

UD 6

## POLIMORFISMO

*Cristina Cachero, Pedro J. Ponce de León*

Versión 20111024



# Tema 4. Polimorfismo

## Objetivos básicos



- Comprender el concepto de polimorfismo
- Conocer y saber utilizar los diferentes tipos de polimorfismo.
- Comprender el concepto de enlazado estático y dinámico en los lenguajes OO.
- Comprender la relación entre polimorfismo y herencia en los lenguajes fuertemente tipados.
- Apreciar la manera en que el polimorfismo hace que los sistemas sean extensibles y mantenibles.



- 1. Motivación y conceptos previos
  - Signatura
  - Ámbito
  - Sistema de tipos
- 1. Polimorfismo y reutilización
  - Definición
  - Tipos de polimorfismo
- 1. Sobrecarga
  - Sobrecarga basada en ámbito
  - Sobrecarga basada en signatura de tipo
  - Alternativas a la sobrecarga
- 1. Polimorfismo en jerarquías de herencia
  - Redefinición
  - Shadowing
  - Sobrescritura
- 1. Variables polimórficas
  - La variable receptora
  - Downcasting
  - Polimorfismo puro
- 1. Genericidad
  - Métodos genéricos
  - Plantillas de clase
  - Herencia en clases genéricas

# 1. Motivación



- Objetivo de la POO
  - Aproximarse al modo de resolver problemas en el mundo real.
- El polimorfismo es el modo en que los lenguajes OO implementan el concepto de **polisemia** del mundo real:
  - Un único nombre para muchos significados, según el contexto.

# 1. Conceptos previos

## Signatura



- **Signatura de tipo** de un método:
  - Descripción de los tipos de sus argumentos, su orden y el tipo devuelto por el método.
    - Notación: ***<argumentos> → <tipo devuelto>***
    - Omitimos el nombre del método y de la clase a la que pertenece
  - Ejemplos
    - `double power (double base, int exp)`
      - ***double \* int → double***
    - `double distanciaA(Posicion p)`
      - ***Posicion → double***

# 1. Conceptos previos:

## Ámbito



- **Ámbito de un nombre:**

- Porción del programa en la cual un nombre puede ser utilizado de una determinada manera.

- Ejemplo:

```
double power (double base, int exp)
```

- La variable *base* sólo puede ser utilizada dentro del método *power*

- **Ámbitos activos:** puede haber varios simultáneamente

- Las clases, los cuerpos de métodos, cualquier bloque de código define un ámbito:

```
class A {  
    private int x,y;  
    public void f() {  
        // Ámbitos activos:  
        // GLOBAL  
        // CLASE (atribos. de clase y de instancia)  
        // METODO (argumentos, var. locales)  
        if (...) {  
            String s;  
            // ámbito LOCAL (var. locales)  
        }  
    }  
}
```

# 1. Conceptos previos:

## Ámbito: **Espacio de nombres**



- Un **espacio de nombres** es un ámbito con nombre
  - Agrupa declaraciones (clases, métodos, objetos...) que forman una unidad lógica.
  - Java: paquetes (package)

Circulo.java

```
package Graficos;
```

```
class Circulo {...}
```

Rectangulo.java

```
package Graficos;
```

```
class Rectangulo {...}
```

# 1. Conceptos previos:

## Ámbito: **Espacio de nombres**



- Un **espacio de nombres** es un ámbito con nombre
  - Agrupa declaraciones (clases, métodos, objetos...) que forman una unidad lógica.
- C++: namespace

### Graficos.h

(declaraciones agrupadas)

```
namespace Graficos {  
    class Circulo {...};  
    class Rectangulo {...};  
    class Lienzo {...};  
    ...  
}
```

### Circulo.h

(cada clase en su .h)

```
namespace Graficos {  
    class Circulo {...};  
}
```

### Rectangulo.h

```
namespace Graficos {  
    class Rectangulo {...};  
}
```



# 1. Conceptos previos:

## Ámbito: **Espacio de nombres**



- Java : instrucción **import**

```
class Main {  
    public static void main(String args[]) {  
        Graficos.Circulo c;  
        c.pintar(System.out);  
    }  
}
```

---

```
import Graficos.*;  
class Main {  
    public static void main(String args[]) {  
        Circulo c;  
        c.pintar(System.out);  
    }  
}
```

# 1. Conceptos previos:

## Ámbito: **Espacio de nombres**



- C++: cláusula **using**

```
#include "Graficos.h"
int main() {
    Graficos::Circulo c;
    Graficos::Rectangulo r;
    c.setRadio(4);
    double a = r.getArea();
}
```

---

```
#include "Graficos.h"
using Graficos::Circulo;
int main() {
    Circulo c;
    Graficos::Rectangulo r;
    c.setRadio(4);
    double a = r.getArea();
}
```

# 1. Conceptos previos:

## Ámbito: **Espacio de nombres**



- C++ : cláusula **using namespace**

```
#include "Graficos.h"
int main() {
    Graficos::Circulo c;
    Graficos::Rectangulo r;
    c.setRadio(4);
    double a = r.getArea();
}
```

```
#include "Graficos.h"
using namespace Graficos;
int main() {
    Circulo c;
    Rectangulo r;
    c.setRadio(4);
    double a = r.getArea();
}
```

# 1. Conceptos previos:

## Sistema de tipos



- Un sistema de tipos de un lenguaje asocia un tipo a cada expresión, con el objetivo de evitar errores en el código. Para ello proporciona

- Un mecanismo para definir tipos y asociarlos a las expresiones.

```
class A {} // definición de un tipo en Java/C++
A objeto; // 'objeto' es de tipo A
```

- Un conjunto de reglas para determinar la equivalencia o compatibilidad entre tipos.

```
String s = "una cadena";
int a = 10;
long b = 100;
a = s; // ERROR en Java/C++, los tipos 'String' e 'int' no son
       compatibles
b = a; // OK en Java/C++
```

