Herencia

Herencia

La herencia es uno de los mecanismos fundamentales de la programación orientada a objetos, por medio del cual una clase se construye a partir de otra. Una de sus funciones más importantes es proveer el polimorfismo.

La herencia relaciona las clases de manera jerárquica; una clase padre, superclase o clase base sobre otras clases hijas, subclases o clase derivada. Los descendientes de una clase heredan todos los atributos y métodos que sus ascendientes hayan especificado como heredables, además de crear los suyos propios.

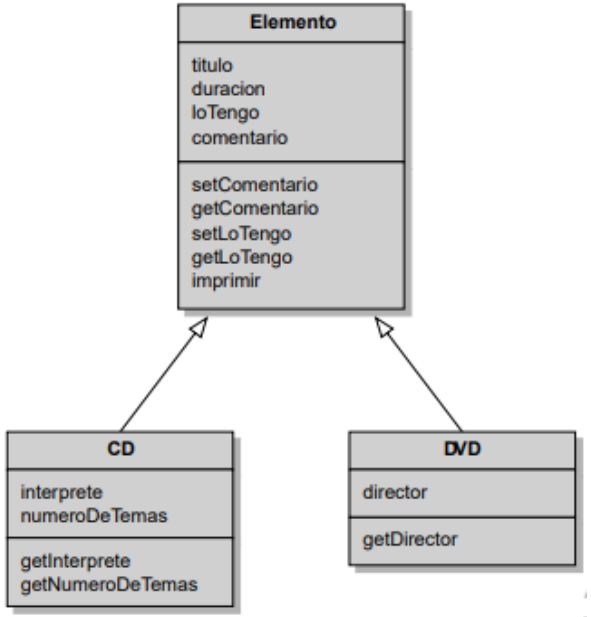
En Java, sólo se permite la herencia simple, o dicho de otra manera, la jerarquía de clases tiene estructura de árbol. El punto más alto de la jerarquía es la clase Object de la cual derivan todas las demás clases.

Para especificar la superclase se realiza con la palabra **extends**; si no se especifica se hereda de Object.

```
public class Punto {...}                  //se herada de Object
public class Punto extends Object {...} //es lo mismo que la anterior
public class Punto3D extends Punto {...}
```

La herencia es un mecanismo que nos ofrece una solución a un problema de duplicación de código. La idea es simple: en lugar de definir las clases CD y DVD completamente independientes, definimos primero una clase que contiene todas las cosas que tienen en común ambas clases. Podemos llamar a esta clase Elemento y luego declarar que un CD es un Elemento y que un DVD es un Elemento.

Finalmente, podemos agregar en la clase CD aquellos detalles adicionales necesarios para un CD y los necesarios para un DVD en la clase DVD. La característica esencial de esta técnica es que necesitamos describir las características comunes sólo una vez.



El diagrama muestra la clase **Elemento** en la parte superior; esta clase define todos los campos y métodos que son comunes a todos los elementos (CD y DVD). Debajo de la clase **Elemento**, aparecen las clases **CD** y **DVD** que contienen sólo aquellos campos y métodos que son únicos para cada clase en particular. Aquí hemos agregado tres métodos: **getInterprete** y **getNumeroDeTemas** en la clase CD y **getDirector** en la clase DVD, para ilustrar el hecho de que las clases CD y DVD pueden definir sus propios métodos.

Esta nueva característica de la programación orientada a objetos requiere algunos nuevos términos. En una situación tal como esta, decimos que la clase CD deriva de la clase Elemento. La clase DVD también deriva de Elemento. Cuando hablamos de programas de Java, también se usa la expresión «la clase CD extiende a la clase Elemento» pues Java utiliza la palabra clave «extends» para definir la relación de herencia.

La flecha en el diagrama de clases (dibujada generalmente con la punta sin rellenar) representa la relación de herencia. La clase Elemento (la clase a partir de la que se derivan o heredan las otras) se denomina clase padre, clase base o superclase. Nos referimos a las clases heredadas (en este ejemplo, CD y DVD) como clases derivadas, clases hijos o subclases.

