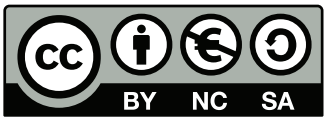
Programación y Diseño Orientado a Objetos

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM



Programación y Diseño Orientado a Objetos

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM

Índice

1 Conceptos básicos

1.1 Definiciones de conceptos

En esta primera sección del temario se introducirán conceptos relacionados con la programación orientada a objetos, al igual que sus definiciones y explicaciones. Los conceptos a tener en cuenta son los siguientes:

- **Objeto:** entidad perfectamente delimitada, queencapsula estado y funcionamiento y posee una identidad. Otra definición sería: elemento, unidad o entidad individual e identificable, real o abstracta, con un papel bien definido en el dominio del problema.
- **Clase:** la clase, entre otras cosas, actúa de molde o plantilla para la creación de objetos y crea un tipo de dato. En algunos lenguajes las clases son también objetos a todos los efectos. Los objetos creados a partir de una clase se denominan **instancias** de esa clase.
- **Identidad:** la identidad la define la posición de memoria. Independientemente de su estado, objetos distintos residirán en zonas de memoria distintas. Cada instancia tiene su propia identidad.
- **Estado y Comportamiento:** el **estado** de un objeto vendrá definido por los valores de sus atributos. Cada objeto tiene unazona de memoria propia para el almacenamiento de sus atributos. Los objetos exhiben textbfcomportamiento. Disponen de una serie de métodos (funciones o procedimientos) que pueden ser llamados/invocados.
- **Paradigma:** teoría o conjunto de teorías cuyo núcleo central se acepta sin cuestionar y que suministra la base y modelo para resolver problemas y avanzar en el conocimiento.
- **Paradigma de programación:** conjunto de reglas que indican como desarrollar software.
- **Base de la orientación a objetos:** se unen los datos y el procesamiento en entidades denominadas objetos.

En la **Programación Orientada a Objetos (POO)**, los objetos son las entidades que se manejan en el software. **Programar** consiste en modelar el problema mediante un universo dinámico de objetos. Cada objeto pertenece a una clase y tiene una responsabilidad en la aplicación. El **funcionamiento del programa** se consigue mediante el envío de mensajes a otros objetos para que realicen la ejecución de métodos. El objetivo es obtener **alta cohesión** y **bajo acoplamiento**.

El diseño de objetos no es una tarea sencilla, pero siempre se trata de que las clases cumplan las siguientes propiedades:

- Deben tener una **responsabilidad muy concreta**.
- Deben ser, en cierta medida, **autónomas**, es decir, tener poca dependencia de otras clases.
- Deben ser **introvertidas y no altruistas**, es decir, su estado solo puede modificarse desde la misma clase y no debe realizar funciones que ya estén implementadas aparte.

2 Clases, objetos y mensajes

3 Reutilización y polimorfismo

3.1 Mecanismos de reutilización

En la programación orientada a objetos existen diferentes mecanismos de reutilización de código, *herencia*, *interfaces* y *clases parametrizables*.

3.1.1

3.1.2 Definición y propiedades de la herencia

La herencia es un mecanismo que permite derivar nuevas clases (subclases o clases hija) a partir de clases existentes (superclase o clase padre).

Las clases padre e hija comparten un código común que es definido en la clase padre y que hereda la clase hija. Se reutiliza el código definido en la clase padre. Podemos hablar de reutilización de código cuando la clase hija hereda el comportamiento de la clase padre pero no modifica la forma de llevarlo a cabo. Hablamos de reutilización de concepto cuando la clase hija hereda el comportamiento de la clase padre y modifica la forma de llevarlo a cabo.

La clase hija es a la vez una extensión de la clase padre, desde el punto de vista de la clase como un módulo, y una especialización o restricción de la clase padre.

Una forma de saber si es conveniente utilizar herencia en cierto caso es mediante el *test de especialización*. Supongamos que tenemos una clase A padre y una clase B que pretendemos que sea hija de A. Si podemos decir que "B es un A" entonces podemos establecer una relación de herencia entre A y B.

