

# Herencia – Polimorfismo - Interfaces

# Herencia

# Relaciones entre clases

La programación orientada a objetos propone abordar el diseño de una aplicación a partir de la definición de una estructura de clases relacionadas entre sí.

La **dependencia** modela un vínculo del tipo **usa-un**

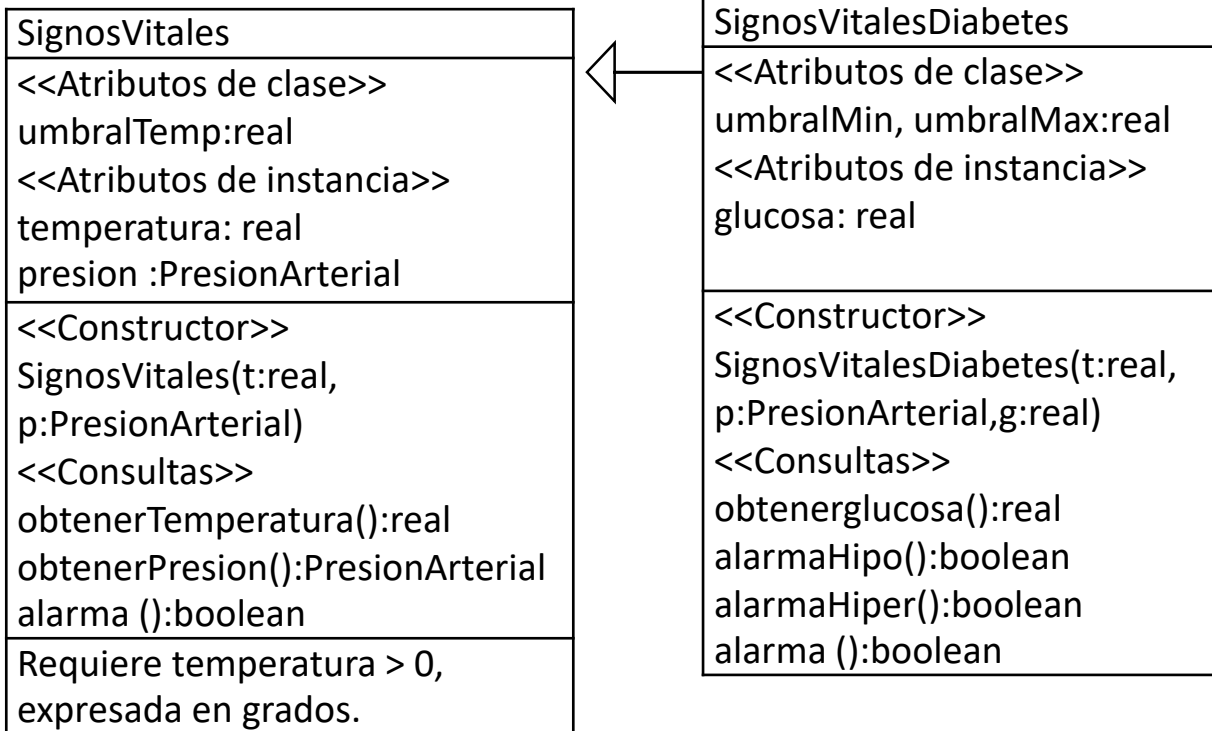
La **asociación** modela un vínculo del tipo **tiene-un**

La **herencia** modela un vínculo del tipo **es-un**

# Herencia

- ▶ El proceso de clasificación realizado en un diseño orientado a objetos se organiza en niveles.
- ▶ En el primer nivel los objetos se agrupan en **clases** de acuerdo a sus atributos y comportamientos.
- ▶ En el segundo nivel del proceso de clasificación las clases se estructuran a través de un mecanismo de especialización-generalización llamado **herencia**.
- ▶ La herencia favorece la **reusabilidad** y la **extensibilidad** del software.

## Caso de Estudio: Signos Vitales Diabetes



## Caso de Estudio: Signos Vitales Diabetes

- ▶ La clase SignosVitalesDiabetes se vincula con la clase SignosVitales a través de un relación de **herencia**.
- ▶ La clase SignosVitalesDiabetes **especializa** a la clase SignosVitales.
- ▶ La clase SignosVitalesDiabetes es una **subclase** o **clase derivada** de la **clase base** SignosVitales.
- ▶ En Java la clase más general es Object.
- ▶ Todas las clases que implementamos heredan implícitamente de Object.

## Caso de Estudio: Signos Vitales Diabetes

```
class SignosVitales{  
    //Atributos de clase  
    protected static final int umbralTemp=38;  
    //Atributos de instancia  
    protected float temperatura;  
    protected PresionArterial presion ;  
}
```

El acceso a los atributos y servicios de una clase desde sus clases derivadas depende del nivel de **encapsulamiento**.

## Caso de Estudio: Signos Vitales Diabetes

```
class SignosVitalesDiabetes
    extend SignosVitales {
//Atributos de clase
protected static final umbralMin=80;
protected static final umbralMax=120;
//Atributos de instancia
protected real glucosa;
...
}
```

La palabra reservada **extend** indica que la clase **SignosVitalesDiabetes** **hereda** de la clase **SignosVitales**.



## Caso de Estudio: Signos Vitales Diabetes

```
public SignosVitalesDiabetes (float t,  
                               PresionArterial p,  
                               float g ){  
  
    //Requiere g > 0 t > 0 y p ligada  
    super(t,p);  
    glucosa = g;  
}
```

La clase **SignosVitalesDiabetes** puede acceder al constructor de la clase **SignosVitales** usando el comando **super()** .

Si un constructor usa el constructor de la clase base, la instrucción **super()** debe ser la primera del bloque.

## Caso de Estudio: Signos Vitales Diabetes

```
//Comandos
public void establecerGlucosa(float g) {
    glucosa = g;
}
//Consultas
public float obtenerGlucosa () {
    return glucosa;
}
```

La clase **SignosVitalesDiabetes** agrega comandos y consultas.

## Caso de Estudio: Signos Vitales Diabetes

```
class historiaClinica{  
    SignosVitalesDiabetes s;  
    s = new SignosVitalesDiabetes (38.5,new  
    PresionArterial(70,130),110);  
    PresionArterial p = s.obtenerPresion();  
    float g = s.obtenerGlucosa();  
}
```

Un objeto de clase **SignosVitalesDiabetes** puede recibir mensajes considerando los métodos que brindan la clase **SignosVitalesDiabetes** y **SignosVitales**.

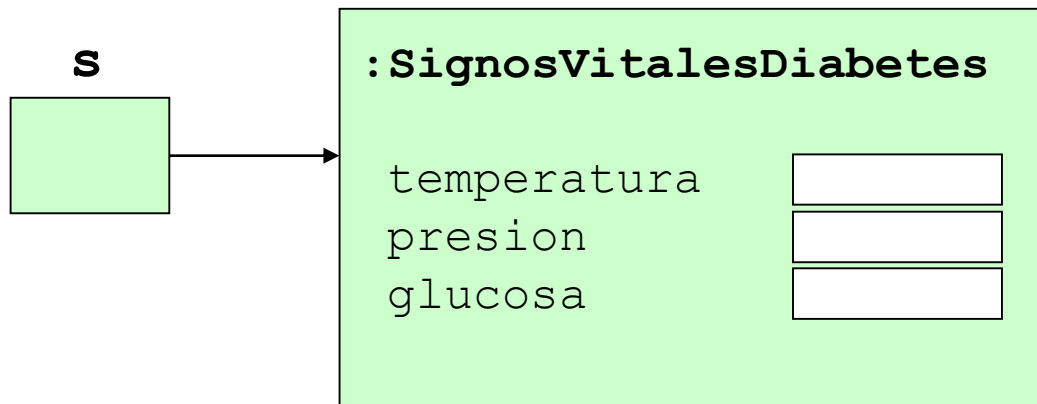
## Caso de Estudio: Signos Vitales Diabetes

```
class historiaClinica{  
    SignosVitalesDiabetes s;  
    PresionArterial p1,p2;  
    p1 = new PresionArterial(70,130),  
    s = new SignosVitalesDiabetes (38.5,p1, 110);  
    p2 = s.obtenerPresion();  
    float g = s.obtenerGlucosa();  
}
```

Un objeto de clase **SignosVitalesDiabetes** puede recibir mensajes considerando los métodos que brindan la clase **SignosVitalesDiabetes** y **SignosVitales**.

## Caso de Estudio: Signos Vitales Diabetes

- La clase derivada **hereda** de la clase base todos los atributos y métodos.



El estado interno de un objeto de clase **SignosVitalesDiabetes** tiene todos los atributos de **SignosVitales** más los específicos de su clase.

# Herencia

Si hablamos de **abstracción** cuando en la etapa de diseño clasificamos los objetos del problema, podemos llamar **superabstracción** al proceso de organizar las clases a través de una relación de **herencia**.

Los lenguajes soportan el mecanismo de herencia de manera diferente, algunos de manera más compleja y flexible, otros brindan alternativas más simples pero menos poderosas.

La **herencia simple** permite modelar una organización **jerárquica** de clases.

## Sobrecarga y Redefinición de métodos

- ▶ Cuando en una clase se define un método con el mismo nombre que otro método dentro de la misma clase o de alguna clase ancestro, pero con diferente número o tipo de parámetros, el método queda **sobrecargado**.
- ▶ Cuando en una clase derivada se define un método con el mismo nombre que en alguna clase ancestro y con el mismo número y tipo de parámetros, el método queda **redefinido**.

# Ejemplo

```
public abstract class Figura {  
    private String color;  
  
    public Figura(String color){//Constructor  
        this.color=color;  
    }  
  
    abstract double area();//Método abstracto  
    abstract double perimetro();//Método abstracto  
  
    public String getColor() {//Método no abstracto  
        return color;  
    }  
}
```

```
public class Circulo extends Figura {  
    private double r;  
    public Circulo(double r,String color) {  
        super(color);  
        this.r=r;  
        // TODO Auto-generated constructor stub  
    }  
  
    @Override  
    double area() {  
        // TODO Auto-generated method stub  
        return Math.PI*r*r;  
    }  
  
    @Override  
    double perimetro() {  
        // TODO Auto-generated method stub  
        return (2*Math.PI)*r;  
    }  
}
```



## Caso de Estudio: Expendedora Café

*Una fábrica produce dos tipos diferentes de máquinas expendedoras de café, M111 y R101.*

*Las máquinas del tipo M111 preparan **café y café con leche**. Tienen depósitos para los siguientes ingredientes secos: **café y leche**.*

*Las máquinas de tipo R101 preparan **café** , y **café carioca** . Tienen depósitos para **café, crema y cacao**.*

*Los dos modelos tienen un depósito de agua.*

# Caso de Estudio: Expendedora Café

- ▶ *Los depósitos tienen las siguientes capacidades máximas:*

<b>Agua</b>	<b>1500 mililitros</b>
<b>Café</b>	<b>1500 gramos</b>
<b>Leche</b>	<b>600 gramos</b>
<b>Cacao</b>	<b>300 gramos</b>
<b>Crema</b>	<b>600 gramos</b>

- ▶ *Además de la capacidad máxima de cada ingrediente, cada máquina mantiene registro de la cantidad disponible.*

# Caso de Estudio: Expendedora Café

*Cuando se habilita una máquina las cantidades disponibles comienzan con el valor máximo de cada ingrediente.*

*La cantidad disponible aumenta cuando se carga el depósito con un ingrediente específico y disminuye cada vez que se prepara una café.*

*Cuando se recarga se completa el depósito hasta su máxima capacidad.*

*Cada infusión consume 200 mililitros de agua.*

# Caso de Estudio: Expendedora Café

*Cada vez que se solicita una infusión se reducen los ingredientes de acuerdo a la siguiente tabla:*

	Café	Café con leche	Café Carioca
Café	40	40	30
Leche		20	
Cacao			10
Crema			30

# Caso de Estudio: Expendedora Café

El diseñador de un modelo para las máquinas expendedoras podría especificar dos clases de acuerdo al siguiente diagrama:

M111	R101
<pre>&lt;&lt;atributos de clase&gt;&gt; maxCafe : entero maxAgua:entero maxLeche:entero &lt;&lt;atributos de instancia&gt;&gt; cantCafé : entero cantAgua:entero cantLeche : entero</pre>	<pre>&lt;&lt;atributos de clase&gt;&gt; maxCafe : entero... maxAgua:entero maxCacao:entero maxCrema:entero &lt;&lt;atributos de instancia&gt;&gt; cantCafé : entero cantAgua:entero cantCacao : entero cantCrema : entero</pre>

# Caso de Estudio: Expendedora Café

M111

<<Constructor>>

M111()

<<Comandos>>

cafe()

cafeConLeche()

recargarCafe()

recargarAgua()

recargarLeche()

<<consultas>>

vasosCafe():entero

valosCafeConLeche(): entero

R101

<<Constructor>>

R101()

<<Comandos>>

cafe()

cafeCarioca()

recargarCafe()

recargarAgua()

recargarCrema()

recargarCacao()

<<consultas>>

vasosCafe():entero

vasosCafeCarioca(): entero

# Caso de Estudio: Expendedora Café

Alternativamente el diagrama podría incluir a una única clase que modele los dos tipos de máquinas:

ExpendedoraCafe

<<atributos de clase>>

maxCafe : entero

maxAgua:entero

maxLeche:entero

maxCacao:entero

maxCrema:entero

<<atributos de instancia>>

cantCafé : entero

cantAgua:entero

cantLeche : entero

cantCacao : entero

cantCrema : entero

# Caso de Estudio: Expendedora Café

ExpendedoraCafe	
<b>&lt;&lt;Constructor&gt;&gt;</b> <b>ExpendedoraCafe()</b> <b>&lt;&lt;Comandos&gt;&gt;</b> <b>cafe()</b> <b>cafeConLeche()</b> cafeCarioca() <b>recargarCafe()</b> <b>recargarAgua()</b> recargarLeche() recargarCrema() recargarCacao() <b>&lt;&lt;consultas&gt;&gt;</b> <b>vasosCafe():entero</b> vasosCafeConLeche():entero vasosCafeCarioca(): entero	<b>ExpendedoraCafe()</b> Las cantidades disponibles se inicializan con los máximos
	<b>cafe()</b> Requiere disponible 40 grs. de café y 200 ml de agua.
	<b>cafeConLeche()</b> Requiere disponible 40 grs de café, 200 ml de agua y 20 grs. de leche.
	<b>recargarCafe()</b> Se carga el depósito completo
	<b>vasosCafe():entero</b> Calcula la cantidad máxima de vasos que pueden prepararse con las cantidades en depósito

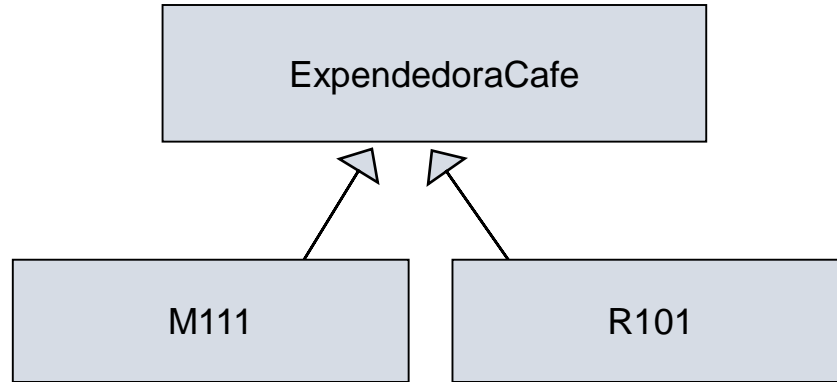


# Caso de Estudio: Expendedora Café

Un modelo más adecuado consistiría en factorizar los atributos y comportamiento compartidos de M111 y R101 en una clase general y retener los atributos y comportamientos específicos en clases especializadas.

Este proceso se conoce como **generalización** porque parte de dos clases específicas para obtener una más general.

