Polimorfismo

Introducción

Una de las funciones más importantes de la herencia es la de propiciar el polimorfismo.
Polimorfismo quiere decir "muchas formas". La definición general de polimorfismo según la RAE[1]es:
Propiedad que tienen algunos cuerpos de poder cambiar de forma sin cambiar de naturaleza o composición.

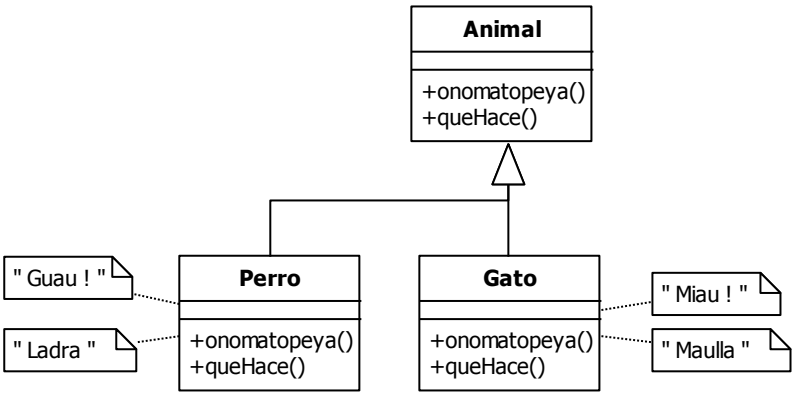
En programación, polimorfismo es una característica de un lenguaje de programación que permite a los valores de diferentes tipos de datos ser manejados usando una interfaz uniforme.

En el paradigma de objetos, en lenguajes de programación estructurados en clases, se refiere a la posibilidad de que objetos de clases diferentes respondan a mensajes con el mismo nombre, cada uno de ellos con su propio comportamiento.

El Polimorfismo, característica sobresaliente de la POO, permite reconocer y explotar las similitudes entre diferentes clases de objetos, basándose en el hecho de que una misma operación adopta diversas formas de implementación. Este concepto está fuertemente basado en el concepto de interfaz.

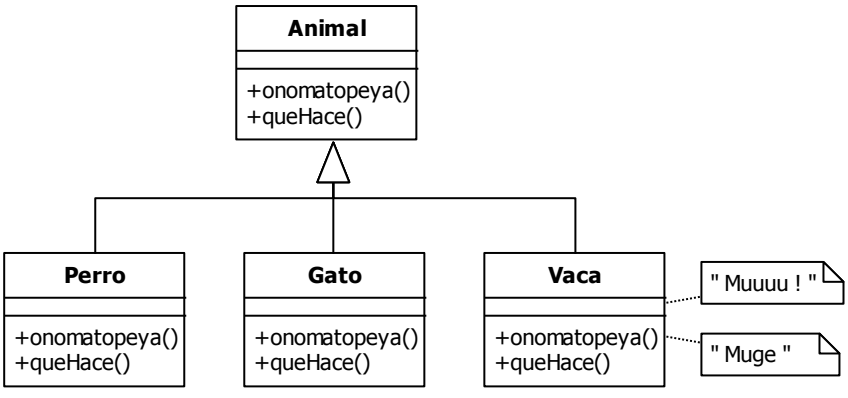
Cuando se reconoce que **tipos** diferentes de objetos pueden responder al mismo mensaje, se hace evidente la diferencia entre mensaje y método. Cuando un objeto (emisor) envía un mensaje, si el receptor entiende ese mensaje, ejecutará el método que él tiene asociado al mismo. Instancias de distintas clases pueden asociar, cada una, un método distinto a un mismo mensaje común a todas. El objeto emisor envía el mensaje sin preocuparse de la clase del receptor (bajo acoplamiento, "conoce" lo menos posible).

En el siguiente ejemplo, un objeto de la clase Perro y otro de la clase Gato responderán con el sonido esperado.



Una vez implementada esta jerarquía, una aplicación podría trabajar indistintamente con gatos o perros, solicitándoles el sonido, y cada animal respondería con el sonido adecuado. Las acciones desencadenadas dependerán de cuál es la clase del objeto receptor. Esto significa que el objeto emisor no necesita saber a quién le está enviando un mensaje. Solo debe saber que ese mensaje está en la interfaz pública.

El polimorfismo permite hacer programas más flexibles. Por ejemplo, con posterioridad se pueden añadir nuevas clases derivadas de *Animal* (Vaca, Pato), sin que afecte a la aplicación. Cada nueva clase sólo debe tener la misma interfaz (responder a los mismos mensajes), aunque las implementaciones particulares sean diferentes.



