Guía práctica de estudio 07: Herencia

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 07: Herencia

Objetivo:

Implementar los conceptos de herencia en un lenguaje de programación orientado a objetos.

Actividades:

- Crear clases que implementen herencia.
- Generar una jerarquía de clases.

Introducción

En la programación orientada a objetos, la **herencia** está en todos lados, de hecho, se podría decir que es casi imposible escribir el más pequeño de los programas sin utilizar **herencia**. Todas las clases que se crean dentro de la mayoría de los lenguajes de programación orientados a objetos heredan implícitamente de la clase *Object* y, por ende, se pueden comportar como objetos (que es la base del paradigma).

La herencia permite crear nuevos objetos que asumen las propiedades de objetos existentes. Una clase que es usada como base para heredarse es llamada **súper clase** o **clase base**. Una clase que hereda de una súper clase es llamada **subclase** o **clase derivada**. La clase derivada hereda todas las propiedades y métodos visibles de la clase base y, además, puede agregar propiedades y métodos propios.

NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

Herencia

La **herencia** es el proceso que implica la creación de clases a partir de clases ya existentes, permitiendo, además, agregar más funcionalidades. Utilizando herencia la relación jerárquica queda establecida de manera implícita, partiendo de la clase más general (clase base) a la clase más específica (clase derivada).

Las dos razones más comunes para utilizar herencia son:

- Para promover la reutilización de código.
- Para usar polimorfismo (el cuál se abordará en la siguiente práctica).

Para heredar en Java se utiliza la palabra reservada *extends* al momento de definir la clase, su sintaxis es la siguiente:

[modificadores] class NombreClaseDerivada ***extends*** *NombreClaseBase*

Los objetos de las clases derivadas (subclases) se crean (instancian) igual que los de la clase base y pueden acceder tanto a atributos y métodos propios, así como a los de la clase base.

Existen lenguajes de programación que permiten heredar de más de una clase, lo que se conoce como multiherencia, sin embargo, **Java solo soporta herencia simple**.

Ejemplo:

Dada la siguiente jerarquía de clases:

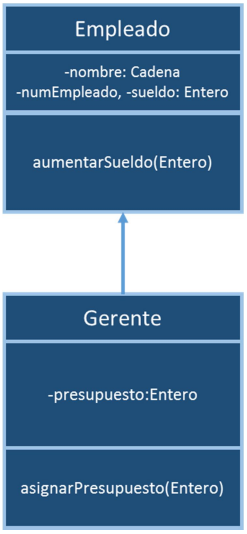

Figura 1. Jerarquía de clases de Empleado y Gerente.

Se crea la clase **Empleado** (clase base)

```
public class Empleado {
    private String nombre;
    private int numEmpleado, sueldo;

    public void setNombre(String nombre){
        this.nombre = nombre;
    }
    public String getNombre(){
        return this.nombre;
    }
    public void setNumEmpleado(int numEmpleado){
        this.numEmpleado = numEmpleado;
    }
    public int getNumEmpleado(){
        return this.numEmpleado;
    }
    public void setSueldo(int sueldo){
        if (sueldo >= 0) {
            this.sueldo = sueldo;
        }
    }
    public int getSueldo(){
        return this.sueldo;
    }
    public void aumentarSueldo(int porcentaje) {
        sueldo += (int)(sueldo * porcentaje / 100);
    }
}
```

Y la clase **Gerente** (subclase)

```
public class Gerente extends Empleado {
    private int presupuesto;
    public void setPresupuesto(int presupuesto){
        this.presupuesto = presupuesto;
    }
    public int getPresupuesto(){
        return this.presupuesto;
    }
    void asignarPresupuesto(int p) {
        presupuesto = p;
    }
}
```

Se crea también una clase de **PruebaEmpleado** para validar el comportamiento de las clases creadas y sus métodos.

```
public class PruebaEmpleado {
    public static void main (String [] args){
        Gerente gerente = new Gerente();
        // Gerente hereda todos los atributos y métodos
        // de la Empleado (reutiliza código)
        gerente.setNombre("Luis Aguilar");
        gerente.setNumEmpleado(8524);
        gerente.setSueldo(16000);
        System.out.println("Nombre: " + gerente.getNombre());
        System.out.println("Número de empleado: " + gerente.getNumEmpleado());
        System.out.println("Sueldo: " + gerente.getSueldo());
        gerente.aumentarSueldo(10);
        System.out.println("Nuevo sueldo: " + gerente.getSueldo());
        // Y tiene métodos y atributos propios
        gerente.setPresupuesto(50000);
        System.out.println("Presupuesto: " + gerente.getPresupuesto());
        gerente.asignarPresupuesto(65000);
        System.out.println("Nuevo presupuesto: " + gerente.getPresupuesto());
    }
}
```

Relaciones IS-A y HAS-A

La **relación IS-A** (es un) se basa en la herencia y permite afirmar que un objeto es de un tipo (clase) en específico. Por ejemplo:

public class Animal { }
public class Caballo extends Animal { }
public class Purasangre extends Caballo { }
Dada la jerarquía de clases anterior se puede afirmar que:

> **Caballo hereda de Animal, lo que significa que Caballo IS-A (es un) Animal.**
> **Purasangre hereda de Caballo, lo que significa que Purasangre IS-A (es un) Caballo.**

La **relación HAS-A (tiene un)** es un concepto que se vio en la práctica de Abstracción y encapsulamiento como **composición**. Se basa en el uso más que en la herencia, es decir, una clase X HAS-A Y si el código en la clase X tiene como atributo una referencia de la clase Y. Este Por ejemplo:

public class Animal { }
public class Caballo extends Animal {
    private SillaMontar miSilla;
}

Dada la jerarquía de clases anterior se puede afirmar que:

> **Un Caballo HAS-A (tiene una) SillaMontar, debido a que cada instancia de Caballo tendrá una referencia hacia una SillaMontar.**

Clase Object

En Java todas las clases que se crean son una subclase de la clase **Object** (excepto, por supuesto, Object), es decir, cualquier clase que se escriba o que se use hereda de Object. Los métodos *clone*, *equals*, *hashCode*, *notify*, *toString*, *wait* y otros son declarados dentro de la clase Object y, por ende, todas las clases los poseen (los heredan).

Por otro lado, el operador *instanceof* es utilizado por las referencias de los objetos para verificar si un objeto es de un tipo (clase) específico.

Ejemplo:

```
public class Instancias{
	public static void main (String [] args){
		Gerente gerente = new Gerente();
		// Instanceof permite validar si un objeto
		// es de un tipo específico
		if (gerente instanceof Gerente){
			System.out.println("Instancia de Gerente.");
		}
		// Como Gerente hereda de Empleado también es de tipo Empleado
		if (gerente instanceof Empleado){
			System.out.println("Instancia de Empleado.");
		}
		// Se verifica que cualquier objeto es una instancia de Object
		if (gerente instanceof Object){
			System.out.println("Instancia de Object.");
		}
	}
}
```

Sobrescritura (overridding)

Como ya se mencionó, cuando una clase B hereda de otra A, la clase B puede acceder a todos los atributos y métodos visibles de la clase A, sin embargo, ¿qué pasa cuando el comportamiento de un método no es el adecuado o es parcialmente adecuado? La **sobrescritura** se refiere a la habilidad de **redefinir** el comportamiento de un método específico en una subclase, generando así un comportamiento acorde a las necesidades de cada clase.

El método *toString* es definido dentro de la clase *Object*. Este método es utilizado para mostrar información de un objeto. Por ejemplo, la clase *Empleado* posee los atributos *nombre*, *numEmpleado* y *sueldo*, los cuales se desean mostrar al momento de imprimir un objeto de esta clase, sin embargo, como el método *toString* está definido en la clase *Object* y ésta no conoce los atributos de *Empleado*, dichos atributos no se van a imprimir. Por lo tanto, para mostrar la información deseada es necesario sobrescribir el método.

Un método sobrescrito en una clase derivada debe seguir las siguientes reglas:

- Debe tener el mismo nombre.
- Debe tener el mismo tipo y número de parámetros.
- El tipo de nivel de acceso debe ser igual o más accesible.
- El valor de retorno debe ser del mismo tipo o un subtipo.

Por lo tanto, la **sobrescritura** solo tiene sentido en la herencia, es decir, no es posible sobrescribir un método dentro de la misma clase porque el compilador detectaría que existen dos métodos que se llaman igual y reciben el mismo número y tipo de parámetros.

Ejemplo:

```java
public class Empleado {
    private String nombre;
    private int numEmpleado, sueldo;

    public void setNombre(String nombre){
        this.nombre = nombre;
    }
    public String getNombre(){
        return this.nombre;
    }
    public void setNumEmpleado(int numEmpleado){
        this.numEmpleado = numEmpleado;
    }
    public int getNumEmpleado(){
        return this.numEmpleado;
    }
    public void setSueldo(int sueldo){
        if (sueldo >= 0) {
            this.sueldo = sueldo;
        }
    }
    public int getSueldo(){
        return this.sueldo;
    }
    public void aumentarSueldo(int porcentaje) {
        sueldo += (int)(sueldo * porcentaje / 100);
    }
    public String toString (){
        return "Nombre: " + this.nombre +
                "\nNúmero: " + this.numEmpleado +
                "\nSueldo: " + this.sueldo;
    }
}
```

```java
public class Gerente extends Empleado {
    private int presupuesto;
    public void setPresupuesto(int presupuesto){
        this.presupuesto = presupuesto;
    }
    public int getPresupuesto(){
        return this.presupuesto;
    }
    void asignarPresupuesto(int p) {
        presupuesto = p;
    }
    public String toString(){
        return super.toString() +
            "\nPresupuesto: " + this.presupuesto;
    }
}
```

```
public class PruebaEmpleado {
    public static void main (String [] args){
        Gerente gerente = new Gerente();
        gerente.setNombre("Luis Aguilar");
        gerente.setNumEmpleado(8524);
        gerente.setSueldo(16000);
        gerente.setPresupuesto(50000);
        // Se manda llamar el método toString, el cual
        // fue sobrescrito en la clase Empleado.
        System.out.println(gerente);
    }
}
```

