

MP0485
Programación
UF6. Interrelación entre clases

6.3. Clases abstractas, interfaces y polimorfismo

Índice

☰	Objetivos	3
☰	Raíz etimológica	4
☰	Concepto de polimorfismo aplicado a la POO	5
☰	Polimorfismo y relación de herencia	6
☰	Las clases abstractas	7
☰	Conversión o cash entre clases	10
☰	Las interfaces	13
☰	Las interfaces y la herencia múltiple	16
☰	Polimorfismo en tiempo de compilación (sobrecarga)	17
☰	Ligadura dinámica: referencias polimórficas	19
☰	Clases genéricas	22
☰	Comprobación dinámica de tipos	24
☰	Introspección	25
☰	Resumen	27

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

- 1 Crear clases abstractas e interfaces.
- 2 Comprender el concepto de polimorfismo.
- 3 Desarrollar programas Java empleando todas las técnicas de polimorfismo posibles: sobrecarga, ligadura dinámica y clases genéricas.
- 4 Comprobar la clase a la que pertenece un objeto en tiempo de ejecución.

¡Ánimo y adelante!

Raíz etimológica

En cuanto a la raíz etimológica de la palabra polimorfismo, es de origen griego y significa múltiples formas.

Sus componentes léxicos son:

1

Polys: muchos.

2

Morfo: formas.

3

Sufijo -ismo: actividad o sistema.

En cuanto al polimorfismo como propiedad se aplica a todo ser vivo u objeto capaz de adoptar múltiples formas o capaz de pasar por numerosos estados.

La comprensión del concepto general de polimorfismo te ayudará a asimilar mejor su aplicación en el mundo de la programación.

Concepto de polimorfismo aplicado a la POO

En el contexto de la programación orientada a objetos el polimorfismo se refiere a la capacidad de un objeto para adoptar distintos comportamientos. Se puede lograr de varias formas.

Tipos de polimorfismo

Polimorfismo en tiempo de compilación (sobrecarga)

Capacidad de un método para adaptar distintos comportamientos en función de los parámetros recibidos. El mismo método se comporta de distintas formas.

Polimorfismo en tiempo de ejecución (ligadura dinámica)

Capacidad de una referencia a un objeto para apuntar a distintos tipos de objetos. Una referencia a un objeto de tipo *Figura* puede apuntar a objetos de tipo *Triangulo*, *Rectangulo* o *Circulo*.

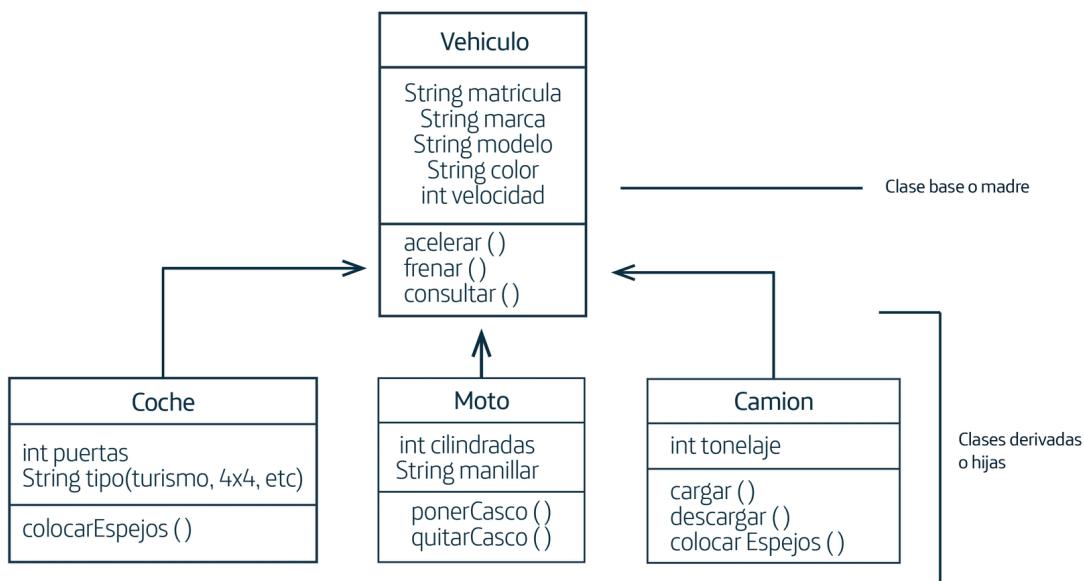
Clases genéricas

Una propiedad de una clase genérica puede variar de tipo según las necesidades. Por ejemplo: una clase llamada *Sumador* podría recibir dos argumentos que podrían ser números enteros (en cuyo caso halla la suma), cadenas de caracteres (en cuyo caso concatena ambas cadenas), dos arrays numéricos (en cuyo caso genera un nuevo array que suma los elementos que ocupan la misma posición de cada array).

