

Polimorfismo y Ligadura Dinámica

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
 - ▶ Emojis, <https://pixabay.com/images/id-2074153/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Entender los conceptos polimorfismo y ligadura dinámica
- Saber usar dichos mecanismos
- Saber detectar situaciones en las que es procedente el uso de dichos mecanismos
- Saber realizar diseños para dar solución a dichas situaciones

Contenidos

1 Introducción

2 Polimorfismo

3 Ligadura dinámica

- Casts
- Comprobaciones explícitas de tipos
- Detalles adicionales

Introducción

Java: Introducción a polimorfismo y ligadura dinamica

```

1 class Empleado { // entre otras cosas ...
2     float calculaSueldo() { return . . . }
3 }
4 class Comercial extends Empleado { // entre otras cosas ...
5     float calculaSueldo() {
6         return super.calculaSueldo() + . . .
7     }
8 }
9 class Directivo extends Empleado { // entre otras cosas ...
10    float calculaSueldo() {
11        return super.calculaSueldo() + . . .
12    }
13 }
14
15 // En algún otro sitio ...
16 ArrayList<Empleado> listaDeEmpleados = new ArrayList<>();
17 listaDeEmpleados.add (new Empleado (. . .));
18 listaDeEmpleados.add (new Comercial (. . .));
19 listaDeEmpleados.add (new Directivo (. . .));
20
21 // Se quiere calcular el total de todos los sueldos
22 float sueldosTotales = 0.0f;
23 for (Empleado unEmpleado : listaDeEmpleados) {
24     sueldosTotales += unEmpleado.calculaSueldo();
25 }

```

Polimorfismo

- Capacidad de un identificador de **referenciar objetos de diferentes tipos** (clases)
 - ▶ En lenguajes sin declaración de variables se da de forma natural y sin limitaciones
 - ▶ Ruby no utiliza el mecanismo de declaración de variables. Cualquier variable puede referenciar cualquier tipo de objeto
 - ▶ En lenguajes con declaración de variables con un tipo específico existen limitaciones al respecto
- **Principio de sustitución de Liskov:**
 - ▶ Si B es un subtipo de A, se pueden utilizar instancias de B donde se esperan instancias de A
 - ▶ Por ejemplo:
 - ★ Si `Director` es subclase de `Persona` se puede usar una instancia de `Director` donde se puedan usar instancias de `Persona`.
 - ★ Recordar la relación **es-un**:
`Director es-una Persona` (a todos los efectos)

Tipo estático y dinámico

- **Tipo estático**: tipo (clase) del que se declara la variable
- **Tipo dinámico**: clase al que pertenece, **en un momento determinado**, el objeto referenciado por una variable

Java: Tipo estático y dinámico

```

1 ArrayList<Empleado> listaDeEmpleados = new ArrayList<>();
2 listaDeEmpleados.add (new Empleado ( . . . ));
3 listaDeEmpleados.add (new Comercial ( . . . ));
4 listaDeEmpleados.add (new Directivo ( . . . ));
5
6 // Se quiere calcular el total de todos los sueldos
7 float sueldosTotales = 0.0f;
8 for (Empleado unEmpleado : listaDeEmpleados) {
9     sueldosTotales += unEmpleado.calculaSueldo();
10 }

```

★ ¿Cuál es el tipo estático de unEmpleado?

★ ¿Y su tipo dinámico?

★ ¿Se puede saber con solo mirar el código?

Ligadura dinámica

- **Ligadura estática:**

El enlace del código a ejecutar asociado a una llamada a un método se hace en tiempo de compilación (permitida en C++)

- **Ligadura dinámica:**

El tipo dinámico determina el código que se ejecutará asociado a la llamada de un método

- ▶ Hace que cobre sentido el polimorfismo

Java: Ligadura dinámica

```

1 class Empleado { // entre otras cosas ...
2   float calculaSueldo() { return . . . }
3 }
4 class Comercial extends Empleado { // entre otras cosas ...
5   float calculaSueldo() { return super.calculaSueldo() + . . . }
6 }
7 class Directivo extends Empleado { // entre otras cosas ...
8   float calculaSueldo() { return super.calculaSueldo() + . . . }
9 }
10
11 // En algún otro sitio . . .
12 for (Empleado unEmpleado : listaDeEmpleados) {
13   sueldosTotales += unEmpleado.calculaSueldo();
14 }

```

// ¿Qué implementación de calculaSueldo se va a ejecutar?

