UT07. 03-POO UTILIZACIÓN AVANZADA DE CLASES

Programación de 1DAW

C.I.F.P. Carlos III - Cartagena

Sobreescritura vs sobrecarga

- **Sobreescritura**
  - La subclase vuelve a implementar el método heredado, es decir, permite modificar el comportamiento de la clase padre.
  - Tiene que tener el **mismo nombre**.
  - El **retorno** de la clase padre e hijo deberá ser del mismo tipo.
  - Deberá conservar la **misma lista de argumentos** que el método de la clase padre.

Sobreescritura vs sobrecarga

```
public class Pajaro{
   protected String nombre;
   protected String color;

   public String getDetalles() {
      return "Nombre: " + nombre + "\n" + "Color: " + color;
   }
}

public class Loro extends Pajaro{
   protected String pedigri;

   public String getDetalles(){
      return "Nombre: " + nombre + "\n" + "Color: " + color + "\n" + "Pedigrí: " + pedigri;
   }
}
```

La clase Loro, que hereda de la clase Pajaro, sobreescribe el método getDetalles()—devuelve el mismo tipo de datos (String) y sin argumentos en ambos casos.

Sobreescritura vs sobrecarga

- **Sobrecarga**
  - Es la implementación del **mismo método** con ligeras diferencias adaptadas a las distintas necesidades de dicho método
  - Deben **cambiar la lista de argumentos obligatoriamente**
  - Puede estar sobrecargado **en una subclase o en la clase**
  - Se pueden utilizar las **mismas excepciones** o **añadir** algunas
  - Pueden **cambiar el tipo de retorno** o el **modificador de acceso**
  - **Ejemplo:** constructores sobrecargados

Conversiones entre objetos

- **Casting**. Tiene que haber una relación de herencia.

```
public static void m(Empleado e){
    System.out.println(e.getNombre());
}
  public static void main(String[] args){
    Persona p1;
    p1 = new Empleado();
    p1.setNombre("Isaac Sánchez");
    Empleado enc1 = new Encargado();
    enc1.setNombre("Andrés Rosique");
    m(enc1);
    m((Empleado)p1);
    Persona p2 = new Persona();
    p2.setNombre("Juan Serrano");
    m((Empleado)p2);
```

Método estático m(), que visualiza el nombre del empleado

m(p1) sin casting daría error de compilación

esta llamada dará error de ejecución, no tiene el método.

Acceso a métodos de la superclase

```
public class Padre{
  protected int dato;
  public void m(){
    System.out.println("Método clase padre");
  }
}
public class Hijo extends Padre{
  private int dato;
  public void m(){
    System.out.println("Método clase hijo");
    super.dato = 10;
    dato = 20;
  }
  public void getDato(){
    System.out.println(super.dato);
  }
  public void mostrar(){
    this.m();
    m();
    super.m();
  }
}
```

```
public class Test{
  public static void main(String[] args){
    Hijo h = new Hijo();
    h.mostrar();
    h.getDato();
  }
}
```

Resultado en pantalla:

Método clase hijo

Método clase hijo

Método clase padre

10

Acceso a métodos de la superclase

```
public class Padre{
        protected int dato1, dato2;

        public Padre(int x, int y){
          dato1=x;
          dato2=y;
        }
        public Padre(){
          this(5,5);
        }
    }
```

```
public class Test{
        public static void main(String[] args){
            Hijo h1 = new Hijo(1,1);
            h1.getDato();
            Hijo h2 = new Hijo();
            h2.getDato();
          }
      }
```

```
public class Hijo extends Padre{
        private int dato1, dato2;

        public Hijo(int x, int y){
          super(2,2);
          dato1=x;
          dato2=y;
        }
        public Hijo(){
          dato1=3;
          dato2=3;
        }
        public void getDato(){
          System.out.println("Padre dato1: " + super.dato1);
          System.out.println("Padre dato2: " + super.dato2);
         System.out.println("Hijo dato1: " + this.dato1);
          System.out.println("Hijo dato2: " + this.dato2);
        }
      }
```

Acceso a métodos de la superclase

Resultado

Padre dato1: 2
Padre dato2: 2
Hijo dato1: 1
Hijo dato2: 1
Padre dato1: 5
Padre dato2: 5
Hijo dato1: 3
Hijo dato2: 3

Acceso a métodos de la superclase

- Uso del operador **instaceof**
- Uso de los métodos **getClass()**, **getSuperclass()** y **newInstance()**

