Conceptos básicos de herencia

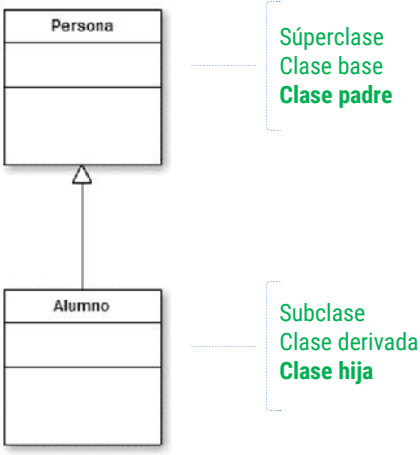
Prof. Mg. Rafael Mellado S.
EII147 – Introducción a las tecnologías de información
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso

Herencia

- Es la derivación de una clase a partir de otra existente.
- El objetivo es la reutilización del software desarrollado.

Súperclases y subclases

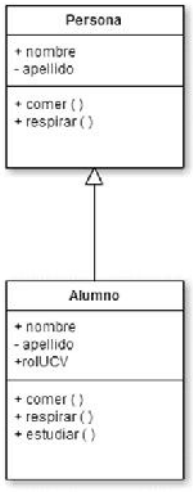
- La clase de la cual se deriva otra clase se denomina clase base, súperclase o **clase padre**.
- Una clase derivada de una súperclase recibe también el nombre de subclase o **clase hija**.
- La herencia aplica en relaciones de naturaleza "*B es un tipo de A*".



219

Súperclases y subclases

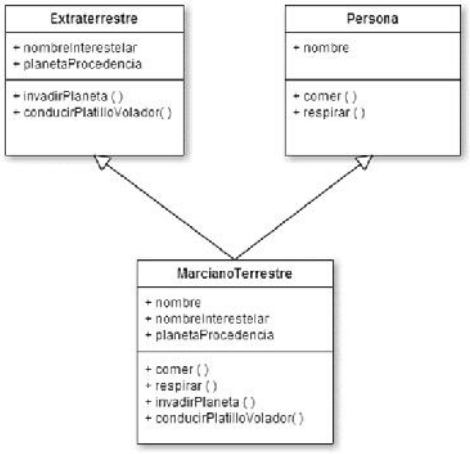
- La subclases heredan propiedades de su súperclase.
- Una subclase, respecto de su súperclase:
  - Agrega nuevas propiedades
  - Modifica propiedades heredadas.
- Una clase es subclase de una única súperclase.



220

Herencia múltiple

- Una clase es subclase de más de una súperclase.
- Java **no soporta** la herencia múltiple.



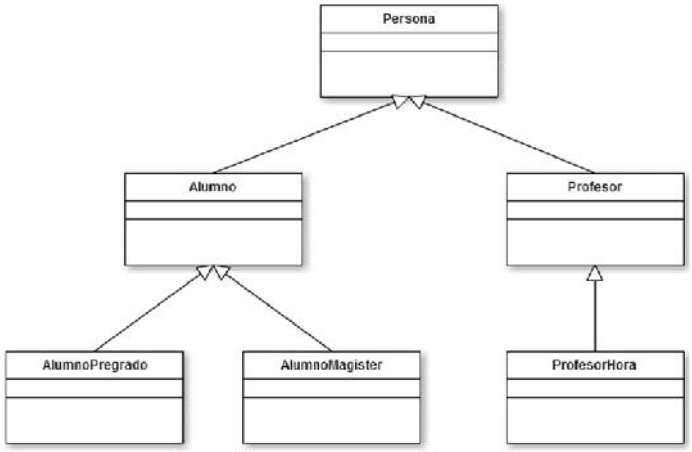
221

Jerarquías de herencia

- La herencia organiza clases bases y derivadas en jerarquías de clases.
- Se heredan los métodos y atributos públicos.
- La organización gráfica (UML) es como se muestra a continuación:

222

Jerarquías de herencia



223

Ejemplos

– Determinar si en las siguientes situaciones existe una relación de herencia entre las clases (indicadas en negrita):

Caso	¿Existe herencia?
Todo **Electrodoméstico** se enciende y apaga. El **Horno microondas** tiene además abre y cierra su puerta.	Si
Los **Bienes raíces** tienen un Rol de identificación. Una **Casa** tiene también un Jefe de hogar y un **Negocio** tiene una Patente comercial	Si
Un **Camión** tiene Patente. Un **Conductor** tiene un camión conducido.	No

224

Ejemplos

- Determinar si en las siguientes situaciones existe una relación de herencia entre las clases (indicadas en negrita):

Caso	¿Existe herencia?
Los **Archivos Multimediales** pueden ser **Imágenes** o **Música**. Las imágenes pueden ser a **Color** o **Blanco y Negro**.	Si
Un **Avión** tiene **Fuselaje**, **Alas** y **Motores**.	No
Una **Gaviota** vuela. Una **Abeja** vuela y además tiene aguijón.	No

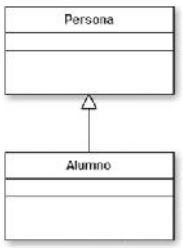
225

Implementación de jerarquías de herencia

- Para crear una clase hija a partir de una clase padre, en Java la clase hija debe declararse:

```
public class NombreSubclase extends NombreSuperclase
```

- Ejemplo:

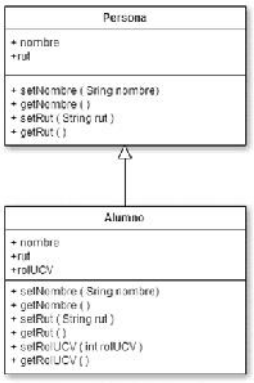



```
public class Alumno extends Persona
```

226

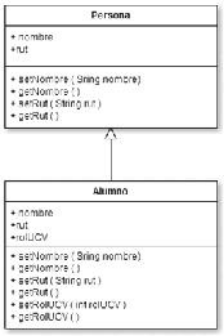
Implementación de jerarquías de herencia

- Implementar las clases Persona y Alumno, de acuerdo con lo siguiente:



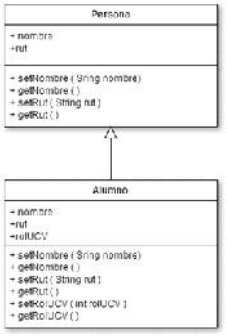
Implementación de jerarquías de herencia

```
public class Persona
{
    public String rut;
    public String nombre;

    public Persona()
    {
        rut = "00000000-0";
        nombre = null;
    }

    public void setRut(String r){
        rut = r;
    }

    public String getRut(){
        return rut;
    }

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

    public String getNombre(){
        return nombre;
    }
}
```



Implementación de jerarquías de herencia

```
public class Alumno extends Persona
{
   public String rolUCV;

   public Alumno() {
       rolUCV = "000000-0";
   }

   public void setRolUCV(String r){
       rolUCV = r;
   }

   public String getRolUCV(){
       return rolUCV;
   }

   public String quiénSoy(){
      return rut + nombre + rolUCV;
   }
}
```



229

Miembros heredados

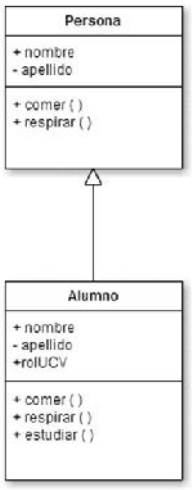
- Una subclase hereda de su súperclase (por el momento):
  - Variables de instancia públicas
  - Métodos públicos
- Todos los anteriores pueden ser utilizados en la subclase “como si hubieran sido declarados en ella”.

