



Grado en Ingeniería Informática  
Grado en Matemáticas e Informática



Asignatura: PROGRAMACIÓN II

# Herencia y Polimorfismo

Ángel Lucas González Martínez  
Jaime Ramírez Rodríguez  
Guillermo Román

DLSIIS - E.T.S. de Ingenieros Informáticos  
Universidad Politécnica de Madrid

Octubre 2013

# Introducción

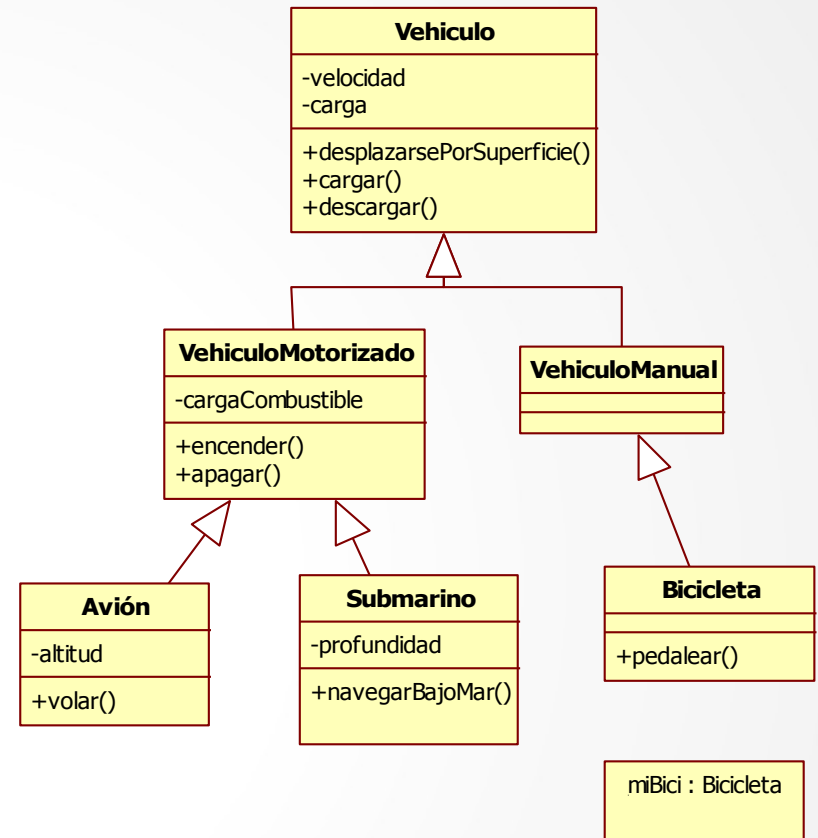
- Supongamos que queremos crear una clase que incluya código de otra clase.
  - ➔ Por ejemplo, tenemos una clase Persona y queremos crear una clase Trabajador
- El mecanismo de **herencia** nos permite hacer que la clase Trabajador **derive** de la clase Persona sin tener que reescribir (ini depurar!) algunos métodos.
- En este caso se dice que:
  - ➔ La clase Trabajador es una **subclase** o **clase hija** de la clase Persona
  - ➔ La clase Persona es una **superclase** o **clase padre** de la clase Trabajador

# Clases como conjuntos

- Una clase la podemos ver como un conjunto, y por lo tanto los objetos de esa clase los podemos ver como miembros de ese conjunto.
- Una subclase o clase hija de una clase (padre) A, la podemos considerar un subconjunto del conjunto A.
  - ➔ De esta forma los elementos que pertenecen a un subconjunto pertenecen también al conjunto más general

# Generalización de clases

- Una clase hija **es una correcta subclase** de una clase padre si cumple las siguientes reglas:
- Regla Es-un:** *todos los objetos (miembros) de una clase hija (subconjunto) son objetos también de la clase padre (conjunto).*
- Regla del 100%:** *el 100% de la definición de la clase padre debe ser parte de la definición de la clase hija, es decir:*
  - ➡ *La clase hija debe incorporar al menos los atributos de la clase padre*
  - ➡ *La clase hija debe saber realizar al menos las operaciones de la clase padre ⇒ la debe poder sustituir en cualquier contexto*



