Programación Orientada a Objetos

Tema 3: Propiedades Básicas de la Orientación a Objetos

Eduardo Mosqueira Rey

LIDIA
Laboratorio de Investigación y desarrollo en Inteligencia Artificial

Departamento de Computación
Universidade da Coruña, España

Objetivos

- **Conocer y analizar las principales propiedades de la orientación a objetos**
- **Estas propiedades incluirán aquellas que son compartidas por los tipos abstractos de datos (como abstracción, encapsulación, modularidad), como aquellas propias de la orientación a objetos (herencia, polimorfismo, ligadura dinámica, etc.).**
- **Estudiar cómo un lenguaje como Java implementa, en mayor o menor medida, dichas propiedades básicas.**

Índice

1. Abstracción
2. Encapsulamiento
3. Modularidad
4. Jerarquía
5. Polimorfismo
6. Tipificación
7. Ligadura dinámica

Índice

Abstracción
Definición y características

- **Definición**
  - **Representación de las características fundamentales de algo sin incluir antecedentes o detalles irrelevantes**
- **Características**
  - **Es uno de los métodos fundamentales para enfrentarse a la complejidad inherente al software (ya visto en los TADs).**
  - **La OO fomenta que el uso de abstracciones en los datos y procedimientos para simplificar la descripción del problema**
  - **El elemento clave de la abstracción es la clase**
    - **Clase ≡ Descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por su estado específico y por la posibilidad de realizar una serie de operaciones.**
  - **Ejemplo, Esfera**
    - **Estado: coordenadas del centro y radio**
    - **Operaciones: mover el centro, cambiar el radio.**

Índice

2. Encapsulamiento

- Definición y características
- Ventajas

Encapsulamiento
Definición y características

- **Definición**
  - **Proceso de almacenar en un mismo compartimiento los elementos de una abstracción que constituyen su estructura y su comportamiento**
- **Características**
  - **Abstracción y el encapsulamiento son conceptos complementarios:**
    - **La abstracción se centra en el comportamiento observable de un objeto**
    - **El encapsulamiento se centra en la implementación que da lugar a ese comportamiento.**
  - **El encapsulamiento también implica ocultación de información**
    - **Cada objeto revela lo menos posible de su estructura interna**
    - **parte pública $\Rightarrow$ interfaz, parte privada $\Rightarrow$ implementación.**
  - **Ejemplos**
    - **Una operación es vista por sus usuarios como si fuera una simple entidad, aunque está formada por una secuencia de operaciones a bajo nivel.**
    - **Un objeto es visto como un simple objeto en vez de como una composición de sus partes individuales.**

Encapsulamiento
Ventajas

- **Ventajas**
  - **La supresión de los detalles de bajo nivel nos permite razonar acerca de la operación u objeto de forma más eficiente.**
  - **Un cambio en la representación de una abstracción puede no obligar a un cambio en los clientes que la utilicen**
    - **Cambios en el diseño que no afecten al interfaz no se propagan**
    - **Podemos cambiar una función por otra más eficiente sin afectar a los usuarios de dicha función**
    - **Muy importante ya que facilita el mantenimiento del software**
  - **Java**
    - **La encapsulación se consigue a través del concepto de clase combinado con los especificadores de acceso que limitan la visibilidad de los atributos y métodos.**

Índice

3. Modularidad

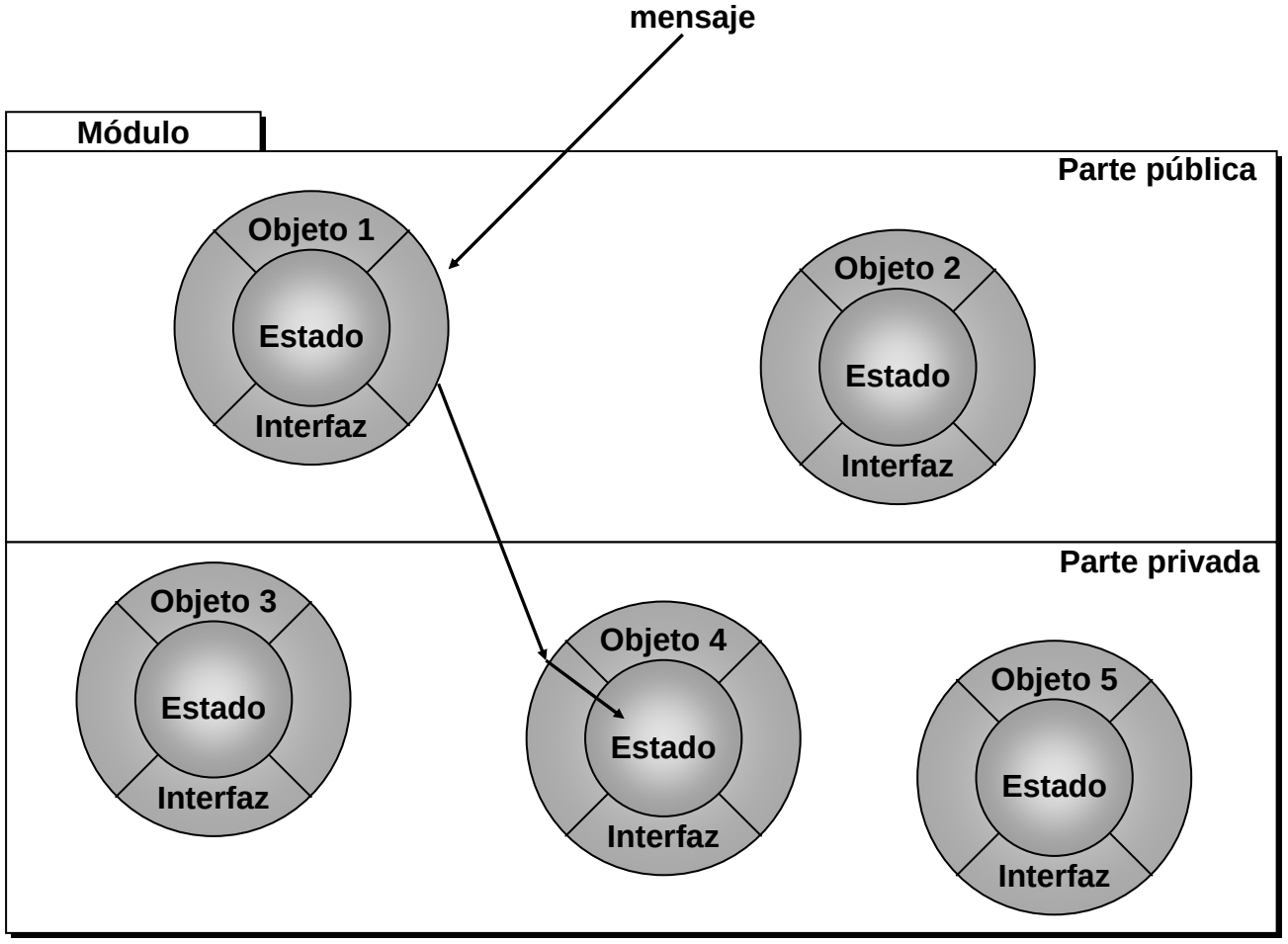
- Definición y características
- Modularidad en Java

Modularidad
Definición y características

- **Definición**
  - **Propiedad que tiene un sistema que ha sido descompuesto en un conjunto de partes o módulos que sean cohesivos y débilmente acoplados**
    - **Cohesivos ≡ agrupan abstracciones que guardan relación lógica**
    - **Débilmente acoplados ≡ minimizan las dependencias entre módulos**
- **Ventajas**
  - **El hecho de fragmentar un programa en componentes individuales suele contribuir a reducir su complejidad**
  - **Permite crear una serie de fronteras bien definidas y dentro del programa ⇒ aumenta la comprensión del mismo.**
- **Sinergia entre abstracción, encapsulamiento y modularidad**
  - **Un objeto proporciona una frontera bien definida alrededor de una sola abstracción, el encapsulamiento y la modularidad proporcionan barreras que rodean a esa abstracción.**

Modularidad
Definición y características

- **Abstracción, Encapsulamiento y Modularidad**



Modularidad
Modularidad en Java

- **Clases**
  - **Encapsulan los atributos y métodos de un tipo de objetos en un solo compartimiento**
  - **Ocultan, mediante los especificadores de acceso, los elementos internos que no se pretende publicar al exterior.**
  - **Esta protección es altamente configurable al existir varios niveles de acceso:**
    - **public**
    - **protected**
    - **por defecto (package)**
    - **private**

Modularidad
Modularidad en Java

- **Ficheros**
  - **Unidades físicas de compilación**
  - **Dentro de un fichero “.java” pueden residir varias clases con las siguientes restricciones**
    - **Sólo puede haber una clase pública por fichero**
    - **El nombre del fichero debe ser el mismo que el de la clase pública**
    - **Si no existe ninguna clase pública tampoco existe ninguna restricción con respecto al nombre del fichero**
  - **Compilación**
    - **La compilación de un fichero “.java” genera tantos ficheros “.class” (bytecodes) como clases existen en dicho fichero**
    - **Para facilitar la compilación Java recompila los ficheros “.java” si tienen el mismo nombre que un fichero “.class” pero con una fecha posterior.**

Modularidad
Modularidad en Java

- **Paquetes**
  - **Unidades lógicas de agrupación de clases**
  - **Las clases pueden ser**
    - **Públicas: forman parte del interfaz de su paquete**
    - **No públicas: sólo son visibles a clases de su mismo paquete**
- **Definición**
  - **Se utiliza la directiva package al principio de cada fichero.**
  - **En caso de que no se especifique paquete se considera que todas las clases que se encuentran en el paquete por defecto**
- **Jerarquías de paquetes**
  - **El nombre del paquete puede tener varios niveles, lo que facilita su organización (java.util, java.util.jar, java.awt, java.awt.color, etc.).**
  - **De todas formas no existe el concepto de subpaquete ni relaciones jerárquicas entre paquetes (la relación entre java.util y java.util.jar es la misma que entre java.util y javax.swing $\Rightarrow$ son dos paquetes distintos).**
  - **Simplemente se permite una organización jerárquica de los nombres de los paquetes por motivos de organización y claridad**

Modularidad
Modularidad en Java

- **Objetivos de los paquetes:**
  - **Diseñar un dispositivo de modularidad de nivel superior a las clases.**
    - **Cada paquete puede tener sus propias clases privadas desconocidas para aquella persona que quiera utilizar el paquete.**
  - **Agrupar clases con funcionalidades similares.**
    - **De forma que sean más fáciles de localizar y pueda verse claramente que las clases están relacionadas.**
  - **Organizar físicamente los ficheros fuente.**
    - **Para poder utilizar las clases compiladas, los ficheros .class deben estar disponibles en el directorio indicado por la variable de entorno CLASSPATH.**
    - **Para organizar el directorio del CLASSPATH Java equipara los nombres de paquetes con los directorios en disco (Java buscará las clases del paquete com.miempresa.utils en el directorio CLASSPATH/com/miempresa/utils).**

Modularidad
Modularidad en Java

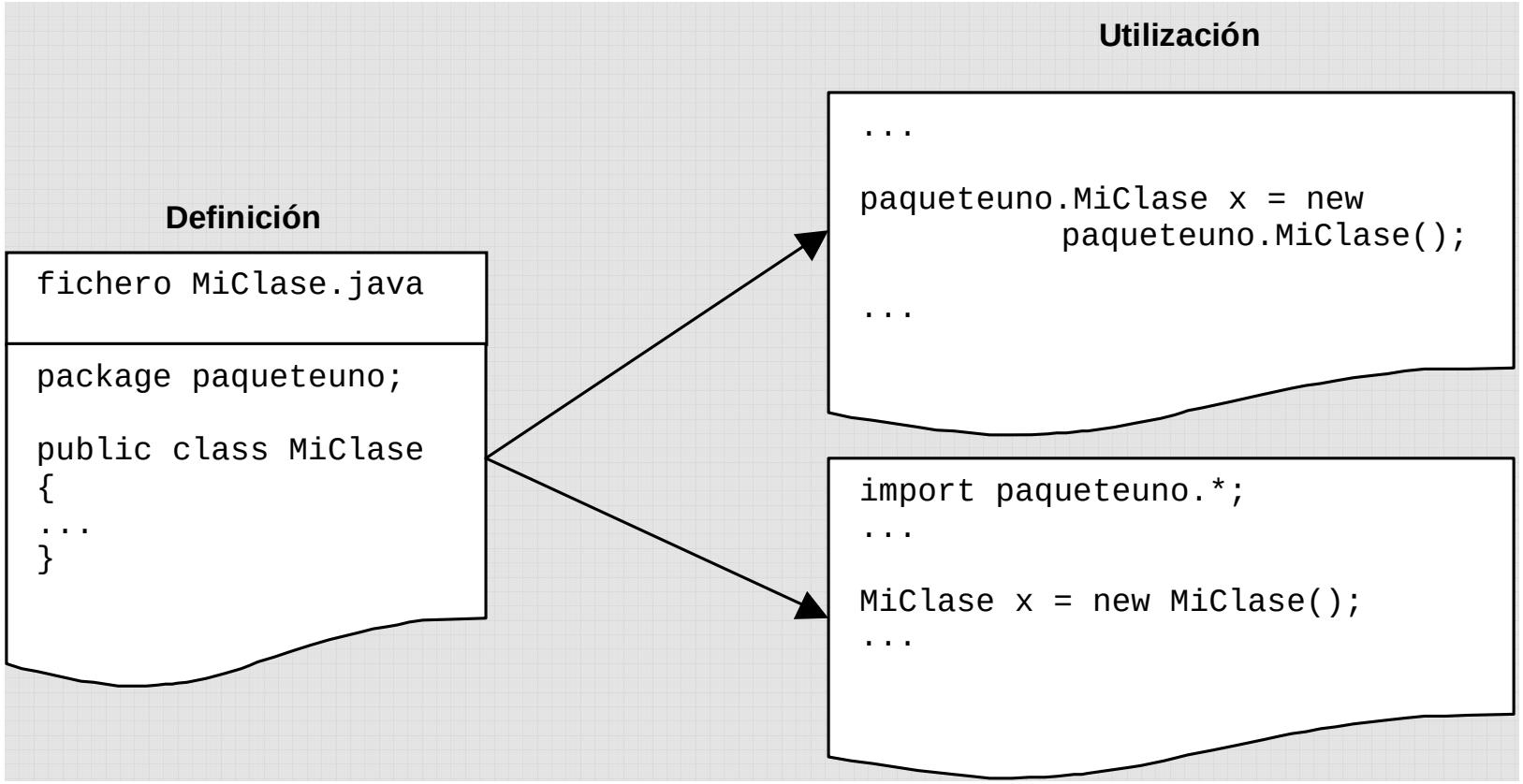
- **Objetivos de los paquetes (cont.):**
  - **Prevenir conflictos de nombre y favorecer la reutilización.**
    - **Los nombre de las clases pueden entrar en conflicto, para evitarlo la clase se compone de el nombre del paquete al que pertenece, un punto y el nombre de la clase**
    - **Pueden existir conflictos con los nombres de los paquetes**
    - **Para evitar esto existe el convenio de utilizar como prefijo a los nombres de los paquetes los nombres invertidos del dominio de Internet de la empresa que los ha desarrollado**
    - **Ejemplo, los paquetes de Borland tendrían el prefijo “com.borland”, los paquetes del OMG tendría el prefijo “org.omg”, etc.**
    - **Los nombres de dominio no pueden repetirse, evitando así los posibles conflictos**

Modularidad
Modularidad en Java

- **La sentencia import**
  - **Para simplificar la sintaxis y no tener que introducir un nombre largo de paquete cada vez que se utiliza una clase puede utilizarse la sentencia import al principio de cada fichero**
  - **De esta forma puede utilizarse el nombre de la clase sin indicar el paquete al que pertenece**
  - **La sentencia import puede utilizarse de dos formas distintas:**
    - **import nombrepaquete.* ⇒ importa todas las clases del paquete**
    - **import nombrepaquete.Clase ⇒ importa sólo la clase especificada**
  - **La sentencia import simplemente especifica que los contenidos públicos del paquete destino entran en el espacio de nombres del origen. No es necesario incluir la sentencia import para dar privilegios de acceso de un paquete a otro**
  - **Si dos clase de distintos paquetes comparten nombre se deberá usar su path completo.**