Polimorfismo y relación de herencia

El polimorfismo se logra principalmente a través de relaciones de herencia.

Observa la siguiente imagen; una referencia a un objeto de tipo *Vehiculo* puede adoptar forma de objeto de tipo *Coche*, *Moto* o *Camion*.



El hecho de poder tratar un *Coche*, una *Moto* y un *Camion* de la misma forma (como un *Vehiculo*) favorece enormemente la reutilización de código. Lo descubrirás con los ejemplos que vayamos desarrollando.

Las clases abstractas

Una clase abstracta se diseña específicamente para que sirva de base a otras clases derivadas que la hereden, pero no sirve para crear objetos a partir de ella.

Están pensadas para crear otras clases derivadas que implementen sus métodos abstractos.

En nuestro ejemplo de las figuras de la unidad anterior podríamos haber implementado la clase *Figura* como abstracta, ya que se prevé que no será utilizada directamente para crear objetos, es decir, raramente escribiremos “new *Figura*(...)”, pero sí “new *Triangulo*(...)” o “new *Rectangulo*(...)”.

Ahora vamos a partir de otro ejemplo de abstracción; la clase *Animal* con sus derivadas *Pulga* y *Tiranosauro*.

Crea un nuevo proyecto y comienza por crear la clase abstracta *Animal*.

```
/*
Esta es una clase abstracta.
Sólo puede ser usada para crear clases derivadas
*/
public abstract class Animal {
    // Variable de instancia de clase.
    String nombre;

    // Constructor.
    public Animal(String n) {
        nombre = n;
    }

    // Métodos abstractos, deben ser sobre escritos en una clase derivada.
    public abstract String morder(Animal ani);
    public abstract String mover();

    // Método no abstracto, podrá ser o no sobre escrito en la clase derivada.
    @Override
    public String toString() {
        return "Saludos desde Animal";
    }
}
```

Reglas de diseño de las clases abstractas:

- 1 Una clase abstracta se define con el modificador *abstract* (*abstract class ...*).
 - 2 No es posible crear objetos a partir de una clase abstracta (*new Animal(...)*).
 - 3 Las clases abstractas tienen como objetivo servir de base a otras clases derivadas.
 - 4 Una clase abstracta debe tener algún método abstracto.
 - 5 Los métodos abstractos no tienen implementación, solo se indica la cabecera del método. Estos métodos deberán ser implementados por las clases derivadas.
 - 6 Las clases abstractas también pueden tener métodos no abstractos (ya implementados). Pensando en el ejemplo de las figuras, la clase *Figura* podría implementar un método llamado *modificarCoordenadas()*, pero un método como *calculaArea()*, por lógica, debería ser abstracto.
 - 7 Una clase abstracta puede heredar de otra clase abstracta.
-

Ahora crea las clases derivadas de la clase abstracta *Animal* (*Pulga* y *Tiranosauro*):

```
/*
Esta es una clase derivada que hereda una clase
abstracta. Será obligatorio implementar los métodos
abstractos de la clase base.
*/
public class Pulga extends Animal {
    Pulga() {
        super ("Pulga");
    }

    @Override
    public String morder(Animal ani) {
        return "Pulga muerde "+ani.nombre;
    }

    @Override
    public String mover() {
        return "Pulga se mueve";
    }

    @Override
    public String toString() {
        return "Saludos de la pulga";
    }
}
```

```
public class Tiranosaurio extends Animal {  
    public Tiranosaurio() {  
        super ("Tiranosaurio");  
    }  
  
    @Override  
    public String morder(Animal ani) {  
        return "Tiranosaurio muerde "+ani.nombre;  
    }  
  
    @Override  
    public String mover() {  
        return "Tiranosaurio se mueve";  
    }  
  
    @Override  
    public String toString() {  
        return "Saludos del tiranosaurio";  
    }  
  
    // Esté método es exclusivo del tiranosaurio  
    public String pisar(Animal ani) {  
        return "Tiranosaurio pisa a "+ani.nombre;  
    }  
}
```



Recuerda: la anotación **@Override** indica que el método ha sido heredado y lo estamos sobrescribiendo.

