TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN UTN-FRC

TUP

UTN

Facultad Regional Córdoba

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN

LABORATORIO DE COMPUTACIÓN III

Unidad Temática :
Herencia y Polimorfismo

Material de Estudio

2do Año - 3er Cuatrimestre

2020

V.0.1

Índice

1. HERENCIA 2
Introducción 2
Cláusula extends 3
Referencia super 4
Herencia y constructores 5
2. POLIMORFISMO 10
Referencias a la clase base 10
Sobreescritura de métodos 11
Métodos polimórficos 12
Identificación de la clase de un objeto. Operador is 14
Casting hacia abajo (downcasting) 15
3. CLASES ABSTRACTAS 22
Colecciones de referencias a la clase base 24
BIBLIOGRAFÍA 31

1. HERENCIA

Introducción

La herencia es una relación entre clases, con una de ellas denominada clase base y la que hereda las características denominada clase derivada. Esto permite tener varias clases con un comportamiento similar en ciertas situaciones (por ejemplo, al llamar a ciertos métodos) y un comportamiento específico en otras. Como se muestra en la siguiente figura

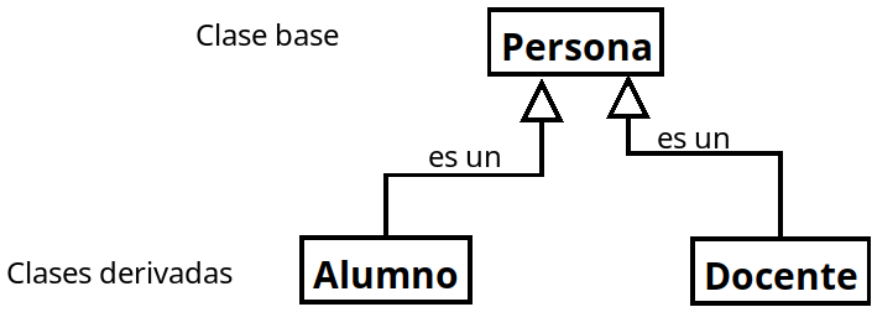

Imagen 1: Elaboración propia

La herencia se utiliza cuando dos clases son suficientemente similares como para programarse como una única clase, pero algunos de los objetos tienen características que los diferencian o que representan casos particulares.

La relación de herencia siempre debe poder interpretarse con la expresión "**es un**". Entre las clases Persona y Alumno puede plantearse una relación de herencia, ya que un Alumno siempre **es una** Persona.

Entre Alumno y Curso no tiene sentido la herencia, un Alumno no **es un** Curso. (Aunque puede haber otro tipo de reacción entre esas clases, p.ej. Composición).

La relación de herencia permite que al programar las clases derivadas, sólo se incluya en éstas las variantes o los agregados que posean con respecto a la clase base. Si en la clase Persona se incluyeron los atributos documento, nombre y fecha de nacimiento, en las derivadas no se necesitan esos atributos porque los van a heredar desde la clase base. Es decir, que establecer la relación de herencia, todo Alumno es una Persona, y por lo tanto ya tiene documento, nombre y fecha de nacimiento, sin tener que programar ni la definición de esos atributos ni los métodos relacionados con ellos.

Java posee herencia simple, es decir que cada clase puede tener una única clase base, sin embargo, cada clase puede tener múltiples clases derivadas. Por otro lado, las clases derivadas pueden tener a su vez otras que deriven de ellas, y a causa

de esto, el conjunto de relaciones de herencia de una clase se lo denomina Árbol de herencia. A continuación se presenta un posible árbol de herencia de la clase Persona.

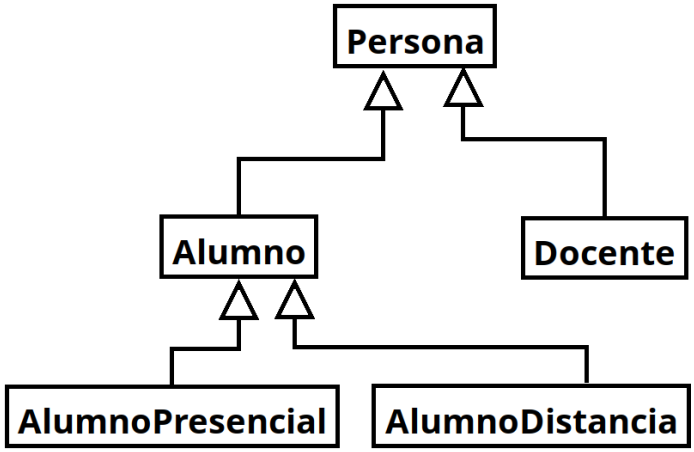

Imagen 2: Elaboración propia

Miembros de las clases derivadas

Los objetos de la clase derivada poseen todos los miembros definidos en la clase base y a su vez la clase derivada puede poseer otros miembros (atributos y métodos).

Por este motivo la cláusula que establece la herencia se denomina extends (extiende), ya la clase derivada "es más grande" que la clase base. En el código de la clase derivada se puede acceder a los miembros visibles de la clase base como si fueran miembros propios directamente o a través de la referencia this.

Además de los modificadores de acceso public y private se le agrega el modificador protected. Un miembro marcado como protected se lo denomina protegido y funciona como accesible desde las clases derivadas pero no desde otras. Por estar ocultos desde otras clases no relacionadas por medio de la herencia, a los miembros protegidos se les requiere programar los métodos get y set como si fueran privados.