# Caso de Estudio: Expendedora Café



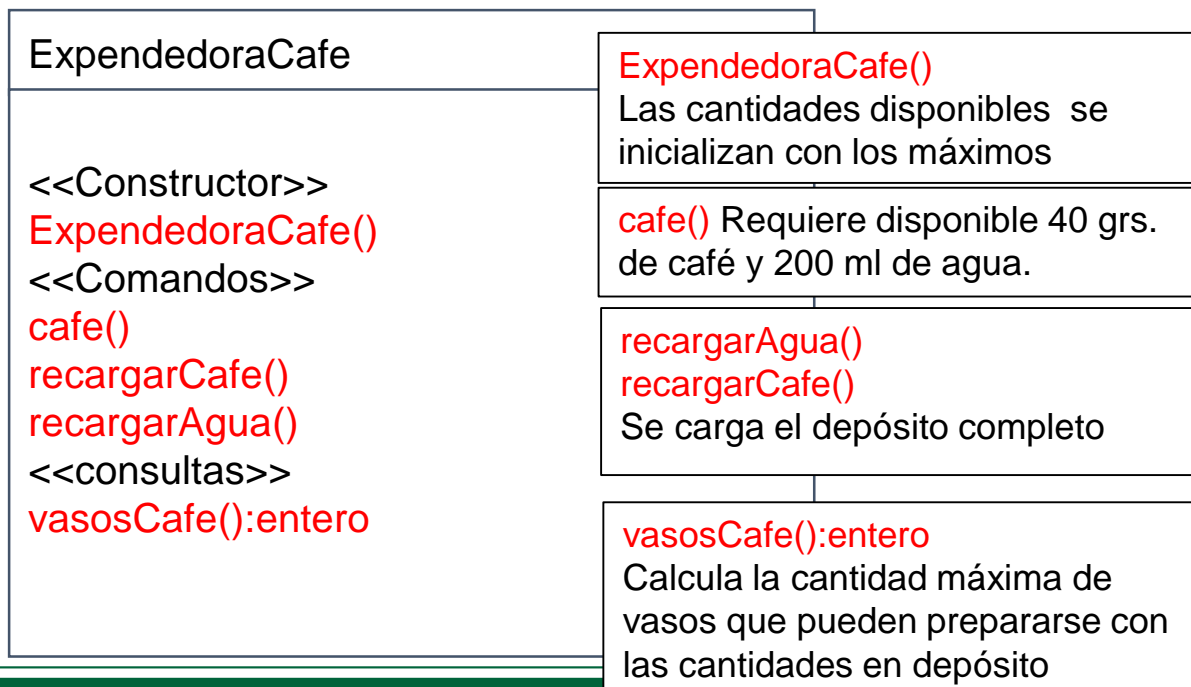
# Caso de Estudio: Expendedora Café

La clase más general incluye los **atributos** compartidos por todas las instancias:

ExpendedoraCafe
<<atributos de clase>> maxCafe : entero maxAgua:entero <<atributos de instancia>> cantCafé : entero cantAgua:entero

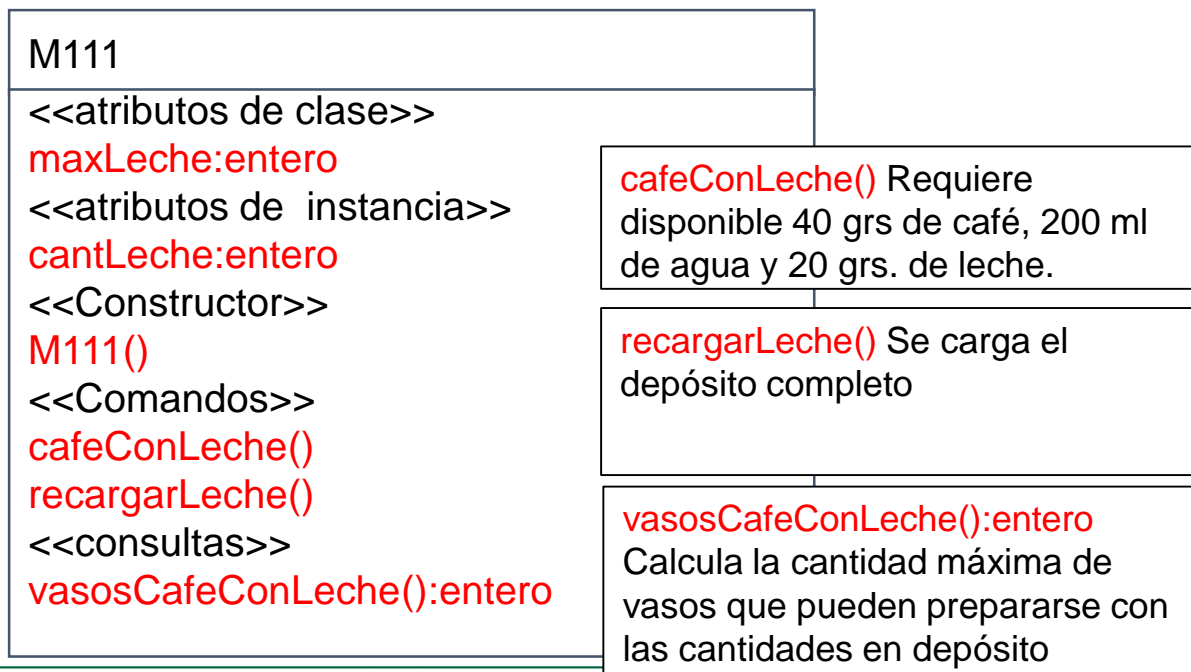
## Caso de Estudio: Expendedora Café

La clase más general incluye los **servicios** compartidos por todas las instancias:



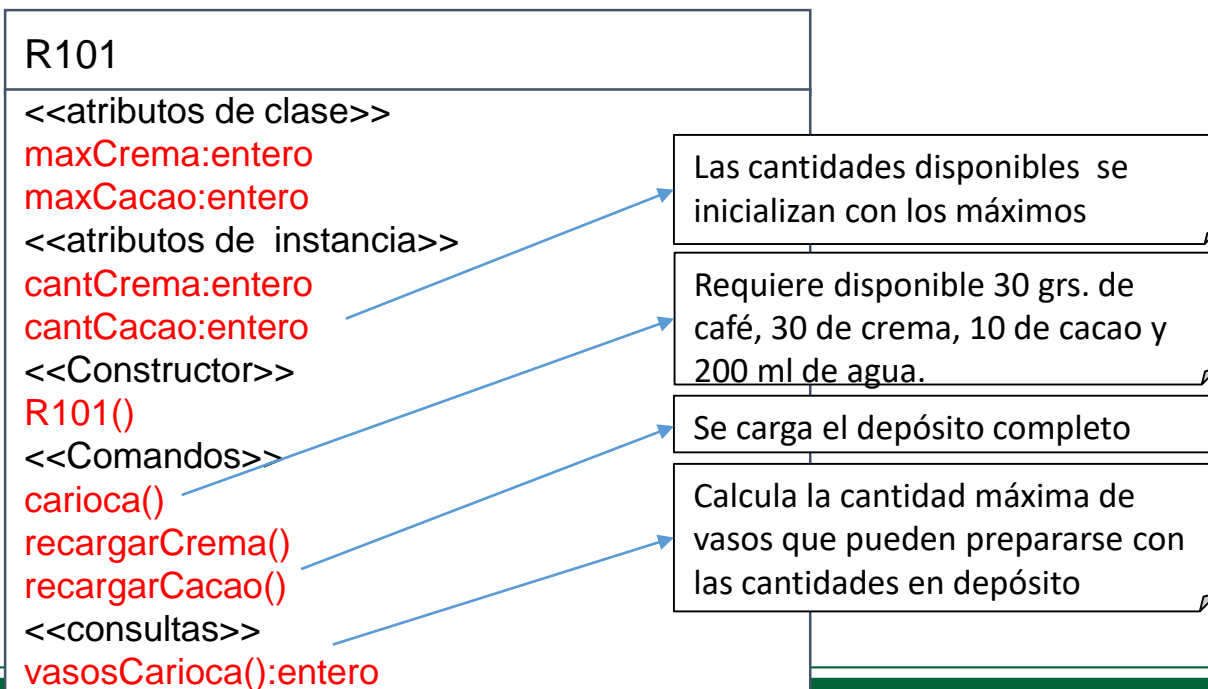
# Caso de Estudio: Expendedora Café

Las clases especializadas incluyen atributos y servicios específicos.



# Caso de Estudio: Expendedora Café

Las clases especializadas incluyen atributos y servicios específicos.



# Caso de Estudio: Expendedora Café

La clase M111 está vinculada a la clase ExpendedoraCafe por una relación de **herencia**.

Todo objeto de clase M111 es también un objeto de clase ExpendedoraCafe.

Un objeto de clase M111 estará caracterizado por todos los atributos y el comportamiento propio de la clase, pero además por todos los atributos y el comportamiento de la clase ExpendedoraCafe.

# Caso de Estudio: Expendedora Café

La clase R101 está vinculada a la clase ExpendedoraCafe por una relación de **herencia**.

Todo objeto de clase R101 es también un objeto de clase ExpendedoraCafe.

Un objeto de clase R101 estará caracterizado por todos los atributos y el comportamiento propio de la clase, pero además por todos los atributos y el comportamiento de la clase ExpendedoraCafe.



# Caso de Estudio: Expendedora Café

Como los atributos están **protegidos**, las clases que **heredan** a **ExpendedoraCafe** tiene acceso a ellos.

```
class ExpendedoraCafe {  
    //atributos de clase  
    //medido en gramos  
    protected static final int maxCafe = 1500;  
    //medido en mililitros  
    protected static final int maxAgua = 1500;  
    //atributos de instancia  
    protected int cantCafe;  
    protected int cantAgua;  
    //Constructor de ExpendedoraCafe  
    public ExpendedoraCafe() {  
        //Cada depósito se carga completo  
        cantCafe = maxCafe;  
        cantAgua = maxAgua;  
    }  
}
```

# Caso de Estudio: Expendedora Café

```
//Comandos de ExpendedoraCafe
public void cafe() {
    /*Requiere disponibles 40 gramos de café y 200 ml de
    agua*/
    cantCafe = cantCafe - 40;
    cantAgua = cantAgua - 200;
}
public void recargarCafe() {
    //Carga el depósito completo
    cantCafe = maxCafe;
}
public void recargarAgua() {
    //Carga el depósito completo
    cantAgua = maxAgua;
}
```

# Caso de Estudio: Expendedora Café

```
//Consultas de ExpendedoraCafe
public int vasosCafe() {
  /*Computa cuántos vasos de café pueden prepararse con
  las cantidades disponibles*/
  int c = (int) cantCafe / 40;
  int a = (int) cantAgua / 200;
  if (c < a) return c;
  else return a;
}
```

# Caso de Estudio: Expendedora Café

En Java la palabra **extend** especifica que la clase **M111** hereda de la clase **ExpendedoraCafe**.

```
class M111 extends ExpendedoraCafe {  
    //atributos de clase  
    //gramos  
    protected static final int maxLeche = 600;  
    //atributos de instancia  
    protected int cantLeche;
```

# Caso de Estudio: Expendedora Café

Una clase derivada hereda de la clase base todos sus atributos y métodos, pero **no los constructores**.

Cada constructor de la clase derivada puede usar invocar a un constructor de la clase base usando **palabra clave super**.

Si se invoca un constructor de la clase base **siempre tiene que ser en la primera línea** del bloque de código.

```
//Constructor de M111
public M111() {
//Cada depósito se carga completo
    super() ;
    cantLeche = maxLeche;
}
```

# Caso de Estudio: Expendedora Café

El estado interno de cada objeto de clase **M111** incluye los atributos definidos en esa clase más los atributos heredados de **ExpendedoraCafe**.

:M111	
cantCafe	<input type="text"/>
cantAgua	<input type="text"/>
cantLeche	<input type="text"/>

# Caso de Estudio: Expendedora Café

```
//Comandos de M111
public void cafeConLeche() {
    /*Requiere disponibles 40 gramos de café, 200 ml de
    agua y 20 grs de leche*/
    cafe();
    cantLeche = cantLeche -20;
}
public void recargarLeche() {
    //Carga el depósito completo
    cantLeche = maxLeche;
}
```

# Caso de Estudio: Expendedora Café

```
//Consultas de M111
public int vasosCafeConLeche() {
    /*Computa cuántos vasos de café con leche pueden
    prepararse con las cantidades disponibles*/
    int c = vasosCafe();
    int l = (int) cantLeche / 20;
    if (c < l) return c;
    else return l;
}
```



# Caso de Estudio: Expendedora Café

```
class R101 extends ExpendedoraCafe {  
    //atributos de clase  
    //gramos  
    protected static final int maxCrema = 600;  
    protected static final int maxCacao = 300;  
    //atributos de instancia  
    protected int cantCrema;  
    protected int cantCacao;  
    //Constructor de R101  
    public R101() {  
        //Cada depósito se carga completo  
        super();  
        cantCacao = maxCacao;  
        cantCrema = maxCrema;  
    }  
}
```

# Caso de Estudio: Expendedora Café

El atributo **cantCafe** es accesible porque está declarado como protegido.

```
//Comandos de R101
public void carioca() {
    /*Requiere disponibles 30 gramos de café, 200 ml
    de agua, 30 de crema y 10 grs de cacao*/
    cantCafe = cantCafe -30;
    cantAgua = cantAgua -200;
    cantCrema = cantCrema -30;
    cantCacao = cantCacao -10; }
public void recargarCrema() {
    //Carga el depósito completo
    cantCrema = maxCrema; }
public void recargarCacao() {
    //Carga el depósito completo
    cantCacao = maxCacao; }
```

# Caso de Estudio: Expendedora Café

```
R101 m1 = new R101 ();  
M111 m2 = new M111 ();  
m1.cafe();  
m2.cafe();
```

En el problema, Las máquinas **R101** y **M111** ofrecen café.


En la solución, las clases **R101** y **M111** heredan los atributos de instancia y los servicios de la clase **ExpendedoraCafe**, de modo que un objeto de cualquiera de esas clases puede recibir el mensaje **cafe()** .

## Caso de Estudio: Expendedora Café

```
R101 m1 = new R101 ();
```

```
M111 m2 = new M111 ();
```

```
m1.cafeConLeche();
```

 Error de compilación

```
m2.cafeConLeche();
```

En el problema, solo las máquinas del modelo **M111** ofrecen café con leche.

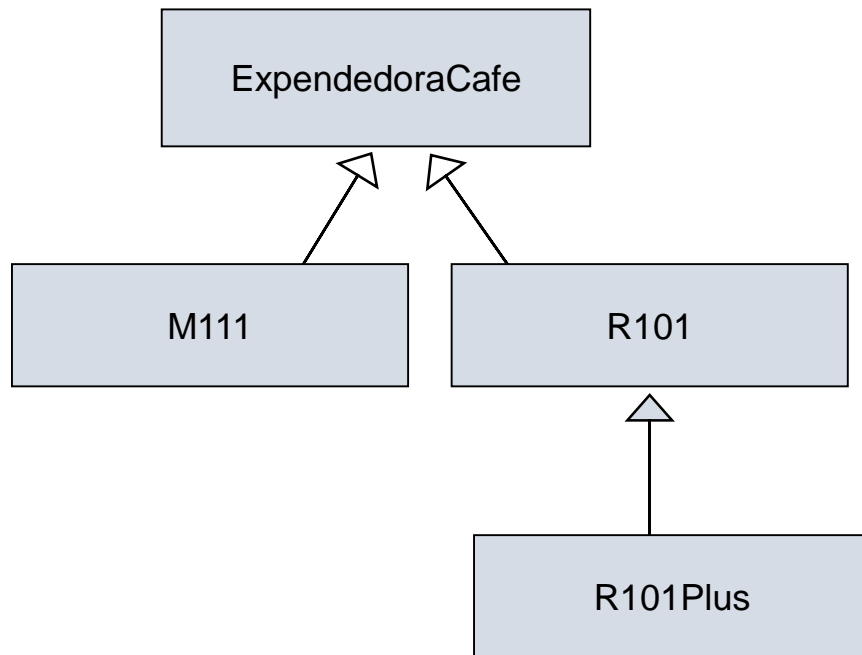
En la solución, solo los objetos de clase **M111** pueden recibir el mensaje **cafeConLeche**.

## Caso de Estudio: Expendedora Café

*La fábrica incorpora ahora un nuevo modelo R101 Plus que tiene la funcionalidad de R101 pero prepara un café más fuerte usando 50 grs. y agrega como ingrediente a la canela con capacidad máxima de 300 grs. y prepara café bahiano.*

*El café bahiano requiere la preparación de un café carioca al cual se le agregan 10 gramos de canela.*

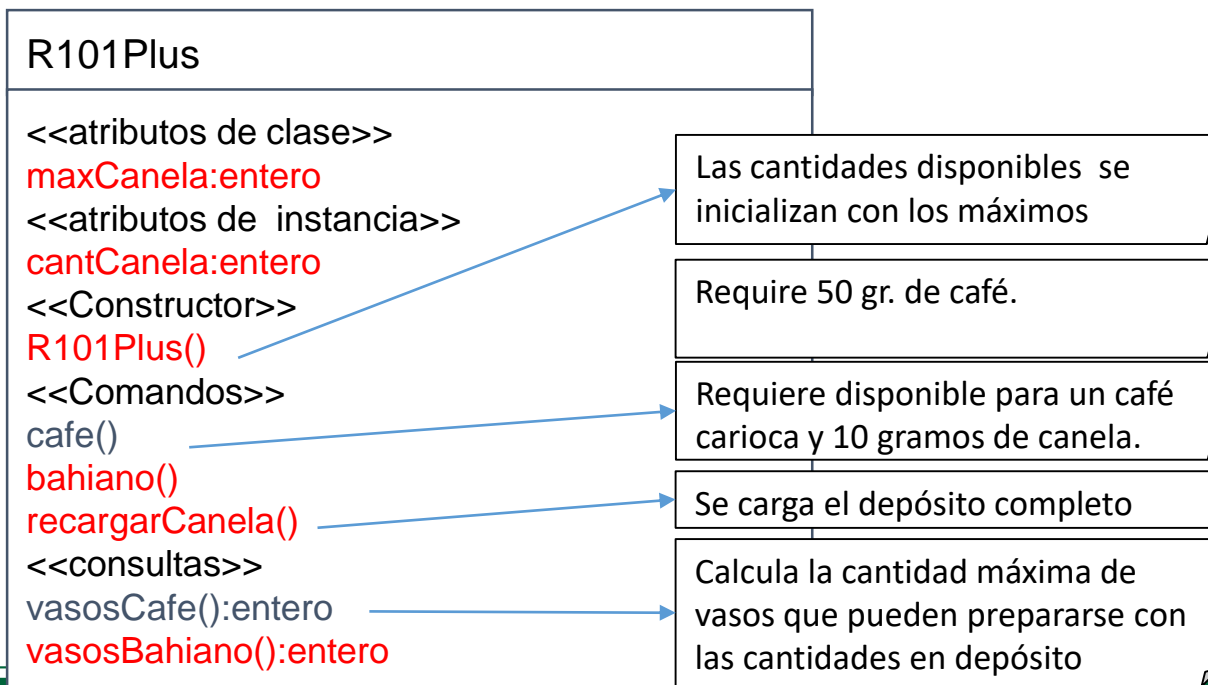
## Caso de Estudio: Expendedora Café



La clase **R101Plus** especializa a la clase **R101**.

