

Herança

Dalton Serey
© 2004 DSC/UFCG

Motivação

- Como podemos reusar código?
- O que é especialização ou herança?
- O que é composição?
- O que é polimorfismo?

Reúso de Código

- Às vezes é conveniente escrever uma classe a partir de outra...
- Podemos fazer isso de duas formas:
 - por **composição**
 - usarmos instâncias de classes existentes para “compor” a funcionalidade da nova classe
 - por **especialização** ou **herança**
 - estendemos uma classe existente, adicionando-lhe funcionalidades específicas

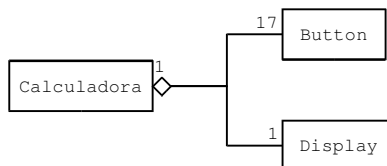
Composição

- As instâncias das classes existentes são atributos da nova classe

```
class Button { ... }  
class Display { ... }  
  
class Calculadora {  
    private Display display = new Display();  
    private Button ce = new Button();  
    private Button tecla0 = new Button();  
    private Button tecla1 = new Button();  
    ...  
}
```

Sintaxe de Composição

- Graficamente, costumamos usar a seguinte notação (ao estilo UML)



Sintaxe de Composição

- Com composição, a interface das classes existentes é escondida do código cliente!
- Há duas possíveis abordagens
 - tornar o objeto-parte **public**
 - tornar o objeto-parte **private**
- Contudo, se o objeto-parte é **private**, ainda é possível controlar sua visibilidade
 - podemos criar métodos na classe composta que externalizam parte da interface da parte

Especialização ou Herança

- Conveniente para **estender/especializar** a interface de uma classe existente
- A intenção é descrever um tipo de objeto a partir da **interface** e da **funcionalidade** de um tipo concreto existente
 - compare esta idéia com a de implementar uma interface...

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Sintaxe de Especialização

- Exemplo:
 - suponha que temos a classe `Contador`, definida abaixo
 - como podemos criar uma classe `ContadorResetável`, usando composição?

```
class Contador {
    private int valor = 0;
    public void inc() { valor++; }
    public int valor() { return valor; }
}
```

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Sintaxe de Especialização

- Eis uma solução... (não satisfatória!)

```
class Contador {
    private int valor = 0;
    public void inc() { valor++; }
    public int valor() { return valor; }
}

class ContadorResetável {
    private c = new Contador();
    public void inc() { c.inc(); }
    public int valor() { return valor; }
    public void reset() { c = new Contador(); }
}
```

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Sintaxe de Especialização

- Eis a solução usando especialização...

```
class Contador {
    protected int valor = 0;
    public void inc() { valor++; }
    public int valor() { return valor; }
}

class ContadorResetável extends Contador {
    public void reset() { valor = 0; }
}
```

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Sintaxe de Especialização

- Note a palavra-chave `extends`
- Indicar que a classe `ContadorResetável` é obtida pela especialização de `Contador`

```
class Contador {
    protected int valor = 0;
    ...
}

class ContadorResetável extends Contador {
    public void reset() { valor = 0; }
}
```

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Sintaxe de Especialização

- Para quê o modificador `protected`?
- Indica que `valor` é acessível de sub-classes
 - atenção: todo campo `protected` é automaticamente *friendly*!

```
class Contador {
    protected int valor = 0;
    ...
}

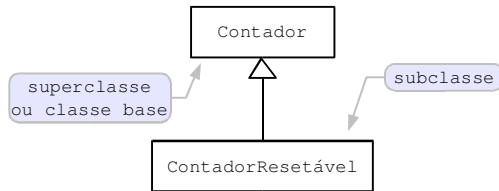
class ContadorResetável extends Contador {
    public void reset() { valor = 0; }
}
```

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Sintaxe de Especialização

- Graficamente, costumamos usar a seguinte notação (ao estilo UML)



Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Semântica de Especialização

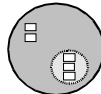
- Especializar uma classe permite
 - adicionar atributos específicos
 - adicionar novos métodos
 - sobrescrever métodos da superclasse

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Semântica de Especialização

- Idéia básica: objetos da subclasse contêm um objeto (interno) da superclasse
- Quando uma mensagem chega ao objeto:
 - inicialmente, tenta-se resolvê-la, usando métodos específicos da subclasse
 - se não há métodos que casem, a mensagem é repassada para o “objeto” da superclasse



Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Semântica de Especialização

- Ao instanciar um objeto de uma subclasse, um objeto da superclasse também é criado
 - por default, o construtor padrão da superclasse é chamado
- Através desse objeto, a subclasse tem acesso direto aos métodos e atributos (visíveis) da superclasse
 - para isto, usamos a palavra-chave “`super`”
 - semelhante a “`this`”

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Semântica de Especialização

- A palavra-chave “`super`” permite que métodos sobrescritos sejam usados
 - e também métodos protegidos
- Isto permite que o método original seja chamado pelo método que o sobrescreve

```
public void umMetodo() {
    super.umMetodo();
    valor++; // dá um tratamento mais específico
}
```

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Construtores e Especialização

- Java adiciona uma chamada a `super()`, mesmo que não esteja no código
- Podemos ser explícitos e indicar a Java qual construtor da superclasse deve ser usado
- Para isso, `super(...)` deve ser a primeira linha do construtor da subclasse

```
public AlgumaSubclasse(int valor) {
    super(valor);
}
```

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

A classe Object

- Especialização é parte fundamental em Java
- De fato, qualquer classe é formada por especialização de outra
- Quando não especificamos uma classe a ser estendida, Java assume “`extends Object`”
 - logo, `Object` é a raiz da hierarquia de classes
 - é dela que vêm os métodos `clone()`, `wait()`, `notify()`, `toString()`, etc.

