UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 04 – Associação, Agregação, Composição, Enumeradores e *Garbage Collector*

1. ENUMERADORES

Em determinadas situações é necessário representar características discretas/enumeráveis dos elementos que fazem parte do contexto do sistema. Por exemplo, um alimento que pode ser doce ou salgado, ou um livro de biblioteca que pode estar livre, reservado ou emprestado, ou ainda a topografia de um terreno que pode ser plana, ondulada ou montanhosa, ou então os dias da semana, ou opções de um sistema. Como você modelaria tais características em um sistema computacional?

Se a sua resposta à pergunta anterior foi algo no sentido de representar tais elementos por uma analogia com um valor inteiro ou caractere, por exemplo, essa era uma boa opção para as linguagens de programação antigas ou quando usamos um modelo relacional (muito utilizado em banco de dados). Apesar deste modelo funcionar na prática, ele pode dificultar o entendimento da representação e facilitar possíveis confusões. Nos exemplos dados acima, seria muito provável que uma boa quantidade de tempo seria desperdiçada na verificação de qual valor representa doce e qual valor representa salgado. Para auxiliar neste tipo de situação foram criados os enumeradores.

Um enumerador é utilizado para definir um novo "tipo de dado", definindo uma lista de valores nominais que podem ser atribuídos a este tipo. No Java, o enumerador permite a definição de valores para a analogia com outros tipos primitivos ou objetos e também pode ser utilizado para definir métodos, inclusive construtores privados. Estes elementos, porém, não podem ser instanciados como OBJETOS e por convenção (boas práticas) são definidos por letras maiúsculas, assim como as constantes.

No Netbeans um novo arquivo para enumerador pode ser criado usando a opção de menu: "Arquivo", então "Novo Arquivo...", selecionar "Enum Java" no menu de Tipos de Arquivos, pressionar o botão "Próximo >", definir um nome para o enumerador, e pressionar o botão "Finalizar". Um exemplo de definição de enumerador é apresentado no quadro da próxima página.

O enumerador Sabor apresentado ilustra a possibilidade de uso de atributos para o enumerador a fim de controlar algum dado adicional referente ao enumerador. Neste exemplo, foram utilizados dois atributos, um valor inteiro que faz analogia e identifica o elemento como doce ou salgado e um caractere que é utilizado de forma ilustrativa neste exemplo, mostrando que não é necessário escrever um construtor (privado) que recebe parâmetros para todos os atributos. Falando em construtor, note que não foi definido um escopo para o construtor do enumerador. Caso Sabor fosse uma classe, o escopo seria do pacote (lembrando que vamos tratar deste tipo de escopo mais adiante), mas para enumerados, obrigatoriamente todos os construtores são privados, não sendo necessário especificar isto. Ainda sobre a definição do enumerador do exemplo, note que foi definido também um método público que retorna uma String informando o tipo de sabor.

Arquivo: Sabor.java

```
 1.  package poo;
 2.
 3.  /** @author Diego Buchinger */
 4.  public enum Sabor {
 5.      /** Representa o sabor doce */
 6.      DOCE(0),
 7.      /** Representa o sabor salgado */
 8.      SALGADO(1);
 9.
10.      private int valor;
11.      private char letra;
12.
13.      Sabor(int valor){
14.          this.valor = valor;
15.          if( valor == 0 )
16.              letra = 'd';
17.          else
18.              letra = 's';
19.      }
20.
21.      /** Informa o nome do sabor por uma String
22.       * @return uma String informando se o sabor
23.       * do enumerador é doce ou salgado
24.       */
25.      public String getSabor(){
26.          if(valor == 0) return "Sabor Doce!";
27.          else return "Sabor Salgado";
28.      }
29.  }
```

Feita a definição de enumeradores, o seu uso é simples. O quadro abaixo ilustra um possível uso do enumerador utilizando uma variável, fazendo uso do método definido e comparando o valor da variável. Por fim, o exemplo demonstra ainda uma outra propriedade dos enumeradores na linguagem de programação Java. Todo enumerador tem um método chamado *values()* que retorna um vetor com todos os valores definidos para o enumerador. E toda variável ou atributo referente a um enumerador, possui um método *name()* que retorna uma String igual ao nome do enumerador.

