ACH 2003 - Computação Orientada a Objetos

Apresentação da disciplina e conceitos fundamentais

Prof. Flávio Luiz Coutinho
flcoutinho@usp.br

O que vamos aprender nesta disciplina?

- Fundamentos/conceitos de orientação a objetos.
- Algumas questões/recursos específicos da linguagem Java.
- Ferramentas de apoio ao desenvolvimento.
- Biblioteca úteis do Java e como os princípios de OO estão presentes nelas.

Importância da disciplina

Criação de software de melhor qualidade:

- melhor organizado.
- mais fácil de manter.
- mais fácil de estender.
- mais seguro.

Importância da disciplina

Criação de software de melhor qualidade:

- melhor organizado.
- mais fácil de manter.
- mais fácil de estender.
- mais seguro.

Princípios de OO ajudam a atingir estes objetivos.

Tópicos

- Conceitos/fundamentos de OO (classe, objeto, atributo, métodos, etc.)
- Tratamento de erros com exceções.
- Tipos genéricos.
- Ferramentas de apoio ao desenvolvimento (IDE, Git, JUnit)
- Refatoração
- Coleções Java
- I/O
- UML
- SOLID
- Noções de programação concorrente

Foco da disciplina

Em COO, o foco é menos sobre:

- como resolver um problema (IP, AED)
- como avaliar sua eficiência (IAA)

e mais sobre:

- como criar um bom software que resolve o problema.

Foco da disciplina

Em COO, o foco é menos sobre:

- como resolver um problema (IP, AED)
- como avaliar sua eficiência (IAA)

e mais sobre:

- como criar um bom software que resolve o problema.

Correta aplicação dos conceitos de POO favorece a criação de software de melhor qualidade.

Motivação

Problema: escrever um programa que leia um sequência de valores inteiro, e os imprima na ordem inversa à ordem de leitura.

Motivação

Problema: escrever um programa que leia um sequência de valores inteiro, e os imprima na ordem inversa à ordem de leitura.

Uma pilha resolve o problema!

Motivação

Problema: escrever um programa que leia um sequência de valores inteiro, e os imprima na ordem inversa à ordem de leitura.

Uma pilha resolve o problema! Para realizar a inversão, basta inserir, um a um, os valores lidos na pilha e, em seguida, removê-los um a um.

Motivação

Problema: escrever um programa que leia um sequência de valores inteiro, e os imprima na ordem inversa à ordem de leitura.

Uma pilha resolve o problema! Para realizar a inversão, basta inserir, um a um, os valores lidos na pilha e, em seguida, removê-los um a um.

Apesar de termos clareza de como resolver o problema, e qual estrutura de dedos é a mais adequada para isso, ainda assim são possíveis escrever várias versões de um programa que resolva este problema, com graus distintos de qualidade (e aqui não estamos falando sobre resolver ou não o problema, ou se é eficiente ou não, mas sim se o programa é bem escrito).

Motivação

Problema: escrever um programa que leia um sequência de valores inteiro, e os imprima na ordem inversa à ordem de leitura.

Uma pilha resolve o problema! Para realizar a inversão, basta inserir, um a um, os valores lidos na pilha e, em seguida, removê-los um a um.

Apesar de termos clareza de como resolver o problema, e qual estrutura de dedos é a mais adequada para isso, ainda assim são possíveis escrever várias versões de um programa que resolva este problema, com graus distintos de qualidade (e aqui não estamos falando sobre resolver ou não o problema, ou se é eficiente ou não, mas sim se o programa é bem escrito).

Implementações...

Retomando o exemplo da pilha

Versão 1 (escrita em C): ausência de separação do código “usuário” da pilha, da implementação da estrutura. Duas responsabilidades misturadas.

Versão 2 (escrita em C): código que implementa as operações da pilha separados em funções. Alguma separação de responsabilidade. Mas o conhecimento de que as variáveis “free” e “values” representam juntas o estado de uma única estrutura do tipo pilha ainda é papel do “usuário”.

Versão 3 (escrita em C): criação de uma *struct* para representar o tipo pilha. Bem melhor que as versões anteriores, diminui ainda mais o conhecimento que o “usuário” tem que ter a respeito do funcionamento da pilha. Mas o estado interno da pilha está exposto para quem a manipula. Tal exposição é de fato necessária? Daria para evitar?

Retomando o exemplo da pilha

Versão 3b (escrita em Java): mera tradução da versão 3 em C. Conceito de classe empregado de forma limitada.

Versão 4 (escrita em Java): conceito de classe aplicado de forma um pouco melhor, mas ainda limitada. Uma classe define um tipo (atributos) e o conjunto de operações associados ao tipo (métodos). Melhor organização do código, mas estrutura interna da pilha ainda exposta para seus “usuários”.

Versão 5 (escrita em Java): uso dos modificadores de acesso, protegendo estado interno, e expondo apenas o realmente essencial para os “usuários”. Reforça a razão de os comportamentos serem declarados na própria classe (impossível restringir o estado interno da pilha na versão 3b).

Conceitos básicos: classes, objetos e instâncias

Classe:

- Definição de um tipo contendo dados e as operações associadas (molde).

Objeto/instância:

- Um representante concreto de uma certa classe (criado a partir do molde).
- Cada instância possui seu conjunto particular de atributos, ou seja cada instância possui seu próprio contexto.
- Embora as instruções codificadas nos métodos sejam as mesmas para todas as instâncias, o que é “visível” para cada execução são os atributos próprios da instância que invocou a chamada do método. O objeto que invoca o método é um parâmetro implícito da chamada do método (para comparar: versão 3b - instância da pilha passada como parâmetro dos métodos estáticos - com 4 e 5 - pilha é parâmetro implícito da chamada ao método.