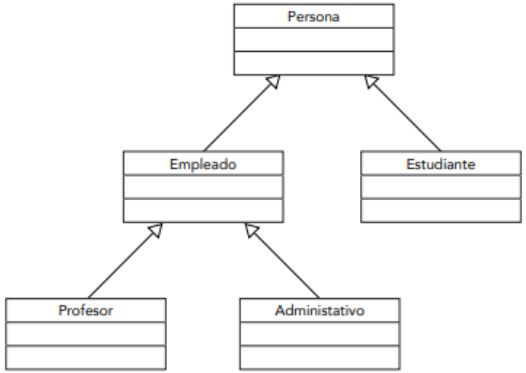
Algunas veces, la herencia también se denomina relación «es un». La razón de esta nomenclatura radica en que la subclase es una especialización de la superclase. Podemos decir que «un CD es un elemento» y que «un DVD es un elemento»

El propósito de usar herencia ahora resulta bastante obvio. Las instancias de la clase CD tendrán todos los campos que están definidos en la clase CD y todos los de la clase Elemento. (CD hereda los campos de la clase Elemento) Las instancias de DVD tendrán todos los campos definidos en las clases DVD y Elemento. Por lo tanto, logramos tener lo mismo que teníamos antes, con la diferencia de que ahora necesitamos definir los campos titulo, duracion, loTengo y comentario sólo una vez (pero podemos usarlos en dos lugares diferentes).

Concepto de superclase y subclase

La generalización es una relación entre clases en las que hay una clase padre, llamada **superclase,** y una o más clases hijas especializadas, a las que se les denomina **subclases**. La herencia es el mecanismo mediante el cual se implementa la relación de generalización.

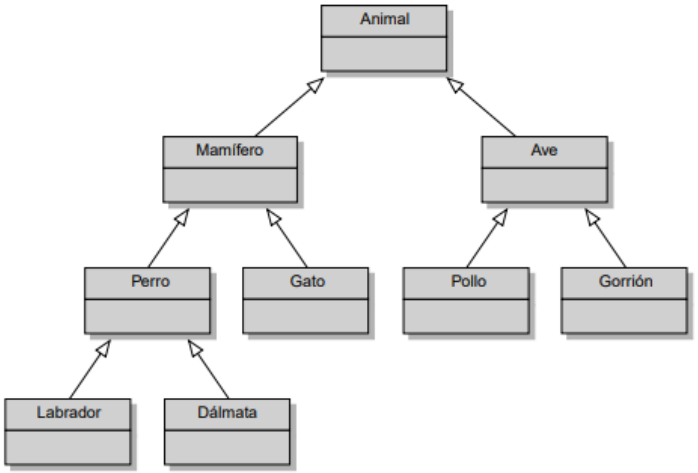
En la práctica, cuando se codifica un sistema, se habla de herencia en lugar de generalización. Cuando hay herencia, la clase hija o subclase adquiere los atributos y métodos de la clase padre. Como se ilustra en la figura, hay herencia cuando existe la relación es un entre los objetos de una clase hijo y una padre. Por ejemplo: un Profesor es un Empleado, un Administrativo es un Empleado, un Empleado es una Persona, un Estudiante es una Persona.

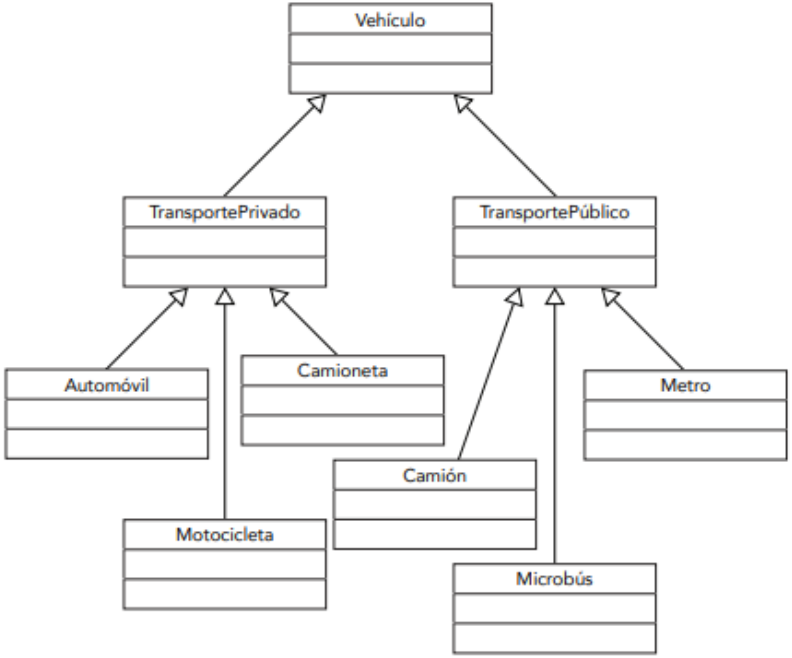


Jerarquías de clase

Se pueden heredar más de dos subclases a partir de la misma superclase y una subclase puede convertirse en la superclase de otras subclases. En consecuencia, las clases forman una jerarquía de herencia. Probablemente, el ejemplo más conocido de una jerarquía de herencia es la clasificación de las especies que usan los biólogos.

