

INTRODUCCIÓN A LA PROGRAMACIÓN

**BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA
FACULTAD DE CIENCIAS DE LA COMPUTACIÓN**

Luz A. Sánchez Gálvez

sanchez.galvez@correo.buap.mx

lgalvez@cs.buap.mx

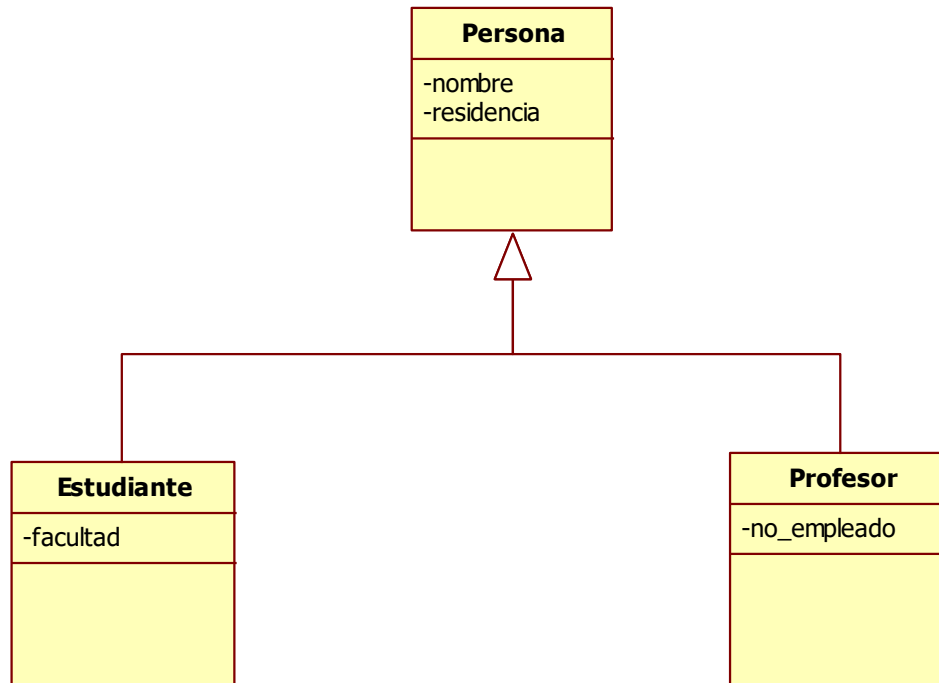
luzsg@hotmail.com

Herencia

- ▶ La **herencia** en lenguajes orientados a objetos consiste en la posibilidad de definir una clase, que tenga las mismas propiedades de una clase existente, pudiendo añadir nuevas funcionalidades y/o nueva información.
- ▶ En lugar de modificar la clase ya definida, se crea una nueva clase derivada de ésta.
- ▶ Una **subclase hereda** todos los métodos y todas las variables de instancia de la superclase y, además, puede tener sus propios métodos y variables de instancia.

Herencia

- ▶ La superclase **Persona** con las subclases **Estudiante** y **Profesor**.



Herencia

- ▶ La clase Persona:

```
public class Persona {  
    ...  
}
```

- ▶ La clase Estudiante se deriva o se extiende de la clase Persona:

```
public class Estudiante extends Persona {  
    ...  
}
```

- ▶ **Estudiante** es una **subclase** de **Persona**. Persona es una *superclase* de Estudiante.
- ▶ Estudiante es una *clase derivada* de la *clase base* Persona.



La Superclase Persona

```
public class Persona {  
    private String nombre;  
    private String residencia;  
  
    public Persona (String n, String r) {           // constructor  
        nombre = n;  
        residencia = r;  
    }  
    public String getNombre() {  
        return nombre;  
    }  
    public String getResidencia() {  
        return residencia;  
    }  
    public void setResidencia(String nuevaResidencia) {  
        residencia = nuevaResidencia;  
    }  
}
```

La Subclase Estudiante

```
public class Estudiante extends Persona {  
    private String facultad;  
  
    public Estudiante (...) {                // constructor  
        ...  
    }  
    public String getFacultad() {  
        return facultad;  
    }  
}
```

- ▶ Los objetos de la clase **Estudiante** se caracterizan por las propiedades **heredadas** de la clase **Persona** y, además, por la facultad en la que el estudiante esta registrado.

Características de las subclases

- ▶ Todas las propiedades (variables de instancia y métodos) definidas en la superclase o clase base se especifican implícitamente también en la clase derivada, es decir, se heredan.
- ▶ La clase derivada puede tener propiedades adicionales con respecto a las heredadas de la clase base.
- ▶ Cada instancia de la clase derivada es también una instancia de la clase base. Por lo tanto, es posible utilizar un objeto de la clase derivada en cada situación en la que es posible utilizar un objeto de la clase base.