# 1. Conceptos previos:

## Sistema de tipos



- Según sea el mecanismo que asocia (enlaza) tipos y expresiones, tendremos:
  - Sistema de tipos **estático**
    - El enlace se realiza en tiempo de compilación. Las variables tienen siempre asociado un tipo.  
`String s; // (Java/C++) 's' se define como una cadena.`
  - Sistema de tipos **dinámico**
    - El enlace se realiza en tiempo de ejecución. El tipo se asocia a los valores, no a las variables.  
`my $a; //(Perl) 'a' es una variable`  
`$a = 1; // 'a' hace referencia a un entero...`  
`$a = "POO"; // ... y ahora a una cadena`

# 1. Conceptos previos:

## Sistema de tipos



- Según las reglas de compatibilidad entre tipos, tendremos:
  - Sistema de tipos **fuerte**
    - Las reglas de conversión implícita entre tipos del lenguaje son muy estrictas:

```
int a=1;
bool b=true;
a=b; // ERROR
```
  - Sistema de tipos **débil**
    - El lenguaje permite la conversión implícita entre tipos

```
int a=1;
bool b=true;
a=b; // OK
```

Nota: 'fuerte' y 'débil' son términos relativos: un lenguaje puede tener un sistema de tipos más fuerte/débil que otro.

# 1. Conceptos previos:

## Sistema de tipos



- El **sistema de tipos** de un lenguaje determina su soporte al enlace dinámico:
  - **Lenguajes Procedimentales:** habitualmente tiene sistemas de tipos estáticos y fuertes y en general no soportan enlace dinámico: el *tipo* de toda expresión (identificador o fragmento de código) se conoce en tiempo de compilación.
    - C, Fortran, BASIC
  - **Lenguajes orientados a objetos:**
    - Con sistema de tipos estático (C++, Java, C#, Objective-C, Pascal...)
      - Sólo soportan enlace dinámico dentro de la *jerarquía de tipos* a la que pertenece una expresión (identificador o fragmento de código).
    - Con sistema de tipos dinámico (Javascript, PHP, Python, Ruby,...)
      - soportan enlace dinámico (obviamente)



1. Motivación y conceptos previos
  - Signatura
  - Ámbito
  - Sistema de tipos
1. Polimorfismo y reutilización
  - Definición
  - Tipos de polimorfismo
1. Sobrecarga
  - Sobrecarga basada en ámbito
  - Sobrecarga basada en signatura de tipo
  - Alternativas a la sobrecarga
1. Polimorfismo en jerarquías de herencia
  - Redefinición
  - Shadowing
  - Sobrescritura
1. Variables polimórficas
  - La variable receptora
  - Downcasting
  - Polimorfismo puro
1. Genericidad
  - Métodos genéricos
  - Plantillas de clase
  - Herencia en clases genéricas



# 2. Polimorfismo

## Definición



- **Capacidad de una entidad de referenciar distintos elementos en distintos instantes de tiempo.**
  
- Estudiaremos cuatro formas de polimorfismo, cada una de las cuales permite una forma distinta de **reutilización de software**:
  - Sobrecarga
  - Sobreescritura
  - Variables polimórficas
  - Genericidad



## ■ **Sobrecarga** (*Overloading*, Polimorfismo ad-hoc)

- Un sólo nombre de método y muchas implementaciones distintas.
- Las funciones sobrecargadas normalmente se distinguen en tiempo de compilación por tener distintos parámetros de entrada y/o salida.

`Factura.imprimir()`

`Factura.imprimir(int numCopias)`

`ListaCompra.imprimir()`

## ■ **Sobreescritura** (*Overriding*, Polimorfismo de inclusión)

- Tipo especial de sobrecarga que ocurre dentro de relaciones de herencia en métodos con enlace dinamico.
- Dichos métodos, definidos en clases base, son refinados o reemplazados en las clases derivadas.



- **Variables polimórficas** (Polimorfismo de asignación)

- Variable que se declara con un tipo pero que referencia en realidad un valor de un tipo distinto (normalmente relacionado mediante herencia).

```
Figura2D fig = new Circulo();
```

- **Genericidad** (plantillas o *templates*)

- Clases o métodos parametrizados (algunos elementos se dejan sin definir).
- Forma de crear herramientas de propósito general (clases, métodos) y especializarlas para situaciones específicas.

```
Lista<Cliente> clientes;  
Lista<Articulo> articulos;  
Lista<Alumno> alumnos;
```



- 1. Motivación y conceptos previos
  - Signatura
  - Ámbito
  - Sistema de tipos
- 1. Polimorfismo y reutilización
  - Definición
  - Tipos de polimorfismo
- 1. Sobrecarga
  - Sobrecarga basada en ámbito
  - Sobrecarga basada en signatura de tipo
  - Alternativas a la sobrecarga
- 1. Polimorfismo en jerarquías de herencia
  - Redefinición
  - Shadowing
  - Sobrescritura
- 1. Variables polimórficas
  - La variable receptora
  - Downcasting
  - Polimorfismo puro
- 1. Genericidad
  - Métodos genéricos
  - Plantillas de clase
  - Herencia en clases genéricas

