Tópicos Especiais em Sistemas para Internet-I

Composição e Herança
Membros *static e final*

Neste tópico iremos diferenciar composição de herança e explorar o conceito de membros *static* e *final*.

Prof. Ciro Cirne Trindade

Combinando Composição e Herança

- É usual utilizar os conceitos de composição e herança juntos
- As facilidades oferecidas pelo conceito de herança como sobrecarga de construtores e métodos, podem ser aplicadas para instanciar todo um conjunto de classes
- No desenvolvimento a composição é a técnica predominante
- Herança geralmente ocorre mais no *design* de tipos

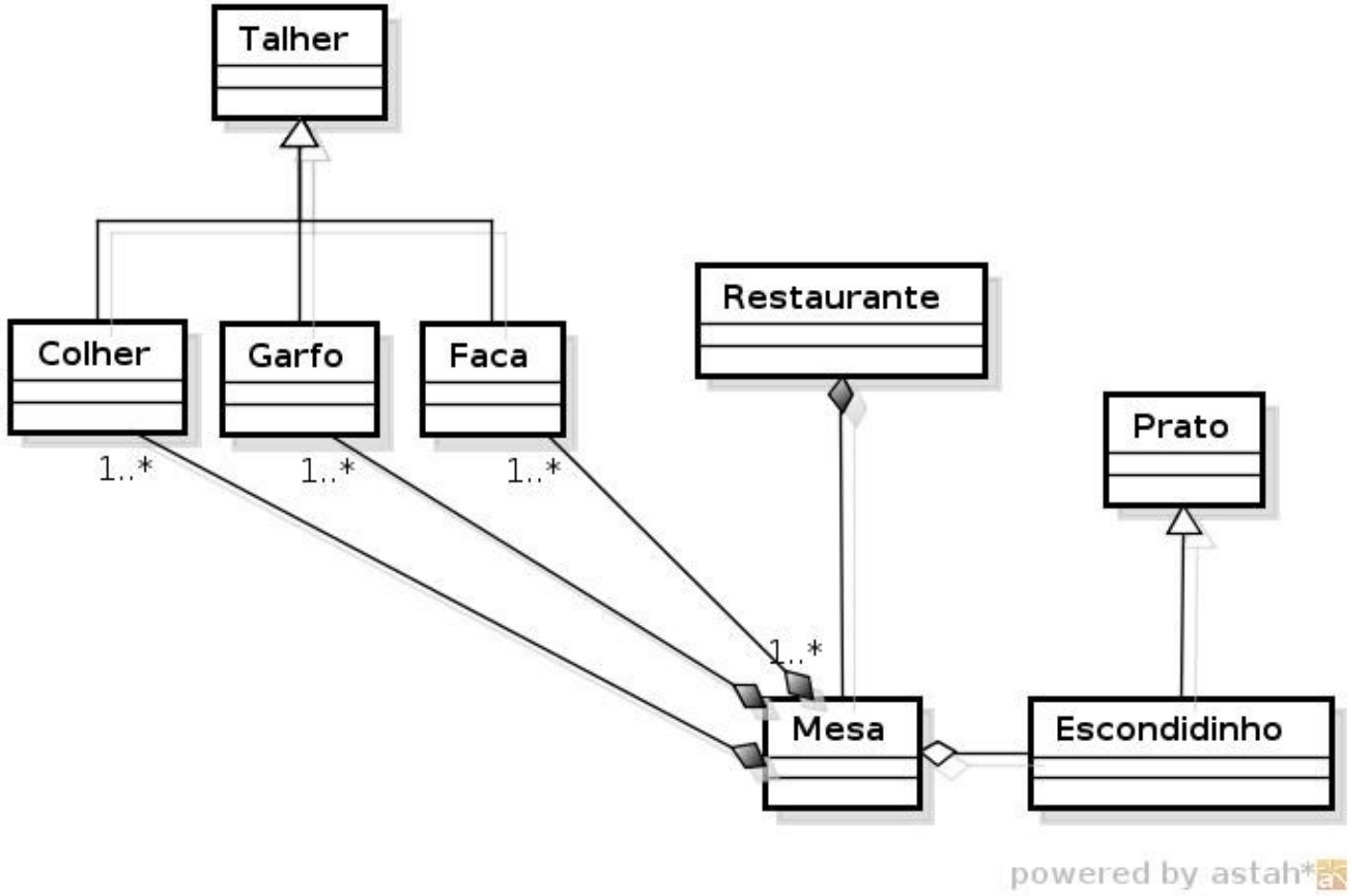
Quando usar Composição ou herança?

