7.F. Polimorfismo.

Sitio: Formación Profesional a Distancia
Curso: Programación
Libro: 7.F. Polimorfismo.
Imprimido por: Iván Jiménez Utiel
Día: lunes, 10 de febrero de 2020, 15:52

Tabla de contenidos

1. Polimorfismo.

El **polimorfismo** es otro de los grandes pilares sobre los que se sustenta la **Programación Orientada a Objetos** (junto con la **encapsulación** y la **herencia**). Se trata nuevamente de otra forma más de establecer diferencias entre interfaz e implementación, es decir, entre **el qué** y **el cómo**.

La **encapsulación** te ha permitido agrupar **características** (**atributos**) y **comportamientos** (**métodos**) dentro de una misma unidad (**clase**), pudiendo darles un mayor o menor componente de **visibilidad**, y permitiendo separar al máximo posible la **interfaz** de la **implementación**. Por otro lado la **herencia** te ha proporcionado la posibilidad de tratar a los objetos como pertenecientes a una **jerarquía de clases**. Esta capacidad va a ser fundamental a la hora de poder manipular muchos posibles objetos de clases diferentes como si fueran de la misma clase (**polimorfismo**).

El **polimorfismo** te va a permitir mejorar la **organización** y la **legibilidad** del código así como la posibilidad de desarrollar aplicaciones que sean más fáciles de ampliar a la hora de incorporar nuevas funcionalidades. Si la implementación y la utilización de las clases es lo suficientemente genérica y extensible será más sencillo poder volver a este código para incluir nuevos requerimientos.

Autoevaluación

¿Cuál de las siguientes características dirías que no es una de las que se suelen considerar como uno de los tres grandes pilares de la Programación Orientada a Objetos?

- Recursividad.
- Herencia.
- Polimorfismo.
- Encapsulación.

1.1. Concepto de polimorfismo.

El **polimorfismo** consiste en la capacidad de poder utilizar una referencia a un objeto de una determinada clase como si fuera de otra clase (en concreto una **subclase**). Es una manera de decir que una clase podría tener varias (poli) formas (morfismo).

Un método "**polimórfico**" ofrece la posibilidad de ser distinguido (saber a qué clase pertenece) en **tiempo de ejecución** en lugar de en **tiempo de compilación**. Para poder hacer algo así es necesario utilizar métodos que pertenecen a una **superclase** y que en cada **subclase** se implementan de una forma en particular. En **tiempo de compilación** se invocará al método sin saber exactamente si será el de una subclase u otra (pues se está invocando al de la **superclase**). Sólo en **tiempo de ejecución** (una vez instanciada una u otra **subclase**) se conocerá realmente qué método (de qué **subclase**) es el que finalmente va a ser invocado.

Esta forma de trabajar te va a permitir hasta cierto punto "desentenderte" del tipo de objeto **específico** (**subclase**) para centrarte en el tipo de objeto **genérico** (**superclase**). De este modo podrás manipular objetos hasta cierto punto "desconocidos" en tiempo de compilación y que sólo durante la ejecución del programa se sabrá exactamente de qué tipo de objeto (**subclase**) se trata.

El **polimorfismo** ofrece la **posibilidad de que toda referencia a un objeto de una superclase pueda tomar la forma de una referencia a un objeto de una de sus subclases**. Esto te va a permitir escribir programas que procesen objetos de clases que formen parte de la misma jerarquía como si todos fueran objetos de sus **superclases**.

El **polimorfismo** puede llevarse a cabo tanto con **superclases** (abstractas o no) como con **interfaces**.

Dada una **superclase** **X**, con un método **m**, y dos **subclases** **A** y **B**, que redefinen ese método **m**, podrías declarar un objeto **O** de tipo **X** que en durante la **ejecución** podrá ser de tipo **A** o de tipo **B** (algo desconocido en **tiempo de compilación**). Esto significa que al invocarse el método **m** de **X** (**superclase**), se estará en realidad invocando al método **m** de **A** o de **B** (alguna de sus **subclases**). Por ejemplo:

```
// Declaración de una referencia a un objeto de tipo X
```

```
ClaseX obj;  // Objeto de tipo X (superclase)
```

```
...
```

```

```

```
// Zona del programa donde se instancia un objeto de tipo A (subclase) y se le
asigna a la referencia obj.
```

```

```

```
// La variable obj adquiere la forma de la subclase A.
```

```
obj = new ClaseA ();
```

```
...
```

```