Reglas para las clases que heredan una clase abstracta:

1

Las clases derivadas de una clase abstracta están obligadas a sobrescribir sus métodos abstractos.

2

En cuanto a los métodos no abstractos, podrán optar por sobrescribirlos o utilizarlos tal cual están originariamente.

Por último, puedes crear la clase *Principal* con método *main* para poner en práctica el uso de la estructura de clases.

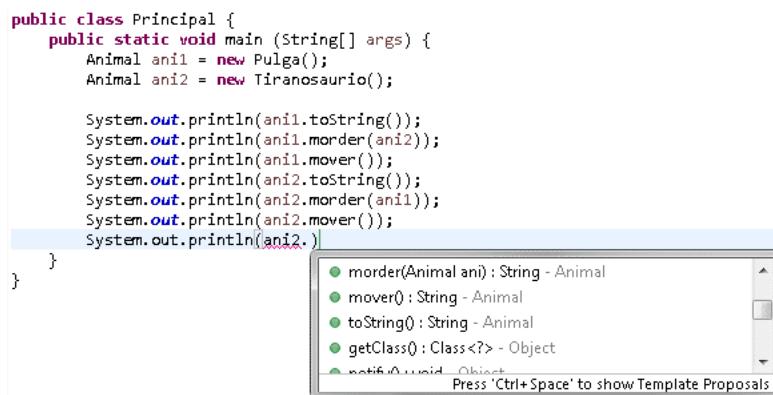
```
public class Principal {  
    public static void main (String[] args) {  
        Animal ani1 = new Pulga();  
        Animal ani2 = new Tiranosaurio();  
  
        System.out.println(ani1.toString());  
        System.out.println(ani1.morder(ani2));  
        System.out.println(ani1.mover());  
        System.out.println(ani2.toString());  
        System.out.println(ani2.morder(ani1));  
        System.out.println(ani2.mover());  
    }  
}
```

Observa que una referencia de tipo *Animal* puede ser indistintamente una *Pulga* o un *Tiranosaurio*. **Se está aplicado polimorfismo.**

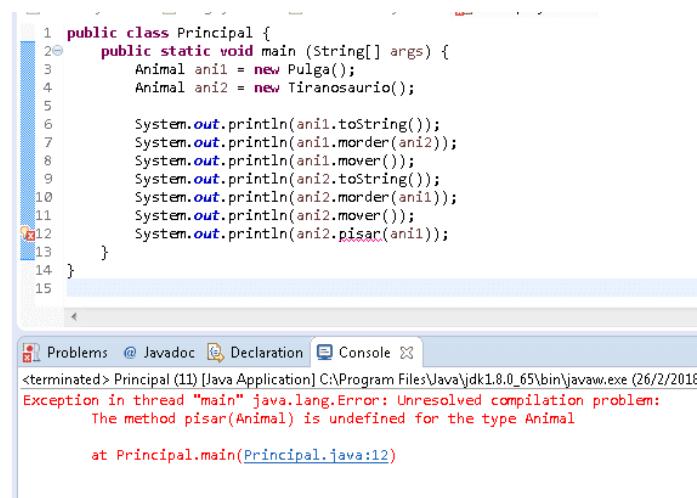
Conversión o cash entre clases

Un objeto *Pulga* y un objeto *Tiranosaurio* tienen ambos la capacidad de *mover* y *morder*, pero además, un *Tiranosaurio* puede *pisar*, y eso es algo que no estaba contemplado en la clase *Animal*. Vamos a ver con qué problema podemos encontrarnos al invocar al método *pisar*.

