de Poligono
   }
```

- En ambas situaciones, donde se desconoce el tipo exacto de `p`:
  - sólo tiene sentido aplicar operaciones generales de `POLIGONO`.
  - es cuando tiene sentido el uso de entidades polimorfas como `p`

5.- Genericidad

- Hasta la versión 1.4 no se incluye la genericidad como elemento del lenguaje.
- Se consigue gracias a que **toda clase es compatible con la clase `Object`**.
  - Las colecciones son contenedores de objetos
- Problemas => Se pierde la información del tipo:
  - La colección puede contener cualquier tipo.
  - Hay que efectuar un **cast** antes de utilizar el objeto que se recupera de la colección.
  - Se detecta un objeto del tipo no deseado en tiempo de ejecución.

Genericidad en Java hasta JDK1.4

```
public class Pila {
  private ArrayList contenido; //contenedor de Object
  public void push (Object obj){...}
  public Object pop () {...}
}
```

```
Pila p; //quiero que sea de Movimientos de Cuenta
Movimiento m; Animal a;
p.push (m);
p.push(a);                    //NO daría error el compilador
m=p.pop();                    //error asignamos un Object
m=(Movimiento)p.pop()         //OK
```

Perdemos la ventaja de la comprobación estática de tipos, las comprobaciones las hace el programador

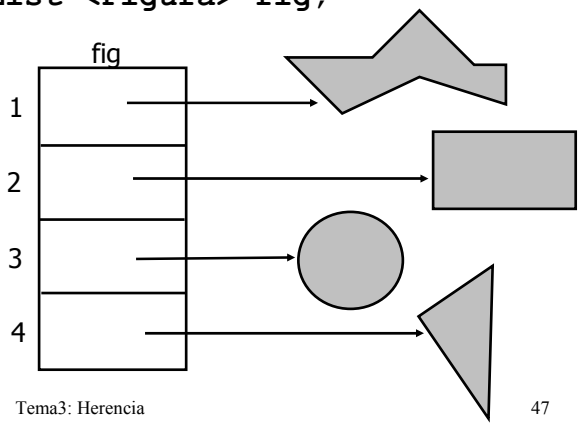
Genericidad y polimorfismo

- **Estructuras de datos polimorfas**:

  Pueden contener instancias de una jerarquía de clases

- Ejemplo: **`ArrayList <Figura> fig;`**

```
Poligono p;
Rectangulo r;
Circulo c;
Triangulo t;
//se crean …
fig.add(0,p);
fig.add(1,r);
fig.put(2,c);
fig.put(3,t);
...
```



Tipo dinámico de los objetos

- Las estructuras condicionales en las que se pregunta por el tipo de los objetos van en contra de los principios de Elección Única y Abierto-Cerrado.

```
if "f es de tipo RECTANGULO" then
    …
elseif "f es de tipo CIRCULO" then
    ...
```

- Sin embargo, como consecuencia del manejo de estructuras de datos polimorfas puede ser inevitable tener que preguntar por el tipo dinámico de los objetos.

Ejemplo:

```
public float maxDiagonal (LinkedList<Figura> listaFig) {
   Figura f;
   double actual,result=-1;
   for (Figura figura : listaFig){
        if (figura instanceof Rectangulo){
             actual = (Rectangulo)figura.getDiagonal();
             if (actual>result) result = actual;
        }
    }
    return result;
}
```


¿Qué ocurre si quiero sumar vectores?

```
public class Vector <T> {
   private int count;
   public T item (int i){
       …
   }
   public void put(T v, int index){
    …
   }
   public Vector<T> suma (Vector<T> otro){
       Vector<T> result = new Vector<T>();
       for (int i=1; i <= count; i++)
             result.put(this.item(i) + other.item(i),i);
       return result;
   }
}
```

¿Se pueden sumar dos objetos de tipo T?

¿Es posible ampliar el nº de operaciones?

Solución: Genericidad Restringida

- Es posible restringir las clases a las que se puede instanciar el parámetro genérico formal de una clase genérica.

  `public class C <T extends C>`

- Sólo es posible instanciar T con descendientes de la clase C.
- Las operaciones permitidas sobre entidades de tipo T son aquellas permitidas sobre una entidad de tipo C.
- **Ejemplos:**