En la Figura, se muestra una pequeña parte de esta clasificación: podemos ver que un dálmata es un perro, que a su vez es un mamífero y que también es un animal. Sabemos algunas cosas sobre los labradores, por ejemplo, que son seres vivos, pueden ladrar, y comen carne. Si miramos un poco más de cerca, vemos que sabemos algunas de estas cosas no porque son labradores sino porque son perros, mamíferos o animales. Una instancia de la clase Labrador (un labrador real) tiene todas las características de un labrador, de un perro, de un mamífero y de un animal, porque un labrador es un perro, que a su vez es un mamífero, y así sucesivamente.





Los métodos creados en la clase hija tienen acceso tanto a los métodos, como a los atributos públicos de la clase padre.

Jerarquía de herencia: la **Motocicleta** es un **TransportePrivado** y a la vez es un **Vehículo**, ya que todo **TransportePrivado** es un **Vehículo**. Lo mismo puede decirse del **Automóvil** y la **Camioneta**. El Camión es un **TransportePublico** y también es un **Vehículo**, lo mismo que el **Microbús** y el **Metro**.

Si la clase **Vehículo** tiene los métodos públicos **ponerEnMarcha(), acelerar()** y **detener(),** entonces todos los objetos descendientes de Vehículo, es decir, Automóvil, Camión, etc., pueden usar estos métodos.

Otras consideraciones sobre la jerarquía

- ✓ La herencia permite definir nuevas clases denominadas “derivadas, hijas o subclases”, a partir de clases ya existentes, llamadas “base, padre o superclase”. De esta manera los objetos pueden construirse en base a otros objetos ya creados.

- ✓ La “herencia” o “derivación de clases” es el mecanismo para compartir automáticamente los métodos y datos de la “clase base”, añadiendo otros nuevos a las “clases derivadas”. Es decir, la herencia impone una organización jerárquica entre clases, permitiendo que los datos y métodos de una clase sean heredados por sus descendientes.

- ✓ Los constructores no se heredan, aunque existe llamadas implícitas a los constructores de las clases padre. Al crear una instancia de una clase hija, lo primero que se hace es llamar al constructor de la clase padre para que se inicialicen los atributos de la clase padre, sino se especifica, se llama al constructor sin parámetros. Por ello, si se quiere invocar otro constructor, la primera sentencia del constructor de la clase hija debe ser la llamada al constructor de la clase padre.

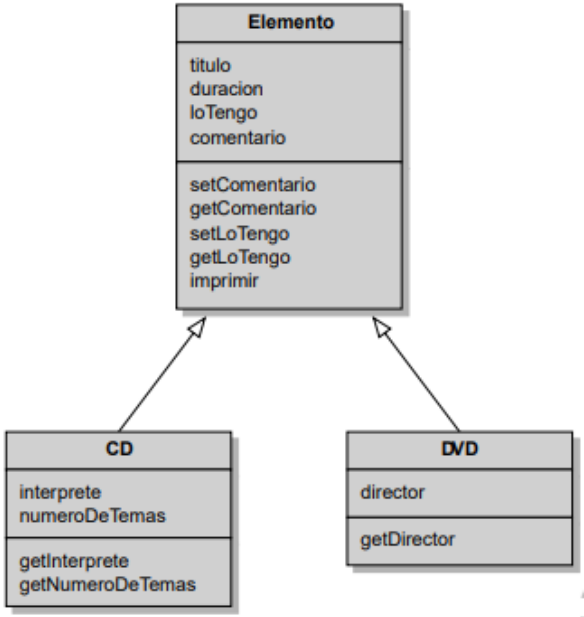
- ✓ Cuando definimos una clase hija, indicamos de alguna manera cuál es su clase padre y luego definimos los atributos y métodos adicionales propios de la clase hija. En Java, la relación de herencia se especifica con la palabra reservada **extends**, lo que significa que la definición de una clase extiende la definición de la superclase.

