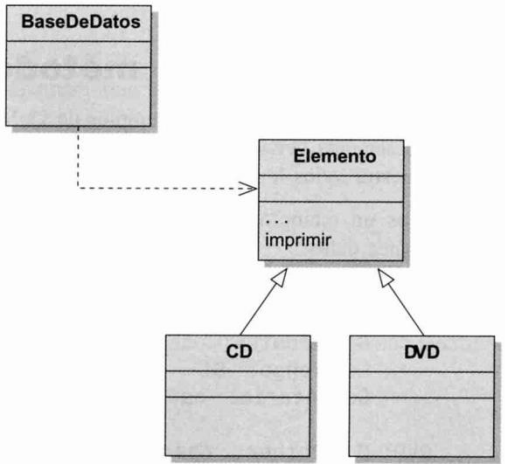
Programación orientada a objetos

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.



Impresión, versión 1:
el método **imprimir**
en la superclase

```
título:   A Swingin' Affair (64 minutos) *
              es mi álbum favorito de Sinatra

título:   O Brother, Where Art Thou? (106 minutos)
              ¡La mejor película de los hermanos Coen!
```

Vemos en este caso que falta la información sobre el intérprete del CD y el número de temas que contiene, así como también falta el director de la película en DVD. El motivo de esto es muy simple: en esta versión, el método `imprimir` está implementado en la clase `Elemento`, no en las clases `DVD` o `CD` (Figura 9.1). En los métodos de `Elemento` sólo están disponibles los campos declarados en la clase `Elemento`. Si tratamos de acceder al campo intérprete del CD desde el método `imprimir` de `Elemento`, se informará un error. Este hecho ilustra el importante principio de que la herencia tiene una sola vía: `CD` hereda los campos de `Elemento` pero `Elemento` continúa sin conocer nada sobre los campos de sus subclases.

¿Cuál es el problema?

- El método print en la clase Elemento solo imprime los campos comunes
- La herencia es de sentido único:
  - Una subclase hereda los campos de la superclase
  - Pero la superclase no sabe nada de los campos de las subclases

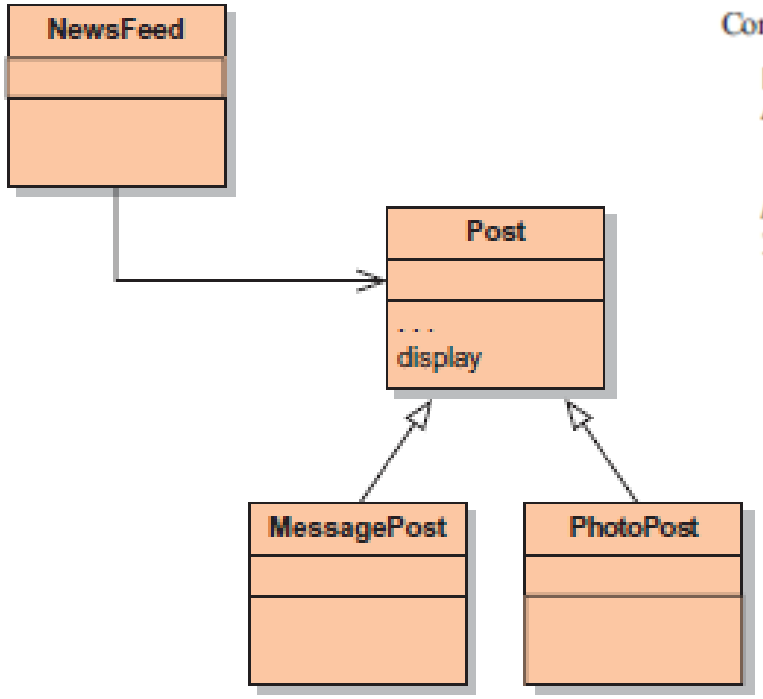
Lo que queremos

```
CD: A Swingin' Affair (64 mins)*
    Frank Sinatra
    tracks: 16
    my favourite Sinatra album

DVD: O Brother, Where Art Thou? (106 mins)
     Joel & Ethan Coen
     The Coen brothers' best movie!
```

Lo que tenemos

```
title: A Swingin' Affair (64 mins)*
       my favourite Sinatra album

title: O Brother, Where Art Thou? (106 mins)
       The Coen brothers' best movie!
```