En primer lugar, ya te habrás dado cuenta de que cuando escribes el nombre de un objeto y un punto, Eclipse te ofrece una lista con los métodos y propiedades para hacerte el trabajo más fácil. Sin embargo, tal como observas en la imagen, no aparece el método *pisar* del objeto *ani2*. Aunque nosotros sabemos que *ani2* contiene una referencia a un *Tiranosaurio*, no deja de ser de tipo *Animal* y lo que está ofreciendo son las propiedades y métodos de un *Animal*, no de un *Tiranosaurio*.



Aunque escribas la sentencia completamente a mano para invocar al método *pisar*, seguirás teniendo un problema como el que ves en la imagen:



Nosotros sabemos que *ani2* es un *Tiranosauro*, pero el compilador no lo sabe, y tampoco lo sabe la máquina virtual de Java. Hay que hacer un *cast* o conversión de tipos de la siguiente manera:

(*Tiranosauro*) *ani2*

El código completo de la clase *Principal* quedará así:

```
public class Principal {  
    public static void main (String[] args) {  
        Animal ani1 = new Pulga();  
        Animal ani2 = new Tiranosauro();  
  
        System.out.println(ani1.toString());  
        System.out.println(ani1.morder(ani2));  
        System.out.println(ani1.mover());  
        System.out.println(ani2.toString());  
        System.out.println(ani2.morder(ani1));  
        System.out.println(ani2.mover());  
        System.out.println((Tiranosauro) ani2).pisar(ani1);  
    }  
}
```

Puedes construir un objeto *Tiranosauro* de dos formas distintas:

A partir de una referencia de tipo *Animal* o a partir de una referencia de tipo *Tiranosauro*.

```
Animal t1 = new Tiranosauro();  
Tiranosauro t2 = new Tiranosauro();
```

¿Cuándo usar una referencia de tipo más general?

Con una referencia más general nos referimos a una clase abstracta, una interfaz o una clase normal que tiene un grupo de clases derivadas. En nuestro caso, la clase *Animal*.

1

Cuando queremos aplicar polimorfismo, es decir, queremos que nuestra referencia a un objeto *Tiranosauro* pueda apuntar también a una *Pulga*.

```
Animal ani1 = new Tiranosauro();  
System.out.println(ani1);  
ani1 = new Pulga();  
System.out.println(ani1);
```

Ten en cuenta que un *Animal* puede ser un *Tiranosauro*, pero un *Tiranosauro* no puede ser ni un *Animal* ni una *Pulga*:

```
Tiranosauro t1 = new Animal();  
Tiranosauro t2 = new Pulga();
```

2

Cuando no necesitamos los métodos más especializados para cumplir el objetivo del programa. Por ejemplo: no necesitamos que el *Tiranosauro* tenga la capacidad de *pisar*. Aunque siempre podremos usar el método utilizando la conversión, como has aprendido en este apartado.

¿Cuándo usar una referencia de tipo más específico?

Con una referencia de tipo mas específico nos referimos a las clases derivadas que ocupan los niveles más bajos en la jerarquía. En nuestro ejemplo las clases *Pulga* y *Tiranosauro*.

1

Cuando no necesitamos aplicar polimorfismo. Quiero una referencia a un objeto *Tiranosauro* y no necesito que pueda apuntar a un objeto *Pulga*.

2

Quiero utilizar los métodos más específicos, en nuestro ejemplo *pisar*, sin necesidad de tener que hacer una conversión o *cast*.

Las interfaces

Una interfaz, aunque tiene una finalidad muy parecida a una clase abstracta y se utiliza de forma similar, tiene muchas diferencias.

Veamos las características de las interfaces:

- 1 Una interfaz especifica la forma de sus métodos, pero no da ningún detalle de su implementación; por lo tanto no se puede pensar en ella como la declaración de una clase.
- 2 Una interfaz no puede tener constructor.
- 3 Todos los métodos de una interfaz son abstractos, aunque no se incluya la palabra clave *abstract*. Por lo tanto, no se podrá incluir implementación en ninguno de los métodos.
- 4 Las variables miembro definidas en la interfaz son por defecto finales (constantes, su valor no se puede modificar) aunque se pueden redefinir en las clases derivadas.
- 5 Una interfaz se define con la palabra clave *interface*, ya que no es una clase.
- 6 Una interfaz puede heredar de otra interfaz.
- 7 Una clase abstracta puede implementar una interfaz.