# Caso de Estudio: Expendedora Café

Las clases especializadas incluyen atributos y servicios específicos.



# Caso de Estudio: Expendedora Café

La clase **R101Plus** está vinculada a la clase **R101** por una relación de **herencia**.

Todo objeto de la clase **R101Plus** es también un objeto de la clase **R101** y además un objeto de la clase **ExpendedoraCafe**.

La clase **R101** es una clase derivada de la clase **ExpendedoraCafe**, pero a su vez es una clase base para la clase **R101Plus**.

**NOTEMOS QUE** la modificación en la especificación del problema no requiere modificar las clases ya implementadas y verificadas, sino agregar una nueva clase.



# Caso de Estudio: Expendedora Café

El constructor de **R101Plus** puede acceder al constructor de la clase **R101** pero no al constructor de **ExpendedoraCafe**.

```
class R101Plus extends R101 {  
    //Atributo de clase  
    protected static final int maxCanela = 300;  
    //Atributo de instancia  
    protected int cantCanela;  
    //Constructor  
    public R101Plus () {  
        super();  
        cantCanela = maxCanela;  
    }  
}
```

# Caso de Estudio: Expendedora Café

```
//Comandos en la clase R101Plus
public void cafe() {
    /*Requiere 50 gramos de café en el depósito*/
    cantCafe = cantCafe -50;
    cantAgua= cantAgua -200;
}
public void bahiano() {
    /*Requiere disponible para un carioca y 10 gramos de
    canela*/
    carioca ();
    cantCanela = cantCanela-10;
}
public void recargarCanela() {
    //Carga el depósito completo
    cantCanela = maxCanela; }
```

# Caso de Estudio: Expendedora Café

La consulta **vasosCafe()** de la clase **R101Plus** redefine a la consulta **vasosCafe()** definido en **ExpendedoraCafe**.

```
//Consultas en la clase R101Plus
public int vasosCafe(){
    /*Computa cuántos vasos de café pueden prepararse con
    las cantidades disponibles*/
    int c = (int) cantCafe / 50;
    int a = (int) cantAgua / 200;
    if (c < a) return c;
    else return a;
}
```

# Caso de Estudio: Expendedora Café

El método `cafe()` de la clase `R101Plus` redefine al método `cafe()` definido en `ExpendedoraCafe`.

```
//Comandos en la clase ExpendedoraCafe
public void cafe() {
    /*Requiere 40 gramos de café en el depósito*/
    cantCafe = cantCafe -40;
}
```

```
//Comandos en la clase R101Plus
public void cafe() {
    /*Requiere 50 gramos de café en el depósito*/
    cantCafe = cantCafe -50;
}
```

## Caso de Estudio: Expendedora Café

```
R101 m1 = new R101();  
R101Plus m2 = new R101Plus();  
  
m1.cafe(); Se liga al método definido en ExpendedoraCafe  
m2.cafe(); Se liga al método definido en R101Plus
```

El método **cafe()** de la clase **ExpendedoraCafe** queda derogado para los objetos de clase **R101Plus**.

## Caso de Estudio: Expendedora Café

```
R101 m1 = new R101();  
R101Plus m2 = new R101Plus();  
  
m1.carioca();  
m2.carioca();  
m1.bahiano(); ← Error de compilación  
m2.bahiano();
```

## Caso de Estudio: Expendedora Café

```
ExpendedoraCafe e1,e2,e3;  
e1 = new M111() ;  
e2 = new R101() ;  
e3 = new R101Plus() ;
```

Las variables **e1**, **e2** y **e3** son **polimórficas**, pueden quedar **ligadas** a objetos de la clase **ExpendedoraCafe** o de sus clases derivadas.

## Caso de Estudio: Expendedora Café

```
ExpendedoraCafe e1,e2,e3;
```

```
e1 = new M111();
```

```
e2 = new R101();
```

```
e3 = new R101Plus();
```

```
e1.cafe(); Se liga al método definido en ExpendedoraCafe
```

```
e2.cafe(); Se liga al método definido en ExpendedoraCafe
```

```
e3.cafe(); Se liga al método definido en R101Plus
```

La **clase del objeto** determina la **ligadura** entre el **mensaje** y el **método**.



## Caso de Estudio: Expendedora Café

```
ExpendedoraCafe e1,e2,e3;
```

```
e1 = new M111() ;
```

```
e2 = new R101() ;
```

```
e3 = new R101Plus() ;
```

```
e1.cafeConLeche() ;
```

```
e2.carioca() ;
```

```
e3.bahiano() ;
```

ERRORES DE  
COMPILACION

El **tipo de la variable** determina los **mensajes** que puede recibir el **objeto**.

## Caso de Estudio: Expendedora Café

```
ExpendedoraCafe e;  
if (cond)  
    e = new M111();  
else  
    e = new R101();  
e.cafe();
```

## Caso de Estudio: Expendedora Café

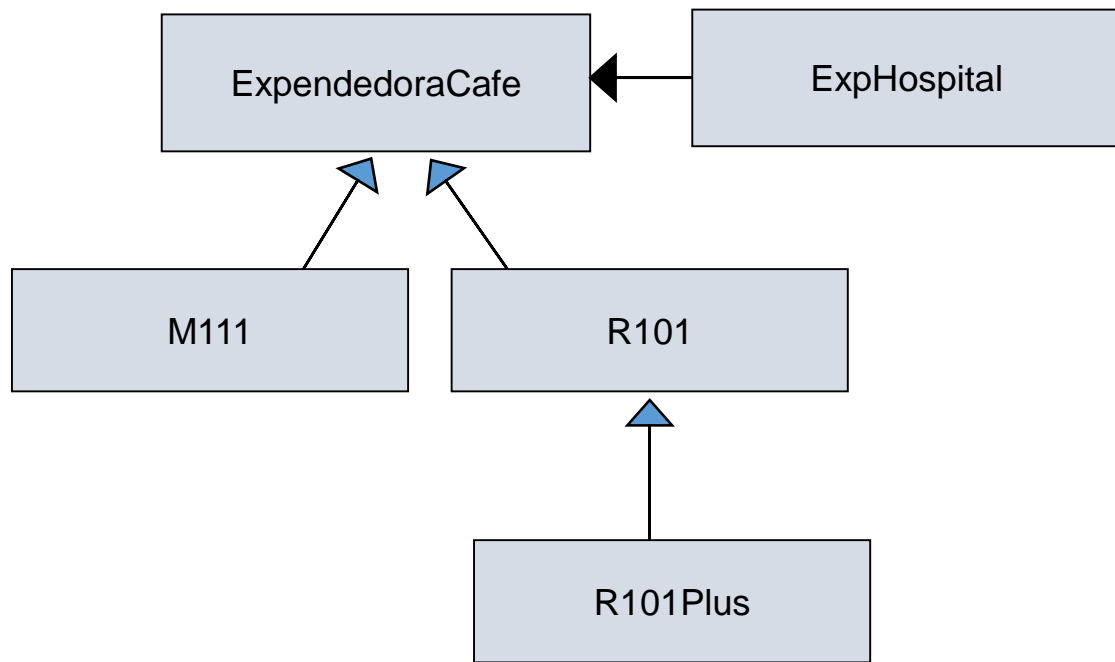
```
ExpendedoraCafe e;  
if (cond)  
    e = new M111() ;  
else  
    e = new R101() ;  
e.carioca() ;      ERROR
```

## Caso de Estudio: Expendedora Café

*Como parte de sus actividades de responsabilidad social la fábrica instala y mantiene en funcionamiento las máquinas expendedoras de café de algunos hospitales.*

*Cada máquina se asigna a un sector del hospital y el encargado realiza un relevamiento que le permite instalar nuevas máquinas, desinstalar y efectuar consultas.*

# Caso de Estudio: Expendedora Café



# Caso de Estudio: Expendedora Café

ExpHospital
T [] ExpendedoraCafe
<<constructores>> ExpHospital (max : entero) <<comandos>> instalar (r : ExpendedoraCafe, s : entero) desinstalar (s : entero)

# Caso de Estudio: Expendedora Café

ExpHospital

T [] ExpendedoraCafe

<<consultas>>

cantSectores():entero

cantSectoresOcupados(): entero

todosOcupados () : boolean

estaExpendedoraCafe (r: ExpendedoraCafe) : boolean

existeSector(s:entero):boolean

cantDisponible(vasos:entero):entero

# Caso de Estudio: Expendedora Café

```
class ExpHospital {  
    private ExpendedoraCafe[] T;  
  
    //Constructor  
    public ExpHospital(int max) {  
        /*Crea un arreglo con max elementos, cada  
        elemento representa un sector de la fábrica*/  
        T= new ExpendedoraCafe [max];  
    }  
    ...  
}
```



# Caso de Estudio: Expendedora Café

```
public void instalar(ExpendedoraCafe r,  
                    int s) {  
    /*Asigna el la máquina r al sector s.  
    Requiere 0<=s<cantSectores() */  
    T[s] = r;  
}
```

```
public void desinstalar(int s) {  
    /*Elimina la asignación de la máquina r  
    del sector s.  
    Requiere 0<=s<cantSectores()*/  
    T[s] = null;  
}
```

## Caso de Estudio: Expendedora Café

```
public int cantSectores() {  
    return T.length;  
}  
public int cantSectoresOcupados () {  
    int i = 0; int cant = 0;  
    while (i < cantSectores()) {  
        if (T[i] != null) cant++;  
        i++;  
    }  
    return cant;  
}
```

## Caso de Estudio: Expendedora Café

```
public boolean todosOcupados () {  
    /*Retorna true si hay al menos un sector  
    que no tiene una máquina instalada*/  
    int i = 0; boolean hayNulo = false;  
    while (i < cantSectores() && !hayNulo) {  
        hayNulo = T[i] == null;  
        i++;  
    }  
    return !hayNulo;  
}
```

## Caso de Estudio: Expendedora Café

```
public boolean
    estaExp(ExpendedoraCafe r){
/* Decide si algún sector tiene asignado
una máquina con la misma identidad que r
*/
    int i = 0; boolean esta = false;
    while (i < cantSectores() && !esta ){
        esta = T[i] == r ;
        i++;
    }
    return esta;
}
```

## Caso de Estudio: Expendedora Café

```
public boolean existeSector (int s){  
    return s>= 0 & s< cantSectores();  
}
```

```
public ExpendedoraCafe  
    expSector (int s){  
    /*Retorna la máquina instalada en el  
    sector s, requiere 0<=s<cantSectores()*/  
    return T[s];  
}
```

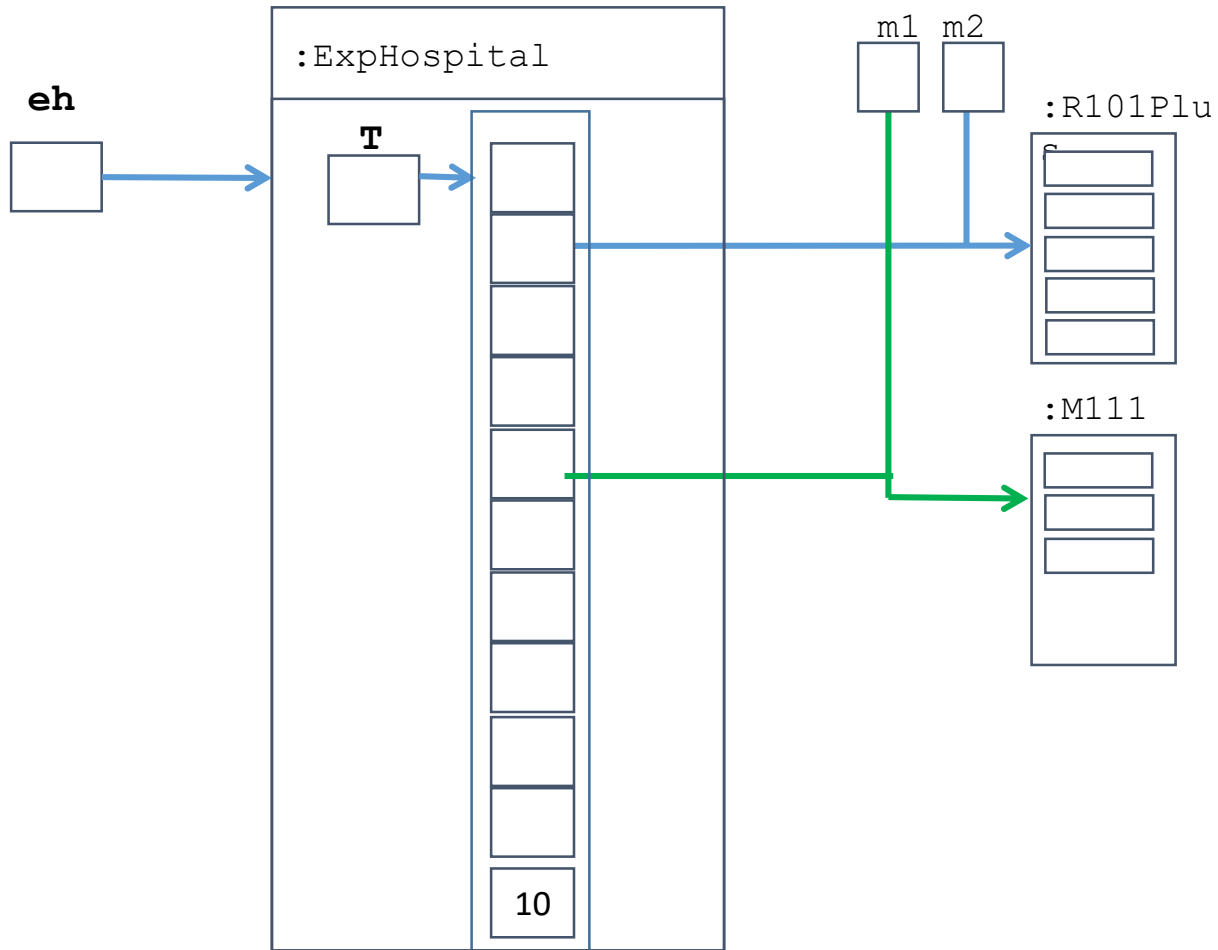
# Caso de Estudio: Expendedora Café

El mensaje  
referencia

```
public int cantDisponible(int n){  
    /*Cuenta los sectores con máquinas con  
    ingredientes para preparar al menos n  
    vasos de café*/  
    int cont =0;  
    for (int i=0;i<cantSectores();i++)  
        if (T[i] != null)  
            if(T[i].vasosCafe() >= n)  
                cont++;  
  
    return cont;  
}
```

# Caso de Estudio: Expendedora Café

```
class gestionHospital{  
  
    ...  
    ExpHospital eh = new ExpHospital(10);  
    M111 m1 = new M111();  
    R101Plus m2 = new R101Plus();  
  
    eh.instalar(m1,5);  
    eh.instalar(m2,1);  
  
}
```





# Herencia y abstracción

La **abstracción de datos** permite clasificar objetos en clases.

La **herencia jerárquica** aumenta el nivel de abstracción porque las clases son a su vez clasificadas a partir de un proceso de **generalización o especialización**.

Los conceptos clase, abstracción de datos y encapsulamiento, NO son exclusivos de la POO.

El mecanismo de herencia aplicado al desarrollo de software, surge con la POO.

# Herencia y Encapsulamiento

- ▶ El encapsulamiento permite oscurecer los detalles de la definición de una clase, mostrando sólo aquellos elementos que permiten crear y manipular objetos.
- ▶ La interfaz está constituida por todos los miembros que van a ser visibles desde otras clases.
- ▶ En Java el modificador de acceso **protected** establece que las clases derivadas tienen acceso a los miembros protegidos de sus clases ancestro

# Herencia y Encapsulamiento

- ▶ Existen diferentes criterios referidos al nivel de encapsulamiento que debería ligar a clases vinculadas por una relación de herencia.
- ▶ Un argumento a favor de que las clases derivadas accedan a todos sus atributos, es que una instancia de una clase específica es también una instancia de las clases más generales de modo que debería poder acceder y modificar su estado interno.
- ▶ El argumento en contra es que si se modifica la implementación de la clase base, el cambio afectará a todas las clases derivadas que accedan directamente a la representación.

# Redefinición y sobrecarga

- ▶ Una clase derivada puede **redefinir** un método de una de sus clase ancestro, si especifica el mismo nombre, número y tipo de parámetros.
- ▶ Decimos que el método en la clase derivada **deroga** al método de la clase base.
- ▶ Una clase derivada puede **sobrecargar** un método de una de sus clase ancestro, si especifica el mismo nombre, con distinto número o tipo de parámetros.
- ▶ ¿Qué pasa si el método se declara privado en la clase derivada?