Version 1: método display en la superclase

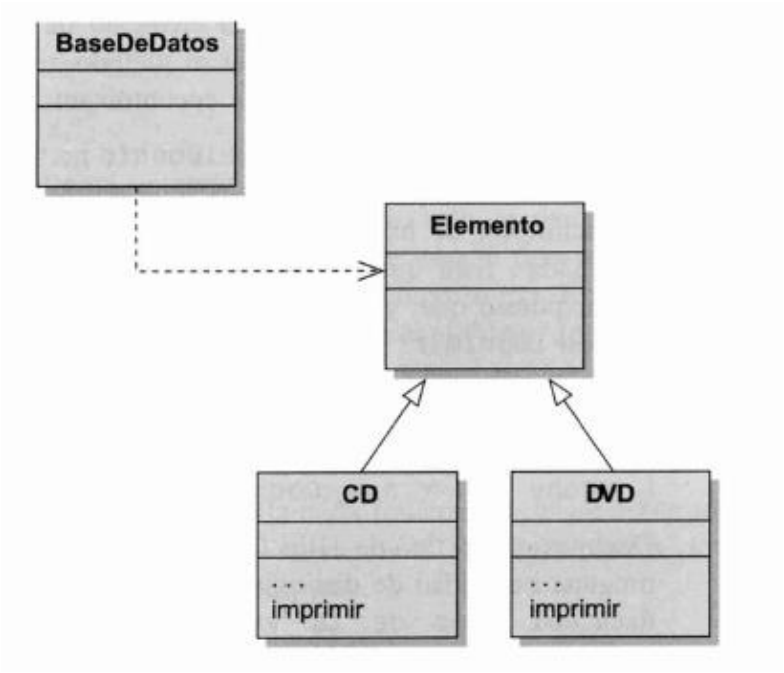


Compare this with the second *network* version (with inheritance), which prints only

```
Leonardo da Vinci
40 seconds ago   -  2 people like this.
  No comments.

Alexander Graham Bell
12 minutes ago   -  4 people like this.
  No comments.
```

¿Cómo solucionarlo?

- Una forma sería implementar *print()* en cada subclase
- Elemento no tiene método imprimir.
- El método imprimir está en la subclase
- Pero las subclases no tienen acceso a los campos *private*
  - elemento.print() el compilador dará error
- Y *Database* no podría encontrar el método *print()* en *Elemento*

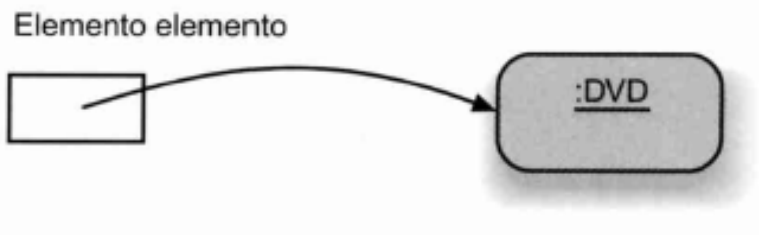


Impresión, versión 2:
el método `imprimir`
en las subclases

9.2 Tipos estáticos y dinámicos

- 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.

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



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

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

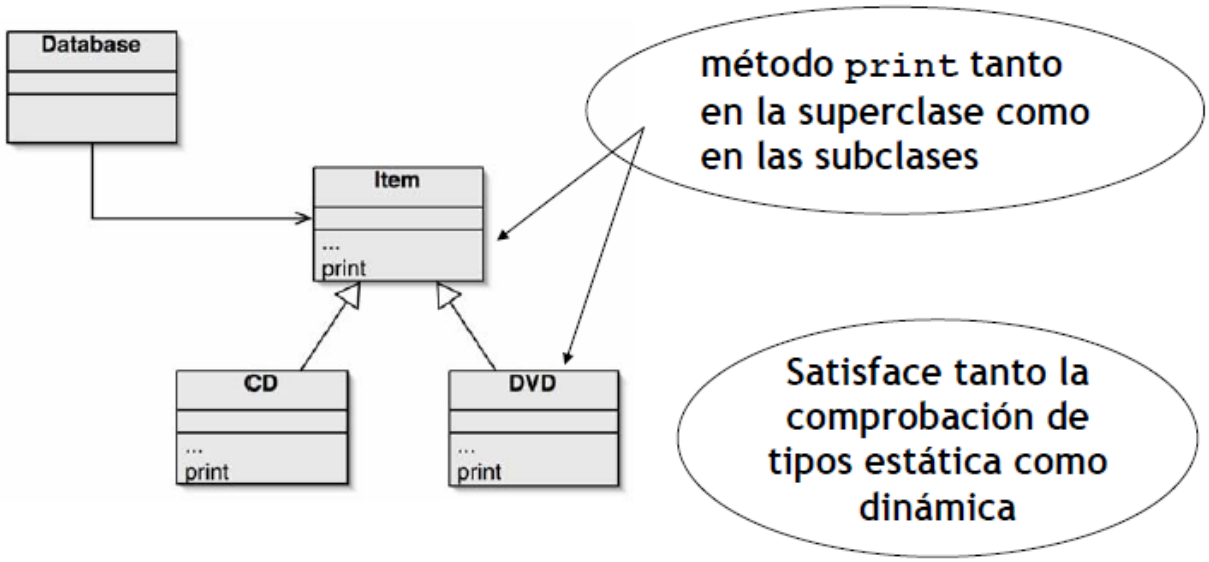
- El compilador comprueba si se producen violaciones de tipos estáticos
  - El compilador no conoce “el tipo dinámico”, porque se asigna en tiempo de ejecución
- Para que funcione, la clase Elemento debe tener un método “imprimir”

```
for(Item item : items) {
        item.print(); // Error en compilación
}
```