El formato de creación de una interfaz es el siguiente:

```
modificadores interface nombreInterface {  
    // Declaración de variables y métodos  
}
```

Para ponerlo en práctica puedes hacer otro proyecto en Eclipse y crear la interfaz *Vehiculo*. Seleccionando el nombre del proyecto, tendrás que hacer clic derecho y elegir en el menú contextual “*New / Interface*”. Escribe *Vehiculo* como nombre de la interfaz.

```
public interface Vehiculo {  
    int VELOCIDAD_MAXIMA=120;  
  
    String frenar(int cuanto);  
    String acelerar(int cuanto);  
}
```

Una interfaz viene a ser como una declaración de intenciones, algo así como la norma para la creación de un conjunto de clases que deben cumplir unos criterios. Las clases *Moto*, *Coche*, *Camion*, *Avion*, etc., pueden implementar la interfaz *Vehiculo*, lo que significaría que están obligadas a implementar los métodos *acelerar()* y *frenar()* respetando la estructura que dicta la interfaz *Vehiculo*.

Ahora vamos a crear las clases *Coche* y *Moto* implementando la interfaz *Vehiculo*.

Para especificar que una clase implementa una interfaz se utiliza la palabra clave *implements*.

```
public class Coche implements Vehiculo {  
    int velocidad=0;  
  
    public String acelerar(int cuanto) {  
        String cadena="";  
        velocidad=velocidad+cuanto;  
        if (velocidad>VELOCIDAD_MAXIMA) cadena="Exceso de velocidad ";  
        cadena=cadena+"El coche a acelerado y va a "+velocidad+" km/hora";  
        return cadena;  
    }  
    public String frenar(int cuanto) {  
        velocidad=velocidad-cuanto;  
        return "El coche ha frenado y va a "+velocidad+" km/hora";  
    }  
}
```

```
public class Moto implements Vehiculo {  
    int velocidad=0;  
  
    public String acelerar(int cuanto) {  
        String cadena="";  
        velocidad=velocidad+cuanto;  
        if (velocidad>VELOCIDAD_MAXIMA)  
            cadena="Exceso de velocidad ";  
        cadena=cadena+"La moto a acelerado y va a "+velocidad+" km/hora";  
        return cadena;  
    }  
  
    public String frenar(int cuanto) {  
        velocidad=velocidad-cuanto;  
        return "La moto ha frenado y va a "+velocidad+" km/hora";  
    }  
}
```



Una clase que implementa una interfaz está obligada a implementar todos los métodos definidos en dicha interfaz.

Ahora podemos declarar un objeto de la clase *Vehiculo* y asignarle indistintamente una *Moto* o un *Coche*, de la misma forma que ocurría con las clases abstractas.

```
public class Principal {  
    public static void main (String[] args) {  
        Vehiculo v1 = new Moto();  
        Vehiculo v2 = new Coche();  
        System.out.println(v1.acelerar(100));  
        System.out.println(v1.frenar(25));  
        System.out.println(v2.acelerar(130));  
        System.out.println(v2.frenar(25));  
    }  
}
```

De nuevo estamos aplicando polimorfismo, ya que una referencia a un objeto *Vehiculo* puede apuntar indistintamente a un *Coche* o a una *Moto*, es decir, puede adaptar múltiples formas.

Las interfaces y la herencia múltiple

Java no admite herencia múltiple, pero sí la implementación de múltiples interfaces y la herencia de una clase base.

Es importante que tengas en cuenta que en el caso de las interfaces no se habla de herencia, sino de implementación. Lo que hacemos es implementar una interfaz que solo nos dicta lo que debe hacer la clase (los métodos que debe tener) pero no cómo hacerlo (la implementación).

Algo así sería correcto en Java:

```
public class Hipopotamo extends Animal implements Mamifero, Anfibio {  
}
```



Las interfaces definen normas o declaraciones de intenciones. Un hipopótamo, aparte de ser un animal, debe comportarse como un mamífero y un anfibio, características que se definen en las interfaces.

Polimorfismo en tiempo de compilación (sobrecarga)

Recuerda: la sobrecarga es una técnica de POO que permite realizar varias implementaciones para el mismo método.