```
class Vector <T extends Number>
class Dictionary <G, H extends Comparable>
class ListaOrdenada <T extends Comparable &
                                Serializable>
```


Genericidad Restringida

- La genericidad no restringida equivale a:

  `<T extends Object>`

- ¿Sería legal la declaración:

  `Vector<Vector<Number>> vectorDeVectores;` ?

- ¿Son equivalentes las declaraciones:
  - **`Vector<Number>`**
  - **`Vector<T extends Number>`**

  ?

Ejercicio: ¿Son equivalentes estas dos definiciones de la clase Vector?

Eiffel	Java

`class Vector<T extends Number>{`
`  private T [] contenido;`
 	
`  public void insertar(T n,`
`  int i){`
`  ...`
`  }`
`  ...`
`}`
	`public class Vector{`
	`private Number[] contenido;`
	
	`public void insertar(Number n,`
	`  int i){`
	`  ...`
	`}`
	`...`
	`}`

Instanciar el tipo genérico al heredar

- La subclase puede seguir siendo un tipo genérico:

  `class ArrayOrdenado<T> extends ArrayList<T>{ … }`

- Al heredar de una clase genérica se puede instanciar el tipo:

  `class Caja <T extends ObjetoValor> { … }`

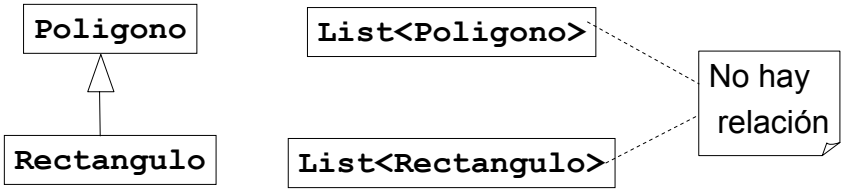
  `class CajaSeguridad extends Caja<Joya> { … }`

  Siendo `Joya` una subclase de `ObjetoValor`

Genericidad y máquina virtual

- La máquina virtual no tiene objetos de tipo genérico
- Todo tipo genérico se transforma en un *tipo puro*
  - Se elimina el parámetro de tipo
    - `List<T>` se sustituye `T` por `Object`
    - `List<T extends Comparable>` sustituye la `T` por `Comparable`
- Todas las consultas sobre el tipo dinámico siempre devuelven el tipo puro:
  - `(lp instanceof Lista<Poligono>)` equivale a `(lp instanceof Lista)`
  - `lp1 = (List<Poligono>)l;` //sólo comprueba si `l` es una `List`

Genericidad y el sistema de tipos




- ```
  List<Poligono> lp;
  List<Rectangulo> lr;
  lp = lr; //ERROR
  ```
- Si se puede asignar al tipo puro

  ```
  List lista = lp; //OK
  ```
  ```

Tipos comodín

- Dado el código de la clase `Paint`:

```
public static void imprimir (List<Poligono> poligonos){
      for (Poligono p : poligonos)
            System.out.println(p);
}
```

- El siguiente código daría error:

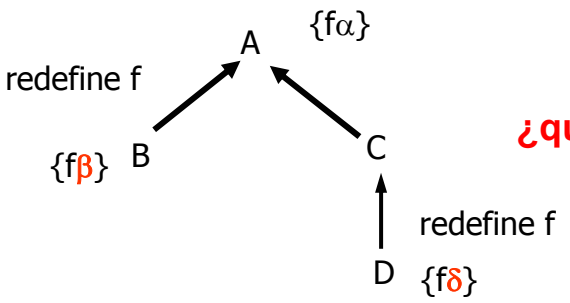
```
List<Rectangulo> lr = new LinkedList<Rectangulo>(); …
Paint.imprimir(lr); //Error no es una List<Poligono>
```

- Hay que usar el comodín en el argumento:

```
void imprimir (List<? extends Poligono> poligonos)
```

- Con el comodín el código anterior no da error.

6.- Ligadura dinámica




```
A oa;
oa.f(..);
```

¿qué versión de f se ejecuta?

Regla de la ligadura dinámica

La forma dinámica del objeto determina la versión de la operación que se aplicará.

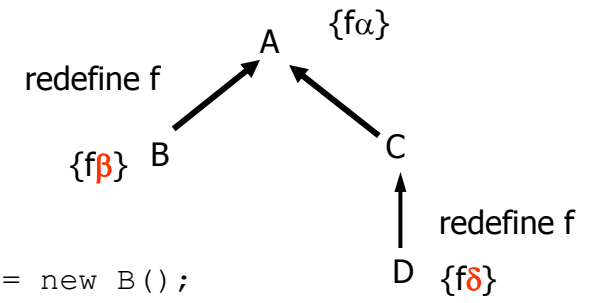
Ligadura dinámica

- La **versión** de una rutina en una clase es la introducida por la clase (redefinición u original) o la heredada.
- **Ejemplo 1:**

```
Poligono p = new Poligono();
Rectangulo r = new Rectangulo();

x = p.perimetro() →  perimetro_POLIGONO
x = r.perimetro() →  perimetro_RECTANGULO
p = r;
x = p.perimetro() →  perimetro_RECTANGULO
```


Ejercicio: ¿qué versión se ejecuta?




```
A oa;
B ob = new B();
D od = new D();
oa = ob;
oa.f();

oa = od
oa.f();
```

Ligadura Dinámica

Ejemplo 2:

```
void visualizar (Figura [] figuras){
        for (Figura figura : figuras)
            figura.dibujar();
}
```

¿Qué sucede si aparece un nuevo tipo de figura?

- ¿Qué relación existe entre ligadura dinámica y comprobación estática de tipos?