En definitiva, lo que permite el polimorfismo es retardar la decisión sobre el **tipo** del objeto hasta el momento en que vaya a ser utilizado el método. En este sentido, el polimorfismo está asociado al concepto de **vinculación tardía** o **late binding.** Se llama vinculación tardía al enlace de una petición con el código (mensaje con el método) en tiempo de ejecución, en contraposición al mecanismo de enlace estático, llevado a cabo en tiempo de compilación. Recibe también los nombres de binding dinámico o vinculación en tiempo de ejecución.

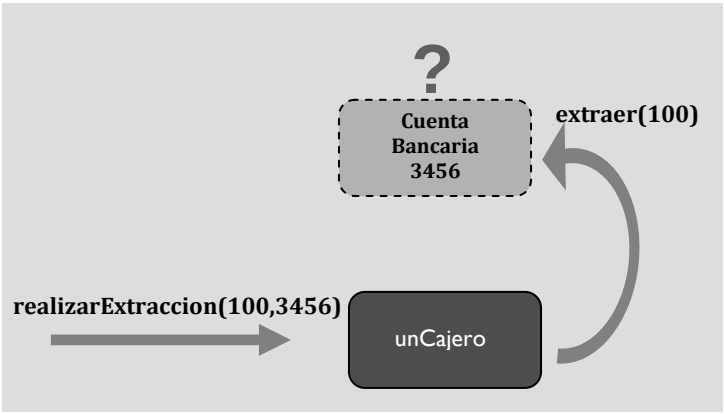
[1] Real Academia Española

Utilidad del Polimorfismo

Los objetos se definen por los servicios que pueden brindar, lo cual queda expresado por los mensajes a los que puede responder. Por lo tanto, si se modela la noción de una cuenta bancaria como un objeto al que se puede enviar el mensaje *extraer(unMonto)*, cualquier otro objeto al que se pueda enviar dicho mensaje podrá cumplir el rol de una cuenta bancaria.

En las cuentas bancarias se puede depositar y extraer un monto de dinero y además consultar su saldo. Un usuario del banco puede acceder a su cuenta por medio de un cajero automático, para realizar cualquiera de estas operaciones, sabiendo el número de la cuenta con la cual quiere operar.

Supongamos que se desea extraer $100 de la cuenta número 3456. Para ello se debe enviar el mensaje *realizarExtraccion(100, 3456)* al objeto cajero automático. Este objeto sabrá de alguna manera obtener la cuenta bancaria 3456, a la cual le enviará el mensaje *extraer(100)* para completar la extracción. Sin embargo, no sabe de qué tipo de cuenta se trata.



Asumiendo que las cajas de ahorro y las cuentas corrientes tienen distinto modo de realizar una extracción, la implementación podría ser:

```
public class Cajero{
   public static void main(String args[]){
         CuentaBancaria unaCuenta = unBanco.buscarCuenta(3456);
         if(unaCuenta.esCajaDeAhorro()){
               unaCuenta.extraerDeCajaDeAhorro(100);
         }else{
               unaCuenta.extraerDeCuentaCorriente(100);
              }
      }
}
```

Esto funcionaría correctamente. Sin embargo, qué ocurriría si fuera necesario agregar un nuevo tipo de cuenta, por ejemplo de nombre *SuperCuenta*. Se asume que la *SuperCuenta* tiene una manera particular de realizar las extracciones, como también ocurre con los otros dos tipos de cuenta. Se debería modificar el código de la siguiente manera:

```
if(unaCuenta.esCajaDeAhorro()){
      unaCuenta.extraerDeCajaDeAhorro(100);
}else
      if(unaCuenta.esCuentaCorriente(){
         unaCuenta.extraerDeCuentaCorriente(100);
      }else{
            unaCuenta.extraerDeSuperCuenta(100);
      }
```

Este tipo de implementación va en contra de la escalabilidad del sistema, dado que, en el caso de agregar un nuevo tipo de cuenta se debe modificar el código en todos aquellos sitios en los que aparezcan decisiones como la presentada en el ejemplo.