Herencia en Java

Antes de discutir más detalles de la herencia, veremos cómo se expresa en el lenguaje Java. Volviendo al ejemplo inicial del CD y DVD. Aquí presentamos un segmento de código de la clase Elemento:

```
public class Elemento
{
  private String titulo;
  private int duracion;
  private boolean lotengo;
  private String comentario;
  // se omitieron constructores y métodos
}
```

Hasta ahora, esta clase no tiene nada especial: comienza con una definición normal de clase y declara los campos de la manera habitual.



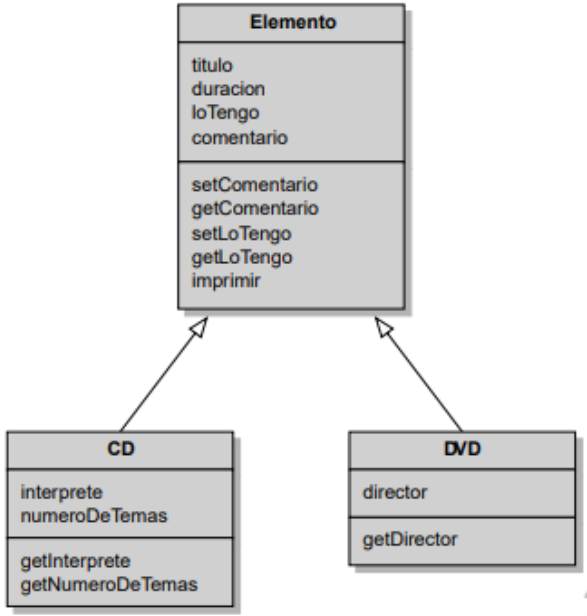
Herencia en Java

A continuación, examinamos el código de la clase CD:

```
public class CD extends Elemento
{
  private String interprete;
  private int numeroDeTemas;
  // se omitieron constructores y métodos
}
```

En este código hay dos puntos importantes para resaltar. En primer lugar, la palabra clave **extends** define la relación de herencia. La frase «extends Elemento» especifica que esta clase es una subclase de la clase Elemento.

En segundo término, la clase CD define sólo aquellos campos que son únicos para los objetos CD (interprete y numeroDeTemas). Los campos de Elemento se heredan y no necesitan ser listados en este código. No obstante, los objetos de la clase CD tendrán los campos titulo, duración y así sucesivamente.



Herencia en Java

Para los objetos de las otras clases, los objetos DVD o CD aparecen como todos los otros tipos de objetos. En consecuencia, los miembros definidos como públicos, ya sea en la superclase o en la subclase, serán accesibles para los objetos de otras clases, pero los miembros definidos como privados serán inaccesibles.

En realidad, la regla de privacidad también se aplica entre una subclase y su superclase: una subclase no puede acceder a los miembros privados de su superclase. Se concluye que si un método de una subclase necesita acceder o modificar campos privados de su superclase, entonces la superclase necesitará ofrecer los métodos de acceso y/o métodos de modificación apropiados. Una subclase puede invocar a cualquier método público de su superclase como si fuera propio, no se necesita ninguna variable.

Herencia e inicialización

Cuando creamos un objeto, el constructor de dicho objeto tiene el cuidado de inicializar todos los campos con algún estado razonable. Tenemos que ver más de cerca cómo se hace esto en las clases que se heredan a partir de otras clases. Cuando creamos un objeto CD, pasamos varios parámetros al constructor de CD: el título, el nombre del intérprete, el número de temas y el tiempo de duración.

Algunos de estos parámetros contienen valores para los campos definidos en la clase Elemento y otros valores para los campos definidos en la clase CD. Todos estos campos deben ser correctamente inicializados y el Código de la figura muestra los segmentos de código que se usan para llevar a cabo esta inicialización en Java.

```
public class Elemento
{
  private String titulo;
  private int duracion;
  private boolean loTengo;
  private String comentario;
  /**
  * Inicializa los campos del elemento.
  * @param elTitulo el título de este elemento.
  * @param tiempo La duración de este elemento.
  */
 public Elemento(String elTitulo, int tiempo)
 {
      titulo = elTitulo;
      duracion = tiempo;
      loTengo = false;
      comentario = "";
  }
// Se omitieron métodos
}
```

Se pueden hacer varias observaciones con respecto a estas clases.