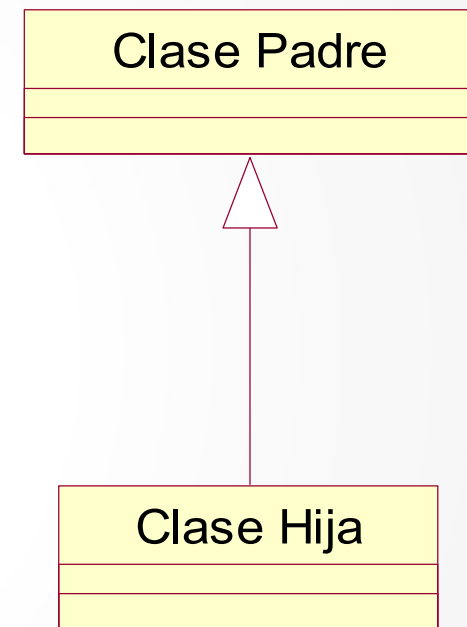
# Herencia de clases

- Es un **mecanismo que incorporan los lenguajes OO** que permite implementar una clase (derivada) a partir de otra clase (base) de forma que:
  - ➔ La clase derivada incluye todos los atributos de la clase base
  - ➔ La clase derivada incluye todos los métodos definidos en la clase base.
- Es un mecanismo para reutilizar código ya existente que permite evitar la duplicidad de código
- USO CORRECTO: sólo se debe usar cuando la clase derivada sea conceptualmente una subclase de la clase base.

# Herencia de clases en Java

- Java permite definir una clase como clase derivada o subclase de una clase padre.

```
class ClaseHija extends ClasePadre  
{  
    .....  
}
```

