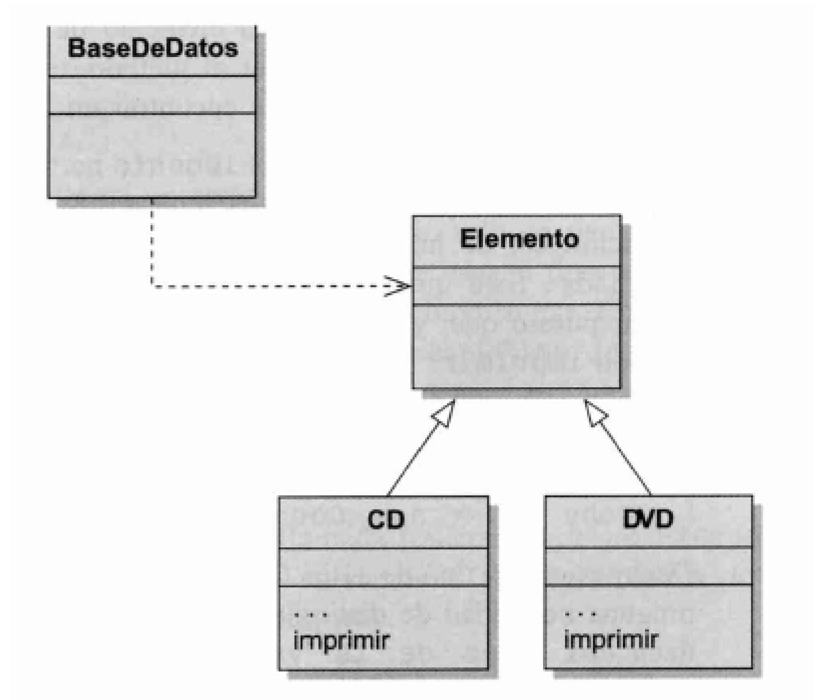
Programación orientada a objetos

Capítulo 9

Algo mas sobre herencias

Tema 9: Algo más sobre herencia. Semana 9	
1- El método imprimir. 2- Tipo Estático y Tipo Dinámico. 3- Sobreescribir. 4- Búsqueda dinámica del método. 5- Llamada a super en métodos. 6- Método Polimórfico. 7- Métodos de Object: toString. 8- Acceso protegido. 9- Otro ejemplo de herencia con sobreescritura	1- Estudiar el capítulo 9 del libro base para la "Unidad Didáctica II". 2- Realizar los ejercicios correspondientes del libro base. 3- Realizar los ejercicios resueltos en examines de años anteriores en los que se utilice la herencia.


BaseDeDatos
Elemento
CD
...
imprimir
DVD
...
imprimir

Tipos estáticos y dinámicos

El **tipo estático** de una variable v es el tipo declarado en el código fuente en la sentencia de declaración de la variable.

- Denominamos *tipo estático* al tipo declarado de una variable porque la variable se declara en el código fuente, la representación estática del programa.
- Denominamos *tipo dinámico* al tipo del objeto almacenado en una variable porque depende de su asignación en tiempo de ejecución, el comportamiento dinámico del programa.

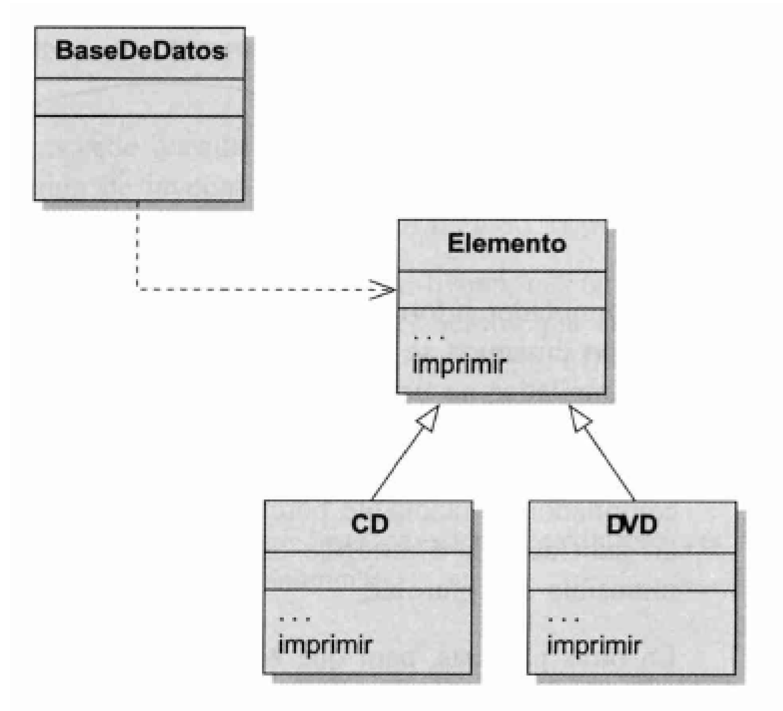
El **tipo dinámico** de una variable v es el tipo del objeto que está almacenado actualmente en la variable v.

```
Vehiculo v1 = new Coche();
```

Elemento elemento

Sobreescribir

Sobrescritura. 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.



