Polimorfismo

Programação 2 – aulas 15 e 16

Objetivos da seção

- Apresentar o polimorfismo e sua utilidade em desacoplar classes
- Apresentar a implementação de polimorfismo através de ligação dinâmica (*dynamic* ou *late binding*)

Um afinador de instrumentos

- Devemos escrever um afinador capaz de afinar diferentes instrumentos
- Um afinador afina realizando uma série de operações, dentre elas, ele toca o instrumento
- Um possível código segue

```
package p2.exemplos;

public abstract class Instrumento {

   public enum Nota {
```

```
        C, D, E, F, G, A, B; //muitas outras notas aqui!
    }

    public abstract void toca(Nota n);

    //muitas outras coisas!!!!
}
```

```
package p2.exemplos;

public class Violao extends Instrumento {

    @Override
    public void toca(Nota n) {
        System.out.println("Violao.toca() " + n);
    }

    //muitas outras coisas!!!

}
```

```
package p2.exemplos;
```

```java
public class Sax extends Instrumento {

    @Override
    public void toca(Nota n) {
        System.out.println("Sax.toca() " + n);
    }

    //muitas outras coisas!!!

}
```

```java
package p2.exemplos;

public class Flauta extends Instrumento {

    @Override
    public void toca(Nota n) {
        System.out.println("Flauta.toca() " + n);
    }

    //muitas outras coisas!!!
```

```
}
```

```
package p2.exemplos;

public class Baixo extends Instrumento {

    @Override
    public void toca(Nota n) {
        System.out.println("Baixo.toca() " + n);
    }

    //muitas outras coisas!!!
}
```

```
package p2.exemplos;

import java.util.Scanner;

import p2.exemplos.Instrumento.Nota;

public class AfinadorTipado {

    public static void afina(Flauta f) {
```

```
        //...
        f.toca(Nota.C);
        //...
    }

    public static void afina(Sax s) {
        //...
        s.toca(Nota.C);
        //...
    }

    public static void afina(Violao v) {
        //...
        v.toca(Nota.C);
        //...
    }

    public static void afina(Baixo b) {
        //...
        b.toca(Nota.C);
        //...
    }

    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        int escolha = 4;
        while (escolha <= 4 && escolha > 0) {
            prompt();
            escolha = sc.nextInt();
            switch (escolha) {
            case 1:
                Flauta flauta = new Flauta();
                afina(flauta);
                break;
            case 2:
                Sax sax = new Sax();
                afina(sax);
                break;
            case 3:
                Baixo baixo = new Baixo();
                afina(baixo);
                break;
            case 4:
                Violao violao = new Violao();
                afina(violao);
                break;
            }
        }
    }
```

```
    private static void prompt() {
        System.out.println("Que instrumento voce quer
afinar?");
        System.out.println("1. Para afinar a flauta;");
        System.out.println("2. Para afinar o sax;");
        System.out.println("3. Para afinar o baixo;");
        System.out.println("4. Para afinar o violao;");
        System.out.println(">4 ou <0 Para sair.");
    }
}
```

- Vamos rodar o AfinadorTipado?
  - Note que ele é capaz de afinar quaisquer dos tipos de instrumentos já existentes

- Qual o problema com este código?
  - Não esgotamos ainda as possibilidades de instrumentos!
    i) Novos instrumentos podem surgir
  - Toda vez que um novo instrumento é adicionado, um novo método afina deve ser escrito;

- Se você esquecer de escrever o método **`afina(NovoInstrumento i)`** para o novo instrumento, não vai ocorrer erro de compilação, até que você tente afinar o novo instrumento
- E se um outro método, como por exemplo, **`testaAfinacao (TipoDeInstrumento i)`** tiver que ser criado na classe Afinador? Assim como **`afina(TipoDeInstrumento i)`**, deve haver um "testaAfinacao" para cada tipo de instrumento
  i) E se novos tipos de instrumento forem criados, então novos métodos "testaAfinacao" também devem ser criados
  ii) Mas também tem os métodos "afina"...
- Note que gerenciar isso se torna muito complicado e sujeito a erros