Modularidad
Modularidad en Java

- **Ejemplo**
  - **Por convección los nombres de los paquetes de java se escriben siempre en minúsculas (incluidas las primeras letras de cada palabra)**



Índice

4. Jerarquía

- Herencia
- Herencia simple vs. múltiple
- Clases abstractas
- Interfaces
- Composición

Jerarquía

- **Definición de jerarquía**
  - **Una jerarquía es una clasificación de las abstracciones**
- **Jerarquía de generalización/especialización:**
  - **Define relaciones ES_UN**
  - **También se conoce como herencia**
- **Jerarquía de agregación:**
  - **Define relaciones ES_PARTE_DE**
  - **También se conoce como composición**

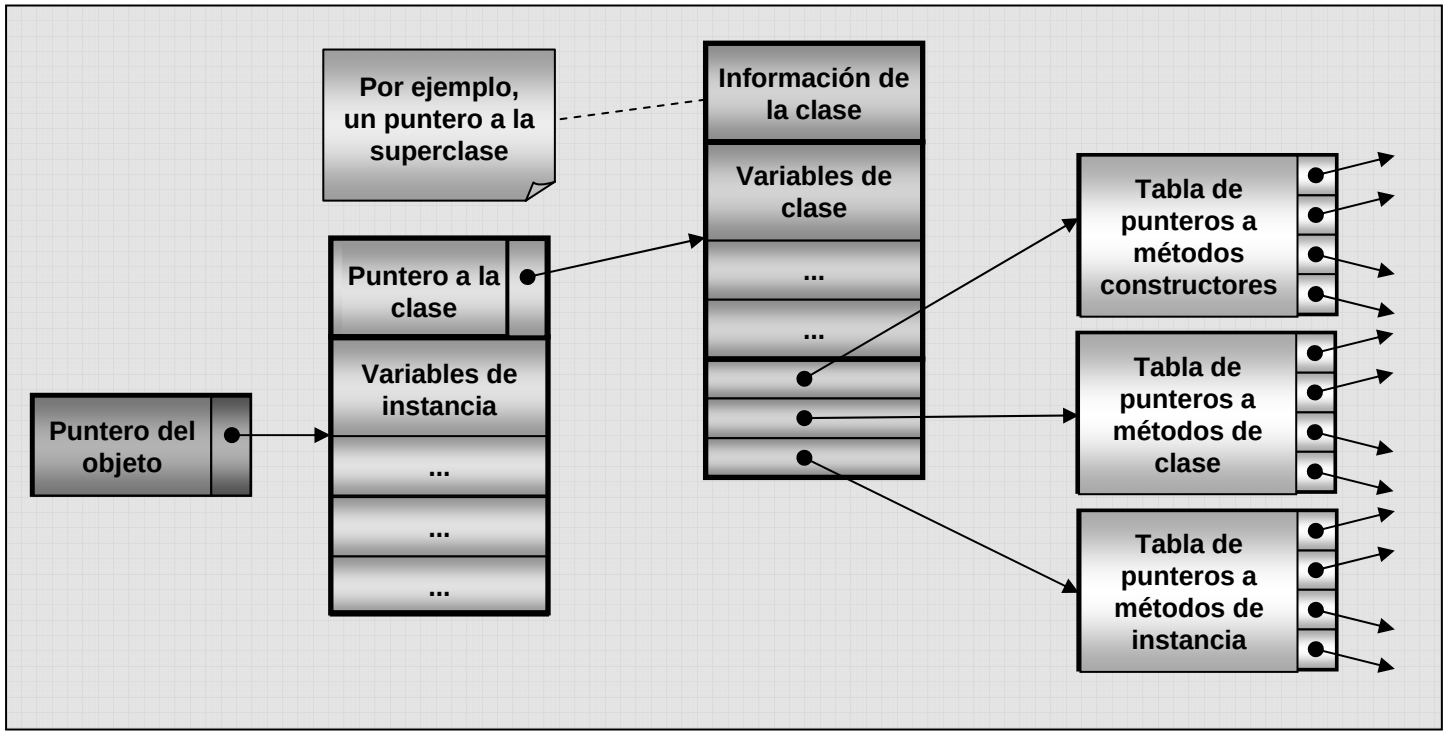
Jerarquía
Herencia

- **Herencia**
  - **Define una relación entre clases, en las que una clase comparte la estructura de comportamiento definida en una o más clases**
  - **La herencia permite declarar las abstracciones con economía de expresión**

- **Subclases y superclases**
  - **Una subclases hereda de una o más superclases y aumenta o redefine la estructura y el comportamiento de dichas superclases**
  - **Las subclases representan conceptos especializados**
  - **Las superclases representan generalizaciones de los aspectos comunes de las subclases**

Jerarquía
Herencia

- **Representación interna de los objetos**



Jerarquía
Herencia

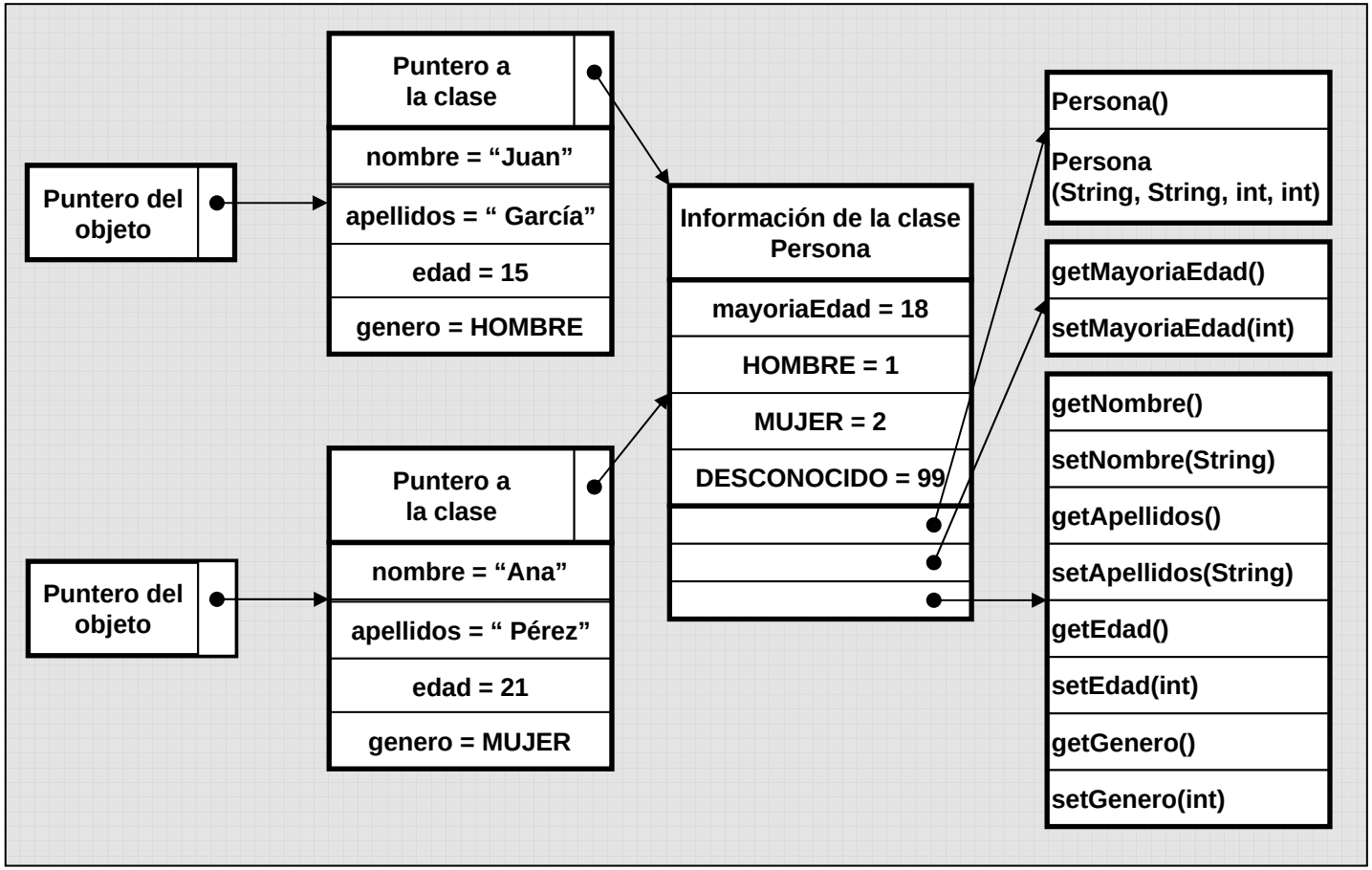
- **Representación interna de los objetos**

```
class Persona
{
  protected String nombre;
  protected String apellidos;
  protected int edad;
  protected int genero;
  protected static
                int mayoriaEdad = 18;
  public static final int HOMBRE = 1;
  public static final int MUJER  = 2;
  public static final
                int DESCONOCIDO = 99;

  public Persona()
  {
    nombre = "";
    apellidos = "";
    edad = 0;
    genero = DESCONOCIDO;
  }

  public Persona(String n, String a,
                int e, int g)
  {
    nombre = n;
    apellidos = a;
    edad = e;
    genero = g ;
  }
```

```
  public static int getMayoriaEdad()
  { return mayoriaEdad;}

  public static void setMayoriaEdad
                                (int me)
  { if (me > 0) mayoriaEdad = me; }

  public String getNombre()
  { return nombre;}

  public void setNombre(String n)
  { nombre = n; }

  public String getApellidos()
  { return apellidos;}

  public void setApellidos(String a)
  { apellidos = a; }

  public int getEdad()
  { return edad;}

  public void setEdad(int e)
  { if (e > 0) edad = e; }

  public int getGenero()
  { return genero;}

  public void setGenero(int g)
  { if (g==1 || g==2 || g==99)
      genero = g;}
}
```

Jerarquía
Herencia

- **Representación interna de los objetos**



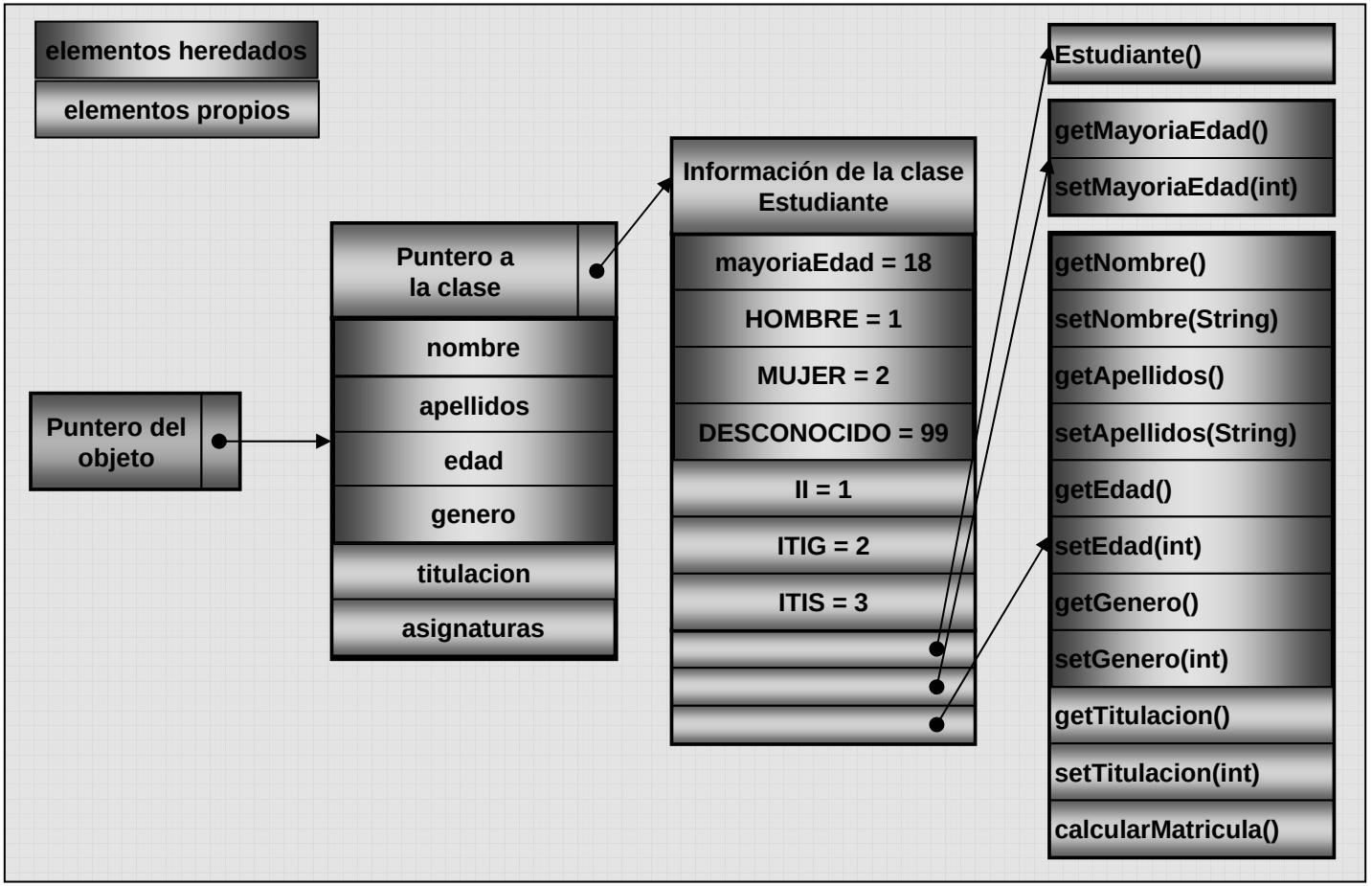
Jerarquía
Herencia

- **Representación interna de los objetos con herencia**

```
class Estudiante extends Persona
{
  public final static int II   = 1;
  public final static int ITIG = 2;
  public final static int ITIS = 3;
  private int titulacion;
  public String [] asignaturas;

  public int getTitulacion()
  { return titulacion;}

  public void setTitulacion(int t)
  { if (t == 1 || t==2 || t==3)
      titulacion = t; }

  public float calcularMatricula()
  { ... }
}

class Profesor extends Persona
{
  public String departamento;
  public String categoria;
  public String dedicacion;
  public java.util.Date antiguedad;

  public float calcularSueldo()
  { ... }
}
```