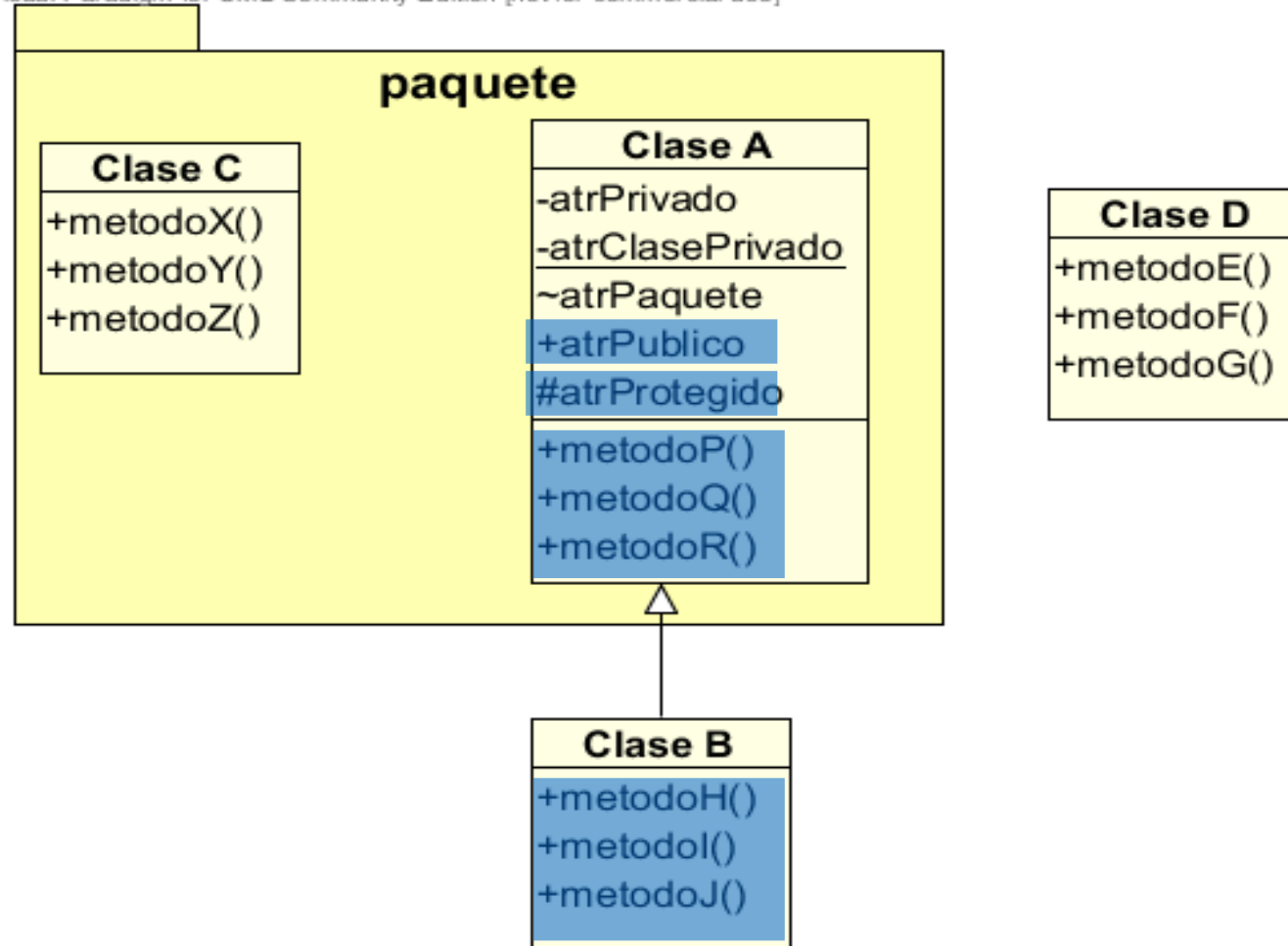


# Herencia en Java

- No permite la herencia múltiple (**sólo un padre**)
- Todas las clases derivan implícitamente de **Object**
- Una clase derivada (o subclase) puede acceder directamente a todos los atributos y métodos heredados sean ***public o protected***
- Se permite la especialización o redefinición de servicios de la clase padre (**sobreescribir /redefinir**), por ejemplo, el equals().

# Ocultación entre módulos

Visual Paradigm for UML Community Edition [not for commercial use]





# Constructores y Herencia

- Cuando se declara un objeto de una clase derivada, se ejecutan los constructores siguiendo el orden de derivación, es decir, primero el de la clase base (padre), y después los de las clases derivadas (hijas) de arriba a abajo.
- Para pasar parámetros al constructor de la clase padre:  
***super*** (*para1, para2, ..., paraN*)
- Si el padre no tiene ni constructor por defecto ni un constructor sin parámetros hay que llamar a ***super*** obligatoriamente

# Constructores y Herencia

```
public class Persona {
    private String nombre;
    private int edad;
    public Persona() {}
    public Persona (String nombre, int edad) {
        this.nombre = nombre; this.edad = edad;
    }
}

public class Alumno extends Persona {
    private int curso;
    private String nivelAcademico;
    public Alumno (String nombre, int edad, int curso, String nivel) {
        super(nombre, edad);
        this.curso = curso; nivelAcademico = nivel;
    }
    public static void main(String[] args) {
        Alumno a = new Alumno("Pepe", 1, 2, "bueno");
    }
}
```

# Referencias a objetos de clases hijas

- Si tenemos *ClaseHijo* *hijo* = **new** *ClaseHijo*(...);
- Entonces es posible *padre*=*hijo* (**upcasting**) donde *padre* es una variable de tipo *ClasePadre*
  - ➔ No es posible *hijo*=*padre*
  - ➔ Es posible con casting *hijo*= (*ClaseHijo*) *padre* (**downcasting**). Si *padre* no contiene una instancia de *ClaseHijo* se produce la excepción:

**java.lang.ClassCastException**

- Ahora bien:
  - ➔ Con *padre* sólo podemos acceder a atributos y métodos de la clase padre