### 3. Sobrecarga (*Overloading*, polimorfismo *ad-hoc*)

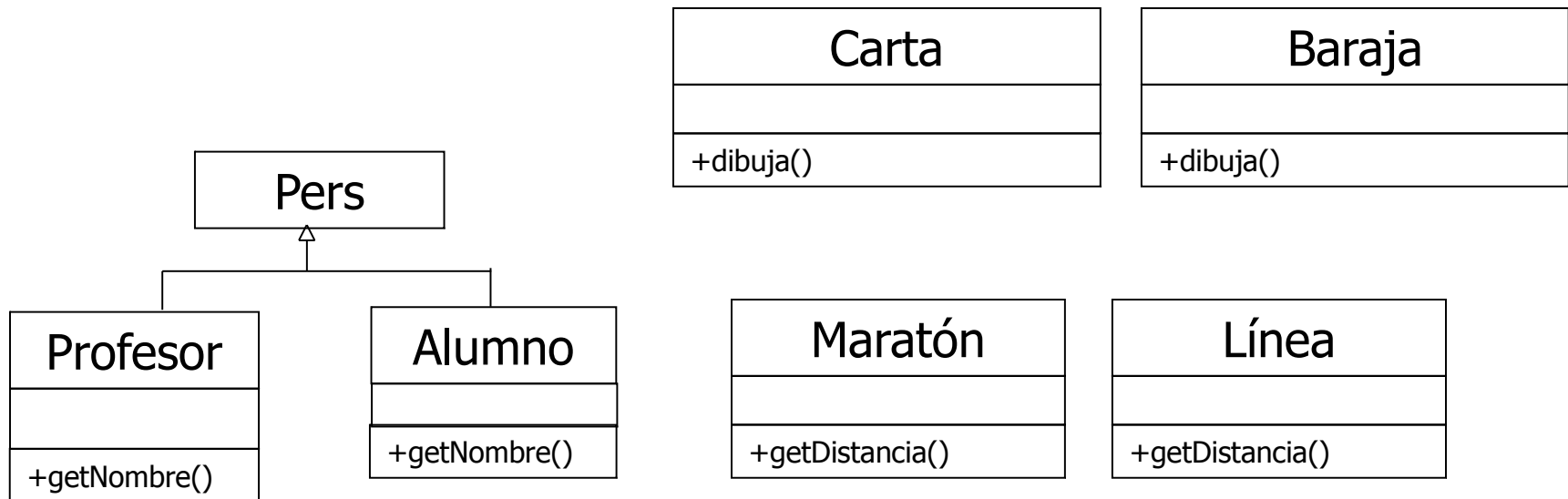


- Un mismo nombre de mensaje está asociado a varias implementaciones
- La sobrecarga se realiza en **tiempo de compilación** (enlace estático) en función de la signatura completa del mensaje.
- Dos tipos de sobrecarga:
  - **Basada en ámbito:** Métodos con **diferentes ámbitos de definición**, independientemente de sus signaturas de tipo.
    - P. ej. método toString() en Java.
  - **Basada en signatura:** Métodos con **diferentes signaturas de tipo en el mismo ámbito de definición**.

# Sobrecarga basada en **ámbito**



- Distintos ámbitos implican que el mismo nombre de método puede aparecer en ellos sin ambigüedad.
- La signatura de tipo puede ser la misma.



- ¿Son Profesor y Alumno ámbitos distintos?
- ¿Y Pers y Profesor?

# Sobrecarga basada en **signaturas de tipo**

- Métodos en el mismo ámbito pueden compartir el mismo nombre siempre que difieran en número, orden y tipo de los argumentos que requieren (el tipo devuelto no se tiene en cuenta).
- C++ y Java permiten esta sobrecarga de manera implícita siempre que la selección del método requerido por el usuario pueda establecerse de manera no ambigua en tiempo de compilación.

- Esto implica que la signatura no puede distinguirse sólo por el tipo de retorno

```
int f() {}  
string f() {}  
System.out.println( f() ); // ???
```

Suma
+add(int a) : int +add(int a, int b) : int +add(int a, double c) : double

# Sobrecarga basada en firmas de tipo



- Ejercicio: Si usamos sobrecarga basada en firma de tipos, y los métodos tienen enlace estático ¿qué ocurre cuando los tipos son diferentes pero relacionados por herencia?

```
class Base{...}
class Derivada extends Base {...}
class Cliente {
    public static void Test (Base b)
        {System.out.println("Base");}
    public static void Test (Derivada d) // sobrecarga
        {System.out.println("Derivada");}
    public static void main(String args[]){
        Base obj;
        if (...) obj = new Base();
        else obj = new Derivada(); //ppio de sustitución
        Test (obj); //¿a quién invoco?
    }
```





- **No todos los LOO permiten la sobrecarga:**
  - Permiten sobrecarga de métodos y operadores: C++
  - Permiten sobrecarga de métodos pero no de operadores: Java, Python, Perl
  - Permiten sobrecarga de operadores pero no de métodos: Eiffel



- Dentro de la sobrecarga basada en firmas de tipo, tiene especial relevancia la **sobrecarga de operadores**
- **Uso:** Utilizar operadores tradicionales con tipos definidos por el usuario.
- Forma de sobrecargar un operador @ en C++:  
`<tipo devuelto> operator@(<args>)`
- Para utilizar un operador con un objeto de tipo definido por usuario, éste debe ser sobrecargado.
  - Definidos por defecto: operador de asignación (=) y el operador de dirección (&)



- En la sobrecarga de operadores no se puede cambiar
  - Precedencia (qué operador se evalúa antes)
  - Asociatividad  $a=b=c \rightarrow a=(b=c)$
  - Aridad (operadores binarios para que actúen como unarios o viceversa)
- No se pueden crear nuevos operadores
- No se pueden sobrecargar operadores para tipos predefinidos.
- Algunos operadores no se pueden sobrecargar: `"."`, `"*"`, `"::"`, **`sizeof`**, `"? :"`



- La sobrecarga de operadores se puede realizar mediante funciones miembro o no miembro de la clase.
  - Como función miembro: el operando de la izquierda (en un operador binario) debe ser un objeto (o referencia a un objeto) de la clase.
    - Ejemplo: sobrecarga de + para la clase Complejo:

```
Complejo Complejo::operator+(const Complejo&)
```

```
...
```

```
Complejo c1(1,2), c2(2,-1);
```

```
c1+c2; // c1.operator+(c2);
```

```
c1+c2+c3; // c1.operator+(c2).operator+(c3)
```



- Como función no miembro:
  - Útil cuando el operando de la izquierda no es miembro de la clase