- Qual a solução?
  - Apenas um método afina (e apenas um método testaAfinacao, se este existir), que recebam como argumento um objeto da super classe de todos os instrumentos (Instrumento) e não um objeto do tipo específico de instrumento
  - É como se esquecêssemos que as classes derivadas existem ao criar a classe Afinador

- Na hora de afinar mesmo ou testar a afinação de um instrumento específico e o afinador precisar tocar o instrumento, então este instrumento específico vai ser passado como argumento para os métodos afina (e testaAfinacao)
  i) Apesar do método estar esperando um Instrumento, isso não daria erro de compilação, pois trata-se de um *upcasting*. Princípio da substituição tem que ser válido.

- Como seria essa solução em Java?

```
package p2.exemplos;

import p2.exemplos.Instrumento.Nota;

public class Afinador {

   public static void afina(Instrumento i) {
        //...
        i.toca(Nota.C);
        //...
   }

   public static void main(String[] args) {
        Instrumento instrumento = null;
```

```
        Scanner sc = new Scanner(System.in);
        int escolha = 4;
        while (escolha <= 4 && escolha > 0) {
            prompt();
            escolha = sc.nextInt();
            switch (escolha) {
            case 1:
                instrumento = new Flauta();
                break;
            case 2:
                instrumento = new Sax();
                break;
            case 3:
                instrumento = new Baixo();
                break;
            case 4:
                instrumento = new Violao();
                break;
            }
            if (instrumento != null)
                afina(instrumento);
        }
    }

    private static void prompt() {
```

```
        System.out.println("Que instrumento voce quer
afinar?");
        System.out.println("1. Para afinar a flauta;");
        System.out.println("2. Para afinar o sax;");
        System.out.println("3. Para afinar o baixo;");
        System.out.println("4. Para afinar o violao;");
        System.out.println(">4 ou <0 Para sair.");
    }
}
```

- Quando olhamos para o método afina acima, está claro o que ele vai fazer?
  - A princípio, isto é, em tempo de compilação, não se sabe o que o método afina vai fazer ☹
  - O método afina espera uma referência a um instrumento, mas quando recebe uma referência a um instrumento que deriva de Instrumento, como saberá que instrumento é?
  - Como sabe que deve chamar o toca da flauta e não do violão?

- O que é complicado nesse mundo? Qual a mágica?
  - Como o compilador sabe que método chamar? Se ele vai chamar o de Flauta, ou o de Violao, ou o de outro instrumento qualquer…
  - Esta mágica ocorre em Java (e em outras linguagens) devido ao momento em que a chamada a um método é associada ao corpo (implementação) do método
    i) A esta atividade chamamos *binding*
  - Em muitas linguagens isso é feito em tempo de compilação (*early binding*), então em tempo de compilação deve-se saber exatamente que método chamar
    i) Isto não serviria para o nosso caso
    ii) Veja o exemplo acima… não está claro olhando o código se iremos chamar o método afina de Flauta, de Violao, de Sax ou de Baixo
  - A associação pode ser feita em tempo de execução (*late binding*)
    i) É assim que ocorre em java
    ii) A escolha é feita com base no tipo de objeto realmente passado como parâmetro
      (1) Esperava-se receber um Instrumento, mas de fato, passou-se um objeto que deriva de Instrumento
      (2) A associação é feita levando em consideração este objeto que foi passado em substituição à superclasse

(3) O objetos precisam de alguma forma saber responder sobre seu tipo, para que em tempo de execução o método correto seja chamado

(4) Em Java toda associação de chamada de método com corpo de método (*binding*) é feito em tempo de execução, exceto se o método for final ou static

- Uma vez que se sabe que objeto se tem na mão, então procura-se dentre os métodos definidos na classe deste objeto a implementação deste método

i) Lembre que o método em questão pode ser um método herdado e não sobrescrito, em cujo caso levará para a implementação da classe mãe mais próxima