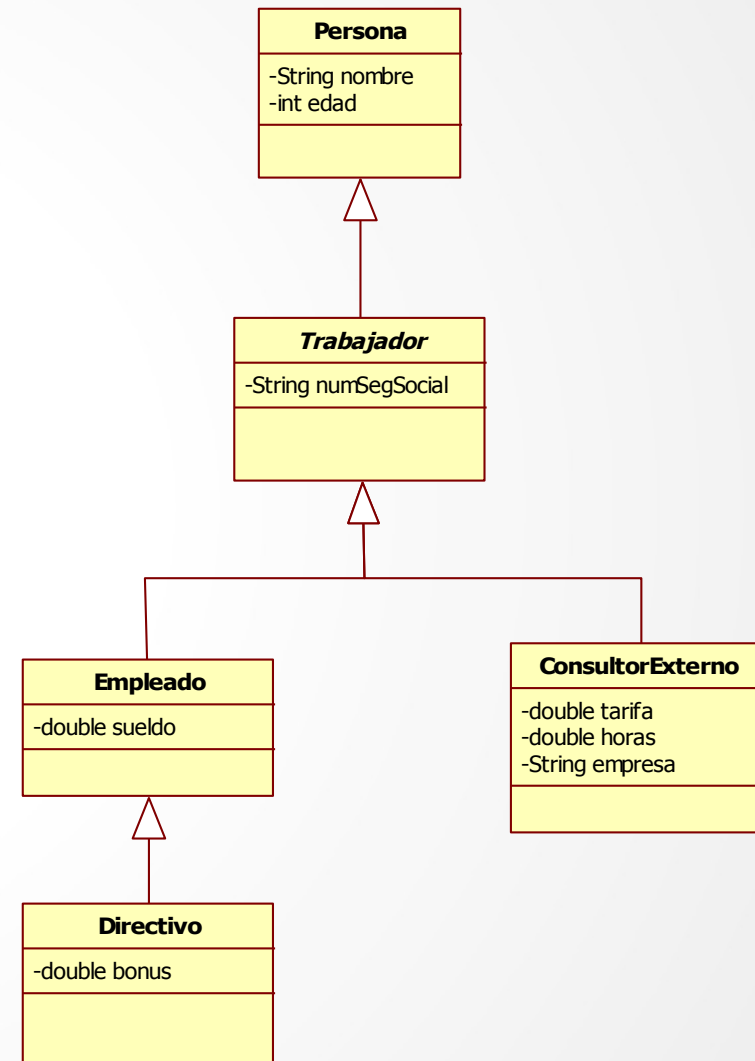
# Referencias a objetos de clases hijas

```
public static void main(String[] args) {  
    Persona persona;  
    Alumno alumno = new Alumno("pepe",23,1,"universitario");  
  
    persona = alumno; /* upcasting: ref. padre (persona) señala  
                        ahora al objeto hijo (alumno) */  
  
    persona.setEdad(24); /* acceso al objeto hijo (alumno)  
                        mediante la ref. padre (persona) */  
  
    /* Pero no es posible acceder directamente a un miembro  
       de la clase hija usando una ref. a la clase padre */  
    persona.setCurso(88); // ERROR  
    Alumno alumno1 = (Alumno) persona;  
    alumno1.setCurso(88); // Esto es correcto (downcasting)  
}
```

# Ejercicio

- Crear una clase **Persona** con nombre(String) y edad(int)
  - ➔ Implementar el constructor que inicialice el nombre y la edad con parámetros de entrada
- Crear una clase **Trabajador** que herede de Persona y tenga un número de la seguridad social (String)
  - ➔ Implementar el constructor con todos los atributos de la clase con parámetros de entrada
- Crear una clase **Empleado** que herede de Trabajador y tenga un atributo sueldo (double)
  - ➔ Implementar el constructor con todos los atributos de la clase con parámetros de entrada
- Crear una clase **Directivo** que herede de Empleado y tenga el atributo (double) bonus
  - ➔ Implementar el constructor con todos los atributos de la clase con parámetros de entrada
- Crear una clase **ConsultorExterno** que herede de Trabajador y tenga tarifa (double) y horas (double) y una empresa (String)
  - ➔ Implementar el constructor con todos los atributos de la clase con parámetros de entrada

No ponemos los constructores



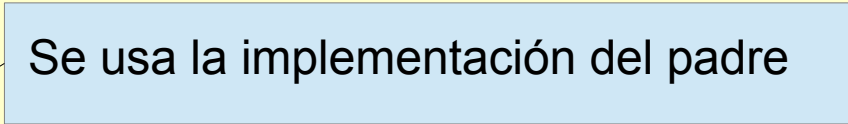
# Redefinir métodos de la clase padre

- Una clase hija puede completar o cambiar el comportamiento de un método (**sobreescribir**)
- El método redefinido debe cumplir:
  - ➔ Ofrecer al menos una POST igual o más fuerte
  - ➔ Exigir como mucho una PRE igual o más débil
- El método redefinido puede usar la implementación dada por el padre (**super**)
  - ➔ O usar cualquier otro método accesible, tanto del padre como de sí mismo