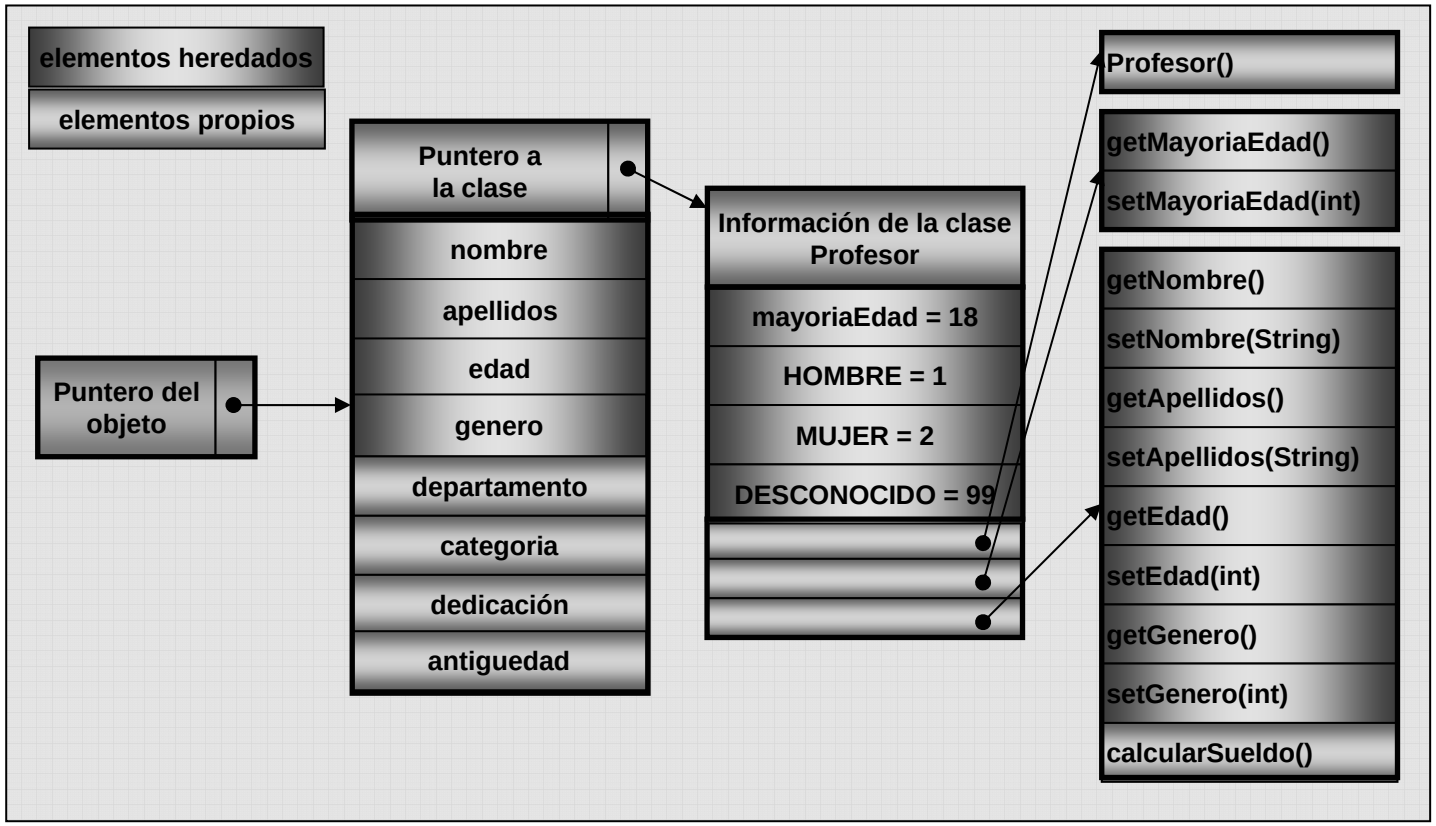
Jerarquía
Herencia

- **Representación interna de los objetos con herencia**



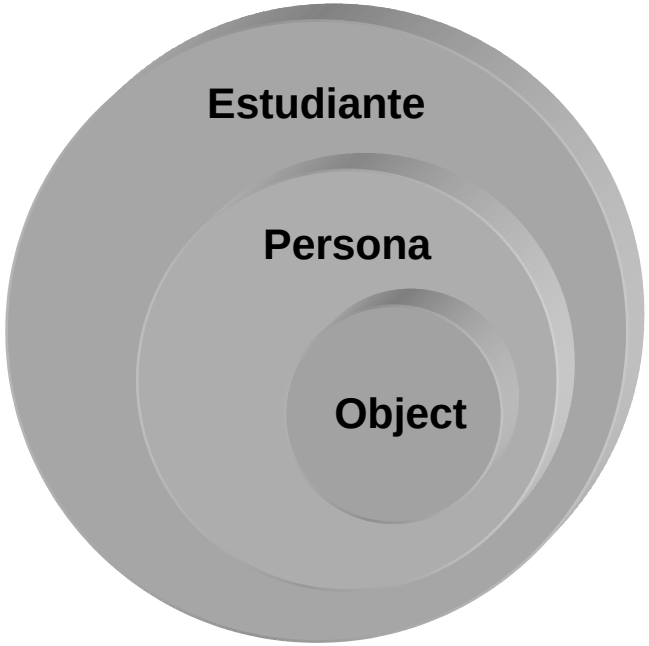
Jerarquía
Herencia

- **Representación interna de los objetos con herencia**



Jerarquía
Herencia

- **Aspectos importantes a recordar**
  - **Un objeto de una subclase siempre incluye en su interior un objeto de la superclase**



  - **Los elementos definidos en una superclase aparecen siempre en la misma posición en las subsecuentes subclases**

Jerarquía
Herencia

- **¿Qué constructor habrá que poner en SubClase?**
  - **Todo objeto de una subclase es un objeto de una superclase**
  - **La palabra clave para llamar a un constructor de la superclase es *super* (la palabra clave *this* se utilizaba para llamar a un constructor de la propia clase)**

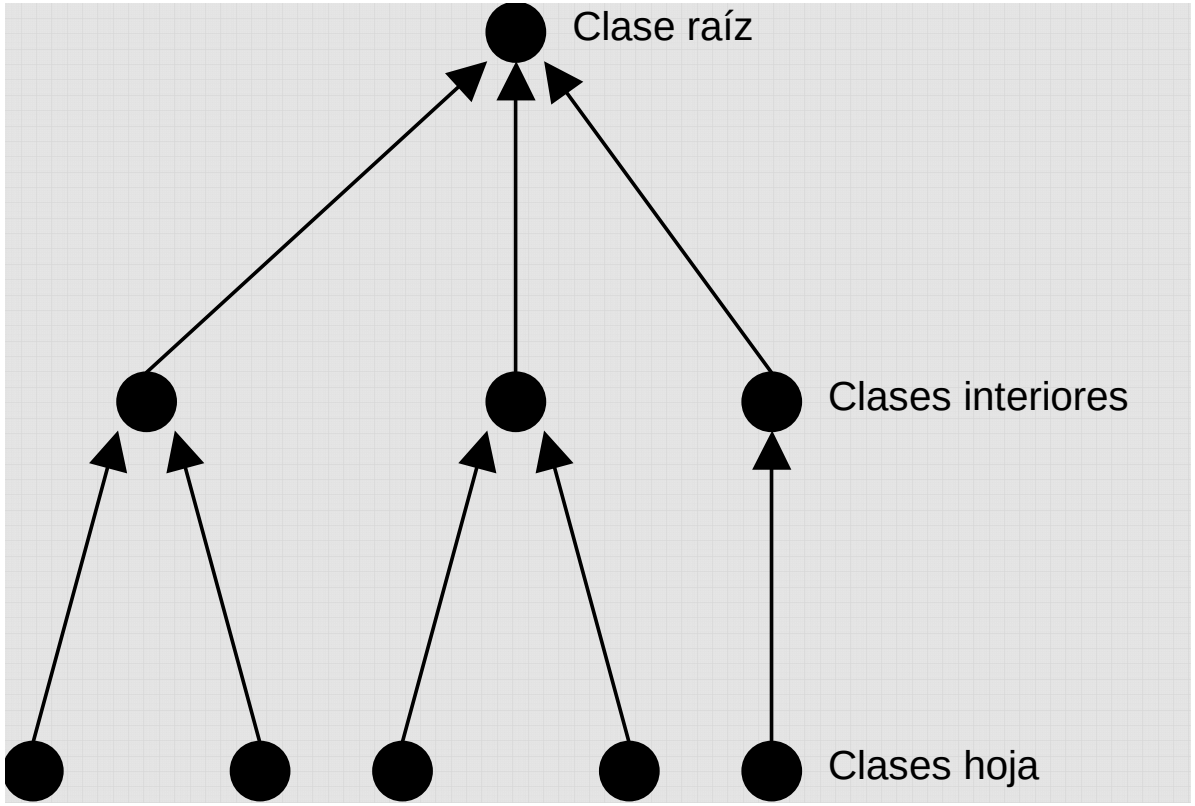
```
class SuperClase
{
  int valor;
  public SuperClase(int valor)
  { this.valor = valor;  }
}

class SubClase extends SuperClase
{




}
```

Jerarquía
Herencia

- **Herencia y constructores**
  - **Los constructores no se heredan, son exclusivos de la clase en la que se definen**
  - **Por ello la primera instrucción de un constructor de una subclase es llamar al constructor de la superclase, y este a su vez al de su superclase hasta llegar a la clase Object**
  - **Esta llamada generalmente está implícita y consiste en una llamada al constructor sin parámetros. Si queremos hacerla explícita deberemos poner "super()" como primera instrucción**
  - **Si el constructor sin parámetros no existe la llamada implícita fallará y será necesario hacer una llamada explícita: "super(param1, param2)"**

Jerarquía
Herencia

- **Estructura de las jerarquías de herencia**



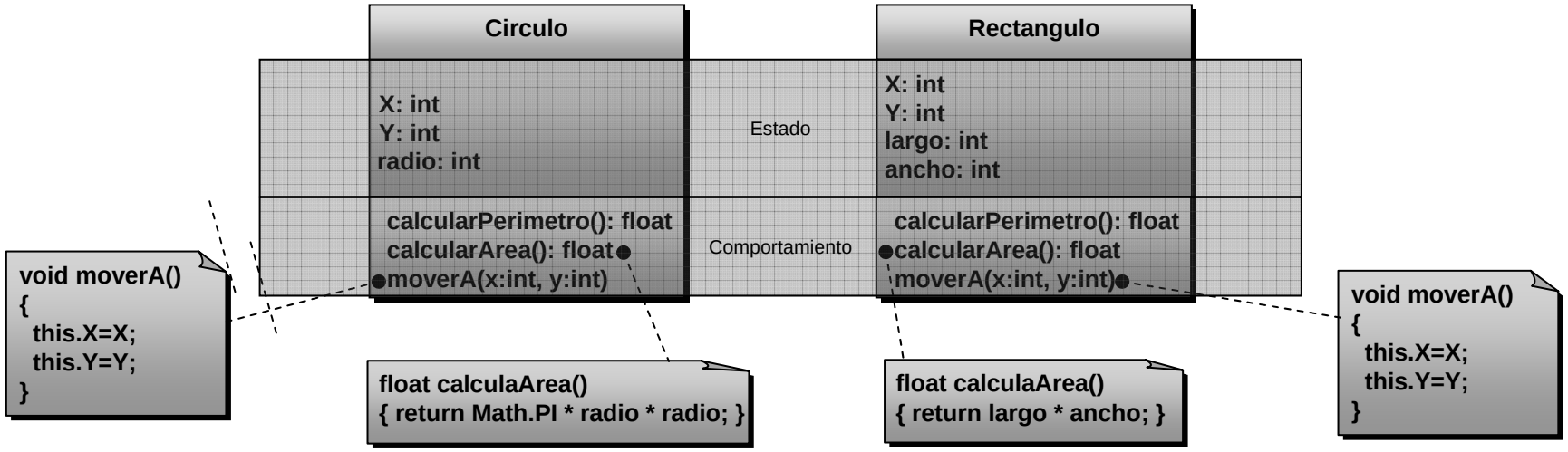
Jerarquía
Herencia simple vs. múltiple

- **Herencia simple**
  - **Cada clase tiene, como máximo, un ancestro**
  - **Ejemplos: Java, Object Pascal y C#**
- **Herencia múltiple**
  - **Una clase puede heredar de varias clases simultáneamente**
  - **Ejemplos: C++ y Eiffel**
- **Tendencia actual:**
  - **Eliminación de la herencia múltiple a favor de la herencia simple.**
  - **Motivo: los conflictos que genera y su resolución**
    - **¿Que pasa cuando heredamos el mismo elemento de dos clases distintas?**
    - **Los mecanismos para resolver estos conflictos son específicos de cada lenguaje y aumentan la complejidad de la programación y reducen la comprensibilidad**
    - **Existen mecanismos como los interfaces que permiten simular una cierta herencia múltiple que evita los conflictos y permiten diseños elegantes de clases.**
    - **Es posible desarrollar jerarquías de herencia complejas y flexibles evitando la herencia múltiple**

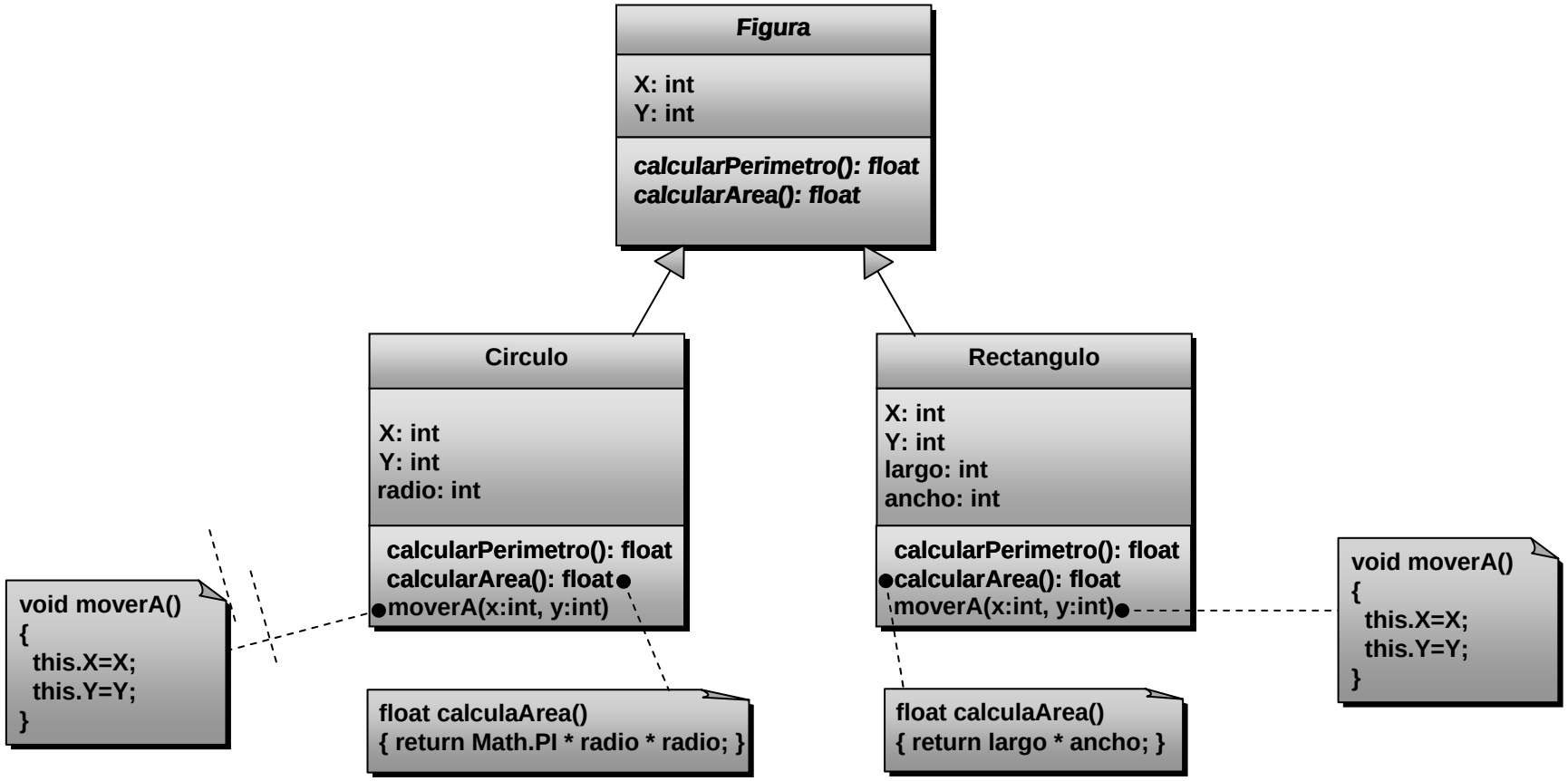
Jerarquía
Clases abstractas

- **Características**
  - **Representan conceptos generales que agrupan métodos comunes y dejan para la implementación de las subclases métodos específicos.**
  - **No pueden ser instanciadas**
  - **Un método abstracto sólo puede pertenecer a una clase abstracta, pero una clase abstracta puede tener métodos no abstractos**
  - **Las subclases de una clase abstracta deben implementar los métodos abstractos o declararse como abstractas.**

Jerarquía
Clases abstractas



Jerarquía
Clases abstractas



Jerarquía
Clases abstractas

```
// Nota: Se omiten constructores y métodos
// de lectura y escritura de atributos

abstract class Figura
{
  private int X;
  private int Y;

  public abstract float calcularPerimetro();
  public abstract float calcularArea();

  public void moverA (int X, int Y)
  {
    this.X=X;
    this.Y=Y;
  }
}
```

```
class Circulo extends Figura
{
  private int radio;

  public float calcularPerimetro()
  { return 2*(float)Math.PI*radio; }

  public float calcularArea()
  { return (float)Math.PI*radio*radio; }
}

class Rectangulo extends Figura
{
  private int largo;
  private int ancho;

  public float calcularPerimetro()
  { return (largo*2)+(ancho*2); }

  public float calcularArea()
  { return largo*ancho; }
}
```