Cláusula extends

Para establecer la herencia en el código debe agregarse al bloque class de la derivada la cláusula extends y a continuación el nombre de la clase base.

Clase base	
public class Base {...}	public class Persona {}
Clase derivada	
public class Derivada extends Base {...}	public class Alumno extends Persona {}

Tabla 1: Elaboración propia

Todas las clases que no posean cláusula extends no establecen ninguna clase base, en este caso derivan implícitamente de una clase especial llamada Object.

Referencia super

La referencia super es similar a this, pero sólo puede acceder a los miembros heredados, es decir, aquellos que están definidos en la clase base. Sólo debe usarse cuando hay que diferenciar entre un método programado en la clase base que puede generar conflicto con los de la clase derivada.

Tiene dos usos principales:

- Para llamar a los constructores de la clase base.

- Para llamar a la versión existente en la clase base de un método sobreescrito en la derivada.

El caso más habitual se da con el método toString, ya que cada clase posee una versión del mismo, y dado que en la clase derivada se puede requerir acceder al toString de la base, se necesita diferenciar que no se está queriendo ejecutar el toString de la misma clase derivada. En esta situación invocando super.toString(), se está indicando explícitamente que se debe ejecutar la versión del método que posee super, que es el mismo objeto this, pero “con forma” de la base.

En el siguiente ejemplo, el método toString de la clase Persona retorna la concatenación de los atributos correspondientes al nombre y apellido de la misma. Por su lado, la clase Alumno pretende reutilizar la versión de toString de la clase Persona, y dado que Alumno hereda de Persona, solamente debería incluir una llamada al toString de la base.

```
public class Persona {
private String nombre, apellido;
```

```
public String toString() {
return nombre + " " + apellido;
}

}
public class Alumno extends Persona {
private int legajo;
// otros atributos y métodos

public String toString()
{
return toString() + " "  + legajo;
}
}
```

Pero el código anterior presenta un problema: la llamada a toString inicia una nueva ejecución de ese mismo método de la clase Alumno (en lugar de llamar al toString de Persona). Esa segunda llamada comienza ejecutando nuevamente toString y así continúa llamandose el método a sí mismo generando un ciclo infinito del cual el programa no puede salir nunca.

Para evitar este problema es necesario que el toString de Alumno indique explícitamente que se quiere ejecutar el toString de la clase base usando la referencia super:

```
public String toString()
{
return super.toString() + " "  + legajo;
}
```

Herencia y constructores

Los constructures son los únicos métodos que no se heredan y por ello las clases derivadas deben proveer sus propios constructores.

Cuando se instancia un objeto de una clase derivada, se usa new con el nombre de la derivada y los parámetros de algún constructor de la misma. En ese punto primero se ejecuta el constructor de la clase base y luego continúa la ejecución del constructor de la derivada.

Si el constructor de la clase derivada no llama a ningún constructor de la clase base, se ejecuta el constructor sin parámetros de ésta. Mientras que para invocar al constructor de la clase base en forma explícita se agrega como primera línea del constructor de la clase derivada una llamada a super() y para usar un constructor de la clase base con parámetros se indica dentro de la lista de parámetros de super().

El constructor de la derivada requiere parámetros tanto para los atributos propios como para los de la clase base.

Clase base	

public class Base {
  public Base(tipo param1, tipo param2) {
  this.atributo1 = param1;
  this.atributo2 = param2;
  }
}
	public class Persona {
	  public Persona(int documento, String nombre) {
	  this.documento = documento;
	  this.nombre = nombre;
	  }
	}

Clase derivada	

public class Derivada extends Base {
  public Derivada (tipo param1, tipo param2, tipo param3){
  super(param1,param2);
  this.atributo3 = param3;
  }
}
	public class Alumno extends Persona {
	  public Alumno(int documento, String nombre, int legajo) {
	  super(documento, nombre);
	  this.legajo = legajo;
	  }
	}

Tabla 2: Elaboración propia

Ejercicio

En un comercio se mantienen los datos de sus clientes, algunos de los cuales son de confianza suficiente para comprar al fiado. Programar dos clases llamadas Cliente y ClientePreferencial respectivamente aplicando herencia.

Sobreescribir el método toString e incluir constructores con parámetros en cada una de ellas.

De cada cliente se conoce su número, nombre y teléfono, mientras que de los clientes preferenciales se conoce adicionalmente su saldo, límite (saldo máximo) y su domicilio.

