Java First-Tier: Aplicações

Orientação a Objetos em Java (III)

Grupo de Linguagens de Programação

Departamento de Informática
PUC-Rio

Herança: Simples × Múltipla

- O tipo de herança que usamos até agora é chamado de *herança simples* pois cada classe herda de apenas uma outra.
- Existe também a chamada *herança múltipla* onde uma classe pode herdar de várias classes.

2

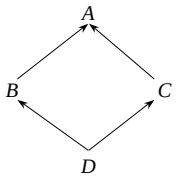
Herança Múltipla

- Herança múltipla não é suportada por todas as linguagens OO.
- Esse tipo de herança apresenta um problema quando construímos hierarquias de classes onde uma classe herda duas ou mais vezes de uma mesma superclasse. O que, na prática, torna-se um caso comum.

3

Problemas de Herança Múltipla

- O problema de herdar duas vezes de uma mesma classe vem do fato de existir uma herança de código.



4

Compatibilidade de Tipos

Inúmeras vezes, quando projetamos uma hierarquia de classes usando herança múltipla, estamos, na verdade, querendo declarar que a classe é *compatível* com as classes herdadas. Em muitos casos, a herança de código não é utilizada.

5

Revendo Classes Abstratas

- O uso de classe abstrata para expressar compatibilidade impõe a restrição sobre herança de classes.

```
abstract class ItemCompra {
   public abstract float obtemPreço();
}
class Processador extends ItemCompra {
    public float obtemPreço();
}
```

Como definir uma classes Processador que é um Equipamento e um Item de Compra ao mesmo tempo?

6

Interfaces

- Algumas linguagens OO incorporam o conceito de duas classes serem compatíveis através do uso de compatibilidade estrutural ou da implementação explícita do conceito de *interface*.

7

Em Java

- Java não permite herança múltipla com herança de código.
- Java implementa o conceito de *interface*.
- É possível herdar múltiplas interfaces.
- Em Java, uma classe *estende* uma outra classe e *implementa* zero ou mais interfaces.
- Para implementar uma interface em uma classe, usamos a palavra **implements**.

8

Exemplo de Interface

- Ao implementarmos o TAD Pilha, poderíamos ter criado uma interface que definisse o TAD e uma ou mais classes que a implementassem.

```
interface Stack {
  boolean isEmpty();
  void push(int n);
  int pop();
  int top();
}
```

9

Membros de Interfaces

- Uma vez que uma interface não possui implementação, devemos notar que:
  - seus atributos devem ser públicos, estáticos e constantes;
  - seus métodos devem ser públicos e abstratos.
- Como esses qualificadores são fixos, não precisamos declará-los (note o exemplo anterior).

10

Membros de Interfaces (cont.)

- Usando os modificadores explicitamente, poderíamos ter declarado nossa interface da seguinte forma:

```
interface Stack {
  public abstract boolean isEmpty();
  public abstract void push(int n);
  public abstract int pop();
  public abstract int top();
}
```

11

Pilha revisitada

```
class StackImpl implements Stack {
  private int[] data;
  private int top_index;
  StackImpl(int size) {
    data = new int[size];
    top_index = -1;
  }
  public boolean isEmpty() { return (top_index < 0); }
  public void push(int n) { data[++top_index] = n; }
  public int pop() { return data[top_index--]; }
  public int top() { return data[top_index]; }
}
```

12

Resumindo Interfaces

- Não são classes
- Oferece compatibilidade de tipos de objetos

```
Comparable  x; // Comparable é uma interface
x = new Pessoa(); // Pessoa implementa Comparable
```

- Permite o uso de *instanceof*

```
if (x instanceof Comparable) {…}
```

- Uma interface pode estender outra

```
public interface Compativel extends Comparable
{ … }
```

13

Pacotes

14

Modularidade em Java: Pacotes

- Além das classes, Java provê um recurso adicional que ajuda a modularidade: o uso de pacotes.
- Um pacote é um conjunto de classes e outros pacotes.
- Pacotes permitem a criação de espaços de nomes, além de mecanismos de controle de acesso.