Veamos un ejemplo para refrescar conceptos:

```
public class Venta {  
    private String producto;  
    private int cantidad;  
    private double precio;  
  
    public Venta(String producto, int cantidad, double precio) {  
        this.producto = producto;  
        this.cantidad = cantidad;  
        this.precio = precio;  
    }  
  
    public Venta(String producto, double precio) {  
        this.producto = producto;  
        this.cantidad = 1;  
        this.precio = precio;  
    }  
  
    public double calcularImporte() {  
        return cantidad * precio;  
    }  
  
    public double calcularImporte(float descuento) {  
        return cantidad * precio - (cantidad * precio * descuento);  
    }  
  
    @Override  
    public String toString() {  
        return "Venta [producto=" + producto + ", cantidad=" + cantidad + ", precio=" + precio + "]";  
    }  
}
```

Clase con sobrecarga en el constructor y sobrecarga en el método *calcularImporte()*.

La clase *Venta* dispone de dos implementaciones diferentes para el mismo método *calcularImporte*, esto significa que ejecutará una implementación u otra en función de que se introduzca un argumento o ninguno. También provee dos implementaciones para el método constructor, de modo que podemos construir un objeto *Venta* de dos formas distintas.

En la clase *Principal* podemos construir dos objetos *Venta* de dos formas distintas (indicando la cantidad o sin indicarla, en cuyo caso será 1). También podemos calcular el importe con o sin descuento.

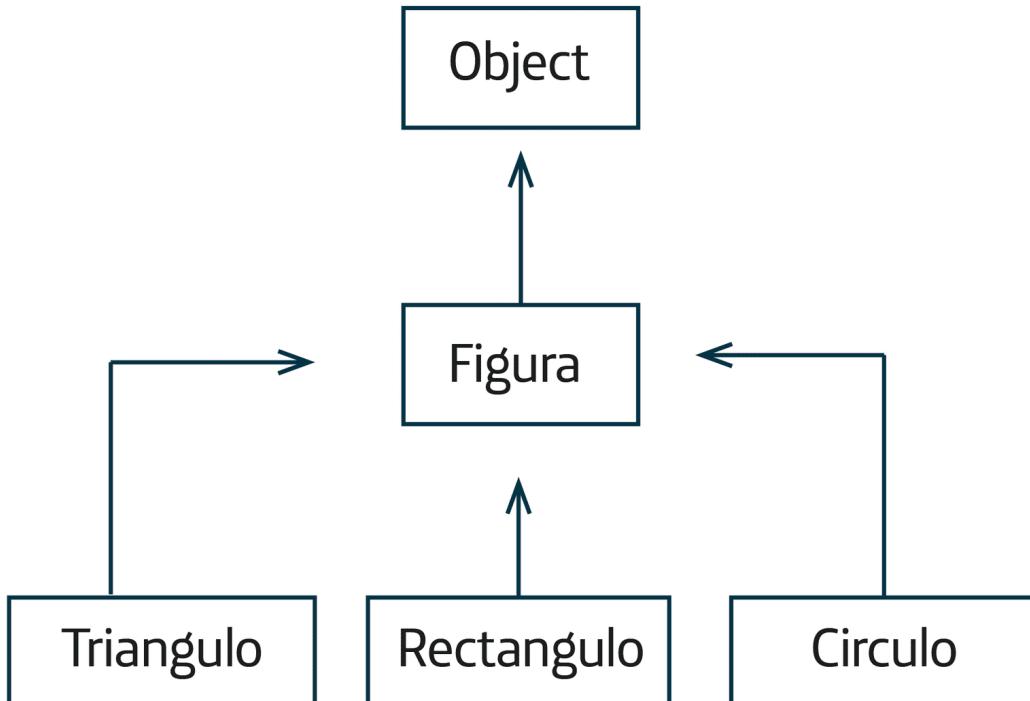
```
public class Principal {  
    public static void main(String[] args) {  
        Venta pro1 = new Venta("Impresora", 2, 100);  
        Venta pro2 = new Venta("Pen Drive", 10);  
        System.out.println(pro1.toString());  
        System.out.println("Importe Venta: " + pro1.calcularImporte());  
        System.out.println(); // Salto de línea  
        System.out.println(pro2.toString());  
        System.out.println("Importe Venta con descuento: " + pro2.calcularImporte(0.1f));  
    }  
}
```

Estamos ante una de las técnicas para lograr polimorfismo, ya que disponemos de métodos que actúan de múltiples formas en función del número y tipo de argumentos que le pasemos.