Ejemplo: sobrecarga de operadores << y >> para la clase Complejo:

```
ostream& operator<<(ostream&, const Complejo&);  
istream& operator>>(istream&, Complejo&);  
...  
Complejo c;  
cout << c; // operator<<(cout,c)  
cin >> c; // operator>>(cin,c)
```



### ■ **Funciones poliádicas**

- Funciones con número variable de argumentos
- Se encuentran en distintos lenguajes
  - P. ej. printf de C y C++

- Si el número máximo de argumentos es conocido, en C++ podemos acudir a la definición de valores por defecto:

```
int sum (int e1, int e2, int e3=0, int e4=0);
```



### ■ Métodos poliádicos en Java

```
void f(Object... args)
{   for (Object obj : args) {...} }

f("A", new A(), new Float(10.0));
f();

void g(int a, int... resto) {...}

g(3, "A", "B"); g(3);
```

- Los métodos poliádicos complican la sobrecarga. Deben usarse con precaución.



## ■ COERCIÓN

- Un valor de un tipo se convierte DE MANERA IMPLÍCITA en un valor de otro tipo distinto
  - P. ej. Coerción implícita entre reales y enteros en C++/Java.

```
double f(double x) {...}  
f(3); // coerción de entero a real
```

- El principio de sustitución en los LOO introduce además una forma de coerción que no se encuentra en los lenguajes convencionales
  - `// ppio. sustitución (coerción entre punteros)`  
`class B extends A {...}`  
`B pb = new B();`  
`A pa = pb;`





## ■ **CONVERSIÓN**

- Cambio en tipo de manera explícita
- Operador de conversión: se denomina CAST

- Ejemplo:

```
double x; int i;  
x= i + x; // COERCION  
x= (double)i + x; // CONVERSION
```

- Java permite
  - Conversión entre tipos escalares (excepto boolean)
  - Entre tipos relacionados por herencia (upcasting, downcasting)
  - Otro tipo de conversiones: mediante métodos específicos:
    - `Integer.valueOf("15.4");`

# Alternativas a sobrecarga: Coerción y Conversión



## ■ **CONVERSIÓN en C++**

- Definición de operación de conversión (*cast*) en C++:
  - De un tipo externo al tipo definido por la clase:
    - Constructor con un solo parámetro del tipo desde el cual queremos convertir.
  - Del tipo definido por la clase a otro tipo distinto:
    - Implementación de un operador de conversión.

```
class Fraccion{  
    private: int num, den;  
    public : operator double() {  
        return (numerador() / (double)denominador());  
    }  
};  
Fraccion f; double d = f * 3.14;
```



1. Motivación y conceptos previos
  - Signatura
  - Ámbito
  - Sistema de tipos
1. Polimorfismo y reutilización
  - Definición
  - Tipos de polimorfismo
1. Sobrecarga
  - Sobrecarga basada en ámbito
  - Sobrecarga basada en signatura de tipo
  - Alternativas a la sobrecarga
1. Sobrecarga en jerarquías de herencia
  - Redefinición
  - Shadowing
  - Sobrescritura
1. Variables polimórficas
  - La variable receptora
  - Downcasting
  - Polimorfismo puro
1. Genericidad
  - Métodos genéricos
  - Plantillas de clase
  - Herencia en clases genéricas



- TIEMPO DE ENLACE POR DEFECTO
  - JAVA
    - Enlace dinámico para métodos de instancia **públicos y protegidos.**
    - Enlace estático para métodos privados, de clase (estáticos) y atributos.
  - C++
    - Enlace estático para todas las propiedades (métodos de instancia, de clase y atributos).



- Modificación del tiempo de enlace por defecto
  - JAVA
    - Métodos de instancia con enlace estático: **No hay**
      - En realidad, un método declarado como final en la raíz de una jerarquía de herencia, se comporta como si tuviera enlace estático.
        - `public final void doIt() {...}`
  - C++
    - Métodos de instancia con enlace dinámico:
      - `virtual void doIt();`



- **Shadowing:** Métodos con el mismo nombre, la misma signature de tipo y enlace estático:
  - Refinamiento/reemplazo en clase derivada: las signatures de tipo son las mismas en clases base y derivadas. El método a invocar se decide en tiempo de compilación.
- **Redefinición:** Métodos con el mismo nombre y distinta signature de tipo y enlace estático:
  - La clase derivada define un método con el mismo nombre que en la base pero con **distinta signature de tipos en los argumentos**.



- Dos formas de resolver la **redefinición** en LOO:
  - Modelo ***merge*** (Java):
    - Los diferentes significados que se encuentran en todos los ámbitos actualmente activos se unen para formar una sola colección de métodos.
  - Modelo **jerárquico** (C++):
    - Una redefinición en clase derivada oculta el acceso directo a otras definiciones en la clase base:

```
class Padre{
    public void ejemplo(int a){System.out.println("Padre");}
}
class Hija extends Padre{
    public void ejemplo (int a, int b){System.out.println("Hija");}
}
```

```
Hija h;
h.ejemplo(3); // OK en Java
               //pero ERROR DE COMPILACIÓN EN C++
h.Padre::ejemplo(3); // OK (C++)
```

# Sobrecarga en jerarquías de herencia

## Sobreescritura



- Decimos que un método en una clase derivada sobrescribe un método en la clase base si los dos métodos tienen el mismo nombre, la misma signatura de tipos y enlace dinámico.
  - El método en la clase base tiene enlace dinámico.
  - Los métodos sobrescritos en clase derivada pueden suponer un reemplazo del comportamiento o un refinamiento del método base.
  - La resolución del método a invocar se produce en **tiempo de ejecución** (enlace dinámico) en función del tipo dinámico del receptor del mensaje.