- **Composição**
  - Identifique os componentes do objeto, suas partes
    - Essas partes devem ser agregadas ao objeto via composição (***é parte de***)

Quando usar Composição ou herança?

- **Herança**
  - Classifique seu objeto e tente encontrar uma semelhança de identidade com classes existentes
    - Herança só deve ser usada se você puder comparar seu objeto A com outro B dizendo, que A "***É UM tipo de...***" B
    - Tipicamente, herança só deve ser usada quando você estiver construindo uma família de tipos (relacionados entre si)

Diagrama “Restaurante”



Composição e Herança (1/5)

```
package aula6;

public class Prato {
   public Prato() {
        System.out.println("Construtor de Prato");
   }
}
```

```
package aula6;

public class Escondidinho extends Prato  {
   public Escondidinho() {
        System.out.println("Construtor de Escondidinho");
   }
}
```

Composição e Herança (2/5)

```
package aula6;

public class Talher {
   public Talher() {
        System.out.println("Construtor de Talher");
   }
}
```

```
package aula6;

public class Colher extends Talher {
   public Colher() {
        System.out.println("Construtor de Colher");
   }
}
```

Composição e Herança (3/5)

```
package aula6;
public class Garfo extends Talher {
    public Garfo() {
        System.out.println("Construtor de Garfo");
    }
}
```

```
package aula6;
public class Faca extends Talher {
    public Faca() {
        System.out.println("Construtor de Faca");
    }
}
```

Composição e Herança (4/5)

```
package aula6;

public class Mesa {

   private Colher[] colheres;
   private Garfo[] garfos;
   private Faca[] facas;
   private Escondidinho jantar;
   public Mesa (int lugares){
        colheres = new Colher[lugares];
        garfos = new Garfo[lugares];
        facas = new Faca[lugares];
        jantar = new Escondidinho();
        System.out.println("Construtor de Mesa");
   }
}
```

Composição e Herança (5/5)

```
package aula6;

public class Restaurante {
    private Mesa[] mesas;

    public Restaurante(int numMesas) {
        mesas = new Mesa[numMesas]
        System.out.println("Construtor de Restaurante com
   " + numMesas + " mesas");
     }

    public static void main(String[] args) {
        Restaurante r = new Restaurante(5);
        ...
    }
}
```

Exercício

- No método *main()* da classe *Restaurante* Implemente uma aplicação para gerenciar a reserva de mesas em um restaurante. No final exiba o número de mesas reservadas. Acrescente os métodos que você julgar necessários às classes mostradas anteriormente.

Atributos estáticos (1/3)

- Também conhecidos como variáveis de classe
- Utilizados quando:
  - Todos os objetos da classe devem compartilhar a mesma cópia dessa variável de instância; ou
  - Essa variável de instância deve ser acessível mesmo quando não existir nenhum objeto da classe

Atributos estáticos (1/3)

- Podem ser acessados com o nome da classe ou com o nome de um objeto e um ponto (.)
  - Devem ser públicos para serem acessíveis fora da classe
  - Devem ser inicializados nas suas declarações ou, caso contrário, o compilador irá inicializá-los com um valor-padrão (zero para tipos numéricos)

Atributos estáticos (2/2)

- Para declarar um atributo como estático, utilize a palavra-chave **`static`** antes do tipo do atributo
- Forma geral:
  - `[acesso] static tipo atributo [= valor];]`
- Utilize um atributo estático quando todos os objetos de uma classe precisarem utilizar a mesma cópia da variável
- Exemplo: suponha que você quer utilizar um contador para saber quantas vezes uma classe foi instanciada

Métodos estáticos (1/2)

