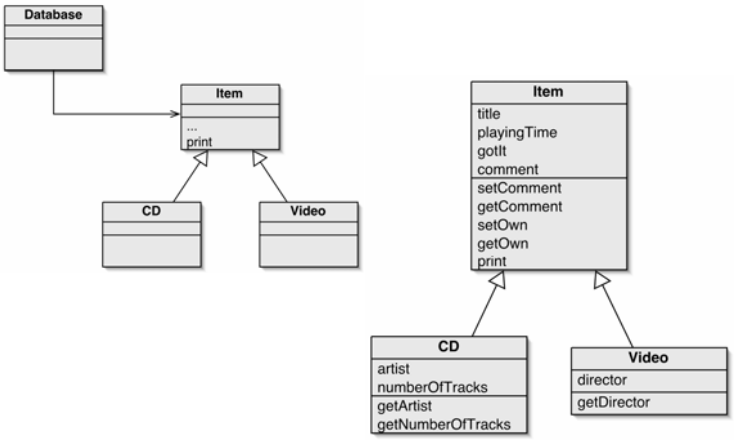
Herança e Polimorfismo
- Parte II -

Mário Meireles Teixeira
mario@deinf.ufma.br

A hierarquia do exemplo DoME



Saída incompleta

O que queremos

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

video: The Matrix (136 mins)
    Andy & Larry Wachowski
    must see if interested in virtual reality!
```

O que temos de fato

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

title: The Matrix (136 mins)
    must see if interested in virtual reality!
```

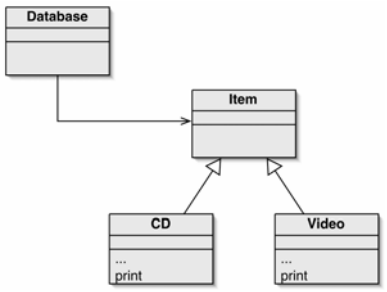
3

O problema

- O método **`print`**, implementado em **`Item`**, imprime apenas os campos comuns às classes
- A herança é uma via de mão única:
  - uma subclasse herda os campos da super-classe; porém
  - a superclasse nada sabe sobre os campos de suas subclasses

4

Tentando resolver o problema



- Coloque **`print`** onde ele tem acesso às informações de que precisa
- Cada subclasse tem agora sua própria versão de **`print`**
- Mas os campos de **`Item`** são privados, portanto não acessíveis às subclasses
- **`Database`** não consegue encontrar um método **`print`** em **`Item`**, pois a superclasse não enxerga os métodos das subclasses

5

Tipo estático e dinâmico (1)

- Uma hierarquia de tipos mais complexa requer mais conceitos para descrevê-la
- Alguns novos termos:
  - tipo estático;
  - tipo dinâmico; e
  - encaminhamento e pesquisa de método (method dispatch/lookup)

6

Tipo estático e dinâmico (2)

Qual é o tipo de c1?

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

Qual é o tipo de v1?

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

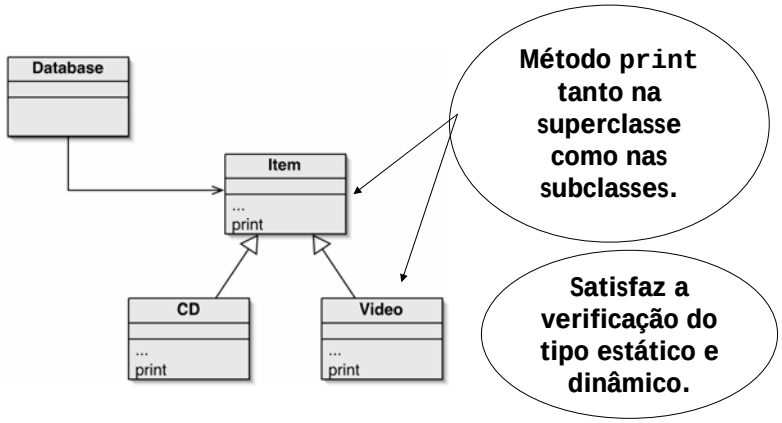

Tipo estático e dinâmico (3)