## Caso de Estudio: Fábrica de juguetes

*En una fábrica de juguetes parte de la producción la realizan robots. Todos los robots son responsables de recargar su energía hasta el máximo cuando queda por debajo del mínimo.*

*Cada robot construye un auto consumiendo 70 unidades de energía y usando 1 chasis, 4 ópticas y 4 ruedas. La vida útil es 1000 menos la cantidad de recargas realizadas.*

*Los robots del modelo Alfa arman un auto usando 1 óptica y 1 rueda adicional (que colocan en el interior) y arman también camiones consumiendo 80 unidades de energía y usando 6 ruedas, 6 ópticas y 1 chasis de camión. La vida útil es 5000 menos la cantidad de recargas realizadas.*

*Cuando un robot recibe la orden de preparar un auto o un camión asume que se controló que dispone de piezas para hacerlo.*

## Caso de Estudio: Fábrica de juguetes

La clase cliente solo envía mensajes a un robot si su vidaUtil es mayor a 0.

## Robot

```
<<atributos de clase>>
energiaMaxima : 5000
energiaMinima : 100
<<atributos de instancia>>
nroSerie:entero
energia: entero
ruedas: entero
opticas: entero
chasisA: entero
cantRecargas:entero
<<constructor>>
Robot (ns:entero)
<<comandos>>
armarAuto()
abrirCaja(caja:Caja)
recargar()
<<consultas>>
obtenerEnergia():entero
...
vidaUtil():entero
cantAutos() : entero
```



## RobotAlfa

```
<<atributos de instancia>>
chasisC: entero
<<constructor>>
RobotAlfa (ns:entero)
<<comandos>>
armarAuto()
armaCamion()
abrirCaja(caja:CajaC)
<<consultas>>
vidaUtil():entero
cantAutos() : entero
cantCamiones():entero
```

## Caso de Estudio: Fábrica de juguetes

La clase RobotAlfa **especializa** a la clase Robot.

Las instancias de una clase RobotAlfa son también instancias de la clase Robot, de modo que **heredan** sus atributos y métodos.

RobotAlfa es una **subclase** o **clase derivada** de la **superclase** o **clase base** Robot.

Los métodos armarAuto, cantAutos y vidaUtil de la clase Robot están **redefinidos** en la clase RobotAlfa, que además agrega nueva funcionalidad.

El método abrirCaja de la clase Robot está sobrecargado en la clase RobotAlfa.



## Caso de Estudio: Fábrica de juguetes

```
class Robot {  
    //atributos de clase  
    protected static final int energiaMaxima = 5000;  
    protected static final int energiaMinima = 100;  
    //atributos de instancia  
    protected int nroSerie;  
    protected int cantRecargas;  
    protected int energia;  
    protected int ruedas;  
    protected int opticas;  
    protected int chasisA;
```

## Caso de Estudio: Fábrica de juguetes

```
//Constructores
public Robot (int nro){
    nroSerie = nro;
    energia=energiaMaxima;
    ruedas = 100;
    opticas = 100;
    chasisA = 100;
}
public void recargar(){
    energia=energiaMaxima;
    cantRecargas++;
}
```

## Caso de Estudio: Fábrica de juguetes

```
//En Robot
public void armarAuto () {
/*Requiere que se haya controlado si hay piezas
disponibles*/
    ruedas -= 4 ;
    opticas -=4;
    energia -= 70;
    chasisA --;
/*Controla si es necesario recargar energía*/
    if (energia < energiaMinima)
        this.recargar() ;
}
```

## Caso de Estudio: Fábrica de juguetes

```
//En Robot
public void abrirCaja (Caja caja) {
    /*Aumenta sus cantidades según las de la caja. Requiere que se vacíe la caja
    después de que el robot la abra*/
    ruedas += caja.obtenerRuedas();
    opticas += caja.obtenerOpticas();
    chasisA += caja.obtenerChasisA();
    energia -= 50;
    /*Controla si es necesario recargar energía*/
    if (energia < energiaMinima)
        this.recargar();
    caja.vaciar();
}
```

## Caso de Estudio: Fábrica de juguetes

```
//Consultas
public int obtenerNroSerie(){
    return nroSerie;}
public int obtenerEnergia(){
    return energia;}
public int obtenerRuedas(){
    return ruedas;}
public int obtenerOpticas(){
    return opticas;}
public int obtenerChasisA(){
    return chasisA;}
}
```

## Caso de Estudio: Fábrica de juguetes

```
//En Robot  
public int vidaUtil(){  
    return 1000-cantRecargas;  
}
```

## Caso de Estudio: Fábrica de juguetes

//En Robot

```
public int cantAutos(){  
    /*Calcula la cantidad de autos según las piezas, no considera la  
    energía requerida*/  
    int n;  
    if (ruedas/4 < opticas/4)  
        if (ruedas/4 < chasisA)  
            n = (int) ruedas/4;  
        else n = chasisA;  
    else  
        if (opticas/4 < chasisA)  
            n = (int) opticas/4;  
        else n = chasisA;  
    return n;  
}
```

## Caso de Estudio: Fábrica de juguetes

```
class RobotAlfa extends Robot {  
  //atributos de instancia  
  protected int chasisC;  
  //Constructores  
  public RobotAlfa (int nro){  
    super(nro) ;  
    chasisC = 100;  
  }  
}
```

El estado interno de cada instancia de **RobotAlfa** mantendrá los atributos de cualquier objeto de clase **Robot** más los específicos de **RobotAlfa**.



## Caso de Estudio: Fábrica de juguetes

```
//En RobotAlfa
public void armarAuto () {
/*Requiere que se haya controlado si hay piezas
disponibles*/
    super.armarAuto();
    ruedas -= 1 ;
    opticas -=1;
}
```

El método **armarAuto** de la clase **Robot** queda **derogado** para las instancias de la clase **RobotAlfa**.

## Caso de Estudio: Fábrica de juguetes

```
//En RobotAlfa
public void abrirCaja (CajaC caja) {
/*Aumenta sus cantidades según las de la caja. Requiere que se vacíe la caja
después de que el robot la abra*/
    ruedas += caja.obtenerRuedas();
    opticas += caja.obtenerOpticas();
    chasisA += caja.obtenerChasisA();
    chasisC += caja.obtenerChasisC();
    energia -= 50;
/*Controla si es necesario recargar energía*/
    if (energia < energiaMinima)
        this.recargar();
    caja.vaciar();
}
```

El comando está **sobrecargado**, tiene una signatura distinta que el método definido en **Robot**.

## Caso de Estudio: Fábrica de juguetes

```
//En RobotAlfa
public void armarCamion () {
/*Requiere que se haya controlado si hay piezas
disponibles*/
    ruedas -= 6 ;
    opticas -=6;
    energia -= 80;
    chasisC --;
/*Controla si es necesario recargar energía*/
    if (energia < energiaMinima)
        this.recargar() ;
}
```

El método `armarCambion` es específico de la clase `RobotAlfa`.

## Caso de Estudio: Fábrica de juguetes

```
//En RobotAlfa  
public int vidaUtil(){  
    return 5000-cantRecargas;  
}
```

El método `vidaUtil` queda **redefinido** en la clase `RobotAlfa`.

## Caso de Estudio: Fábrica de juguetes

```
//En RobotAlfa
public int cantAutos(){
/*Calcula la cantidad de autos según las piezas, no considera la
energía requerida*/
int n;
if (ruedas/5 < opticas/5)
    if (ruedas/5 < chasisA)
        n = (int) ruedas/5;
    else n = chasisA;
else
    if (ruedas/5 < chasisA)
        n = (int) opticas/5;
    else n = chasisA;
return n;
}
```

## Caso de Estudio: Fábrica de juguetes

```
//En RobotAlfa
public int cantCamiones(){
/*Calcula la cantidad de autos según las piezas, no considera la
energía requerida*/
int n;
if (ruedas/6 < opticas/6)
    if (ruedas/6 < chasisC)
        n = (int) ruedas/6;
    else n = chasisC;
else
    if (ruedas/6 < chasisC)
        n = (int) opticas/6;
    else n = chasisC;
return n;
}
```

## Caso de Estudio: Fábrica de juguetes

Un método derogado o redefinido conserva la **signatura** de la clase base, esto es tiene exactamente el mismo nombre y el mismo número y tipo de parámetros y el mismo tipo de resultado.

```
//En Robot
public int vidaUtil(){
    return 1000-cantRecargas;
}
```

```
//En RobotAlfa
public int vidaUtil(){
    return 5000-cantRecargas;
}
```

## Caso de Estudio: Fábrica de juguetes

```
//En Robot
public void armarAuto () {
/*Requiere que se haya controlado si hay piezas disponibles*/
    ruedas -= 4 ;
    opticas -=4;
    energia -= 70;
    chasisA --;
/*Controla si es necesario recargar energía*/
    if (energia < energiaMinima)
        //En RobotAlfa
        public void armarAuto () {
            /*Requiere que se haya controlado si hay piezas
            disponibles*/
            super.armarAuto();
            ruedas -= 1 ;
            opticas -=1;
        }
```



## Caso de Estudio: Fábrica de juguetes

Dada la siguiente declaración:

```
Robot rob;
```

La variable **rob** se dice **polimórfica** porque puede referenciar a objetos de su propia clase o de cualquier clase derivada.

## Caso de Estudio: Fábrica de juguetes

```
Robot rob;  
rob= new RobotAlfa (123) ;
```

La asignación es **polimórfica**, una variable declarada de una clase queda ligada a un objeto de una clase derivada.

## Caso de Estudio: Fábrica de juguetes

```
Robot rob;  
rob= new RobotAlfa (123) ;  
if (rob.cantAutos()>1)  
    rob.armarAuto() ;
```

Cuando el método está **redefinido**, ligadura de código es **dinámica**, la **clase del objeto** determina qué método se ejecuta en respuesta a cada mensaje.

## Caso de Estudio: Fábrica de juguetes

```
Robot rob;  
rob= new RobotAlfa (123) ;  
if (rob.cantCamiones()>1)  ERROR  
    rob.armarCamion() ;    ERROR
```

El chequeo de tipos es **estático**, la **declaración de la variable** determina qué mensajes se pueden enviar.

## Caso de Estudio: Fábrica de juguetes

```
Robot rob = new Robot (123) ;  
RobotAlfa bob1 , bob2 ;  
bob1 = new Robot (124) ;      ERROR  
bob2 = rob ;                  ERROR
```

El chequeo de tipos también rechaza una asignación en la cual el tipo de la variable a la izquierda de la asignación es más específico que el tipo de la derecha.

## Caso de Estudio: Fábrica de juguetes

```
Robot rob;  
rob= new Robot (123) ;  
Caja caja = new Caja() ;  
rob.abrirCaja(caja) ;
```

El mensaje se liga al método definido en **Robot**.

## Caso de Estudio: Fábrica de juguetes

```
RobotAlfa bob;  
bob= new RobotAlfa (123) ;  
CajaC caja = new CajaC() ;  
bob.abrirCaja(caja) ;
```

El mensaje se liga al método definido en **RobotAlfa**.

## Caso de Estudio: Fábrica de juguetes

```
Robot rob;  
rob= new RobotAlfa (123) ;  
Caja caja = new Caja() ;  
rob.abrirCaja(caja) ;
```

El mensaje se liga al método definido en **Robot**.

Como el **método abrirCaja** está **sobrecargado** la ligadura entre el mensaje y el método es **estática**.



## Caso de Estudio: Fábrica de juguetes

```
Robot rob;  
rob= new RobotAlfa (123) ;  
CajaC caja = new CajaC() ;  
rob.abrirCaja(caja) ;
```

**ERROR**

Como el método está **sobrecargado** la ligadura entre el mensaje y el método es **estática**.

## Caso de Estudio: Fábrica de juguetes

*Cada robot está asignado a uno o más sectores.*

*Algunos sectores pueden no tener asignado un robot.*

*El conjunto de sectores pueden mantenerse en un arreglo en el cual cada componente representa a un **sector** y puede mantener una referencia nula o estar ligado a un robot.*

## Caso de Estudio: Fábrica de juguetes

*Inicialmente todas las componentes del arreglo mantienen referencias nulas.*

*Cada vez que se asigna un robot a un sector, se liga un objeto a una componente del arreglo*

*Cada vez que se retira un robot de un sector, se asigna nulo a una componente del arreglo.*

*En todo momento puede procesarse el arreglo. Por ejemplo para calcular cuántos robots pueden preparar más de cierta cantidad de autos con las piezas disponibles.*

## Caso de Estudio: Fábrica de juguetes

La clase **SectoresFabrica** encapsula entonces un arreglo de objetos de clase **Robot** y brinda servicios para:

- ▶ *asignar un Robot  $r$  en un sector  $s$ , requiere que  $s$  represente un sector de la fábrica.*
- ▶ *asignar un Robot  $r$  en un sector libre, requiere que haya al menos un sector libre*
- ▶ *desasignar un Robot  $r$  de todos los sectores a los que está asignado*
- ▶ *desasignar el Robot de un sector  $s$ , requiere que  $s$  represente un sector de la fábrica.*

## Caso de Estudio: Fábrica de juguetes

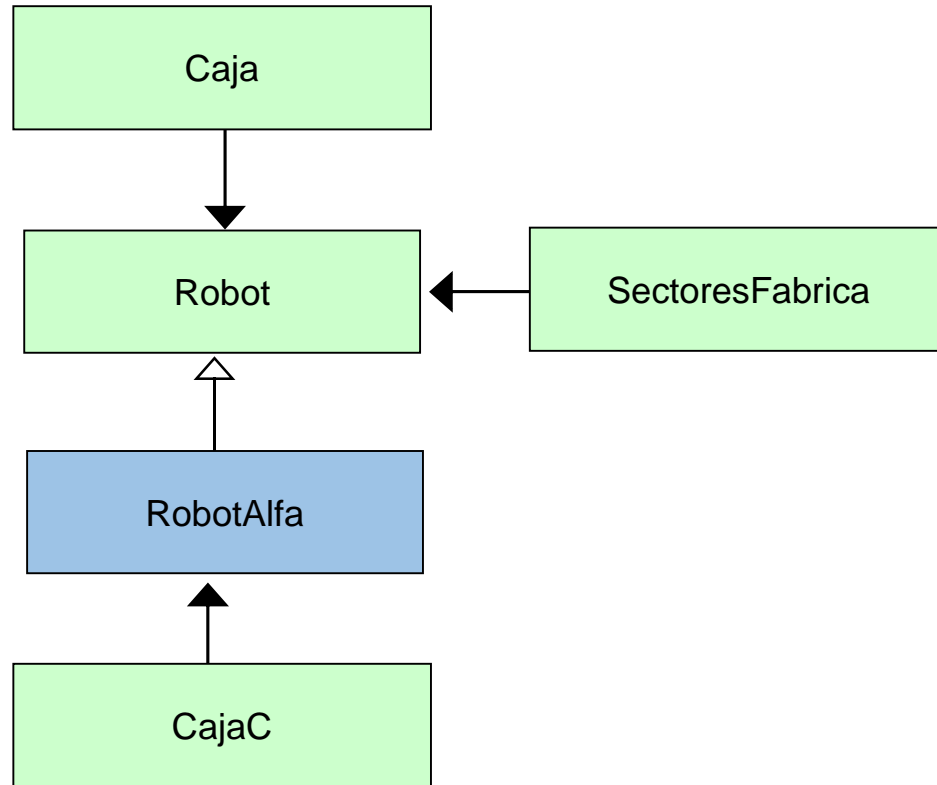
*La clase **SectoresFabrica** encapsula entonces un arreglo de objetos de clase **Robot** y brinda servicios para:*

- ▶ *asignar un Robot  $r$  en un sector  $s$ , requiere que  $s$  represente un sector de la fábrica.*
- ▶ *asignar un Robot  $r$  en un sector libre, requiere que haya al menos un sector libre*
- ▶ *desasignar un Robot  $r$  de todos los sectores a los que está asignado*
- ▶ *desasignar el Robot de un sector  $s$ , requiere que  $s$  represente un sector de la fábrica.*

## Caso de Estudio: Fábrica de juguetes

- ▶ *Decidir si algún sector tiene asignado un robot con la misma identidad que un robot dado.*
- ▶ *Recuperar el Robot asignado a un sector  $s$ , requiere que  $s$  represente un sector de la fábrica.*
- ▶ *Calcular la cantidad de sectores de la fábrica, esto es, la cantidad de componentes del arreglo.*
- ▶ *Calcular cuántos sectores tienen asignado un robot, esto es, cuántas referencias del arreglo están ligadas.*
- ▶ *Decidir si todos los sectores tienen asignado un robot, es decir, todas las componentes del arreglo están ligadas.*
- ▶ *Contar la cantidad de sectores asignados a robots con piezas para armar más de  $a$  autos.*

## Caso de Estudio: Fábrica de juguetes



SectoresFabrica

T [] Robot

<<constructores>>

SectoresFabrica (max : entero)

<<comandos>>

asignar (r : Robot, s : entero)

asignar (r : Robot)

desasignar (s : entero)

desasignar (r : Robot)

Robot

<<atributos de clase>>

energiaMaxima : 5000

energiaMinima : 100

<<atributos de instancia>>

nroSerie:entero

energia: entero

ruedas: entero

opticas: entero

chasisA: entero

cantRecargas:entero



## Caso de Estudio: Fábrica de juguetes

```
class SectoresFabrica {  
    private Robot[] T;  
  
    //Constructor  
    public SectoresFabrica(int max) {  
        /*Crea un arreglo con max elementos, cada  
        elemento representa un sector de la fábrica*/  
        T= new Robot [max];  
    }  
    ...  
}
```

La variable **T** mantiene una referencia a un arreglo de variables polimórficas.