- Métodos estáticos de uma classe existem e podem ser utilizados, mesmo se **nenhum objeto** dessa classe tiver sido instanciado
- Assim como os atributos estáticos, podem ser referenciados através do nome da classe seguido de um . (ponto)
  - `Classe.metodo([argumentos])`

Métodos estáticos (2/2)

- Métodos estáticos também são declarados através da palavra-chave **`static`** antecedendo o tipo do método
- Forma geral:
  - `[acesso] static tipo metodo([argumentos]) { … }`

Exemplo

- A classe Math possui vários métodos estáticos
  - `public static double pow(double, double);`
  - `public static double sqrt(double);`
  - `public static double sin(double);`
  - `...`
- Para obter a raiz quadrada de um número poderíamos fazer:

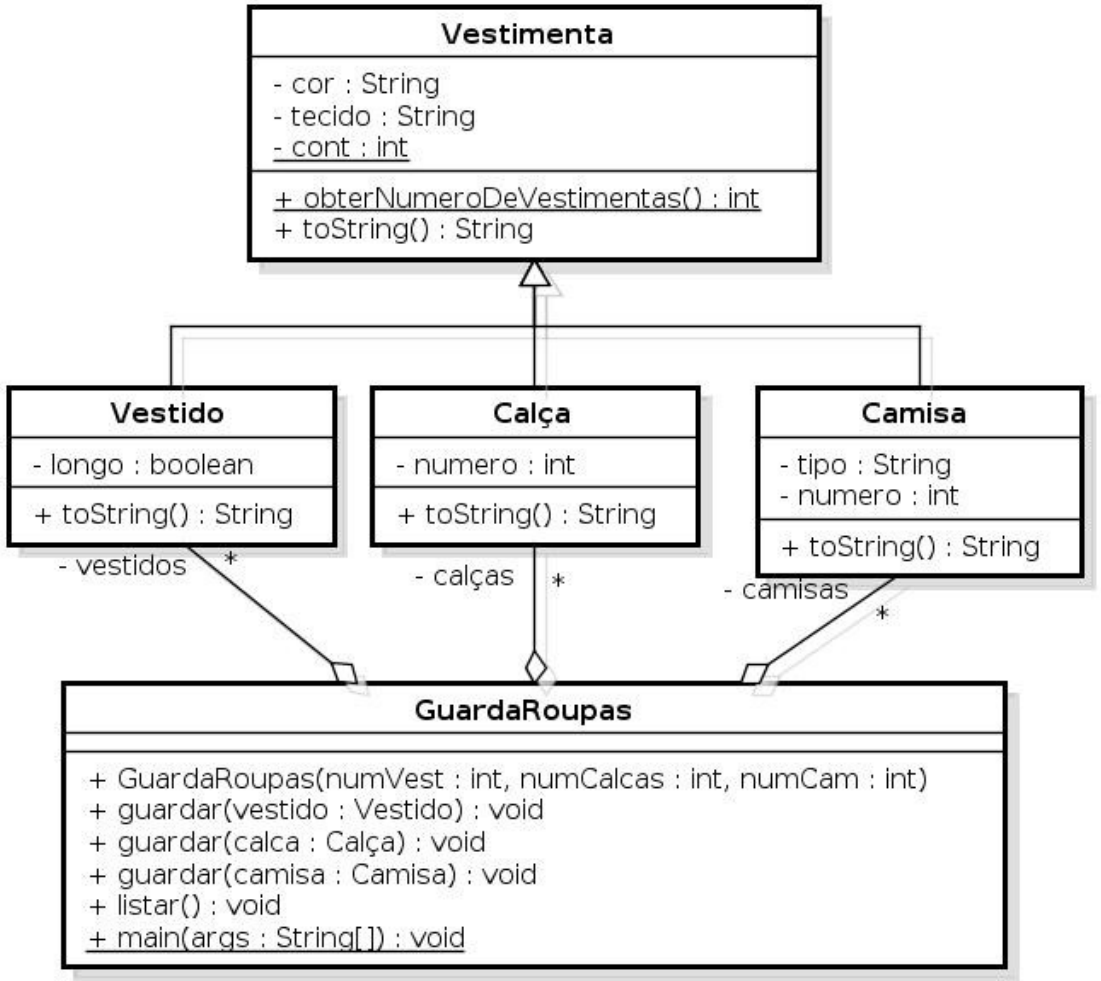
```
double raiz = Math.sqrt(val);
```

  - Repare que não há um objeto da classe Math

Restrições sobre métodos estáticos

- Métodos declarados como estáticos possuem várias restrições:
  - Eles só podem chamar outros métodos estáticos
  - Eles só podem acessar atributos estáticos
  - Eles não podem fazer referência a **`this`** e **`super`**