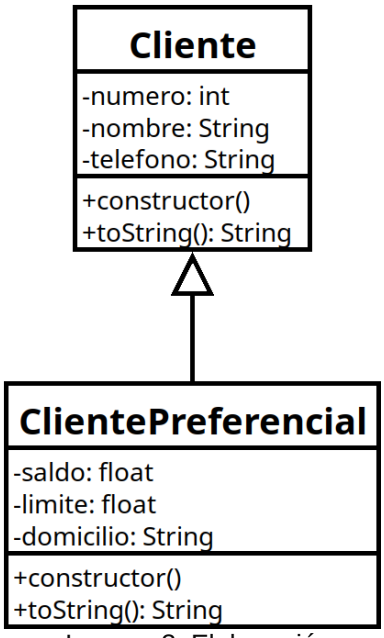

Imagen 3: Elaboración propia

```
package clientes;

public class Cliente {
    protected String nombre;
    protected String telefono;
    protected int numero;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
```

```
    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    public int getNumero() {
        return numero;
    }

    public void setNumero(int numero) {
        this.numero = numero;
    }

    public Cliente() {
    }

    public Cliente(String nombre, String telefono, int cuit) {
        this.nombre = nombre;
        this.telefono = telefono;
        this.numero = cuit;
    }

    @Override
    public String toString() {
        return "Cliente{" + "nombre=" + nombre + ", telefono=" + telefono + ",
cuit=" + numero + '}';
    }

}


package clientes;

public class ClientePreferencial extends Cliente {
    private String domicilio;
    private float saldo;
```

```
    private float limite;

    public String getDomicilio() {
        return domicilio;
    }

    public void setDomicilio(String domicilio) {
        this.domicilio = domicilio;
    }

    public float getSaldo() {
        return saldo;
    }

    public void setSaldo(float saldo) {
        this.saldo = saldo;
    }

    public float getLimite() {
        return limite;
    }

    public void setLimite(float limite) {
        this.limite = limite;
    }

    public ClientePreferencial(String nombre, String telefono, int numero, String domicilio, float saldo, float limite) {
        super(nombre, telefono, numero);
        this.domicilio = domicilio;
        this.saldo = saldo;
        this.limite = limite;
    }

    @Override
    public String toString() {
        return "ClientePreferencial{" + "domicilio=" + domicilio + ", saldo=" + saldo + ", limite=" + limite + '}';
    }
}
```

2. POLIMORFISMO

El mecanismo de herencia es uno de los pilares fundamentales en los que se basa la programación orientada a objetos. Es un mecanismo que permite definir nuevas clases a partir de otras ya definidas de modo que si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera (la clase derivada) será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda (la clase base).

Pero además la herencia habilita el tercer pilar fundamental de la programación orientada a objetos: el polimorfismo.

Este concepto si bien es relativamente más difícil de comprender y asimilar resulta central en la programación ya que permite entre otras ventajas que los programas puedan crecer en sus alcances sin obligar a una gran reescritura de código.

El polimorfismo se apoya fuertemente en que la relación de herencia siempre establece que un objeto de la derivada “es un” objeto de la base. Con el ejemplo de las personas y alumnos, establecida la relación de herencia, siempre se puede decir que “un alumno es una persona”.

Referencias a la clase base

En los programas, el significado de “es un” se traduce en una característica que inicialmente parece contraintuitiva: una variable de tipo referencia puede apuntar a objetos de otras clases!

En realidad, toda variable de tipo referencia puede apuntar a objetos de la misma clase de la definición y a objetos de cualquier otra clase derivada, a cualquier nivel del árbol de herencia. Con el ejemplo de los alumnos, una variable de tipo Persona puede referenciar a cualquier objeto Persona pero tambien a objetos Alumno, ya que los alumnos efectivamente **son objetos Persona**.

```
Persona p = new Persona(“Juan Perez”);
Persona q = new Alumno(100394,”Pedro Gomez”);

System.out.println(p.toString());
System.out.println(q.toString());
```

La porción de código presentada es totalmente válida, p y q son referencias a objetos de tipo persona y por ello pueden apuntar a objetos Persona o Alumno (o de cualquiera otra de sus derivadas, como Docente, AlumnoPresencial, AlumnoDistancia o las que se puedan programar en el futuro).

Las llamadas a toString() ejecutan efectivamente el toString() de Persona. Sin embargo, ¿qué ocurre si en la clase Alumno también se agregó un método toString()? Ambas clases poseen el mismo método, con el mismo nombre y parámetros.

Sobreescritura de métodos

Cuando se plantea una relación de herencia, se permite realizar una operación denominada **sobreescritura**, la cual consiste en que las clases derivadas pueden **reemplazar** la implementación de cualquiera de los métodos de la clase base, para que su funcionamiento tenga sentido o sea más adecuado para la derivada. El caso más evidente se da con el método toString, ya que en la clase base sólo retorna los valores de sus atributos propios, y si en las derivadas se agregaron atributos nuevos, ellas requieren otra versión de toString, que muestre los valores de esos atributos o de todos los que posee.

La sobre escritura permite que una clase derivada provea una nueva implementación de un método heredado, con las características o funcionalidad específica que ella necesite. En el ejemplo de las personas, es razonable que el toString de los alumnos muestre sus legajos; mientras que el toString de las personas no podría ya que no posee un atributo donde almacenar ningún legajo.

Los métodos sobre escritos pueden llevar una anotación que los marca como tales. Para ello debe indicarse ANTES del modificador de acceso (normalmente en la línea anterior) la nomeclatura **@Override**.

Esta anotación le indica al compilador que el método efectivamente está sobreescribiendo a otro presente en la clase base (en realidad en cualquier clase existente en el camino de herencias entre la clase object y la clase en cuestión.

La anotación es opcional pero tiene una utilidad muy importante. Cuando el compilador la encuentra puede verificar que efectivamente en la clase base exista dicho método para que pueda ser sobreescrito; en caso contrario el compilador muestra un mensaje de error, ya que no puede sobreescribirse un método inexistente.

Si no se indica @Override, se corre el riesgo de indicar un nombre incorrecto del método y el compilador creerá que se intenta agregar un método nuevo en lugar de una sobreescritura.

El próximo ejemplo muestra tal situación:

```
public class Docente {


public String ToString() {
...
}
}
```

En este caso, el método ToString fue programado erróneamente con la inicial en mayúscula. Dado que cumple correctamente con la sintaxis del lenguaje, el compilador no puede detectar el error. La consecuencia de esto es que la clase Docente queda con un método toString (lo trae heredado de Persona) y otro llamado ToString, que no sobreescribe al toString de Persona porque tienen nombres diferentes. Cuando el programa intente usar el método, se ejecuta la versión de toString de la clase base y nunca la versión de la derivada, ya que es otro método diferente.