- En Java, al tener los métodos de instancia enlace dinámico por defecto, su reimplementación en clases derivadas implica sobreescritura.
  - No obstante, se utiliza la anotación *@Override* en la clase derivada para indicar expresamente la decisión de sobrescribir un método de la clase base.
- En Java, podemos indicar que un método no puede ser sobrescrito, mediante la palabra clave 'final'



```
class Base { public void f() {} }  
class Derivada {  
    @Override  
    // Error de compilación si 'void f()' es privada, final, o  
    // escribimos mal su nombre o la lista de argumentos  
    public void f() {}  
}
```

# Sobrecarga en jerarquías de herencia

## Sobreescritura



- Java:

```
class Padre {  
    public int ejemplo(int a)  
        {System.out.println("padre");}  
    public final void f() {}  
}  
class Hija extends Padre {  
    @Override // anotación opcional recomendada  
    public int ejemplo (int a)  
        {System.out.println("hija");}  
    //public void f() { ... }  
    // ERROR, f() no se puede sobrescribir  
}
```

// código cliente

```
Padre p = new Hija(); //ppio. de sustitución  
p.ejemplo(10); // ejecuta Hija.ejemplo(10)
```

# Sobrecarga en jerarquías de herencia

## Sobreescritura: Covarianza



- Tipos de retorno covariantes: Al sobreescribir un método en clase derivada, podemos cambiar el tipo de retorno del método a un subtipo del especificado en la clase base:

```
class A {...}
class B extends A {...}
class Base {
    A objA = new A();
    public A getA() { return objA; } }
class Derivada {
    B objB = new B();
    @Override
    public B getA() { return objB; } }
```

```
Base b = new Derivada();
A objetoA = b.getA(); // Upcasting.
// objetoA apuntará a b.objB
```

# Sobrecarga en jerarquías de herencia

## Sobreescritura



En otros lenguajes:

**C++:** la clase base debe indicar que el método tiene enlace dinámico (y puede por tanto sobrescribirse).

**Smalltalk:** como en Java.

**Object Pascal:** la clase derivada debe indicar que sobreescrive un método: `procedure setAncho(Ancho: single); override;`

**C#, Delphi Pascal:** exigen que tanto la clase base como la derivada lo indiquen. Ej. C#:

En la base: `public virtual double Area() {...}`

En la derivada: `public override double Area() {...}`



- Es importante distinguir entre **Sobreescritura**, **Shadowing** y **Redefinición**
  - **Sobreescritura**: la signature de tipo para el mensaje es la misma en clase base y derivada, pero el método se enlaza con la llamada en función del tipo real del objeto receptor en tiempo de ejecución.
  - **Shadowing**: la signature de tipo para el mensaje es la misma en clase base y derivada, pero el método se enlaza en tiempo de compilación (en función del tipo declarado de la variable receptora).
  - **Redefinición**: La clase derivada define un método con el mismo nombre que en la clase base y con **distinta signature de tipos**.



1. Motivación y conceptos previos
  - Signatura
  - Ámbito
  - Sistema de tipos
1. Polimorfismo y reutilización
  - Definición
  - Tipos de polimorfismo
1. Sobrecarga
  - Sobrecarga basada en ámbito
  - Sobrecarga basada en signatura de tipo
  - Alternativas a la sobrecarga
1. Polimorfismo en jerarquías de herencia
  - Redefinición
  - Shadowing
  - Sobrescritura
1. Variables polimórficas
  - La variable receptora
  - Downcasting y RTTI
  - Polimorfismo puro
1. Genericidad
  - Métodos genéricos
  - Plantillas de clase
  - Herencia en clases genéricas



- Una variable polimórfica es aquélla que puede referenciar más de un tipo de objeto
  - Puede mantener valores de distintos tipos en distintos momentos de ejecución del programa.
- En un lenguaje con sistema de tipos dinámico todas las variables son potencialmente polimórficas
- En un lenguaje con sistema de tipos estático la variable polimórfica es la materialización del principio de sustitución.
  - En Java: las referencias a objetos.
  - En C++: punteros o referencias a clases polimórficas



## Clase polimórfica

- En Java, por defecto todas las clases son polimórficas.
- En C++, clase con al menos un método virtual.
- Podemos indicar que no se pueden crear clases derivadas con final:
  - **final** class ClaseNoDerivable { ... }

El efecto es que las referencias de tipo *ClaseNoDerivable* ya no son polimórficas: sólo pueden referenciar objetos de tipo *ClaseNoDerivable*.





- **Variables polimórficas simples**

- `Figura2D img; // Puntero a clase base polimórfica que en realidad apuntará a  
// objetos de clases derivadas (Circulos, Cuadrados,...)`

- **Variables receptoras: `this` y `super`**

- En un método, hacen referencia al receptor del mensaje.
- En cada clase representan un objeto de un tipo distinto.

(en otros lenguajes recibe otros nombres, como `'self'`)



- **Downcasting** (polimorfismo inverso):

- Conversión de una referencia a clase base a referencia a clase derivada.
- Implica 'deshacer' el ppio. de sustitución.
- Tipos
  - **Estático** (en tiempo de compilación)
  - **Dinámico** (en tiempo de ejecución)

C++ soporta ambos tipos  
En Java es siempre dinámico.



## ■ Downcasting dinámico

- Se comprueba en tiempo de ejecución que la conversión es posible
- En Java, sólo permitido dentro de jerarquías de herencia
- Si no es posible se lanza *ClassCastException*

```
class Base {  
    public void f() {}  
}
```

```
class Derivada extends Base {  
    public void f() {}  
    public void g() {}  
}
```

```
// Downcasting no seguro  
Base[] x = { new Base(), new Derivada() };  
Derivada y = (Derivada)x[1]; // Downcasting OK  
y = (Derivada)x[0]; // ClassCastException thrown  
y.g();
```



- **Downcasting seguro y RTTI**
  - **RTTI: Run Time Type Information**
    - Mecanismo que proporciona información sobre tipos en tiempo de ejecución
    - Permite averiguar y utilizar información acerca de los tipos de los objetos mientras el programa se está ejecutando.
    - En particular, podemos identificar subtipos a partir de referencias al tipo base: downcasting seguro