(1) A busca vai subindo na hierarquia de classes até que o método seja encontrado em alguma superclasse, ou até que se chegue na raiz, em cujo caso ocorrerá um erro de compilação

(2) Assim, a redefinição de um método em uma subclasse esconde os métodos definidos nas superclasses das subclasses que esta classe possa vir a ter

Definindo polimorfismo

Acepções

■ substantivo masculino

1 qualidade ou estado de ser capaz de assumir diferentes formas

Obs.: p.opos. a *monomorfismo*

- Diante de tudo que vimos, o que é polimorfismo?
- A chamada `i.toca(Nota.C)` é polimórfica, pois faz coisas diferentes dependendo do objeto que a recebe
  - Se for um sax, então tocará uma nota no sax, se for um violão, tocará uma nota do violão e assim sucessivamente
- **A mesma chamada a um método faz coisas diferentes dependendo do objeto que recebe a mensagem**

Alguns detalhes adicionais...

Polimorfismo e métodos de classe

- Métodos de classe não permitem polimorfismo. Vejamos um exemplo.

```
package p2.exemplos;

/**
 * Exemplo que demonstra que métodos estáticos não são
polimórficos.<br>
 * (Exemplo derivado de "Thinking in Java")
 * @author Raquel Lopes
 *
 */
public class StaticSuper {
   public static String STATIC_STRING = "Base Static
String";

   public static String staticGet() {
      return "Base staticGet()";
   }

   public String dynamicGet() {
      return "Base dynamicGet()";
   }
}
```

```java
package p2.exemplos;

/**
 * Exemplo que demonstra que métodos estáticos não são
polimórficos.<br>
 *
 * @author Raquel Lopes
 *
 */
public class StaticDerived1 extends StaticSuper {
   public static String STATIC_STRING = "Derived1
Static String";

   public static String staticGet() {
      return "Derived1 staticGet()";
   }

   @Override
   public String dynamicGet() {
      return "Derived1 dynamicGet()";
   }
```

```
}
```

```
package p2.exemplos;

/**
 * Exemplo que demonstra que métodos estáticos não são
polimórficos.<br>
 *
 * @author Raquel Lopes
 *
 */
public class StaticDerived2 extends StaticSuper {
   public static String STATIC_STRING = "Derived2
Static String";

   public static String staticGet() {
      return "Derived2 staticGet()";
   }

   @Override
      public String dynamicGet() {
      return "Derived2 dynamicGet()";
```

```
    }
}
```

```
package p2.exemplos;

/**
 * Exemplo que demonstra que métodos estáticos não são
polimórficos.<br>
 *
 * @author Raquel Lopes
 *
 */
public class StaticPolymorphism {

    /**
     * @param args
     */
    public static void main(String[] args) {
        StaticSuper s1 = new StaticDerived1();
        StaticSuper s2 = new StaticDerived2();
        StaticDerived1 d1 = new StaticDerived1();
        StaticDerived2 d2 = new StaticDerived2();
```

```
        printStaticObject(s1);
        printStaticObject(s2);
        printStaticObject(d1);
        printStaticObject(d2);
    }

    private static void printStaticObject(StaticSuper s)
{
        System.out.println(s.staticGet());
        System.out.println(s.dynamicGet());
        System.out.println(s.STATIC_STRING);
        System.out.println("_____________________________");
    }

    private static void printStaticObject(StaticDerived1
s)  {
        System.out.println(s.staticGet());
        System.out.println(s.dynamicGet());
        System.out.println(s.STATIC_STRING);
        System.out.println("_____________________________");
    }
```

```
    private static void printStaticObject(StaticDerived2
s) {
        System.out.println(s.staticGet());
        System.out.println(s.dynamicGet());
        System.out.println(s.STATIC_STRING);
        System.out.println("_____");
    }

}
```