Sea el mensaje **x.f ()**, la **comprobación estática de tipos** garantiza que al menos existirá una versión aplicable para **f,** y la **ligadura dinámica** garantiza que se ejecutará la versión más apropiada

Ejemplo ligadura dinámica y `super`

```
class Uno {
    public int test(){return 1;}
    public int result1(){return this.test();}
}

class Dos extends Uno{
    public int test(){return 2;}
}

class Tres extends Dos{
    public int result2(){return this.result1();}
    public int result3(){return super.test();}
}

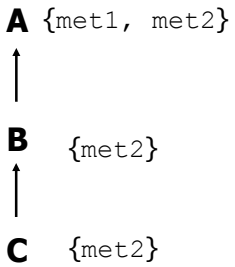
class Cuatro extends Tres{
    public int test(){return 4;}
}
```

Ligadura dinámica y `super`

```
public class PruebaSuperThis{
   public static void main (String args[]){
      Uno ob1 = new Uno();
      Dos ob2 = new Dos();
      Tres ob3 = new Tres();
      Cuatro ob4 = new Cuatro();

      System.out.println("ob1.test = "+ ob1.test());                  1
      System.out.println("ob1.result1 = " + ob1.result1());           1
      System.out.println("ob2.test = " + ob2.test());                 2
      System.out.println("ob2.result1 = " + ob2.result1());           2
      System.out.println("ob3.test = " + ob3.test());                 2
      System.out.println("ob4.result1 = " + ob4.result1());           4
      System.out.println("ob3.result2 = " + ob3.result2());           2
      System.out.println("ob4.result2 = " + ob4.result2());           4
      System.out.println("ob3.result3 = " + ob3.result3());           2
      System.out.println("ob4.result3 = " + ob4.result3());           2
   }
}
```

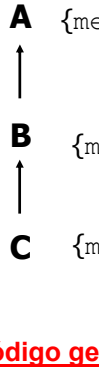

Patrones para conseguir código genérico




```
1) void met1(){
      ...
      this.met2();
      -- this es una entidad polimorfa
      ...
   }

2) void met3 (A p){
      ...
      p.met2();
      ...
   }
```

Patrones para conseguir código genérico




```
3) void met1(A p){
       ...
    this.met2();
    p.met2();
       ...
   }

4) void met3 (A[] tabla){
      for (A a: tabla)
           a.met2();
   }
```

Código genérico:

- Un único código con diferentes interpretaciones en tiempo de ejecución.
- Un mismo código con diferentes implementaciones:

"variación en la implementación"

Ligadura dinámica y eficiencia

- **¿Tiene un coste de ejecución inaceptable?**

  `x.f(a,b,c,…)`

  - la característica `f` depende del tipo del objeto al que esté conectado `x`
  - el tipo de `x` no se puede predecir a partir del texto software
- Lenguajes SIN comprobación estática de tipos
  - Si `f` no está en el tipo del objeto al que está conectado `x` se busca en el padre recursivamente
  - Se puede tener que recorrer todo el camino hasta la raíz
  - la penalización **es** IMPREDECIBLE
  - crece con la profundidad de la estructura de herencia
  - conflicto entre reutilización y eficiencia

Ligadura dinámica y eficiencia

- Lenguajes **CON comprobación estática de tipos**
  - Los tipos posibles para `x` controlados por la herencia
  - se reduce a los descendientes del tipo estático de `x`
  - El compilador puede preparar una estructura basada en arrays que contenga la información de tipos necesaria
  - El coste de la ligadura dinámica es **CONSTANTE**
    - cálculo de índice y acceso a un array
  - No hay que preocuparse de eficiencia y reutilización
  - Cierto tanto para herencia simple como múltiple

7.- Clases abstractas

Sea la declaración

```
Figura f; Poligono p;
```

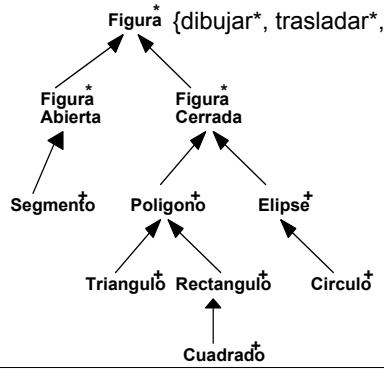
y el código

```
p = new Poligono(…);
f = p;
f.dibujar();
```

¿Sería legal?

- ¿Cómo se implementa **`dibujar`** en la clase **`Figura`**?
- La rutina **`dibujar`** no puede ser implementada en **`Figura`** pero **`f.dibujar`** es **¡dinámicamente correcto!**
- ¿Tendría sentido incluir **`dibujar`** en **`Figura`** como una rutina que no hace nada?

Solución: Métodos abstractos



En `Figura`:

```
abstract void dibujar();
```

Clases abstractas

```
public abstract class Figura {...}
```

- Una subclase de una clase diferida puede seguir siendo diferida

```
public abstract class FIGURA_ABIERTA extends Figura { ... }
public abstract class FIGURA_CERRADA extends Figura { ... }
```


Clases abstractas

- Toda clase que contenga algún método abstracto (heredado o no) es abstracta.
- Una clase puede ser abstracta y no contener ningún método abstracto.
- Especifica una **funcionalidad que es común** a un conjunto de subclases aunque no es completa.
- Puede ser total o parcialmente abstracta.
- **No es posible crear instancias** de una clase abstracta, pero si declarar entidades de estas clases.
  - Aunque la clase puede incluir la definición del constructor.
- Las clases abstractas **sólo tienen sentido en un lenguaje con comprobación estática de tipos**.