Constructor de una subclase (1)

- ▶ Para definir un constructor de una subclase se deben considerar los campos de la clase base.
- ▶ Esto se logra, insertando en el constructor de la subclase, la llamada a un constructor de la clase base, utilizando el constructor especial **super()** de Java .
- ▶ La sentencia **super()** debe aparecer como la *primera instrucción ejecutable* en el cuerpo del constructor de la clase derivada.



Constructor de una subclase (2)

```
public class Estudiante extends Persona {  
    private String facultad;  
  
    public Estudiante (String n, String r, String f) {  
        super(n,r); //Llama al constructor de Persona(String n, String r)  
        facultad = f;  
    }  
    ...  
}
```

- La sentencia `super(n, r);` llama a `Persona(n, r)`, que inicializa las variables de instancia nombre y residencia, heredadas de la superclase `Persona`, respectivamente, para las cadenas `n` y `r`.



Uso de `super()`

- ▶ En general, cuando la subclase tiene sus propias variables de instancia, su constructor debe primero crear un objeto de la superclase (usando `super()`) y luego crear sus propias variables de instancia.
- ▶ ¿Qué sucede si se olvida insertar `super()`?
El constructor sin argumentos de la superclase es llamado automáticamente (obviamente, si el constructor sin argumentos no es definido por la superclase, se produce un error de compilación).



Uso de `super()`

- ▶ ¿Qué sucede si se olvida definir constructores para la subclase?

El constructor sin argumentos es definido automáticamente; tal constructor llama al constructor sin argumentos de la superclase e inicializa las propias (no heredadas) variables de instancia de la subclase a los valores por default.

- ▶ Por tanto, se recomienda definir explícitamente los constructores de una subclase, de tal manera que en su primera declaración llamen a `super()`; para evitar hacer uso de estas definiciones automáticas.



Las variables y métodos heredados

- ▶ Todos los objetos de la clase Estudiante, además de tener los propios métodos y variables de instancia definidas en Estudiante, heredan todos los métodos y variables de instancia de Persona.

```
public class PruebaEstudiante {  
    public static void main(String[] args) {  
        Persona p = new Persona("Daniel", "Puebla");  
        System.out.println(p.getNombre());  
        System.out.println(p.getResidencia());  
        Estudiante e = new Estudiante("Jacobo", "Merida", "Ingeniería");  
        System.out.println(e.getNombre());           //método heredado de Persona  
        System.out.println(e.getResidencia());       //método heredado de Persona  
        System.out.println(e.getFacultad());         //método de Estudiante  
    }  
}
```

- ▶ Los métodos getNombre() y getResidencia(), heredados de Persona son métodos de la clase Estudiante.

Compatibilidad

- ▶ Cada objeto de la clase derivada es también un objeto de la clase base, por tanto, es posible utilizar un objeto de la clase derivada en cada situación o contexto en el que es posible utilizar un objeto de la clase base.
- ▶ Es decir, los objetos de la clase derivada son **compatibles** con los objetos de la clase base.



Compatibilidad

```
public class PruebaCompatibilidad {  
    public static void main(String[] args) {  
        Persona p = new Persona("Daniel", "Puebla");  
        Estudiante e = new Estudiante("Jacobo", "Merida", "Ingeniería");  
        Persona pp;  
        Estudiante ee;  
        pp = e;           // Estudiante es compatible con Persona  
        ee = p;           // ERROR Persona no es compatible con Estudiante  
        System.out.println(pp.getNombre());    // método de Persona  
        System.out.println(pp.getResidencia()); // método de Persona  
        System.out.println(pp.getFacultad());  
    }                       //ERROR getFacultad no es método de Persona  
}
```

El error se debe a que la variable pp es una referencia a una Persona, y por lo tanto, no es posible acceder a los métodos de Estudiante (aunque pp se refiere en realidad a un objeto Estudiante). Esto se debe a que Java implementa la *comprobación de tipos estáticos*.

Compatibilidad entre los parámetros actuales y formales

```
public class PruebaCompatibilidad2 {  
    public static void imprimePersona(Persona p) {  
        System.out.println(p.getNombre());  
        System.out.println(p.getResidencia());  
    }  
    public static void imprimeEstudiante(Estudiante e) {  
        System.out.println(e.getNombre());  
        System.out.println(e.getResidencia());  
        System.out.println(e.getFacultad());  
    }  
    public static void main(String args[]) {  
        Persona per = new Persona("Daniel", "Puebla");  
        Estudiante est = new Estudiante ("Jacobo", "Merida", "Ingeniería");  
        imprimePersona(per);  
        imprimePersona(est);           // Estudiante es compatible con Persona  
        imprimeEstudiante(est);  
        imprimeEstudiante(per);       //ERROR Persona no es compatible con Estudiante  
    }  
}
```