- El control de tipos que realiza el compilador es sobre el tipo estático, pero en tiempo de ejecución los métodos que se ejecutan son los que corresponden al tipo dinámico.

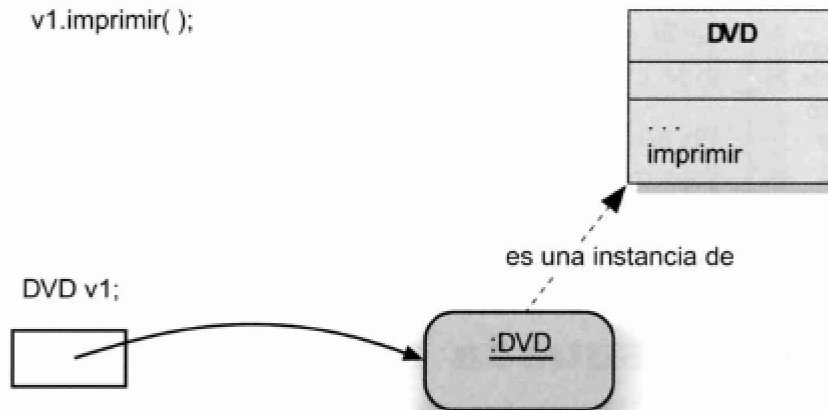
```
public class Elemento
{
   . . .
   public void imprimir()
   {
      System.out.print(titulo + " (" + duracion + "
minutos) ");
      if (loTengo) {
         System.out.println("*");
      } else {
         System.out.println();
      }
      System.out.println(" " + comentario);
   }
}
public class CD extends Elemento
{
   . . .
   public void imprimir()
   {
      System.out.println(" " + interprete);
      System.out.println(" temas: " + numeroDeTemas);
   }
}
public class DVD extends Elemento
{
   . . .
   public void imprimir()
   {
      System.out.println(" director: " + director);
   }
}
```

Búsqueda de un método un único objeto

```
v1.imprimir();
```

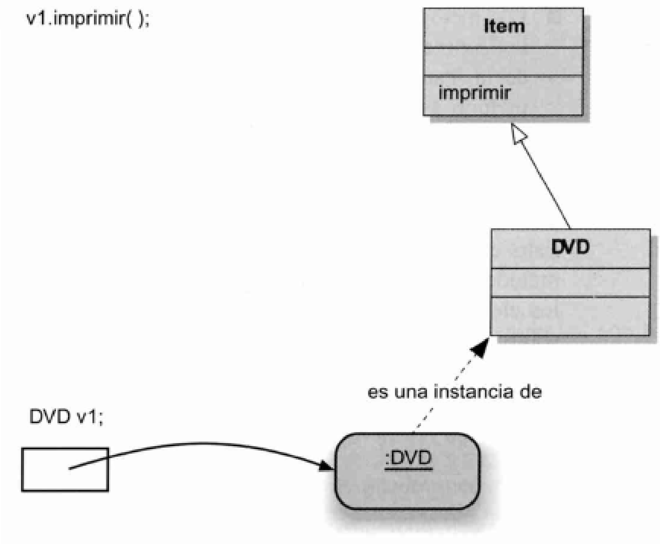
Cuando se ejecute esta sentencia, se invoca al método `imprimir` en los siguientes pasos:

1. Se accede a la variable `v1`.
2. Se encuentra el objeto almacenado en esa variable (siguiendo la referencia).
3. Se encuentra la clase del objeto (siguiendo la referencia «es instancia de»).
4. Se encuentra la implementación del método `imprimir` en la clase y se ejecuta.