# Redefinir métodos de la clase padre

```
public class Persona {  
    private String nombre;  
    private int edad;  
    .....  
    public String toString() { return nombre + " " + edad; }  
    public void setEdad(int edad) { this.edad = edad; }  
}
```

```
public class Alumno extends Persona {  
    private int curso;  
    private String nivelAcademico;  
    .....  
    public String toString() {  
        return super.toString() + " " + curso + " " + nivelAcademico;  
    }  
    public void setCurso(int curso) { this.curso = curso; }  
}
```



# Redefinición equals de la clase Object

```
// Método implementado dentro de la clase Fecha
// el parámetro no puede ser de tipo Fecha!!
public boolean equals (Object o) {
    Fecha fecha = (Fecha) o; // downcasting: Object → Fecha
    return (day == fecha.day) &&
        (month == fecha.month) &&
        (year == fecha.year);
}

//Programa principal en un fichero .java distinto de
//Fecha.java
public static void main(String[] args) {
    Fecha ob1, ob2;
    ob1 = new Fecha(12, 4, 96);
    ob2 = new Fecha("12/4/96");
    System.out.println("La primera fecha es " + ob1);
    System.out.println("La segunda fecha es " + ob2);
    System.out.println(ob1 == ob2);
    System.out.println(ob1.equals(ob2));
}
```



# Polimorfismo

- [RAE 2001]: Cualidad de lo que tiene o puede tener distintas formas
- Polimorfismo en Java:
  - ➔ **Herencia**
    - ★ Un alumno ES-UNA persona
    - ★ Un alumno ES-UN alumno
    - ★ Un alumno ES-UN objeto
  - ➔ **Sobrecarga**
    - ★ Métodos con el mismo nombre y que devuelven lo mismo pero con distinto número de argumentos, o distinto tipo de los argumentos o distinto orden de los argumentos (p.ej., métodos constructores)
  - ➔ **Sobreescritura**
    - ★ Métodos con el mismo nombre pero distinta implementación en la clase padre que en la clase hija (ej: toString o equals)

# Polimorfismo

- El polimorfismo en POO se da por el uso de la herencia
- Se produce por distintas implementaciones de los métodos definidos en la clase padre (**sobreescribir**):
  - ➔ Distinta implementación entre clase hija y padre
  - ➔ Distinta implementación entre clases hija
- Una misma llamada ejecuta distintas sentencias dependiendo de la clase a la que pertenezca el objeto
- El código a ejecutar se determina en tiempo de ejecución ⇒ **Enlace dinámico**

# Polimorfismo

- Supongamos que declaramos: *Persona p*;
- Podría referenciar a un profesor o a un alumno en distintos momentos
  - ➔ Si  $p = \text{new Alumno}(\dots)$ , es decir,  $p$  referencia a un **alumno**, con  **$p.\text{toString}()$**  se ejecuta el `toString` de la clase `Alumno`.
  - ➔ Si  $p = \text{new Profesor}(\dots)$ , es decir,  $p$  referencia a un **profesor**, con  **$p.\text{toString}()$**  se ejecuta el `toString` de la clase `Profesor`.
- **Enlace dinámico**: Cuando hay sobreescritura de métodos se decide en **tiempo de ejecución** qué implementación del método se ejecuta.
- **OJO!**: la Sobrecarga de métodos no es lo mismo, utiliza enlace estático, por tanto se decide en **tiempo de compilación**.

# Ejemplo de Polimorfismo

```
public class Persona { ..... }  
class Alumno extends Persona {  
    .....  
    public String toString() {  
        return super.toString() + " " + curso + " " + nivelAcademico;  
    }  
}  
  
public class Profesor extends Persona {  
    private String asignatura;  
    public Profesor (String nombre, int edad, String asignatura) {  
        super(nombre, edad);  
        this.asignatura = asignatura;  
    }  
    public String toString() {  
        return super.toString() + asignatura;  
    }  
}
```