Arquivo: POO.java

```
 1.  package poo;
 2.
 3.  /** @author Diego Buchinger */
 4.  public class POO{
 5.      public static void main(String[] args) {
 6.          Sabor banana = Sabor.DOCE;
 7.          System.out.println("Banana: "+banana.getSabor());
 8.
 9.          if( banana == Sabor.DOCE )
10.              System.out.println("Banana é doce mesmo!");
11.
12.          for( Sabor sab : Sabor.values() )
13.              System.out.println( sab.name() );
14.      }
15.  }
```

Iteração conhecida como *foreach*

2. ASSOCIAÇÃO, AGREGAÇÃO E COMPOSIÇÃO

Estudamos como definir CLASSES, seus atributos e métodos. Uma observação adicional que merece ser feita neste ponto é a possibilidade de vínculo entre CLASSES, situações nas quais uma determinada CLASSE pode ter como atributo um elemento de tipo composto, ou seja, uma outra CLASSE (ou inclusive a mesma classe). A este vínculo simples entre duas CLASSES damos o nome de associação. Imagine, por exemplo, que em uma possível modelagem de uma CLASSE Carro seja importante representar os seus pneus. Poderíamos criar a classe Pneu, com seus métodos e atributos, e a CLASSE Carro com seus métodos e atributos, onde teríamos quatro atributos do tipo Pneu. Outro exemplo poderia ser a representação de uma pessoa com seus dados e, entre estes dados, há o endereço que é representado como uma CLASSE. A codificação dos exemplos é ilustrada abaixo.

Arquivo: Carro.java

```
 6.  public class Carro {
 7.      Marca nome;
 8.      Pneu p1, p2, p3, p4;
 9.  }
10.
11.  class Marca{
12.      String nome;
13.  }
14.
15.  class Pneu{
16.      int pressao;
17.      void encherPneu(){ }
18.  }
```

Arquivo: Pessoa.java

```
public class Pessoa {
    String nome, cpf;
    Endereco enderecoResidencial;
}

class Endereco{
    String rua, bairro, cidade;
    int numero;
    void showEndereco(){ }
}
```

Outro relacionamento usualmente representado na orientação a objetos é a agregação, que é uma associação com relacionamento do tipo parte-todo, no qual uma CLASSE representa o todo e uma ou mais CLASSES são as partes. Considere a representação de uma turma que é constituída de estudantes e instrutor. A turma seria o todo, uma CLASSE que representa essa relação, ao passo que os estudantes e os instrutores seriam as partes, que constituem a relação e fazem parte da turma.

Existe ainda um terceiro categoria de relacionamento, chamado de composição, muito similar a agregação. Nesta, também há um relacionamento do tipo parte-todo, mas a parte que representa o todo é responsável pelo ciclo de vida (instanciação e destruição) da ou das partes. Esta última característica é exatamente o que distingue uma agregação de uma composição. Na primeira, a agregação, o todo não possui responsabilidade pelo ciclo de vida das partes e as partes podem existir sem o todo, enquanto que na segunda, a composição, além da responsabilidade mencionada, as partes não existem sem o todo.

Para citar um exemplo de composição, imagine a representação de vários bancos e suas contas corrente e poupança. Uma conta só pode ser aberta por um banco, ou seja, na codificação, a criação de um objeto da classe Conta será feita em um método da classe Banco (assim como o fechamento ou destruição da conta). Para reforçar os exemplos citados, apresenta-se no próximo quadro alguns trechos de código que representam uma relação de agregação e composição.

Dica: a seguinte analogia pode ajudar a identificar qual a relação que se deseja usar:

- associação – relações A tem B;
- agregação – relação A é composto por B, mas B pode existir por si só;
- composição – relação A é composto por B e B não pode existir por si só.