230

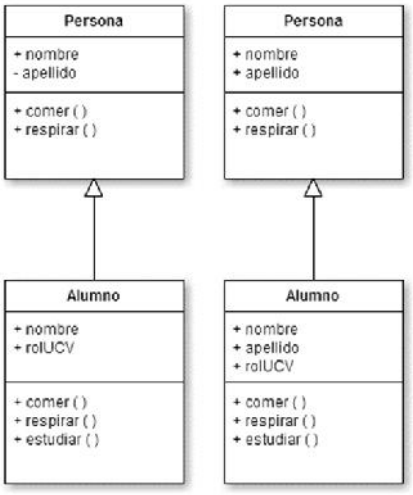
Miembros heredados

El atributo ***apellido*** al ser privado no se hereda

Esto está erroneo



Maneras correctas:



231

Ejercicio

- Implemente las clases Vehículo, Autobús y Camión, dados los siguientes antecedentes:
  - Todo Vehículo tiene patente y marca.
  - Los Autobuses y los Camiones son Vehículos.
  - Todo Autobús tiene cantidad de asientos.
  - Todo Camión tiene carga en toneladas.

232

Consideraciones sobre herencia

Prof. Mg. Rafael Mellado S.
EII147 – Introducción a las tecnologías de información
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso

Recordando

- Una subclase hereda de su clase padre (por el momento):
  - Variables de instancia públicas
  - Métodos públicos
- Todos los anteriores pueden ser utilizados en la clase hija "como si hubieran sido declarados en ella".

Miembros no heredados

- Una subclase no hereda:
  - Propiedades privadas
  - Constructores
- Los miembros no heredados no pueden aparecer en el código de la clase hija.

235

Miembros no heredados

```
public class Persona
{
    private String rut;
    private String nombre;

    public Persona()
    {
        rut = "00000000-0";
        nombre = null;
    }

    public void setRut(String r){
        rut = r;
    }

    public String getRut(){
        return rut;
    }

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

    public String getNombre(){
        return nombre;
    }
}
```

236

Miembros no heredados

```
public class Alumno extends Persona
{
   public String rolUCV;

   public Alumno() {
       rolUCV = "000000-0";
   }

   public void setRolUCV(String r){
       rolUCV = r;
   }

   public String getRolUCV(){
       return rolUCV;
   }

   public String quiénSoy(){
      return rut + nombre + rolUCV;
   }
}
```

Error (E.C): no pueden ser accesadas directamente

237

Miembros no heredados

- Las variables privadas **no son heredadas**, por lo que no pueden aparecer en el código de la clase hija.
- Sin embargo se puede hacer uso indirecto de ellas en la clase hija, a través de los métodos públicos de manipulación implementados en la respectiva súperclase.
- En una clase hija es también posible referenciar al constructor de la clase padre.

238

Miembros no heredados

```
public class Alumno extends Persona
{
   public String rolUCV;

   public Alumno() {
       rolUCV = "000000-0";
   }

   public void setRolUCV(String r){
       rolUCV = r;
   }

   public String getRolUCV(){
       return rolUCV;
   }

   public String quiénSoy(){
      return getRut() + getNombre() + rolUCV;
   }
}
```

Forma correcta

239

Miembros no heredados: Constructores

- Los constructores no son heredados, por lo que cada clase hija debe tener su(s) propio(s) constructor(es).
- Sin embargo en los constructores se puede invocar al constructor de la clase padre con la instrucción:

```
super( lista parámetros )
```

- **Importante**: La instrucción super debe ser la primera instrucción del constructor.

240

Miembros no heredados: Constructores

```java
public class Persona
{
    public String rut;
    public String nombre;

    public Persona(String rut, String nombre)
    {
        this.rut=rut;
        this.nombre=nombre;
    }

    public void setRut(String r){
        rut = r;
    }

    public String getRut(){
        return rut;
    }
}
```

```java
public class Alumno extends Persona
{
    private String rolUCV;

    public Alumno(String nombre, String rut, String rolUCV)
    {
        super(nombre, rut);
        this.rolUCV = rolUCV;
    }

    public void setRolUCV(String r){
        rolUCV = r;
    }

    public String getRolUCV(){
        return rolUCV;
    }

    public String quiénSoy(){
        return getRut() + rolUCV;
    }
}
```

241

Herencia y constructores

- Toda clase hija debe incluir una referencia ***super*** a algún constructor de la superclase.
- Si no se incluye la referencia ***super***, Java incluye automáticamente una referencia al constructor sin parámetros de la superclase. Es decir incluye:

```java
super();
```

- Notar que se produce un **error** cuando no existe un constructor sin parámetros en la superclase y se omite la referencia super en la subclase.