Clases parcialmente abstractas

- Contienen métodos abstractos y efectivos.
- Los métodos efectivos pueden hacer uso de los abstractos.
- Importante mecanismo para incluir **código genérico**.
- Incluyen comportamiento abstracto común a todos los descendientes.

"programs with holes"

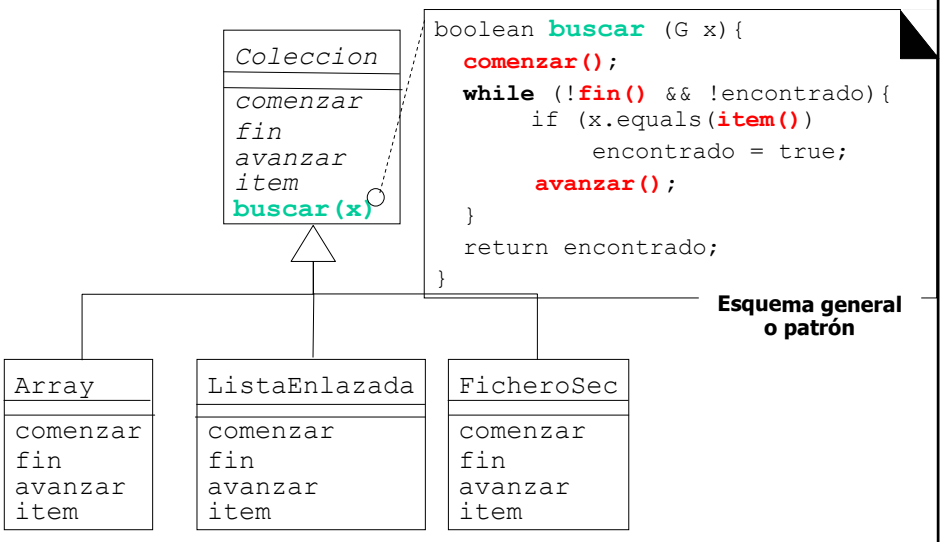
Clases parcialmente abstractas

- "Permiten **capturar lo conocido** sobre el comportamiento y estructuras de datos que caracterizan a cierta área de aplicación, **dejando una puerta abierta a la variación**:

  SON UNA CONTRIBUCION IMPORTANTE DE LA OO A LA REUTILIZACIÓN" [B. Meyer]

- Reciben el nombre de **CLASES COMPORTAMIENTO** aquellas clases que incluyen un **comportamiento común** a varias subclases ("familia de implementaciones de TAD´s")

Clases parcialmente abstractas

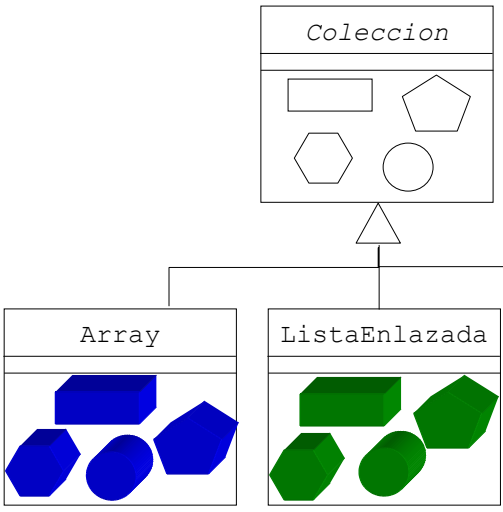


Método plantilla - Clase comportamiento

```
public abstract class Coleccion<G> {

  public boolean buscar (G x){
      comenzar();
      while (!fin() && !encontrado){
            if (x.equals(item()))
                  encontrado = true;
            avanzar();
      }
      return encontrado;
  }
  public abstract void comenzar();
  public abstract boolean fin();
  public abstract G item();
  public abstract void avanzar();
}
```

"No nos llame, nosotros le llamaremos"




```
Array<Integer> a;
a = new Array();
a.buscar(31);
```

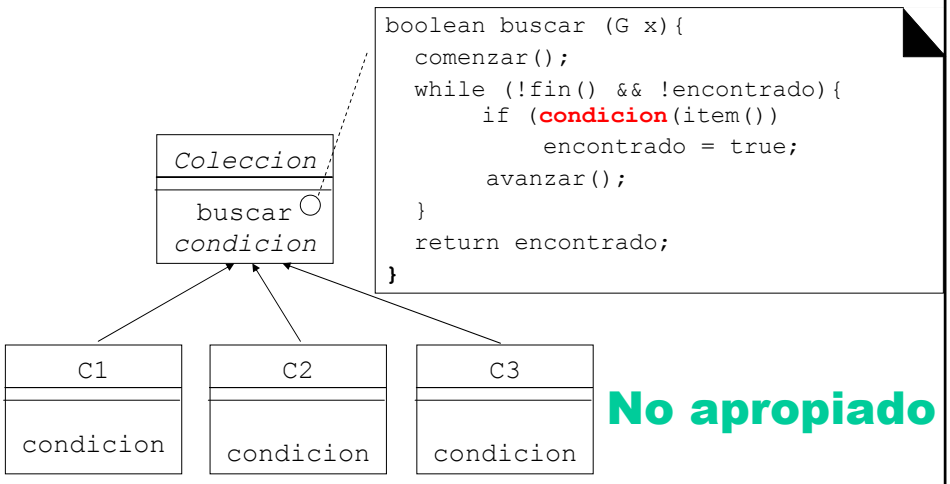

Parametrizar una rutina con una acción

- ¿Cómo podemos buscar si existe un elemento que cumple una condición?