Arquivo: Turma.java

```
 6.  public class Turma {
 7.     Aluno[] alunos;
 8.     Instrutor instrutor;
 9.     public Turma(int n){
10.        alunos = new Aluno[n];
11.     }
12.     //...
13.  }
14.  class Aluno{
15.     String nome;
16.     int matricula;
17.  }
18.  class Instrutor{
19.     String nome;
20.     int codInstrutor;
21.  }
22.
```

Arquivo: Banco.java

```
public class Banco {
   String nomeFantasia;
   int numIdentificador;
   Conta[] contas;
   //...
   public void abrirConta(){ }
   public void fecharConta(…){ }
}
class Conta{
   int agencia, numero;
   int saldo; // em centavos
   public void sacar(int valor){ }
   public void depositar(…){ }
}
```

Atenção: a definição do tipo de relação de um mesmo cenário pode variar em diferentes contextos. Geralmente esta definição é feita quando já se idealiza a dinâmica entre as classes.

2.1. Modelo O.O vs. Modelo Relacional

Talvez para muitos programadores acostumados com o paradigma imperativo simplificado, sem o uso de muitos ponteiros, ou para aqueles familiarizados com o modelo relacional dos principais bancos de dados comerciais, podem achar estranho implementar da forma exemplificada estes tipos de relacionamentos. Estes indivíduos (e isto pode incluir você ☺) estão acostumados com o uso do modelo relacional que trabalha com analogias entre registros. Por exemplo, ao invés de associar o OBJETO aluno com a turma, utilizam um valor de referência a este aluno (matrícula).

Usar o modelo relacional é uma maneira válida de se trabalhar, simples de entender e que possui os seus benefícios, mas é um modelo que também possui desvantagens. Para exemplificar, considere a possível representação relacional de uma turma com diversos alunos:

```
Turma A:  int alunos[] = {13245, 3215, 7856, 231, 1234, 2201, 1654, 7821, 6632};
      Aluno(a): matricula 13245, nome: Alice Alencar, dataNasc: 23/05/1999
       Aluno(a): matricula 3215, nome: Bruno Bletz, dataNasc: 10/02/1998
                                  (...)
```

Caso seja necessário acessar um atributo ou usar um método de um determinado aluno, seria necessário varrer a lista de registro de alunos procurando pela matrícula até encontrar o registro completo do aluno procurado. No exemplo mencionado existem poucos registros a serem pesquisados, mas considere que esta listagem pode conter centenas, milhares ou milhões de registros. Utilizando a implementação ilustrada acima, com orientação a objetos, esta etapa de varredura não seria necessária e o registro do aluno poderia ser acessado de forma direta. Todavia, note que, usando o modelo apresentado, seria fácil acessar o registro de um aluno através da sua turma, mas não seria tão fácil encontrar, por exemplo, todas as turmas nas quais o aluno está matriculado. Neste sentido entra o conceito/termo de navegabilidade.

2.2 Navegabilidade

A navegabilidade faz menção à maneira que podemos "navegar" através de um OBJETO até outros OBJETOS, sem a necessidade de fazer uma varredura pela lista de registros de um determinado tipo de OBJETO. Como mencionado anteriormente, a proposta de modelagem orientada a objetos apresentada no arquivo `Turma.java` privilegia o acesso rápido aos atributos e métodos dos alunos. Este mesmo modelo, contudo, não simplifica o acesso às turmas que um determinado aluno frequenta. Talvez para o contexto de um determinado sistema esse acesso não é necessário ou significativo, e assim poderia ser implementado desta maneira sem problemas. Mas e se a busca pelas turmas que um aluno frequenta for importante, o que fazer?