```

```
// Otra zona del programa.
```

```

```

```
// Aquí se instancia un objeto de tipo B (subclase) y se le asigna a la referencia obj.

// La variable obj adquiere la forma de la subclase B.
obj = new ClaseB ();
...

// Zona donde se utiliza el método m sin saber realmente qué subclase se está utilizando.

// (Sólo se sabrá durante la ejecución del programa)
obj.m ()  // Llamada al método m (sin saber si será el método m de A o de B).
...
```

Imagina que estás trabajando con las clases `Alumno` y `Profesor` y que en determinada zona del código podrías tener objetos, tanto de un tipo como de otro, pero eso sólo se sabrá según vaya discurriendo la ejecución del programa. En algunos casos, es posible que un determinado objeto pudiera ser de la clase `Alumno` y en otros de la clase `Profesor`, pero en cualquier caso serán objetos de la clase `Persona`. Eso significa que la llamada a un método de la clase `Persona` (por ejemplo `devolverContenidoString`) en realidad será en unos casos a un método (con el mismo nombre) de la clase `Alumno` y, en otros, a un método (con el mismo nombre también) de la clase `Profesor`. Esto será posible hacerlo gracias a la **ligadura dinámica**.

Autoevaluación

El polimorfismo ofrece la posibilidad de que toda referencia a un objeto de una clase A pueda tomar la forma de una referencia a un objeto de cualquier otra clase B. ¿Verdadero o Falso?

- Verdadero
- Falso

1.2. Ligadura dinámica.

La conexión que tiene lugar durante una llamada a un método suele ser llamada **ligadura**, **vinculación** o **enlace** (en inglés **binding**). Si esta **vinculación** se lleva a cabo durante el proceso de compilación, se le suele llamar **ligadura estática** (también conocido como **vinculación temprana**). En los lenguajes tradicionales, no orientados a objetos, ésta es la única forma de poder resolver la **ligadura** (en **tiempo de compilación**). Sin embargo, en los **lenguajes orientados a objetos** existe otra posibilidad: la **ligadura dinámica** (también conocida como **vinculación tardía, enlace tardío** o **late binding**).

La **ligadura dinámica** hace posible que sea el **tipo de objeto** instanciado (obtenido mediante el **constructor** finalmente utilizado para crear el objeto) y no el **tipo de la referencia** (el tipo indicado en la declaración de la variable que apuntará al objeto) lo que determine qué versión del método va a ser invocada. El **tipo de objeto** al que apunta la variable de tipo referencia sólo podrá ser conocido durante la **ejecución** del programa y por eso el **polimorfismo** necesita la **ligadura dinámica**.

En el ejemplo anterior de la clase **X** y sus **subclases A** y **B**, la llamada al método **m** sólo puede resolverse mediante ligadura dinámica, pues es imposible saber en tiempo de compilación si el método **m** que debe ser invocado será el definido en la subclase **A** o el definido en la subclase B:

```
// Llamada al método m (sin saber si será el método m de A o de B).
```

```
obj.m ()  // Esta llamada será resuelta en tiempo de ejecución (ligadura dinámica)
```

Ejercicio resuelto

Imagínate una clase que represente a **instrumento musical** genérico (**Instrumento**) y dos subclases que representen tipos de instrumentos específicos (por ejemplo **Flauta** y **Piano**). Todas las clases tendrán un método **tocarNota**, que será específico para cada subclase.

Haz un pequeño programa de ejemplo en Java que utilice el **polimorfismo** (referencias a la **superclase** que se convierten en instancias específicas de **subclases**) y la **ligadura dinámica** (llamadas a un método que aún no están resueltas en **tiempo de compilación**) con estas clases que representan instrumentos musicales. Puedes implementar el método **tocarNota** mediante la escritura de un mensaje en pantalla.

Solución:

La clase **Instrumento** podría tener un único método (**tocarNota**):

```
public abstract class Instrumento {
```

```
    public void tocarNota (String nota) {
```

```
        System.out.printf ("Instrumento: tocar nota %s.\n", nota);
```

```
    }
```

```
}
```

En el caso de las clases **Piano** y **Flauta** puede ser similar, heredando de **Instrumento** y redefiniendo el método **tocarNota**:

```
public class Flauta extends Instrumento {
```

```
    @Override
```

```
    public void tocarNota (String nota) {
        System.out.printf ("Flauta: tocar nota %s.\n", nota);
    }
}


public class Piano extends Instrumento {
    @Override
    public void tocarNota (String nota) {
        System.out.printf ("Piano: tocar nota %s.\n", nota);
    }
}
```

A la hora de declarar una **referencia** a un objeto de tipo instrumento, utilizamos la **superclase** (`Instrumento`):

```
Instrumento instrumento1; // Ejemplo de objeto polimórfico (podrá ser Piano o Flauta)
```