  `buscar(condicion)`

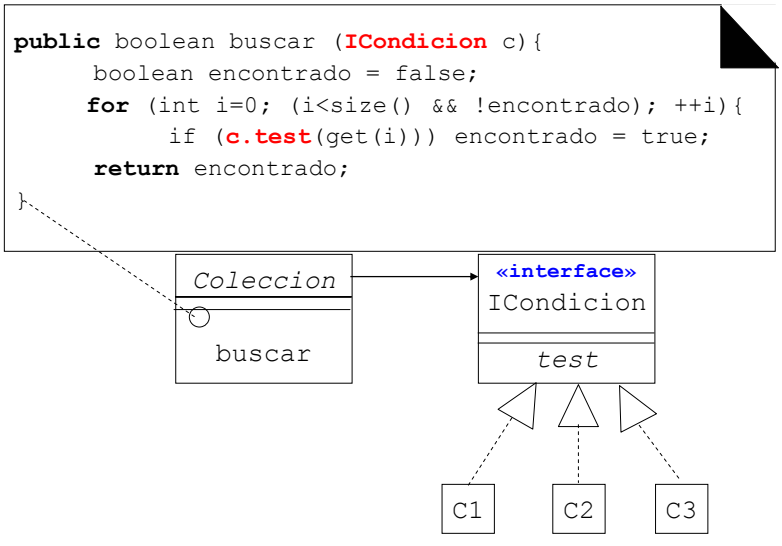
- Dos soluciones:
  a) Herencia → Método plantilla
  b) Composición

Parametrizar una rutina con una acción.
a) Método plantilla



No apropiado

Parametrizar una rutina con una acción.
b) Composición



Parametrizar una rutina con una acción. b) Composición

```
interface ICondicion<T>{
      boolean test(T objeto);
}

public class Contacto{
   private String nombre;
   private String nick;
   private double telefono;
   …
}
```

«Interface»

- Equivale a una clase totalmente abstracta
- **Sólo** contiene definiciones de métodos y constantes
- Todo es `public`
- No se puede crear un objeto del tipo de la interfaz, pero si utilizarlo en la declaración de variables.

```
List<Contacto> agenda=new LinkedList<Contacto>();
```

EJERCICIO: Suponiendo que el método buscar(ICondicion) está disponible para toda colección, utilizarlo para consultar si existe en la Agenda un contacto que se llame "Yago".

Parametrizar una rutina con una acción. b) Composición

1º) Implementar la condición de búsqueda

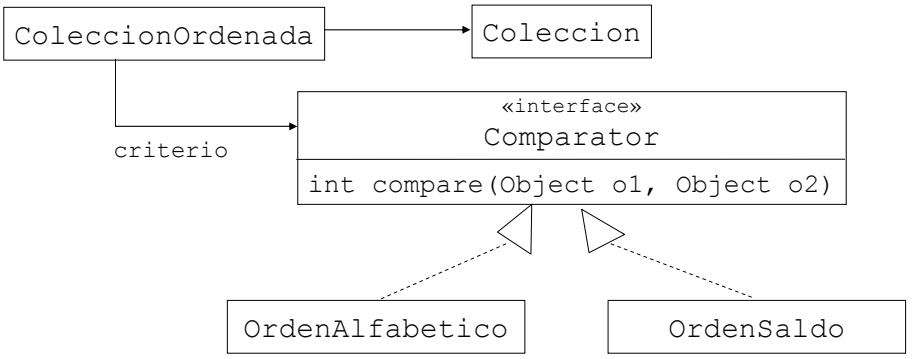
```
//Una clase que implemente una interface debe implementar
  todos sus métodos o declararse como abstract
public class ExisteNombre implements ICondicion<Contacto>{
  private String nombre;

  public ExisteNombre(String n){
      nombre = n;
  }
  public boolean test (Contacto objeto){
      return nombre.equals(objeto.getNombre());
  }
}
```

2º) Invocar el método de búsqueda pasándole como parámetro la condición

```
boolean resp = agenda.buscar(new ExisteNombre("Yago"));
```

Ejemplo composición. ColeccionOrdenada




```
ColeccionOrdenada<Cuenta> cuentas;
cuentas = new ColeccionOrdenada<Cuenta> (new OrdenAlfabetico());
```

- Ejercicio: implementar **add** en **ColeccionOrdenada**.

Ejemplo herencia. Iteradores internos

"Iterar" significa ejecutar un cierto procedimiento (**accion**) sobre todos los elementos de una estructura de datos (**coleccion**) o sobre aquellos que cumplan una condición (**test**).

```
List<Cuenta> listaCuentas;

for (Cuenta cta : listaCuentas)
   if (cta.estaNumerosRojos())
       cta.informarTitular();
```

El iterador interno permite capturar un esquema de recorrido que recibe como parámetros la acción y condición.

