Estructura de las Aplicaciones Orientadas a Objetos

Herencia y Polimorfismo

Programación Orientada a Objetos
Facultad de Informática

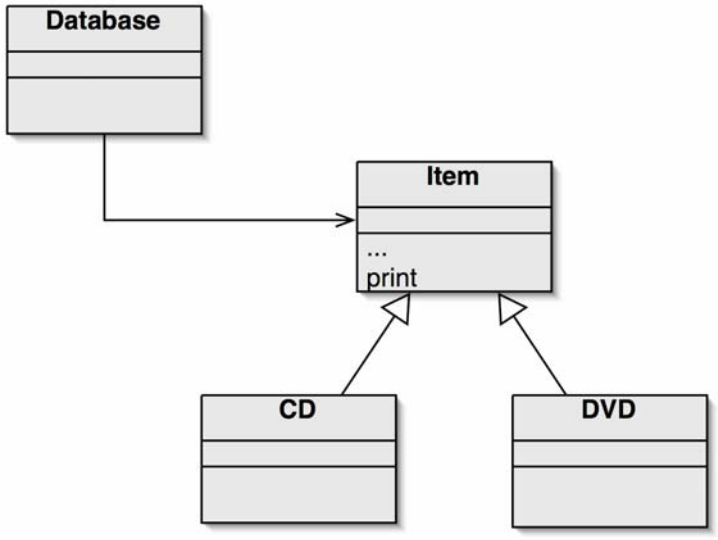
Juan Pavón Mestras
Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense Madrid

Basado en el curso Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling

Conceptos

- Polimorfismo de métodos
- Tipos estáticos y dinámicos
- Sobreescritura
- Método de búsqueda dinámica
- Acceso *protected*

Recordemos la jerarquía del ejemplo DoME



Salida conflictiva

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

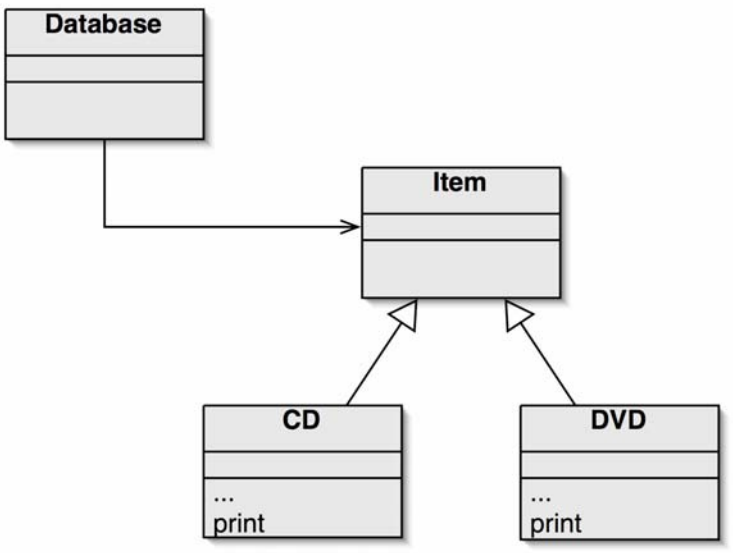
```
title: A Swingin' Affair (64 mins)*
       my favourite Sinatra album

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

¿Cuál es el problema?

- El método `print` en la clase `Item` 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

¿Cómo solucionarlo?

- Una forma sería implementar *print()* en cada subclase
  - Pero las subclases no tienen acceso a los campos *private*
  - Y *Database* no podría encontrar el método *print()* en *Item*



Tipos estáticos y dinámicos

¿De qué tipo es c1?