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Classes Abstratas

- Às vezes, é conveniente criar classes que tenham métodos não implementados
- Esquema usado para fatorar código comum
 - somente nas subclasses é que tais métodos serão realmente implementados
- Nesses casos, é conveniente evitar que a superclasse possa ser instanciada
- Para isso, declaramos tais classes e métodos como `abstract`

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Classes Abstratas

- Se um método é declarado abstrato não deve ter implementação (naquele ponto)
- Classes com pelo menos um método abstrato, também devem ser declaradas abstratas
- Classes abstratas
 - não podem ser instanciadas
 - podem incluir métodos abstratos
 - podem incluir implementações parciais
- Interfaces são classes abstratas ao extremo!

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Especialização ou Composição?

- Especialização
 - preserva a interface da classe usada
 - útil quando há uma relação de **subtipo**
 - o mau uso leva a hierarquias inadequadas e difíceis de gerenciar
 - impõe relação estática entre as classes

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Especialização ou Composição?

- Composição
 - não preserva a interface da classe usada
 - embora possamos adaptá-la
 - não estabelece uma relação de **subtipo**
 - é fácil de usar e, em muitos casos, é fácil de converter em especialização
 - permite relacionar classes dinamicamente
 - é usada, por exemplo, para criar objetos cujo comportamento muda dinamicamente

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Especialização ou Composição?

- Conclusão: **na dúvida, use composição!**

Versão 1.2

© 2003, 2004 Dalton Serey DSCI/UFMG

Polimorfismo

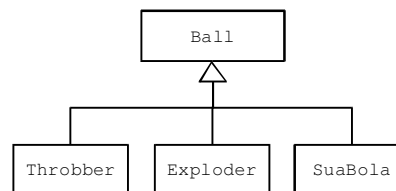
- Um dos usos importantes de especialização é para permitir **polimorfismo**!
- De fato, já vimos isto quando falamos de interfaces...
- A idéia é que uma classe possa lidar diretamente com um supertipo (ou tipo base), ignorando detalhes específicos

Versão 1.2

© 2003, 2004 Dalton Serey DSC/UFG

Polimorfismo

- Eis o diagrama de tipos da interface Ball

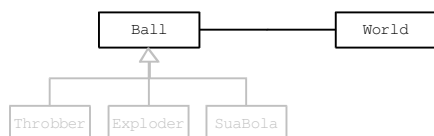


Versão 1.2

© 2003, 2004 Dalton Serey DSC/UFG

Polimorfismo

- E eis como um Ball se relaciona ao World
 - o que a classe `World` sabe sobre `SuaBola`?

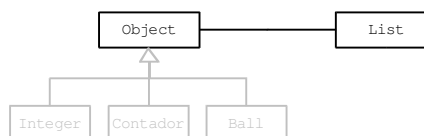


Versão 1.2

© 2003, 2004 Dalton Serey DSC/UFG

Polimorfismo

- O mesmo efeito pode ser conseguido se o tipo base é uma classe



Versão 1.2

© 2003, 2004 Dalton Serey DSC/UFG

Polimorfismo

- Assim, mensagens enviadas por List para Objects são na realidade resolvidas pelas classes concretas

```
list.add(new Integer(10));
list.add(new MeuContador());
list.add("teste");
System.out.println(list);
```

```
String toString() {
    String s;
    Iterator i = list.iterator();
    while ( i.hasNext() )
        s += i.next().toString(); //qual o tipo?
}
```

Versão 1.2

© 2003, 2004 Dalton Serey DSC/UFG

Upcasts e Downcasts

- Coerções que “sobem” na hierarquia em direção à base são ditas *upcasts*
 - upcasts são sempre válidos
 - princípio da substitutabilidade
- Coerções que “descem” na hierarquia são chamadas de *downcasts*
 - nem sempre válidas (`ClassCastException`)
 - requer todo o cuidado (`instanceof`, etc...)

Versão 1.2

© 2003, 2004 Dalton Serey DSC/UFG

Exercício

- Quais as diferenças e semelhanças entre classes abstratas e interfaces? Como, em Java, um objeto pode ser de vários tipos ao mesmo tempo? O que é polimorfismo?
- Escreva classes para representar figuras geométricas simples, tais como triângulos, quadrados, retângulos, círculos, elipses, polígonos, etc, desenhadas sobre um plano cartesiano. Todas as figuras devem ser capazes de verificar se um dado ponto pertence ou não a elas e também devem ser capazes de calcular sua área. Também deve ser possível verificar se duas figuras se sobrepõem no espaço.