9.3 Sobreescribir (overriding) sustitución de métodos

La técnica que usamos acá se denomina *sobrescritura* (algunas veces también se hace referencia a esta técnica como *redefinición*). 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.

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.



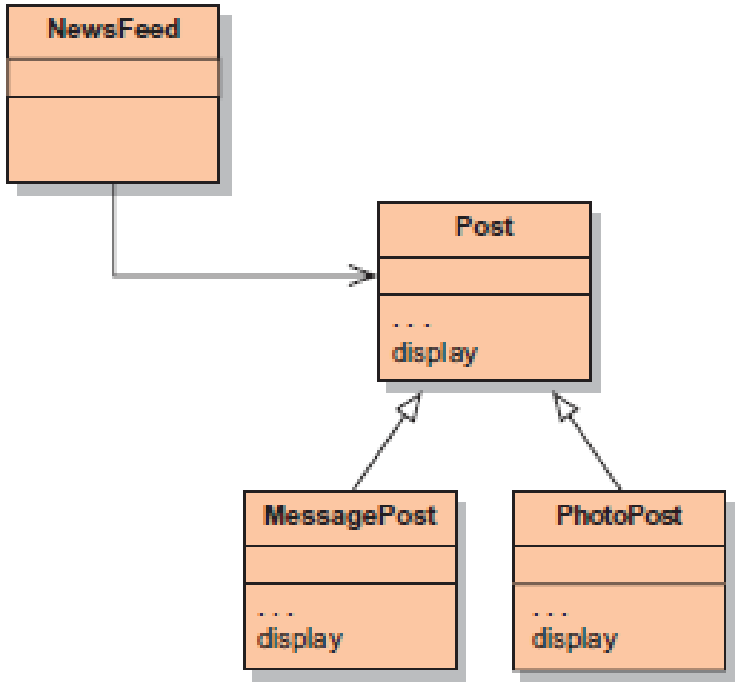
Sobreescritura

- La superclase y las subclases definen métodos con la misma signatura
  - Mismo nombre y tipo de parámetros y valor de retorno
- Cada uno tiene acceso a los campos de su clase
- La superclase satisface la comprobación de tipos estática
- El método de la subclase se llama durante la ejecución: sobreescribe la versión de la superclase

```
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);
   }
}
```

El Código 9.1 muestra los detalles relevantes del código de las tres clases. 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.

Display, version 3: `display` method in subclasses and superclass



```
public class Post
{
    ...
    public void display()
    {
        System.out.println(username);
        System.out.print(timeString(timestamp));
```

```
        if(likes > 0) {
            System.out.println("  -  " + likes + " people like this.");
        }
        else {
            System.out.println();
        }

        if(comments.isEmpty()) {
            System.out.println("   No comments.");
        }
        else {
            System.out.println("   " + comments.size() +
                               " comment(s). Click here to view.");
        }
    }
}
```

```
public class MessagePost extends Post
{
    ...
    public void display()
    {
        System.out.println(message);
    }
}
```

```
public class PhotoPost extends Post
{
    ...
    public void display()
    {
        System.out.println("  [" + filename + "]");
        System.out.println("  " + caption);
    }
}
```