Sin embargo, a la hora de instanciar el objeto, utilizamos el **constructor** de alguna de sus **subclases** (`Piano`, `Flauta`, etc.):

```
  if (<condición>) {
            // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Piano)
            instrumento1= new Piano ();
       }
       else if (<condición>) {
            // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Flauta)
            instrumento1=  new Flauta ();
       } else {
            ...
       }
```

Finalmente, a la hora de invocar el método `tocarNota`, no sabremos a qué versión (de qué **subclase**) de `tocarNota` se estará llamando, pues dependerá del tipo de objeto (**subclase**) que se haya instanciado. Se estará utilizando por tanto la **ligadura dinámica**:

```
   // Interpretamos una nota con el objeto instrumento1
```

```
  // No sabemos si se ejecutará el método tocarNota de Piano o de Flauta
```

```
 // (dependerá de la ejecución)
```

```
       instrumento1.tocarNota ("do");   // Ejemplo de ligadura dinámica (
```

Solución completa (ficheros):

Instrumento.java

```
package ejemplopolimorfismoinstrumentos;
/**
 *
 * Clase Instrumento
 */
public abstract class Instrumento {
   public void tocarNota (String nota) {
        System.out.printf ("Instrumento: tocar nota %s.\n", nota);
    }
}
```

Flauta.java

```
package ejemplopolimorfismoinstrumentos;
/**
 *
 * Clase Flauta
 */
public class Flauta extends Instrumento {
    @Override
    public void tocarNota (String nota) {
        System.out.printf ("Flauta: tocar nota %s.\n", nota);
    }
}
```

Piano.java

```
package ejemplopolimorfismoinstrumentos;
/**
 *
 * Clase Piano
 */
public class Piano extends Instrumento {
    @Override
    public void tocarNota (String nota) {
        System.out.printf ("Piano: tocar nota %s.\n", nota);
    }
}
```

EjemploPolimorfismoInstrumentos.java

```
/*
 * Ejemplo de polimorfismo y ligadura dinámica
 */
package ejemplopolimorfismoinstrumentos;

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class EjemploPolimorfismoInstrumentos {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
         String tipo= null;
         String nota= null;
```

```
 Instrumento instrumento1= null;
        // ¿Flauta o Piano?
        do {
            System.out.println("Elija instrumento: flauta(F) o piano(P): ");

            try {
                tipo= lecturaTeclado();
            }
            catch (Exception e) {
                System.err.println(e.getMessage());
            }
            if (tipo.equals("P") || tipo.equals("p")) tipo="piano";
            else if (tipo.equals("F") || tipo.equals("f")) tipo="flauta";
            else tipo="X";

        } while (tipo.equals("X"));

        // Nota musical
        System.out.println("Escriba nota musical: ");
        try {
            nota= lecturaTeclado();
        }
        catch (Exception e) {
            System.err.println(e.getMessage());
        }

        // Creación del objeto instrumento1 (desconocido en tiempo de compilación)
        // Sabemos que será subclase de Instrumento, pero no sabemos si será Flauta
o Piano
        // (dependerá de la ejecución)
        if (tipo.equals("piano")) {
```

```
            instrumento1= new Piano (); // Ejemplo de objeto polimórfico (puede ser
Piano o Flauta)
        }
        else if (tipo.equals("flauta")) {
            instrumento1=  new Flauta ();  // Ejemplo de objeto polimórfico (puede
ser Piano o Flauta)
        } else {
        }

        // Interpretamos una nota con el objeto instrumento1
        // No sabemos si se ejecutará el método tocarNota de Piano o de Faluta
        // (dependerá de la ejecución)
        instrumento1.tocarNota(nota);   // Ejemplo de ligadura dinámica (tiempo de
ejecución)
     }

    //---------------------------------------------------------------
    // MÉTODO lecturaTeclado: Captura de una cadena de teclado
    //---------------------------------------------------------------
    private static String lecturaTeclado () throws Exception {
        try {
            InputStreamReader inputStreamReader = new InputStreamReader(System.in);
            BufferedReader reader = new BufferedReader(inputStreamReader);
            String line = reader.readLine();
            return line;
        }
        catch (Exception e) {
            throw e;
        }
    }
}
```

1.3. Limitaciones de la ligadura dinámica.

Como has podido comprobar, el **polimorfismo** se basa en la utilización de **referencias** de un tipo más "amplio" (**superclases**) que los objetos a los que luego realmente van a apuntar (**subclases**). Ahora bien, existe una importante **restricción** en el uso de esta capacidad, pues el tipo de referencia limita cuáles son los métodos que se pueden utilizar y los atributos a los que se pueden acceder.