Para simplificar o acesso às turmas que um aluno frequenta, deveríamos adicionar um vetor ou lista (estrutura de dados) de turmas na CLASSE Aluno, adicionando cada registro de turma que um aluno frequenta. Note que se deve ter o cuidado de não criar um novo OBJETO turma, duplicando o registro de turma para cada aluno. Essa possível adaptação poderia ser implementada assim:

Arquivo: Turma.java (atualizada)

```
 6.  public class Turma {
 7.      Aluno[] alunos;
 8.
 9.      //...
10.
11.      public Turma(int n){
12.        alunos = new Aluno[n];
13.      }
14.
15.      //...
16.      /**
17.       * Tenta matricula um aluno na turma
18.       * @param novoAluno o objeto do aluno a ser matriculado
19.       * @return retorna true se for possível matricular o
20.       * novo aluno ou false caso contrário
21.       */
22.      public boolean matricular(Aluno novoAluno){
23.          // verifica se existe uma vaga disponível na turma
24.          boolean matricula = false;
25.          int vaga=0;
26.          for( ; vaga<alunos.length; vaga++)
27.              if( alunos[vaga] == null ){
28.                  alunos[vaga] = novoAluno;
29.                  matricula = true;
30.                  break;
31.              }
32.          if( matricula==false ) return false;
33.          // verifica se o aluno pode se matricular
34.          boolean res = novoAluno.matricular( this );
35.          if( res==false ){
36.              // se ele não pode, remove-o da turma
37.              alunos[vaga] = null;
38.              return false;
39.          }
40.          return true;
41.      }
```

Note o uso do **this**, a referência para o objeto em questão que está executando o método matricular – ou seja, o objeto da turma que está matriculando o novo Aluno

```
45.  class Aluno{
46.      private String nome;
47.      private int matricula;
48.      private Turma[] turmas;
49.      // no máximo pode participar de 10 turmas
50.      public final int MAX_TURMAS = 10;
51.
52.      public Aluno(){
53.          turmas = new Turma[MAX_TURMAS];
54.      }
55.
56.      boolean matricular(Turma novaTurma){
57.          // verifica se o aluno pode se matricular
58.          boolean matricula = false;
59.          for(int i=0; i<turmas.length; i++)
60.              if( turmas[i] == null ){
61.                  turmas[i] = novaTurma;
62.                  matricula = true;
63.                  break;
64.              }
65.          return matricula;
66.      }
67.  }
```

OBS: perceba no exemplo acima que se optou por tornar responsabilidade da Turma repassar sua referência para o método matricular do Aluno, garantindo que o aluno tenha registros das turmas nas quais está matriculado.

Ainda no que tange o quesito de navegabilidade, cada vez deve ser mais usual e de seu entendimento o funcionamento de acesso aos atributos e métodos. Com a introdução dos relacionamentos entre tipos compostos, será uma necessidade usual navegar entre objetos compostos. Se, por exemplo, for de interesse descobrir se um aluno de uma determinada turma possui um sobrenome específico, poderíamos fazer a consulta da seguinte forma:

```
boolean ehOliveira = alunos[index].getNome().contains("Oliveira");
```

alunos[index] — Acessa um objeto Aluno

alunos[index].getNome() — Sendo um Aluno, uso o método que resgata o seu nome – uma String

alunos[index].getNome().contains("Oliveira") — Sendo uma String, usa o método para verificar se ela contém uma substring especificada, retornando um valor booleano como resposta

4. GARBAGE COLLECTOR

Assim como existe um método construtor que é responsável por instanciar um novo objeto de uma determinada classe, também existe em algumas linguagens de programação que utilizam o paradigma orientada a objetos, um método especial denominado de destrutor. Este método é responsável por limpar a memória ocupada por um determinado objeto que passará a não mais existir depois de sua execução. Neste quesito, as linguagens orientadas a objetos optam por diferentes abordagens.