242

Herencia y constructores

- Dos clases equivalentes:

```
public class Alumno extends Persona
{
      private String rolUCV;

      public Alumno()
      {
            super();
            rolUCV = null;
      }
}
```

```
public class Alumno extends Persona
{
      private String rolUCV;

      public Alumno()
      {
            rolUCV = null;
      }
}
```

243

Herencia y constructores

```
public class Persona
{
    private String rut;
    private String nombre;

    public Persona(String rut, String nombre)
    {
        this.rut=rut;
        this.nombre=nombre;
    }

    public void setRut(String r){
        rut = r;
    }

    public String getRut(){
        return rut;
    }

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

    public String getNombre(){
        return nombre;
    }
}
```

244

Herencia y constructores

```
public class Alumno extends Persona
{
    private String rolUCV;

    public Alumno(String rut, String nombre, String rolUCV)
    {
        super(rut, nombre);
        this.rolUCV=rolUCV;
    }

    public void setRolUCV(String r){
        rolUCV = r;
    }

    public String getRolUCV(){
        return rolUCV;
    }

    public String quiénSoy(){
        return getRut()+getNombre()+ rolUCV;
    }
}
```

245

Herencia y constructores: Cuidado!!

```
public class Alumno extends Persona
{
    private String rolUCV;

    public Alumno(String rolUCV)
    {
        this.rolUCV=rolUCV;
    }

    public void setRolUCV(String r){
        rolUCV = r;
    }

    public String getRolUCV(){
        return rolUCV;
    }

    public String quiénSoy(){
        return getRut()+getNombre()+ rolUCV;
    }
}
```

Java incluirá automáticamente una referencia `super()`, pero invocará a un constructor inexistente en la clase padre. El equivalente sería:

```
public Alumno(String rolUCV)
{
    super();
    this.rolUCV=rolUCV;
}
```

246

Herencia y constructores

- En Java toda clase extiende otra clase.
- Las clases que no declaran extender a otras, extienden a la clase **Object**. Donde **Object** es la superclase (directa o indirecta) de todas las clases.
- Todas las clases son subclases de Object.
- Por lo tanto: todos los constructores incluyen (explícitamente o no) una referencia al constructor de su superclase.

247

Ejercicio

- Un videojuego tiene Personajes. Cada personaje tiene un nombre (String) y un nivel propio de energía (int). Además implementan el método alimentarse, que recibe por parámetro una cantidad de energía (int) con el que incrementa el nivel propio de energía. Los personajes pueden ser:
  - Guerreros: tienen además un arma (String). Al momento de la instanciación reciben su nombre, arma y nivel propio de energía inicial. Los guerreros tienen un método combatir que recibe por parámetro la cantidad de energía a gastar en el ataque, la cual es descontada de su nivel propio de energía. El método combatir retorna el arma y la cantidad de energía del ataque concatenados.

248

Ejercicio

- Magos: tienen además un poder (String). Al momento de la instanciación reciben su nombre y poder. Los magos son siempre creados con un nivel propio de energía igual a 100. Proveen un método encantar, que disminuye en 2 unidades el nivel propio de energía y que retorna el poder del mago.

249

Aplicaciones y herencia

EII147-01-02 Introducción a las tecnologías de información
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso
Rafael Mellado Silva
rafael.mellado@ucv.cl

Sobreescritura de métodos

- Un método declarado e implementado en una súperclase puede ser reimplementado en una subclase. Esto se denomina **sobreescritura de métodos**.
- Conceptualmente significa que la subclase realiza la operación de la súper clase, pero de un modo distinto.
- Esto es un caso de polimorfismo.