## Caso de Estudio: Fábrica de juguetes

```
//Comandos
public void asignar (Robot r) {
    /*Busca el primer sector libre y asigna el robot
    r al sector. Requiere que haya un sector libre*/
    int i = 0;
    while (T[i] != null)
        i++;
    T[i] = r;
}
```

El comando **asignar** es un **método polimórfico**, recibe como parámetro a una variable polimórfica.

Es decir, **r** puede estar ligado a un objeto de clase **Robot** o a un objeto de clase **RobotAlfa**, en cualquier caso es una instancia de **Robot**.

## Caso de Estudio: Fábrica de juguetes

```
public void asignar (Robot r, int s) {  
    /*Asigna el robot r al sector s. Requiere  
    0<=s<cantSectores() */  
    T[s] = r;  
}
```

Si no se cumple el requerimiento, se produce un **error de ejecución**, la terminación va a ser anormal.

Si el sector ya tenía un robot asignado, implícitamente queda eliminado al asignarse un nuevo robot, probablemente sea un **error de aplicación**, aunque el diseñador no lo especificó como una responsabilidad.

Observemos que los **errores de compilación** son los más sencillos de detectar y corregir.

## Caso de Estudio: Fábrica de juguetes

```
public void desasignar(int s) {  
    /*Elimina la asignación del robot r del  
    sector s. Requiere  $0 \leq s < \text{cantSectores}()$  */  
    T[s] = null;  
}
```

Si no se cumple el requerimiento la terminación va a ser anormal.

Si el sector no tenía un robot asignado no se produce ningún cambio.

## Caso de Estudio: Fábrica de juguetes

```
public void desasignar(Robot r) {  
    /*Elimina la asignación del robot r de  
    todos los sectores a los que está  
    asignado*/  
    int i = 0;  
    while (i < cantSectores()){  
        if (T[i] ==r)  
            T[i] = null;  
        i++;  
    }  
}
```

Busca todos los sectores que tengan asignado un robot con la misma identidad que r.

## Caso de Estudio: Fábrica de juguetes

```
public int cantSectores() {  
    return T.length;  
}  
public int cantSectoresOcupados () {  
    int i = 0; int cant = 0;  
    while (i < cantSectores()) {  
        if (T[i] != null) cant++;  
        i++;  
    }  
    return cant;  
}
```

## Caso de Estudio: Fábrica de juguetes

```
public boolean todosOcupados () {  
    int i = 0; boolean hayNulo = false;  
    while (i < cantSectores() && !hayNulo) {  
        hayNulo = T[i] == null;  
        i++;  
    }  
    return !hayNulo;  
}
```

Algunos sectores pueden estar ocupados por objetos de clase **Robot** y otros por objetos de clase **RobotAlfa**, que también son instancias de **Robot**.

## Caso de Estudio: Fábrica de juguetes

```
public boolean estaRobot (Robot r){  
    /* Decide si algún sector tiene asignado  
    un robot con la misma identidad que r */  
    int i = 0; boolean esta = false;  
    while (i < cantSectores() && !esta ){  
        esta = T[i] == r ;  
        i++;  
    }  
    return esta;  
}
```

Busca un sector que tenga asignado un robot con la misma identidad que r. Observemos que si r es nulo y hay un sector libre retorna true.



## Caso de Estudio: Fábrica de juguetes

```
public boolean existeSector (int s){  
    return s>= 0 & s< cantSectores();  
}
```

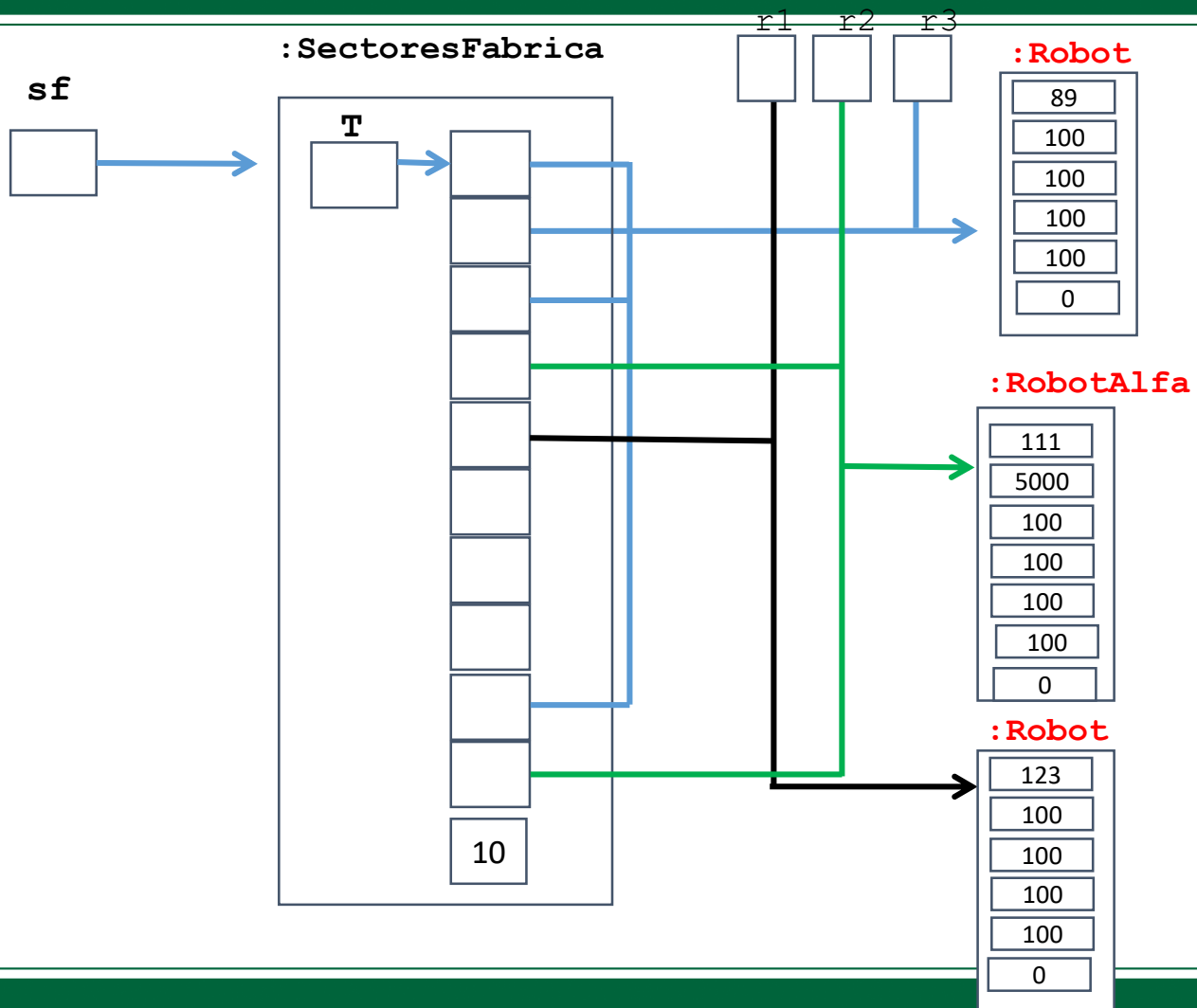
```
public Robot robotSector (int s){  
    /*Retorna el Robot en un sector s,  
    requiere 0<=s<cantSectores()*/  
    return T[s];  
}
```

## Caso de Estudio: Fábrica de juguetes

```
public int cantMasAutos(int a){  
    /*Cuenta la cantidad de sectores asignados  
    a robots que pueden armar más de a autos*/  
    int cont =0;  
    for (int i=0;i<cantSectores();i++)  
        if (T[i] != null)  
            if(T[i].cantAutos() > a)  
                cont++;  
  
    return cont;  
}
```

**T[i]** es una variable **polimórfica**. El mensaje **cantAutos()** se liga al método definido en la clase del objeto referenciado por **T[i]**.

```
class Fabrica{
public static void main (String a[]){
    SectoresFabrica sf = new
        SectoresFabrica (10);
    Robot r1,r2,r3;
    r1 = new Robot (123);
    r2 = new RobotAlfa (111);
    r3 = new Robot (89);
    sf.asignar(r3,1);
    sf.asignar(r2,3);
    sf.asignar(r3,8);
    sf.asignar(r3);
    sf.asignar(r2,9);
    sf.asignar(r3);
    sf.asignar(r1);}
}
```



# Polimorfismo

- ▶ El concepto de polimorfismo es central en la programación orientada a objetos.
- ▶ Polimorfismo significa **muchas formas** y en ciencias de la computación en particular se refiere a “**la capacidad de asociar diferentes definiciones a un mismo nombre, de modo que el contexto determine cuál corresponde usar**”.
- ▶ En el contexto de la programación orientada a objetos el polimorfismo está relacionado con **variables, asignaciones y métodos**.

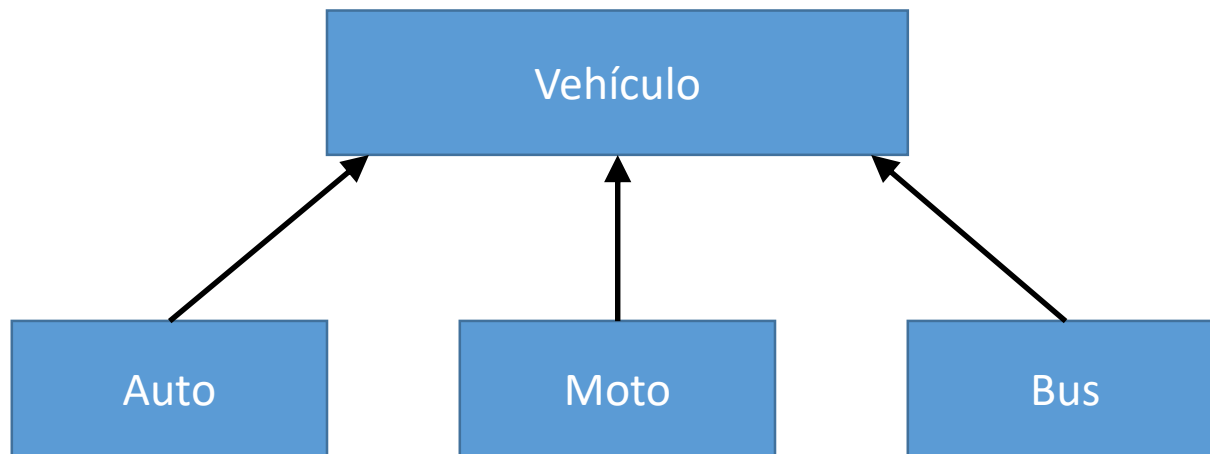
# Polimorfismo en Java

Una **variable polimórfica** puede quedar asociada a objetos de diferentes clases.

Una **asignación polimórfica** liga un objeto de una clase a una variable declarada de otra clase

Un **método polimórfico** incluye una o más variables polimórficas como parámetro.

# Polimorfismo



# Polimorfismo

Declaro la función:

```
function estacionar( Vehiculo ) { }
```

Invoco la función: (soporto polimorfismo)

```
estacionar( Coche ) ;
```

```
estacionar( Moto ) ;
```

```
estacionar( Bus ) ;
```

No puedo invocar la función: (no lo permitiría, porque no ser clasificación de herencia de vehículos)

```
estacionar( Mono ) ;
```

```
estacionar( INT ) ;
```

En el futuro si podría: (Si creo las clases "Van" o "Nave espacial" y heredan de Vehiculo)

```
estacionar( Van ) ;
```

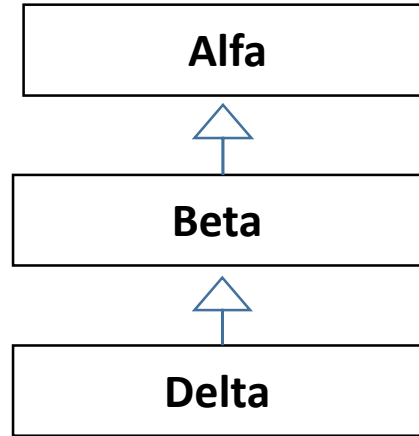
```
estacionar( Nave espacial ) ;
```



# Tipo Estático y Dinámico

- ▶ Dado que una variable puede estar asociada a objetos de diferentes tipos, distinguiremos entre:
- ▶ El **tipo estático** de una variable, es el tipo que aparece en la declaración.
- ▶ El **tipo dinámico** de una variable se determina en ejecución y corresponde a la clase a la que corresponde el objeto referenciado.
- ▶ El tipo estático de una variable determina el conjunto de tipos dinámicos a los que puede quedar asociada y los mensajes que puede recibir.

# Tipo Estático y Dinámico



Tipo Estático	Tipos Dinámicos
Alfa	Alfa Beta Delta
Beta	Beta Delta
Delta	Delta

# Tipo Estático y Dinámico

```
Alfa v0 = new Beta (...);  
Beta v1 = new Delta (...);  
Delta v2 = new Delta (...);  
Alfa v3 = new Delta (...);
```

Variable	Tipo Estático	Tipo Dinámico
v0	Alfa	Beta
v1	Beta	Delta
v2	Delta	Delta
v3	Alfa	Delta

# Asignación polimórfica

- Una **asignación polimórfica** liga un objeto de una clase a una variable declarada de otra clase.

```
Alfa w0;
```

```
Beta w1 = new Beta (...);
```

```
Delta w2 = new Delta (...);
```

Son válidas las siguientes asignaciones polimórficas:

```
w0 = w1;
```

```
w0 = w2;
```

```
w1 = w2;
```

```
w0 = new Beta (...);
```

```
w0 = new Delta (...);
```

```
w1 = new Delta (...);
```

# Método polimórfico

- El pasaje de parámetros puede involucrar una asignación polimórfica:

```
Beta v1 = new Beta();
```

```
Delta v2 = new Delta();
```

El método definido en la clase **Beta** como:

```
public boolean p( Beta e )  
{ ... }
```

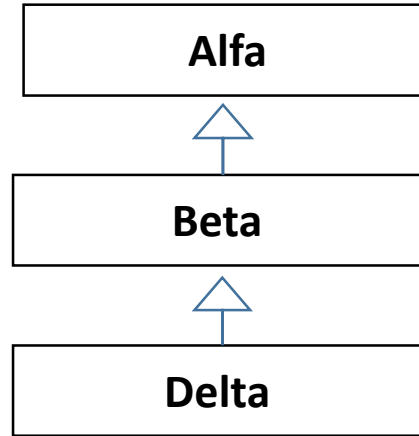
Puede usarse con un argumento de clase **Delta**:

```
v1.p ( v2 ) ;
```

# Ligadura dinámica de código

- ▶ La **ligadura dinámica de código** es la vinculación en ejecución de un mensaje con un método.
- ▶ Polimorfismo, redefinición de métodos y ligadura dinámica de código son conceptos fuertemente ligados.
- ▶ La posibilidad de que una variable pueda referenciar a objetos de diferentes clases y de que existan varias definiciones para una misma signatura, brinda flexibilidad al lenguaje siempre que además exista ligadura dinámica de código.

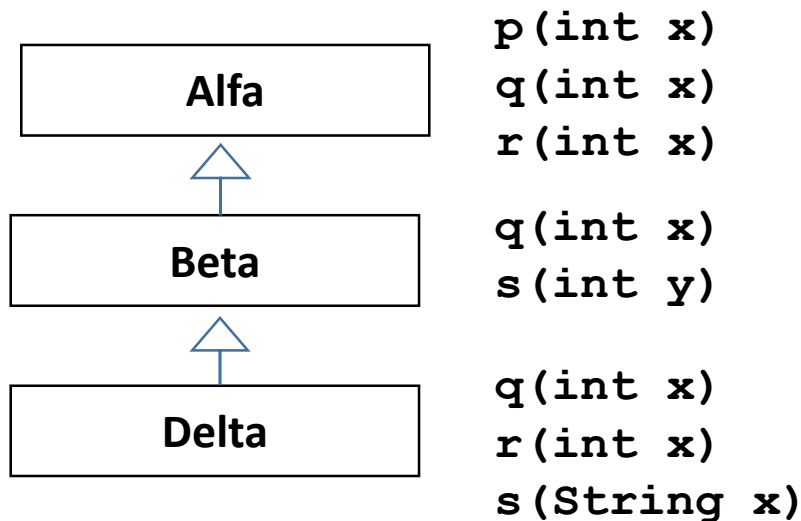
# Tipo Estático y Dinámico



Tipo Estático	Tipos Dinámicos
Alfa	Alfa Beta Delta
Beta	Beta Delta
Delta	Delta

# Ligadura dinámica de código

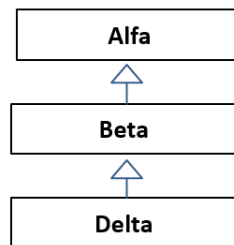
```
Alfa v1 = new Alfa (...);  
Alfa v2 = new Beta (...);  
Alfa v3 = new Delta (...);
```





# Ligadura dinámica de código

```
Alfa v1 = new Alfa(...);  
Alfa v2 = new Beta(...);  
Alfa v3 = new Delta(...);  
Delta v4 = new Delta(...);
```



```
p(int x)  
q(int x)  
r(int x)  
  
q(int x)  
s(int y)  
  
q(int x)  
r(int x)  
s(String x)
```

```
v1.p(1);
```

Alfa

```
v2.p(1);
```

Alfa

```
v3.p(1);
```

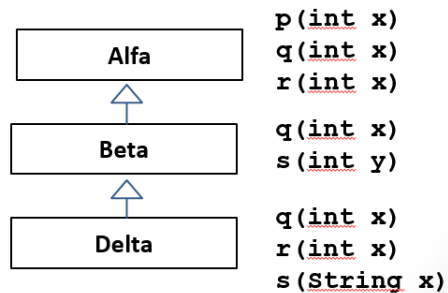
Alfa

```
v4.p(1);
```

Alfa

# Ligadura dinámica de código

```
Alfa v1 = new Alfa (...);  
Alfa v2 = new Beta (...);  
Alfa v3 = new Delta (...);
```



```
v1.q(1);
```

Alfa

```
v2.q(1);
```

Beta

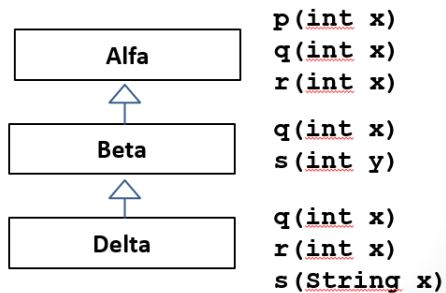
```
v3.q(1);
```

Delta

El **tipo dinámico** determina la **ligadura** entre el mensaje y el método.