Jerarquía
Interfaces

- **Definición**
  - **Son clases abstractas puras ya que sólo definen un protocolo de conducta y no como debe implementarse dicha conducta.**

```
[public] interface NombreInterfaz [extends SuperInterfaz, ...]
{ /* cuerpo del interfaz */ }
```

- **Características**
  - **Se definen con la palabra clase interface y no class**
  - **Contienen las cabeceras de métodos que son, implícitamente, públicos y abstractos, pero no incluyen los cuerpos de los métodos**
  - **Pueden contener atributos pero serán implícitamente static y final, es decir, constantes de clase**
- **Ejemplo:**

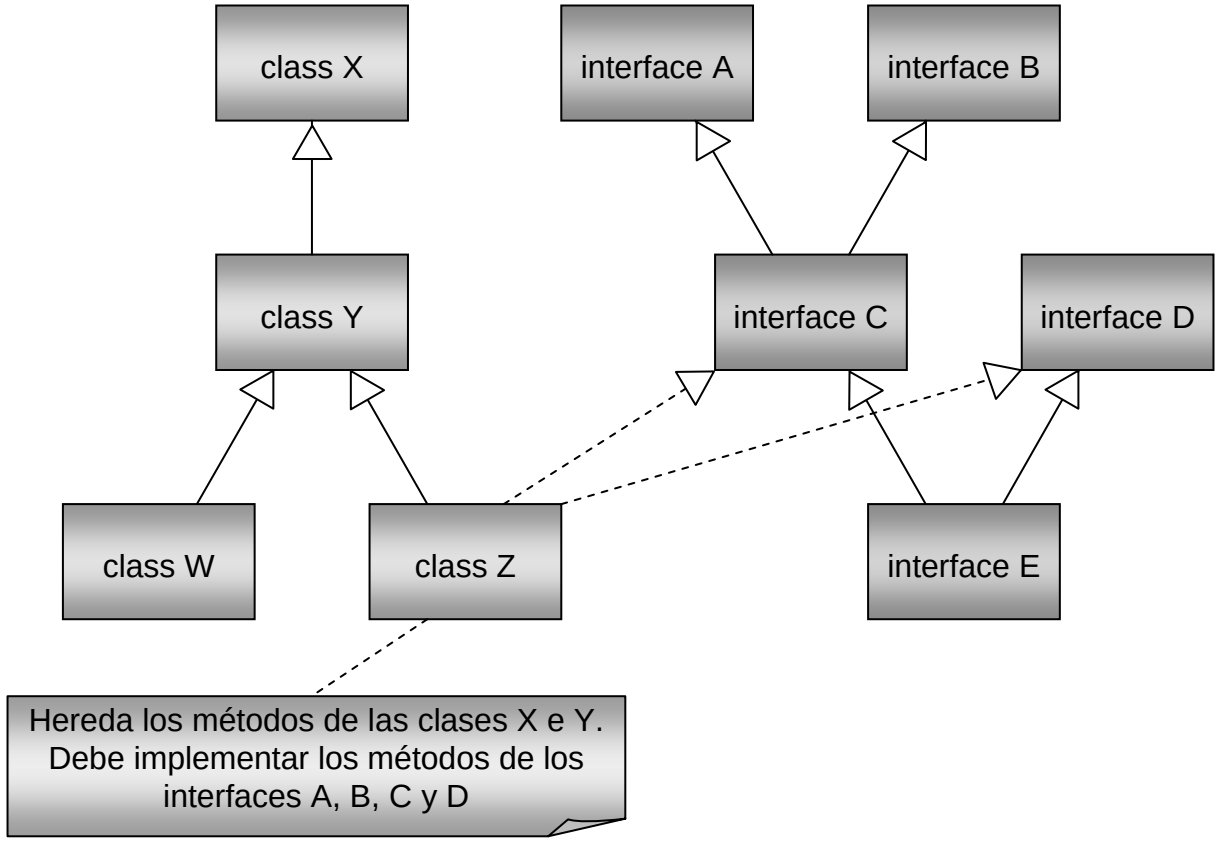
```
interface FiguraInterfaz
{
   float calcularPerimetro();
   float calcularArea();
}
```

Jerarquía
Interfaces

- **Herencia EN los interfaces**
  - **Siguen una jerarquía de herencias paralela a la jerarquía de herencias de las clases.**
  - **Permite la herencia múltiple entre interfaces**
- **Herencia DE interfaces (implementación)**
  - **Cualquier clase puede heredar (definido a través de la palabra clave implements) de varios interfaces**
  - **No se dan los problemas que veíamos con la herencia múltiple y las clases ya que no se hereda la implementación de los métodos sino las cabeceras, por lo que los posibles conflictos son más sencillos de solucionar.**
  - **Si una clase implementa un interfaz debe dar implementación a todos los métodos incluidos en su interfaz (incluidos los superinterfaces que extiende) o bien declararse abstracta y diferir la implementación del interfaz a sus subclases**

Jerarquía
Interfaces

- **Representación de las jerarquías de clases e interfaces**



Jerarquía
Interfaces

- **Definición del interfaz Iterator**

```
public interface Iterator
{
  boolean hasNext();
  Object next();
}
```

- **Utilización del interfaz Iterator**

```
public class PostOrden implements Iterator
{
  boolean hasNext()
  {
    // Comprueba si hemos terminado
    // el recorrido post-orden
  }

  Object next()
  {
    // Devuelve el siguiente elemento en
    // el recorrido post-orden
  }
  …
}
```

Jerarquía
Interfaces

- **Usos más típicos de los interfaces**
  1. **Capturar las similitudes entre clases no relacionadas**
  2. **Revelar el interfaz de programación de un objeto sin revelar su clase**
  3. **Definir nuevos tipos de datos**
  4. **Marcadores de clase**
  5. **Contenedores de elementos globales**
  6. **Simular un tipo de herencia múltiple**

Jerarquía
Interfaces

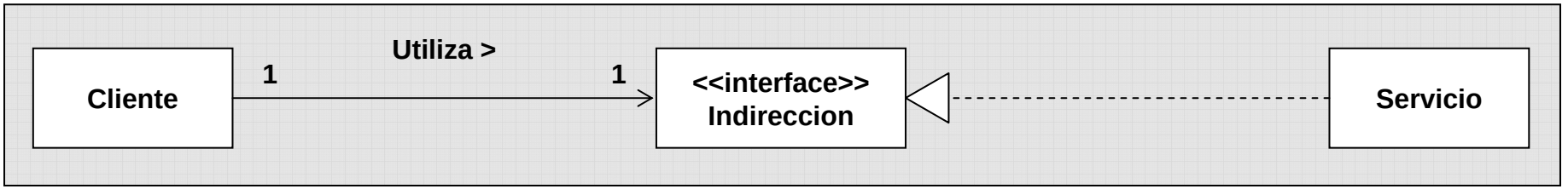
1. **Capturar las similitudes entre clases no relacionadas sin forzar una relación de clases artificiosa.**
   - **Los interfaces no imponen ninguna restricción con respecto a la jerarquía de clases**
   - **Sin embargo, si una clase hereda de una clase abstracta no puede heredar de ninguna otra clase**

```
public class PreOrden implements Iterator { ... }

public class InOrden implements Iterator { ... }

public class PostOrden implements Iterator { ... }
```

Jerarquía
Interfaces

2. Revelar el interfaz de programación de un objeto sin revelar su clase.

- **Los interfaces se utilizan a menudo como intermediarios para desacoplar las clases usuarias de funciones de las clases que implementan dichas funciones.**
- **Por ejemplo: una clase Cliente utiliza la clase Servicio a través del interfaz Indireccion. La clase Cliente no conoce realmente que clase implementa Indireccion por lo que se pueden realizar cambios en la clase Servicio sin que el cliente se vea alterado.**



Jerarquía
Interfaces

3. Definir nuevos tipos de datos

- Múltiples objetos de clases diferentes pueden ser tratados como si fueran de un mismo tipo común, donde este tipo viene indicado por el interfaz
- No se pueden instanciar objetos en sí del tipo interfaz ya que la definición de un interfaz no tiene constructor, por lo que no es posible invocar el operador new sobre un tipo interfaz
- Sin embargo si pueden declararse atributos del tipo del interfaz y asignarle a estos atributos objetos de una clase que implemente dicho interfaz

```
interface Celda
{
    void draw();
    void toString();
    void toFloat();
}
```

```
...
Celda [][] matriz;
matriz = new Celda[10][10];
...
matriz [0][0] = new Ecuacion();
matriz [0][1] = new Fecha();
...
matriz [0][0].draw();
matriz [0][1].draw();
...
```

Jerarquía
Interfaces

4. Marcadores de clase

- Pueden utilizarse interfaces vacios como un flag, un marcador para señalar a una clase con una propiedad determinada.
- Por ejemplo Java incluye en su API el interfaz Cloneable que sirve para indicar que dicho objeto se puede clonar
- Actualmente suele ser más recomendable usar las *annotations* del Java SE 5

```
class MiClase implements Cloneable
{ ... }

MiClase x = new MiClase();
if (x instanceof Cloneable)
  {...}
```

Jerarquía
Interfaces

- **Anotaciones (Anexo)**
  - **Las anotaciones son metadatos, es decir, información sobre los elementos del programa**
  - **Vienen a ser algo similar a las etiquetas del javadoc (@author) pero aplicadas al código, no a los comentarios**
  - **Por defecto Java incluye tres anotaciones**
    - **Override: anota a un método e indica que sobreescribe a otro de la superclase**
    - **Deprecated: indica que un método o clase está anticuado y no debe usarse (“depreciado” en la jerga de Java)**
    - **SupressWarnings: el compilador no avisa de warnings ocurridos sobre el elemento anotado**
  - **Las anotaciones no son mas que interfaces por lo que pueden declarar métodos o constantes**
  - **La potencia de las anotaciones viene de la posibilidad de crear anotaciones personalizadas**

Jerarquía
Interfaces

- **Ejemplos de anotaciones**
  - **¿Cuál es el resultado de la compilación y ejecución del siguiente código?**

```
class MiClase
{
  @Override
   public boolean equals(MiClase o)
   { return false; }
}
```

Jerarquía
Interfaces

- **Ejemplos de anotaciones**

```
@SuppressWarnings(value={"unchecked"})
class MiClase
{
  List l = new ArrayList();

  public void hola()
  { l.add("Hola"); }

  @Deprecated public void hazAlgo()
  {}
}
```

Indica que el compilador obvie los *warnings* generados en esta clase. Por ejemplo, usar un ArrayList sin usar genericidad

Declaramos el método como *depreciado* para que lo sepa el compilador.

No confundir con la etiqueta del javadoc, aunque el compilador también la reconoce (pero por motivos de compatibilidad *hacia atrás*)

Debe usarse las dos etiquetas al mismo tiempo

Jerarquía
Interfaces

- **Anotaciones personalizadas**

Mas ejemplos en "Java 1.5 Tiger: A developer's notebook"

```
Import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@interface EnDesarrollo
{   int porcentaje();  }

@EnDesarrollo (porcentaje=50)
class MiClase {  }

public class AnotacionSimple
{
  public static void main(String[] args)
  {
    MiClase x = new MiClase();
    Class c = x.getClass();
    if (c.isAnnotationPresent(EnDesarrollo.class))
    {
       System.out.println("Esta clase está en desarrollo");
       EnDesarrollo ed = (EnDesarrollo)c.getAnnotation(EnDesarrollo.class);
       System.out.println("Porcentaje = " + ed.porcentaje());
    }
  }
}
```

Anotamos la anotación indicando que queremos que su información se mantenga en tiempo de ejecución (para utilizarla con la reflexión)

La palabra clave para definir anotaciones es @interface, el resto de la declaración es como si fuera un interface típico. En este caso indicamos que existe un método que devuelve un porcentaje

Anotamos una clase con la anotación EnDesarrollo, le damos un valor a porcentaje (el compilador nos evita tener que escribir la sobreescritura del método porcentaje por MiClase)

Usamos la reflexión para obtener información en tiempo de ejecución del objeto de tipo MiClase

Jerarquía
Interfaces

5. **Como contenedores de elementos globales al programa**
   - **En Java no existen variables globales como en otros lenguajes. Los interfaces pueden utilizarse para incluir aspectos comunes que deban compartir clases heterogéneas, como por ejemplo: constantes**

```
public interface Constantes
{
  public double PI = 3.1416;
  public int MAXFILAS = 15;
}
```

```
public class Circulo implements Constantes
{ int radio ;
  public double perimetro()
  { return (2 * PI * radio); }
}
```

Jerarquía
Interfaces

- **Como contenedores de elementos globales al programa**
  - **Mala solución: polución del API exportado**
    - **Confusión entre los clientes**
    - **Compromiso a largo plazo**
  - **Soluciones**
    - **Definir las constantes en una clase o interfaz relacionado: Como MIN_VALUE y MAX_VALUE en Integer**
    - **Crear clases que sean tipos enumerados**
    - **Definir las constantes en clases de utilidad (utility classes) como por ejemplo *Math***

```
public class Constantes
{
  private Constantes () {} // Para prevenir instanciación
  public static final double PI = 3.1416;
  public static final int MAXFILAS = 15;
}
```

Jerarquía
Interfaces

- **Como contenedores de elementos globales al programa**
  - **Nuevo problema:**
    - **Hay que anteceder el nombre de la clase al nombre de la constante**

```
perimetro = 2 * Constantes.PI * radio;
```

  - **Soluciones al nuevo problema**
    - **Utilizar constantes locales: mala y confusa solución**

```
private static final double PI = Constantes.PI;
```

    - **Utilizar la facilidad de importación estática (versión 1.5)**

```
import static es.udc.Constantes.*;

public class Circulo
{
  ...
  int radio ;
  public double perimetro()
  { return (2 * PI * radio); }
  ...
}
```

Los *import static* importan un elemento de una clase o todos los elementos de una clase. Los *import* tradicionales importaban una clase o todas las clases de un paquete

No es necesario indicar a qué clase pertenece PI

¿Qué problema plantean?