Interfaces

- Define el **comportamiento** y la **estructura** mediante la declaración de métodos no estáticos y campos estáticos finales.
- Los elementos son públicos por definición: no es necesario poner **public**.
- Los atributos son finales sin necesidad de poner **final** ➔ poner un valor inicial
- Puede ser **implementada por cualquier clase**, sin tener en cuenta la herencia.
- Una clase que implementa un interfaz tiene que sobrecargar todos y cada uno de los métodos del interfaz, o se define como **abstract**.
- Una clase con **todos** los métodos abstractos.
- Las **constantes** de la interfaz se pueden usar en cualquier parte del código de la clase simplemente con el nombre.
- En otras clases, importando la interfaz y haciendo referencia de la siguiente forma: **NombreInterfaz.NombreCte**
- **Uso**: para **definir un comportamiento común** a clases sin relación de herencia. **Herencia múltiple**
- **NO** variables, **SOLO** métodos.

Interfaces

- **¿Una clase puede heredar de varias clases,** aunque sean abstractas?
- ¿Podría **implementar una o varias interfaces** y además seguir heredando de una clase?
- **Una interfaz no puede definir métodos (no implementa su contenido),** tan solo los declara o enumera.
- **Una interfaz puede hacer que dos clases tengan un mismo comportamiento** independientemente de sus ubicaciones en una determinada jerarquía de clases (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).
- **Una interfaz permite establecer un comportamiento de clase sin apenas dar detalles,** pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la interfaz).
- **Las interfaces tienen su propia jerarquía,** diferente e independiente de la jerarquía de clases.

Clase abstracta vs. Interfaces

- **Clase abstracta proporciona una interfaz (comportamiento) disponible sólo a través de la herencia.** Sólo quien herede de esa clase abstracta dispondrá de esa interfaz (comportamiento). Si una clase no pertenece a esa misma jerarquía (no hereda de ella) no podrá tener esa interfaz.
- Sin embargo, **una interfaz sí puede ser implementada por cualquier clase,** permitiendo que clases que no tengan ninguna relación entre sí (pertenecen a distintas jerarquías) puedan compartir un determinado comportamiento (una interfaz) sin tener que forzar una relación de herencia que no existe entre ellas.

Recomendación de uso

- Si sólo vas a proporcionar una lista de **métodos abstractos (interfaz),** sin definiciones de métodos ni atributos de objeto ➔**interfaz** antes que **clase abstracta.**
- **Al definir clase base,** puedes comenzar declarándola como **interfaz** y sólo cuando veas que necesitas implementar métodos o definir variables miembro, puedes entonces convertirla en **clase abstracta** (no instanciable) o incluso en una **clase instanciable** (concreta)

Interfaces

- Sintaxis del interface.

```
<modificador_acceso> interface <nombre_interface>{
    //declaración de métodos abstractos y públicos
}
```

- Sintaxis de las constantes.