Sobreescritura de métodos

```
public class Persona {
 private String rut;
 private String nombre;
 public String getRut(){
    return rut; }
 public String getNombre(){
    return nombre; }
 public String identificarse(){
    return rut + nombre;
 }
 //otros miembros de la clase…
}
```

```
public class Alumno extends
                Persona {

  private String carrera;

 public String identificarse(){
    return getRut() + getNombre()
           + carrera;
 }

 //otros miembros de la clase…
}
```

3

Sobreescritura de métodos

```
public class Persona {
 private String rut, nombre;
 public Persona(String r, String n){
    rut = r; nombre = n;
 }
 public String getRut(){ return
rut;}
 public String getNombre(){
    return nombre;}
 public String identificarse(){
    return rut + nombre;
 }
}
```

```
public class Alumno extends
                Persona {
  private String carrera;
  public Alumno(String r, String
                n, String c){
    super(r,n); carrera = c;
  }
  public String identificarse(){
     return getRut() + getNombre()
            + carrera;
  }
}
```

```
Persona a = new Persona("100", "Matías");
System.out.println( a.identificarse() );
```

→ 100Matías

```
Alumno a = new Alumno("100", "Matías", "Ind");
System.out.println( a.identificarse() );
```

→ 100MatíasInd

4

Sobreescritura de métodos

```
public class Persona {
 private String rut, nombre;
 public Persona(String r, String n){
    rut = r; nombre = n;
 }
 public String getRut(){ return
rut;}
 public String getNombre(){
    return nombre;}
 public String identificarse(){
    return rut + nombre;
 }
}
```

```
public class Alumno extends
                    Persona {
  private String carrera;
  public Alumno(String r, String
                      n, String c){
    super(r,n); carrera = c;
  }
  public String identificarse(){
     return getRut() + getNombre()
            + carrera;
  }
}
```

```
Persona a = new Alumno( "100", "Matías", "Ind");
System.out.println( a.identificarse() );
```

→ 100MatíasInd

Java resuelve en tiempo de ejecución la asociación entre la variable y el método que debe invocar, en función del objeto que se encuentre referenciado en ella.

5

Sobreescritura de métodos

- El compilador Java es responsable de verificar que el método pertenezca al tipo de dato declarado por la variable.
- El intérprete Java es responsable de identificar y ejecutar la implementación del método correspondiente al tipo de objeto referenciado en el momento por la variable.

```
Persona a = new Alumno( "100", "Matías", "Ind");
System.out.println( a.identificarse() );
```

6

Reconocimiento de clases: Instanceof

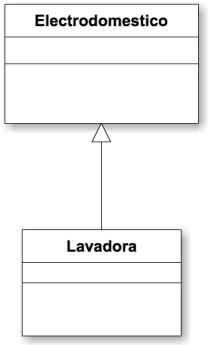
- El operador instanceof permite reconocer la clase a la que pertenece un objeto referenciado desde una variable determinada.
- Formato:
  - NombreVar **instanceof** NombreClase
- Ejemplo:

```
if( pers instanceof Persona )
   System.out.println( "La variable pers referencia a una Persona" );
else
   System.out.println( "La variable pers no referencia a una Persona" );
```

7

Reconocimiento de clases: Instanceof

- Todo objeto es instancia de la clase a la que pertenece, como también instancia de su superclase.