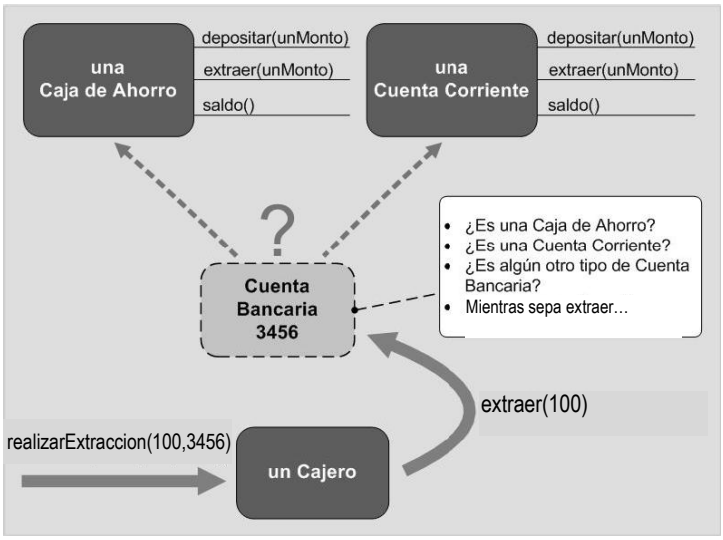
Alternativa de implementación

Al trabajar con objetos, la idea es que el objeto emisor sepa lo menos posible respecto del objeto al que le está enviando el mensaje (objetos independientes, bajo acoplamiento).

Fuertemente asociado a la idea de polimorfismo aparece el principio "Tell, don't ask". Este principio indica que un objeto emisor no necesita seleccionar qué mensaje enviar en base a las características del objeto receptor, sino que debe ser independiente de éste, y simplemente pedirle que lleve adelante la tarea indicada. De esta forma, los cambios impactarán en menor medida.

En el caso de las cuentas bancarias, lo único que interesa es que el objeto unaCuenta sea una cuenta bancaria de la cual se pueda extraer. Para esto, se pide que todo objeto que quiera ser tomado como una cuenta bancaria sepa responder al mensaje *extraer(unMonto)*. En otras palabras, debe tener esa interfaz, sin importar su implementación.

Los objetos involucrados y la situación planteada puede observarse en la siguiente figura:



Se observan dos objetos distintos (caja de ahorro y cuenta corriente) que entienden los mismos mensajes, es decir, son polimórficos. Además hay un objeto de tipo cajero automático, el cual sabe extraer un monto de dinero en cualquier cuenta, ya sea de ahorro o corriente.

En casos como este se aprecia la utilidad del polimorfismo, ya que el código para resolver esta situación debería reducirse a lo siguiente:

```
public class Cajero{
   public static void main(String args[]){
         CuentaBancaria unaCuenta = unBanco.buscarCuenta(3456);
         unaCuenta.extraer(100);
   }
}
```

De este modo, cualquiera sea el tipo de nuevas cuentas que se agreguen, no se desencadenarán modificaciones en el código.

Ventajas del Polimorfismo

El uso del polimorfismo brinda una serie de ventajas a la hora de construir programas.

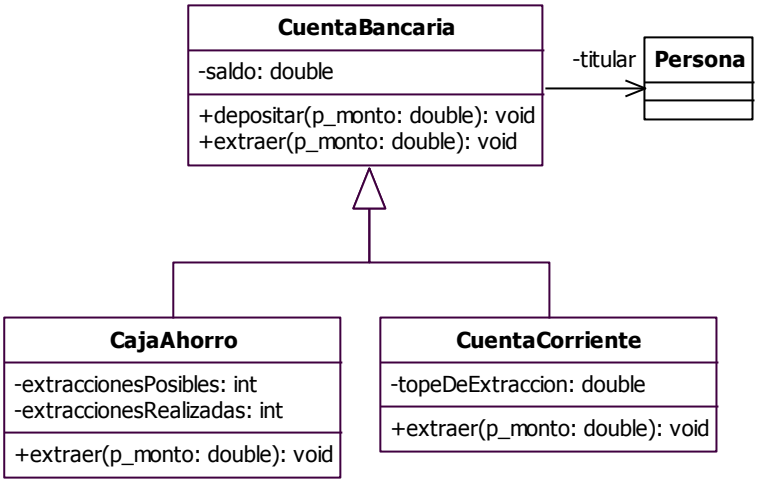
- **Código genérico**: en primer lugar permite producir código genérico, basándose en el principio "Tell, don't ask". Al utilizar esta técnica, el objeto emisor es independiente del tipo específico del objeto receptor, lo que permite agregar nuevos tipos de objetos que cumplan con el mismo protocolo sin impactar en el diseño del sistema.

- **Objetos desacoplados**: Al utilizar objetos polimórficos se dice que se trabaja con un bajo nivel de acoplamiento[2], dado que el objeto emisor sabe lo mínimo indispensable del objeto al que le está enviando el mensaje (no es necesario saber su tipo ni cómo implementa su comportamiento), lo cual brinda flexibilidad al programa y evita la propagación de cambios locales en el resto del sistema.
- **Objetos intercambiables**: de acuerdo al ejemplo mencionado, en cualquier parte del programa en que pueda aparecer un objeto "unaCajaDeAhorro" puede aparecer un objeto "unaCuentaCorriente".
- **Objetos reutilizables**: un objeto caja de ahorro bien definido, puede ser utilizado en un sistema de débitos automáticos, para el cual no había sido pensado inicialmente.
- **Programar por protocolo, no por implementación**: el protocolo es el conjunto de servicios que el objeto puede ofrecer. Es decir, qué es lo que sabe hacer, su "responsabilidad". Al utilizar los objetos en un programa, sólo debe tenerse en cuenta si el servicio que ofrece es el que se necesita para el programa, sin interesar cómo implementa esa funcionalidad.