Si se agrega la anotación @Override el compilador detecta que se intentó sobreescribir un método ToString inexistente en la clase base y sí puede interrumpir presentando un error de compilación.

Métodos polimórficos

Un método sobre eescrito en una clase derivada posee una característica esencial: cada vez que se lo invoque se ejecuta la versión del mismo que corresponde con la instancia y no con la variable que apunta al objeto.

En el código presentando anteriormente, la llamada a p.toString indudablemente ejecuta la versión correspondiente a la clase Persona, ya que la variable p está declarada como persona y a ella se le asignó el resultado de new Persona, es decir, una instancia de Persona.

Pero en la línea siguiente, lo que ocurre con q.toString() parece no tener sentido: a pesar de que q es una variable de tipo Persona, **se ejecuta la versión de toString correspondiente a la clase Alumno!** Es decir que a pesar que se q de tipo Persona, como apunta a un objeto Alumno, se ejecutan los métodos de esa clase, sin tener que indicarlo en el código.

Esta característica es central en la programación orientada a objetos y funciona de esta manera en todos los lenguajes de programación. Un método sobreescrito que es invocado mediante referencias a la clase base se denomina polimórfico. “Polimórfico” puede interpretarse como “que tiene muchas formas” y en

la programación se refiere a las distintas versiones que un mismo método puede tener entre las diferentes clases derivadas.

Estrictamente esta denominación cobra mayor significado cuando una variable de tipo referencia cambia de valor, apuntando a diferentes objetos. Si se ejecuta el mismo método cada vez que apunta a cada objeto diferente, la misma instrucción puede estar ejecutando diferentes versiones del método, por lo tanto, funcionando diferente cada vez. En el siguiente ejemplo, ¿qué ocurre en cada println?

```
Persona p;
p = new Persona(“Juan Perez”);
System.out.println(p.toString());

p = new Alumno(100394,”Pedro Gomez”);
System.out.println(p.toString());
```

Las dos invocaciones a p.toString ejecutan porciones de código diferentes cada vez, la primera ejecutando el toString de Persona y la segunda el toString de Alumno. De ahí la denominación de polimorfismo, ya que a pesar de ser exactamente la misma línea de código, en cada llamada funciona de diferentes formas.

El polimorfismo parece ir en contra de la lógica, ya que el programador no necesariamente sabe qué es lo que va a ocurrir cuando ejecuta una llamada a un método polimórfico. Sin embargo esto es deseable, ya que siempre se va a ejecutar la version correcta. Sin polimorfismo el programa necesitaría una serie de condiciones con if o switch que determine a cuál versión invocar. Gracias al polimorfismo el programador no tiene que preocuparse con nada, únicamente ejecuta el método polimórfico, si una derivada lo sobreescribe, se ejecuta la versión correcta y si no, se ejecuta la versión de la clase base.

En el siguiente diagrama, casi todas las clases han provisto su propia versión del método toString(), por lo tanto, cualquier referencia a Persona puede ejecutar toString.

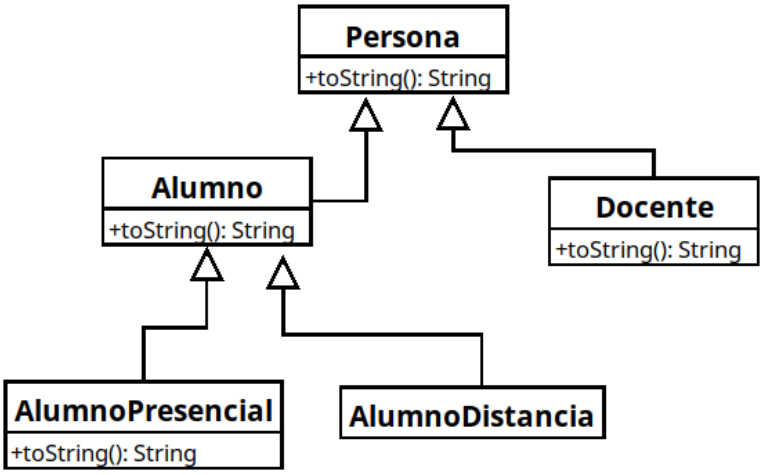

Imagen 4: Elaboración propia

Incluso en el caso de la clase AlumnoPresencial, si una variable de tipo Persona apunta a un AlumnoPresencial, también puede ejecutar toString. Dado que la clase no provee una implementación propia del método, se ejecuta el toString de la clase base, es decir, Alumno.

Identificación de la clase de un objeto. Operador is

Cuando se trabaja con referencias a la clase base, es muy habitual necesitar conocer la clase concreta del objeto apuntado, ya que puede ser instancia de la clase base o de cualquiera de las derivadas. Para obtener esa información java provee un operador llamado **is** que recibe la variable y un nombre de clase retornando verdadero únicamente si el objeto apuntado por la variable es instancia de la clase o de alguna de sus derivadas. Nunca tiene sentido consultar is con la clase base, ya que obligatoriamente retorna verdadero siempre, a causa de esto este operador es útil sólo para consultar por alguna de las derivadas.