```
Car c1 = new Car();
```

¿De qué tipo es v1?

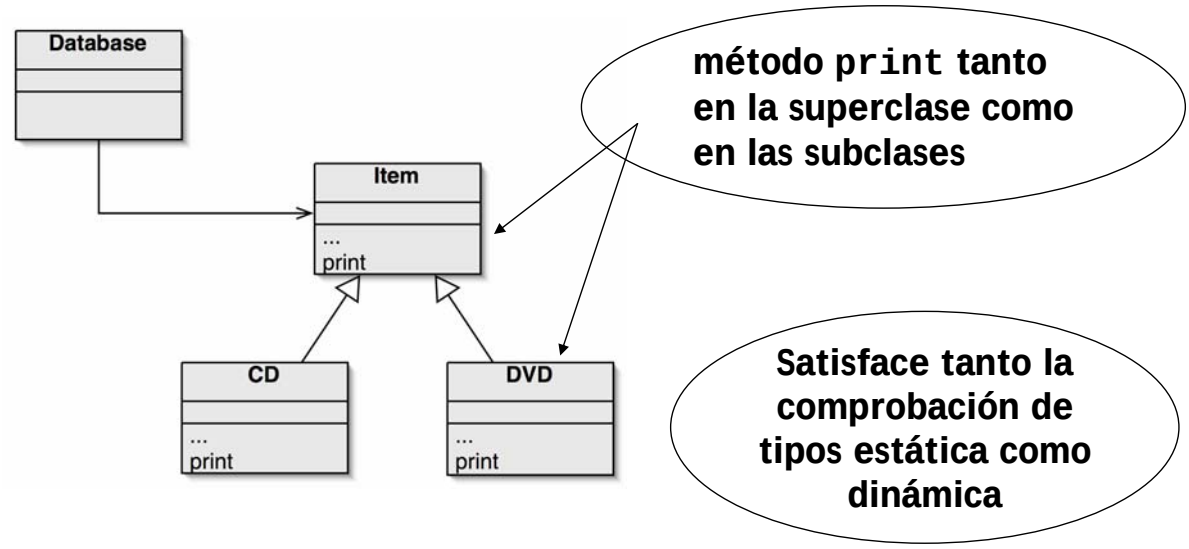
```
Vehicle v1 = new Car();
```


Tipos estáticos y dinámicos

- El tipo declarado de una variable es su tipo estático
- El tipo del objeto al que se refiere una variable es su tipo dinámico
- El compilador comprueba si se producen violaciones de tipos estáticos

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

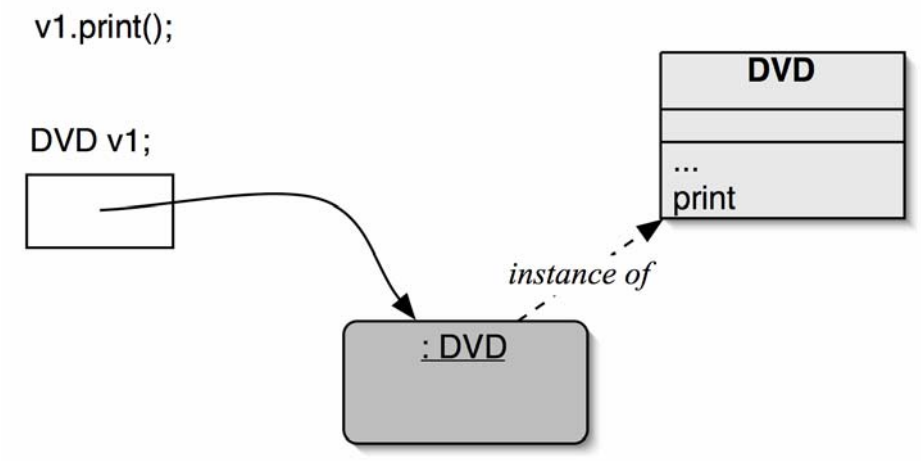
La solución: sobreescritura (*overriding*)



Sobreescritura

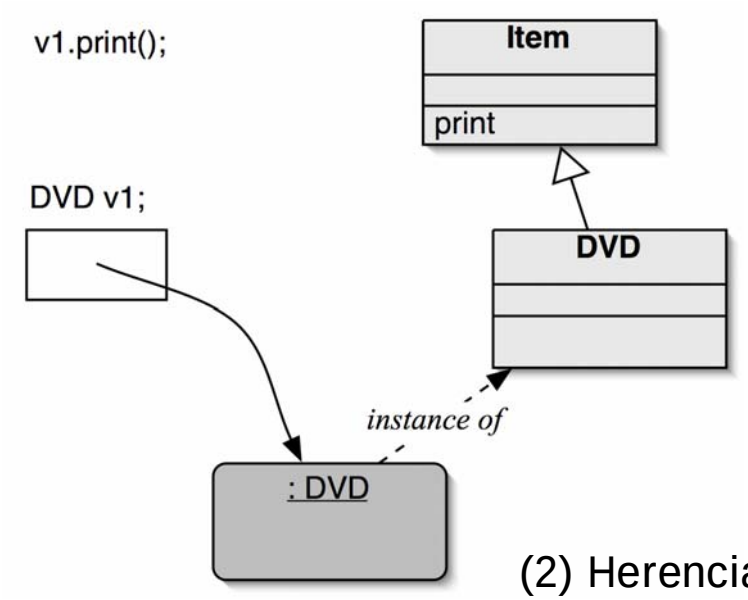
- La superclase y las subclases definen métodos con la misma signatura
  - Mismo nombre y tipo de parámetros y valor de retorno
- Cada uno tiene acceso a los campos de su clase
- La superclase satisface la comprobación de tipos estática
- El método de la subclase se llama durante la ejecución: sobreescribe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

Búsqueda dinámica del método



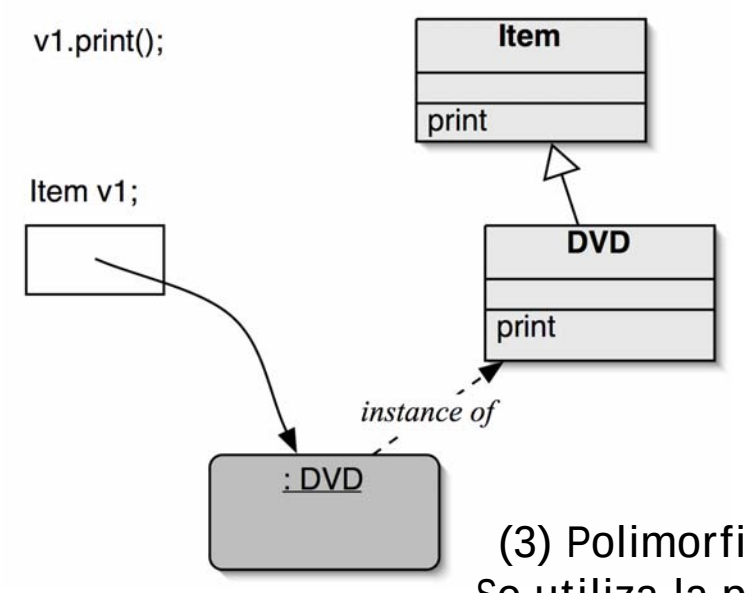
(1) Si no se aplica ni herencia ni polimorfismo
Se seleccionará el método obvio

Búsqueda dinámica del método



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

Búsqueda dinámica del método



(3) 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

Llamada a métodos sobreescritos

- Los métodos sobreescritos quedan ocultos…

  … pero podría ser útil llamarlos
- Se puede llamar al método sobreescrito (de la superclase) con `super`:
  - `super.method(...)`
  - Compárese con el uso de `super` en los constructores