En primer lugar, la clase **Elemento** tiene un constructor aun cuando no tenemos intención de crear, de manera directa, una instancia de la clase Elemento. 2 Este constructor recibe los parámetros necesarios para inicializar los campos de Elemento y contiene el código para llevar a cabo esta inicialización.

```
public class Elemento
{
  private String titulo;
  private int duracion;
  private boolean loTengo;
  private String comentario;
  /**
  * Inicializa los campos del elemento.
  * @param elTitulo el título de este elemento.
  * @param tiempo La duración de este elemento.
  */
 public Elemento(String elTitulo, int tiempo)
 {
        titulo = elTitulo;
        duracion = tiempo;
        loTengo = false;
        comentario = "";
  }
// Se omitieron métodos
}
```

```java
public class CD extends Elemento
{
  private String interprete;
  private int numeroDeTemas;
  /**
  * Constructor de objetos de la clase CD
  * @param elTitulo El título del CD.
  * @param elInterprete El intérprete del CD.
  * @param temas El número de temas del CD.
  * @param tiempo La duración del CD.
  */
  public CD(String elTitulo, String elInterprete, int temas, int tiempo)
  {
      super(elTitulo, tiempo);
      interprete = elInterprete;
      numeroDeTemas = temas;
  }
// Se omitieron métodos
}
```

En segundo lugar, el constructor CD recibe los parámetros necesarios para inicializar tanto los campos de Elemento como los de CD. La clase Elemento contiene la siguiente línea de código:

```java
super(elTitulo, tiempo);
```

La palabra clave **super** es, en realidad, una llamada al constructor de la superclase. El efecto de esta llamada es que se ejecuta el constructor de Elemento, formando parte de la ejecución del constructor del CD. Cuando creamos un CD, se invoca al constructor de CD, quien en su primer sentencia lo convierte en una llamada al constructor de **Elemento.** El constructor de Elemento inicializa sus campos y luego retorna al constructor de CD que inicializa los restantes campos definidos en la clase CD. Para que esta operación funcione, los parámetros necesarios para la inicialización de los campos del elemento se pasan al constructor de la superclase como parámetros en la llamada a **super.**

Consideraciones sobre constructores

- ✓ En Java, un constructor de una subclase siempre debe invocar en su primer sentencia al constructor de la superclase. Si no se escribe una llamada al constructor de una superclase, el compilador de Java insertará automáticamente una llamada a la superclase, para asegurar que los campos de la superclase se inicialicen adecuadamente.

- ✓ La inserción automática de la llamada a la superclase sólo funciona si la superclase tiene un constructor sin parámetros (ya que el compilador no puede adivinar qué parámetros deben pasarse); en el caso contrario, Java informa un error.

- ✓ En general, es una buena idea la de incluir siempre en los constructores llamadas explícitas a la superclase, aun cuando sea una llamada que el compilador puede generar automáticamente. Consideramos que esta inclusión forma parte de un buen estilo de programación, ya que evita la posibilidad de una mala interpretación y de confusión en el caso de que un lector no esté advertido de la generación automática de código.

Consideraciones sobre la herencia

- ✓ La clase derivada (subclase o clase hija) heredan datos y métodos de clases base (superclase o clase padre) con visibilidad pública, protegida y de paquetes.
- ✓ Las subclases pueden añadir nuevos miembros es decir, datos y métodos específicos.
- ✓ Los métodos de clase padre pueden ser sobrescritos o redefinidos (override) en clases hijas.
- ✓ En una subclase es posible referirse al objeto intrínseco propio con el término this y al objeto de la clase padre mediante super.

Subtipos y asignación

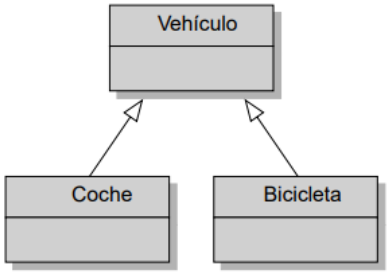
Cuando queremos asignar un objeto a una variable, el tipo del objeto debe coincidir con el tipo de la variable. Por ejemplo:

```
coche   miCoche = new Coche();
```

Es una asignación válida porque se asigna un objeto de tipo coche a una variable declarada para contener objetos de tipo Coche.

Ahora que conocemos la herencia debemos establecer la regla de tipos de manera más completa: una variable puede contener objetos del tipo declarado o de cualquier subtipo del tipo declarado.