## ■ RTTI: La clase Class

- Es una metaclass cuyas instancias representan clases
- Cada clase tiene asociado un objeto Class

```
class MiClase {}
```

```
MiClase c = new MiClase();
```

```
Class clase = MiClase.class; // literal de clase
```

```
clase = c.getClass(); // idem
```

**Literal de clase:** es el objeto Class que representa a MiClase



## ■ RTTI: La clase Class

```
class Animal {}  
class Perro extends Animal {  
    public void ladrar() {}  
}  
class PastorBelga extends Perro {}
```

```
// Downcasting seguro  
Animal a = new PastorBelga();  
if (a.getClass() == PastorBelga.class) // cierto  
{ PastorBelga pb = (PastorBelga)a; }  
if (a.getClass() == Perro.class) // falso  
{ Perro p = (Perro)a; }
```



- **RTTI: instanceof**
  - **Instrucción que devuelve cierto si el objeto referenciado es del tipo indicado**

```
class Animal {}  
class Perro extends Animal {  
    public void ladrar() {}  
}  
class PastorBelga extends Perro {}  
  
// Downcasting seguro  
Animal a = new PastorBelga();  
if (a instanceof PastorBelga) // cierto  
{ PastorBelga pb = (PastorBelga)a; }  
if (a instanceof Perro) // cierto  
{ Perro p = (Perro)a; }
```



- **RTTI:**

- **Class.isInstance(): instanceof dinámico**

- instanceof necesita conocer el nombre de la clase objetivo en tiempo de compilación
    - ¿y si no lo conozco?

```
// Downcasting seguro
Animal a = new Perro();
Animal b = new PastorBelga();

Class clasePerro = a.getClass();
if (clasePerro.isInstance(b)) { //cierto
    Perro p = (Perro)b; // seguro
    p.ladrar();
}
```



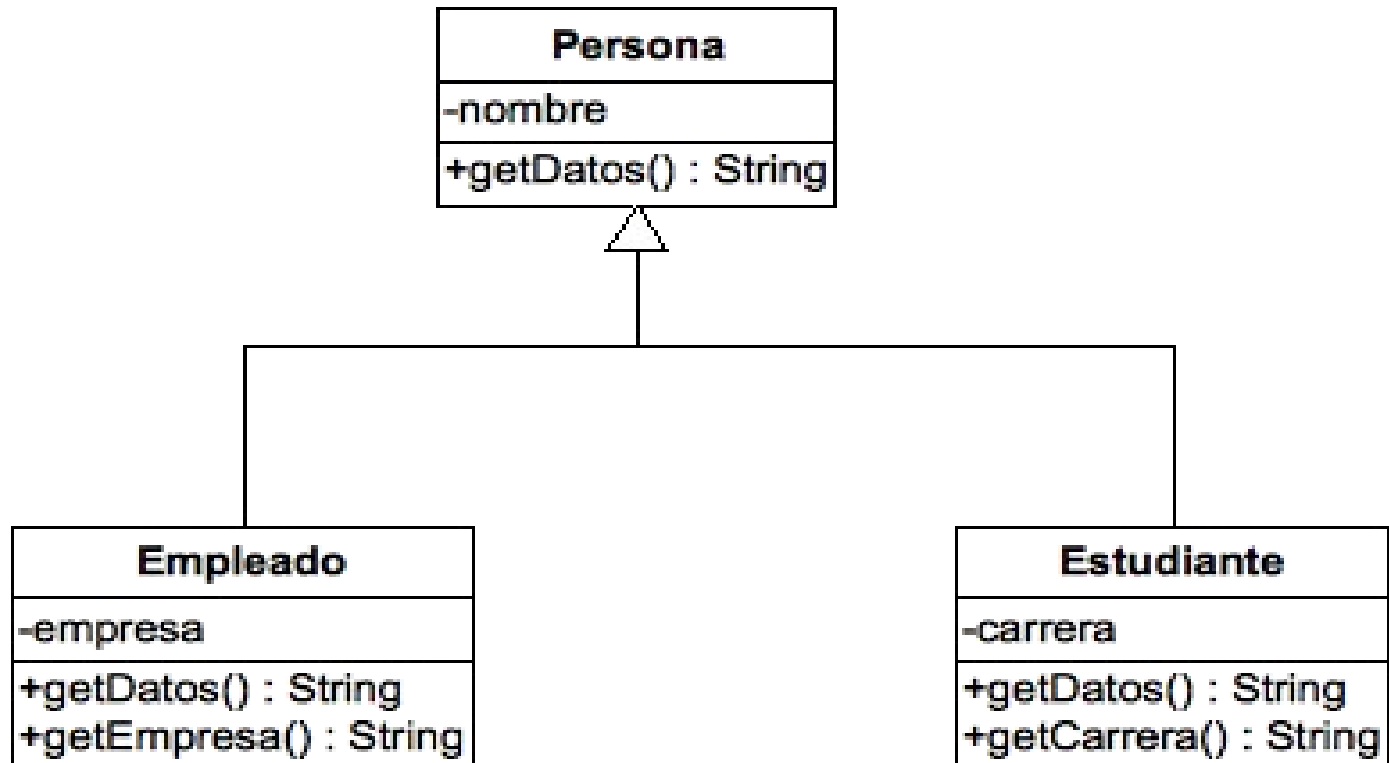


## ■ Método con **polimorfismo puro** o **método polimórfico**

- Alguno de sus argumentos es una variable polimórfica:
  - Un solo método puede ser utilizado con un número potencialmente ilimitado de tipos distintos de argumento.