Instanciação

O que acontece quando executo a linha: `Stack stack = new Stack();`

- Alocação de memória suficiente para representar um objeto do tipo pilha. De forma simplificada, memória suficiente para guardar uma referência (atributo `values`) e um valor inteiro (atributo `free`).
- Após a alocação, o código do construtor é executado para preparar o objeto recém criado.
- Ao final, da execução do construtor, o endereço de memória do objeto, já alocado e iniciado, é devolvido e atribuído à variável local `stack`.

Instanciação

O que acontece quando executo a linha: `Stack stack = new Stack();`

- Alocação de memória suficiente para representar um objeto do tipo pilha. De forma simplificada, memória suficiente para guardar uma referência (atributo `values`) e um valor inteiro (atributo `free`).
- Após a alocação, o código do construtor é executado para preparar o objeto recém criado.
- Ao final, da execução do construtor, o endereço de memória do objeto, já alocado e iniciado, é devolvido e atribuído à variável local `stack`.
- E quanto ao vetor (*array*)? O vetor é um objeto a parte. Ele é alocado separadamente no construtor da classe `Stack` (`new int[100]`) e sua referência guardada no atributo `values`.

Encapsulamento

O conceito de encapsulamento está ligado à ideia de ocultar o estado interno de um objeto perante o mundo exterior (código “usuário”), expondo apenas o mínimo necessário para que os “usuários” possam usufruir de suas funcionalidades.

No exemplo da pilha, queremos que seus “usuários” possam criar uma pilha vazia, guardar coisas no topo dela, remover coisas do topo, e verificar se está vazia ou não. Ou seja, que saibam apenas o que a pilha pode fazer.

Mas não queremos que os usuários precisem saber como a pilha implementa sua funcionalidade. E nem que tenham acesso ao seu estado interno.

O encapsulamento adequado é obtido com o bom uso dos modificadores de acesso (`public`, `protected`, `<default>`, `private`), para se restringir o que não deve ser visto/usado, e permitir o que pode ser usado.

Encapsulamento

“*Ah, mas se o ‘usuário’ sabe como é implementado, apesar de não ser necessário, isso não seria um bônus?*”

- É um bônus saber como é implementado para embasar o bom uso.
- Mas ainda assim o acesso ao estado interno deve ser proibido.
- Exemplo:

> Iteração pelos valores armazenados na pilha através do acesso direto ao vetor `values`. Em um momento posterior, a implementação da pilha muda. Ao invés de vetor como estrutura de armazenamento, passe-se a usar uma lista ligada. O código “usuário” que “tirou vantagem” do conhecimento e visibilidade do estado interno da pilha não funciona mais. Uma alteração de em um local do código afeta o funcionamento de outro!
>
> Sem falar no risco adicional de corromper o estado da pilha.

Modificadores de acesso

`public`: atributos e métodos visíveis para código em qualquer classe.

`protected`: atributos e métodos visíveis para código das classes que pertencem ao mesmo pacote, e também para classes derivadas (mesmo que em pacotes distintos)

`<default>`: atributos e métodos visíveis para código das classes que pertencem ao mesmo pacote.

`private`: atributos e métodos visíveis apenas para o código da classe que os declara.

Modificador static

Por padrão, todo atributo declarado em uma classe vai existir para cada instância dela que venha a ser criada, e cada instância tem sua "própria cópia" do atributo.

Da mesma forma, os métodos declarados em uma classe enxergam além dos parâmetros recebidos e variáveis locais declaradas no método, o conjunto de atributos da instância que chama o método (independente dos modificadores de acesso associados).

Mas em certas situações (que não devem ser a regra) queremos guardar algum tipo de informação não está vinculada a instâncias específicas, e que haja apenas "uma cópia" desta informação. Para estes casos, podemos usar o modificador `static` para declarar um atributo. Dizemos que este atributo é um atributo da classe (ao contrário do padrão, que são atributos de instâncias).

Modificador static

Do mesmo modo, para métodos que implementam alguma funcionalidade que independe da existência de um objeto sobre o qual irá atuar, podemos declarar métodos `static`.

Quando a gente aprende Java sem ter muita bagagem dos conceitos de OO, podemos ficar habituados a abusar do modificador `static`, programando em um estilo procedural/imperativo ao invés de OO.

Por isso, o uso do **`static`** não deve ser algo frequente em um sistema OO bem projetado.

Tipos primitivos e objetos

No Java, um atributo ou variável pode ser de dois tipos:

- Tipos primitivos: tipos mais elementares de dados (boolean, byte, char, int, long, float, double). Equivalência razoável com tipos de dados nativos do processador, e também da linguagem C.

- Objetos: tudo aquilo que é instância de uma classe. Strings e vetores (*arrays*) de tipos primitivos também são objetos. Vetores possuem atributos e são alocados com o new. A alocação de Strings pode ser implícita, mas possuem diversos métodos.

Tipos primitivos e objetos

Por que é importante saber a diferença?

- Variáveis e atributos cujo tipo é uma classe, são na verdade são ponteiros para as instâncias alocadas. No Java estes ponteiros são chamados de referências.

- Saber esta diferença é especialmente importante para entender o que acontece na passagem de parâmetros a métodos:

    - Tipos primitivos são passados como cópia (parâmetro recebido é uma cópia do valor passado na chamada).

    - Objetos são passados como referência (parâmetro recebido é também é uma cópia do valor passado na chamada. Mas como o valor representa um endereço, na prática o objeto que o método manipula é o mesmo que foi passado para chamada).