Ejemplo

Java: Ejemplo de polimorfismo y ligadura dinámica

```
1 class Persona {
2     public String andar() {
3         return ("Ando como una persona");
4     }
5
6     public String hablar() {
7         return ("Hablo como una persona");
8     }
9 }
10
11 class Profesor extends Persona{
12     @Override
13     public String hablar() {
14         return ("Hablo como un profesor");
15     }
16 }
17 // *****
18
19 public static void main(String[] args) {
20     Persona p=new Persona();
21     Persona p2=new Profesor(); // Puede también referenciar un Profesor
22
23     p.hablar(); // "Hablo como una persona"
24     p2.hablar(); // "Hablo como un profesor"
25 }
```

Reglas

- El tipo estático limita:
 - ▶ Lo que puede referenciar una variable
 - ★ Instancias de la clase del tipo estático o de sus subclases
 - ▶ Los métodos que pueden ser invocados
 - ★ Los disponibles en las instancias de la clase del tipo estático

Java: Ejemplo

```

1 class Profesor extends Persona{
2
3     public String impartirClase() {
4         // Este método no lo tiene Persona ni ninguna de sus superclases
5         return ("Impartiendo clase");
6     }
7 }
8 // *****
9
10 public static void main(String[] args) {
11     Persona p=new Profesor();
12
13     // Error de compilación. Las personas no tienen ese método
14     p.impartirClase();
15
16     //Error de compilación. Object no es subclase de Persona
17     p=new Object();
18 }

```

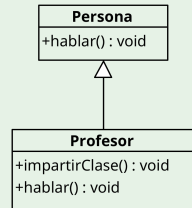
Casts

- Se le indica al compilador que considere, **temporalmente**, que el tipo de una variable es otro
 - ▶ Solo para la instrucción en la que aparece y **con limitaciones**
- **Downcasting**:
 - ▶ Se indica al compilador que considere, temporalmente, que el tipo de la variable es una subclase del tipo con que se declaró
 - ▶ Permite invocar métodos que sí existen en el tipo del cast pero que no están en el tipo estático de la variable
- **Upcasting**:
 - ▶ Se indica al compilador que considere, temporalmente, que el tipo de la variable es superclase del tipo con que se declaró
 - ▶ Normalmente es innecesario y redundante
- **Importante**:
 - ▶ Las operaciones de casting no realizan ninguna transformación en el objeto referenciado
 - ▶ Tampoco cambian el comportamiento del objeto referenciado

Ejemplo

Java: Ejemplo de casts

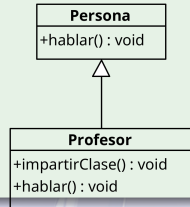
```
1 public static void main(String[] args) {  
2     Persona p = new Profesor(); // El objeto es un Profesor  
3                                     // y siempre lo será, a pesar de los casts  
4  
5     // Error de compilación. Las personas no tienen ese método  
6     p.impartirClase();  
7  
8     // Error de compilación. En general una Persona no es un Profesor  
9     Profesor prof = p;  
10  
11     ((Profesor) p).impartirClase();  
12     Profesor profe = (Profesor) p;  
13  
14     profe.hablar(); // "Hablo como un profesor"  
15  
16     // Upcast innecesario y sin efectos  
17     ((Persona) profe).hablar(); // "Hablo como un profesor"  
18  
19     // Upcast implícito y sin efectos  
20     Persona p2 = profe;  
21     p2.hablar(); // "Hablo como un profesor"  
22 }
```



Ejemplo

Java: Ejemplo de casts con errores de ejecución

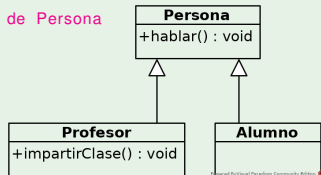
```
1 public static void main(String[] args) {  
2  
3     // Errores en tiempo de ejecución  
4     // java.lang.ClassCastException: Persona cannot be cast to Profesor  
5  
6     Persona p = new Persona();  
7     Profesor profe = (Profesor) p;    // Error  
8  
9     profe = ((Profesor) new Persona());    // Error  
10  
11     ((Profesor) p).impartirClase(); // Error  
12  
13     ((Profesor) ((Object) new Profesor())).impartirClase(); // OK  
14  
15 }
```