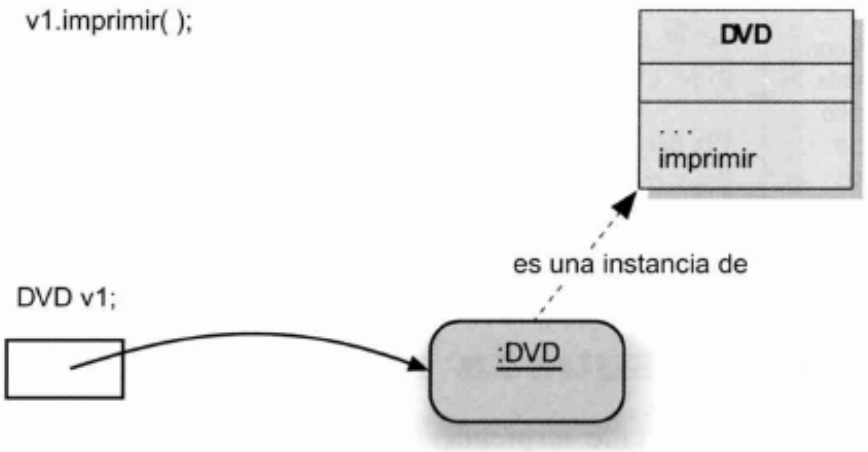
9.4 Búsqueda dinámica de métodos

- 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.

```
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.

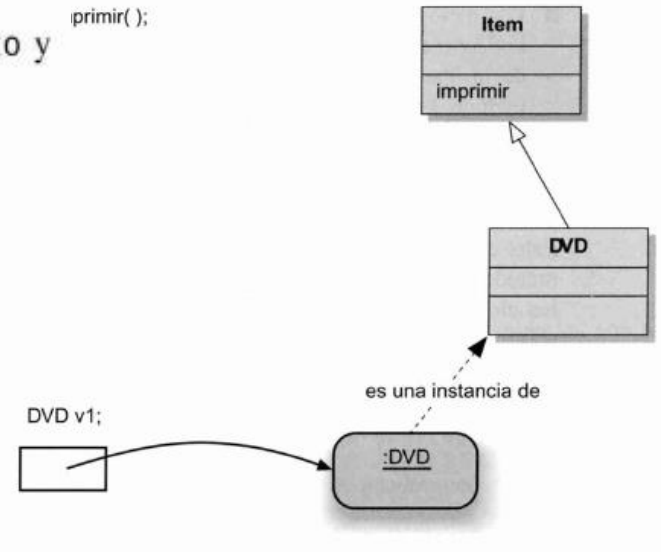


Si no se aplica ni herencia ni polimorfismo
Se seleccionará el método obvio

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.

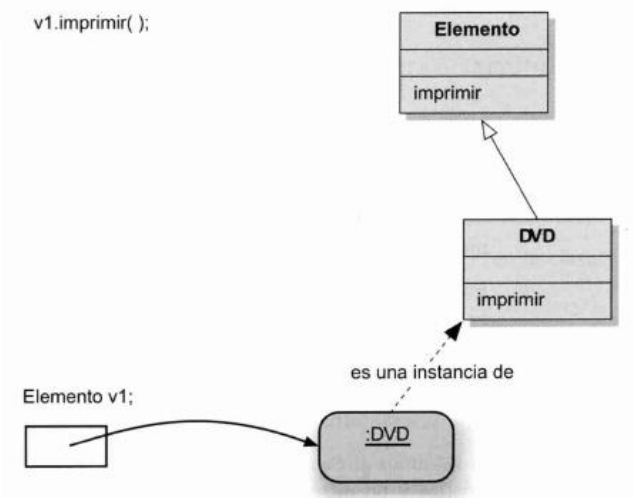
Herencia sin sobreescritura.
Se va subiendo por la jerarquía de herencia hasta encontrar el método llamado



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.

Polimorfismo y sobreescritura.
Se utiliza la primera versión que se encuentre en la jerarquía



Búsqueda dinámica del método

- En resumen:
  - Se accede una variable
  - Se encuentra el objeto referenciado por la variable
  - Se encuentra la clase del objeto
  - Se busca el método correspondiente en la clase
  - Si no se encuentra, se busca en la superclase
  - Esto se repite hasta encontrar el método correspondiente
    - Si se llega al final de la jerarquía sin encontrarlo, se producirá una excepción

9.5 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.

9.6 Polimorfismo de métodos

```
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 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.

9.7 Método object: toString

El propósito del método `toString` es crear una cadena de representación de un objeto. Esto es útil para cualquier objeto que pueda ser representado textualmente en la interfaz de usuario pero también es de ayuda para todos los otros objetos; por ejemplo, los objetos pueden ser fácilmente impresos con fines de depuración de un programa.