No se puede acceder a los **miembros específicos** de una **subclase** a través de una **referencia** a una **superclase**. Sólo se pueden utilizar los miembros declarados en la **superclase**, aunque la definición que finalmente se utilice en su ejecución sea la de la **subclase**.

Veamos un ejemplo: si dispones de una clase **A** que es subclase de **B** y declaras una variable como referencia un objeto de tipo **B**. Aunque más tarde esa variable haga referencia a un objeto de tipo **A** (**subclase**), los miembros a los que podrás acceder sin que el compilador produzca un error serán los miembros de **A** que hayan sido heredados de **B** (**superclase**). De este modo, se garantiza que los métodos que se intenten llamar van a existir cualquiera que sea la subclase de **B** a la que se apunte desde esa referencia.

En el ejemplo de las clases **Persona**, **Profesor** y **Alumno**, el **polimorfismo** nos permitiría declarar variables de tipo **Persona** y más tarde hacer con ellas referencia a objetos de tipo **Profesor** o **Alumno**, pero no deberíamos intentar acceder con esa variable a métodos que sean específicos de la clase **Profesor** o de la clase **Alumno**, tan solo a métodos que sabemos que van a existir seguro en ambos tipos de objetos (métodos de la **superclase** **Persona**).

Ejercicio resuelto

Haz un pequeño programa en Java en el que se declare una variable de tipo `Persona`, se pidan algunos datos sobre esa persona (**nombre**, **apellidos** y si es **alumno** o si es **profesor**), y se muestren nuevamente esos datos en pantalla, teniendo en cuenta que esa variable no puede ser instanciada como un objeto de tipo `Persona` (es una **clase abstracta**) y que tendrás que instanciarla como `Alumno` o como `Profesor`. Recuerda que para poder recuperar sus datos necesitarás hacer uso de la **ligadura dinámica** y que tan solo deberías acceder a métodos que sean de la **superclase**.

Solución:

Si tuviéramos diferentes variables referencia a objetos de las clases `Alumno` y `Profesor`tendrías algo así:

```
Alumno obj1;
```

```
Profesor obj2:
```

```
...
```

```
// Si se dan ciertas condiciones el objeto será de tipo Alumno y lo tendrás en obj1
```

```
System.out.printf ("Nombre: %s\n", obj1.getNombre());
```

```
// Si se dan otras condiciones el objeto será de tipo Profesor y lo tendrás en obj2
```

```
System.out.printf ("Nombre: %s\n", obj2.getNombre());
```

Pero si pudieras tratar de una manera más genérica la situación, podrías intentar algo así:

```
Persona obj;
```

```
// Si se dan ciertas condiciones el objeto será de tipo Alumno y por tanto lo instanciarás como tal
```

```
obj = new Alumno (<parámetros>);
```

```
// Si se otras condiciones el objeto será de tipo Profesor y por tanto lo instanciarás como tal
```

```
obj = new Profesor (<parámetros>);
```

De esta manera la variable `obj` obj podría contener una referencia a un objeto de la **superclase** `Persona` de **subclase** `Alumno` o bien de **subclase** `Profesor` (**polimorfismo**).

Esto significa que independientemente del tipo de **subclase** que sea (`Alumno` o `Profesor`), podrás invocar a métodos de la **superclase** `Persona` y durante la ejecución se resolverán como métodos de alguna de sus **subclases**:

```
//En tiempo de compilación no se sabrá de qué subclase de Persona será obj.
```

```
//Habrá que esperar la ejecución para que el entorno lo sepa e invoque al método adecuado.
```

```
System.out.printf ("Contenido del objeto usuario: %s\n", stringContenidoUsuario);
```

Por último recuerda que debes de proporcionar **constructores** a las **subclases** `Alumno` y `Profesor` que sean "compatibles" con algunos de los **constructores** de la **superclase** `Persona`, pues al llamar a un **constructor** de una **subclase**, su formato debe coincidir con el de algún **constructor** de la **superclase** (como debe suceder en general con cualquier método que sea invocado utilizando la **ligadura dinámica**).