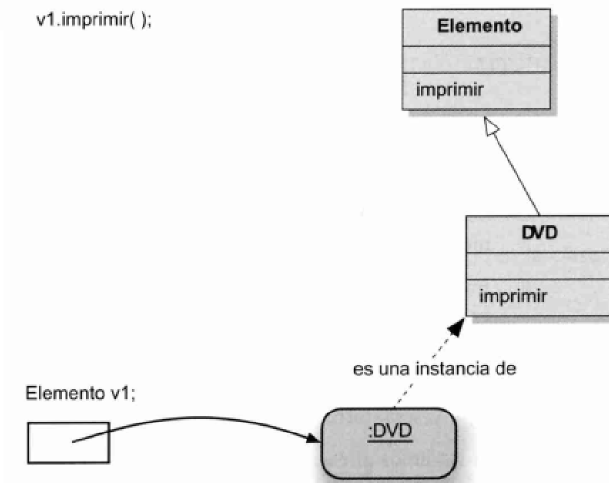
Búsqueda de un método cuando hay herencia

1. Se accede a la variable `v1`.
2. Se encuentra el objeto almacenado en esa variable (siguiendo la referencia).
3. Se encuentra la clase del objeto (siguiendo la referencia «es instancia de»).
4. No se encuentra ningún método `imprimir` en la clase `DVD`.
5. Dado que no se encontró ningún método que coincida, se busca en la superclase un método que coincida. Si no se encuentra ningún método en la superclase, se busca en la siguiente superclase (si es que existe). Esta búsqueda continúa hacia arriba por toda la jerarquía de herencia de la clase `Object` hasta que se encuentre definitivamente un método. Tenga en cuenta que, en tiempo de ejecución, debe encontrarse definitivamente un método que coincida, de lo contrario la clase no habría compilado.
6. En nuestro ejemplo, el método `imprimir` es encontrado en la clase `Elemento` y es el que será ejecutado.



Búsqueda con pilomorfismo y sobre escritura

- El tipo declarado de la variable `v1` ahora es `Elemento`, no `DVD`.
- El método `imprimir` está definido en la clase `Elemento` y redefinido (o sobrescrito) en la clase `DVD`.
- El método que se encuentra primero y que se ejecuta está determinado por el tipo dinámico, no por el tipo estático. En otras palabras, el hecho de que el tipo declarado de la variable `v1` ahora es `Elemento` no tiene ningún efecto. La instancia con la que estamos trabajando es de la clase `DVD`, y esto es todo lo que cuenta.
- Los métodos sobrescritos en las subclases tienen precedencia sobre los métodos de las superclases. Dado que la búsqueda de método comienza en la clase dinámica de la instancia (al final de la jerarquía de herencia) la última redefinición de un método es la que se encuentra primero y la que se ejecuta.
- Cuando un método está sobrescrito, sólo se ejecuta la última versión (la más baja en la jerarquía de herencia). Las versiones del mismo método en cualquier superclase no se ejecutan automáticamente.



Llamada a "super" en métodos

```
public void imprimir()
{
  super.imprimir();
  System.out.println("          " + interprete);
  System.out.println("          temas: ") + numeroDeTemas);
}
```

- Al contrario que las llamadas a `super` en los constructores, el nombre del método de la superclase está explícitamente establecido. Una llamada a `super` en un método siempre tiene la forma

  `super.`*`nombre-del-método`* `(` *`parámetros`* `)`

  La lista de parámetros por supuesto que puede quedar vacía.

- Nuevamente, y en contra de la regla de las llamadas a `super` en los constructores, la llamada a `super` en los métodos puede ocurrir en cualquier lugar dentro de dicho método. No tiene por qué ocurrir en su primer sentencia.

- Al contrario que en las llamadas a `super` en los constructores, no se genera automáticamente ninguna llamada a `super` y tampoco se requiere ninguna llamada a `super`, es completamente opcional. De modo que el comportamiento por defecto presenta el efecto de un método de una subclase ocultando completamente (sobrescribiendo) la versión de la superclase del mismo método.

Métodos polimórficos

Método polimórfico. Las llamadas a métodos en Java son polimórficas. El mismo método puede invocar en diferentes momentos diferentes métodos dependiendo del tipo dinámico de la variable usada para hacer la invocación.

```
elemento.imprimir();
```

puede invocar al método `imprimir` de `CD` en un momento dado y al método `imprimir` de `DVD` en otro momento, dependiendo del tipo dinámico de la variable `elemento`.

Método object: toString

```
public class Elemento
{
    . . .
    public String toString()
    {
      String linea1 = titulo + " (" + duracion + "
minutos)");
      if (loTengo) {
        return linea1 + "*\n" + "    " + comentario +
"\n";
      }
      else {
        return linea1 + "\n" + "    " + comentario + "\n";
    }
  }
  public void imprimir()
  {
    System.out.println(toString());
  }
}
public class CD extends Elemento
{
  . . .
  public String toString()
  {
    return super.toString() + "    " + interprete +
      "\n    temas: " + numeroDeTemas + "\n";
  }
  public void imprimir()
  {
    System.out.println(toString());
  }
}
```