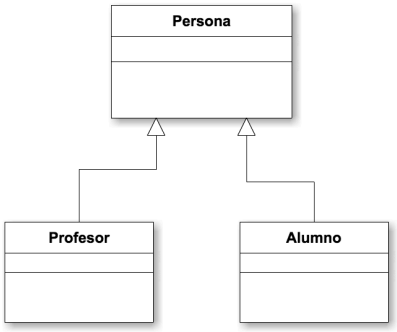


Un objeto de la clase lavadora es también un electrodoméstico

8

Reconocimiento de clases: Instanceof

- Suponer:




```
Persona p1 =null:
Profesor p2 = new Profesor();
Alumno p3 = new Alumno();
Persona p4 = new Alumno();

if( p1 instanceof Persona )  --> false
if( p2 instanceof Profesor ) --> true
if( p3 instanceof Alumno ) --> true
if( p2 instanceof Alumno ) --> false
if( p3 instanceof Persona ) --> true
if( p2 instanceof Persona ) --> true
if( p4 instanceof Persona ) --> ??
if( p4 instanceof Alumno ) --> ??
if( p4 instanceof Profesor ) --> ??
```

9

Acceso a miembros de una clase

```
public class Persona {
  private String rut;
  private String nombre;
  public setDatos(String r, String n) {
     rut = r;
     nombre = n; }
  public String getRut(){
     return rut; }
  public String getNombre(){
     return nombre; }
}
```

```
public class Alumno extends Persona {

   private String carrera;

   public String matricularse( String c){
      carrera = c;
   }

}
```

```
Persona  a = new Persona();
a.setDatos(  "1000-2",  "Luis");

Alumno  b = new Alumno();
b.setDatos(  "2000-3",  "Pamela"  );
b.matricularse(  "Industrial"  );
```

Correcto :D

10

Acceso a miembros de una clase

```
public class Persona {
  private String rut;
  private String nombre;
  public setDatos(String r, String n) {
     rut = r;
     nombre = n; }
  public String getRut(){
     return rut; }
  public String getNombre(){
     return nombre; }
}
```

```
public class Alumno extends Persona {

   private String carrera;

   public String matricularse( String c){
      carrera = c;
   }

}
```

```
Persona  c = new Alumno();
c.setDatos(  "1000-2",  "Luis");
c.matricularse(  "Comercial"  );
```

Error: el compilador determina que este método no pertenece a Persona. Sin embargo la variable contiene referencia a un Alumno, que es un tipo de Persona (y que posee este método).

11

Acceso a miembros de una clase

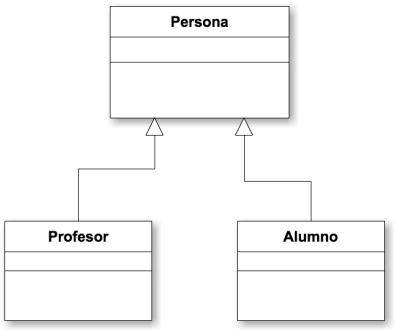
- Es necesario introducir un casting para indicar al compilador que el objeto referenciado en la variable es efectivamente una instancia de Alumno:

```
Persona  c = new Alumno();
c.setDatos(  "1000-2",  "Luis");
(  (Alumno)c  ).matricularse(  "Industrial"  );
```

12

Acceso a miembros de una clase

- El casting no convierte objetos, simplemente explicita el tipo de objeto referenciado en una variable.




```
Persona a = new Profesor();
Profesor b = (Profesor) a;   // OK
Alumno c = (Alumno) a;    // ERROR

Persona d = new Persona();
Profesor e = (Profesor) d; // ERROR
Alumno f = (Alumno) d;   // ERROR

Alumno g = new Alumno();
Profesor h = (Profesor) g; // ERROR
```

13

Miembros protegidos de una clase

- El modificador de visibilidad **protected**, permite declarar visibilidad "protegida" en variables de instancia y métodos.
- Los miembros de una clase con visibilidad protegida son sólo accesibles desde la misma clase o desde cualquier subclase de ella (no son accesibles desde otras clases).
- Por lo tanto, una subclase hereda de su superclase:
  - Variables de instancia protegidas y públicas
  - Métodos protegidos y públicos