- O tipo declarado de uma variável (tempo de compilação) é seu *tipo estático*
- O tipo de objeto que uma variável referencia (tempo de execução) é seu *tipo dinâmico*
- O trabalho do compilador é verificar violações nos tipos estáticos

```
Item item = (Item) iter.next();
item.print(); // Erro em tempo de compilação
```

- Durante a execução do programa, **`Item`** deverá referenciar **`CD`** ou **`Video`**, mas o compilador não tem como saber disso

Sobrescrever: a solução



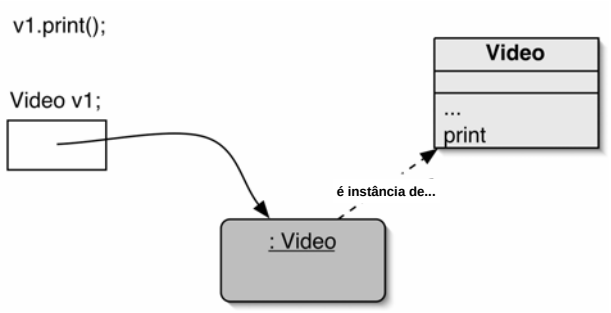
9

Sobrescrição

- Superclasse e subclasse definem métodos com a mesma assinatura
- Cada uma das versões do método tem acesso aos campos da sua classe
- O método na superclasse satisfaz a verificação do tipo estático
- O método da subclasse é chamado em tempo de execução — ele *sobrescreve* (redefine) a versão da superclasse e tem precedência de execução sobre ela

10

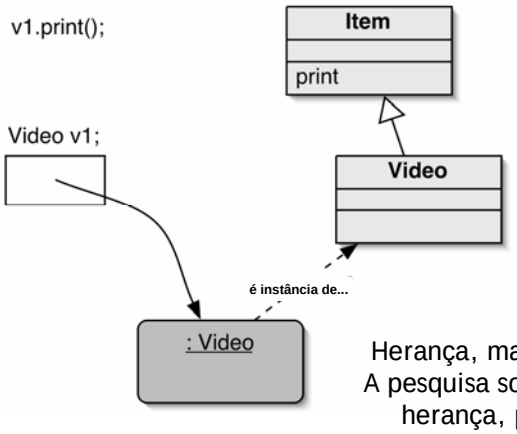
Pesquisa de método (1)



Nenhuma herança ou polimorfismo.
O método óbvio é selecionado

11

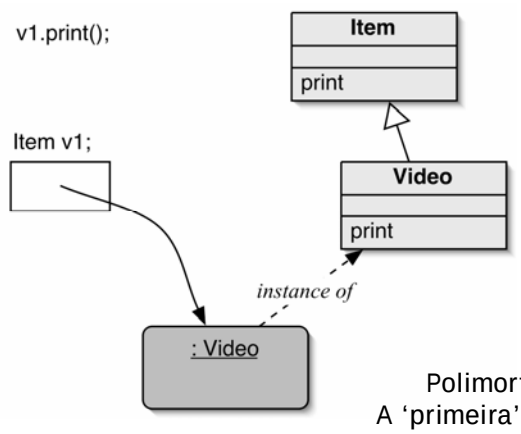
Pesquisa de método (2)



Herança, mas sem sobrescrição.
A pesquisa sobe na hierarquia da herança, procurando por uma correspondência de método

12

Pesquisa de método (3)



Polimorfismo e sobrescrição.
A 'primeira' versão encontrada é utilizada

Resumo da pesquisa de método

- A variável v1 é acessada
- O objeto armazenado na variável é encontrado
- A classe do objeto é encontrada
- É pesquisada na classe uma correspondência de método (mesma assinatura)
- Se nenhuma correspondência for encontrada, a superclasse é pesquisada
- Isso é repetido até que uma correspondência seja encontrada ou a hierarquia da classe seja exaurida
- Métodos sobrescritos têm precedência sobre os métodos das superclasses