Ligadura dinámica: referencias polimórficas

La ligadura dinámica es la capacidad de una referencia para apuntar a diferentes tipos de objetos en tiempo de ejecución.

Volvamos a nuestro ejemplo de las *Figuras*:



Recuerda que **Figura** puede ser:

1

Una clase normal, siendo *Triangulo*, *Rectangulo* y *Círculo* sus clases derivadas.

2

Una clase abstracta, siendo *Triangulo*, *Rectangulo* y *Circulo* las clases derivadas que implementan sus métodos abstractos.

Recuerda que no es posible declarar objetos a partir de una clase abstracta, solo a partir de sus derivadas.

3

Una interfaz, siendo *Triangulo*, *Rectangulo* y *Circulo* las clases que implementan la interfaz.

Recuerda que una interfaz solo especifica la forma de sus métodos (cabecera), pero no da ningún detalle de su implementación. La implementación es trabajo para las clases que implementan dicha interfaz.

De cualquiera de las maneras, podríamos declarar una referencia a un objeto de tipo *Figura* que dinámicamente podrá cambiar de forma entre *Triangulo*, *Circulo* o *Rectangulo*. Lo puedes comprobar en el siguiente ejemplo:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        Figura fig = new Triangulo(3,3,5,7,5,7);  
        System.out.println(fig.consultar());  
        System.out.println("Area = " + fig.area());  
  
        fig = new Circulo(7,8,10,10,5);  
        System.out.println(fig.consultar());  
        System.out.println("Area = " + fig.area());  
  
        fig = new Rectangulo(7,8,7,10);  
        System.out.println(fig.consultar());  
        System.out.println("Area = " + fig.area());  
    }  
}
```

La variable *fig* es la referencia a nuestro objeto polimórfico. En concreto *fig* es una referencia que tiene la capacidad de apuntar a objetos de distinto tipo y, según el tipo de figura a la que apunta, se ejecutará el método *consultar()* y *area()* que le corresponda, y esto se decide en tiempo de ejecución.

El resultado del programa será este:

```
Triángulo: CoordenadaX=3, CoordenadaY=3, Ancho=5, Alto=7, Base= 5, Altura=7  
Area = 17.0  
Círculo: CoordenadaX=7, CoordenadaY=8, Ancho=10, Alto=10  
Area = 78.53981633974483  
Rectángulo: CoordenadaX=7, CoordenadaY=8, Ancho=7, Alto=10  
Area = 70.0
```

Una de las grandes ventajas del polimorfismo es la reutilización de código, lo que nos permite escribir el programa anterior de la siguiente manera:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        procesarFigura(Triangulo(3,3,5,7,5,7));  
  
        procesarFigura(new Circulo(7,8,10,10,5));  
  
        procesarFigura(new Rectangulo(7,8,7,10));  
    }  
  
    static void procesarFigura(Figura f){  
        System.out.println(fig.consultar());  
        System.out.println("Area = " + fig.area());  
    }  
}
```

Gracias al polimorfismo podemos crear un método común para todos los tipos de figura, dado que las instrucciones a ejecutar son las mismas en los tres casos.

Clases genéricas

Las clases genéricas tienen la posibilidad de declarar una o varias propiedades, cuyo tipo puede variar. El tipo de dicha propiedad será establecido en el momento de crear un objeto.

En el siguiente ejemplo *T* es el tipo genérico, que será reemplazado por un tipo real (*Double*, *Integer*, *String*, *Triangulo*,...).

```
public class Generica<T> {
    private T dato;

    public Generica(T dato) {
        this.dato = dato;
    }

    public String informa() {
        return "El objeto contiene un dato de tipo: " +
            this.dato.getClass().getName() +
            "\nValor = " + this.dato.toString();
    }
}
```

La propiedad *dato* es de tipo genérico. Se desconoce el tipo hasta el momento de crear el objeto.

this.dato.getClass().getName() nos suministra el nombre de la clase a la que pertenece el objeto actual (*Double*, *String*, *Integer*, etc.).