Operador is	

referencia is clase	Persona p = new Persona();
	Persona q = new Alumno();
	Alumno r = new Alumno();
	if (p is Persona) // true
	if (p is Alumno) // false
	if (p is Docente) // false
	
	if (q is Persona) // true
	if (q is Alumno) // true
	if (q is Docente) // false
	
	if (r is Persona) // true
	if (r is Alumno) // true
	if (r is Docente) // false

Tabla 3: Elaboración propia

Casting hacia abajo (downcasting)

Cuando una referencia a una clase base apunta a un objeto de una derivada el compilador permite acceder únicamente a los miembros de la clase base, a pesar de que el objeto sea instancia de una derivada, la cual puede ofrecer más atributos y métodos.

Sin embargo, en muchas ocasiones se requiere poder tener acceso a los miembros agregados en la derivada y esta imposibilidad debe ser resuelta de alguna manera.

Java provee para esta situación un mecanismo denominado downcasting, que se puede traducir como casting hacia abajo.

A una referencia a la clase base se le puede aplicar una conversión explícita (casting) convirtiendo con el nombre de una clase derivada. El resultado de la conversión es el mismo objeto, que no sufre ninguna modificación, pero que el compilador sí interpreta como objeto de la derivada. El uso más habitual de esta conversión es la de asignar en una referencia de la clase derivada. Esta última sí tendrá acceso a los miembros agregados en la herencia.

En el siguiente ejemplo se logra imprimir el legajo de un alumno a pesar de estar siendo apuntado por una variable de tipo Persona:

```
Persona p = new Alumno(1234, “Juan”, “Perez”);
System.out.println(p.getLegajo()); // Error de compilación, las personas no tienen atributo legajo.

Alumno q = (Alumno) p; // Conversión explícita de la referencia p a Alumno, el resultado de la conversión se asigna en la variable q.
System.out.println(q.getLegajo()); // Válido, imprime 1234.
// Las variables q y p apuntan ambas al mismo objeto, pero con q el compilador reconoce que posee el método getLegajo.

Docente r = (Docente) p; // Error en tiempo de ejecución, p no es un Docente.
```

En la última línea del ejemplo anterior se intenta convertir la referencia p a una referencia a Docente. Dado que p apunta a un alumno, esa conversión lanza una excepción de tipo ClassCastException, la cual si no es capturada y manipulada adecuadamente provocará una interrupción irregular del programa y la presentación de un mensaje de error al usuario.

Para evitar el riesgo de que el programa finalice de esta manera es recomendable que siempre que se intente un downcasting se verifique con el uso del operador is que la referencia efectivamente sea instancia de la clase a la que se intenta convertir.

Corrigiendo esto en el ejemplo anterior, el programa quedaría como se presenta a continuación:

```
Persona p = new Alumno(1234, “Juan”, “Perez”);
System.out.println(p.getLegajo());

if (p is Alumno) {
Alumno q = (Alumno) p;
System.out.println(q.getLegajo());
```

```
}

if (p is docente) {
Docente r = (Docente) p;
....
}
```

Ejercicio

En una empresa hay tres tipos de empleados, cada uno de ellos con diferentes fórmulas para el cálculo de sus sueldos.

- Obrero: cobra un monto fijo por cada día trabajado igual al sueldo básico dividido en 22.
- Administrativo: cobra un monto fijo por mes más un 13 % si cumplió con el presentismo.
- Vendedor: cobra un monto fijo por mes más una comisión de 1 % del total de sus ventas.

Se requiere representar esta situación con una o más clases aplicando polimorfismo.

Solución:

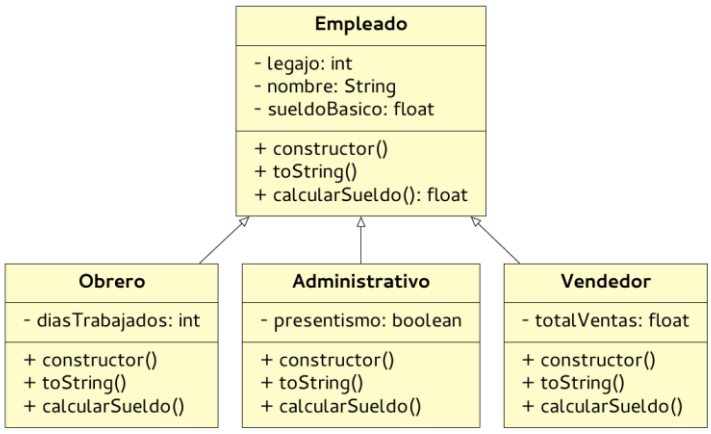

Imagen 5: Elaboración propia

```
package empleados;
```

```
public class Empleado {
    protected int legajo;
    protected String nombre;
    protected float sueldoBasico;

    public int getLegajo() {
        return legajo;
    }

    public void setLegajo(int legajo) {
        this.legajo = legajo;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public float getSueldoBasico() {
        return sueldoBasico;
    }

    public void setSueldoBasico(float sueldoBasico) {
        this.sueldoBasico = sueldoBasico;
    }

    public Empleado(int legajo, String nombre, float sueldoBasico) {
        this.legajo = legajo;
        this.nombre = nombre;
        this.sueldoBasico = sueldoBasico;
    }

    @Override
    public String toString() {
        return "Empleado{" + "legajo=" + legajo + ", nombre=" + nombre + ",
sueldoBasico=" + sueldoBasico + '}';
```

```
    }

    public float calcularSueldo() {
return 0;
}
}
```

```
package empleados;

public class Obrero extends Empleado {
    private int diasTrabajados;

    public Obrero(int legajo, String nombre, float sueldoBasico, int diasTrabajados) {
        super(legajo, nombre, sueldoBasico);
        this.diasTrabajados = diasTrabajados;
    }

    @Override
    public String toString() {
        return "Obrero{" + "diasTrabajados=" + diasTrabajados + '}';
    }

    @Override
    public float calcularSueldo() {
        return sueldoBasico / 22 * diasTrabajados;
    }
}
```

```
package empleados;

public class Administrativo extends Empleado {
    private boolean presentismo;

    public Administrativo(int legajo, String nombre, float sueldoBasico, boolean presentismo) {
        super(legajo, nombre, sueldoBasico);
        this.presentismo = presentismo;
    }
```

```
    @Override
    public String toString() {
        return "Administrativo{" + super.toString() + "presentismo=" + presentismo + '}';
    }

    @Override
    public float calcularSueldo() {
        float sueldo = sueldoBasico;
        if (presentismo)
            sueldo = sueldo * 1.13f;
        return sueldo;
    }
}

package empleados;

public class Vendedor extends Empleado {
    private float totalVentas;

    public float getTotalVentas() {
        return totalVentas;
    }

    public void setTotalVentas(float totalVentas) {
        this.totalVentas = totalVentas;
    }

    public Vendedor(int legajo, String nombre, float sueldoBasico, float totalVentas) {
        super(legajo, nombre, sueldoBasico);
        this.totalVentas = totalVentas;
    }

    @Override
    public String toString() {
        return "Vendedor{"+ super.toString() + "totalVentas=" + totalVentas + '}';
    }
```

```
@Override
public float calcularSueldo() {
    return sueldoBasico + 0.01f * totalVentas;
}
```

3. CLASES ABSTRACTAS

Cuando se plantea una relación de herencia es habitual identificar clases bases que no deban ser instanciadas, es decir, que nunca existan objetos de la clase, si no que únicamente se puedan crear objetos de las derivadas. Esta situación puede ser motivada por diversos factores, tales como:

- Concepto de las clases: puede ocurrir que no tenga sentido la existencia de objetos de las clases bases ya que representan una entidad demasiado genérica. Por ejemplo una clase Persona, que sea heredada por las clases Vendedor y Cliente. Si se está programando un punto de venta para un comercio, poco sentido tendría que se creen objetos de la clase persona y sólo sería útil crear o bien vendedores o bien clientes. Generalmente cuando se identifica esta necesidad, la clase base fue creada para reutilizar código que sin la relación de herencia sería duplicado.

- Sobreescritura de métodos: puede ocurrir que un método no pueda ser escrito en la clase base. En el ejemplo de los empleados, se plantea la fórmula de cálculo de los obreros, de los vendedores y de los administrativos pero no hay una fórmula genérica, ya que cada una de las clases derivadas posee la suya. En ese caso, si se pudieran crear objetos de la clase Empleado, el método de cálculo del sueldo no podría retornar nada, entonces es preferible directamente impedir la instanciación.

Para resolver ambos problemas se puede plantear que una clase base NUNCA pueda ser instanciada. Las clases con esta característica se denominan **clases abstractas**. Conseguir que una clase se convierta en abstracta requiere únicamente agregarle el modificador **abstract** antes de la palabra clave class:

```
public abstract class Persona {}
```

Una clase marcada como abstracta no puede ser instanciada, el compilador logra esto prohibiendo la ejecución de new con esa clase. Sin embargo, no se prohibe crear variables de tipo referencia a esa clase y a causa de eso tampoco se prohibe crear arreglos de la clase base. Por lo tanto el polimorfismo sigue siendo posible sin ninguna modificación.

Por otro lado, cuando una clase es abstracta se permite que uno o varios métodos no tengan implementación, es decir que no presenten código fuente. Un método sin código fuente se denomina **método abstracto**. Nuevamente para lograr que un método sea abstracto se le agrega el modificador **abstract** antes del tipo de retorno:

```
…
   public abstract float calcularSueldo();
…
```

Es importante recalcar que dado que los métodos abstractos no tienen código, en lugar de las llaves debe indicarse un punto y coma.

Al marcar un método como abstracto, todas las clases derivadas deben obligatoriamente realizar la sobreescritura para proveerle código con la lógica que le corresponda a la misma.

Debe recordarse siempre que para marcar un método como abstracto la clase también debe ser abstracta, sin embargo una clase abstracta no necesariamente tiene métodos abstractos. Es válido y muy habitual que una clase abstracta no tenga ningún método abstracto.

En el anterior ejemplo de los empleados, el método calcularSueldo retorna 0. Esto es muy peligroso ya que si en el futuro se agrega una nueva clase derivada y no se sobreescribe el método, a todos los empleados de la nueva clase se le pagará $0 de sueldo. Para evitar este error, si se marcan abstractos tanto el método como la clase, al aparecer la nueva derivada el compilador exigirá la programación del método correspondiente.

En el ejercicio de los empleados, la clase Empleado debería ser abstracta y su método calcularSueldo también. Las clases derivadas no sufren cambios adicionales y la clase base quedaría de la siguiente manera:

```
public abstract class Empleado {
...
public abstract float calcularSueldo();
...
}
```

Se pueden encontrar mayores ejemplos en Ceballos (2010) y Corso (2012).