Redefinición y Polimorfismo

La redefinición (override) es una técnica de la POO que permite darle especificidad al comportamiento de una clase, cuando éste no es exactamente igual al de la clase ancestro, pero **manteniendo la semántica**. Junto con el concepto de polimorfismo, brindan gran flexibilidad a la hora de desarrollar sistemas.

Por ejemplo, en el caso de las cuentas bancarias, el método *extraer()* para una caja de ahorro puede tener una cantidad máxima de extracciones en el mes, y no se puede extraer un monto mayor al saldo. En el caso de la cuenta corriente puede tener autorizado un descubierto. Cada subclase definirá diferentes métodos *extraer()*, que **redefinen** al de su ancestro. Esto implica que, cada vez que se invoque al método *extraer* para una instancia de *CajaAhorro* se hará referencia, no al heredado de *CuentaBancaria*, sino al redefinido en *CajaAhorro*.



Esto permite que la respuesta a un mismo mensaje dependa de la clase a la que pertenece el objeto.
Por ejemplo:

```
unaCajaAhorro.extraer(100);

unaCuentaCorriente.extraer(100);
```

Como principio general, se debe utilizar redefinición cuando en una subclase pueda realizarse una implementación más eficiente o más completa que la de la clase ancestro.

En consecuencia, la redefinición:

- Debe preservar la semántica de la definición del método en el ancestro. Es decir que la tarea en esencia debe ser la misma, aunque se haga de diferente forma. Ésta es la única forma de garantizar el cumplimiento de la relación "es un", o principio de sustitución. Nunca debe redefinirse un método de manera que sea semánticamente inconsistente con el originalmente heredado. Una subclase es un caso especial de su superclase y debería ser compatible con ella en todo aspecto.
- Será **obligatoria**, cuando el comportamiento de una clase descendiente sea diferente al de la clase ancestro. Por ejemplo, si hay un método que se llama *imprimirAtributos* que imprime todos los atributos de un objeto, éste deberá redefinirse en las clases derivadas que agreguen atributos si se desea que funcione correctamente.
- Será **optativa**, cuando por razones de eficiencia o claridad se desea modificar la implementación del método del ancestro. Por ejemplo, a través de un algoritmo más eficiente para la subclase (Perímetro Rectángulo/Cuadrado).

[2] Dependencia entre objetos

La redefinición se efectúa definiendo en la subclase un método con igual firma que en el ancestro. El método en la subclase sobrescribe y reemplaza al método de la superclase. También se dice que lo oculta.

Se debe tener en cuenta que la redefinición se introdujo para cambiar **comportamiento** de una clase descendiente respecto del ancestro, no para introducir cambios en la **estructura**. Por lo tanto, se pueden redefinir métodos, pero no atributos.

Los métodos privados no pueden ser redefinidos. Si se lo intenta, en realidad se estará definiendo un nuevo método. Al fin y al cabo, la redefinición sólo tiene sentido si el método es parte de la interfaz del ancestro, y los métodos privados no son parte de la interfaz.

Para evitar la sobrecarga y efectivamente lograr el ocultamiento del método de la clase ancestro, el método deberá **mantener** la misma cantidad y el mismo tipo de parámetros del método de la clase ancestro.

Sobrecarga y redefinición

La sobrecarga (overloading), es un concepto que en realidad no está directamente relacionado con POO. Existía previamente en lenguajes como Pascal y Ada.

