

# **Polimorfismo y Ligadura Dinámica**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

**Programación y Diseño Orientado a Objetos**

(Curso 2019-2020)

# 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

- ¿Recordáis el ejemplo de las figuras geométricas?

## Java: Introducción a polimorfismo y ligadura dinamica

```

1 class FiguraGeometrica { // entre otras cosas ...
2     float area() { return 0.0f; }
3 }
4 class Circulo extends FiguraGeometrica { // entre otras cosas ...
5     float area() {
6         return Math.PI * radio * radio;
7     }
8 }
9 class Rectangulo extends FiguraGeometrica { // entre otras cosas ...
10    float area() {
11        return lado1*lado2;
12    }
13 }
14
15 // En algún otro sitio ...
16 ArrayList<FiguraGeometrica> coleccionDeFiguras = new ArrayList<>();
17 coleccionDeFiguras.add (new Circulo (radio));
18 coleccionDeFiguras.add (new Rectangulo (lado1, lado2));
19
20 float suma = 0.0f;
21 for (FiguraGeometrica unaFigura : coleccionDeFiguras) {
22     suma += unaFigura.area();
23 }

```

# 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 el objeto referenciado en un momento determinado por una variable

## Java: Tipo estático y dinámico

```
1 ArrayList<FiguraGeometrica> coleccionDeFiguras = new ArrayList<>();
2 coleccionDeFiguras.add (new Circulo (radio));
3 coleccionDeFiguras.add (new Rectangulo (lado1 , lado2));
4
5 float suma = 0.0f;
6 for (FiguraGeometrica unaFigura : coleccionDeFiguras) {
7     suma += unaFigura.area();
8 }
```

- ★ ¿Cuál es el tipo estático de unaFigura?
- ★ ¿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 FiguraGeometrica { // entre otras cosas ...
2     float area() { return 0.0f; }
3 }
4 class Circulo extends FiguraGeometrica { // entre otras cosas ...
5     float area() { return Math.PI * radio * radio; }
6 }
7 class Rectangulo extends FiguraGeometrica { // entre otras cosas ...
8     float area() { return lado1*lado2; }
9 }
10
11 // En algún otro sitio ...
12 for (FiguraGeometrica unaFigura : coleccionDeFiguras) {
13     suma += unaFigura.area();
14 } // ¿Qué implementación de area 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) ((Object) new Profesor()));  
17  
18 }
```

# Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

## Java: Ejemplo sobre comprobaciones explícitas de tipos

```
1 class Persona {  
2     private String nombre;  
3  
4     public Persona(String n) {  
5         nombre=n;  
6     }  
7  
8     public void setNombre(String n) {  
9         nombre=n;  
10    }  
11 }  
12  
13 class Profesor extends Persona {  
14  
15     public Profesor(String n) {  
16         super(n);  
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     Persona p;
4     Random r = new Random();
5
6     int dado = r.nextInt(6)+1;
7
8     if (dado<=3) {
9         p=new Persona("Pepe");
10    }
11    else {
12        p=new Profesor("Pepe");
13    }
14
15    // Nada recomendable
16    // Mal diseño
17    // No lo hagáis
18    // Lo digo en serio, no hagáis este tipo de diseños
19    if (p instanceof Profesor) {
20        p.setNombre("Prof. Pepe");
21    }
22    else {
23        p.setNombre("Don/Dña Pepe");
24    }
25 }
```



# Comprobaciones explícitas de tipos

- Deben evitarse las comprobaciones explícitas de tipos

## Java: Forma correcta de proceder

```
1 class Persona {
2     private String nombre;
3
4     public Persona(String n) { nombre=n; }
5 }
6
7     protected void setNombre(String n) { nombre=n; }
8
9     public void cambiarNombre(String n) {
10         setNombre("Don/Dña "+n);
11     }
12 }
13
14 class Profesor extends Persona {
15     public Profesor(String n) {
16         super(n);
17     }
18
19     @Override
20     public void cambiarNombre(String n) {
21         setNombre("Prof. "+n);
22     }
23 }
```

# 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     Persona p;
4     Random r=new Random();
5
6     int dado=r.nextInt(6)+1;
7
8     if (dado<=3) {
9         p=new Persona("Pepe");
10    }
11    else {
12        p=new Profesor("Pepe");
13    }
14
15    // Tenemos el comportamiento correcto de forma automática
16
17    p.cambiarNombre("Pepe");
18
19    // También será válido si se añaden nuevos descendientes
20    // de Persona/Profesor simplemente redefiniendo el método
21
22 }
```

## 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
12 end
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 19 a 24 de la transparencia 16 con la línea 17 de la transparencia 18  
(¡y solo hay involucradas 2 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**

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

**Programación y Diseño Orientado a Objetos**

(Curso 2019-2020)