3.1.3

3.1.4

3.1.5

3.1.6

3.1.7

3.1.8

3.1.9

3.1.10

3.1.11 El concepto de Interfaz: Java

FIXME: Qué es una Interfaz

Los métodos pueden implementarse en las interfaces. Se hace con dos métodos diferentes: `default` y `static`.

Los métodos `static` son código *estático* en la clase que no se redefine.

En Java las interfaces se utilizan para simular *herencia múltiple*.

```
1   interface A {
2     default void met1() {
3       System.out.println("A");
4     }
5   }
6
7   interface B {
8     default void met1() {
9       System.out.println("B");
10    }
11  }
12
13  class Prueba implements A,B {
14
15    // Tenemos que redefinir el método porque está
16    // implementado en las dos interfaces.
17    @Override
18    void met(1) {
19      B.super.met1();
20    }
21
22  }
23
24  public class Interaces {
```

```
25
26      public static void main(String[] args) {
27
28        B b = new Prueba();
29        b.met1();
30
31      }
32
33    }
```

Produce la salida:

```
1   run:
2   B
```

3.1.12 Simulando herencia múltiple

En Java

En Java se presenta un conflicto de nombres con los métodos default cuando una clase implementa varias interfaces o cuando una interfaz hereda de varias interfaces con miembros del mismo nombre. Para evitarlo podemos o bien proporcionar una nueva implementación, elegir una de las implementaciones que hay o crear un método abstracto.

En Ruby

Para simular herencia múltiple en Ruby hay una forma posible llamada comúnmente *mixin* que consiste en heredar de una clase e e incluir un módulo dentro de la clase.

3.1.13 Clases parametrizables

Una clase parametrizable presenta un alto grado de reutilización, pero tiene limitaciones. Encapsulan operaciones válidas para varios tipos de datos, generalizando los tipos y sus operaciones.

Su uso más frecuente es cuando sus atributos están formados por varios objetos del mismo tipo, siendo este un parámetro que tomará valor uando usemos la clase parametrizable. En Ruby no existe este concepto.

```
1   public class Tienda<T> {
2
3       String nombre;
4       ArrayList<T> stock;
5       float ganancias;
6
7       Tienda (String n) {
8
```

```
 9       nombre=n;
10       stock = new ArrayList();
11       ganancia=0;
12     }
13
14     public void comprarAProveedr(T objeto, float precioCoste)
          {
15
16       stock.add(objeto);
17       ganancias = ganancias-precioCoste;
18
19     }
20
21     public void venderACliente(T objeto, float pvp) {
22
23       stock.remove(objeto);
24       ganancias=ganancias+pvp;
25
26     }
27
28     public float getGanancias() {
29       return ganancias
30     }
31
32   }
```

4 Polimorfismo

4.1 Definición de polimorfismo

Se define como la capacidad de una entidad (indentificador) de referenciar a objetos de diferentes clases (tipo) durante la ejecución de un programa.

4.2 Tipo estático y dinámico

- **Tipo estático:** Tipo(clase) del que se declara la variable, no cambia durante la ejecución.
- **Tipo dinámico:** Clase a la que pertenece el objeto al que va referenciando una variable a lo largo de la ejecución Puede cambiar durante la ejecución.

Java es un lenguaje con tipo estático y dinámico. Dado que Ruby no es un

lenguaje fuertemente tipado, no tiene sentido hablar de tipo estático, todos los tipos son dinámicos. Un ejemplo es el siguiente. Supongamos que tenemos una clase Persona y una clase Alumno, la cual hereda de persona.

```
1 Persona p = new Persona();
```

El tipo estático de *p* es Persona, su tipo dinámico es Persona. No obstante, como Alumno hereda de Persona, podemos cambiar su tipo dinámico a Alumno:

```
1 p = new Alumno();
```

Ahora el tipo dinámico de *p* es Alumno.

4.3 Polimorfismo y ligadura dinámica

- **Ligadura estática:** el enlace del método al mensaje se basa en el tipo estático de la variable y se realiza en tiempo de compilación. Se debe especificar el C++ y Objetive C. Java y Ruby no permiten hacer esto.
- **Ligadura dinámica:** el enlace del método al mensaje se basa en el tipo dinámico de la variable y se realiza en tiempo de ejecución. Es lo que emplean Java y Ruby.