Solución completa (ficheros):

Imprimible.java

```
/*
 * Interfaz Imprimible
 */
package ejemplopolimorfismopersona;

import java.util.Hashtable;
import java.util.ArrayList;

/**
```

```
 *
 * Interfaz Imprimible
 */
public interface Imprimible {
      String devolverContenidoString ();
      ArrayList devolverContenidoArrayList ();
      Hashtable devolverContenidoHashtable ();

}
```

Persona.java

```
/*
 * Clase Persona
 */
package ejemplopolimorfismopersona;

import java.util.GregorianCalendar;
import java.util.Hashtable;
import java.util.ArrayList;
import java.util.Enumeration;
import java.text.SimpleDateFormat;
/**
 * Clase Persona
 */
public abstract class Persona implements Imprimible {
        protected String nombre;
        protected String apellidos;
        protected GregorianCalendar fechaNacim;
```

```
        // Constructores
        // -------------

        // Constructor
        public Persona (String nombre, String apellidos, GregorianCalendar
fechaNacim) {
            this.nombre= nombre;
            this.apellidos= apellidos;
            this.fechaNacim= (GregorianCalendar) fechaNacim.clone();
        }

        // Métodos get
        // -------------

        // Método getNombre
        protected String getNombre (){
            return nombre;
        }

        // Método getApellidos
        protected String getApellidos (){
            return apellidos;
        }

        // Método getFechaNacim
        protected GregorianCalendar getFechaNacim (){
            return this.fechaNacim;
        }

        // Métodos set
        // -------------
```

```
// Método setNombre
protected void setNombre (String nombre){
    this.nombre= nombre;
}

// Método setApellidos
protected void setApellidos (String apellidos){
    this.apellidos= apellidos;
}

// Método setFechaNacim
protected void setFechaNacim (GregorianCalendar fechaNacim){
    this.fechaNacim= fechaNacim;
}

// Implementación de los métodos de la interfaz Imprimible
// -------------------------------------------------------

// Método devolverContenidoHashtable
public Hashtable devolverContenidoHashtable () {
    // Creamos la Hashtable que va a ser devuelta
    Hashtable contenido= new Hashtable ();

    // Añadimos los atributos específicos
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
    String stringFecha= formatoFecha.format(this.fechaNacim.getTime());
    contenido.put ("nombre", this.nombre);
    contenido.put ("apellidos", this.apellidos);
    contenido.put ("fechaNacim", stringFecha);
```

```
        // Devolvemos la Hashtable
        return contenido;
    }

    // Método devolverContenidoArrayList
    public ArrayList devolverContenidoArrayList () {
        ArrayList contenido= new ArrayList ();
        SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
        String stringFecha= formatoFecha.format(this.fechaNacim.getTime());

        contenido.add(this.nombre);
        contenido.add (this.apellidos);
        contenido.add(stringFecha);

        return contenido;
    }


    // Método devolverContenidoString
    public String devolverContenidoString () {
        String contenido=
Persona.HashtableToString(this.devolverContenidoHashtable());
        return contenido;
    }

    // Métodos estáticos (herramientas)
    // --------------------------------

    // Método HashtableToString
    protected static String HashtableToString (Hashtable tabla) {
        String contenido;
```

```
            String clave;
            Enumeration claves= tabla.keys();

            contenido= "{";
            if (claves.hasMoreElements()) {
                clave= claves.nextElement().toString();
                contenido= contenido + clave + "=" + tabla.get(clave).toString();
            }
            while (claves.hasMoreElements()) {
                    clave= claves.nextElement().toString();
                    contenido += ",";
                    contenido= contenido.concat (clave) ;
                    contenido= contenido.concat ("=" + tabla.get(clave));
            }
            contenido= contenido + "}";

            return contenido;
        }

}
```

Alumno.java

```
/*
 * Clase Alumno.
 */
package ejemplopolimorfismopersona;
import java.util.*;
import java.text.*;
```

```
/**
 *
 * Clase Alumno
 */
public class Alumno extends Persona  {
        protected String grupo;
        protected double notaMedia;



        // Constructores
        // -------------
        public Alumno (String nombre, String apellidos, GregorianCalendar
fechaNacim, String grupo, double notaMedia) {
            super (nombre, apellidos, fechaNacim);
            this.grupo= grupo;
            this.notaMedia= notaMedia;
        }

        public Alumno (String nombre, String apellidos, GregorianCalendar
fechaNacim) {
            super (nombre, apellidos, fechaNacim);
            // Valores por omisión para un alumno: Grupo "GEN" y nota media de 0.
            this.grupo= "GEN";
            this.notaMedia= 0;
        }

        // Métodos get
        // ------------

        // Método getGrupo
        public String getGrupo (){
```

```
        return grupo;
    }

    // Método getNotaMedia
    public double getNotaMedia (){
        return notaMedia;
    }

    // Métodos set
    // -----------

    // Método setGrupo
    public void setGrupo (String grupo){
        this.grupo= grupo;
    }

    // Método setNotaMedia
    public void setNotaMedia (double notaMedia){
        this.notaMedia= notaMedia;
    }

    // Redefinición de los métodos de la interfaz Imprimible
    // -----------------------------------------------------

    // Método devolverContenidoHashtable
    @Override
    public Hashtable devolverContenidoHashtable () {
        // Llamada al método de la superclase
        Hashtable contenido= super.devolverContenidoHashtable();
        // Añadimos los atributos específicos
        contenido.put ("grupo", this.grupo);
```

```
            contenido.put ("notaMedia", this.notaMedia);
            // Devolvemos la Hashtable rellena
            return contenido;
        }

        // Método devolverContenidoArray
        @Override
        public ArrayList devolverContenidoArrayList () {
            // Llamada al método de la superclase
            ArrayList contenido= super.devolverContenidoArrayList ();
            // Añadimos los atributos específicos
            contenido.add(this.grupo);
            contenido.add (this.notaMedia);
            // Devolvemos el ArrayList relleno
            return contenido;
        }

        // Método devolverContenidoString
        @Override
        public String devolverContenidoString () {
            // Aprovechamos el método estático para transformar una Hashtable en
String
            String contenido=
Persona.HashtableToString(this.devolverContenidoHashtable());
            // Devolvemos el String creado.
            return contenido;
        }

}
```