15

Pacotes: Espaços de Nomes

- Pacotes, a princípio, possuem nomes.
- O nome do pacote qualifica os nomes de todas as classes e outros pacotes que o compõem.
- Exemplo: classe **Math**.

```
int a = java.lang.Math.abs(-10); // a = 10;
```

16

Implementação de Pacotes

- Pacotes são tipicamente implementados como diretórios.
- Os arquivos das classes pertencentes ao pacote devem ficar em seu diretório.
- Hierarquias de pacotes são construídas através de hierarquias de diretórios.

17

"Empacotando" uma Classe

- Para declararmos uma classe como pertencente a um pacote, devemos:
  - declará-la em um arquivo dentro do diretório que representa o pacote;
  - declarar, na primeira linha do arquivo, que a classe pertence ao pacote.

18

Importação de Pacotes

- Podemos usar o nome simples (não qualificado) de uma classe que pertença a um pacote se *importarmos* a classe.
- A importação de uma classe (ou classes de um pacote) pode ser feita no início do arquivo, após a declaração do pacote (se houver).
- As classes do pacote padrão **java.lang** não precisam ser importadas (Ex.: **Math**).

19

Exemplo de Arquivo

```
package datatypes; // Stack pertence a datatypes.
import java.math.*; // Importa todas as classes.
import java.util.HashTable; // Importa HashTable.
/*
  A partir desse ponto, posso usar o nome
  HashTable diretamente, ao invés de usar
  java.util.HashTable. Assim como posso usar
  diretamente o nome de qualquer classe que
  pertença ao pacote java.math.
*/
public class Stack { // Stack é exportada.
   ...
}
```

20

Controle de Acesso

21

Encapsulamento

- Na classe Stack, nós encapsulamos a definição de pilha que desejávamos, porém, por falta de *controle de acesso*, é possível forçar situações nas quais a pilha não se comporta como desejado.

```
Stack s = new Stack(10);
s.push(6);
s.top_index = -1;
System.out.println(s.isEmpty()); // true!
```

22

Controle de Acesso

- As linguagens OO disponibilizam formas de controlar o acesso aos membros de uma classe. No mínimo, devemos poder fazer diferença entre o que é *público* e o que é *privado*.
- Membros públicos podem ser acessados indiscriminadamente, enquanto os privados só podem ser acessados pela própria classe.

23

Redefinição de **Stack**

```
class Stack {
  private int[] data;
  private int top_index;
  Stack(int size) {
    data = new int[size];
    top_index = -1;
  }
  boolean isEmpty() { return (top_index < 0); }
  void push(int n) { data[++top_index] = n; }
  int pop() { return data[top_index--]; }
  int top() { return data[top_index]; }
}
```

24

Exemplo de Controle de Acesso

- Com a nova implementação da pilha, o exemplo anterior não pode mais ser feito pois teremos um erro de compilação.

```
Stack s = new Stack(10);
s.push(6);
s.top_index = -1; // ERRO! A compilação pára aqui!
System.out.println(s.isEmpty());
```

25

Pacotes: Controle de Acesso

- Além de membros públicos e privados, temos também membros de pacote.
- Um membro de pacote só pode ser acessado por classes declaradas no mesmo pacote da classe que declara esse membro.
- Quando omitimos o modificador de controle de acesso, estamos dizendo que o membro é de pacote.

26

Mais sobre Visibilidade

Pelo que foi dito até agora, membros públicos podem ser acessados indiscriminadamente, membros privados só podem ser acessados pela própria classe, e membros de pacote são acessados por classes declaradas no mesmo pacote da classe.

27

Mais sobre Visibilidade

- Às vezes precisamos de um controle de acesso intermediário: um membro que seja acessado somente nas sub-classes e nas classes declaradas no mesmo pacote. As linguagens OO tipicamente dão suporte a esse tipo de acesso.
- Para isso usamos o modificador de controle de acesso **protected** em Java.

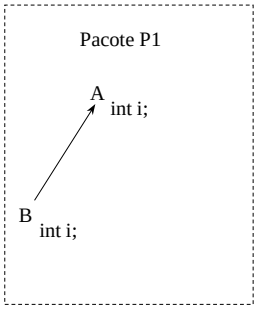
28

Resumo de Visibilidade em Java

- Resumindo todos os tipos de visibilidade:
  - **private**: membros que são acessados somente pela própria classe;
  - **protected**: membros que são acessados pelas suas sub-classes e pelas classes do pacote;
  - **public**: membros são acessados por qualquer classe;
  - sem modificador ou *default*: membros que são acessados pelas classes do pacote.