# Ejemplo de Polimorfismo

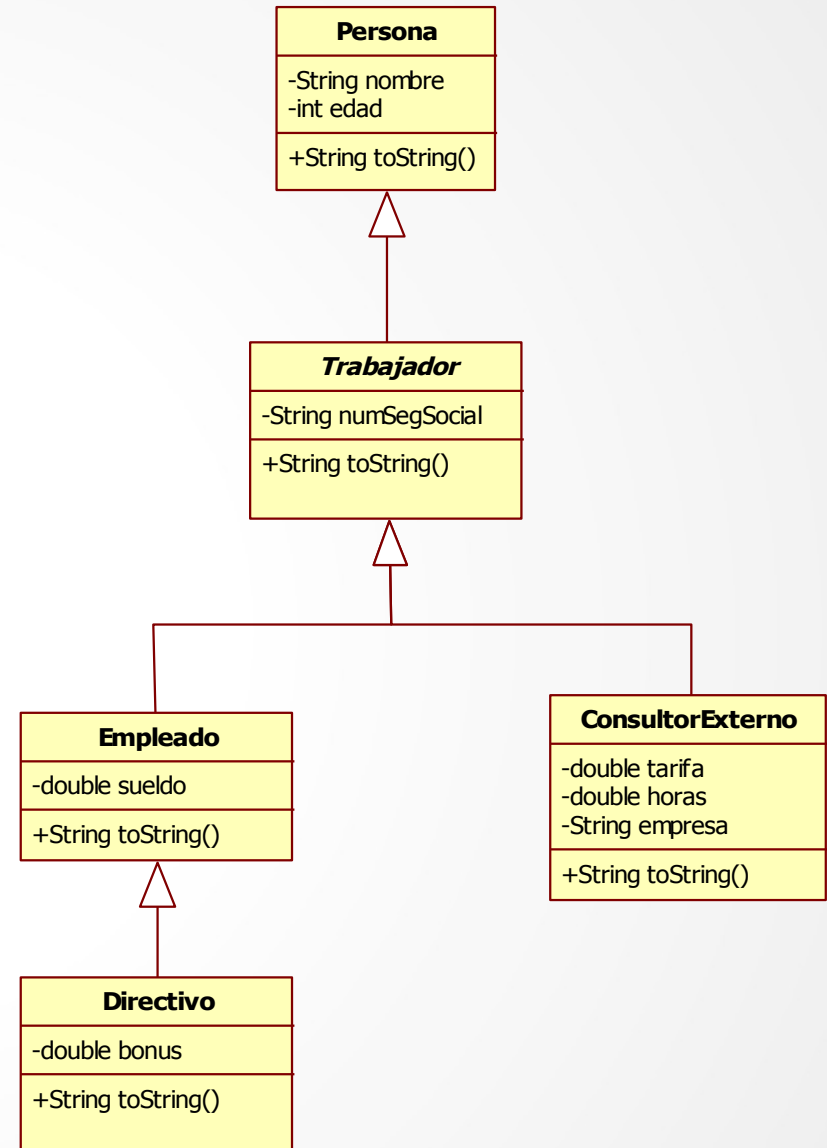
```
public static void main(String[] args) {  
    Persona v[]=new Persona[10];  
    /* Se introducen alumnos, profesores y personas en v */  
    for (int i=0 ; i<10; i++){  
        /*Se piden datos al usuario de profesor, alumno o persona */  
        switch (tipo) {  
            case /* profesor */: v[i] = new Profesor (...); break;  
            case /* alumno */: v[i] = new Alumno(...); break;  
            case /* persona */: v[i] = new Persona(...); break;  
            default: /* ERROR */ }  
        }  
  
    for (int i=0 ; i<10; i++) {  
        System.out.println(v[i]); // enlace dinámico con toString()  
    }  
}
```

# ¿Cómo sería sin polimorfismo?

- No tendríamos enlace dinámico.
- Necesitaríamos otro **switch** en el segundo **for** para cada una de las clases que queremos tratar.
- Si en el futuro se añade una nueva subclase de Persona, tendríamos que añadir una nueva rama a cada **switch**:
  - ➔ En un programa que utilice esta jerarquía sin polimorfismo, ¡Podría haber muchos **switch** como el del ejemplo!
  - ➔ En el caso de la creación es inevitable, pero
  - ➔ En el caso de las llamadas a métodos no constructores (p.e. el toString()), se podría evitar por medio del polimorfismo.