- A saída deste programa é como segue:

```
Base staticGet()
Derived1 dynamicGet()
Base Static String

_____
Base staticGet()
Derived2 dynamicGet()
Base Static String

_____
Derived1 staticGet()
Derived1 dynamicGet()
```

```
Derived1 Static String
_____________________
Derived2 staticGet()
Derived2 dynamicGet()
Derived2 Static String
_____________________
```

- Note que a chamada ao método static não é polimórfica. O método da classe mãe sempre é chamado.
  - Isso ocorre porque o binding de métodos static ocorre em tempo de compilação, quando você define o tipo de sua referência. Se sua referência é para um objeto StaticSuper, em tempo de compilação o compilador irá associar a chamada ao método staticGet de StaticSuper
  - Mesmo que em tempo de execução a referência aponte para um objeto de uma classe derivada de StaticSuper, isso não é levado em consideração, pois o binding dos métodos de classe já foi realizado
- Veja que você não consegue usar @Override para um método estático
  - Porque métodos estáticos não são sobrescritos
  - Se você cria um método estático com mesmo nome, como ocorreu no exemplo, então todos existirão, e será chamado o método da classe que foi indicada ao definir a referência (não ao fazer new!)

Polimorfismo e construtores

- Herdamos tudo da classe base, mas só conseguimos ter acesso ao que é public ou protected
- Para garantir que o objeto da classe base que foi automaticamente inserido como um membro da classe derivada foi corretamente inicializado, então temos que chamar o construtor da classe base para iniciá-lo
- Quando temos muitos níveis de herança, em que ordem os construtores são chamados?
- Vamos criar um programa que demonstra esta ordem?

```
package p2.exemplos;

public class Refeicao {
    public Refeicao() {
        System.out.println("Refeicao()");
    }
}
```

```
package p2.exemplos;
```

```
public class Almoco extends Refeicao {
    public Almoco() {
        System.out.println("Almoco()");
    }
}
package p2.exemplos;

public class AlmocoPortatil extends Almoco {
    public AlmocoPortatil() {
        System.out.println("AlmocoPortatil()");
    }
}

package p2.exemplos;

public class MistoQuente extends AlmocoPortatil {
    private Pao pao = new Pao();
    private Queijo queijo = new Queijo();
    private Presunto presunto = new Presunto();

    public MistoQuente() {
        System.out.println("MistoQuente()");
    }
```

```
    public static void main(String[] args) {
        new MistoQuente();
    }
}
package p2.exemplos;

public class Pao {
    public Pao() {
        System.out.println("Pao()");
    }
}
package p2.exemplos;

public class Presunto {
    public Presunto() {
        System.out.println("Presunto()");
    }
}
package p2.exemplos;

public class Queijo {
    public Queijo() {
```

```
        System.out.println("Queijo()");
    }
}
```

- O que vocês acham que acontecerá?
- Veja a saída:

```
Refeicao()
Almoco()
AlmocoPortatil()
Pao()
Queijo()
Presunto()
MistoQuente()
```

- O que aprendemos?
  - › No construtor do objeto sendo criado, há uma chamada (que pode ser escondida) ao construtor da super classe
    - i) E isso ocorre recursivamente até que chegue na raiz e o objeto da classe mãe raiz seja criado
    - ii) Então os objetos de todas as classes base são criados antes que o corpo do construtor seja executado

- Antes de executar o corpo do construtor os objetos inicializados na declaração dos atributos são criados
- O corpo do construtor da classe derivada é executado