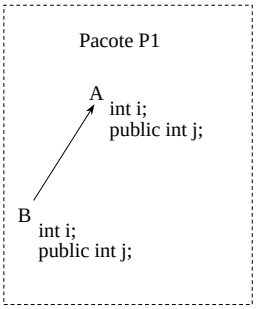
29

Visibilidade de Membros



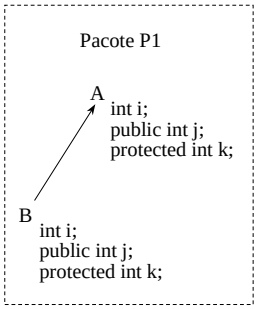
30

Visibilidade de Membros



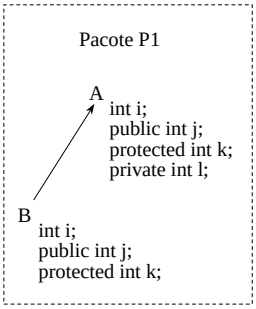
31

Visibilidade de Membros



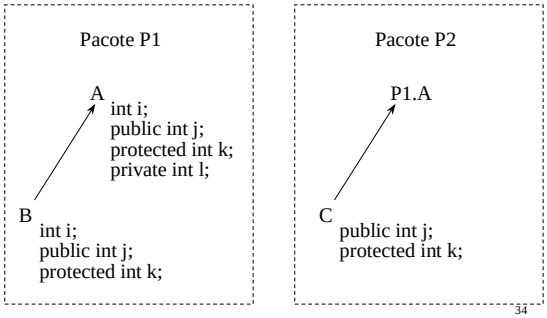
32

Visibilidade de Membros



33

Visibilidade de Membros



34

Aninhamento de Classes

35

Aninhamento de Classes

- Em diversas circunstâncias precisamos criar classes cujo único objetivo é auxiliar na implementação de uma outra classe. Nesses casos, podemos declarar uma classe aninhada, ou seja, declarar uma nova classe como um membro de uma outra.
- Diversas linguagens OO suportam esse recurso.

36

Aninhamento em Java

- Java permite dois tipos diferentes de aninhamento de tipos:
  - Aninhamento estático;
  - Aninhamento dinâmico.

37

Aninhamento Estático

- Gera classes e interfaces normais, cuja única singularidade é o nome, que passa a ser qualificado pelo nome da classe que as declara.
- Em particular, sendo um membro de uma classe, uma interface ou classe aninhada está sujeita aos modificadores de controle de acesso.

38

Exemplo de Aninhamento Estático

```
package p;

public class A {
  public static class B {
    ...
  }
}
```

```
p.A a = new p.A();
p.A.B b = new p.A.B();
```

39

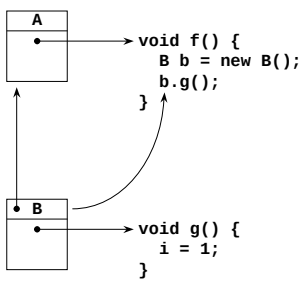
Aninhamento Dinâmico

- Gera classes associadas a objetos.
- Cada instância da classe aninhada possui uma referência para o objeto a partir do qual ela é criada.
- Como ela está associada a um objeto, ela tem acesso a todos os membros desse objeto.

40

Exemplo de Aninhamento Dinâmico

```
class A {
  int i;
  class B {
    void g() {...}
  }
  void f() {...}
}
```




```
void f() {
  B b = new B();
  b.g();
}
```

```
void g() {
  i = 1;
}
```

41

Próxima Aula de Laboratório

Será aplicado o conceito de interfaces a um dicionário. Um dicionário é um TAD que permite a armazenagem de valores associados a chaves.

Além deste exercício, será pedido para que se crie um pacote SistemaBancário, a partir das implementações desenvolvidas no laboratório sobre heranças.

42