Cada objeto en Java tiene un método **`toString`** que puede usarse para devolver un `String` de su representación. Típicamente, para que resulte útil, un objeto debe sobrescribir este método.

- el método *toString() d*evuelve una cadena que representa el objeto por defecto como:
- nombreClase@hashcode
- Al sobreescribirlo se puede hacer que devuelva una cadena formada a partir de valores de atributos del objeto

9.7 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"

http://docs.oracle.com/javase/7/docs/api/

```
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.

9.9 Acceso “protegido”

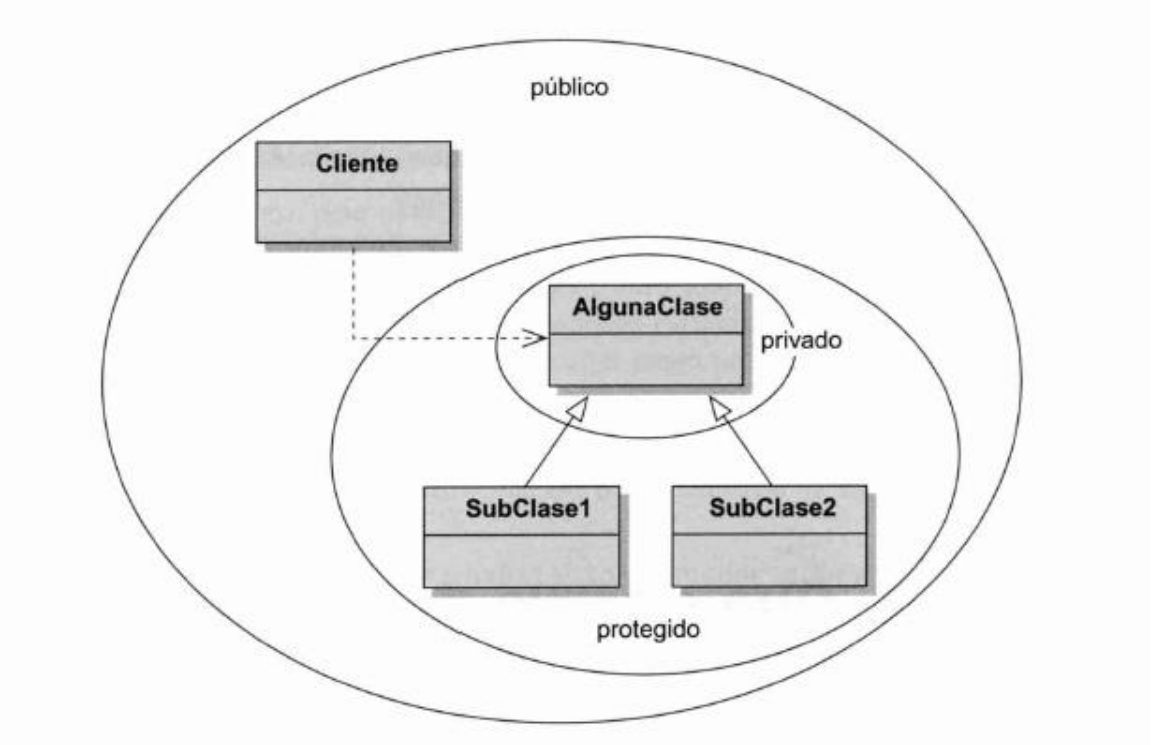
- El acceso *private* en la superclase puede ser demasiado restrictivo para una subclase
- Para permitir a las subclases el acceso a campos y métodos de la superclase se utiliza *protected*
  - Es más restrictivo que el acceso *public*
- No obstante, en general sigue siendo recomendable el acceso *private*
  - Y usar métodos de lectura (get) y modificación (set)

9.9 Acceso “protegido”

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

```
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.



Control de acceso a miembros de una clase

- private
  - Acceso sólo dentro de la clase
- public
  - Acceso desde cualquier lugar
- protected
  - Acceso en las subclases (en cualquier paquete) y desde las clases del propio paquete
- package
  - (por defecto, no se pone nada).

Resumen de niveles de acceso

VISIBILIDAD	public	protected	nada	private
Propia clase	SI	SI	SI	SI
Mismo paquete	SI	SI	SI	NO
Otro paquete	SI	NO	NO	NO
Subclase en paquete	SI	SI	SI	NO
Subclase en otro paquete	SI	SI	NO	NO

9.10 el operador “instanceof”

- Para averiguar el tipo dinámico de un objeto

```
if(currentRoom.getName().equals("Transporter room")) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

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).