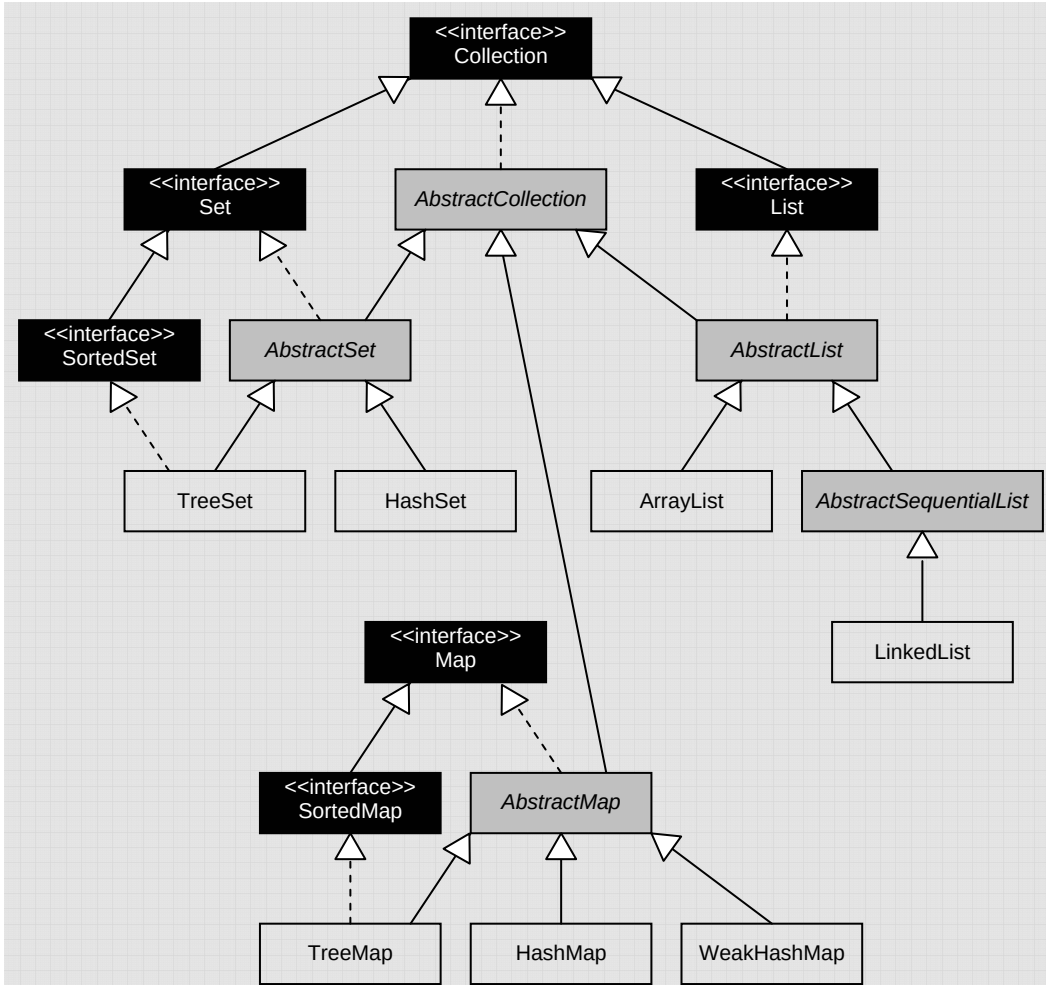
Jerarquía
Interfaces

6. Para simular un tipo de herencia múltiple

- Hay autores que destacan que los interfaces son la forma que tiene Java de implementar la herencia múltiple.
- Sin embargo esto no es del todo cierto ya que la herencia múltiple como se conoce por ejemplo en C++ y los interfaces presentan una serie de importantes diferencias:
  - No se pueden heredar variables desde un interface.
  - No se pueden heredar implementaciones de métodos desde un interface.
  - La herencia de un interface es independiente de la herencia de la clase, las clases que implementan el mismo interface pueden o no estar relacionadas a través del árbol de clases.
- Lo que sí es cierto es que los interfaces parecen una alternativa para resolver algunos problemas que en otros lenguajes se resuelven utilizando herencia múltiple.

Jerarquía
Interfaces

- **¿Interfaces o clases abstractas?**
  - **API de colecciones**



Jerarquía
Composición

- **Características**
  - **La composición define relaciones ES_PARTE_DE y ocurre cuando los atributos de un objeto son a su vez objetos.**

```
class Baraja
{
  private Carta [] cartas;

  public Baraja()
  {
    int i=0, j;
    cartas = new Carta[40];
    for (j=1; i<=10; i++, j++)
    { cartas[i]=new Carta ("Espadas", j);
    }
    for (j=1; i<=10; i++, j++)
    { cartas[i]=new Carta ("Copas", j);
    }
    for (j=1; i<=10; i++, j++)
    { cartas[i]=new Carta ("Bastos", j);
    }
    for (j=1; i<=10; i++, j++)
    { cartas[i]=new Carta ("Oros", j);
    }
  }

  public void barajar()
  { ... }
  public Carta devuelveCarta (int n)
  { return cartas[n]; }
}
```

```
class Carta
{
  private int numero;
  private String palo;

  public Carta (String p, int n)
  { palo = p;
    numero = n;
  }

  public int numero()
  { return numero; }

  public String palo()
  { return palo; }

  public String toString()
  { return numero + " " + palo; }
}
```

Jerarquía
Composición

- **Composición como mecanismo de reutilización**
  - **Estudiante podría contener una persona en vez de heredar de persona**
  - **Se usa la delegación, los métodos de Estudiante que debe resolver persona (getNombre) se delegan a la instancia interna de Persona**
  - **¿Qué os parece más cómodo, la herencia o la delegación?**

```
class Estudiante
{
  public String titulacion;
  public String [] asignaturas;
  private Persona p;

  public float calcularMatricula()
  { ... }

  public String getNombre()
  { return p.getNombre();}

  public void setNombre(String n)
  { p.setNombre(n); }

  public String getApellidos()
  { return p.getApellidos();}

  public void setApellidos(String d)
  { p.setApellidos(d); }

  public int getEdad()
  { return p.getEdad();}

  public void setEdad(int e)
  { p.setEdad(e); }

  public int getGenero()
  { return p.getGenero();}

  public void setGenero(int e)
  { p.setGenero(e); }
}
```

Índice

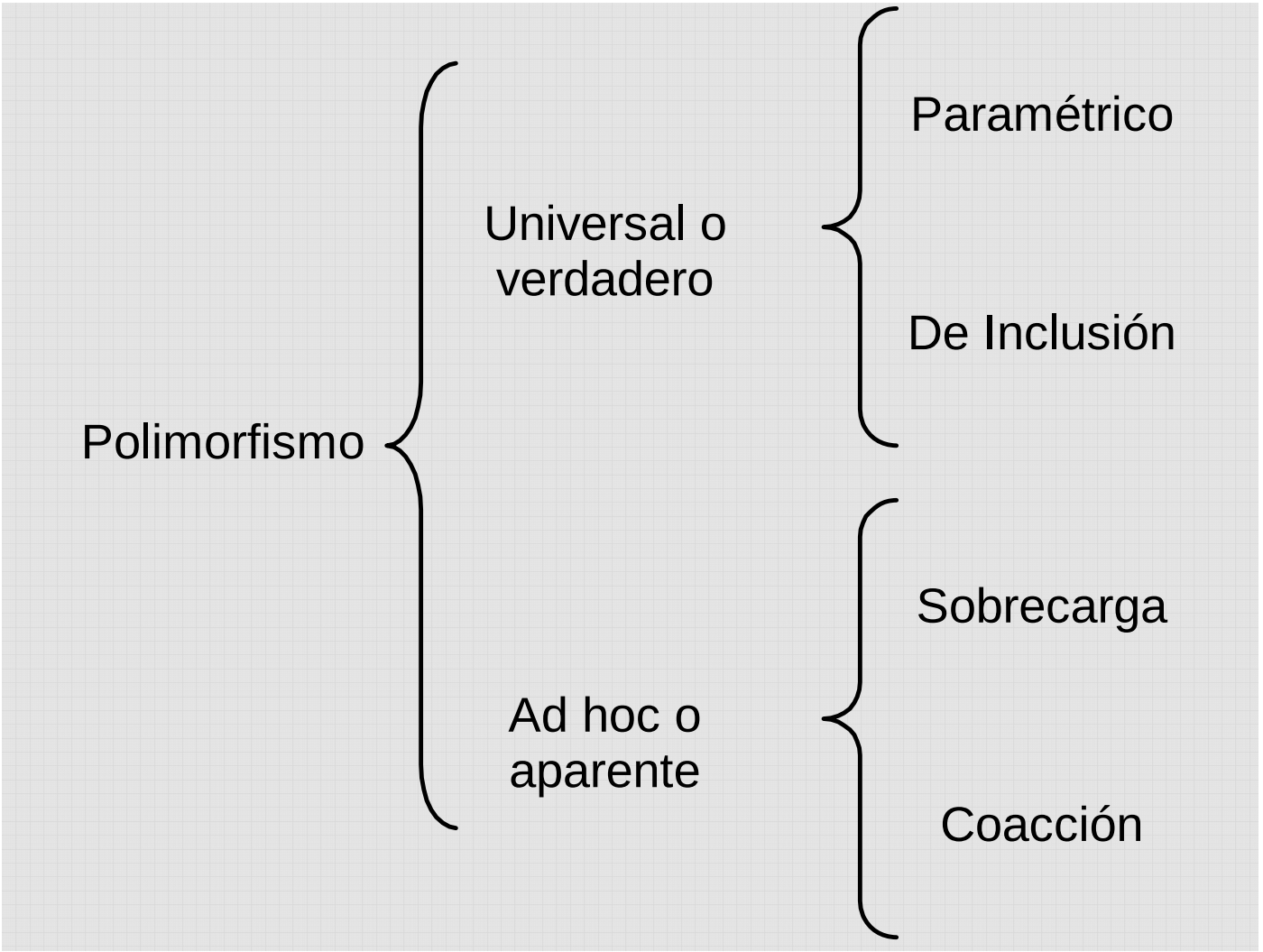
5. Polimorfismo

- Tipos de polimorfismo
- Polimorfismo de inclusión
- Polimorfismo paramétrico
- Sobrecarga
- Coerción

Polimorfismo

- **Definición**
  - **El término viene del griego y significa “muchas formas”**
  - **En lenguajes OO se podría definir como: “la capacidad de una variable de tener más de un tipo y de una función de ser aplicada sobre parámetros de distintos tipos”.**
- **Lenguajes monomórficos**
  - **Las variables sólo pueden tener un tipo y las funciones sólo admitían parámetros con un único tipo (ej. Pascal estándar)**
- **Tipos de polimorfismo**
  - **El polimorfismo involucra distintos aspectos, de funcionamiento similar, pero que se basan en conceptos completamente distintos**
  - **Para intentar poner un poco de orden entre todas las definiciones que implicaban polimorfismo Cardelli y Wegner (1985) realizaron una clasificación de las mismas**

Polimorfismo
Tipos de polimorfismo



Polimorfismo
Tipos de polimorfismo

- **Polimorfismo universal o verdadero**
  - **Consiste en que pueden existir valores que pueden pertenecer a varios tipos, y se utiliza el mismo código para tratar los diferentes tipos**
  - **Tenemos dos variantes del polimorfismo universal, el polimorfismo paramétrico y el de inclusión**
- **Polimorfismo ad hoc o aparente**
  - **Se utiliza distinto código para tratar diferentes tipos**
  - **De esta forma una función polimórfica sería implementada a través de un conjunto de funciones monomórficas**
  - **Dentro del polimorfismo ad hoc podemos distinguir la sobrecarga y la coerción**

Polimorfismo
Polimorfismo de inclusión

- **Características**
  - **Es propio de los lenguajes orientados a objetos.**
  - **Es aquel que se consigue a través de la herencia. Por eso se llama también polimorfismo de subclases, o de herencia, o simplemente polimorfismo**
  - **Indica que un objeto de una subclase puede utilizarse en aquellos lugares en los que se requiere un objeto de sus superclases**
  - **Recordemos que un objeto de la subclase incluye internamente un objeto de la superclase**

```
Animal listaAnimales[] = new Animal[10];
listaAnimales [0] = new Perro();
listaAnimales [1] = new Gato();
listaAnimales [2] = new Animal();
…
```

Polimorfismo
Polimorfismo de inclusión

- **Métodos genéricos**
  - **En lenguajes, como Java u Object Pascal (Delphi), en los que existe una superclase común para todas las clases (Object), es sencillo crear métodos genéricos que acepten parámetros de cualquier tipo o que devuelvan valores de cualquier tipo**
  - **Simplemente los parámetros o el tipo de retorno se especifican de tipo Object ya que todas las clases son subclases de Object y pueden suplantarlo si es necesario**

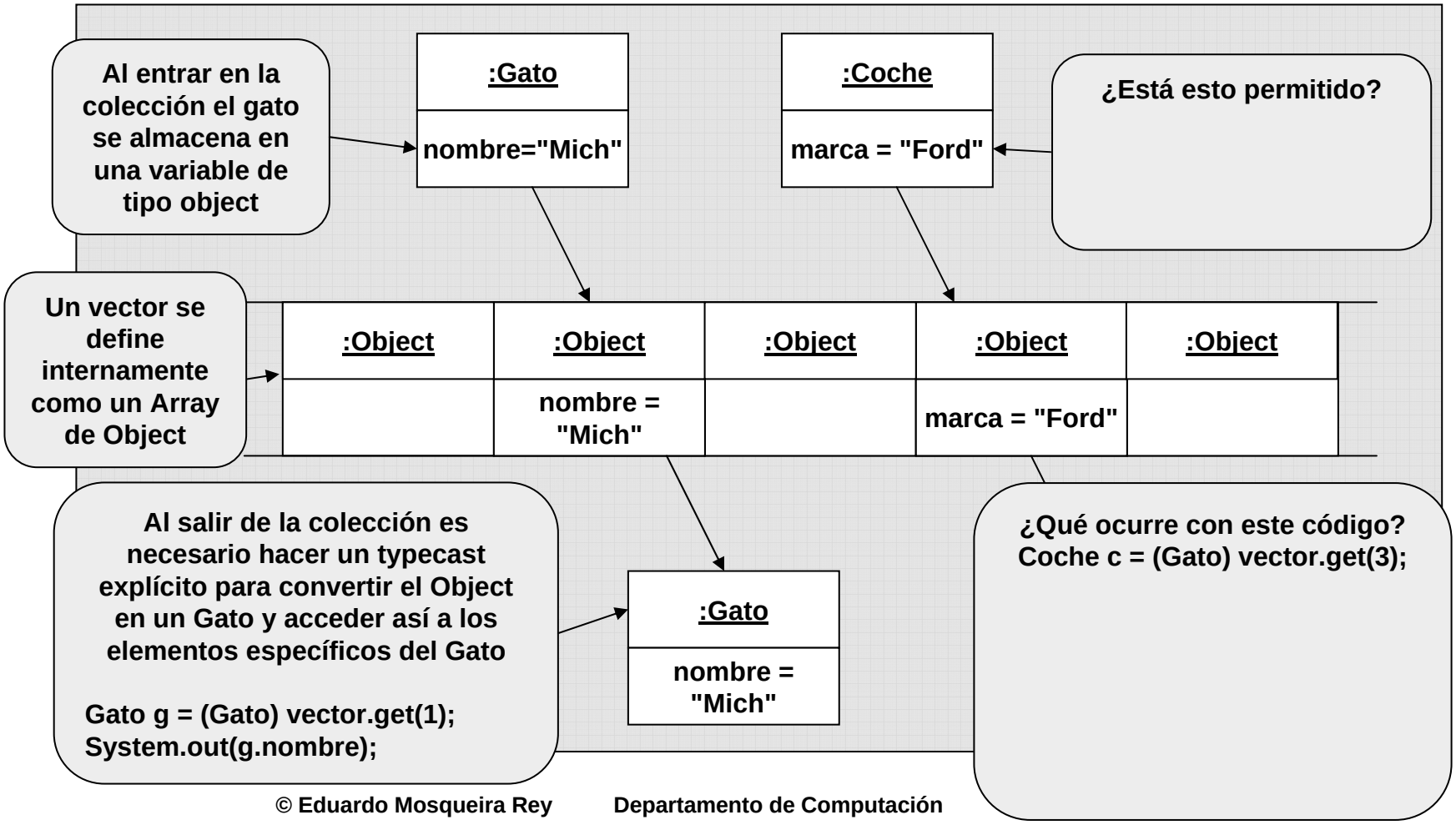
```
public Object metodoGenerico(Object o)
{ … }
```

Polimorfismo
Polimorfismo de inclusión

- **Colecciones y polimorfismo de inclusión**
  - **Java utilizaba hasta la versión 1.4 profusamente el polimorfismo de inclusión en las clases destinadas a almacenar colecciones de datos (como Vector, List, HashTable, etc.).**
  - **Así un vector se define cono un contenedor de objetos de tipo Object, por lo que en un vector podría ir cualquier tipo de clase**

Polimorfismo
Polimorfismo de inclusión

- **Colecciones y polimorfismo de inclusión**



Polimorfismo
Polimorfismo de inclusión

- **Colecciones y polimorfismo de inclusión**

```
import java.util.*;

public class ListaSinGenericidad
{
  public static void main(String [] args)
  {
    List listaGatos = new ArrayList();
    listaGatos.add(new Gato());
    listaGatos.add(new Coche());
    Gato gato1 = (Gato)listaGatos.get(0);
    Gato gato2 = (Gato)listaGatos.get(1);
  }
}
```

No existe *seguridad de tipos* en las colecciones que utilizan el polimorfismo de inclusión

El type-cast es necesario para evitar un error en tiempo de compilación

Error ClassCastException, estamos intentando convertir un Coche en un Gato

Polimorfismo
Polimorfismo paramétrico

- **Características**
  - **Se conoce habitualmente como “genericidad”**
  - **Consiste en que una misma función es aplicada sobre una variedad de tipos distintos**
  - **Se denomina paramétrico porque las funciones necesitan un parámetro para saber qué tipo debe de ser utilizado**
  - **En Java pueden definirse métodos y clases parametrizadas (sólo a partir de la versión 1.5)**
  - **La principal ventaja de la genericidad consiste en la posibilidad de definir colecciones de objetos con comprobación de tipos en tiempo de compilación**

Polimorfismo
Polimorfismo paramétrico

- **Colecciones y polimorfismo paramétrico**

```
import java.util.*;

public class ListaConGenericidad
{
  public static void main(String [] args)
  {
    List<Gato> listaGatos = new ArrayList<Gato>();
    listaGatos.add(new Gato());
    //listaGatos.add(new Coche());
    Gato gato1 = listaGatos.get(0);
  }
}
```

Se añade un parámetro entre los símbolos "<" y ">" que indica el tipo de la colección.