Cuando se imprime el objeto gerente, implícitamente se manda llamar el método *toString*, como éste fue sobrescrito por la clase *Empleado*, esa información es la que se muestra. Sin embargo, para la clase *Gerente* el método es parcialmente correcto, ya que falta imprimir el atributo presupuesto propio de *Gerente*, por tanto, es necesario volver a sobrescribir el método *toString* para agregar dicho atributo.

Constructores en la herencia

Como se mencionó en la práctica de Clases y objetos, un constructor es un método que tiene el mismo nombre que la clase y cuyo propósito es inicializar los atributos de un nuevo objeto. Se ejecuta automáticamente cuando se crea un objeto o instancia de la clase.

Cuando se crea un objeto de una clase derivada se crea, implícitamente, un objeto de la clase base que se inicializa con su constructor correspondiente.

Si en la creación del objeto se usa el constructor sin argumentos (*constructor no-args*), entonces se produce una llamada implícita al constructor sin argumentos de la clase base.

Sin embargo, si se quiere utilizar constructores sobrecargados es necesario invocarlos explícitamente.

Así mismo, dentro de una clase derivada se puede acceder a los elementos de la clase base a través de la palabra reservada **super**.

Ejemplo

```java
public class Empleado {
    private String nombre;
    private int numEmpleado, sueldo;

    public Empleado (String nombre, int sueldo, int numEmpleado) {
        this.nombre = nombre;
        this.sueldo = sueldo;
        this.numEmpleado = numEmpleado;
    }

    public void setNombre(String nombre){
        this.nombre = nombre;
    }
    public String getNombre(){
        return this.nombre;
    }
    public void setNumEmpleado(int numEmpleado){
        this.numEmpleado = numEmpleado;
    }
    public int getNumEmpleado(){
        return this.numEmpleado;
    }
    public void setSueldo(int sueldo){
        if (sueldo >= 0) {
            this.sueldo = sueldo;
        }
    }
    public int getSueldo(){
        return this.sueldo;
    }
    public void aumentarSueldo(int porcentaje) {
        sueldo += (int)(sueldo * porcentaje / 100);
    }
    @Override
    public String toString (){
        return "Nombre: " + this.nombre +
            "\nNúmero: " + this.numEmpleado +
            "\nSueldo: " + this.sueldo;
    }
}
```

```
public class Gerente extends Empleado {
    private int presupuesto;
    public Gerente (String nombre, int sueldo,
                    int numEmpleado, int presupuesto) {
        super(nombre, sueldo, numEmpleado);
        this.presupuesto = presupuesto;
    }
    public void setPresupuesto(int presupuesto){
        this.presupuesto = presupuesto;
    }
    public int getPresupuesto(){
        return this.presupuesto;
    }
    void asignarPresupuesto(int p) {
        presupuesto = p;
    }
    @Override
    public String toString(){
        return super.toString() +
            "\nPresupuesto: " + this.presupuesto;
    }
}
```

Como se puede observar, el constructor de Ejecutivo invoca directamente al constructor de Empleado mediante la palabra reservada super. La llamada al constructor de la superclase debe ser **la primera sentencia** del constructor de la subclase.

```
public class PruebaEmpleado {
    public static void main (String [] args){
        Gerente gerente = new Gerente("Luis Aguilar", 16000, 8524, 50000);
        System.out.println(gerente);
    }
}
```

Debido a que el método *toString* de la clase *Empleado* muestra los atributos deseados, se invoca explícitamente y se agrega el atributo presupuesto. La clase *PruebaEmpleado* ahora sí mostrará toda la información del gerente.

Sobrecarga (overloading) vs Sobrescritura (overridding)

Una de las cosas que más confunden a los programadores novatos son las diferencias entre los conceptos de **sobrecarga** y **sobrescritura**. La **sobrescritura** es un concepto que tiene sentido en la **herencia** y se refiere al hecho de **volver a definir** un método heredado.

La **sobrecarga** sólo tiene sentido en la clase misma, se pueden definir varios métodos con el **mismo nombre**, pero **con diferentes tipos y número de parámetros** en ella.

Bibliografía

Barnes David, Kölling Michael
Programación Orientada a Objetos con Java.
Tercera Edición.
Madrid
Pearson Educación, 2007

Deitel Paul, Deitel Harvey.
Como programar en Java
Septima Edición.
México
Pearson Educación, 2008

Martín, Antonio
Programador Certificado Java 2.
Segunda Edición.
México
Alfaomega Grupo Editor, 2008

Dean John, Dean Raymond.
Introducción a la programación con Java
Primera Edición.
México
Mc Graw Hill, 2009