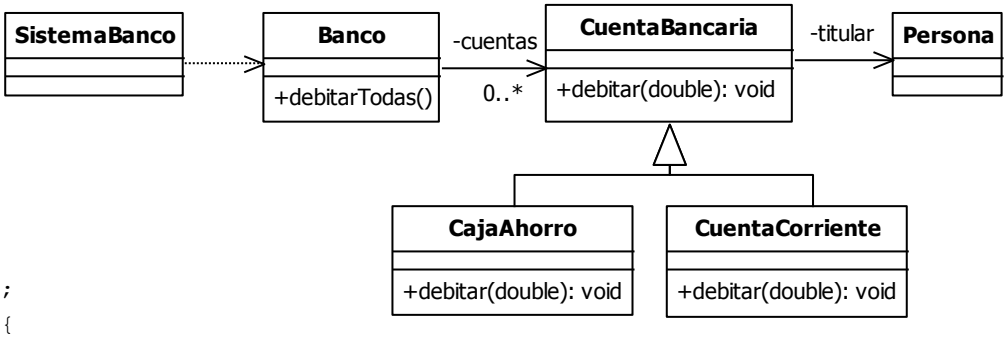
No obstante, hay algunas interacciones con el concepto de redefinición que se debe considerar. La relación más conflictiva de la sobrecarga y la redefinición se da cuando se redefine un método en una clase descendiente con una firma (*signatura*) distinta a la del método ancestro. Las consecuencias de esta acción dependen del lenguaje en que se realice la implementación. En Java, cuando se utilizan firmas diferentes en una clase ancestro y una descendiente, esto no oculta el método de la clase ancestro, sino que provoca una sobrecarga (un nuevo método), por lo cual los métodos que no se redefinen con la misma firma de la clase base, siguen estando disponibles para la clase descendiente.

Métodos virtuales

Un concepto fuertemente relacionado al de polimorfismo es el de ligadura dinámica (dinamic binding), lo cual significa que el enlace entre el receptor del mensaje, y el mensaje, se realiza en tiempo de ejecución. Un **método virtual** es aquel cuya implementación no es determinada hasta el momento de ejecución. En Java, todo método que no sea *final, static o prívate* es considerado método virtual, ya que puede ser sobreescrito.

Es decir que al momento de la compilación no se determina aún cuál implementación se ejecutará. Esto permite que cada objeto responda al mensaje del modo que sabe hacerlo, **durante la ejecución**.

Considere una jerarquía de clases en la cual se define el método *debitar* en *CuentaBancaria*, y que se redefine en las clases descendientes. Considere además un Banco con una colección de cuentas bancarias, de las cuales se desea debitar $20 a fin de mes, en concepto de mantenimiento:




```
import java.util.*;
public class Banco{
     ArrayList <CuentaBancaria > cuentas;
     Banco(ArrayList <CuentaBancaria> p_cuentas){
         this.setCuentas(p_cuentas);
     }
     ...
     public void debitarTodas(){
           for (int i = 0; i < this.getCuentas().size(); i++){
              (this.getCuentas().get(i)).debitar(20);
    }
  }
}
```

```
import java.util.*;
public class SistemaBanco{
    public static void main(String []args){
       ArrayList <CuentaBancaria> cuentas = new ArrayList <CuentaBancaria>();
       Persona persona1 = new Persona(32022113,"Juan","Perez",1978);
       Persona persona2 = new Persona(34367113,"Mario","López",1989);
       CajaDeAhorro caja2 = new CajaDeAhorro(2135, persona1);
       cuentas.add(caja2);
       CajaDeAhorro caja3 = new CajaDeAhorro(1027, persona2, 100.0);
       cuentas.add(caja3);
       CuentaCorriente ctacte1 = new CuentaCorriente();
       cuentas.add(ctacte1);
       CuentaCorriente ctacte2 = new CuentaCorriente(3567, persona1);
       cuentas.add(ctacte2);

       Banco unBanco = new Banco(cuentas);
       unBanco.debitarTodas();
   }
}
```

Al enviarse un mensaje, el intérprete debe realizar dos tareas:

1. Determinar cuál es la clase del objeto receptor (comprobación de tipos).
2. Buscar el método para responder al mensaje. Este proceso se lleva a cabo buscando en la clase del objeto receptor del mensaje. En el caso que dicha clase no tenga definido un método asociado al mensaje enviado, se repite el proceso continuando por su superclase.

Los métodos que son vinculados tardíamente se denominan **métodos virtuales**. Para estos métodos, el compilador no vincula el código invocante con el método invocado, sino que difiere esta vinculación hasta el tiempo de ejecución. Lo único que se hace en tiempo de compilación, si el lenguaje es fuertemente tipado, es chequear cantidades y tipos en argumentos y parámetros, y valores de retorno.

Clases y métodos abstractos

Métodos abstractos

Un método abstracto es aquel que no tiene implementación, y que por lo tanto nunca será ejecutado. A priori parece algo extraño, sin embargo, tiene su razón de ser.

Por ejemplo, el método *extraer* de la clase ***CuentaBancaria*** nunca será ejecutado, dado que no tiene comportamiento común a ambas cuentas, y está definido en cada clase descendiente. Podría pensarse en suprimirlo, ya que no aporta comportamiento, y dejarlo solamente en las clases hijas. Sin embargo, si se observa el ejemplo presentado en el párrafo anterior, el método *debitarTodas* envía el mensaje *debitar()* a una colección de tipo *CuentaBancaria*. Si esta clase no lo tuviera declarado, se presentaría un error en tiempo de compilación. La solución es **declarar** el método *debitar()* en *CuentaBancaria*, especificándolo como *método abstracto*, e implementar el comportamiento en cada subclase.