# Ligadura dinámica de código

```
Alfa v1 = new Alfa (...);  
Alfa v2 = new Beta (...);  
Alfa v3 = new Delta (...);
```



```
v1.r(1);
```

Alfa

```
v2.r(1);
```

Alfa

```
v3.r(1);
```

Delta

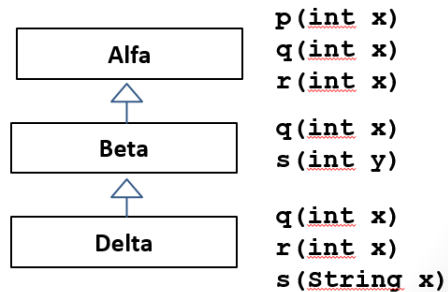
El **tipo dinámico** determina la **ligadura** entre el mensaje y el método.

# Chequeo de tipos en Java

- ▶ El polimorfismo es un mecanismo que favorece la **reusabilidad** pero debe restringirse para brindar **robustez**
- ▶ En Java el polimorfismo y la ligadura dinámica quedan restringidos por el **chequeo de tipos**.
- ▶ Los chequeos de tipos en compilación previenen errores de tipo en ejecución.
- ▶ El chequeo de tipos establece restricciones sobre:
  - las asignaciones polimórficas
  - los mensajes que un objeto puede recibir

# Chequeo de tipos en Java

```
Alfa v1 = new Alfa(...);  
Beta v2 = new Beta(...);  
Delta v3 = new Delta(...);
```



```
v2 = v1;
```

Error

```
v3 = v1;
```

Error

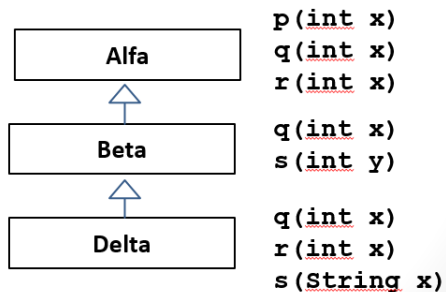
```
v3 = new Beta(...);
```

Error

El **tipo estático** restringe las asignaciones polimórficas.

# Chequeo de tipos en Java

```
Alfa v1 = new Beta (...);  
Beta v2 = new Delta (...);  
Beta v3 = new Delta (...);
```



```
v1.s(1);
```

Error

```
v2.s(1);
```

```
v3.s("abc");
```

Error

El **tipo estático** determina los mensajes que el objeto puede recibir.

# Interfaces

The background of the slide is a light gray with a subtle pattern of overlapping hexagons and thin white lines connecting some of them, creating a network-like or molecular structure. The hexagons vary in opacity and size, with some being solid white and others being semi-transparent.

# Interfaces en Java

En Java una **interface** es un conjunto de métodos relacionados sin una implementación concreta.

Una interface especifica las signaturas de un conjunto de métodos que luego van a ser implementados por una o más clases.

Todos los métodos provistos por una interface son públicos y no están implementados.

Una interface define un **tipo** a partir del cual es posible declarar variables pero no crear instancias.



# Interfaces en Java

Las interfaces pueden organizarse en una **estructura jerárquica**, donde cada nivel especializa al anterior.

Enfatizamos que una interface NO es una clase, no tiene variables de instancia, ni implementa los servicios provistos.

La definición de interfaces permite simular **herencia múltiple**.

Una clase D puede **extender** una clase B e **implementar** una interface I.

# Interfaces en Java

```
interface ObjetoGrafico {  
    void trasladar(int x,int y);  
    void rotar(float x);  
    void dibujar();  
}
```

## Interfaces en Java

```
class Poligono {  
    private ColeccionPuntos l;  
    float perimetro() {...};  
}
```

## Interfaces en Java

```
class Cuadrado extends Poligono
implements ObjetoGrafico {
    float perimetro() {...}
    void trasladar(int x,int y) {...}
    void rotar(float x) {...}
    void dibujar() {...}
}
```

# Interfaces en Java

- ▶ Una interface puede definir variables y constantes de clase, pero no de instancia.
- ▶ Java brinda interfaces y permite definir otras nuevas.

# Clases embebidas

Una **clase embebida** es una clase que se define dentro de otra

Esta característica permite anidar clases relacionadas y controlar la visibilidad

```
class externa {  
...  
    class interna {  
    }  
}
```

## Clases embebidas

Por una cuestión de estilo por lo general las clases internas se declaran a continuación de las variables de instancia y los métodos.

El acceso a una clase interna se limita a la clase externa.

El nombre de la clase interna puede ser reusado fuera de la clase externa.

Desde la clase interna se tiene acceso a todas las entidades de la clase externa, públicas y privadas.

# Classes Embebidas

```
class externa {  
    private int x = 1;  
    public int p () {  
        interna i = new interna();...  
    }  
    class interna {  
        public int q () {  
            x++;  
        }  
    }  
}
```



# Clases Embebidas

```
class prueba {  
    externa e = new externa();  
    e.p();  
    externa.interna i;  
}
```

# Clases Embebidas

Las instancias de la clase externa se crean como siempre.

Los métodos de la clase externa pueden crear instancias de la clase interna.

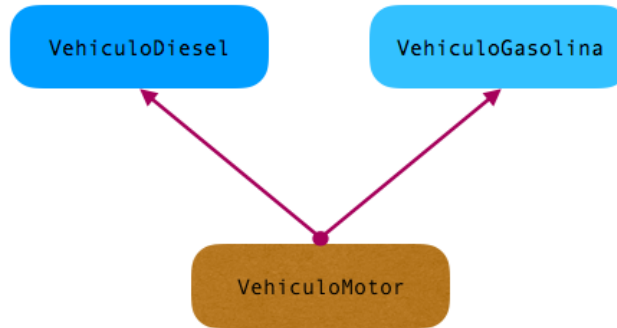
Un objeto de la clase interna estará siempre asociado a una instancia de la clase externa.

Nuevamente el concepto de clase embebida va a ser aplicado en las próximas clases cuando desarrollen interfaces gráficas simples.

# Problema del Diamante

# Problema del diamante

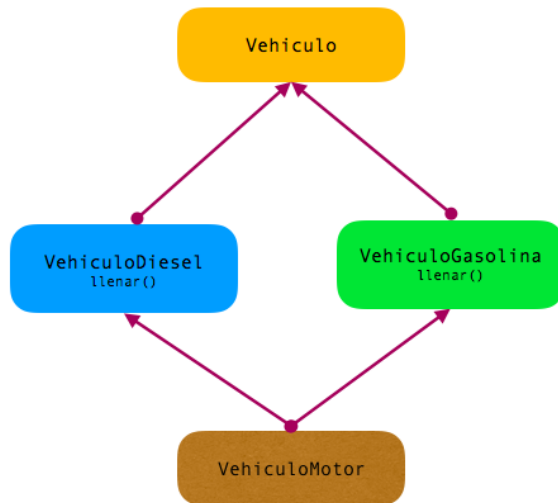
## Herencia Múltiple



La clase VehiculoMotor  
hereda de las clases  
VehiculoDiesel y VehiculoGasolina

# Problema del diamante

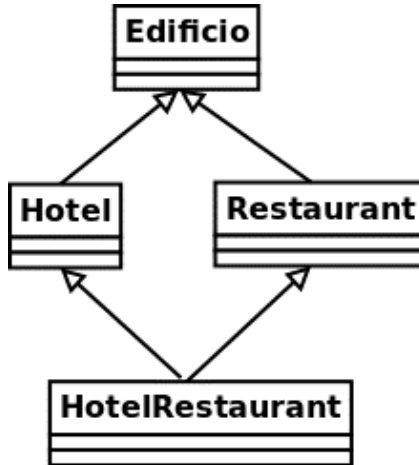
## Problema del Diamante



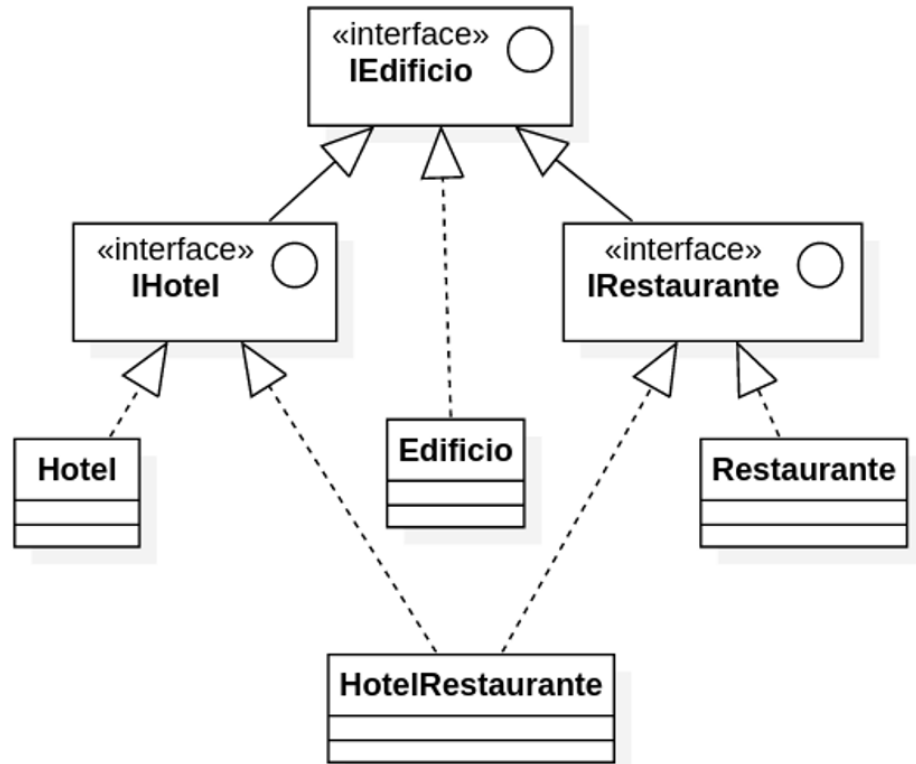
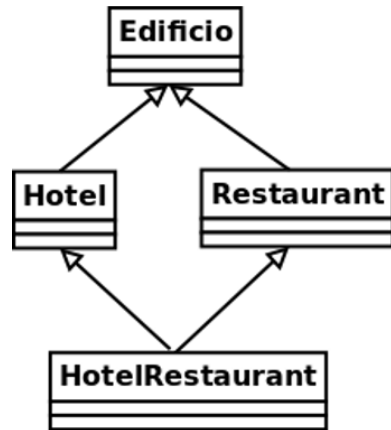
Dado que la clase VehiculoMotor extiende de las clases VehiculoDiesel y VehiculoGasolina y ambas clases tienen un método llenar(). ¿Cuando la clase VehiculoMotor invoque el método llenar(), a cuál de los dos métodos estaría invocando, al de la clase VehiculoDiesel o al de la clase VehiculoGasolina?

# Herencia multiple

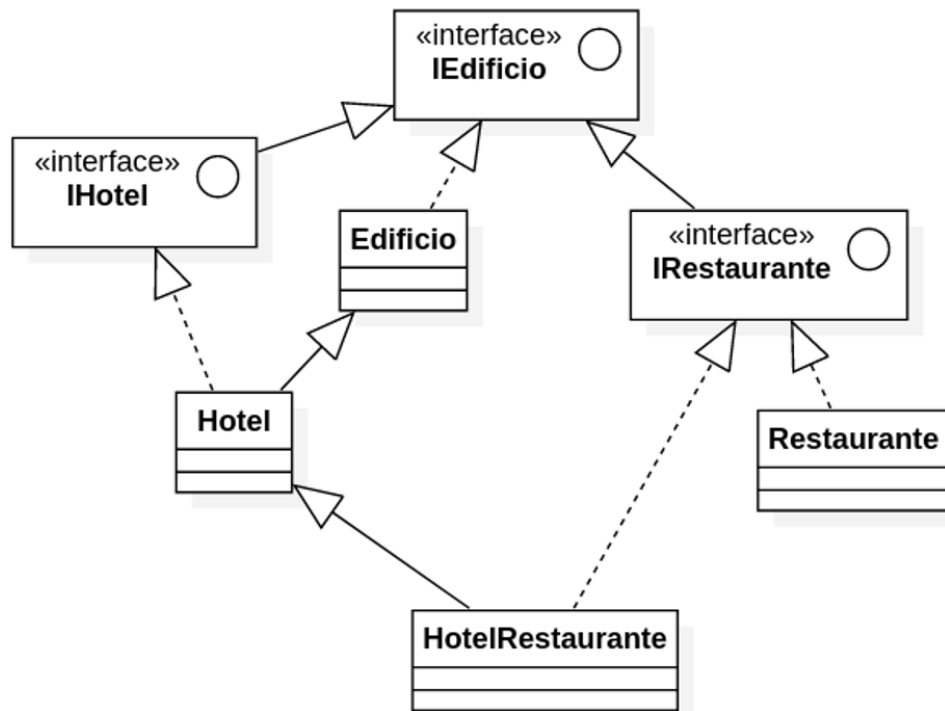
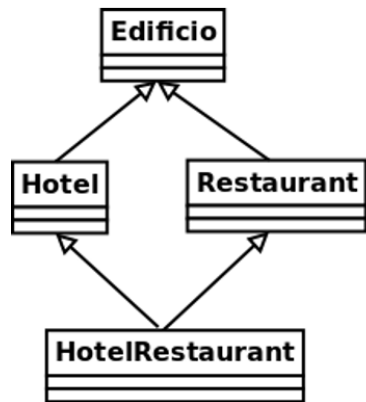
- ▶ Complejidad y Ambigüedad
- ▶ Se puede evadir la herencia multiple???
- ▶ ... tener en cuenta que puede pasar en cada propuesta con la claridad del diseño, polimorfismo y reutilización de código.