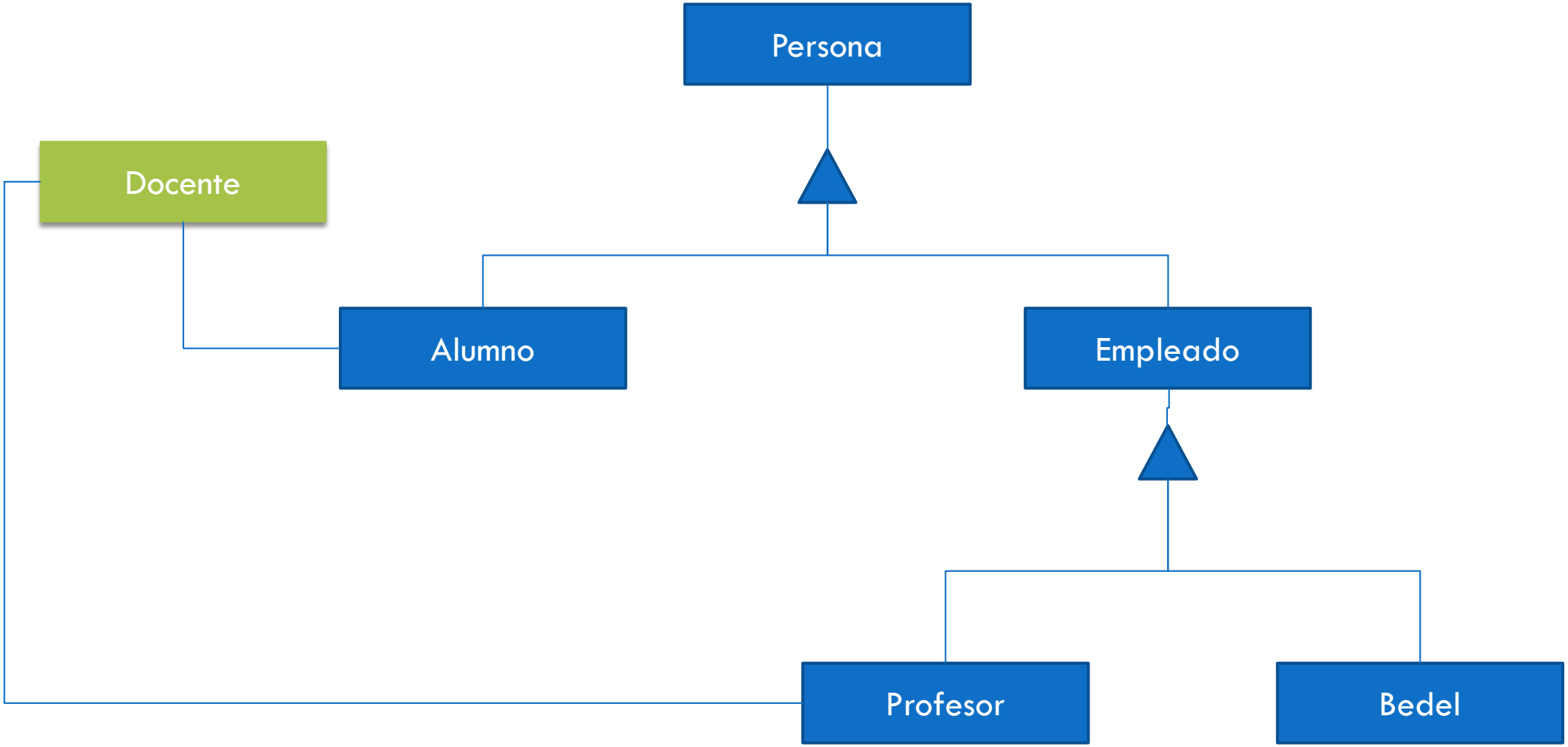
```
public static final <tipo_cte> <nombre_cte> =<valor_cte>;
```

- Sintaxis de la clase con comportamiento de interfaz

```
[public] [final | abstract] class <nombre-clase> [extends
<nombre_superclase>] [implements <lista_interfaces>]
```

Ejemplo

- Aplicación universitaria. Serán docentes los alumnos y los profesores, pero no los bedeles ni otros empleados. Los docentes se distinguen de otras personas en que tendrán un grupo y un horario.
- Definir una interface de nombre **Docente** con operaciones **ponGrupo**(), **dameGrupo**(), **dameHorario**(). Para los no-docentes estas operaciones no tienen sentido.
- Los alumnos, profesores, bedeles, son Personas.


Persona
Docente
Alumno
Empleado
Profesor
Bedel

```
interface Docente{
    void ponGrupo(String grupo, Horario horario);
    String dameGrupo();
    Horario dameHorario();
}
```

```
class Alumno extends Persona implements Docente{
    private String grupo;
    private Horario horario;

    public void ponGrupo(String grupo, Horario horario){
        this.grupo = grupo;
        this.horario = horario;
    }
    public String dameGrupo(){
        return grupo;
    }
    public Horario dameHorario(){
        return horario;
    }
}
```

```
class Profesor extends Empleado implements Docente{
    private String grupo;
    private boolean esMatutino;

    public void ponGrupo(String grupo, Horario horario){
        this.grupo = grupo;
        esMatutino = (horario == Horario.MAÑANA);
    }
    public String dameGrupo(){
        return grupo;
    }
    public Horario dameHorario(){
    ....
    }
}
```

```
class Bedel extends Empleado{
    ...
}
```

```
abstract  class Empleado extends Persona{
    ...
}
```

Interfaces

- Polimorfismo con interface.
- Polimorfismo ligado al concepto de herencia.
- ¿Se puede usar polimorfismo con interfaces? **SÍ**
- Condiciones:
  - El tipo de la variable polimórfica es el propio interface.

    InterfaceA variable;
  - Los objetos que hagan referencia la variable polimórfica, tienen que implementarlo.
  - Una variable polimórfica (variable definida de tipo **InterfazA**) solo puede invocar métodos del interfaz.