Ejemplo

Java: Ejemplo de casts entre clases “hermanas”

```
1 class Alumno extends Persona {  
2     // Clase "hermana" de Profesor  
3     // Alumno y Profesor son descendientes directos de Persona  
4 }  
5  
6 public static void main(String[] args) {  
7  
8     // Error de compilación. Tipos incompatibles  
9     Alumno a1 = new Profesor();  
10  
11    // Error de compilación. Tipos incompatibles  
12    Alumno a2 = (Alumno) new Profesor();  
13  
14    // Error en tiempo de ejecución  
15    // java.lang.ClassCastException: Profesor cannot be cast to Alumno  
16    Alumno a3 = ((Alumno) ((Persona) new Profesor()));  
17  
18 }
```



Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

Java: Mal ejemplo

```
1 public static void main(String[] args) {
2
3     Empleado e;
4     Random r = new Random();
5
6     if (r.nextBoolean()) {
7         e = new Comercial ("Pepe");
8     } else {
9         e = new Directivo ("Pepe");
10    }
11
12    // Nada recomendable
13    // Mal diseño
14    // No lo hagáis
15    // Lo digo en serio, no hagáis este tipo de diseños
16    String s;
17    if (e instanceof Empleado) {
18        s = "Soy " + e.getNombre();
19    } if (e instanceof Directivo) {
20        s = "Soy " + e.getNombre() + ", soy directivo";
21    } else if (e instanceof Comercial) {
22        s = "Soy " + e.getNombre() + ", soy comercial";
23    } else s = "";
24    System.out.println (s);
25 }
```

Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

Java: Forma correcta de proceder

```
1 class Empleado {
2     private String nombre;
3     public Persona(String n) { nombre=n; }
4     public String getNombre() { return nombre; }
5     public String presentacion () { return ("Soy " + nombre); }
6 }
7
8 class Comercial extends Empleado {
9     public Comercial (String n) { super(n); }
10    @Override
11    public String presentacion () { return (super.presentacion() + ", soy comercial"); }
12 }
13
14 class Directivo extends Empleado {
15     public Directivo (String n) { super(n); }
16    @Override
17    public String presentacion () { return (super.presentacion() + ", soy directivo"); }
18 }
```


Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

Java: Forma correcta de proceder

```
1 public static void main(String[] args) {  
2  
3     Empleado e;  
4     Random r = new Random();  
5  
6     if (r.nextBoolean()) {  
7         e = new Comercial ("Pepe");  
8     } else {  
9         e = new Directivo ("Pepe");  
10    }  
11  
12    // Tenemos el comportamiento correcto de forma automática  
13  
14    System.out.println (e.presentacion());  
15 }
```

Detalles adicionales

- Con ligadura dinámica, **siempre se comienza buscando** el código asociado al método invocado en la clase que coincide **con el tipo dinámico** de la referencia
- Si no se encuentra se busca en la clase padre
- Así sucesivamente hasta encontrarlo o hasta que no existan ascendientes
- Esto sigue siendo cierto para métodos invocados desde otros métodos

Detalles adicionales

Ruby: Búsqueda del método a ejecutar

```
1 class Padre
2
3   def interno
4     puts "Interno padre"
5   end
6
7   def metodo
8     puts "Voy a actuar: "
9     interno
10  end
11 end
12
13
14 class Hija < Padre
15
16   def interno
17     puts "Interno hijo"
18   end
19
20 end
21
22 Padre.new.metodo # Voy a actuar: Interno padre
23 Hija.new.metodo  # Voy a actuar: Interno hijo
```

Polimorfismo y ligadura dinámica



- Estos mecanismos permiten crear diseños y codificaciones claros y fácilmente mantenibles
 - ▶ Volver a comparar las líneas 16 a 24 de la transparencia 15 con la línea 14 de la transparencia 17
(¡y solo hay involucradas 3 clases!)
- Deben tenerse en cuenta cuando:
 - ▶ Varias clases tienen el mismo método pero con distintas implementaciones
 - ▶ Existe relación de herencia entre dichas clases
 - ▶ No se sabe, a priori, a qué objeto concreto se le va a enviar el mensaje asociado a dicho método

Polimorfismo y Ligadura Dinámica

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)