El mismo parámetro debe añadirse en la llamada al constructor.

Ahora el compilador hace comprobaciones en tiempo de ejecución. Esta línea está comentada porque sino daría error

Los type-cast ya no son necesarios para extraer elementos de una colección. Existe *seguridad de tipos*

Polimorfismo
Polimorfismo paramétrico

- **Colecciones y polimorfismo paramétrico**
  - **Seguridad:**
    - **Realizan comprobaciones de tipo en tiempo de compilación (en una lista de gatos sólo puede haber gatos)**
    - **Se evitan los fallos en tiempo de ejecución**
  - **Claridad:**
    - **No necesitamos “typecasts” para sacar objetos de la colección**
  - **Comprensibilidad**
    - **Las colecciones se declaran con un tipo, por lo que hace más claro su uso**

Polimorfismo
Polimorfismo paramétrico

- **Creación de clases con polimorfismo paramétrico**
  - **Los tipos genéricos deben ser objetos (no tipos básicos)**

```
class CajaGenerica<T>
{
  private T valor;

  public T getValor()
  { return valor; }

  public void setValor(T valor)
  { this.valor = valor; }

  public static void main(String [] args)
  {
    CajaGenerica<Gato> cajaGato= new CajaGenerica<Gato>();
    CajaGenerica<Integer> cajaInteger= new CajaGenerica<Integer>();
    cajaGato.setValor(new Gato());
    System.out.println("everything is ok");
    //cajaInteger.setValor(new Gato()); // Error de compilación
  }
}
```

Polimorfismo
Polimorfismo paramétrico

- **Métodos con genericidad**
  - **Eliminar de una colección de Strings aquellos que sean de longitud = 4**
  - **Método con polimorfismo de inclusión**

```
static void eliminar(Collection c)
{
  for (Iterator I = c.iterator(); i.hasNext(); )
    if (((String) i.next()).length() == 4)
      i.remove
}
```

  - **Método con genericidad**

```
static void eliminar(Collection<String> c)
{
  for (Iterator<String> I = c.iterator(); i.hasNext(); )
    if (i.next().length() == 4)
      i.remove();
}
```

Polimorfismo
Polimorfismo paramétrico

- **Otras ventajas de la versión 1.5**
  - **Autoboxing / autounboxing**
    - **Los tipos básicos se convierten automáticamente a sus clases asociadas y viceversa**
  - **Foreach**
    - **Las colecciones pueden recorrerse sin usar iteradores**
    - **No hay una palabra clave “foreach” por compatibilidad**
  - **Ejemplo:**
    - **Sumamos de una colección de enteros cuántos son iguales a 5**

```
public int antes(Collection c)
{
  int num=0;
  for (Iterator i = c.iterator(); i.hasNext();)
  {
    if (i.next().equals(new Integer(5)))
      num++;
  }
  return num;
}
```

```
public int ahora(Collection<Integer> c)
{
  int num=0;
  for (Integer i : c)
    if (i.equals(5))
      num++;
  return num;
}
```

Polimorfismo Paramétrico
Genericidad y subclases

- **¿Es este código legal?**

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

- **De otra forma: Si un String puede asignarse a un Object, ¿Puede un List<String> ser asignado a un List<Object>?**

Polimorfismo Paramétrico
Genericidad y subclases

- **Antes de los genéricos para crear un método que imprimiera todos los elementos de una colección hacíamos:**

```
public void printCollection(Collection c)
{
  for(Object e : c)
    System.out.println(e);
}
```

- **Ahora: ¿Es esta solución correcta?**

```
public void printCollection(Collection<Object> c)
{
  for(Object e : c)
    System.out.println(e);
}
```

Polimorfismo Paramétrico
Genericidad y subclases

- **Comodines**
  - **<?>**
    - **Representa a cualquier clase. No es lo mismo que <Object>**

```
public void printCollection(Collection<?> c)
{
  for(Object e : c)
    System.out.println(e);
}
```

  - **<? extends MiClase>**
    - **Representa a cualquier subclase de MiClase, incluida la propia MiClase**
  - **<? super MiClase>**
    - **Representa a cualquier superclase de MiClase, incluida la propia MiClase**

Polimorfismo
Polimorfismo paramétrico

- El proceso de borrado (erasure)
  - Java realiza un proceso denominado “erasure” en el cual toda la información acerca de la genericidad no se propaga a los bytecodes
  - De esta forma la genericidad funciona sólo en tiempo de compilación pero no en tiempo de ejecución
- Consecuencias
  - La genericidad no afecta al rendimiento
  - No es posible la sobrecarga de funciones parametrizadas
    - unMetodo(List<Integer> x) y unMetodo(List<String> x) son iguales en tiempo de ejecución ya que la información parámetrica se elimina
  - Las colecciones parametrizadas pueden corrormperse utilizando reflexión
    - Por ejemplo un atributo List<Integer> obtenido a través de getDeclaredFields es sólo un List
- Motivación del “erasure”
  - Compatibilidad: el bytecode no generificado no se diferencia del que si lo está
  - Evolución, no revolución: podemos generificar un API (por ejemplo Collections) y no actualizar los clientes que lo usan

Polimorfismo
Polimorfismo paramétrico

- **La clase *java.utils.Arrays* contiene un método *asList* que permite representar un array como una lista.**

```
public static <T> List<T> asList(T... a)
```

- **Preguntas:**
  1. **¿Qué clase de tipo de retorno tiene este método? ¿Qué es la <T> que aparece después de static y antes de List<T>**
  2. **¿Qué clase de parámetros acepta este método? ¿Qué es la expresión "T… a"?**

- **El uso del método puede aclarar algunas cosas…**

Polimorfismo
Polimorfismo paramétrico

```
 1 import java.util.*;
 2
 3 public class Clase
 4 {
 5   public static void main(String[] args)
 6   {
 7     List<Object> lista = Arrays.asList("uno", "dos", "tres");
 8     List<Object> lista2 = Arrays.asList(1, "dos", "tres");
 9     List<Object> lista3 = Arrays.<Object>asList("uno", "dos", "tres");
10     List<Object> lista4 = Arrays.<Object>asList(1, "dos", "tres");
11   }
12 }
```

- **Surgen nuevas preguntas:**
  3. **La línea 7 da el siguiente error de compilación, ¿eres capaz de explicar por qué?**
  4. **La línea 8 da otro error ¿puedes explicar también el por qué?**
  5. **Las líneas 9 y 10 muestran como evitar los errores de compilación de las otras líneas pero, ¿qué es el <Object> que aparece antes de la llamada al método?**

Polimorfismo
Polimorfismo paramétrico

- **Solución:**

Polimorfismo
Sobrecarga

- **Características**
  - **La sobrecarga (overloading) consiste en utilizar el mismo nombre para denotar funciones distintas.**
  - **Para saber que función utilizar se utiliza el contexto de cada caso en particular (los parámetros o la clase en la que esté definida).**
  - **El polimorfismo es sólo aparente (ad hoc), no existe una función que acepte distintos parámetros, sino distintas funciones adaptadas a cada tipo de parámetro pero que nos dan la impresión de ser la misma porque comparten el mismo nombre.**

Polimorfismo
Sobrecarga

- **Sobrecarga entre clases:**
  - **La sobrecarga entre clases ocurre cuando clases distintas responden al mismo tipo de mensaje, es decir, distintas clases tienen métodos con el mismo nombre.**
  - **Esta situación es muy común en los lenguajes orientados a objetos porque permite definir nombres comunes a operaciones comunes y facilita el aprendizaje de las bibliotecas de funciones.**
  - **Por ejemplo, en Java el nombre isEmpty es compartido por clases como Vector, Hashtable o Rectangle. En las dos primeras devuelve true si la colección está vacía, en la última devuelve true si el área del rectángulo es cero.**
  - **Como vemos, la sobrecarga entre clases suele implicar que existe una cierta relación semántica entre los métodos sobrecargados, aunque esto no tiene porque ser siempre cierto.**

Polimorfismo
Sobrecarga

- **Sobrecarga paramétrica:**
  - **La sobrecarga paramétrica ocurre dentro de una misma clase cuando varios métodos comparten el mismo nombre pero se diferencian por su número y tipo de parámetros.**
  - **Es importante destacar que los métodos no tienen porque ser definidos todos en la misma clase sino que también se pueden sobrecargar métodos heredados de superclases.**
  - **Este tipo de sobrecarga no es exclusivo de los lenguajes orientados a objetos y también aparece en los lenguajes imperativos tradicionales**
  - **La sobrecarga paramétrica es muy utilizada en los constructores para ofrecer distintas vías en las que crear un objeto. Por ejemplo podemos crear un objeto esfera con parámetros por defecto, especificando todos o parte de sus parámetros o crearlo a partir de otro objeto esfera.**

Polimorfismo
Sobrecarga

- **Sobrecarga de operadores:**
  - **Consiste en darle varios significados a los operadores**
  - **Por ejemplo en Java, “+” significa sumar números pero también concatenar Strings**
  - **Otros lenguajes permiten a los usuarios sobrecargar libremente los operadores (C++)**
  - **Java no permite la sobrecarga de operadores por parte de los usuarios (por motivos de claridad)**

Polimorfismo
Sobrecarga

- **¿Es el siguiente código válido?**

```
int metodoX ();  { … }
float metodoX(); { … }
```

- **De otra forma: ¿El tipo de retorno se tiene en cuenta a la hora de resolver la sobrecarga?**

Polimorfismo
Sobrecarga

- **Sobreescritura (overriding)**
  - **Podría considerarse como un tipo de sobrecarga entre clases**
  - **Aparece cuando una subclase define un método con el mismo nombre, tipo de retorno y parámetros que una superclase**
  - **El comportamiento adecuado es que el método de la subclase sobreescriba al método de la superclase**
  - **La sobreescritura implica que existe sobrecarga entre clases (superclase y subclase), si bien lo contrario no tiene porque ser cierto.**

Polimorfismo
Sobrecarga

- **Tipos de sobreescritura:**
  - **Sobreescritura de reemplazo**
    - **El método de la subclase reemplaza por completo el método de la superclase**
  - **Sobreescritura de refinamiento**
    - **El método de la subclase es una forma refinada del método de la superclase (se utiliza el código de la superclase pero se añaden algunas características propias)**
    - **Para acceder a la versión del método de la superclase se utiliza el puntero super (que apunta a la superclase) seguido del nombre del método. Por ejemplo, super.metodoX()**
    - **Al contrario de la llamada super de los constructores, la llamada super en los métodos sobreescritos no tiene porque aparecer en primer lugar**

Polimorfismo
Sobrecarga

- **Característica importante de la sobreescritura**
  - **En la sobrecarga la decisión de qué método utilizar se hace en tiempo de compilación**
  - **En la sobreescritura, la decisión de qué método utilizar (si el de la superclase o el de la subclase) se toma en tiempo de ejecución**
  - **Esto se debe a que en tiempo de compilación no sabemos si una variable definida con el tipo de la superclase alberga una instancia de la superclase o una instancia de alguna de sus subclases.**
  - **Volveremos a este tema a la hora de hablar de la ligadura dinámica.**

Polimorfismo
Sobrecarga

- **Ejemplo de sobreescritura**

```
abstract class Personal
{
  protected long base;
  protected long anhos;

  long sueldo()
  { return base + (10000 * anhos);  }
}

class A extends Personal
{
  int numProyectos;
  long sueldo()
  { return (100000 * numProyectos);  }
}

class B extends Personal
{
  long extra;
  long sueldo()
  { return super.sueldo() + extra; }
}

class C extends Personal
{ ... }

class D extends Personal
{ ... }
```

Sobreescritura de reemplazo

Sobreescritura de refinamiento

Polimorfismo
Sobrecarga

- **Especificadores de visibilidad en sobrecarga**
  - **El especificador de visibilidad de un método que sobrescribe a otros debe proporcionar por lo menos la misma visibilidad que el método sobrescrito.**
  - **Es decir:**
    - **Si el método sobrescrito tenía visibilidad public, el método que sobreescribe debe ser public.**
    - **Si el método sobrescrito tenía visibilidad protected, el método que sobreescribe debe ser public o protected.**
    - **Si el método sobrescrito tenía visibilidad package, el método que sobreescribe no puede ser private.**
  - **No se especifica nada si el método sobreescrito tenía visibilidad private ya que estos métodos no pueden ser sobrescritos**

Polimorfismo
Coerción

- **Coerción (o coacción)**
  - **Es una operación semántica por la cual se convierte un argumento a el tipo esperado por una función para evitar que se produzca un error de tipos.**
  - **Las coerciones pueden hacerse estáticamente, insertándolas entre los argumentos y las funciones en tiempo de compilación (serían los conocidos type cast o conversiones de tipos), o pueden ser determinadas dinámicamente por tests en tiempo de ejecución sobre los argumentos.**
  - **Suele ser muy común en todo tipo de lenguajes, incluso en los fuertemente tipados, para evitar errores triviales con los tipos.**
  - **Ejemplo: Una función que acepta parámetros “float” aceptará sin problemas que le pasemos un “int”, simplemente convertirá ese int a un float en tiempo de ejecución**

Índice

6. Tipificación

- Introducción
- Comprobación de tipos
  - Estático
  - Estricto o fuerte
  - Dinámico

Tipificación
Introducción

- **Tipos**
  - **Definición:**
    - **“un tipo es una caracterización precisa de las propiedades estructurales y de comportamiento que comparten una serie de entidades”**
  - **Objetivo principal: Noción de congruencia**
    - **La concatenación de cadenas devuelve una cadena, contar los caracteres de una cadena devuelve un entero, etc.**
    - **Permiten detectar errores en un programa y facilitan el funcionamiento interno de los compiladores**
  - **En los lenguajes OO una clase es un tipo pero hay que tener cuidado:**
    - **Hay tipos que no son clases (los tipos básicos como int, float, etc.)**
    - **En ocasiones distintas clases implementan el mismo tipo de datos (el tipo List es implementado por dos clases distintas ArrayList y LinkedList)**
    - **Puede ocurrir que una subclase no sea un subtipo. Ver más información en el principio de sustitución de Liskov (principios de diseño)**

Tipificación
Comprobación de tipos

- **Comprobación de tipos**
  - **Podemos distinguir tres formas de comprobar los tipos: tipado estático, tipado estricto y tipado dinámico.**

- **Tipado estático**
  - **Consiste en que el tipo exacto de cada expresión puede ser determinado en tiempo de compilación mediante un análisis estático del programa.**
  - **El tipado estático permite detectar incongruencias en tiempo de compilación pero, sin embargo, puede llegar a ser muy restrictivo.**
  - **Un ejemplo de tipado estático sería el lenguaje Pascal estándar.**

Tipificación
Comprobación de tipos

- **Tipado estricto o fuerte**
  - **Es un requisito más débil que el tipado estático y consiste en que todas las expresiones de tipos tienen que ser consistentes en tiempo de compilación.**
  - **El compilador garantizará que el programa se ejecutará sin errores de tipos.**
  - **Muchas veces se incluye el tipado estático y el tipado estricto dentro de una misma categoría.**
  - **El tipado fuerte es propio de los lenguajes orientados a objetos como Java, Object Pascal o C++, aunque el nivel de exigencia en cada uno de ellos puede variar.**