Profesor.java

```
/*
 * Clase Profesor
 */
package ejemplopolimorfismopersona;

/**

 */
import java.util.*;
import java.text.*;

/**
 *
 * Clase Profesor
 */
public class Profesor extends Persona {
      String especialidad;
      double salario;

        // Constructores
        // -------------

        public Profesor (String nombre, String apellidos, GregorianCalendar
fechaNacim, String especialidad, double salario) {
            super (nombre, apellidos, fechaNacim);
            this.especialidad= especialidad;
            this.salario= salario;
        }

        public Profesor (String nombre, String apellidos, GregorianCalendar
```

```
fechaNacim) {
            super (nombre, apellidos, fechaNacim);
            // Valores por omisión para un profesor: especialidad "GEN" y sueldo de
1000 euros.
            this.especialidad= "GEN";
            this.salario= 1000;
        }

        // Métodos get
        // -----------

        // Método getEspecialidad
        public String getEspecialidad (){
            return especialidad;
        }

        // Método getSalario
        public double getSalario (){
            return salario;
        }

        // Métodos set
        // -----------

        // Método setSalario
        public void setSalario (double salario){
            this.salario= salario;
        }

        // Método setESpecialidad
        public void setESpecialidad (String especialidad){
```

```
        this.especialidad= especialidad;
    }



    // Redefinición de los métodos de la interfaz Imprimible
    // -----------------------------------------------------

    // Método devolverContenidoHashtable
    @Override
    public Hashtable devolverContenidoHashtable () {
        // Llamada al método de la superclase
        Hashtable contenido= super.devolverContenidoHashtable();
        // Añadimos los atributos específicos
        contenido.put ("salario", this.salario);
        contenido.put ("especialidad", this.especialidad);
        // Devolvemos la Hashtable rellena
        return contenido;
    }

    // Método devolverContenidoArrayList
    @Override
    public ArrayList devolverContenidoArrayList () {
        // Llamada al método de la superclase
        ArrayList contenido= super.devolverContenidoArrayList ();
        // Añadimos los atributos específicos
        contenido.add(this.salario);
        contenido.add (this.especialidad);
        // Devolvemos el ArrayList relleno
        return contenido;
    }
```

```
        // Método devolverContenidoString
        @Override
        public String devolverContenidoString () {
            // Aprovechamos el método estático para transformar una Hashtable en
String
            String contenido=
Persona.HashtableToString(this.devolverContenidoHashtable());
            // Devolvemos el String creado.
            return contenido;
        }
}
```