### ■ Ejemplo de polimorfismo puro

```
class Base { ... }  
class Derivada1 extends Base { ... }  
class Derivada2 extends Base { ... }  
  
void f(Base obj) { // Método polimórfico  
    // Aquí puedo usar sólo la interfaz de Base para manipular obj  
    // Pero obj puede ser de tipo Base, Derivada1, Derivada2,...  
}  
  
public static void main(String args[]) {  
    Derivada1 objeto = new Derivada1();  
    f(objeto); // OK  
}
```



# Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Persona {  
    public Persona(String n)      {nombre=n;}  
    public String getDatos() {return nombre;}  
    ...  
    private String nombre;  
}
```

# Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Empleado extends Persona {  
    public Empleado(String n,String e)  
    {    super(n); empresa=e }  
    public String getDatos()  
    { return super.getDatos()+"trabaja en " + empresa; }  
    ...  
    private String empresa;  
}  
class Estudiante extends Persona {  
    public Estudiante(String n,String c)  
    {    super(n); carrera=c }  
    public String getDatos()  
    { return super.getDatos() + " estudia " + carrera; }  
    ...  
    private String carrera;  
}
```

Refinamiento

Refinamiento

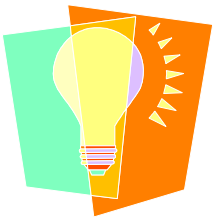
# Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
// código cliente
```

```
Empleado empleado =  
    new Empleado("Carlos", "Lavandería");  
Persona pers =  
    new Persona("Juan");
```

```
empleado = pers;  
System.out.println( empleado.getDatos() );
```



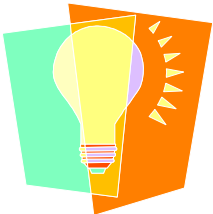
**¿Qué ocurre?**

# Ejemplo: Uso de polimorfismo y jerarquía de tipos



//código cliente

```
Empleado emp = new Empleado("Carlos", "lavanderia");  
Estudiante est = new Estudiante("Juan","Derecho");  
Persona pers;  
pers = emp;  
System.out.println( emp.getDatos() );  
System.out.println( est.getDatos() );  
System.out.println( pers.getDatos() );
```



**¿Qué salida dará este programa?**  
**¿Se produce un enlazado estático o dinámico?**

# Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Persona {  
    public Persona(String n)      {nombre=n;}  
    public final String getDatos() {  
        return (nombre);  
    }  
    ...  
    private String nombre;  
}
```

No se puede  
sobreescribir

# Ejemplo: Uso de polimorfismo y jerarquía de tipos



//código cliente

```
Empleado emp = new Empleado("Carlos", "lavanderia");  
Estudiante est = new Estudiante("Juan","Derecho");  
Persona pers;  
pers = emp;  
System.out.println( emp.getDatos() );  
System.out.println( est.getDatos() );  
System.out.println( pers.getDatos() );
```



**¿Qué salida dará este programa?**  
**¿Se produce un enlazado estático o dinámico?**



# Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
// código cliente 1
```

```
Empleado uno= new Empleado("Carlos", "lavanderia");  
Persona desc = uno;  
System.out.println( desc.getEmpresa() );
```

```
// código cliente 2
```

```
Persona desc = new Persona("Carlos");  
Empleado emp = (Empleado) desc;  
System.out.println( emp.getEmpresa() );
```



**¿Qué sucede en ambos casos?**

# Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
// código cliente 2
Persona desc = new Persona("Carlos");
if (desc instanceof Empleado) {
    Empleado emp = (Empleado)desc;
    System.out.println( emp.getEmpresa() );
}
```



**¿Qué sucede en ambos casos?**  
**¿Se produce un enlazado estático o dinámico?**



- Los métodos con enlace dinámico son algo menos eficientes que las funciones normales.
  - Cada clase no abstracta en Java dispone de un vector de punteros a métodos llamado *Tabla de métodos*. Cada puntero corresponde a un método de instancia con enlace dinámico, y apunta a su implementación más conveniente (la de la propia clase o, en caso de no existir, la del ancestro más cercano que la tenga definida)
  - Cada objeto de la clase tiene un puntero oculto a esa *tabla de métodos*.



- El polimorfismo hace posible que un usuario pueda añadir nuevas clases a una jerarquía sin modificar o recompilar el código escrito en términos de la clase base.
- Permite programar a nivel de clase base utilizando objetos de clases derivadas (posiblemente no definidas aún): Técnica base de las librerías/frameworks (UD 8)



- 1. Motivación y conceptos previos
  - Signatura
  - Ámbito
  - Sistema de tipos
- 1. Polimorfismo y reutilización
  - Definición
  - Tipos de polimorfismo
- 1. Sobrecarga
  - Sobrecarga basada en ámbito
  - Sobrecarga basada en signatura de tipo
  - Alternativas a la sobrecarga
- 1. Polimorfismo en jerarquías de herencia
  - Redefinición
  - Shadowing
  - Sobrescritura
- 1. Variables polimórficas
  - La variable receptora
  - Downcasting
  - Polimorfismo puro
- 1. Genericidad
  - Métodos genéricos
  - Plantillas de clase
  - Herencia en clases genéricas



- La genericidad es otro tipo de polimorfismo
- Para ilustrar la idea de la genericidad se propone un ejemplo:
  - Suponed que queremos implementar una función *máximo*, donde los parámetros pueden ser de distinto tipo



- Solución: usar interfaces

```
interface Comparable { boolean mayorQue(Object); }  
class A implements Comparable { ... }  
class B implements Comparable { ... }
```

```
Comparable maximo(Comparable a, Comparable b) {  
    if (a.mayorQue(b))  
        return a;  
    else  
        return b;  
}
```

```
A a1 = new A(), a2 = new A();  
B b1 = new B(), b2 = new B();  
A mayorA = maximo (a1,a2);  
B mayorB = maximo (b1,b2);
```



- Maximo() está restringido a clases que implementen el interface Comparable.
- ¿Y si queremos algo todavía más general, que funcione con cualquier tipo de objeto? Por ejemplo, una clase Lista que pueda contener cualquier tipo de dato, ya sean tipos primitivos u objetos.
  - Necesitaríamos una función genérica





- *Propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o algunos de sus miembros o argumentos.*
  - Su utilidad principal es la de agrupar variables cuyo tipo base no está predeterminado (p. ej., listas, colas, pilas etc. de objetos genéricos: Java Collection Framework).
  - Es el usuario el que indica el tipo de la variable cuando crea un objeto de esa clase.
  - En C++ esta característica apareció a finales de los 80. En Java, existe desde la versión 1.5.