No existe polimorfismo sin ligadura dinámica, ya que ésta permite que una variable referencia en tiempo de ejecución a objetos de diferente clase, así como que un mensaje se ligue a un método y otro dependiendo de la clase del objeto receptor en ese momento.

En los lenguajes con tipo estático, las clases de objetos que puede referenciar una variable está limitada por el tipo estático de la variable. Existen unas reglas de compatibilidad en orientación a objetos:

- **Entre clases (a través de herencia):** el tipo dinámico de una variable puede ser la clase declarada, coincidiendo con su tipo estático, pero también puede ser de alguna de sus subclases, como veíamos en el ejemplo de la Persona. Podemos aplicar la regla *es un*, *un alumno es una persona*, luego está permitido referenciar a Alumno.
- **Entre interfaz y clases (a través de la realización):** el tipo dinámico de una variable puede ser el tipo estático de la interfaz que implementa. En este caso solo abordamos Java, ya que Ruby no cuenta con el concepto de interfaz.

Daremos un par de conceptos más antes de pasar a ilustrar ejemplos.

4.4 Errores de compilación y ejecución

- **Error de compilación:** aunque la decisión del método a ejecutar dependa del objeto al que se haga referencia, si el tipo estático no dispone del método, obtendremos un error en tiempo de compilación. Podemos solucionarlo

mediante un casting, con el que no convertimos el tipo, pero indicamos el tipo dinámico que se espera tener en tiempo de ejecución.

- **Error de ejecución:** se produce cuando no se corresponde el tipo dinámico de la variable con el tipo que se le ha indicado para evitar un error de compilación.

4.5 Ejemplos prácticos y realizando casting

Con todos estos conceptos teóricos expuestos, ponemos unos ejemplos que ayudarán a entender los conceptos. Disponemos de una clase Persona, de la cual heredan una clase Alumno y una clase Profesor. El código de las clases es el siguiente: *Clase Persona*

```
1  package polimorfismo;
2
3  /**
4    * @author victor
5   */
6  public class Persona {
7      protected String nombre;
8      Persona (String n) {
9          nombre = n;
10     }
11     void hablar() {
12         System.out.println("Soy una persona que habla. Me
               llamo " + nombre);
13     }
14
15 }
```

Clase Alumno

```
1  package polimorfismo;
2
3  /**
4   *
5   * @author victor
6   */
7  public class Alumno extends Persona {
8      Alumno(String n) {
9          super(n);
10     }
11
12     @Override
13     void hablar() {
```

```
14          System.out.println("Soy un alumno que habla, me llamo
                " + nombre);
15      }
16
17      void Estudiar() {
18          System.out.println("Soy un alumno que estudia");
19      }
20
21  }
```

Clase Profesor

```
1   package polimorfismo;
2
3   /**
4    *
5    * @author victor
6    */
7   public class Profesor extends Persona {
8       Profesor(String n) {
9           super(n);
10      }
11
12      @Override
13      void hablar() {
14          System.out.println("Soy un profesor que habla, me
                llamo " + nombre);
15      }
16
17      void Enseñar() {
18          System.out.println("Soy un profesor que enseña");
19      }
20
21  }
```

Creamos una clase Test para probar el código anterior.

```
1   public class Test {
2       public static void main(String args[]) {
3           Persona p; //Tipo estático de p: Persona
4           p = new Persona("Pepe"); //Tipo dinámico de p: Persona
5           p.hablar();
6   }
```

La ligadura dinámica del método hablar() de p se realiza en tiempo de ejecución. Por tanto, durante la ejecución se decide qué método hablar se va a

utilizar. Como el tipo dinámico es Persona, se llamará al método hablar() que se implementó en la clase Persona. Se produce la siguiente salida

```
1 Soy una persona que habla. Me llamo Pepe
```

Si cambiamos el tipo dinámico a Alumno, se decide en tiempo de ejecución que el método hablar() que se utilizará es el que se implementó en la clase.

```
1 p = new Alumno("Pepe");
2 p.hablar()
```