# Interfaces

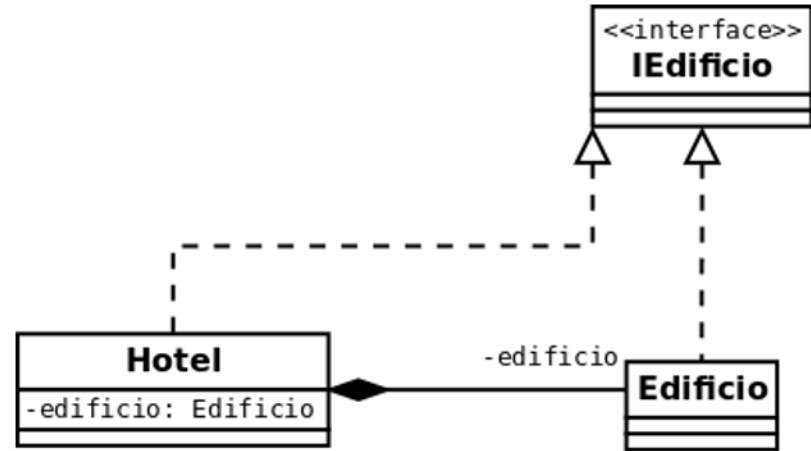
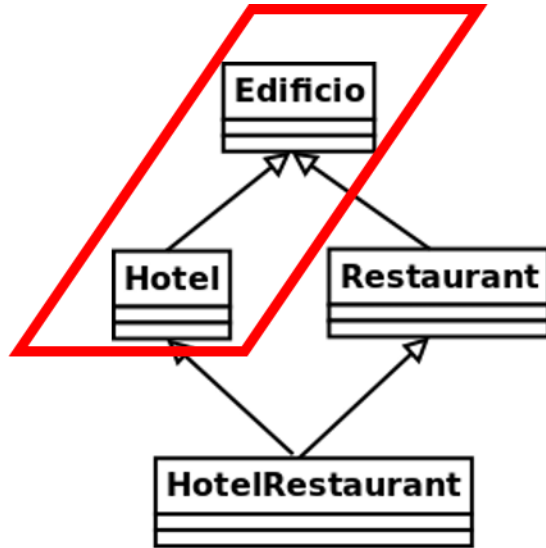


# Interfaces + Herencia Simple

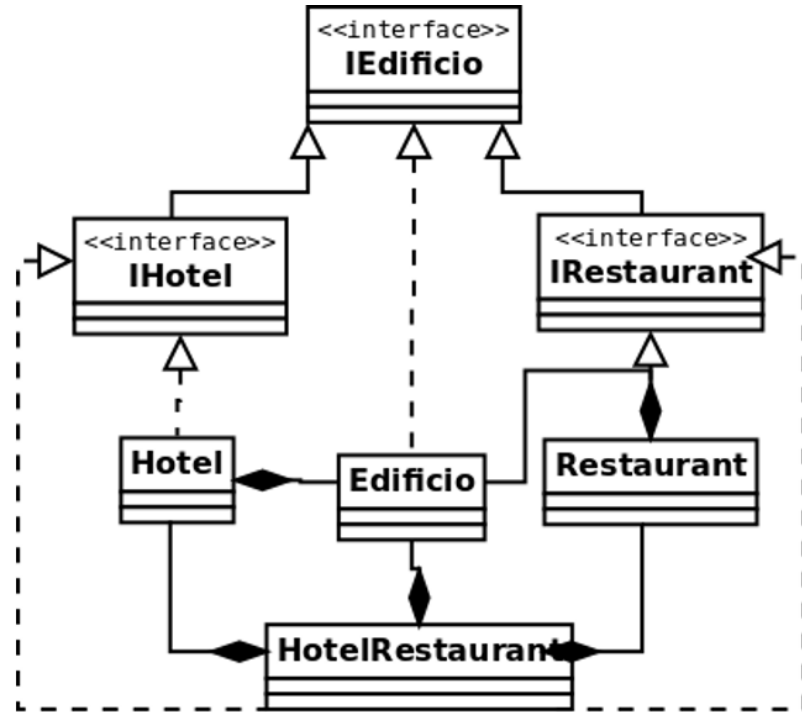
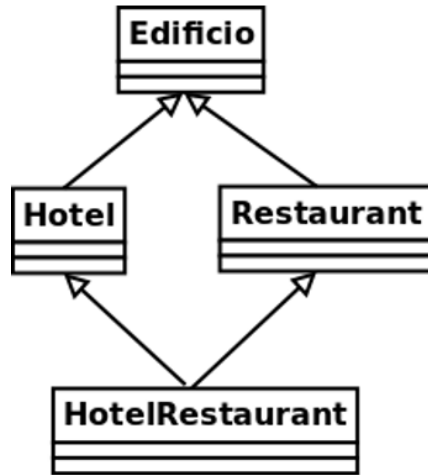




# Interfaces + Composición



# Interfaces + Composición



# Clases Abstractas

# Clases abstractas

En el diseño de una aplicación es posible definir una clase que factoriza propiedades de otras clases más específicas, sin que existan en el problema objetos concretos vinculados a esta clase más general.

En este caso la clase se dice **abstracta** porque fue creada artificialmente para lograr un diseño que modele la realidad.

En ejecución no va a haber objetos de software de una clase abstracta.

# Clases abstractas

Una clase abstracta puede incluir uno, varios, todos o ningún método abstracto.

Si una clase hereda de una clase abstracta y no implementa todos los métodos abstractos, también debe ser definida como abstracta.

Una clase concreta debe implementar todos los métodos abstractos de sus clases ancestro.

El constructor de una clase abstracta sólo va a ser invocado desde los constructores de las clases derivadas.

## Caso de estudio: Agencia Publicitaria

*Una agencia publicitaria publica avisos en diferentes medios de comunicación: televisión, radio, diarios y revistas.*

*Cada aviso tiene asociado un nombre de fantasía, un producto, una empresa y las fechas inicial y una duración en días.*

*Las campañas en radios y televisión tienen una emisora, una duración en segundos y una cantidad de repeticiones por día. No hay dos avisos de radio o TV con el mismo nombre, de una misma empresa. Los atributos nombre-empresa son la **clave***

## Caso de estudio: Agencia Publicitaria

*Los avisos publicados en diarios y revistas tienen un título, una cantidad de centímetros cuadrados de texto. No hay dos avisos impresos con el mismo nombre de una misma empresa. Los atributos nombre-empresa son la **clave***

*El costo de una campaña en radio o televisión se calcula como el producto entre la cantidad de días que dura la campaña, por la cantidad de repeticiones por día, por la duración en segundos, por un monto por segundo fijo.*

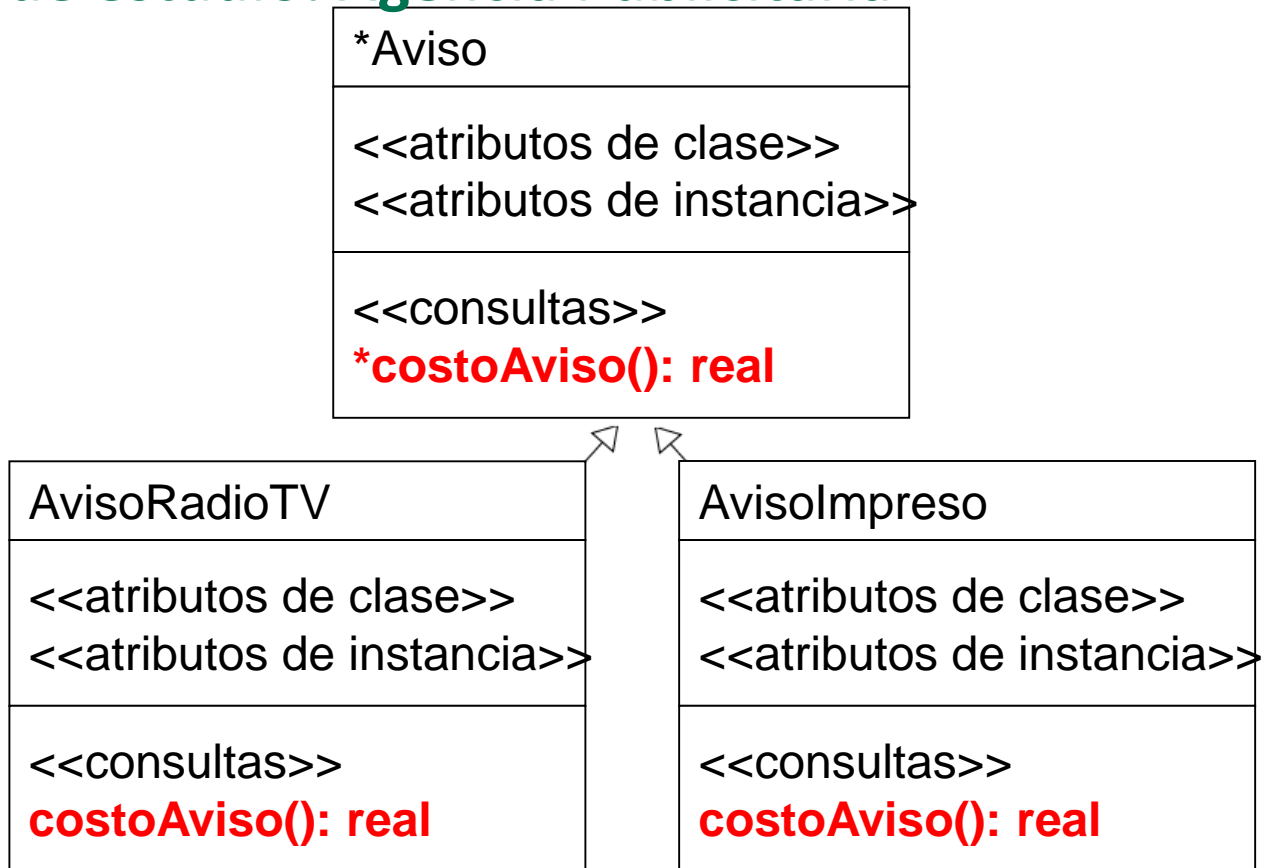
*El costo de una campaña en diarios o revistas se calcula como el producto entre la cantidad de centímetros del aviso, un monto fijo por centímetro y la cantidad de días que dura la campaña.*

## Caso de estudio: Agencia Publicitaria

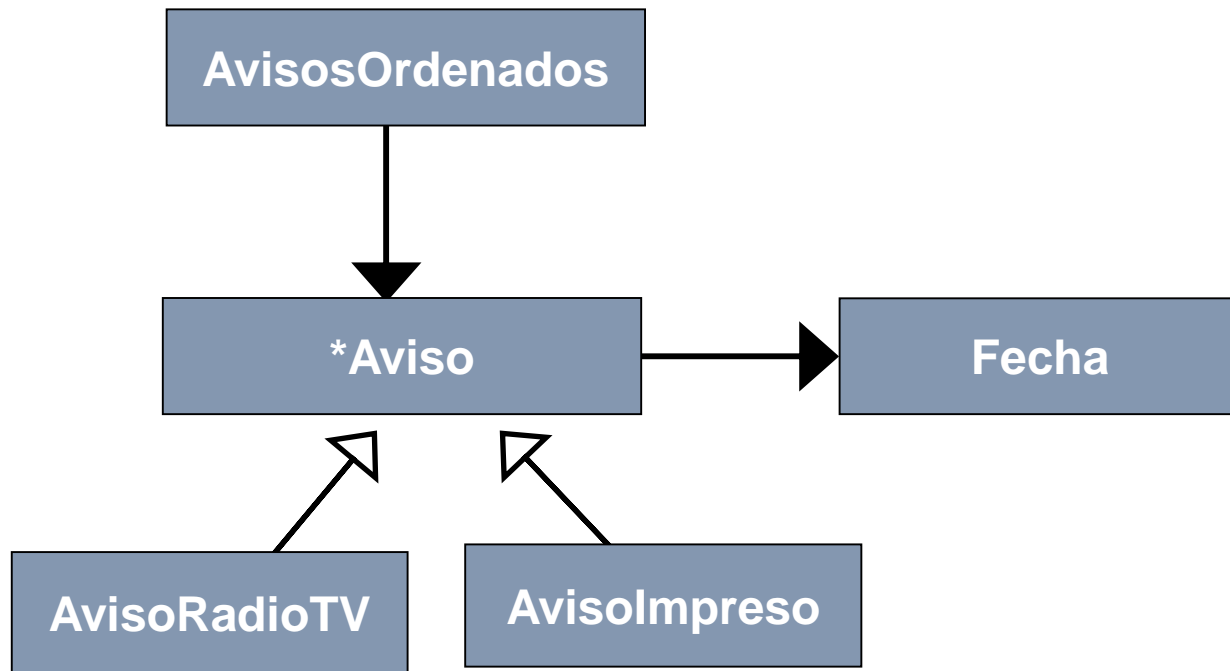
- ▶ En el caso de estudio descrito un aviso concreto se publica o bien en radio o en televisión o en revistas o en diarios.
- ▶ De modo que podemos definir una clase **Aviso** para factorizar atributos y comportamiento compartido. Esta clase no modela ningún objeto del problema real, **no tiene sentido crear objetos de software de esta clase**.
- ▶ El método **costoAviso()** es abstracto, todos los avisos tienen un costo pero la manera de calcularlo depende del medio en el que se publica.



## Caso de estudio: Agencia Publicitaria



## Caso de estudio: Agencia Publicitaria



## Caso de estudio: Agencia Publicitaria

```
abstract class Aviso{  
  
    protected String nombre;  
    protected String producto;  
    protected String empresa;  
    protected Fecha desde;  
    protected int dias;
```

El atributo **dias** indica la cantidad de días que dura la campaña.

No hay dos avisos que coincidan en los atributos nombre-empresa, es decir, puede haber dos avisos de la misma empresa O con el mismo nombre, pero no con de la misma empresa Y con el mismo nombre.

## Caso de estudio: Agencia Publicitaria

```
abstract class Aviso{  
  
    protected String nombre;  
    protected String producto;  
    protected String empresa;  
    protected Fecha desde;  
    protected int dias;  
    //Constructor  
    public Aviso (String n, String p,  
                  String e, Fecha d, int di)
```

Como no existen instancias de una clase abstracta, el constructor de una clase no va a ser invocado explícitamente para crear objetos de la clase, sino desde los constructores de las clases derivadas.

## Caso de estudio: Agencia Publicitaria

```
class AvisoImpreso extends Aviso{
protected static final
    float costoTexto= 58;
protected String titulo;
protected int cmTexto;
//Constructor
public
    AvisoImpreso(String n, String p,
                  String e,
                  Fecha d, int di,
                  String tit, float c){
    super(n,p,e,d,di) ;
    titulo = tit;
    cm = c;
}
```

## Caso de estudio: Agencia Publicitaria

```
class AvisoRadioTV extends Aviso{
protected static final
    float costoSegundo= 100;

protected String emisora;
protected int duracion;
protected int frecuencia;
public
    AvisoRadioTV(String n, String p,
                  String e,
                  Fecha d,int di,
                  String em,int du,int fr){
        super(n,p,e,d,di) ;
        emisora = em;
        duracion = du; frecuencia = fr;    }
```

## Caso de estudio: Agencia Publicitaria

```
AvisoImpreso ai;  
AvisoRadioTV artv;
```

```
Aviso aviso = new Aviso (...);
```

Error en compilación, la clase Aviso es abstracta

## Caso de estudio: Agencia Publicitaria

La clase **Aviso** es **abstracta** porque fue creada artificialmente para factorizar los atributos y el comportamiento común a todos los avisos publicitarios.

Podemos declarar variables de clase **Aviso** pero no crear objetos.

El constructor de la clase **Aviso** solo va a ser invocado desde los constructores de las clases derivadas.

En ejecución no va a haber instancias de una clase abstracta.



## Caso de estudio: Agencia Publicitaria

```
abstract class Aviso {  
    abstract public float costoAviso() ;  
    ... }  

```

```
class AvisoImpreso extends Aviso{  
    public float costoAviso(){  
        return desde.cantDias(hasta)*  
            cmTexto*costoTexto;    }  
    ...}  

```

```
class AvisoRadioTV extends Aviso{  
    public float costoAviso() {  
        return duracion*frecuencia*costoSegundo  
            *dias;                }  
    ...}  

```

## Caso de estudio: Agencia Publicitaria

En este caso la clase **Aviso** declara un método abstracto **costoAviso()** .

Cada clase que especialice a la clase **Aviso** y defina el método **costoAviso()** será una clase concreta.

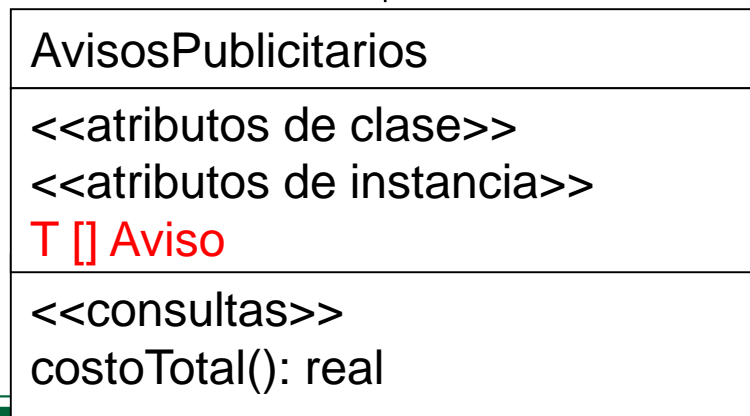
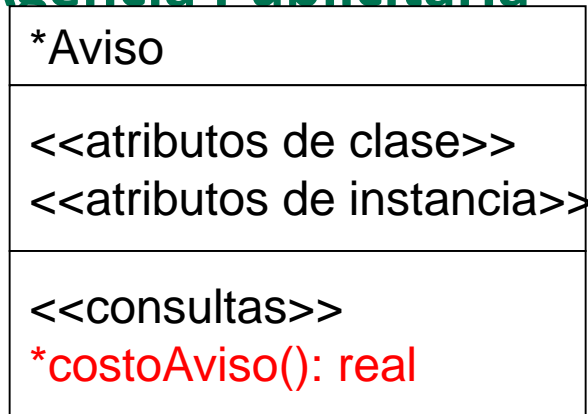
## Caso de estudio: Agencia Publicitaria

- ▶ *La clase AvisosPublicitarios encapsula una colección de elementos de clase Aviso, representada con un arreglo parcialmente ocupado.*
- ▶ *La clase brinda servicios para:*
  - *Insertar un nuevo Aviso, requiere que la colección no esté llena, no exista un Aviso con la misma clave y el aviso no sea nulo.*
  - *Eliminar un Aviso*
  - *Decidir si existe un aviso con una **clave** dada.*

## Caso de estudio: Agencia Publicitaria

- ▶ *La clase AvisosOrdenados encapsula una colección de elementos de clase Aviso, representada con un arreglo parcialmente ocupado y ordenado de acuerdo a la **clave**, primero por empresa y luego por nombre.*
- ▶ *La clase brinda servicios para:*
  - *Insertar ordenadamente un nuevo Aviso, requiere que la colección no esté llena y el aviso no sea nulo.*
  - *Eliminar un Aviso*
  - *Calcular el costo total de todos los avisos*
- ▶ ...

