Material complementar sobre Herança[1]

Este material é complementar aos vídeos "Herança" e "Herança e Polimorfismo", e à Lista 8. É preciso visitar estes conteúdos para reconhecer os conceitos e exemplos usados neste material.

1 Herança e direito de acesso

Considere o diagrama de classes da Figura 1 definida no projeto da Lista 8.

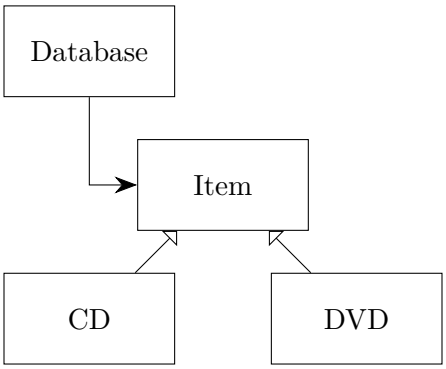

Figura 1: Diagrama de classes do projeto de itens multimídia.

Para objetos de outras classes, objetos `CD` e `DVD` aparecem tal como qualquer outro tipo de objeto. Como consequência, membros definidos como `public` nas partes superclasse ou subclasse estarão acessíveis aos objetos de outras classes, mas para membros definidos como `private` estarão inacessíveis. De fato, a regra sobre privacidade também se aplica entre uma subclasse e sua superclasse: uma subclasse não pode acessar membros privados de sua superclasse. Se um método de subclasse precisasse acessar ou alterar atributos privados em sua superclasse, a superclasse precisaria então fornecer métodos de acesso e/ou mutador (getters e setters) apropriados. Um objeto de uma subclasse pode chamar qualquer método público definido na sua superclasse como se ele fosse definido localmente em subclasse - nenhuma variável é necessária porque todos os métodos são parte do mesmo objeto.

2 Subclasses, subtipos e atribuição

As classes definem tipos. O tipo de um objeto que foi criado a partir da classe `DVD` é DVD. Os tipos definidos pelas classes podem ter subtipos. No exemplo da Figura 1, o tipo `DVD` é um subtipo de `Item`.

[1]Textos extraídos do livro "Programação orientada a objetos com Java" de Michael Kolling e David J. Barnes, capítulos 8 e 9.

Quando queremos atribuir um objeto a uma variável, o tipo do objeto deve corresponder ao tipo da variável. Por exemplo,

```
Carro meuCarro = new Carro();
```

é uma atribuição válida porque um objeto do tipo `Carro` é atribuído a uma variável declarada para armazenar objetos desse tipo. Agora que conhecemos a herança devemos declarar a regra da criação de tipos de maneira mais completa: uma variável pode conter objetos de seu tipo declarado ou de qualquer subtipo de seu tipo declarado.

Imagine que temos uma classe `Veiculo` com duas subclasses `Carro` e `Bicicleta` (Figura 2).

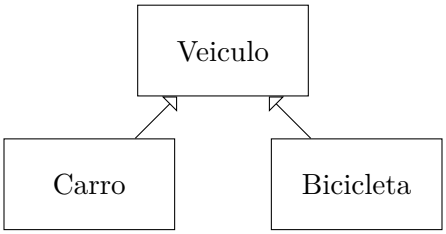

Figura 2: Diagrama de classes.

Nesse caso, a regra de criação de tipos admite que as seguintes atribuições são válidas:

```
Veiculo v1 = new Veiculo();
Veiculo v2 = new Carro();
Veiculo v3 = new Bicicleta();
```

O tipo de variável declara o que se pode armazenar. Declarar uma variável de tipo `Veiculo` expressa que essa variável pode armazenar veículos. Mas como um carro é um veículo, é perfeitamente válido armazenar um carro em uma variável que é concebida para veículos.

Esse princípio é conhecido como *substituição*. Nas linguagens orientadas a objetos podemos substituir um objeto de subclasse em que um objeto da superclasse é esperado, porque o objeto da subclasse é um caso especial da superclasse. Entretanto, fazer isso no sentido inverso não é permitido:

```
Carro c1 = new Veiculo(); //isso é um erro!
```

Essa instrução tenta armazenar um objeto `Veiculo` em uma variável `Carro`. O Java não permite isso e informará um erro se tentar compilar essa instrução. A variável é declarada para ser capaz de armazenar carros. Um veículo, por outro lado, pode ou não ser um carro - não sabemos. Portanto, a instrução pode estar errada e não será permitida.

2.1 Subtipagem e passagem de parâmetro

A passagem de parâmetro comporta-se exatamente da mesma maneira que uma atribuição a uma variável. Essa é a razão por que podemos passar um objeto do tipo `DVD` para um método que tem um parâmetro do tipo `Item`. Na classe `Database` tem a seguinte definição do método `addItem`:

```
public class Database{
    public void addItem(Item novoItem){
        ...
    }
}
```

É possível utilizar esse método para adicionar `DVD` e `CD` ao banco de dados:

```
Database db = new Database();
DVD dvd = new DVD(...);
CD cd = new CD(...);

db.addItem(dvd);
db.addItem(cd);
```

Por causa da regra da subtipagem, só é necessário um método (com um parâmetro do tipo Item) para adicionar objetos `DVD` e objetos `CD`.