Iteradores internos

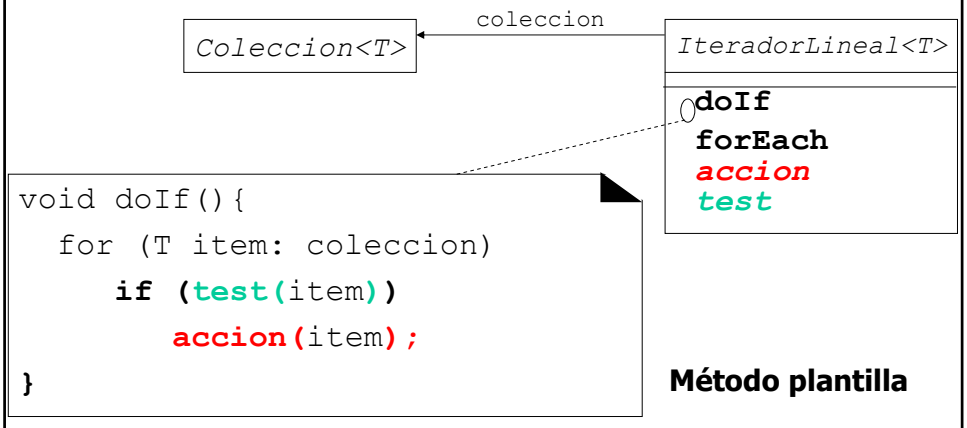
- Interesa capturar "**patrones o esquemas de recorrido** de estructuras de datos": reutilizar en vez de escribir de nuevo.
- Un sistema que haga uso de un mecanismo general para iterar debe ser capaz de aplicarlo para cualquier **`accion`** y **`test`** de su elección.
- El método de iteración debe estar parametrizado por la acción y la condición.
- En Java no es posible pasar una rutina como argumento.

Implementación de Iteradores Internos

A. Definir los métodos de iteración en la clase `Coleccion` → **NO**
   - Una iteración es una propiedad del cliente, no de la colección
   - Deberíamos crear descendientes de las clases que representan colecciones para crear diferentes esquemas de iteración.

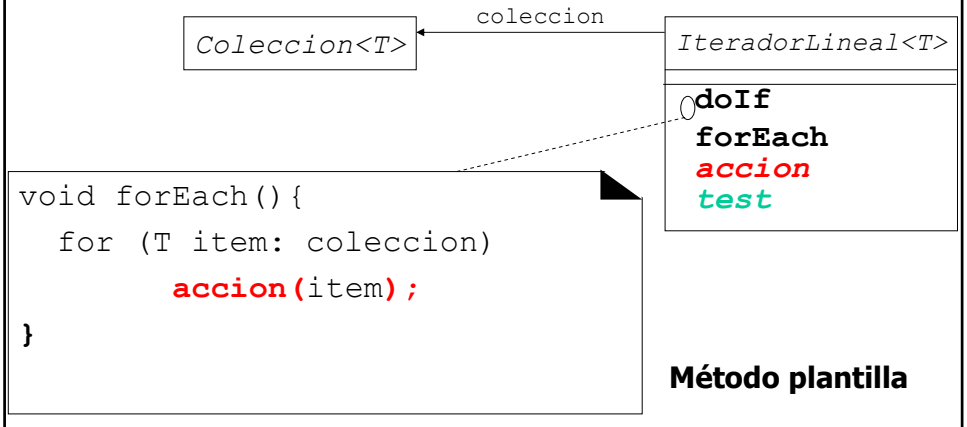
B. Implementar la ***clase comportamiento*** `Iterador` →**SI**
   - Representa objetos con capacidad para iterar sobre colecciones.

Iteradores internos



Método plantilla

Iteradores internos



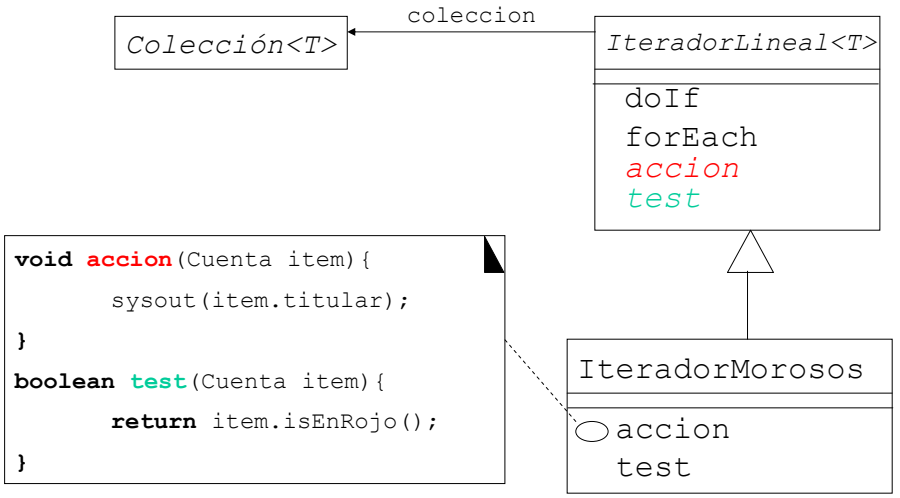
Método plantilla

Iteradores internos

```
abstract class IteradorLineal <T> {
   private List<T> coleccion;
   public IteradorLineal(List<T> c){
        coleccion = c;
   }
   public abstract void accion (T item);
   public abstract boolean test (T item);

   /* Ejecuta una acción sobre todos los elementos de
      la colección */
   public void forEach(){
        for (T item : coleccion)
              action (item);
   }
   // Ejecuta la acción sobre los items que cumplen test
   public void doIf(){
        for (T item : coleccion)
              if test(item) accion(item);
   }
}
```