Acceso a los campos públicos y privados de la superclase

- ▶ La clase derivada hereda todas las variables de instancia y todos los métodos de la superclase.
- ▶ Los campos públicos de la superclase son accesibles desde la clase derivada.
- ▶ Se puede añadir a la subclase Estudiante un método `imprimeNombre()`:

```
public class Estudiante extends Persona {  
    ...  
    public void imprimeNombre() {  
        System.out.println(this.getNombre ());  
    }  
    ...  
}
```

Acceso a los campos públicos y privados de la superclase

- ▶ Los campos privados de la superclase no son accesibles a los métodos de la clase derivada, ya que no son accesibles a cualquier otro método fuera de la superclase.
- ▶ En Estudiante se introduce un método cambiarNombre() que produce un error de compilación.

```
public class Estudiante extends Persona {
```

```
...
```

```
    public void cambiarNombre(String s) {
```

```
        this.nombre = s; /* ¡ERROR! la variable de instancia nombre es privada  
                           en Persona. Por tanto, no es accesible desde la clase  
                           derivada Estudiante */
```

```
    }
```

```
...
```

```
}
```

Acceso a los campos públicos y privados de la superclase

- ▶ Java permite utilizar los campos públicos, privados, y protegidos.
- ▶ Los campos protegidos de una clase no pueden ser accedidos por métodos externos, sino que se puede acceder por los métodos de una clase derivada.



Redefinición de métodos (1)

Overriding

- ▶ Un método `m()` se dice que se redefine en la subclase cuando se tiene el mismo método `m()` en la superclase.
 - ▶ Para redefinir un método, Java requiere que la definición del nuevo método `m()` tenga el mismo tipo devuelto como el método original `m()`.
 - ▶ Es decir, el método que se está redefiniendo debe tener la misma cabecera que el método original.
 - ▶ Cada vez que se invoca el método `m()` sobre un objeto de la clase derivada `D`, el método que se llama es el redefinido en `D`, y no el definido en la clase base `B`, incluso si la referencia utilizada para denotar el objeto de invocación es del tipo `B`.
 - ▶ Este comportamiento se denomina **polimorfismo**.
-



Redefinición de métodos (2)

► En Persona se define un método imprimeDatos() :

```
public class Persona {
```

```
...
```

```
    public void imprimeDatos() {
```

```
        System.out.println(nombre + " " + residencia);
```

```
    }
```

```
...
```

```
}
```



Redefinición de métodos (3)

- ▶ Se redefine el método `imprimeDatos()` en la clase `Estudiante`, de tal manera que imprima también la facultad:

```
public class Estudiante extends Persona {  
    ...  
    public void imprimeDatos() {    // redefine  
        System.out.println(this.getNombre()+" "+this.getResidencia()  
            + " " + facultad);  
    }  
    ...  
}
```



Redefinición de métodos (4)

```
public class ClienteEstudiante {  
    public static void main(String[] args) {  
        Persona p = new Persona("Daniel", "Puebla");  
        Estudiante e = new Estudiante ("Jacobo", "Merida", "Ingeniería");  
        p.imprimeDatos();  
        e.imprimeDatos();  
    }  
}
```



Polimorfismo (1)

- ▶ La redefinición de métodos causa polimorfismo, esto es la presencia de métodos con el mismo distintivo que se comportan diferentes en una jerarquía de clases.

```
public class EstudiantePolimorfismo {  
    public static void main(String[] args) {  
        Persona p = new Persona("Daniel", "Puebla");  
        Estudiante e = new Estudiante ("Jacobo", "Merida", "Ingeniería");  
        Persona pe = e;           // Debido a las reglas de compatibilidad  
        p.imprimeDatos();  
        e.imprimeDatos();  
        pe.imprimeDatos();       //¿qué imprime?  
    }  
}
```

Polimorfismo (2)

- ▶ El método `imprimeDatos()` que es llamado, se elige basándose en el objeto de la clase al que pertenece, y no en el tipo de la variable denotada.
- ▶ Este mecanismo para acceder a los métodos se llama **enlace**.
- ▶ En el ejemplo anterior, el método llamado sobre el objeto `pe` será el definido en la clase `Estudiante`, es decir, se imprime nombre, residencia y facultad.
- ▶ Al ejecutar el programa se imprimirá:
 - ▶ Daniel Puebla
 - ▶ Jacobo Merida Ingeniería
 - ▶ Jacobo Merida Ingeniería



Jerarquía de clases

- ▶ Una clase puede tener varias subclases. Por ejemplo, se podría definir una subclase `PersonaExperta` de `Persona`, cuyos objetos representan personas que son expertos en un tema determinado, donde el tema en el que son expertos es una propiedad específica de la clase.
- ▶ Una subclase de una clase puede tener subclases. Por ejemplo, la clase `Estudiante` derivada de `Persona` podría tener una subclase `EstudianteTrabajador`.
- ▶ Por lo tanto, utilizando varias derivaciones, es posible crear jerarquías de clases.