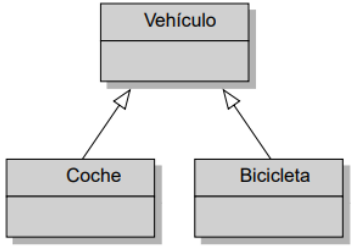
Imagine que tenemos una clase **Vehículo** con dos subclases, coche y Bicicleta. En este caso la regla de tipos admite que las siguientes sentencias son todas legales:



Subtipos y asignación

Imagine que tenemos una clase **Vehículo** con dos subclases, coche y Bicicleta. En este caso la regla de tipos admite que las siguientes sentencias son todas legales:

```
Vehiculo v1 = new Vehiculo();
Vehiculo v2 = new Coche();
Vehiculo v3 = new Bicicleta();
```



El tipo de una variable declara qué es lo que puede almacenar. La declaración de una variable de tipo **Vehículo** determina que esta variable puede contener vehículos. Pero como un coche es un vehículo, es perfectamente legal almacenar un coche en una variable que está pensada para almacenar vehículos. (Puede pensar en la variable como si fuera un garaje: si alguien le dice que puede estacionar un vehículo en un garaje, puede pensar que también es correcto estacionar un coche o una bicicleta en el garaje.)

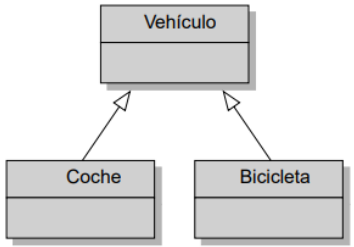
Subtipos y asignación

Este principio se conoce como **sustitución.** En los lenguajes orientados a objetos podemos sustituir por un objeto de una subclase en el lugar donde se espera un objeto de una superclase porque el objeto de la subclase es un caso especial de la superclase.

Sin embargo, no está permitido hacer esto de otra manera:

```
Coche a1 = new Vehiculo();      // ¡Es un error!
```

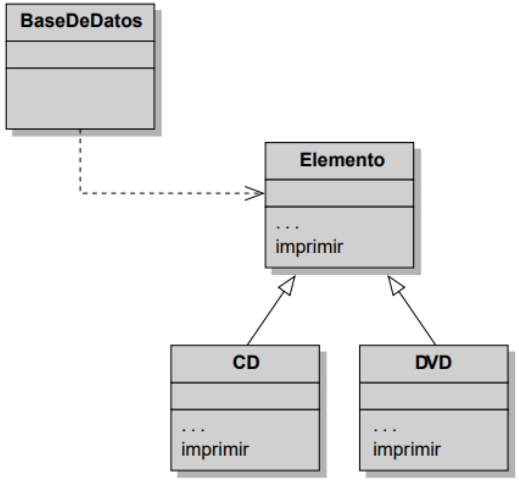
Esta sentencia intenta almacenar un objeto **Vehículo** en un objeto Coche. Java no permitirá esta asignación e informará un error cuando trate de compilar esta sentencia. La variable está declarada para permitir el almacenamiento de coches. Un vehículo, por otro lado, puede o no ser un coche, no sabemos. Por lo tanto, la sentencia es incorrecta y no está permitida.



Sobrescribir

Una subclase puede sobrescribir la implementación de un método. Para hacerlo, la subclase declara un método con la misma signatura que la superclase pero con un cuerpo diferente. El método sobrescrito tiene precedencia cuando se invoca sobre objetos de la subclase.

La sobrescritura es una situación en la que un método está definido en una superclase (en este ejemplo, el método imprimir de la clase Elemento) y un método, con exactamente la misma signatura, está definido en la subclase. En esta situación, los objetos de la subclase tienen dos métodos con el mismo nombre y la misma signatura: uno heredado de la superclase y el otro propio de la subclase.



La clase Elemento tiene un método imprimir que imprime todos los campos que están declarados en Elemento (aquellos que son comunes a los CD y a los DVD) y las subclases CD y DVD imprimen los campos específicos de los objetos CD y DVD respectivamente.

¿Por qué es útil la herencia?

1. Reutilización de código.

Las clases debidamente creadas y verificadas, pueden reutilizarse en otros programas, ahorrando tiempo y esfuerzo en el nuevo desarrollo.

Las clases pueden agruparse en paquetes, librerías o bibliotecas para facilitar su distribución. Los programas que reutilizan estas bibliotecas, sólo necesitan conocer los nombres de dichas clases y su interfaz pública, de esta manera es posible instanciar objetos (composición) o crear nuevas clases derivadas (herencia) sin necesidad de comprender la complejidad de la implementación del código fuente de las clases base.

La mayoría de los lenguajes de programación orientados a objetos cuentan con librerías de clases predefinidas que facilitan el desarrollo de nuevas aplicaciones informáticas.