Ahora puedes comprobar el funcionamiento de la clase genérica creando la siguiente clase *Principal*:

```
public class Principal {
    public static void main(String[] args) {
        Generica<String> miObjeto1 = new Generica<String>("Hola mundo");
        System.out.println(miObjeto1.informa());

        Generica<Integer> miObjeto2 = new Generica<Integer>(35);
        System.out.println(miObjeto2.informa());

        Generica<Double> miObjeto3 = new Generica<Double>(45.30);
        System.out.println(miObjeto3.informa());
    }
}
```

Y el resultado obtenido será este:

El objeto contiene un dato de tipo: java.lang.String

Valor = Hola mundo

El objeto contiene un dato de tipo: java.lang.Integer

Valor = 35

El objeto contiene un dato de tipo: java.lang.Double

Valor = 45.3

Combinación dinámica de tipos

El operador *instanceof* permite comprobar dinámicamente (en tiempo de ejecución) la clase a la que pertenece un objeto con el fin de que el programa pueda responder en consecuencia.

Observa este ejemplo:

```
Figura fig = new Triangulo(3,3,5,7,5,7);
System.out.println(fig.consultar());
System.out.println("Area = " + fig.area());

if (fig instanceof Triangulo) {
    System.out.println("Es un triángulo");
}
```

Introspección

En programación orientada a objetos se denomina introspección a la capacidad de muchos lenguajes de programación para determinar en tiempo de ejecución la clase a la que pertenece un objeto. Incluso podríamos examinar el funcionamiento de dicha clase accediendo a la enumeración de sus propiedades y métodos.

El ejemplo más sencillo de introspección está en el uso del operador *instanceof* que acabamos de ver, pero también podemos utilizar el método *getClass()* para realizar tareas más sofisticadas. Recuerda que todas las clases heredan de la superclase *Object* y *getClass()* es uno de los métodos heredados de *Object*.

El método *getClass()* devuelve una representación conceptual de la clase a la que pertenece un objeto como un nuevo objeto de tipo *Class*.

En el siguiente ejemplo se comprobará la clase a la que pertenece el objeto apuntado por la referencia *fig* y además se realizará un listado con todos los métodos de la clase.

```
import java.lang.reflect.Method;

public class Principal {
    public static void main(String[] args) {

        Figura fig = new Triangulo(3,3,5,5,7);
        System.out.println(fig.consultar());
        System.out.println("Area = " + fig.area());

        Class clase = fig.getClass();
        System.out.println(clase.getName());
        Method[] metodos = clase.getMethods();
        for (int i = 0; i<metodos.length; i++) {
            System.out.println("Método: " + metodos[i].toString());
        }
    }
}
```

El resultado de ejecutar el programa será este:

Triángulo: CoordenadaX=3, CoordenadaY=3, Ancho=5, Alto=7, Base= 5, Altura=7

Area = 17.0

Triangulo

Método: public java.lang.String Triangulo.consultar()

Método: public double Triangulo.area()

Método: public int Figura.getCoordenadaX()

Método: public int Figura.getCoordenadaY()

Método: public int Figura.getAncho()

Método: public int Figura.getAlto()

Método: public final void java.lang.Object.wait() throws java.lang.InterruptedException

Método: public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

Método: public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

Método: public boolean java.lang.Object.equals(java.lang.Object)

Método: public java.lang.String java.lang.Object.toString()

Método: public native int java.lang.Object.hashCode()

Método: public final native java.lang.Class java.lang.Object.getClass()

Método: public final native void java.lang.Object.notify()

Método: public final native void java.lang.Object.notifyAll()

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- La palabra **polimorfismo** es de origen griego y significa **múltiples formas**.
- En el contexto de la programación orientada a objetos el polimorfismo se refiere a la capacidad de un objeto para adoptar distintos comportamientos y se puede lograr de varias formas:
 - Polimorfismo en tiempo de compilación (**sobrecarga**).
 - Polimorfismo en tiempo de ejecución (**ligadura dinámica**).
 - **Clases genéricas**.
- Para que una referencia tenga la capacidad de ser polimórfica debe ser declarada como:
 - Una clase normal pero que tenga sus clases derivadas.
 - Una clase abstracta.
 - Una interfaz.
- El operador `instanceof` permite comprobar dinámicamente (en tiempo de ejecución) la clase a la que pertenece un objeto.
- El método `getClass()` devuelve una representación conceptual de la clase a la que pertenece un objeto como un nuevo objeto de tipo `Class`.



PROEDUCA