14

Miembros protegidos de una clase

```
public class Persona {

 protected String rut;
 protected String nombre;

 public Persona(String r, String n) {
    rut = r;
    nombre = n;
 }
 public void setRut(String r){
    rut = r; }
 public String getRut(){
    return rut; }
 public void setNombre(String n){
    nombre = n; }
 public String getNombre(){
    return nombre; }

}
```

```
public class Alumno extends Persona {

  private String rolUCV;

  public Alumno() {
     super( "000000-0", "N/A" );
     rolUCV = "000000-0";
   }

public String quiénSoy(){
     return rut + nombre
            + rolUCV;
  }
}
```

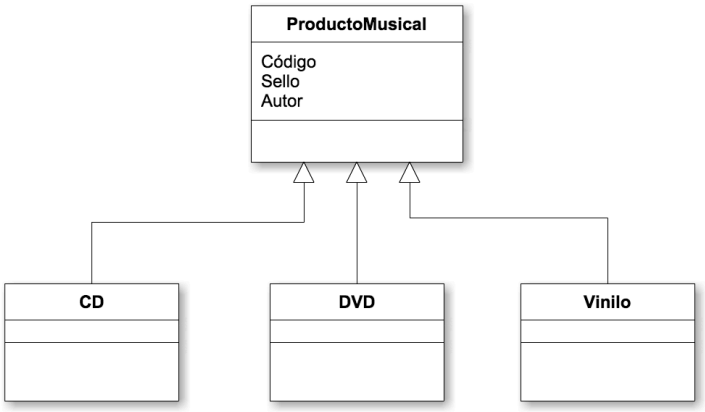
15

Identificación de superclases

- Contexto: se está desarrollando una aplicación que trabaja con CD's, DVD's y discos de vinilo.
- Problema: se establece que a pesar de tener sus propios atributos, todos ellos disponen de código, sello discográfico y autor. Se desea evitar duplicidad de código.
- Decisión: se determina la conveniencia de crear la clase "ProductoMusical", que agrupa las propiedades comunes de los tres tipos de productos.

16

Identificación de superclases



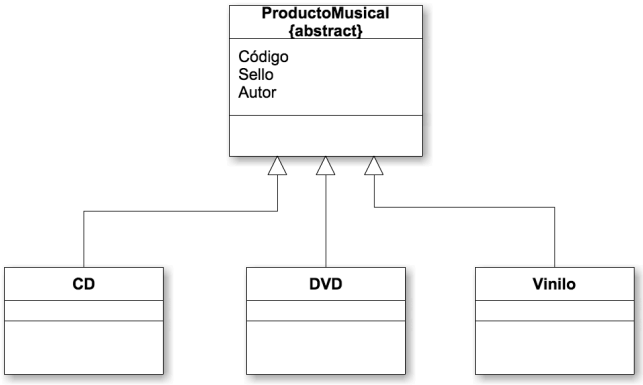
17

Clases abstractas

- En el ejemplo anterior la clase ProductoMusical es **abstracta** (no representa entidades presentes en el dominio).
- Esta condición se explicita en el diseño, declarando la clase como **abstracta**.
- Una clase abstracta no puede ser instanciada, ha sido diseñada sólo para ser extendida.

18

Clases abstractas

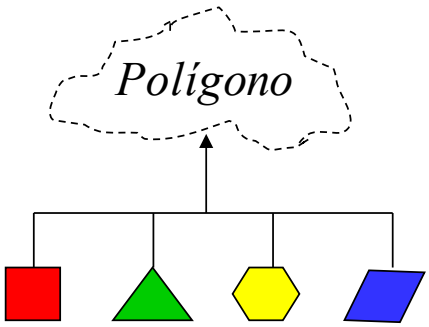



```
public abstract class ProductoMusical {

 private int código;
 private String sello;

 ...

}
```

19

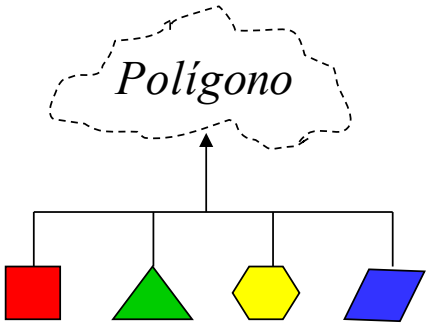
Clases abstractas: ejemplo

- Un software trabaja con distintas figuras geométricas, todas ellas polígonos, con algunas propiedades en común (ej. cantidad de lados).