La clase Object

- ▶ Todas las clases definidas en Java son subclases de la clase predefinida Object.
- ▶ Esto significa que todas las clases heredan de Object varios métodos estándar, tales como equals (), clone() y toString (), definidos en Object.
- ▶ Cuando se invoca uno de estos métodos sobre un objeto perteneciente a una clase C, pero no se le redefine en C (o en una de sus superclases diferentes de Object) se utiliza el método definido en la clase Object.



El método toString()

- ▶ El método toString () es definido en la clase **Object**, y por tanto es heredado a todas las clases de Java.

public String toString ()

- ▶ Este método se utiliza para transformar un objeto en una cadena. Típicamente, se utiliza para construir una cadena que contiene la información sobre un objeto que se puede imprimir, y puede redefinirse para una cierta clase.

El método toString()

- ▶ Si el método toString () de la clase Object no se redefine, se utiliza:

```
public class PruebatoString {  
    public static void main(String[] args) {  
        Persona p = new Persona("Jesus", "Guadalajara");  
        System.out.println(p.toString());  
    }  
}
```



El método **toString()**

- ▶ El método **toString()** se puede redefinir en la clase **Persona**, en la que se devuelve el nombre de la persona.

```
public class Persona {  
    ...  
    public String toString() {  
        return nombre;  
    }  
    ...  
}
```

- ▶ Ahora, la clase **PruebatoString** imprime "Jesus"
-



El uso de toString() en print() y println()

- ▶ La clase **PrintStream** predefinida contiene variantes de los métodos `print()` y `println()`, que tienen un parámetro formal del tipo de referencia a `Object` en lugar de `String`.
- ▶ Estos dos métodos invocan en el parámetro de tipo `Object` al método `toString ()` e imprimen la cadena resultante utilizando el método de impresión de `String`.
- ▶ En la práctica, esto permite evitar el uso explícito de `toString ()` en el argumento de `print ()` y `println()`.



El uso de toString() en print() y println()

```
public class PruebatoString2 {  
    public static void main(String[] args) {  
        Persona p = new Persona("Jesus", "Guadalajara");  
        System.out.println(p);  
        // ésto es equivalente a System.out.println(p.toString());  
    }  
}
```



Composición (1)


Opcional

- ▶ Definir la clase Estudiante2 con funciones análogas a Estudiante, pero que no haga uso de la herencia.
- ▶ La idea es incluir en Estudiante2 una variable de instancia, que sea una referencia a un objeto Persona.
- ▶ Esta variable de instancia se utilizará para mantener las propiedades de nombre y residencia; añadir a ésta una variable de instancia facultad, que se utilice para almacenar la Facultad.



Composición (2)

```
public class Estudiante2{  
    private Persona persona;  
    private String facultad;  
  
    public Estudiante2(String nombre, String residencia, String facultad) {  
        persona = new Persona(nombre, residencia);  
        this.facultad = facultad;  
    }  
    public String getNombre() {  
        return persona.getNombre ();  
    }  
    public String getResidencia() {  
        return persona.getResidencia();  
    }  
    public void setResidencia(String residencia) {  
        persona.setResidencia(residencia);  
    }  
    public String getFacultad() {  
        return facultad;  
    }  
}
```



Composición (3)

- ▶ La clase Estudiante2 utiliza una variable de instancia persona, el objeto persona almacena nombre y residencia del estudiante.
- ▶ Estudiante2 es un cliente de Persona, por eso el campo persona se manipula utilizando los métodos públicos de la clase Persona.
- ▶ La clase Estudiante2 oculta a sus clientes el uso del objeto Persona, ya que ofrece los métodos getNombre(), getResidencia(), y setResidencia (), que operan sobre los objetos de Estudiante2.
- ▶ La clase Estudiante2 ofrece a sus clientes los mismos métodos que la clase Estudiante. Sin embargo, los objetos Estudiante2 no son compatibles con los objetos de la clase Persona, ya que una variable o un parámetro formal del tipo Persona no puede contener un objeto Estudiante2.

Herencia o composición? (opcional)

- ▶ La construcción de la clase Estudiante2 en lugar de Estudiante es una elección cuestionable. Así que, ¿cuándo es razonable utilizar la composición?
- ▶ En general, se pueden adoptar los siguientes criterios:
 - ▶ Si cada objeto de X es un objeto de Y (X is-a Y), entonces se utiliza la herencia.
 - ▶ Si cada objeto de Y tiene un objeto de X (Y has-a X), entonces se utiliza composición.