¿Por qué es útil la herencia?

2. Creación de programas extensibles

Un programa diseñado en torno al "polimorfismo" es fácil de mantener y ampliar. Para ampliar un programa "polimórfico" simplemente se crea una nueva clase derivada a partir de la misma clase base, que heredaron los otros objetos genéricos. La nueva clase derivada puede manejarse por el mismo programa sin modificación, como el programa es únicamente un gestor para un conjunto de objetos genéricos, los errores se aíslan automáticamente en los mismos objetos. Además, una vez que una clase base es depurada, cualquiera de los errores en una nueva clase derivada es consecuencia del nuevo código en la clase derivada.

Polimorfismo. Significa la cualidad de tener más de una forma. En el contexto de la POO, el polimorfismo se refiere al hecho de que una operación de una clase padre puede sobre-escribirse en diferentes clases derivadas. En otras palabras, diferentes objetos pertenecientes a una misma jerarquía de clases o conjunto de clases que implementan una interfaz, reaccionan al mismo mensaje de modo diferente.

Clase abstracta

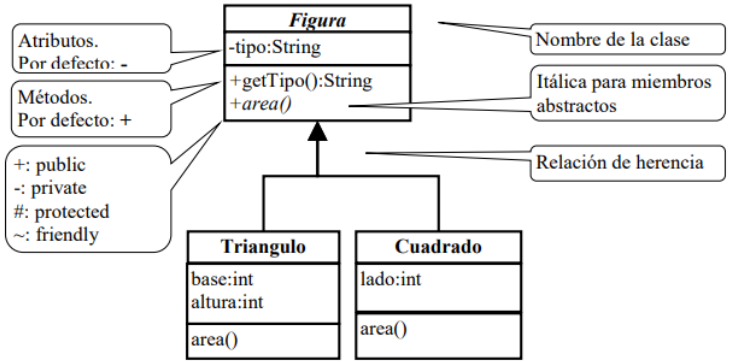
Java proporciona un tipo especial de clase llamada clase abstracta que permite ayudar a la organización de las clases basadas en métodos comunes. Una clase abstracta permite declarar prototipos de métodos comunes en una sola clase sin tener que implementar su código.

Características:

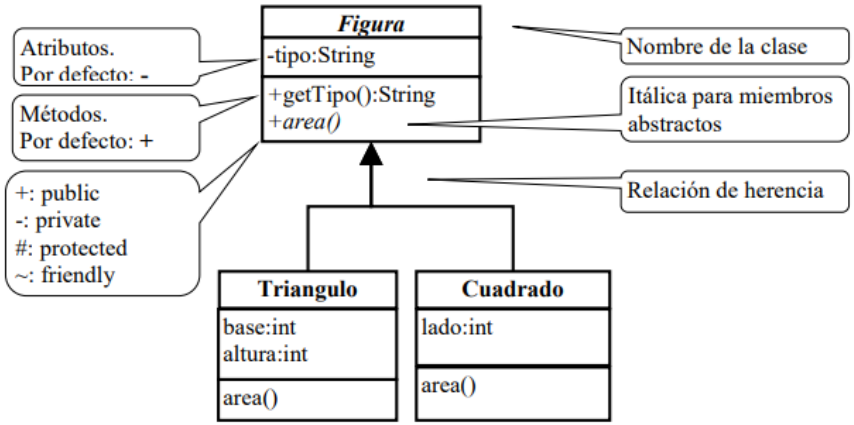
- ✓ •Los métodos abstractos contienen sólo su declaración o prototipo (nombre y parámetros). La responsabilidad de implementar su código se delega a las clases derivadas.
- ✓ Una clase que declare al menos un método abstracto es considerada como clase abstracta.
- ✓ No es posible crear instancias (objetos) de una clase abstracta. Sin embargo, es posible crear referencias.
- ✓ Cualquier subclase que extienda a la clase abstracta debe implementar el código de los métodos abstractos.

Clase abstracta

Una clase abstracta es una clase que no se puede instanciar, pero si heredar. También, se pueden definir constructores. Se utilizan para englobar clases de diferentes tipos, pero con aspectos comunes. Se pueden definir métodos sin implementar y obligar a las subclases a implementarlos. Por ejemplo, podemos tener una clase Figura, que representa una figura general, y subclases con figuras concretas (Triangulo, Circulo...). Podemos establecer métodos comunes como dibujar, que sin conocer el tipo de figura, sea capaz de dibujarla. Pero para que esto funcione correctamente, cada figura concreta debe implementar su versión particular de dibujar.