- Agora vamos complicar um pouco mais
- O que acontece se, dentro de um construtor, chamarmos um método polimórfico (não **`static`** e não **`final`**) do objeto que está sendo "construído"?
  - Isso pode levar a bugs muito difíceis de serem identificados
  - Vamos usar o mesmo exemplo anterior, mas agora o nosso misto tem um construtor

```
package p2.exemplos;

public class Almoco extends Refeicao {
   public Almoco() {
      System.out.println("Almoco() antes do preparo");
      prepara();
      System.out.println("Almoco() depois do preparo");
   }

   public void prepara() {
```

```
        System.out.println("Prepara Almoco");
    }
}
```

```
package p2.exemplos;

public class AlmocoPortatil extends Almoco {
    public AlmocoPortatil() {
        System.out.println("AlmocoPortatil() antes do
preparo");
        prepara();
        System.out.println("AlmocoPortatil() depois do
preparo");
    }

    public void prepara() {
        System.out.println("Prepara AlmocoPortatil");
    }
}
```

```
package p2.exemplos;
```

```
public class MistoQuente extends AlmocoPortatil {
    private Pao pao = new Pao();
    private Queijo queijo = new Queijo();
    private Presunto presunto = new Presunto();

    public MistoQuente() {
        System.out.println("MistoQuente() antes do
preparo");
        prepara();
        System.out.println("MistoQuente() depois do
preparo");
    }

    public void prepara() {
        System.out.println("Prepara MistoQuente");
    }

    public static void main(String[] args) {
        new MistoQuente();
    }
}
```

- A saída deste programa é:

```
Refeicao()
Almoco() antes do preparo
Prepara MistoQuente
Almoco() depois do preparo
AlmocoPortatil() antes do preparo
Prepara MistoQuente
AlmocoPortatil() depois do preparo
Pao()
Queijo()
Presunto()
MistoQuente() antes do preparo
Prepara MistoQuente
MistoQuente() depois do preparo
```

- Note que o método prepara da subclasse esconde os métodos prepara das super-classes, mesmo ao chamar o seus construtores (das super classes)
  - Por isso, muito cuidado!
  - Não é recomendado chamar métodos que não sejam privados dentro do construtor!!!
  - Métodos privados são implicitamente final e associados a uma implementação em tempo de compilação

Projetando com herança/polimorfismo e composição

- Na dúvida? Escolha a composição
- Lembre-se que a herança estabelece uma relação “é um tipo especial de” que gera um acoplamento forte entre as classes
  - Mudança na implementação da classe mãe pode requerer mudança na implementação da classe derivada e da classe que usa a classe derivada (e que talvez nem saiba que a classe mãe existe!)
  - Podemos reduzir um pouco o acoplamento garantindo que todos os atributos da classe mãe são privados e objetos da classe derivada não terão acesso a eles diretamente
- Quando usamos composição, a implementação do objeto que é um membro de uma classe (atributo de uma classe) pode mudar em tempo de execução
  - Isso não é possível quando usamos herança
  - Veja um exemplo simples

```
package p2.exemplos;

public class Filho {
   public void obedece() {
```

```
        System.out.println("Filho obedece");
    }
}
```

```
package p2.exemplos;

public class FilhoChateado extends Filho {
    @Override
    public void obedece() {
        System.out.println("Filho obedece chateado");
    }
}
```

```
package p2.exemplos;

public class FilhoContente extends Filho {
    @Override
    public void obedece() {
        System.out.println("Filho obedece contente");
    }
}
```

```java
package p2.exemplos;

public class Mae {
    private Filho filho = new FilhoContente();
    private boolean contente = true;

    public void mudaHumorDeFilho() {
        if (contente)
            filho = new FilhoChateado();
        else
            filho = new FilhoContente();
    }

    public void manda() {
        filho.obedece();// chamada polimórfica
    }
}
```

```java
package p2.exemplos;
```

```
/**
 * Exemplo de composição e herança com mudança do
atributo em tempo de execução
 * e chamadas polimórficas.
 *
 * @author raquel
 *
 */
public class HoraDaEscola {

    public static void main(String[] args) {
        Mae mae = new Mae();

        mae.manda();
        mae.mudaHumorDeFilho();
        mae.manda();
    }

}
```

- A saída deste programa é:

```
Filho obedece contente
Filho obedece chateado
```

Voltar