Chamadas super em métodos

- Métodos sobrescritos são ocultados...
- ... mas, freqüentemente, queremos ser capazes de chamá-los
- Um método sobrescrito *pode* ser chamado a partir do método que o sobrescreve
  - `super.method(...)`
  - Ao contrário da utilização de **`super`** nos construtores, o método da superclasse tem que ser chamado explicitamente

Chamando um método sobrescrito

```
public class Item
{
   public void print() {
       System.out.println(title + " (" + playingTime +
          " mins)");
       System.out.println("    " + comment);
   }
}
```

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

Polimorfismo de método

- Discutimos o que é conhecido como *polimorfismo de método*
- Como vimos, uma variável polimórfica pode armazenar objetos de diferentes tipos
- Em Java, chamadas de método também são polimórficas:
  - Uma mesma chamada de método pode, em momentos diferentes, invocar diferentes métodos
  - O método real chamado depende do tipo dinâmico da variável (objeto para o qual ela aponta)

17

Os métodos da classe `Object`

- Métodos em **`Object`** são herdados por todas as classes
- Qualquer um desses pode ser sobrescrito.
- O método **`toString`** é comumente sobrescrito:
  - `public String toString()`
  - Retorna uma representação de string do objeto

18

Sobrescrevendo toString (1)

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

```
public class CD extends Item
{
   public String toString()
    {
       return super.toString() + "    " + artist +
                "\n   tracks: " + numberOfTracks + "\n";

    }
}
```


Sobrescrevendo toString (2)

- Freqüentemente, métodos **`print`** explícitos podem ser omitidos de uma classe:
  - `System.out.println(item.toString());`
- Chamadas a **`println`** com apenas um objeto fazem com que **`toString`** seja chamado automaticamente:
  - `System.out.println(item);`

Sobrescrevendo toString (3)

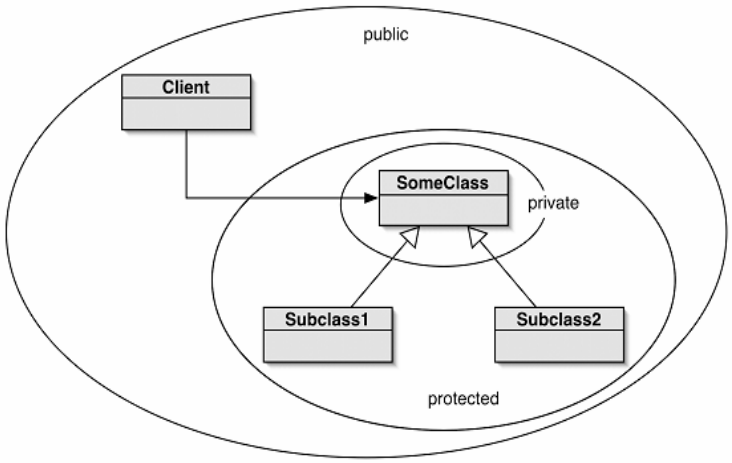
```
public class Database
{
  public void list()
    {
        for(Iterator iter = items.iterator(); iter.hasNext(); )
        {
            System.out.println(iter.next());
        }
    }
}
```

- A chamada a **`iter.next()`** retorna o tipo estático **`Object`**. O tipo dinâmico é **`CD`** ou **`Video`**
- Como **`Object`** está sendo impresso e não é uma String, o método **`toString`** é invocado
- Isso é válido porque **`Object`** declara um método **`toString --`** verificação de tipos estática
- A saída aparece corretamente, pois os tipos dinâmicos possíveis (**`CD`** ou **`Video`**) sobrescrevem **`toString`**

Acesso protegido

- Acesso privado na superclasse pode ser bem restritivo para uma subclasse. O relacionamento mais próximo da herança é suportado pelo acesso *protegido* (**`protected`**)
- O acesso protegido é mais restritivo que o acesso público
- O acesso protegido permite acesso aos campos e métodos dentro da própria classe e todas as suas subclasses
- É recomendável manter os campos como **`private`**:
  - Melhor definir métodos de acesso e métodos modificadores protegidos

Níveis de acesso



23