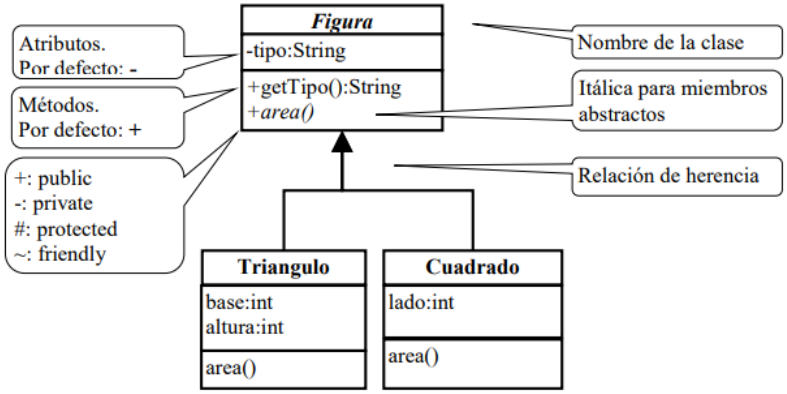
Clase abstracta



Si añadimos el calificador **abstract** después del calificativo **public** del método, le convertimos en abstracto y no hace falta que lo implementemos, basta con definirlo. Si añadimos **abstract** después del calificativo **public** de la clase, la convertimos en clase abstracta; esta acción es obligatoria si algún método es abstracto. La ventaja en utilizar clases abstractas, es que podemos trabajar con variables o parámetros de tipo Figura y llamar a los métodos comunes sin saber a priori el tipo concreto de Figura.

```
public abstract class Figura {
   private String tipo;
   public Figura(String tipo) {
      this.tipo = tipo;
   }
   // getters & setters
   public abstract double area();
}
```

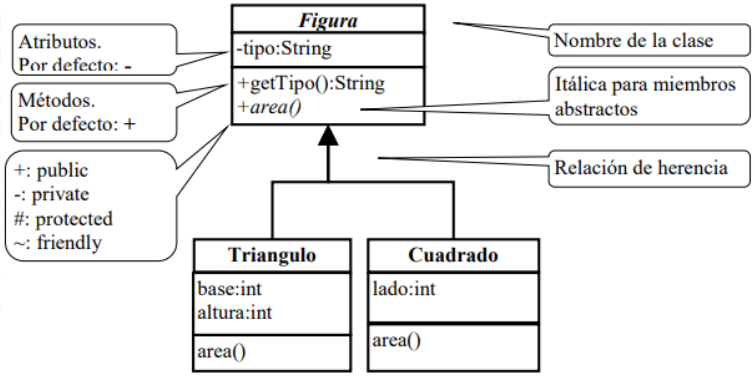
Polimorfismo



La ventaja en utilizar clases abstractas, es que podemos trabajar con variables o parámetros de tipo Figura y llamar a los métodos comunes sin saber a priori el tipo concreto de Figura. Esto permite en el futuro, añadir nuevas Figuras, sin cambiar las clases ya desarrolladas. A este concepto se le llama **polimorfismo.**

El polimorfismo es el modo en que la POO implementa la polisemia, es decir, un solo nombre para muchos conceptos. Este tipo de polimorfismo se debe resolver en tiempo de ejecución y por ello hablamos de polimorfismo dinámico, a diferencia del polimorfismo estático que se puede resolver en tiempo de compilación. Recordemos que éste último se implementaba mediante la sobrecarga de los métodos.

Polimorfismo



Triangulo.java

```
public class Triangulo extends Figura {
   private int base, altura;
   public Triangulo(String tipo, int base, int altura) {
      super(tipo);
      this.setBase(base);
      this.setAltura(altura);
   }
   // getters & setters
   @Override
   public double area() {
      return (double) this.base * this.altura / 2;
   }
}
```

Cuadrado.java

```
public class Cuadrado extends Figura {
   private int lado;
   public Cuadrado(String tipo, int lado) {
      super(tipo);
      this.setLado(lado);
   }
   // getters & setters
   @Override
   public double area() {
      return (double) this.lado * this.lado;
   }
}
```

Polimorfismo.java

```
...
   public void polimorfismo(Figura una){
      System.out.println("Tipo: " + una.getTipo());
      System.out.println("Area: " + una.area());
   }
...
```