Tipificación
Comprobación de tipos

- **Consistencia (definida a través de tres restricciones):**
  - **Restricción de declaración.**
    - **Todas las entidades del lenguaje (variables, objetos, atributos) deben tener un tipo declarado (que como hemos visto puede ser un tipo básico o una clase). Además cada método o función debe declarar cero o más argumentos formales especificando un tipo para cada uno.**
  - **Restricción de compatibilidad.**
    - **En cada asignación x = y, y en cada llamada a una rutina en que se usa y como argumento real para el argumento formal x, el tipo de la fuente y debe ser compatible con el tipo de destino x. La definición de compatibilidad se basa en la herencia, y es compatible con x si es descendiente de x (es lo que habíamos visto en el tema anterior como polimorfismo de inclusión).**
  - **Restricción de llamada a característica.**
    - **Para poder llamar a una característica (atributo o un método) f desde la clase X, f tiene que estar definida en X o en uno de sus antecesores.**

Tipificación
Comprobación de tipos

- **¿Cuál es el resultado de la ejecución del siguiente código?**

```
class Animal
{
  // El método ladra no está definido en Animal
}
class Perro extends Animal
{
 public void ladra() { ... }
}

...
Animal unAnimal = new Perro();
unAnimal.ladra();
...
```

Tipificación
Comprobación de tipos

- **Concepto de “pesimismo” en el tipado fuerte**
  - **Existen ciertas operaciones con los tipos que pueden ser válidas, pero si no hay una seguridad absoluta de que lo son entonces es mejor no permitirlas.**
  - **La idea subyacente es que es mejor detectar los posibles errores en tiempo de compilación y no en tiempo de ejecución (ya que son una mayor molestia para el usuario de la aplicación).**

- **Ejemplo de “pesimismo”**
  - **Al ejecutar unAnimal.ladra() si en unAnimal tenemos un Perro la cosa funcionaría.**
  - **Pero no hay nada que garantice que en unAnimal hay un perro ya que puede haber otro tipo de animales que no ladran (gatos, canarios, etc.)**

Tipificación
Comprobación de tipos

- **Soluciones al “pesimismo”**
  - **Hacer un type-cast explícito**
    - **La siguiente instrucción funcionaría sin problemas:**
      ((Perro) unAnimal).ladra()
    - **Como sabemos que hay un perro hacemos un type-cast explícito y llamamos al método ladra**
    - **Problema: si no hay un perro obtenemos un error de ejecución. El compilador se desentiende porque la responsabilidad de que los type-cast explícitos sea correcta es del programador**
  - **Usar funciones RTTI (Run-Time Type Information)**
    - **Nos podemos asegurar que en unAnimal hay un perro con sentencias como: if (unAnimal instanceof Perro) {...}.**
    - **Hay que tener cuidado porque este tipo de instrucciones suele dar lugar a código muy poco OO (ver ejemplo en lig. dinámica).**
  - **Maximizar el interfaz de Animal incluyendo el método ladra.**
    - **Los animales que no sean perros devolverían algo del estilo “no ladro”**
    - **Se trata de un compromiso entre la flexibilidad que da trabajar con superclases genéricas y la necesidad de que los métodos específicos no suban en la jerarquía de herencia hacia clases más generales (ver ejemplo en patrón composición).**

Tipificación
Comprobación de tipos

- **Ventajas del tipado fuerte:**
  - **Fiabilidad.**
    - **Permite detectar en tiempo de compilación errores que de otra manera sólo se manifestarían en tiempo de ejecución, y sólo en ciertas ejecuciones.**
    - **Es esencial detectar cuanto antes los errores, pues el coste de la corrección es mucho mayor cuanto más se demora la detección.**
  - **Legibilidad.**
    - **Declarar una entidad y una función con un cierto tipo es una potente forma de trasmitirle al lector del código información sobre lo que se intenta que dicho software haga. Esto lo apreciarán sobre todo los encargados de mantener ese software.**
  - **Eficiencia.**
    - **El tipado estático es el tipo más eficiente de tipado ya que permite conocer el tiempo de compilación el tipo exacto del elemento y efectuar las optimizaciones necesarias.**
    - **El tipado estricto de los lenguajes orientados a objetos no es tan eficiente pero permite restringir el conjunto de tipos posibles a unos pocos y, por tanto, generar código casi tan eficiente como en el tipado estático.**

Tipificación
Comprobación de tipos

- **Tipado dinámico**
  - **Todas las comprobaciones de tipos se realizan en tiempo de ejecución**
  - **Para tratar los posibles errores en tiempo de ejecución es necesario desarrollar sistemas que los manejen, como la utilización de excepciones.**
  - **Un ejemplo típico de un lenguaje con tipado dinámico es Smalltalk.**
  - **Principal ventaja: flexibilidad**
    - **Los partidarios del tipado dinámico se acogen frecuentemente a la idea de la flexibilidad para defenderlo.**
    - **Esto sería cierto para los lenguajes con tipado estático, pero los lenguajes con un tipado fuerte o estricto incluyen la suficiente flexibilidad para permitir que los tipos no sean una "camisa de fuerza" beneficiándose de la fiabilidad, legibilidad y eficiencia que aportan.**
  - **Tendencia actual:**
    - **Tal y como se muestra en los nuevos lenguajes aparecidos en los últimos años (Delphi, Java y C#), es incluir un tipado estricto.**
    - **Pero el tipado dinámico es popular en lenguajes de script**

Índice

7. Ligadura dinámica

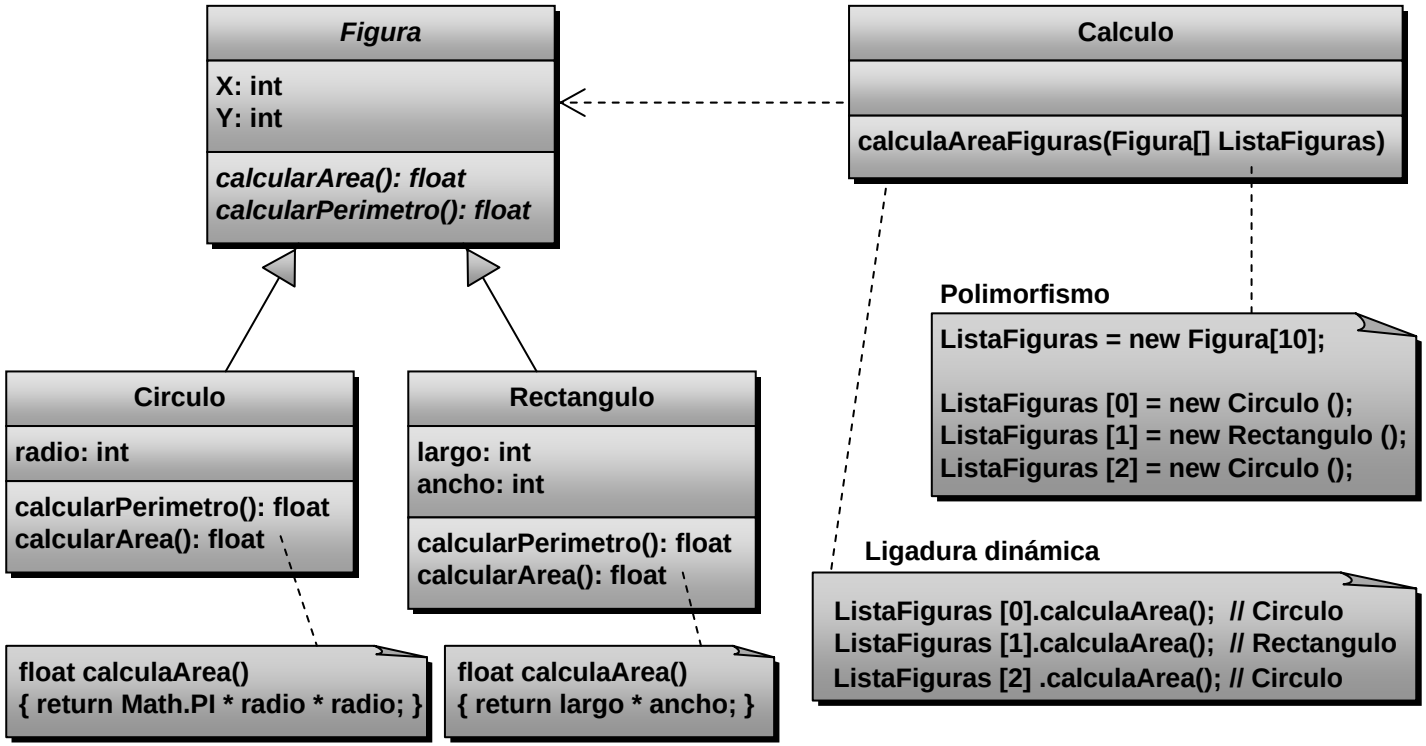
- Funcionamiento
- Implementación
- Eficiencia

Ligadura dinámica

- **Ligadura**
  - **La ligadura se encarga de ligar o relacionar la llamada a un método (mensaje) con el cuerpo del método que se ejecuta finalmente.**
  - **Existen dos tipos de ligadura: estática y dinámica**
  - **Ligadura estática (o temprana):**
    - **Consiste en realizar el proceso de ligadura en tiempo de compilación según el tipo declarado del objeto al que se manda el mensaje.**
    - **La utilizan (en Java) los métodos de clase y los métodos de instancia que son privados o final (ya que estos últimos no pueden ser sobreescritos).**
  - **Ligadura dinámica (o tardía):**
    - **Consiste en realizar el proceso de ligadura en tiempo de ejecución siendo la forma dinámica del objeto la que determina la versión del método a ejecutar.**
    - **Se utiliza en todos los métodos de instancia en Java que no son ni privados ni final.**

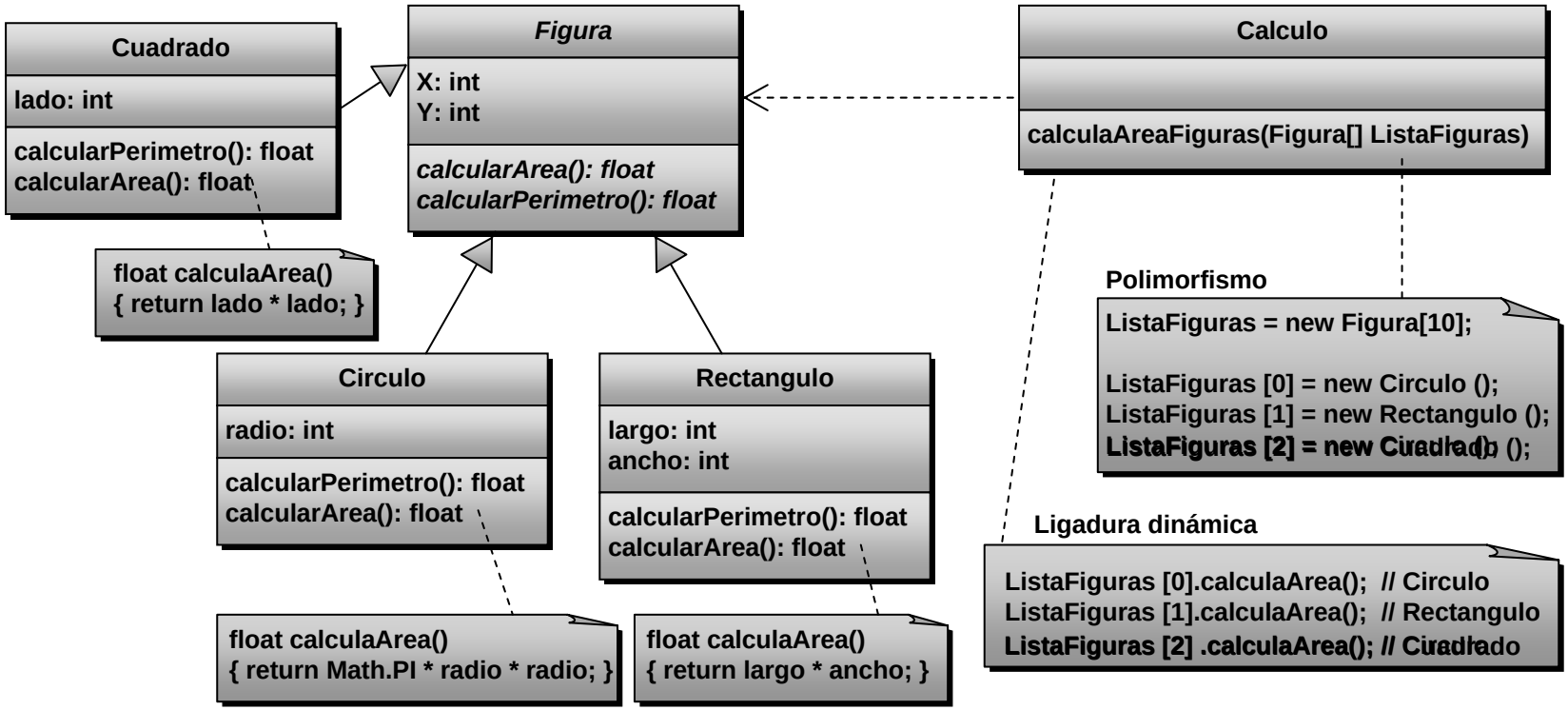
Propiedades básicas
Ligadura dinámica

- **Uso conjunto de herencia, polimorfismo y ligadura dinámica**



Propiedades básicas
Ligadura dinámica

- **Uso conjunto de herencia, polimorfismo y ligadura dinámica**



Propiedades básicas
Ligadura dinámica

- **Uso conjunto de herencia, polimorfismo y ligadura dinámica**
  - **Podemos escribir código que no haga referencia a las subclases concretas pero que utilice código de éstas (como el método calcularArea)**
  - **Al no hacer referencia a las subclases el código funcionará igual (sin necesidad de modificarlo) si creamos una nueva subclase (Cuadrado). Ver principio abierto-cerrado**
  - **En resumen:**
    - **Estamos creando código que será capaz de trabajar, sin modificarlo, con código futuro que aún no está escrito → Flexibilidad, Escalabilidad, Extensibilidad, etc.**
    - **Realmente no conocemos quién está haciendo en trabajo → Abstracción, Seguridad, Facilidad de mantenimiento, etc.**

Ligadura dinámica
Ejemplo

```
abstract class Personal
{
  protected long base;
  protected long anhos;
  private String nombre;

  final String getNombre()
  { return nombre; }

  final void setNombre (String s)
  { nombre = s; }

  long sueldo()
  { return base + (10000 * anhos);  }

  long pagaExtra ()
  { return base; }
}

class A extends Personal
{
  int numProyectos;
  long sueldo()
  { return (100000 * numProyectos);  }

  float rendimiento()
  { return numProyectos / anhos; }
}

class B extends Personal
{
  long extra;
  long sueldo()
  { return super.sueldo() + extra; }
}

class C extends Personal
{ ... }

class D extends Personal
{ ... }
```

Sobreescritura de reemplazo

Sobreescritura de refinamiento

```
...
  A PA = new A();
  B PB = new B();
  C PC = new C();
  D PD = new D();
  Personal [] P = new Personal[4];

  PA.setNombre("A");
  PA.base  = 100000;
  PA.anhos = 1;
  PA.numProyectos = 5;

  PB.setNombre("B");
  PB.base  = 100000;
  PB.anhos = 1;
  PB.extra = 50000;

  PC.setNombre("C");
  PC.base  = 100000;
  PC.anhos = 1;

  PD.setNombre("D");
  PD.base  = 100000;
  PD.anhos = 1;

  P[0] = PA;
  P[1] = PB;
  P[2] = PC;
  P[3] = PD;

  for (int i=0; i<=3; i++)
    System.out.println(
      P[i].getNombre()
      + " = " + P[i].sueldo());
...

  /* Resultados
  A = 500000
  B = 160000
  C = 110000
  D = 110000 */
```

Polimorfismo de inclusión

Ligadura dinámica

Ligadura dinámica
Funcionamiento

- **Empleo de funciones RTTI como alternativa:**

```
long sueldo()
{
  if (this instanceof A)
    return (100000 * ((A)this).numProyectos);
  else if (this instanceof B)
    return (base + (10000 * anhos)) + ((B)this).extra;
  else return base + (10000 * anhos);
}
```

  - **Funciona pero es un antipatrón OO**
  - **El código para calcular el sueldo del personal A no está en la clase A**
  - **Cada vez que se añada una nueva subclase hay que acordarse de modificar el if (ver principio abierto-cerrado)**
  - **Las funciones RTTI no son incorrectas pero sí es fácil que lleven a construcciones que son poco orientadas a objetos.**