- Reescribimos el método "toString"

```
public class BaseDeDatos
{
   // se omitieron los campos, los constructores y los restantes métodos
   /**
   * Imprime una lista en la terminal de texto de todos
los CD y
   * DVD actualmente almacenados.
   */
   public void listar()
   {
      for(Elemento elemento : elementos ) {
         System.out.println(elemento);
      }
   }
}
```

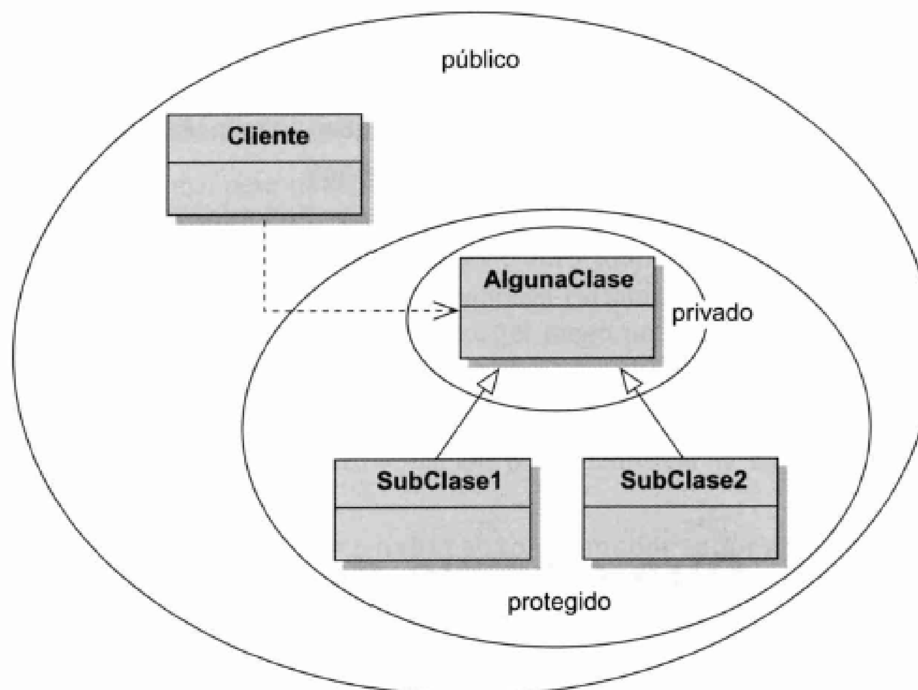
- El *ciclo for-each* recorre todos los elementos y los ubica en una variable con el tipo estático `Elemento`. El tipo dinámico es tanto `CD` como `DVD`.
- Dado que este objeto se imprime mediante `System.out` y no es una cadena, se invoca automáticamente su método `toSTring`.
- La invocación a este método es válida porque la clase `Elemento` (¡el tipo estático!) posee un método `toString`. (Recuerde: el control de tipos se realiza con el tipo estático. Esta llamada no sería permitida si la clase `Elemento` no tiene un método `toString`.) No obstante, el método `toString` de la clase `Object` garantiza que este método esté disponible siempre para cualquier clase.
- La salida aparece adecuadamente con todos los detalles necesarios porque cada tipo dinámico posible (`CD` y `DVD`) sobrescribe el método `toString` y la búsqueda dinámica del método asegura que se ejecute el método redefinido.

Acceso “protegido”

```
protected String getTitulo()
{
  return titulo();
}
```

El acceso protegido permite acceder a los campos o a los métodos dentro de una misma clase y desde todas las subclases, pero no desde otras clases. El método `getTitulo` que se muestra en Código 9.5 puede invocarse desde la clase `Elemento` o desde cualquier subclase, pero desde otras clases.

La declaración de un campo o un método como **protegido** (`protected`) permite su acceso directo desde las subclases (directas o indirectas).



Términos introducidos en este capítulo

tipo estático, tipo dinámico, sobrescritura, redefinición, búsqueda de método, despacho de método, método polimórfico, protegido

Resumen de conceptos

- **tipo estático** El tipo estático de una variable v es el tipo declarado en el código fuente en la sentencia de declaración de la variable.
- **tipo dinámico** El tipo dinámico de una variable v es el tipo del objeto que está actualmente almacenado en v.
- **sobrescritura** 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 en las llamadas a métodos sobre objetos de la subclase.
- **método polimórfico** Las llamadas a métodos en Java son polimórficas. La misma llamada a un método en diferentes momentos puede invocar diferentes métodos, dependiendo del tipo dinámico de la variable usada para hacer la invocación.
- **toString** Cada objeto en Java tiene un método `toString` que puede usarse para devolver una representación String del mismo. Típicamente, para que sea útil, una clase debe sobrescribir este método.
- **protected** La declaración de un campo o de un método como `protected` permite el acceso directo al mismo desde las subclases (directas o indirectas).