EjemploPolimorfismo.java

```
/*
 * Ejemplo de utilización del polimorfismo y la ligadura dinámica.
 */
package ejemplopolimorfismopersona;

import java.util.GregorianCalendar;
import java.util.Date;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.text.SimpleDateFormat;
import java.text.ParseException;
/**
 *
 * Ejemplo de utilización del polimorfismo y la ligadura dinámica.
 */
public class EjemploPolimorfismo {
```

```
    /**
     * Clase principal
     */
    public static void main(String[] args) {
        String stringContenidoUsuario;

        String nombre= null, apellidos= null, tipo= null;
        Persona usuario= null;
        GregorianCalendar fecha= null;

        // PRESENTACIÓN
        // ------------
        System.out.printf ("PRUEBA DE USO DEl POLIMORFISMO Y LA LIGADURA DINÁMICA.
\n");
        System.out.printf
("-----------------------------------------------------\n\n");

        // ENTRADA DE DATOS
        // ----------------
        // Nombre
        System.out.print("Nombre del usuario: ");
        try {
            nombre= lecturaTeclado();
        }
        catch (Exception e) {
            System.err.println(e.getMessage());
        }

        // Apellidos
        System.out.print("Apellidos del usuario: ");
```

```
            try {
                apellidos= lecturaTeclado();
            }
            catch (Exception e) {
                System.err.println(e.getMessage());
            }

            // Fecha de nacimiento
            boolean fechaValida= true;
            do {
                String stringFecha= null;
                SimpleDateFormat formatoFecha= null;
                Date dateFecha= null;

                System.out.print("Fecha de nacimiento del usuario (formato DD/MM/AAAA):
    ");
                try {
                    stringFecha= lecturaTeclado();
                }
                catch (Exception e) {
                    System.err.println(e.getMessage());
                }

                // Conversión del texto en fecha
                formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
                try {
                    dateFecha= formatoFecha.parse(stringFecha);
                } catch (ParseException e)  {
                    fechaValida= false;
                }
                fecha= new GregorianCalendar ();
```

```
            fecha.setTime(dateFecha);

        } while (!fechaValida);

        // ¿Alumno o Profesor?
        do {
            System.out.println("¿Es alumno(A) o profesor(P)?");
            try {
                tipo= lecturaTeclado();
            }
            catch (Exception e) {
                System.err.println(e.getMessage());
            }
            if (tipo.equals("P") || tipo.equals("p")) tipo="profesor";
            else if (tipo.equals("A") || tipo.equals("a")) tipo="alumno";
            else tipo="X";

        } while (tipo.equals("X"));

        // Creación del objeto usuario (desconocido en tiempo de compilación)
        // Sabemos que será subclase de Persona, pero no sabemos si será Alumno o
Profesor
        // (dependerá de la ejecución)
        if (tipo.equals("profesor")) {
            usuario= new Profesor (nombre, apellidos, fecha);
        }
        else if (tipo.equals("alumno")) {
            usuario=  new Alumno (nombre, apellidos, fecha);
        } else {

        }
```

```
        // Obtención del contenido del objeto usuario a través del método devolverContenidoString.
        // El método que se va a ajecutar aún no se sabe cuál es (ligadura dinámica), pues
        // este objeto usuario no sabemos si será Alumno o Profesor. Tan solo sabemos que será de la
        // superclase Persona. En tiempo de ejecución se sabrá de qué tipo de subclase se trata y será
        // también en ese momento cuando el entorno de ejecución pueda resolver qué método se ejecuta
        // (el de método devolverContenidoString de la clase Alumno o el de la clase Profesor)

        stringContenidoUsuario= usuario.devolverContenidoString();

        // Impresión en pantalla del contenido del objeto usuario a través de la estructura obtenida
        System.out.printf ("Contenido del objeto usuario: %s\n", stringContenidoUsuario);

    }


    //---------------------------------------------------------------
    // MÉTODO lecturaTeclado: Captura de una cadena de teclado
    //---------------------------------------------------------------
    private static String lecturaTeclado () throws Exception {
        try {
            InputStreamReader inputStreamReader = new InputStreamReader(System.in);
            BufferedReader reader = new BufferedReader(inputStreamReader);
            String line = reader.readLine();
            return line;
```

```
                }
                catch (Exception e) {
                    throw e;

                }

            }
        }
```

2. Interfaces y polimorfismo.

Es posible también llevar a cabo el **polimorfismo** mediante el uso de **interfaces**. Un objeto puede tener una referencia cuyo tipo sea una **interfaz**, pero para que el compilador te lo permita, la clase cuyo **constructor** se utilice para crear el objeto deberá implementar esa **interfaz** (bien por si misma o bien porque la implemente alguna **superclase**). Un objeto cuya referencia sea de tipo **interfaz** sólo puede utilizar aquellos métodos definidos en la **interfaz**, es decir, que no podrán utilizarse los atributos y métodos específicos de su clase, tan solo los de la **interfaz**.

Las referencias de tipo **interfaz** permiten unificar de una manera bastante estricta la forma de utilizarse de objetos que pertenezcan a clases muy diferentes (pero que todas ellas implementan la misma **interfaz**). De este modo podrías hacer referencia a diferentes objetos que no tienen ninguna relación jerárquica entre sí utilizando la misma variable (referencia a la **interfaz**). Lo único que los distintos objetos tendrían en común es que implementan la misma **interfaz**. En este caso sólo podrás llamar a los métodos de la **interfaz** y no a los específicos de las clases.