# Ejercicio

- Dadas las clases del ejercicio anterior, añadir un método toString() a cada una que devuelva todos los atributos de la clase separados por espacios.
  - ➔ Al redefinir el toString() de una clase, se debe aprovechar el método toString() de la clase padre.
- Implementar un programa principal (main) que haga lo siguiente:
  - ➔ Crear un vector de 4 Personas que contenga: 1 directivo, 2 empleados y 1 consultor externo.
  - ➔ Escribir un bucle que muestre los objetos del vector de personas en la pantalla.



# Métodos abstractos

- Tenemos un método `f()` aplicable a todos los objetos de la clase `A`.
  - ➔ Área de un polígono.
- La implementación del método es completamente diferente en cada subclase de `A`.
  - ➔ Área de un triángulo.
  - ➔ Área de un rectángulo.
  - .....
- Para declarar un método como abstracto, se pone delante la palabra reservada *abstract* y no se define un cuerpo:  
***abstract*** *tipo nombreMétodo(...);*
- Luego en cada subclase se define un método con la misma cabecera y distinto cuerpo.

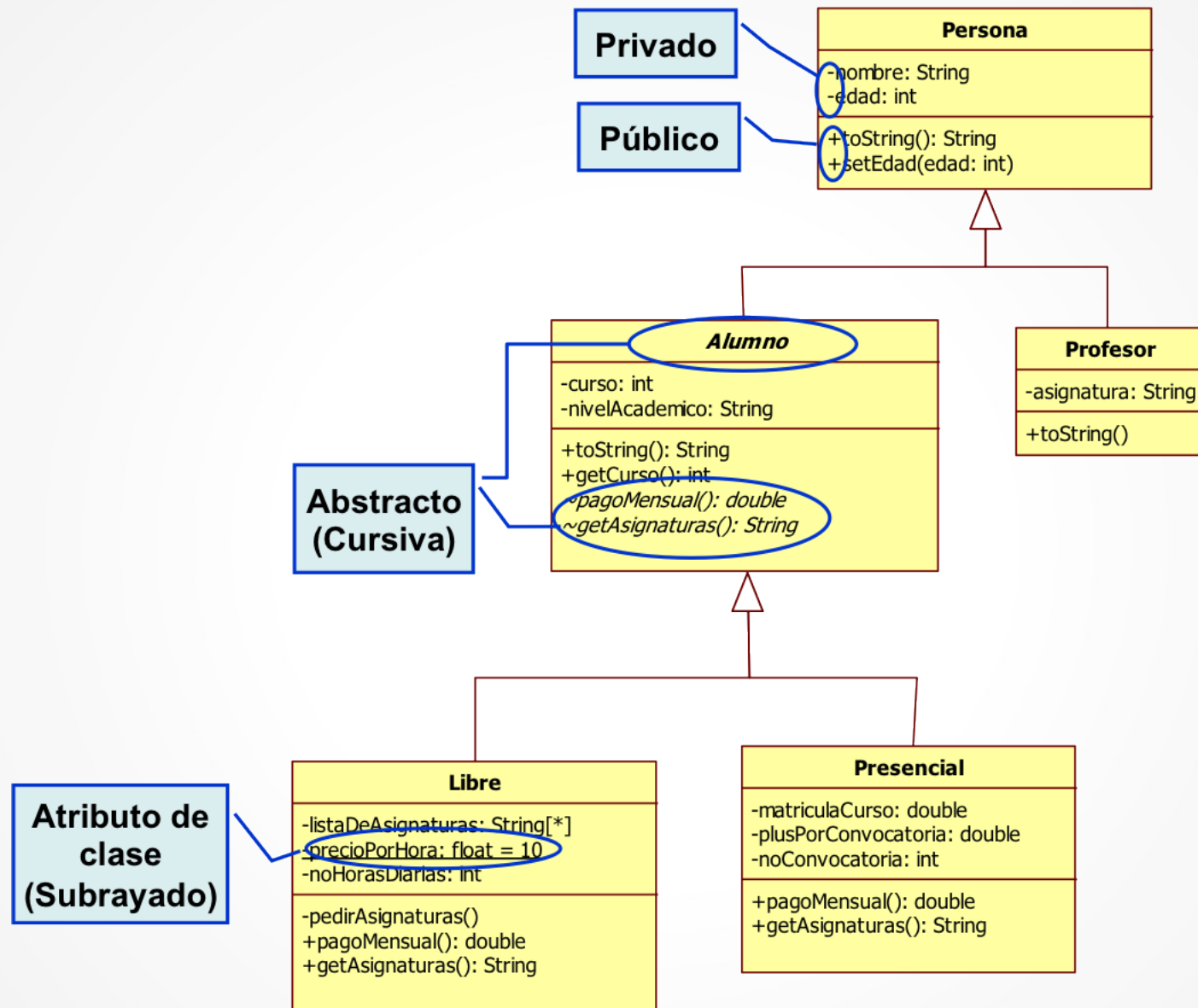


# Clases Abstractas

- Si una clase contiene al menos un método abstracto, entonces es una clase abstracta.
- Una **clase abstracta** es una clase de la que no se pueden crear objetos, pero puede ser utilizada como clase padre para otras clases.
- Declaración:

```
abstract class NombreClase {  
    .....  
}
```

# Ejemplo de clase abstracta



# Ejemplo de clase abstracta

```
public abstract class Alumno extends Persona {  
    private int curso;  
    private String nivelAcademico;  
    public Alumno(String n, int e, int c, String nivel) {  
        super(n, e);  
        curso = c;  
        nivelAcademico = nivel;  
    }  
    public String toString() {  
        return super.toString() + curso + nivelAcademico;  
    }  
    public int getCurso() {  
        return curso;  
    }  
    public abstract double pagoMensual();  
    public abstract String getAsignaturas();  
}
```

# Ejemplo de clase abstracta

```
public class Libre extends Alumno {
    private String[] listaDeAsignaturas;
    private static float precioPorHora = 10;
    private int noHorasDiarias;
    public Libre(String n, int e, int c, String nivel, int horas){