Iteradores internos



Iteradores internos

- `class IteradorMorosos extends IteradorLineal<Cuenta> { }`
- Suponiendo que `LinkedList<Cuenta> cuentas;` contiene todas las cuentas del bando, si queremos imprimir por pantalla los titulares de las cuentas que están en números rojos haríamos:

```
public void imprimeMorosos(){
 IteradorMorosos iterador = new IteradorMorosos (cuentas);
 iter.doIf();
}
```


Ejemplos de Iteradores

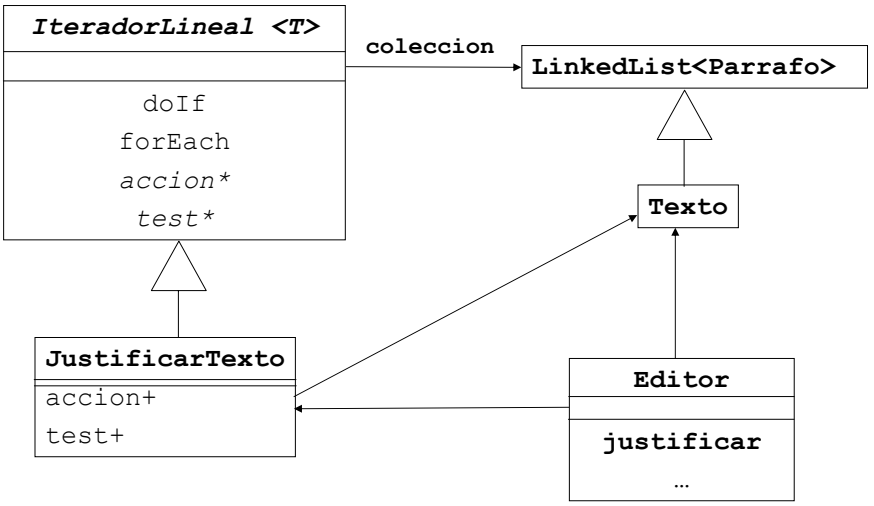
```
class IteradorSuma extends IteradorLineal <Integer>{
    private int suma;

   public IteradorSuma(List lista){
       super(lista);
   }
   public void accion (Integer v){
              suma = suma + v;
   }
   public boolean test(Integer v){ return true;}
}
```

```
public class Aplicacion{
   private List<Integer> listaInt;

   public int sumarTodo() {
        IteradorSuma iterador;
        iterador = new IteradorSuma(listaInt);
        iterador.forEach();
        return iterador.getSuma();
   }
}
```

Ejemplo Iterador: Editor de texto (1/3)



Ejemplo Iterador: Editor de texto (2/3)

```
class JustificarTexto extends IteradorLineal<Parrafo>{

   public JustificarTexto(Texto t){
       super(t);
   }

   public boolean test (Parrafo p){
       return p.sinFormato();
   }

   public void action (Parrafo p){
        p.justificar();
   }
}
```

Ejemplo Iterador: Editor de texto (3/3)

```
public class Editor {
  private Texto texto;
  …
  public void justificar(){
     JustificarTexto jt;
     jt = new JustificarTexto(texto);
     jt.doIf();
  }
}
```


¿Quién controla la iteración?

- Iterador externo:
  - el cliente es el que controla la iteración.
  - el cliente es el que avanza en el recorrido y pide al iterador el siguiente elemento.
  - Ejemplo: `Iterator` de Java.
- Iterador interno:
  - el iterador es quien controla la iteración.
  - Es el iterador el que aplica una operación a cada elemento de la colección.
  - Ejemplo: clase `IteratorLineal`