- Dos tipos de genéricos:
  - **Métodos genéricos**: son útiles para implementar funciones que aceptan argumentos de tipo arbitrario.
  - **Clases genéricas**: su utilidad principal consiste en agrupar variables cuyo tipo no está predeterminado (*clases contenedoras*)



### Un argumento genérico

```
public <T> void imprimeDos(T a, T b)
{
    System.out.println(
        "Primero: " + a.toString() +
        " y Segundo:" + b.toString() );
}
```

```
Cuenta a = new Cuenta(),
Cuenta b = new Cuenta();
imprimeDos(a,b);
```

Inferencia de tipo de los argumentos: la realiza el compilador

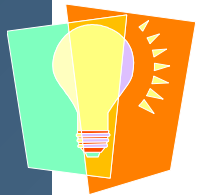




### Más de un argumento genérico

```
public <T,U> void imprimeDos(T a, U b)
{
    System.out.println(
        "Primero: " + a.toString() +
        " y Segundo:" + b.toString() );
}
```

```
Cuenta c = new Cuenta(),
Perro p = new Perro();
imprimeDos(c,p);
```



# Genericidad

## Clases Genéricas en Java



- A continuación se plantea un ejemplo de una clase genérica *Cajon*, que va a contener un elemento de tipo genérico, no se conoce a priori.



```
class Cajon<T> { // un argumento genérico: T

    private T cajon;

    public Cajon(T obj)
    { cajon = obj; }

    T get()
    { return cajon; }

}
```



### ■ Creación de objeto:

```
Cajon<int> ci = new Cajon<int>(10);  
Cajon<Animal> ca =  
    new Cajon<Animal>(new Perro());
```

Hay que indicar el tipo de objeto al instanciar la clase.

En el caso de 'ca', podemos almacenar ahí cualquier Animal o subtipo de Animal.



- Se pueden **derivar clases genéricas** de otras clases genéricas:

### Clase derivada genérica:

```
class DoblePila<T> extends Pila<T>
{
    public void apilar2(T a, T b) {...}
}
```

- La clase doblePila es a su vez genérica:

```
DoblePila<float> dp = new DoblePila(10);
```





- Se pueden **derivar clases no genéricas** de una genérica:

### Clase derivada no genérica:

```
class monton extends Pila<int>
{
    public void amontonar(int a) {...}
}
```

- 'monton' es una clase normal. No tiene ningún parámetro genérico.



- En Java, no existe relación alguna entre dos clases generadas desde la misma clase genérica, aunque los tipos estén relacionados por herencia:

```
class Uno {}  
class Dos extends Uno {}
```

```
ArrayList<Uno> u = new ArrayList<Uno>();  
ArrayList<Dos> d = new ArrayList<Dos>();
```

```
u = d; // Error: incompatible types
```



- Sin embargo,

```
ArrayList<Integer> v = new ArrayList<Integer>();  
ArrayList<String> w = new ArrayList<String>();  
System.out.println(  
    v.getClass() == w.getClass() );  
// imprime 'true'  
//v = w; // Error: incompatible types
```

**Borrado de tipos:** Java no guarda información RTTI sobre tipos genéricos. En tiempo de ejecución, sólo podemos asumir que los parámetros genéricos son de tipo Object.

En tiempo de compilación, obviamente, sí existe dicha información.

# Genericidad

## Interfaces genéricas



- Las interfaces también pueden ser genéricas

```
interface Factoria<T>  
{ T newObject(); }
```

```
class FactoriaDeAnimales implements Factoria<Animal>  
{  
  
    Animal newObject() {  
        if (...) return new Perro();  
        else if (...) return new Gato();  
        else return new ProgramadorDeJava();  
    }  
}
```



- *El problema con la genericidad es que sólo podemos utilizar aquellos métodos que estén definidos en Object.*
- *La genericidad restringida permite indicar que los tipos genéricos pertenezcan a una determinada jerarquía de herencia*
  - *Esto permite utilizar la interfaz de la clase usada como raíz de la jerarquía en los métodos/clases genéricos.*

```
class Perrera<T extends Perro> {  
    public void acoger(T p) { jaula.add(p); }  
    public void alimentar() {  
        for (T p : jaula)  
            if (p.ladrear()) p.alimentar();  
    }  
    private ArrayList<T> jaula = new ArrayList<T>();  
}
```



```
public class NonCovariantGenerics {  
    // Compile Error: incompatible types:  
    List<Fruit> flist = new ArrayList<Apple>();  
} ///:~
```

- Una lista de manzanas NO ES una lista de frutas. Las listas de manzanas no pueden contener cualquier tipo de fruta.
- **Comodines de subtipo**

```
List<? extends Fruit> flist =  
    new ArrayList<Apple>();  
flist.add(new Apple()); // ERROR  
flist.get(0); // retorna un fruit
```



- ***Comodines de tipo base***

```
List<? super Apple> flist =  
    New ArrayList<Apple>();  
flist.add(new Apple()); // OK  
flist.add(new Fruit()); // ERROR  
flist.get(0); // retorna un Apple
```

# Tema 4. Polimorfismo

## Bibliografía



- Cachero et. al.
  - ***Introducción a la programación orientada a Objetos***
    - Capítulo 4
- T. Budd.
  - ***An Introduction to Object-oriented Programming, 3rd ed.***
    - Cap. 11,12,14-18; cap. 9: caso de estudio en C#
- Bruce Eckl
  - ***Piensa en Java***, 4ª edición.
    - Cap. 8, 14 y 15