20

Clases abstractas: ejemplo




```
public abstract class Poligono {
      protected int lados;
      public void setLados(int l){
            lados = l;
      }
      ...
}
```

```
public class Cuadrado extends Poligono {
      private int longitud;
      public void setLongitud(double l) {
            longitud = l;
      }
      ...
}
```

21

Métodos abstractos

- Supongamos que en el ejemplo anterior todos los polígonos deben proveer un método de cálculo de área.
- Conflicto de "fuerzas" en diseño:
  - Todos los polígonos deben proveer el método, por lo tanto debiese "aparecer" a nivel de la superclase Polígono.
  - La operación del método depende de cada polígono concreto (ej. área de cuagrado: lado2, área de triángulo base * altura /2, etc.), por lo tanto no puede establecerse una lógica común a nivel de superclase.
- Solución: declarar método como ***abstracto*** en la superclase.

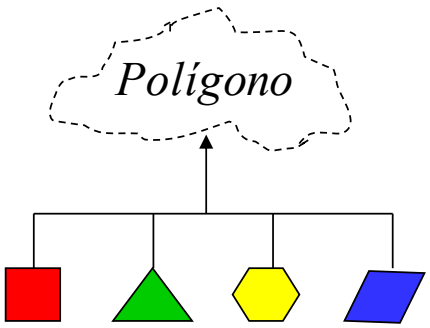
22

Métodos abstractos

- Un método abstracto es un método que se declara en una superclase, pero sin proveer implementación.
- La implementación de un método abstracto se difiere para sus subclases.
- Una clase que declara uno o más métodos abstractos es necesariamente abstracta (y debe ser declarada como tal).
- Si una subclase no provee implementación para un método abstracto que hereda de su superclase, es necesariamente abstracta (y debe ser declarada como tal).

23

Métodos abstractos




```
public abstract class Poligono {
      protected int lados;
      public int setLados(int l){
            lados = l;
      }
      public abstract double getArea();
            ...
}
```

```
public class Cuadrado extends Poligono {
      private double longitud;
      public void setLongitud(double l) {
            longitud = l;
      }
      public double getArea(){
            return longitud*longitud;
      }
      ...
}
```

24

Métodos abstractos

- Si una superclase declara algún método abstracto, entonces:
  - Las subclases concretas deben implementarlo.
  - Las variables de referencia declaradas del tipo de la superclase pueden recibir invocaciones al método abstracto, sin necesidad de casting. Ejemplo:

```
Polígono figura = new
Triángulo();
double sup = figura.getArea();
```

getArea() fue declarado como método abstracto en Polígono.

25

Ejercicios

- Una compañía de teléfonos organiza sus clientes de prepago (rut, nombre, tarifa) y postpago (rut, nombre, dirección, precioPlan, minutosDisponibles) en un arreglo de tamaño 5000:
  - Genere la declaración de las clases
  - Considere un arreglo tipo clase Cliente (aplicando herencia) en el cual se guardarán los distintos clientes. Use plibre.
  - Entregue la cantidad de clientes de prepago existentes
  - Entregue el precio del plan más caro y el más barato.

26

Ejercicios

- Ambos clientes deben poder facturar, lo cual para se solicita rehacer el ejercicio aplicando clase abstracta:
  - Prepago: corresponde a recibir la cantidad de minutos hablados, multiplicarlos por la tarifa y descontarlos del saldo.
  - PostPago: corresponde a recibir la cantidad de minutos hablados y descontarlos del total de minutos disponibles.
  - ¿Cuál es el rut del cliente postpago que más sobrepasó la cantidad de minutos disponibles?

27

Aplicaciones y herencia

EII147-01-02 Introducción a las tecnologías de información
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso
Rafael Mellado Silva
rafael.mellado@ucv.cl