Ejemplo: imprimir los objetos de una colección

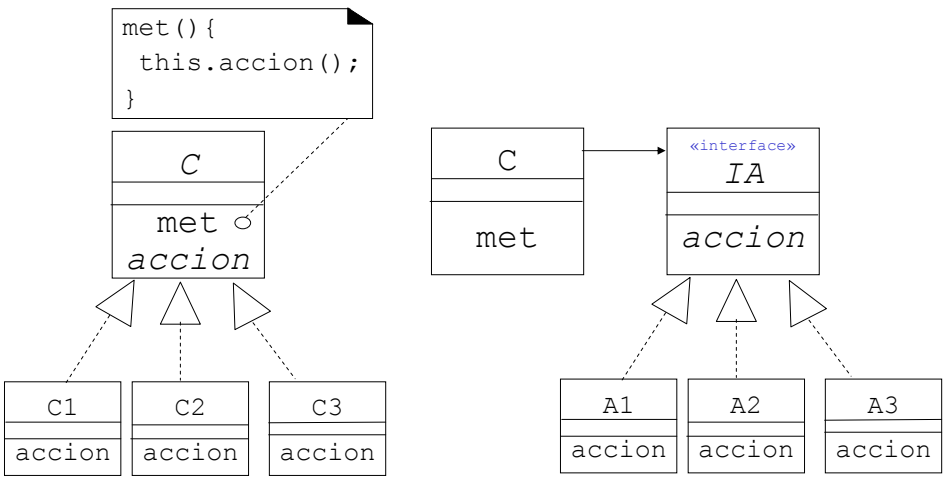
```
public void imprimir(){
  Iterator it = coleccion.iterator();
   while (it.hasNext())
     IO.imprimir(it.next());
}
```

Iterador externo

```
public void imprimir(){
  IteratorImprimir it =
          new IteratorImprimir(coleccion);
  it.forEach();
}
```

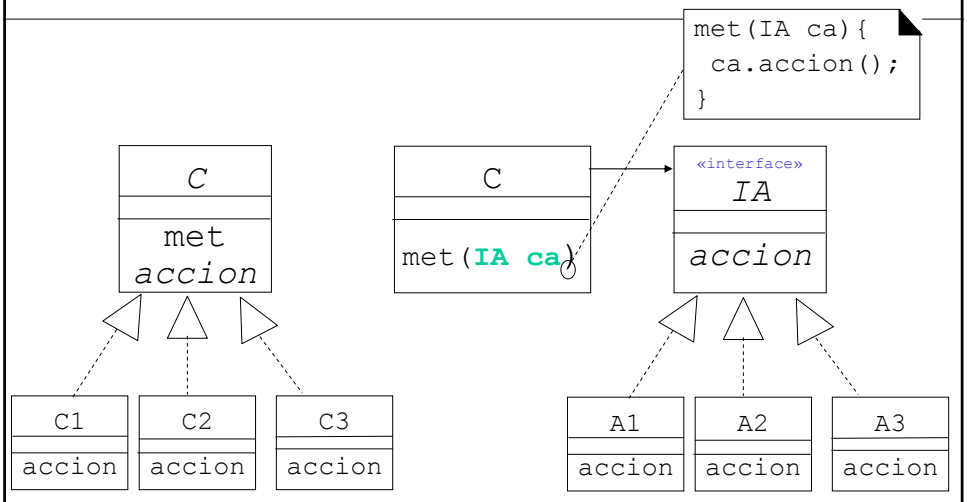
Iterador interno

Esquemas para parametrizar una rutina con una acción
→ `met(accion)`



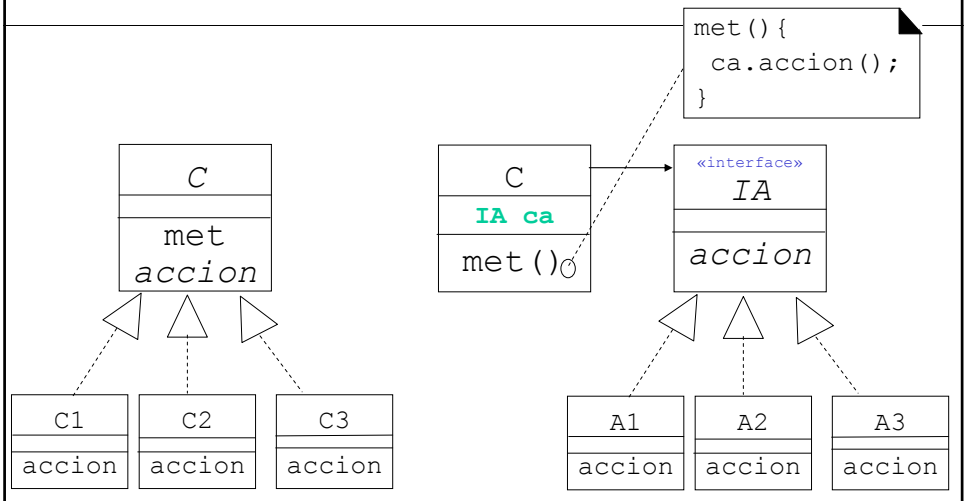
a) Método plantilla **b) Composición**

Esquemas para parametrizar una rutina con una acción
→ `met(accion)`



a) Método plantilla **b) Composición**

Esquemas para parametrizar una rutina con una acción
→ `met(accion)`



a) Método plantilla **b) Composición**

Consejos de diseño de herencia

- Hay que poner las operaciones y campos comunes en la superclase
- No se deben utilizar campos protegidos
  - Pueden ser útiles para indicar que no deben utilizarse por el público en general y deberían ser redefinidos en las subclases. Por ejemplo, `clone`.
- Hay que utilizar la herencia para modelar la relación "es_un"
- No se debe utilizar la herencia salvo que todos los métodos heredados tengan sentido
- No hay que modificar la semántica de un método en la redefinición
- Hay que utilizar el polimorfismo y no la información relativa al tipo