## Caso de estudio: Agencia Publicitaria



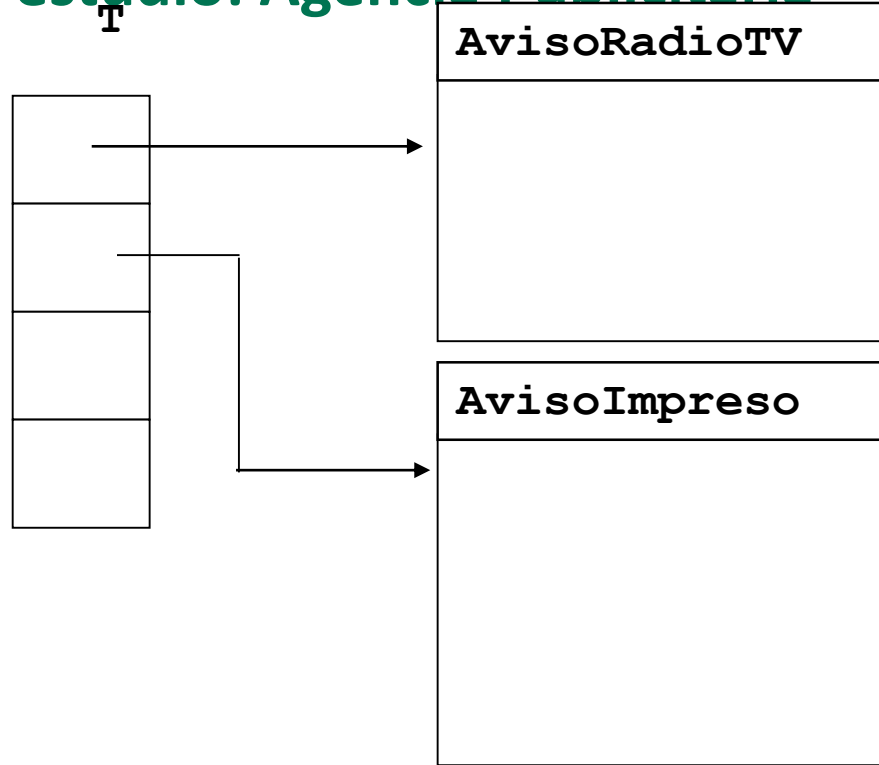
## Caso de estudio: Agencia Publicitaria

```
class AvisosPublicitarios {  
    /*Mantiene una colección de Avisos */  
    //Atributos de instancia  
    private Aviso [] T;  
    private int cant;  
    //Constructor  
    public AvisosPublicitarios(int max) {  
        T = new Aviso[max];  
    }  
    //Comandos  
    public void insertar(Aviso a) {  
        T[cant++] = a;  
    }  
}
```

## Caso de estudio: Agencia Publicitaria

```
class Agencia {  
...  
    AvisosPublicitarios agencia;  
    AvisoImpreso ai;  
    AvisoRadioTV artv;  
    agencia = new AvisosPublicitarios(10);  
    artv = new AvisoRadioTV (...);  
    ai = new AvisoImpreso (...);  
    if (!agencia.estaLlena)  
        agencia.insertar(artv);  
    if (!agencia.estaLlena())  
        agencia.insertar(ai);
```

## Caso de estudio: Agencia Publicitaria



Los elementos son instancias de clases derivadas de la clase **Aviso**



## Caso de estudio: Agencia Publicitaria

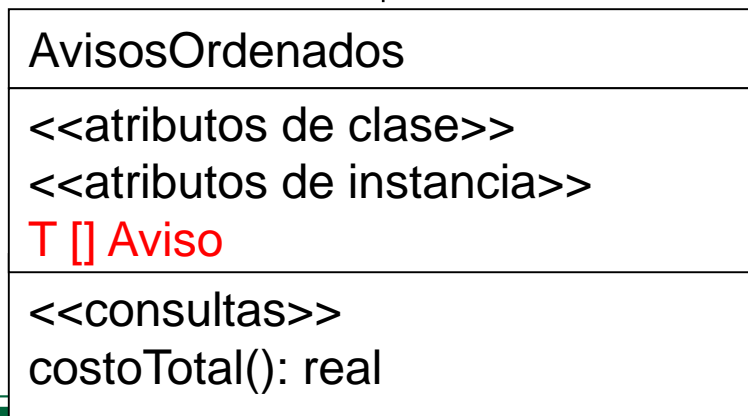
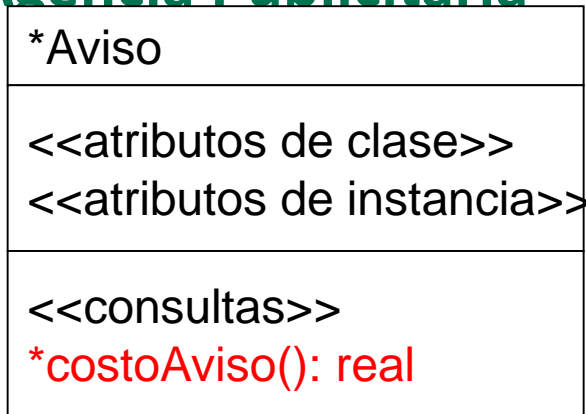
```
class Agencia {  
...  
    AvisosPublicitarios agencia;  
    AvisoImpreso ai;  
    AvisoRadioTV artv;  
    agencia = new AvisosPublicitarios(10);  
    artv = new AvisoRadioTV (...);  
    ai = new AvisoImpreso (...);  
    if (!agencia.estaLlena)  
        agencia.insertar(artv);  
    if (!agencia.estaLlena())  
        agencia.insertar(ai);  
  
    float ct = a.costosTotal();
```

## Caso de estudio: Agencia Publicitaria

```
//Consultas en la clase AvisosPublicitarios
public double costoTotal () {
    float c = 0;
    for (int i=0;i< cantAvisos();i++)
        c = c+T[i].costoAviso();

    return c;
}
```

## Caso de estudio: Agencia Publicitaria



## Caso de estudio: Agencia Publicitaria

```
class AvisosOrdenados {  
    /*Mantiene una colección de Avisos  
    ordenados por empresa y luego por  
    nombre. */  
    //Atributos de instancia  
    private Aviso [] T;  
    private int cant;  
    //Constructor  
    public AvisosOrdenados(int max){  
        T = new Aviso[max];  
    }  
}
```

## Caso de estudio: Agencia Publicitaria

```
class Agencia {
```

```
...
```

- Recorre la colección para decidir si pertenece y luego nuevamente para insertar ordenadamente.

```
AvisosOrdenados a;  
AvisoImpreso ai;  
AvisoRadioTV artv;
```

```
a = new AvisosOrdenados(10);
```

```
artv = new AvisoRadioTV (...);
```

```
ai = new AvisoImpreso (...);
```

```
if (!a.estaLlena() &&  
    !a.pertenece(artv))
```

```
    a.insertar(artv);
```

## Caso de estudio: Agencia Publicitaria

```
class Agencia {  
...  
    AvisosOrdenados a;  
    AvisoImpreso ai;  
    AvisoRadioTV artv;  
    a = new AvisosOrdenados(10);  
    artv = new AvisoRadioTV(Publicitaria);  
    ai = new AvisoImpreso (...);  
    if (!a.estaLlena() &&  
        !a.pertenece(artv))  
        a.insertar(artv);  
    if (!a.estaLlena() &&  
        !a.pertenece(ai))  
        a.insertar(ai);  
    float ct = a.costosTotal();
```

## Caso de estudio: Agencia Publicitaria

```
class AvisosOrdenados {  
...  
//Consultas  
public double costoTotal () {  
    float c = 0;  
    for (int i=0;i < cantAvisos();i++)  
        c = c+T[i].costoAviso() ;  
    return c;  
}  
...  
}
```

La ligadura entre el mensaje y el método **costoAviso** se establece en ejecución y depende de la clase del objeto referenciado por **T[i]**

## Caso de estudio: Agencia Publicitaria

```
abstract class Aviso{  
  
    protected String nombre;  
    protected String producto;  
    protected String empresa;  
    protected Fecha desde;  
    protected Fecha hasta;  
}
```

Si los atributos se acceden desde las clases derivadas, una modificación en la representación puede requerir modificar a las clases derivadas.



## Caso de estudio: Agencia Publicitaria

```
class AvisoRadioTV extends Aviso{  
    public float costoAviso() {  
        return duracion*frecuencia*costoSegundo  
            *dias) ;  
    }  
}
```

Si los atributos se acceden indirectamente a través de los servicios, la implementación puede cambiar y el cambio no afecta a las clases clientes.

```
class AvisoRadioTV extends Aviso{  
    public float costoAviso() {  
        return duracion*frecuencia*costoSegundo  
            *obtenerDias() ) ;  
    }  
}
```

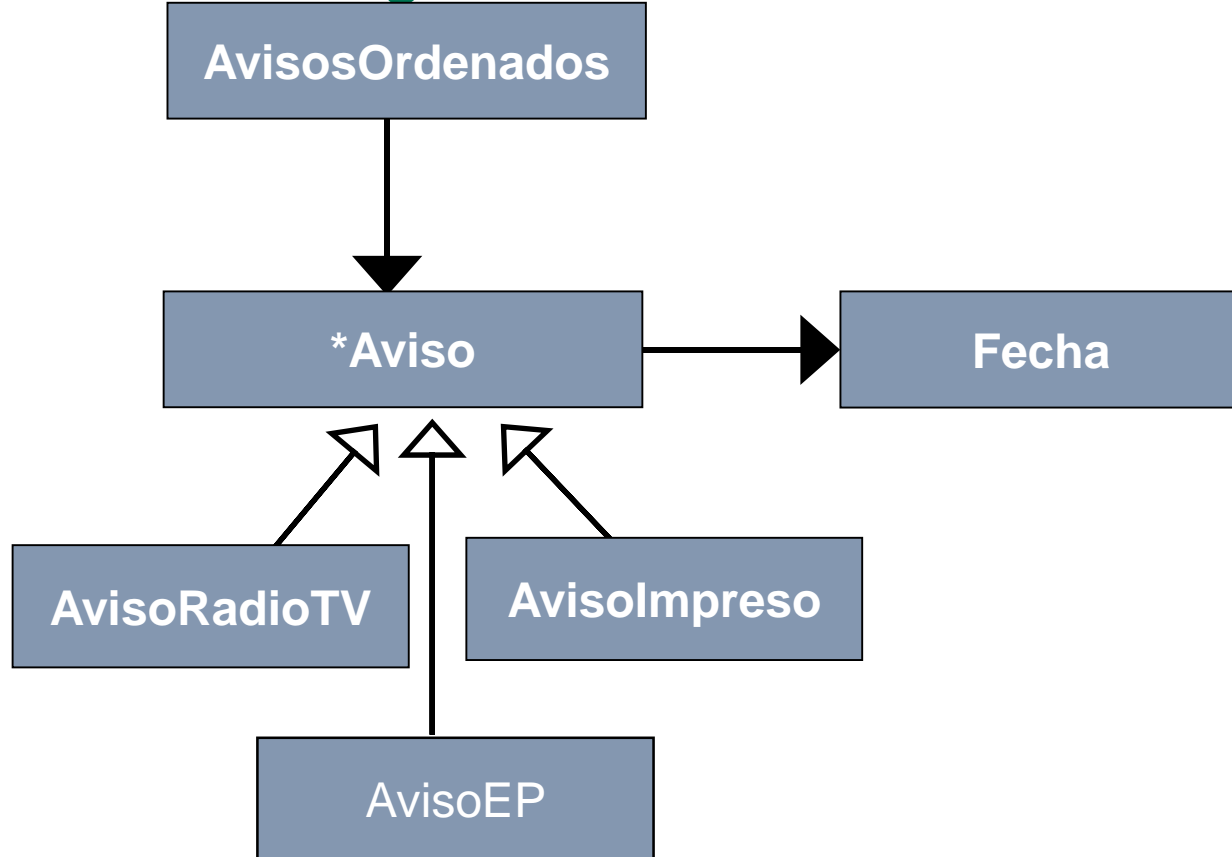
## Caso de estudio: Agencia Publicitaria

*Consideremos que el diseño del problema tiene que extenderse para incluir también avisos en espacios públicos que administra la Municipalidad. La Municipalidad tiene codificados los espacios públicos con un código numérico que los identifica y ofrece tres tipos de carteles para colocar en esos espacios a los que denomina A, B y C.*

*De modo que la clase AvisoEP tiene dos atributos codigoEP (entero) y tipoCartel (carácter).*

*El costo de un aviso en un espacio público depende del tipo de cartel; el costo del cartel de tipo B es el doble que el de tipo A (que es una constante) y el C el doble que el B.*

## Caso de estudio: Agencia Publicitaria



## Caso de estudio: Agencia Publicitaria

AvisoEP

<<atributos de clase>>

costoBase: real

<<atributos de instancia>>

codigoEP : entero

tipoCartel : char

<<consultas>>

**costoAviso(): real**

# Caso de estudio: Agencia Publicitaria

\*Aviso

<<atributos de clase>>

<<atributos de inst.>>

<<consultas>>

**\*costoAviso(): real**

AvisoRadioTV

<<atributos de clase>>

<<atributos de inst.>>

<<consultas>>

**costoAviso(): real**

AvisoImpreso

<<atributos de clase>>

<<atributos de inst.>>

<<consultas>>

**costoAviso(): real**

AvisoEP

<<atributos de clase>>

<<atributos de inst.>>

<<consultas>>

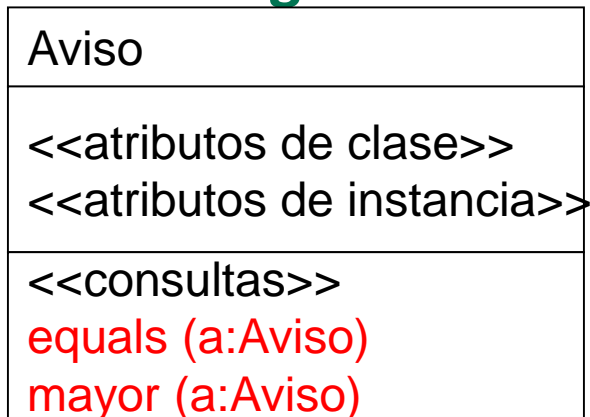
**costoAviso(): real**

## Caso de estudio: Agencia Publicitaria

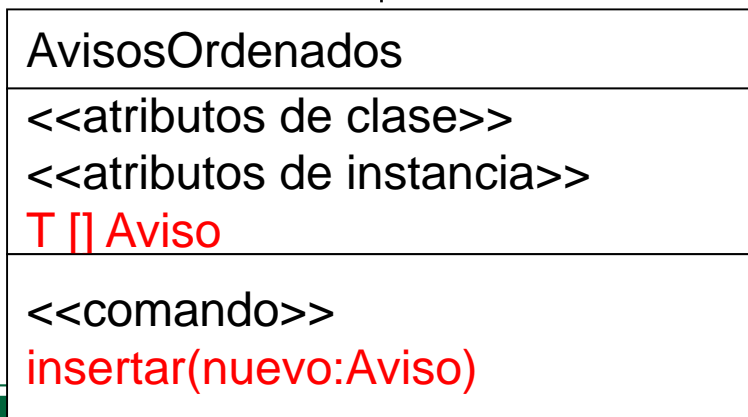
```
public double costoTotal () {  
    float c = 0;  
    for (int i=0;i< cantAvisos();i++)  
        c = c+T[i].costoAviso();  
  
    return c;  
}
```

El cambio en la especificación no afecta a las clases **Aviso**, **AvisoImpreso**, **AvisoRadioTV** ni **AvisosOrdenados**

## Caso de estudio: Agencia Publicitaria



Para decidir la equivalencia o establecer la relación mayor se compara empresa y luego nombre



## Caso de estudio: Agencia Publicitaria

- ▶ El diseñador estableció que la **clave** de un aviso es la combinación empresa-nombre.
- ▶ Dos avisos son iguales si coinciden los atributos nombre y empresa.
- ▶ Un aviso es mayor que otro si es mayor la empresa o las empresas son iguales y es mayor el nombre.



# Caso de estudio: Agencia Publicitaria

- ▶ Algoritmo insertar
- ▶ DE nuevo
- ▶ Buscar la posición de inserción
- ▶ Arrastrar los Avisos
- ▶ Asignar el nuevo Aviso en la posición de inserción
- ▶ Incrementar la cantidad de Avisos

El problema es diferente a los que hemos resuelto previamente, la solución es análoga a otras propuestas anteriores.

## Caso de estudio: Agencia Publicitaria

```
▶ public void insertar (Aviso nuevo){  
▶ /*Requiere que la colección no esté llena y Nuevo esté  
▶   ligada*/  
▶   int pos=posInsercion(nuevo,  
▶                           cantAvisos());  
▶   arrastrarDsp (pos,cantAvisos-pos-1);  
▶   T[pos] = nuevo;  
▶   cant++;  
▶ }
```

## Caso de estudio: Agencia Publicitaria

```
▶ private int  
▶     posInsercion (Aviso con,int n){  
▶     int pos = 0;  
▶     if (n > 0)  
▶         if (con.mayor (T[n-1]))  
▶             pos = n;  
▶         else  
▶             pos = posInsercion (con,--n);  
▶     return pos;  
▶ }
```

Mantenemos nuestro objetivo de obtener soluciones moduladas, aplicando la estrategia de dividir para conquistar en nuestros algoritmos.