La utilidad de los métodos abstractos de una clase (que por ende será abstracta), es **establecer el protocolo** que debe implementar cualquier nueva clase que extienda de ésta. Es decir, conforma la especificación de una **interfaz** común a todas las subclases.

Resumiendo, los métodos abstractos:

- No tienen código fuente ejecutable (no están implementados): No se especifica comportamiento, ya que a nivel de la superclase no se conoce
- Son importantes para establecer el protocolo de un objeto
- Aseguran que todo objeto instancia de una subclase puede responder a ese mensaje
- Siempre deben ser redefinidos, y en algún nivel inferior de la jerarquía deben dejar de ser abstractos, ya que no se los puede invocar directamente.
- Son virtuales (vinculados tardíamente)

En Java, los métodos abstractos se especifican anteponiendo la palabra *abstract.*

```
public abstract void extraer(); // observar que no lleva {}
```

Clases abstractas

En ocasiones es necesario definir clases para las cuales no tiene sentido crear instancias. Un caso podría ser *CuentaBancaria*, dado que las clases del nivel inmediato inferior cubren todos los casos posibles. Es decir, una cuenta en particular será *CajaAhorro o CuentaCorriente*, pero nunca directamente una *CuentaBancaria*.

Estas clases se denominan clases abstractas, y tienen como única finalidad la declaración de atributos y métodos comunes que luego se utilizarán en las clases descendientes. Dicho formalmente, se construyen para generalizar estructura y comportamiento común de varias clases descendientes.

Las clases abstractas no pueden ser instanciadas, y esto se justifica porque no tienen todo el **comportamiento definido**. Es decir, puede ocurrir que al recibir un mensaje no sepa cómo responder.

El propósito o utilidad de las clases abstractas es establecer el protocolo que debe implementar cualquier nueva clase que desee extender esa superclase. Es decir, especifica la interfaz común a todas las subclases.

Al subclasificar una clase abstracta, se presentan dos posibilidades:

- Implementar los mensajes abstractos en la subclase. En este caso la subclase es una clase concreta y por ende puede ser instanciada.
- No implementar los mensajes abstractos en la subclase: la subclase también será abstracta.

En Java, si una clase contiene uno o más métodos abstractos, será abstracta. Pero las clases abstractas también pueden declararse sin métodos abstractos, simplemente anteponiéndoles la palabra *abstract*. Esto tiene sentido en el caso de que se declare una clase para generalizar estructura y comportamiento no abstracto. Se declaran del siguiente modo:

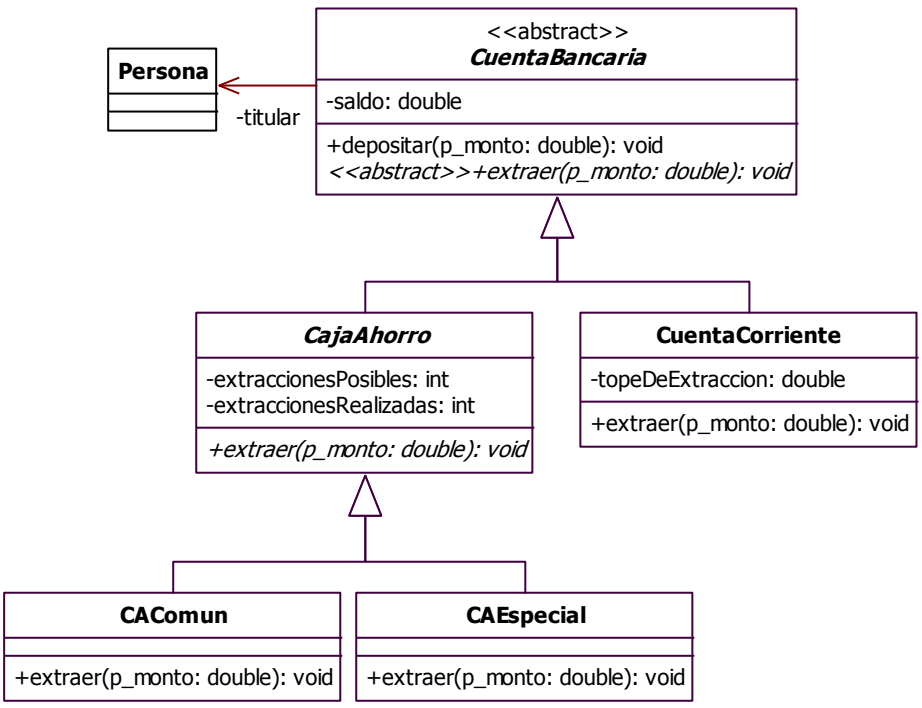
```
public abstract class CuentaBancaria{
   // declaraciones de atributos y métodos
}
```

Las interfaces son un caso especial de clase abstracta. Por eso, si una clase abstracta no necesita implementar métodos (es decir, si todos son abstractos) o atributos, sería mejor definirla como interfaz.