Algumas linguagens orientadas a objetos, como o Java, não permitem a definição de métodos destrutores e utilizam um mecanismo auxiliar chamado de *garbage collector* (coletor de

lixo). Este mecanismo é responsável por limpar objetos da memória que não são referenciadas por nenhum atributo ou variável do programa (veja o exemplo no quadro abaixo no qual um objeto deixa de ser referenciado). Já outras linguagens, como o C++ por exemplo, não utilizam esse mecanismo auxiliar e permitem a definição de métodos destrutores.

```
Carro meu_carro = new Carro( 2010, "Preto", "BluVeículos");
meu_carro = new Carro( 2014, "Vermelho", "BluVeículos");
```

O uso de um *garbage collector* tem seus prós e contras. Por um lado, ele deixa transparente o processo de limpar a memória não utilizada, evitando casos de esquecimento de limpeza ou limpeza incorreta que pode levar a erros no programa. Por outro lado, é um mecanismo que consome e disputa recursos do sistema (processador e memória), podendo não ser adequado para aplicações que possuem exigências temporais rígidas. Portanto lembre-se de verificar se a linguagem orientada a objetos que você for utilizar possui ou não este mecanismo!

Exercícios

1) Crie uma abstração para representar vetores que são constituídos por dois pontos 2D. Cada ponto 2D possui as coordenadas *x* e *y*. A classe vetor deve ter um construtor que recebe dois pontos que representam a sua extremidade e deve manter o atributo módulo que representa a distância entre as extremidades do vetor (essa distância entre pontos deve ser calculada na classe ponto). A classe vetor também deve ter um método que calcula e retorna um vetor resultante de uma operação de soma entre vetores e outro método que retorna o seu ângulo de inclinação em relação ao eixo *x*.

2) Implemente um sistema para gerenciar e armazenar as vendas a vista e a prazo de uma loja de materiais esportivos. Para cada venda deve-se registrar o valor, a data e o horário em que foi realizada, o funcionário que efetuou a venda (precisa-se registrar matrícula e nome), além do cliente comprador (precisa-se registrar CPF, nome, endereço e telefone) caso ele tenha um cadastro na loja (para clientes não cadastrados utilize *null*).

Toda venda realizada pode ser feita de duas formas: a vista ou a prazo. Para ambos os casos, o cliente pode optar por pagar de três formas (enumerador!) distintas: no dinheiro, no cartão de débito ou no cartão de crédito. Toda compra a vista paga no dinheiro tem um desconto de 5% no valor pago. Já para as vendas feitas a prazo, deve-se manter registro do número de parcelas escolhido, o número de parcelas e valor dessas parcelas. O valor das parcelas é calculado como:

```
valorCompra × (1 + nParcelas × 0.025)
```

Implemente duas classes distintas para registrar vendas a vista e a prazo. Ambas as classes devem ter um método construtor que recebe todos os dados referentes à venda e um método show que mostra os detalhes da venda. Para a classe pagamento a prazo, implemente ainda um método pagarParcela que atualiza o valor dos atributos.

3) Implemente um programa para registrar os pedidos em uma restaurante *à la carte*. Para cada pedido devem ser anotado o nome do responsável, qual o número da mesa referente ao pedido (se houver) e a lista de produtos que foram requisitados. Para cada produto é necessário saber apenas o seu código de referencia interno, o preço, a quantidade pedida e se é um lanche, lanche no prato ou uma bebida.

Construa um enumerador para distinguir os produtos entre bebidas e lanches, uma classe para representar os produtos e outra para representar os pedidos. Para a classe de pedidos adicionando e implemente os seguintes métodos:

- `void iniciar( )` – instancia o vetor de produtos (OBS: nunca são pedidos mais do que 50 itens);
- `void adicionarProduto(Produto)` – deve instanciar um novo produto e adicioná-lo no vetor de produtos;
- `int fecharPedido()` – varrer o vetor de produtos e somar o valor total considerando que para lanches no prato deve ser adicionado o valor de 50 centavos ao valor do produto.

Por fim, implemente o método *main* e os métodos auxiliares para: (1) inicialmente instanciar pelo menos dez produtos; (2) mostrar um menu que permite iniciar um pedido, adicionar novos itens/produtos e fechar o pedido, exibindo o valor total.