Exercício (1/2)



Exercício (2/2)

- Considere o diagrama de classes do slide anterior
- O atributo *cont* da classe *Vestimenta* deve registrar o número de objetos dessa classe instanciados
- No método *main()* da classe GuardaRoupas instancie um objeto da classe *GuardaRoupas*
- Permita guardar calças, camisas e vestidos no guarda roupas
- A qualquer momento deve ser possível listar as vestimentas do guarda roupas
- Ao final apresente o número de objetos da classe *Vestimenta* criados

Constantes

- Um atributo pode ser declarado como **`final`**
- Isto faz com que seu conteúdo não possa ser modificado (definição de uma constante)
- Sendo assim, você deve inicializar um atributo **`final`** quando ele é declarado
- O uso de **`final`** é semelhante ao de **`const`** em C/C++
- Exemplos:
  - `final int FILE_NEW = 1;`
  - `final int FILE_OPEN = 2;`

Constantes

- Normalmente atributos **final** são declarados como estáticos, já que todas as instâncias da classe compartilham o mesmo valor
- Exemplo:
  - `static final int FILE_NEW = 1;`
  - `static final int FILE_OPEN = 2;`

Métodos `final` (1/2)

- Há 2 razões para declarar métodos como final:
  - Bloquear a sobreposição (*overrinding*)
    - métodos `final` não podem ser sobrepostos em classes derivadas
  - Eficiência
    - tornar um método `final` permite ao compilador converter as chamadas a este método em chamadas *inline*
    - semelhante a uma macro em C/C++

Métodos `final` (2/2)

- Métodos `private` são implicitamente `final`
  - Métodos privados não são acessíveis à classes derivadas, portanto não podem ser sobrepostos
  - Criar um método na classe derivada com a mesma assinatura de um método privado da super classe não o sobrepõem, você simplesmente cria um novo método

Exemplo de métodos **final** (1/2)

```
// SobreposiçãoIlusória.java
package aula9;
class ComFinal {
   private final void f() {
      System.out.println("ComFinal.f()");
   }
   private void g() {
      System.out.println("ComFinal.g()");
   }
}

class SobreposiçãoFinal extends ComFinal {
   private final void f() {
      System.out.println("SobreposiçãoFinal.f()");
   }
   private void g() {
      System.out.println("SobreposiçãoFinal.g()");
   }
}
```

Exemplo de métodos **final** (2/2)

```
public class SobreposiçãoIlusória extends SobreposiçãoFinal{
    public final void f() {
        System.out.println("SobreposiçãoIlusória.f()");
    }
    public void g() {
        System.out.println("SobreposiçãoIlusória.g()");
    }
    public static void main(String[] args) {
        SobreposiçãoIlusória si = new SobreposiçãoIlusória();
        si.f();
        si.g();
        SobreposiçãoFinal sf = new SobreposiçãoFinal();
        //! Erro: sf.f();
        //! Erro: sf.g();
        ComFinal cf = new ComFinal();
        //! Erro: cf.f();
        //! Erro: cf.g();
    }
}
```

Classes `final`

- Quando uma classe inteira é **`final`** (precedendo sua definição com a palavra-chave **`final`**), ela não pode ser herdada por nenhuma outra classe
- Você pode querer fazer isso por questões de segurança ou eficiência

Referências

- SHILDT, Herbert. *Java 2: the complete reference.* 5. ed., McGraw-Hill, 2002.
- DEITEL, H.M.; DEITEL, P.J.. *Java Como Programar.* 4. ed., Porto Alegre: Bookman, 2002.
- ECKEL, B.. *Thinking in Java.* 3. ed., Prentice Hall, 2002.