Clases y métodos abstractos en UML

En UML, una clase abstracta se representa en cursiva, y lo mismo ocurre con los métodos abstractos, como se muestra en la jerarquía de la figura con las clases *CuentaBancaria* y *CajaAhorro* y el método *extraer* en dichas clases. Para mayor claridad se puede utilizar el estereotipo <<abstract>> como se observa en *CuentaBancaria* y en su método *extraer*:

Nótese que como CajaAhorro no implementa el método *extraer*(), sigue siendo una clase abstracta.



Transformaciones e información de tipos en tiempo de ejecución
Transformación de tipos

En la mayoría de los lenguajes de programación tradicionales existe alguna forma de transformar un valor de un tipo en otro. Pero si bien era algo que podía hacerse en todos los lenguajes, pocas veces era realmente necesario utilizarlo. Por ejemplo, el lenguaje de programación **C** permite la transformación de tipos, y si se realiza en forma indiscriminada, haciendo abuso de esta capacidad del lenguaje, el software resultante puede ser difícil de probar y de mantener. Esta característica se modificó en C++, que es más restrictivo que C, no permitiendo convertir arbitrariamente tipos diferentes.

Por ejemplo, en C, la línea:

c = (char) x;

transforma el contenido de la variable *x* en un *char* y luego lo asigna a c. Esto podría ser factible, por ejemplo, si *x* es un entero y c un *char*.

Esta característica de la transformación de tipos se suele denominar **moldeo,** por el uso del verbo inglés "to cast" que se aplica en estos casos, en el sentido de tomar un papel, una actuación. De allí el término "castear".

Sin embargo, el concepto de transformación de tipos y, en forma más general, de uso de la información sobre tipos **en tiempo de ejecución** tiene un valor importante en el contexto de POO cuando se usa en conjunto con el polimorfismo y la vinculación tardía en general.

También es posible moldear clases, y en Java se hace como en C.

Ejemplo: supongamos una colección de cuentas bancarias *cuentas* que almacena cajas de ahorro. Es posible realizar un moldeo de una cuenta bancaria a caja de ahorro del siguiente modo:

```
CajaAhorro ca = (CajaAhorro)cuentas.get(0);
```

La diferencia fundamental con C es que Java hace una verificación previa que asegure que el objeto almacenado en `cuentas.get(0)` sea realmente una caja de ahorro. Si no fuera así, presenta error **en tiempo de ejecución**.

Transformación de tipos implícita y explícita

1. Implícita (automática)
   1.1. Desde una instancia de clase
   1.2. Por parámetro
2. Explícita (no automática)
   2.1. casteo

1. Implícita (automática)

En Java hay un moldeo que se hace en forma automática, que se da cada vez que se usa una instancia de una clase derivada donde se puede utilizar una instancia de la clase base:

1.1. Desde una instancia de clase

CuentaBancaria unaCuenta = unaCajaAhorro;

ya que *CajaAhorro* es derivada de *CuentaBancaria* (una *CajaAhorro* "es una" *CuentaBancaria*).

1.2. Por parámetro o Polimorfismo de parámetros

El mismo concepto se aplica a los parámetros, de modo que un método con la siguiente firma:

```
public void agregarCuenta(CuentaBancaria p_cuenta);
```

puede ser invocado así → `unBanco.agregarCuenta(unaCajaAhorro);`

En estos casos se dice que se está haciendo un **moldeo automático o implícito**, pues no hace falta especificarle al lenguaje que se quiere hacer una transformación de tipos. También se lo llama **moldeo hacia arriba**, porque se está transformando el valor llevándolo hacia una clase que está más arriba que la propia en la jerarquía.

Nótese que en cuanto a los **atributos** y **métodos**, cuando se trata de un moldeo hacia arriba, el objeto de la clase padre sólo podrá usar los que tiene declarados en su propia declaración de clase, aún cuando albergue un objeto de la clase hija. Es decir, con la referencia *cuenta* de tipo *CuentaBancaria*, si bien alberga una *CajaAhorro*, no se podrá usar el atributo extraccionesPosibles de la caja de ahorro original, ni tampoco el método getExtraccionesPosibles().