Por ejemplo, si tenías una variable de tipo referencia a la interfaz **Arrancable**, podrías instanciar objetos de tipo **Coche** o **Motosierra** y asignarlos a esa referencia (teniendo en cuenta que ambas clases no tienen una relación de herencia). Sin embargo, tan solo podrás usar en ambos casos los métodos y los atributos de la interfaz **Arrancable** (por ejemplo **arrancar**) y no los de **Coche** o los de **Motosierra** (sólo los genéricos, nunca los específicos).

En el caso de las clases `Persona`, `Alumno` y `Profesor`, podrías declarar, por ejemplo, variables del tipo `Imprimible`:

```
Imprimible obj;                 // Imprimible es una interfaz y no una clase
```

Con este tipo de referencia podrías luego apuntar a objetos tanto de tipo **Profesor** como de tipo **Alumno**, pues ambos implementan la interfaz **Imprimible**:

```
// En algunas circunstancias podría suceder esto:
```

```
obj= new  Alumno (nombre, apellidos, fecha, grupo, nota); // Polimorfismo con interfaces
```

```
...
```

```
// En otras circunstancias podría suceder esto:
```

```
obj= new  Profesor (nombre, apellidos, fecha, especialidad, salario); // Polimorfismo con interfaces
```

```
...
```

Y más adelante hacer uso de la **ligadura dinámica**:

```
// Llamadas sólo a métodos de la interfaz
```

```
String contenido;
```

```
contenido= obj.devolverContenidoString();  // Ligadura dinámica con interfaces
```

Autoevaluación

El polimorfismo puede hacerse con referencias de superclases abstractas, superclases no abstractas o con interfaces. ¿Verdadero o Falso?

- Verdadero
- Falso

3. Conversión de objetos.

Como ya has visto, en principio no se puede acceder a los **miembros específicos** de una **subclase** a través de una **referencia** a una **superclase**. Si deseas tener acceso a todos los métodos y atributos específicos del objeto **subclase** tendrás que realizar una **conversión explícita** (**casting**) que convierta la referencia más general (**superclase**) en la del tipo específico del objeto (**subclase**).

Para que puedas realizar conversiones entre distintas clases es obligatorio que exista una relación de **herencia** entre ellas (una debe ser clase derivada de la otra). Se realizará una **conversión implícita o automática** de **subclase** a **superclase** siempre que sea necesario, pues un objeto de tipo **subclase** siempre contendrá toda la información necesaria para ser considerado un objeto de la **superclase**.

Ahora bien, la conversión en sentido contrario (de **superclase** a **subclase**) debe hacerse de forma **explícita** y según el caso podría dar lugar a errores por falta de información (atributos) o de métodos. En tales casos se produce una **excepción** de tipo `ClassCastException`.

Por ejemplo, imagina que tienes una clase **A** y una clase **B**, **subclase** de **A**:

```
class ClaseA {
```

```
    public int atrib1;
```

```
}
```

```

```

```
class ClaseB extends ClaseA {
```

```
    public int atrib2;
```

```
}
```

A continuación declaras una variable referencia a la clase **A** (**superclase**) pero sin embargo le asignas una referencia a un objeto de la clase **B** (**subclase**) haciendo uso del **polimorfismo**:

```
A obj;  // Referencia a objetos de la clase A
```

```
obj= new B (); // Referencia a objetos clase A, pero apunta realmente a objeto clase B (polimorfismo)
```

El objeto que acabas de crear como **instancia de la clase B** (**subclase** de **A**) contiene más información que la que la referencia **obj** te permite en principio acceder sin que el compilador genere un error (pues es de clase **A**). En concreto los objetos de la clase **B** disponen de **atrib1** y **atrib2**, mientras que los objetos de la clase **A** sólo de **atrib1**. Para acceder a esa información adicional de la clase especializada (**atrib2**) tendrás que realizar una **conversión explícita** (**casting**):

```
// Casting del tipo A al tipo B (funcionará bien porque el objeto es realmente del tipo B)
```

```
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib2);
```

Sin embargo si se hubiera tratado de una **instancia** **de la clase** **A** y hubieras intentado acceder al miembro **atrib2**, se habría producido una **excepción** de tipo `ClassCastException`:

```
A obj;  // Referencia a objetos de la clase A
```

```
obj= new A ();  // Referencia a objetos de la clase A, y apunta realmente a un objeto de la clase A
```

```

```

```
// Casting del tipo A al tipo B (puede dar problemas porque el objeto es realmente del tipo A):
```

```

```

```
// Funciona (la clase A tiene atrib1)
```

```
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib1);
```

```

```

```
// ¡Error en ejecución! (la clase A no tiene atrib2). Producirá una ClassCastException.
```

```
System.out.printf ("obj.atrib2=%d\n", ((B) obj).atrib2);
```