```
public class CD
{
    ...
    public void print()
    {
        super.print();
        System.out.println("    " + artist);
        System.out.println("    tracks: " +
                           numberOfTracks);
    }
    ...
}
```


Métodos polimórficos

- Una variable polimórfica puede referirse a una variedad de tipos de objetos
- Las llamadas a métodos en Java son polimórficas
  - El método llamado depende del tipo de objeto dinámico
  - Según el procedimiento que se ha visto

Los métodos de la clase `java.lang.Object`

- Los métodos de la clase *Object* son heredados por todas las clases
  - Luego pueden ser sobreescritos
  - Como ejemplo, véase el método *toString()*
    - Devuelve una cadena que representa el objeto
    - Por defecto:

      `nombreClase@hashcode`

      getClass().getName() + '@' + Integer.toHexString(hashCode())

    - Al sobreescribirlo se puede hacer que devuelva una cadena formada a partir de valores de atributos del objeto

Reescribiendo el método *toString()*

```
public class Item
{
    ...

    public String toString()
    {
        String line1 = title +
                       " (" + playingTime + " mins)");
        if(gotIt) {
            return line1 + "*\n" + "    " +
                   comment + "\n");
        } else {
            return line1 + "\n" + "    " +
                   comment + "\n");
        }
    }
    ...
}
```

Reescribiendo el método *toString()*

- Mejor redefinir toString() que definir métodos `print` para una clase, y cuando se quiera imprimir:

```
System.out.println(item.toString());
```

- Las llamadas a `print/println` indicando un objeto implican la llamada al método `toString`:

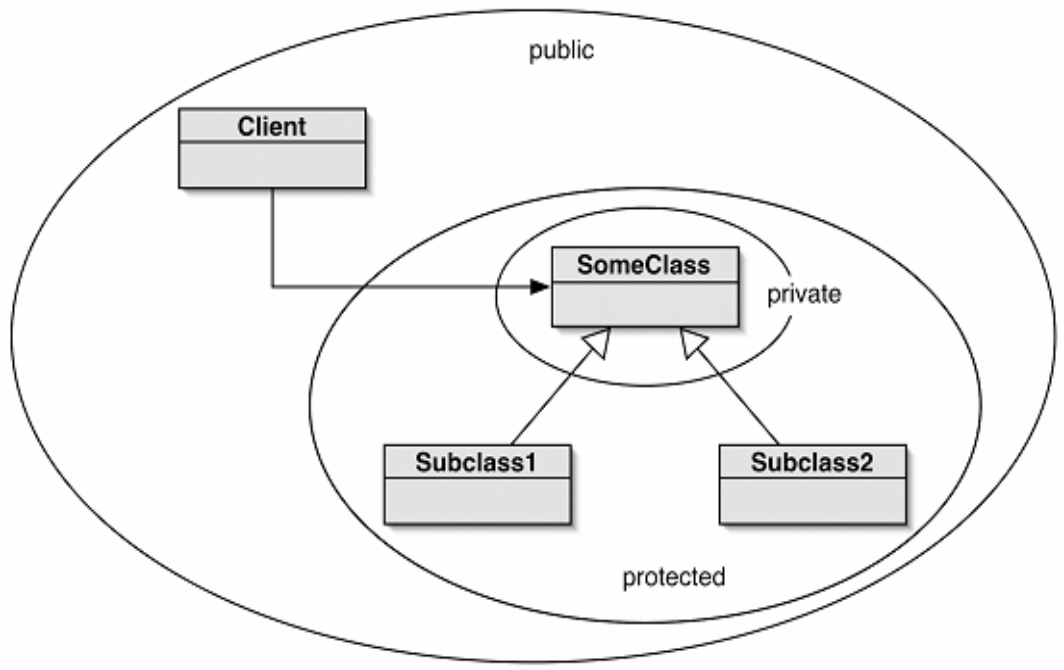
```
System.out.println(item);
```

  - ¿Cómo estará declarado el método *print()* en la clase *PrintWriter*?

Acceso protegido (*protected*)

- 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)

Niveles de acceso



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
  - NOTA: Si la subclase está declarada en otro paquete, entonces sólo puede acceder al miembro protected si se trata de una variable del tipo de la propia subclase
- Si no se indica nada, entonces la visibilidad es dentro del paquete

Control de acceso a miembros de una clase

- El acceso se controla a nivel de clase, no de objeto
  - Por ejemplo:

```
class Valor {
  private int v;
  boolean esIgual(Valor otroValor) {
            if (this.v == otroValor.v) return true;
            return false;
  }
}
```


Control de acceso a miembros de una clase

- Ejemplo de uso de protected

```
package EjemploProtected;
public class Valor {
    protected int v;
}

package EjemploProtected;
class Otra {
    void metodo (Valor unValor) { // legal
      unValor.v = 0;                  // legal
    }
}

package OtroPaquete;
import EjemploProtected.Valor;
class Ilegal extends Valor{
    void metodo(Valor uno, Ilegal dos) {
      uno.v = 1;          // ilegal !!!        (v de Valor, en otro paquete)
      dos.v = 2;          // legal             (v de Ilegal, de la propia clase)
    }
}
```

Control de acceso a miembros de una clase

- Resumen de niveles de acceso

	Clase	*Subclase*	*Paquete*	*Resto*
private	X			
protected	X	X(*)	X	
public	X	X	X	X
paquete	X		X	

- Cuando utilizar…
  - código private se puede cambiar sin afectar el código cliente
  - cambios en cosas public pueden afectar al código cliente
  - protected significa que hay control sobre el código a tocar en caso de cambios

Resumen

- El tipo declarado de una variable es su *tipo estático*
  - El compilador comprueba los tipos estáticos
- El tipo de un objeto es su *tipo dinámico*
  - Los tipos dinámicos se usan en tiempo de ejecución
- Los métodos se pueden sobreescribir en las subclases
- La búsqueda de un método empieza en su tipo dinámico
- El acceso *protected* se utiliza para la herencia