Ligadura dinámica
Funcionamiento

- **¿Cuál es el resultado de la ejecución del siguiente código?**

```
class Padre
{
  int x = 5;

  static void f ()
  { System.out.println( "Padre.f" ); }
}

class Hijo extends Padre
{
  int x = 10;

  static void f ()
  { System.out.println( "Hijo.f" ); }

  public static void main (String [] args )
  { Hijo h = new Hijo();
    h.f();
    System.out.println(h.x);

    Padre p = new Hijo();
    p.f();
    System.out.println(p.x);
  }
}
```

Ligadura dinámica
Implementación

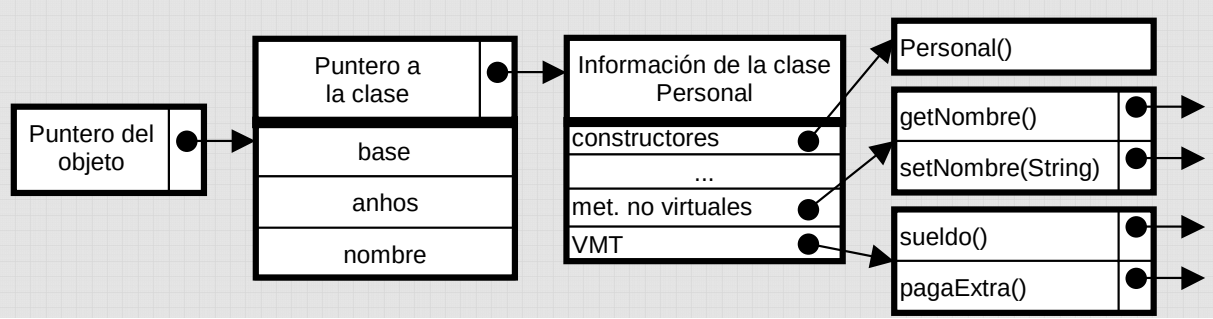
- **Características**
  - **Puede variar de un lenguaje a otro pero básicamente presentan unas características comunes.**
  - **Métodos que necesitan ligadura dinámica**
    - **Deben presentar ligadura dinámica sólo aquellos que pueden ser redefinidos.**
    - **Por ejemplo, en Java los métodos de clase y los métodos de instancia privados y/o finales no presentan ligadura dinámica.**
    - **En Java si no se especifica nada se entenderá que el método puede ser redefinido y por tanto debe presentar ligadura dinámica.**
    - **En otros lenguajes como Object Pascal o C++ los métodos por defecto presentan ligadura estática, si queremos que su ligadura sea dinámica es necesario incluirle la directiva virtual (por eso los métodos con ligadura dinámica se conocen como métodos virtuales). El método que sobreescribe de incluir la directiva *override***

Ligadura dinámica
Implementación

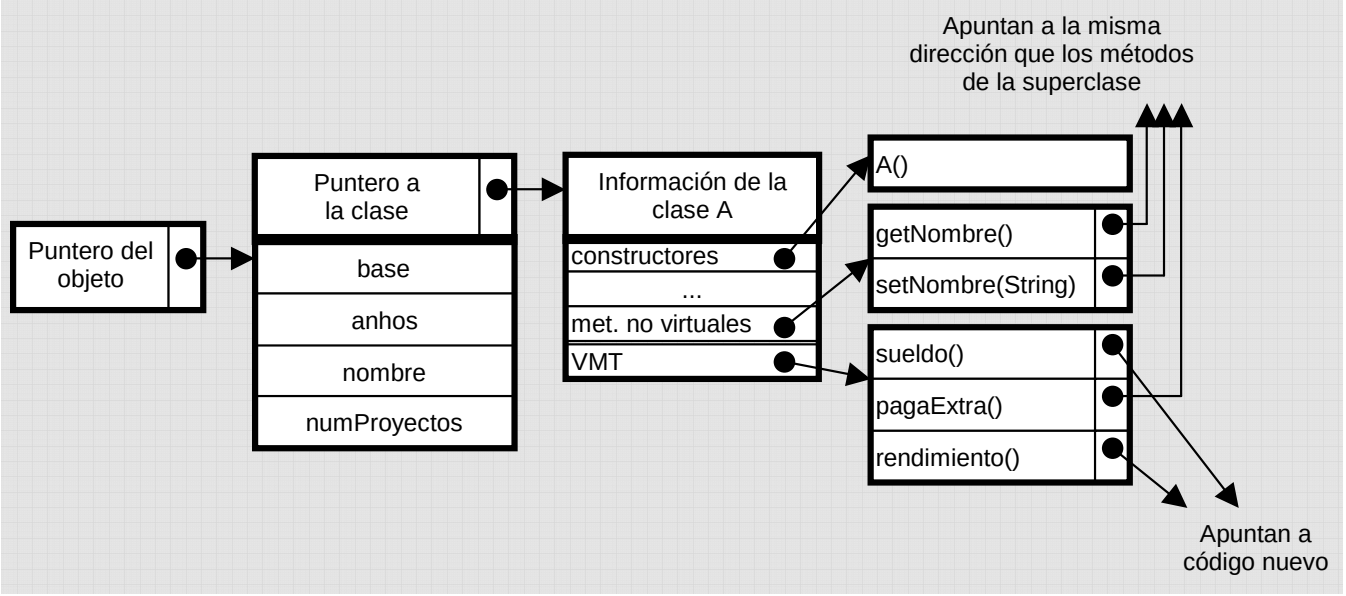
- **Características (cont.)**
  - **Tablas de métodos virtuales**
    - **La tabla que contiene la definición de la clase y que apunta al código de los métodos de instancia se divide en dos: una tabla para los métodos no virtuales y una tabla de métodos virtuales (VMT) que incluye una entrada por cada método virtual de la clase.**
    - **Los objetos de una misma clase comparten una misma VMT.**
    - **Una vez que a un método se le asigna una posición en la tabla virtual, esta posición no cambia en las VMTs de las clases derivadas.**
    - **Los punteros de los métodos que no son virtuales y los métodos virtuales que no han sido sobreescritos apuntan al código de la superclase. Por otro lado los métodos virtuales sobreescritos en la subclase y los métodos virtuales propios de dicha clase apuntan a código nuevo definido en la subclase.**

Ligadura dinámica
Implementación

- **Objeto de la clase Personal**



- **Objeto de la clase A**

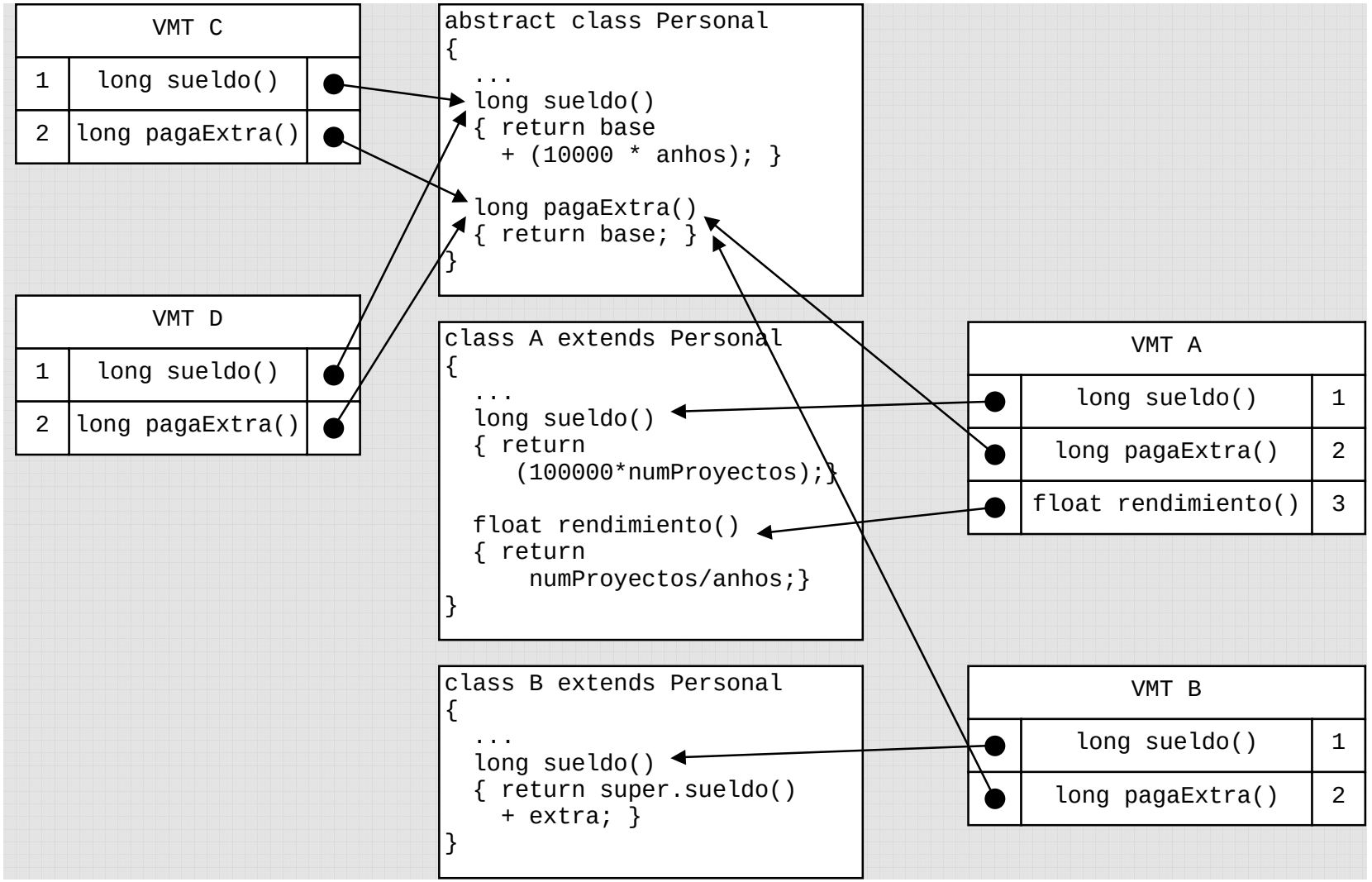


Ligadura dinámica
Implementación

- **Ejemplo de VMTs**
  - **Las VMTs de las clases C y D para los métodos sueldo() y pagaExtra() apuntarán al código incluido en la clase Personal.**
  - **En la clase A sólo la entrada de pagaExtra() apunta a Personal, ya que no ha sido redefinido, sin embargo la entrada de sueldo() apuntará a la versión del método incluida en la subclase A. Lo mismo para B.**
  - **Los métodos getNombre y setNombre no se incluyen en la tabla de métodos virtuales porque se han definido final.**
  - **El método rendimiento() definido en la clase A también debe ser incluido en la VMT ya que, aunque no es redefinido por ninguna clase, puede ser redefinido en un futuro (no es privado ni final).**

- **Proceso seguido en la llamada x.sueldo()**
  - **“Cogemos el puntero del objeto que apunta a la VMT y miramos en esa tabla la entrada 1 que corresponde al método sueldo(), utilizamos el puntero de esta tabla para saltar a la rutina correspondiente y la ejecutamos”.**

Ligadura dinámica
Implementación



Ligadura dinámica
Eficiencia

- **Principal inconveniente de la ligadura dinámica: la eficiencia.**
- **Llamadas a métodos virtuales**
  - **Ocupan más código que la llamada a un método estático y consume más tiempo**
  - **Aunque esta penalización es pequeña y está acotada por una constante (es independiente del nivel de herencia en el que nos encontremos)**

- **Llamadas a métodos no virtuales**
  - **Se puede saltar directamente al código de la función sin necesidad de acceder a una VMT.**
  - **Además también se puede transformar una llamada a un método por una llamada inline.**
    - **Una llamada inline lo que hace es eliminar la llamada al método (con su correspondiente paso de parámetros) y lo reemplaza por una copia de las instrucciones que existen en el cuerpo del método.**
    - **Las llamadas inline son por tanto muy rápidas pero penalizan el tamaño del código, por lo que un compilador inteligente sólo debería convertir en inline métodos pequeños.**

Ligadura dinámica
Eficiencia

- **Consecuencias de la perdida de eficiencia**
  - **En muchos lenguajes se limita el uso de la ligadura dinámica.**
  - **Sin embargo, limitar la capacidad de ligadura dinámica de las clases es limitar su reusabilidad.**
  - **Java permite la definición de métodos finales con dos objetivos:**
    - **Proteger el funcionamiento del método contra posibles sobreescrituras de las subclases.**
    - **Desactivar la ligadura dinámica porque se sabe a ciencia cierta que ese método no va a ser sobreescrito y de esta forma se genera código más eficiente.**
  - **En ambos casos en programador tiene que saber que está limitando la reutilización de la clase**
    - **Los métodos getNombre() y setNombre() se declaran finales porque no tiene sentido su sobreescritura.**
    - **Sin embargo, si aparece un nuevo tipo de personal definido por la subclase Robot, puede ser adecuado redefinir el método getNombre para devolver un número de serie, sin embargo al ser declarado final esto no es posible.**
    - **Los métodos final atentan contra el principio abierto-cerrado porque impiden redefiniciones no previstas de la clase (ver patrón método plantilla para ver un uso adecuado de métodos final).**

Ligadura dinámica
Eficiencia

- **Consecuencias de la perdida de eficiencia (cont.)**
  - **Enfoque deseable**
    - **Nunca un programador debe incluir ligadura estática por motivos de optimización.**
    - **Un compilador lo suficientemente inteligente podría detectar cuando no es necesaria la utilización de ligadura dinámica y optimizar el código consecuentemente.**
  - **Un caso más grave puede ocurrir en C++ u Object Pascal.**
    - **En estos lenguajes se utiliza la ligadura estática por defecto, y es responsabilidad del programador el indicar que los métodos deben usar ligadura dinámica mediante la sentencia virtual.**
    - **Por ejemplo si no indicamos que el método sueldo() es virtual, la llamada x.sueldo(), siendo Personal x = new A(), a pesar de contener una instancia de la clase A, ejecutará el método contenido en la clase Personal. Este funcionamiento es claramente erróneo pero es fácil que ocurra en estos entornos.**
  - **NOTA IMPORTANTE: La ligadura estática sólo es correcta cuando su efecto sea idéntico al de la ligadura dinámica.**

Bibliografía

- **Bibliografía fundamental**
  - Booch, G. “Análisis y diseño orientado a objetos, 2ª ed.”, Addison-Wesley / Díaz de Santos, Wilmington, Delaware, USA, 1996.
  - Budd, T. “Understanding object-oriented programming with Java”, Addison-Wesley, Reading, MA, 1998.
  - Cardelli, L., Wegner, P. “On understanding types, data abstraction, and polimorphism”, Computing Surveys, vol. 17, no. 4, 1985.
  - Graham, I. “Métodos orientados a objetos, 2ª ed.”, Addison-Wesley / Díaz de Santos, Wilmington, Delaware, USA, 1996.
  - Meyer, B. Construcción de software orientado a objetos, Prentice Hall, Madrid, 1999.
  - McLaughlin, B., Flanagan, D. “Java 1.5 Tiger: A Developers Notebook”, O’Reilly, Sebastopol, CA, 2004
- **Bibliografía complementaria**
  - Craig, I. The Interpretation of Object-Oriented Programming Languages, Springer-Verlag, London, 1999.
  - Meyer, B. “Overloading vs. object technology”, Journal of Object-Oriented Programming, Oct-Nov, 2001.
  - Marteens, I. La cara oculta de Delphi 4, Danysoft Internacional, Madrid, 1998.
- **Bibliografía en Internet**
  - Ellmer, E. “Basic Principles and Concepts of Object-Orientation (1993)”. URL: http://citeseer.nj.nec.com/ellmer93basic.html
  - Gosling, J., Joy, B., Steele, G., & Bracha, G. “The JavaTM Language Specification (2nd Edition)”. URL: http://java.sun.com/docs/books/jls/
  - Lindholm, T & Yellin, F. “The JavaTM Virtual Machine Specification (2nd ed.)”. URL: http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html