Colecciones de referencias a la clase base

Como se presentó en la unidad 2, los arreglos o las colecciones pueden ser definidos con el nombre de una clase para poder almacenar objetos. Dado que los arreglos y colecciónes de objetos almacenan referencias, si una colección es definida con una clase que posee derivadas, la misma puede alojar objetos de dicha clase o de cualquiera de las clases que hereden de ella.
Son muchas las situaciones en las que puede ser necesario utilizar esta estrategia. Por ejemplo, en el ejemplo de los alumnos y docentes, para realizar las inscripciones es razonable mantener un ArrayList de alumnos, el cual permitirá guardar todos los alumnos inscriptos, tanto los de la modalidad presencial (clase AlumnoPresencial) como los de la modalidad a distancia (clase AlumnoDistancia).

El diagrama que se presenta a continuación muestra un arreglo de alumnos en el que se cargaron varias instancias de alumnos de distintas clases, todas ellas derivadas de la misma clase base:

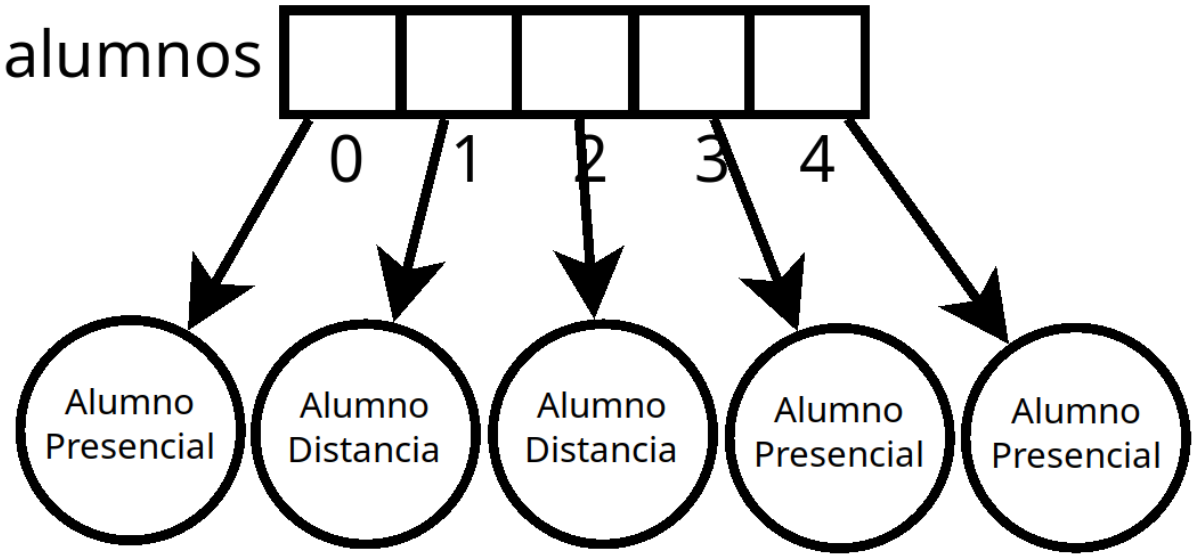

Imagen 6: Elaboración propia

Ese arreglo puede ser obtenido con la siguiente porción de código fuente:

```
...
Alumno []alumnos = new Alumno[5];
alumnos[0] = new AlumnoPresencial();
alumnos[1] = new AlumnoDistancia();
alumnos[2] = new AlumnoDistancia();
alumnos[3] = new AlumnoPresencial();
```

```
alumnos[4] = new AlumnoPresencial();
...
```

De la misma manera, si se prefiere usar una colección tal como ArrayList, se necesitaría la siguiente porcion de programa:

```
...
ArrayList<Alumno> alumnos = new ArrayList<>();
alumnos.add(new AlumnoPresencial());
alumnos.add(new AlumnoDistancia());
alumnos.add(new AlumnoDistancia());
alumnos.add(new AlumnoPresencial());
alumnos.add(new AlumnoPresencial());
...
```

En ambos casos se obtiene la posibilidad de aplicar el polimorfismo de una manera sumamente práctica, ya que si se recorre el arreglo o la colección y se invoca a un método polimórfico, cada llamada puede potencialmente llamar a un método diferente, sin necesidad de verificar la clase de cada objeto. Así por ejemplo, para mostrar un listado de los alumnos, solo se require un ciclo simple:

```
for (Alumno a: alumnos) {
System.out.println(a.toString());
}
```

En el recorrido presentado, cada vuelta posee la variable a apuntando a cada uno de los objetos alumnos, algunos de los cuales son alumnos presenciales y otros son alumnos a distancia. Al invocar al método toString, se ejecuta en cada vuelta la **versión correcta** del método, es decir, la sobreescritura que corresponda a cada clase, sin necesidad de verificar con el operador is o de hacer conversiones hacia abajo.

La potencia del polimorfismo comienza a vislumbrarse con situaciones como la presentada, ya que se simplifican drásticamente los accesos a los métodos polimórficos aún en el caso de que en el futuro surjan nuevas clases derivadas de Alumno. Cuando eso ocurra, el recorrido que muestra el listado seguirá funcionando correctamente sin ninguna modificación, invocando al las nuevas versiones de toString.

Lógicamente los beneficios de esta técnica son más evidentes con otros métodos que manipulen o procesen los datos de los objetos.