        super(n, e, c, nivel); noHorasDiarias = horas;
        pedirAsignaturas();
    }
    private void pedirAsignaturas() {} // se inicializa listaDeAsignaturas
    public double pagoMensual() {
        return precioPorHora * noHorasDiarias * 30;
    }
    public String getAsignaturas() {
        String asignaturas = "";
        for (int i = 0; i < listaDeAsignaturas.length; i++)
            asignaturas += listaDeAsignaturas[i] + ' ';
        return asignaturas;
    }
}
```

# Ejemplo de clase abstracta

```
public class Presencial extends Alumno {
    private double matriculaCurso;
    private double plusPorConvocatoria;
    private int noConvocatoria;
    public Presencial(String n, int e, int c, String nivel,
                      double mc, double pc, int nc) {
        super(n, e, c, nivel);
        matriculaCurso = mc;
        plusPorConvocatoria = pc;
        noConvocatoria = nc;
    }
    public double pagoMensual() {
        return (matriculaCurso +
                plusPorConvocatoria * noConvocatoria) / 12;
    }
    public String getAsignaturas(){
        return "todas las del curso " + getCurso();
    }
}
```

# Ejemplo de clase abstracta

```
// FUNCIONES GLOBALES
void mostrarAsignaturas(Alumno v[]) {
    for (int i=0; i<v.length; i++) {
        System.out.println(v[i].getAsignaturas()); // enlace dinámico
    }
}
double totalPagos(Alumno v[]) {
    double t=0;
    for (int i=0; i<v.length; i++) {
        t += v[i].pagoMensual(); // enlace dinámico
    }
    return t;
}
```

# Ejercicio

- Añadir a la clase **Trabajador** un método abstracto *double calcularSueldoMensual()*
- Añadir a la clase **Empleado** un método *double calcularSueldoMensual()* que devuelve el sueldo dividido entre 14.
- Añadir a la clase **Directivo** un método *double calcularSueldoMensual()* que devuelve el sueldo dividido entre 14 y le suma su bonus.
- Añadir a la clase **ConsultorExterno** un método *double calcularSueldoMensual()* que devuelve  $\text{tarifa} \times \text{horas}$ .
- Implementar un programa principal (**main**) que haga lo siguiente:
  - ➔ Crear un vector de 4 trabajadores que contenga: 1 directivo, 2 empleados y 1 consultor externo.
  - ➔ Calcular el sueldo mensual total de las personas del vector y mostrarlo en pantalla.