2.2 Tipo estático e tipo dinâmico

Considere a seguinte instrução:

```
Carro c1 = new Carro();
```

Dizemos que o tipo de `c1` é `Carro`. Antes de trabalhar com herança, não havia a necessidade de fazer uma distinção, seja entre 'tipo de `c1`' significando 'o tipo da variável `c1`' e o 'o tipo do objeto armazenado em `c1`'. Não importava, porque o tipo da variável e o tipo do objeto sempre foram os mesmos.

Agora que conhecemos a subtipagem, teremos de ser mais precisos. Considere a seguinte instrução:

```
Veiculo v1 = new Carro();
```

Qual é o tipo de `v1`? Isso depende daquilo que precisamente queremos dizer por 'tipo de `v1`'. O tipo da variável `v1` é `Veiculo`; o tipo do objeto armazenado em `v1` é `Carro`. Por meio da subtipagem e das regras de substituição, agora temos situações em que o tipo da variável e o tipo do objeto armazenado não são exatamente os mesmos.

- Denominamos o tipo declarado da variável de *tipo estático*, porque ele é declarado no código-fonte - a representação estática do programa.

- Denominamos o tipo do objeto armazenado em uma variável de *tipo dinâmico*, porque ele depende de atribuições em tempo de execução - o comportamento dinâmico do programa.

Portanto, examinando a instrução anterior, é possível afirmar mais precisamente: o tipo estático de `v1` é `Veiculo`, o tipo dinâmico de `v1` é `Carro`.

3 A classe `Object`

Todas as classes tem uma superclasse. Embora seja possível declarar uma superclasse explícita para uma classe, todas as classes que não tem nenhuma declaração de superclasse implicitamente herdam de uma classe chamada `Object`.

`Object` é uma classe da biblioteca padrão de Java que serve como uma superclasse para todos os objetos. Escrever uma declaração de classe como:

```
public class Pessoa{
    ...
}
```

é equivalente a escrever:

```
public class Pessoa extends Object{
   ...
}
```

O compilador Java insere automaticamente a superclasse `Object` para todas as classes. Portanto, não é necessário fazer isso por conta própria. Cada classe única herda de `Object`, direta ou indiretamente.

O diagrama do projeto de itens multimídia da Lista 8 pode ser representado como na Figura 3. Mesmo que `Item` não tenha declarado explicitamente que estende alguma classe, isso é feito implicitamente pelo compilador.

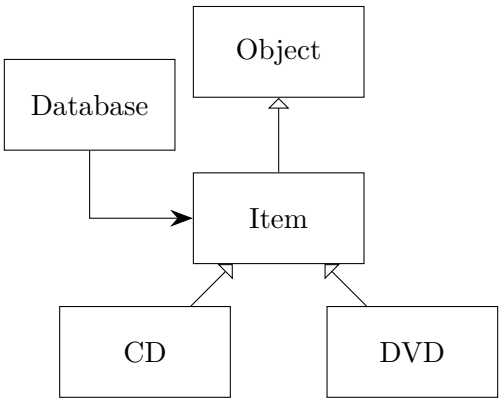

Figura 3: Diagrama de classes do projeto de itens multimídia.

3.1 Métodos de `Object`: `toString()`

A superclasse `Object` implementa alguns métodos que fazem parte de todos os objetos. O mais interessante desses métodos é o `toString()`. O propósito deste método é criar uma representação de string para um objeto. Isso é útil para quaisquer objetos que invariavelmente devem ser representados textualmente na interface com o usuário, mas também ajuda a todos os outros objetos - por exemplo, eles podem ser facilmente impressos para propósitos de depuração.

A implementação padrão de `toString()` na classe `Object` não pode fornecer uma grande quantidade de detalhes. Se, por exemplo, chamássemos `toString()` em um objeto `DVD`, receberíamos uma string sem muito sentido. O valor do retorno simplesmente mostra o nome da classe do objeto e do endereço de memória no qual o objeto está armazenado.

Para tornar esse método mais útil, em geral este método deve ser sobrescrito em cada classe, para que ele mostre as informações relevantes para este objeto. Por exemplo, na classe `Item` o método `print()` pode ser redefinido em termos de uma chamada para o seu método `toString()`. A assinatura do método `toString()` é a seguinte:

```
public String toString()
```

O método `toString()` não recebe parâmetro e retorna uma string. Na sobrescrita a assinatura não pode ser modificada, então o código-fonte fica da seguinte forma:

```
@Override
public String toString() {
    String texto = titulo + " (" + duracao + " mins)";
    if(tenho) {
        return texto + "*\n" + "    " + comentario + "\n";
    } else {
        return texto + "*\n" + "    " + comentario + "\n";
    }
}

public void print(){
    System.out.println(toString());
}
```

Com a sobrescrita ao método `toString()` qualquer cliente (por exemplo, a classe `Database`) tem liberdade de fazer o que quiser com esse texto. A instrução usada no cliente para imprimir o item agora poderia ser semelhante a:

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

Nesse sentido, os métodos `System.out.print()` e `System.out.println()` são especiais: se o parâmetro para um desses métodos não for um objeto `String`, o método então invoca automaticamente o método `toString()` do objeto. Assim, não precisamos escrever a chamada explícita e, em vez disso, poderíamos escrever:

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