Tomando como ejemplo el ejercicio de los empleados, si un programa mantiene un arraylist con todos los empleados, el proceso de calcular los sueldos a pagarles a todos requiere únicamente recorrer la colección llamando al método calcularSueldo. Cada objeto calculará su sueldo con la fórmula que le corresponda según su tipo, sin necesidad de diferenciar entre obreros, vendedores o administrativos. Y si en el futuro se agrega otro tipo de empleados, tales como los gerentes con su propia fórmula para el cálculo del sueldo, no se necesita ninguna modificación en el proceso:

```
float suma = 0;
for (Empleado e: empleados) {
suma += e.calcularSueldo();
// No importa de qué tipo de empleado sea el objeto referenciado por e,
// siempre se ejecuta la sobreescritura correcta del método.
}
System.out.println(suma);
```

Ejercicio:

En base al ejercicio de empleados de la clase 7, hacer un programa que permita cargar los datos de todos los empleados de una empresa en un arraylist y efectúe la liquidación de los sueldos, mostrando un listado de todos los empleados y el sueldo neto que les corresponde cobrar. Finalmente mostrar el total a pagar en sueldos para cada tipo de empleados y el total general.

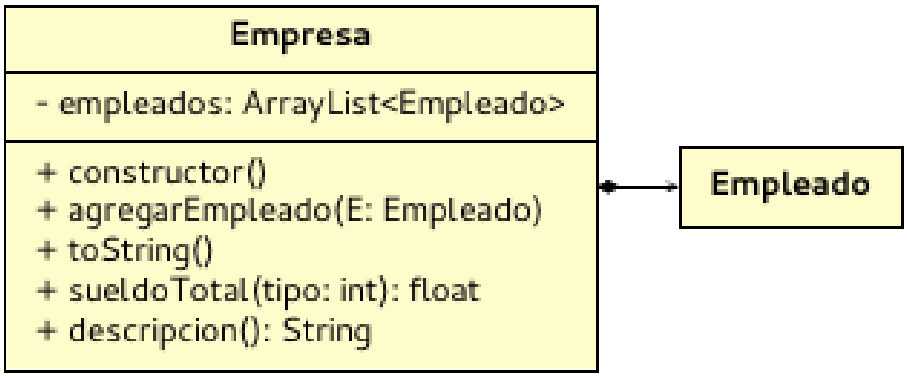

Imagen 7: Elaboración propia

```
public class Empresa {

    private ArrayList<Empleado> empleados;

    public Empresa() {
        this.empleados = new ArrayList<>();
    }

    public void agregarEmpleado(Empleado E) {
        empleados.add(E);
    }

    public float sueldoTipo(int opcion) {
        float sueldoTotal = 0;
        for (Empleado E : empleados) {
            float sueldo = E.calcularSueldo();
            if (opcion == 1 && E instanceof Obrero)
                sueldoTotal += sueldo;
            else if (opcion == 2 && E instanceof Administrativo)
                sueldoTotal += sueldo;
            else if (opcion == 3 && E instanceof Vendedor)
```

```
                sueldoTotal += sueldo;
            else
                sueldoTotal += sueldo;
        }
        return sueldoTotal;
    }

    public float sueldoTotal() {
        float sueldoTotal = 0;
        for (Empleado E : empleados) {
            sueldoTotal += E.calcularSueldo();
        }
        return sueldoTotal;
    }

    public String descripcion() {
        String desc = "";
        for (Empleado E : empleados) {
            desc += E.toString();
        }
        return desc;
    }

    public ArrayList<Empleado> getEmpleados() {
        return empleados;
    }
}

public class Principal {
```

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int opcion2;

    Empresa C = new Empresa();

    do {
        Empleado nuevo;
        System.out.println("Ingrese el legajo");
        int legajo = sc.nextInt();
        System.out.println("Ingrese el nombre");
        String nombre = sc.next();
        System.out.println("Ingrese el sueldo basico");
        float sueldo = sc.nextFloat();
        System.out.println("Ingrese el tipo de empleado" + "\n"
                + "1. Obrero" + "\n" + "2. Administrativo" + "\n"
                + "3. Vendedor");
        int opcion = sc.nextInt();
        switch (opcion) {
            case 1: {
                System.out.println("Ingrese la cantidad de dias trabajados");
                int cantTrabajados = sc.nextInt();
                nuevo = new Obrero(legajo, nombre, sueldo, cantTrabajados);
                break;
            }
            case 2: {
                System.out.println("Tiene presentismo");
                boolean presentismo = sc.nextBoolean();
                nuevo = new Administrativo(legajo, nombre, sueldo, presentismo);
                break;
```

```
            }
            case 3: {
                System.out.println("Ingrese la cantidad de ventas");
                int cantVentas = sc.nextInt();
                nuevo = new Vendedor(legajo, nombre, sueldo, cantVentas);
                break;
            }
        }
        C.agregarEmpleado(nuevo);
        System.out.println("¿Desea agregar un nuevo empleado?");
        System.out.println("Ingrese 1 para cargar otro y 0 para salir");
        opcion2 = sc.nextInt();
    } while (opcion2 != 0);

    System.out.println("Cantidad de empleados: " + C.getEmpleados().size());
    System.out.println("Los empleados son: " + C.descripcion());
    System.out.println("El total a pagar en sueldos: " + C.sueldoTotal(opcion));
    System.out.println("Ingrese el tipo de empleado para conocer el sueldo \n"
            + "1. Obrero\n2. Administrativo\n3. Vendedor");
    int opcion = sc.nextInt();
    System.out.println("Sueldo total del tipo indicado: " + C.sueldoTipo(opcion));
  }
}
```

BIBLIOGRAFÍA

- Ceballos Sierra, F. (2010). *Java 2. Curso de Programación*. 4ta Edición. Madrid, España. RA-MA Editorial.
- Corso, C; Colaccioppo, N. (2012). "*Apunte teórico-práctico de Laboratorio de Computación III*". Córdoba, Argentina. Edición digital UTN-FRC.