2. Explícita (no automática)

Por otra parte, hay casos en los cuales puede ser necesario hacer un moldeo hacia abajo, lo cual de ninguna manera es automático. Por ejemplo, si se necesita asignar a una variable de clase *CajaAhorro* un valor de tipo *CuentaBancaria*, no es una operación que se haga en forma automática, y debe hacerse en forma explícita, como muestra el siguiente fragmento:

```
CajaAhorro unaCajaAhorro = (CajaAhorro)unaCuentaBancaria;
```

Este tipo de moldeo es más peligroso, pues podrían quedar valores de atributos indefinidos, por ejemplo si *unaCuentaBancaria* no albergara realmente una CajaAhorro. Por eso, la mayoría de los lenguajes tiene algún mecanismo para hacer una verificación en tiempo de ejecución, de modo que esto no ocurra.

¿Para qué puede usarse el moldeo explícito? Un caso interesante es el de tener que acceder a un atributo o método que estuviera declarado en la clase descendiente desde el objeto en la ancestro. El siguiente ejemplo ilustra esto:

```
CuentaBancaria unaCuenta = unaCajaAhorro;
((CajaAhorro)unaCuenta).getExtraccionesPosibles();
```

Observemos que esto no sería correcto si no se tuviera la seguridad de que en *unaCuenta* hay realmente una caja de ahorro, pues el método *getExtraccionesPosibles()* solamente está disponible en una cuenta que sea de tipo *CajaAhorro* y no en *CuentaBancaria*. Por eso, el moldeo hacia abajo debe hacerse siempre sobre objetos que en realidad sean de la clase descendiente, aunque por alguna razón de diseño estén asignados a una variable de la clase padre.

Cabe señalar que esto es cierto solamente en los lenguajes en los cuales el modelo de memoria de objetos trabaja sobre referencias, como Java. En estos lenguajes, la asignación de *unaCajaAhorro* a *unaCuenta* sólo copia referencias a objetos, y la referencia sigue apuntando a un objeto que es de tipo *CajaAhorro.*

Información de tipos en tiempo de ejecución

En muchos lenguajes se puede preguntar por el tipo de datos de un objeto en tiempo de ejecución, aunque la sintaxis es diferente en cada caso. Esto se conoce mediante la sigla inglesa **RTTI** (run-time type information). Esta característica permite un grado de seguridad mayor al hacer el moldeo.

La información sobre el tipo en Java se puede obtener con el operador *instanceof:*

```
if (cuentas.get(i) instanceof  CajaDeAhorro){
        ((CajaDeAhorro)cuentas.get(i)).getExtraccionesPosibles();
}
```

Una forma más precisa de hacer esto último[3] (aunque parezca menos clara) es:

```
if ((cuentas.get(i)).getClass()== CajaDeAhorro.class){
        ((CajaDeAhorro)cuentas.get(i)).getExtraccionesPosibles();
}
```

[3] Usando *ínstanceof se* asegura que el objeto es de clase *CajaDeAhorro* **o algún descendiente**. Usando *getClass()* y el atributo *class* se asegura que el objeto es exactamente de tipo *CajaDeAhorro.*

Sin embargo, esta solución sólo se utiliza en casos extremos, dado que atenta contra la independencia de objetos, y la extensibilidad. Lo habitual y recomendable es basarse en el concepto de vinculación tardía: se envía el mensaje, y el objeto receptor responde según su implementación.

Java brinda notables herramientas para trabajar con información sobre tipos de objetos en tiempo de ejecución. Por ejemplo, existen métodos que permiten conocer los métodos y atributos que tiene una determinada clase, así como su clase ancestro y las interfaces que implementa. Incluso implementa el concepto de **reflexión,** que permite conocer tipos de objetos que no tienen significado en tiempo de compilación, por ejemplo, porque las clases se reciben en tiempo de ejecución a través de una red. Éstos son conceptos muy importantes para permitir el uso de componentes distribuidos e invocación remota de métodos.

Conclusiones

Debe entenderse que lo fundamental del polimorfismo es que rompe con la típica vinculación temprana de los lenguajes compilados tradicionales, que vinculan en tiempo de compilación la llamada a un subprograma con la dirección absoluta de éste. Es obvio que el polimorfismo presta un especial servicio para la reutilización y la extensibilidad. En programación procedural se debería escribir un subprograma para cada tipo de dato, repitiendo tediosamente segmentos de código. Con la vinculación tardía, un método escrito para una clase podrá seguir funcionando bien aun cuando se agreguen clases derivadas que no estaban siquiera previstas por el programador del ancestro.