Produce como salida:

```
1 Soy un alumno que habla, me llamo Pepe
```

Esto se puede hacer ya que ambas clases tienen implementado el método hablar. Para no producir un error en tiempo de de compilación, el método debe estar implementado en la clase del tipo estático con el que hemos declarado la variable p. Aunque el tipo dinámico sea Alumno, como Persona no dispone del método Estudiar(), el siguiente código produce un error de compilación.

```
1 Persona p = new Alumno("Pepe");
2 p.Estudiar();
```

4.6 Resolviendo errores con casting

Para resolver el problema anterior podemos realizar un casting, indicarle al compilador el tipo dinámico esperado durante la ejecución. Es importante tener en cuenta que un casting no convierte tipos, simplemente indica explícitamente el tipo dinámico esperado.

```
1 Persona p = new Alumno("Pepe");
2 ((Alumno)p).Estudiar();
```

Compila y produce como salida:

```
1 Soy un alumno que estudia
```

No obstante, debemos tener cuidado con el casting que realizamos. El siguiente código compilaría, pero obtendríamos un error de ejecución ya que aunque nos quitemos el error de compilación, cuando ejecutemos, nuestro tipo dinámico es Profesor, y un Profesor no puede ser un Alumno, por lo que no se puede realizar el cast.

```
1 Persona var4 = new Profesor("Pedro");
2 ((Alumno)var4).Estudiar();
```

El error obtenido es este:

```
Exception in thread "main" java.lang.ClassCastException:
   polimorfismo.Profesor cannot be cast to polimorfismo.Alumno
```

Nótese que el tipo estático era Persona. Si ahora decidimos que el tipo estático es Profesor, obtenemos un error de compilación. NetBeans nos notifica con el siguiente error:

```
incompatible types: Profesor cannot be converted to Alumno
```

Para finalizar, vamos a ver dos o tres ejemplos más, complentado así el listado de casos enrevesados a los que nos podemos enfrentar en Java. El siguiente código produce un error en compilación.

```
Persona persona = new Persona("Pepe");
Alumno alumno = new Alumno("Juan");
alumno = persona;
```

Una persona no es un alumno, por lo que no podemos tener como tipo estático Alumno y como tipo dinámico Persona. Si fuera al revés y hubiéramos hecho `persona = alumno` sería correcto, ya que un Alumno es una Persona, Tendríamos como tipo estático Persona y como tipo dinámico, Alumno. No hay problema si hubiéramos referenciado persona a Alumno en vez de a Persona.

Sin embargo, en este código lo que obtenemos es un error en ejecución

```
Persona persona = new Persona("Pepe");
Alumno alumno = new Alumno("Juan");
alumno = (Alumno) persona;
```

El compilador no se queja, ya que estamos indicando que el tipo dinámico que recibirá la variable *alumno* es del tipo Persona. Pero al igual que nos ocurría antes, al ejecutar, obtenemos una excepción, ya que una Persona no es un Alumno, por lo que no se puede realizar el cast de Persona a Alumno.

Veamos ahora este caso:

```
Persona persona = new Alumno("Pepe");
Alumno alumno = new Alumno("Juan");
alumno = persona;
```

No compila, Persona no se puede convertir en alumno, aunque el tipo dinámico de persona sea alumno. Podemos realizar un casting, consiguiendo que el código compile y ejecute correctamente.

```
Persona persona = new Alumno("Pepe");
Alumno alumno = new Alumno("Juan");
alumno = (Alumno) persona;
```

Los ejemplos, aunque cortos, han sido densos, poniendo de manifiesto las diferencias entre tipos estáticos y dinámicos, así como su respectiva relación con la compilación y ejecución

4.7 Polimorfismo en lenguajes sin tipo estático

En aquellos lenguajes sin tipo estático, como Ruby, no hya una limitación en cuanto a la "forma" que puede tomar una variable, es decir, puede referenciar a cualquier objeto de cualquier clase durante la ejecución. La variable responderá al mensaje que reciba en función de la clase del objeto que se esté referenciando en ese momento

4.8 Qué no es Polimorfismo

La sobrecarga de operadores, funciones o métodos y los tipos y clases parametrizables no son considerados polimorfismos.