Unidade 4 - Estruturas de dados básicas: matrizes, ArrayList, HashMap

1        Vetores e Matrizes

Espelho na parede

Descrição gerada automaticamente com confiança baixa

A linguagem Java dá suporte a vetores e matrizes (*arrays*) de diversas formas. Os vetores constituem uma forma muito conveniente de organizar informações em fileira. Por exemplo, podemos formar um vetor com as notas de cinco alunos do seguinte modo:

double notas[] = { 7.8, 8.4, 4.2, 1.8, 6.4 };

Diagrama, Tabela

Descrição gerada automaticamente

 Neste caso notas[0] é a nota do primeiro aluno, isto é, 7.8, notas[1] é a nota do segundo, ou seja, 8.4, e assim por diante.

A utilização de vetores e matrizes em Java envolve três etapas:

**1. Declarar o vetor ou matriz.** Para isto, basta acrescentar um par de colchetes antes ou depois do nome da variável ou do tipo/classe. A quantidade de colchetes indica quantas dimensões vai possuir a matriz: 1 = unidimensional (vetor); 2 = bidimensional; 3 = tridimensional, ... Por exemplo:

int ind[];  // vetor de inteiros

double pesos[][];  // matriz bidimensional de double

int[] nota;  // vetor de inteiros

**2. Criar o vetor.**Reservar espaço de memória, definindo o tamanho. É preciso definir o tamanho do vetor, isto é, a quantidade total de elementos que terá de armazenar. Em seguida é necessário reservar espaço de memória para armazenar os elementos. Isto é feito de maneira simples pelo operador **new**:

ind = new int[10];  // 10 posições, ou seja, 10 valores inteiros

nota = new int[70];  // 70 posições, os índices vão de 0 a 69

pesos = new double[10][20];  // matriz: podemos pensar em 10 linhas com 20 colunas em cada linha

**3. Armazenar elementos no vetor ou matriz**. Para armazenar uma informação em um dos elementos de um vetor ou matriz, é necessário fornecer um índice que indique a posição desse elemento. Por exemplo, para armazenar um valor na quarta posição do vetor **nota**, fazemos o seguinte:

nota[3] = 5.2;

Como podemos observar, os índices começam em zero e vão até o número de posições reservadas, menos um. No vetor *nota* criado acima, os índices válidos vão de 0 até 69. Caso haja a tentativa de atribuir um valor a um elemento cujo índice esteja fora desse intervalo, ocorrerá uma exceção que impedirá a execução do programa. Por isso, é necessário um certo cuidado ao manejar com esses índices, garantindo o perfeito funcionamento do programa.

Existe um atalho para esses três passos quando desejamos criar um vetor com valores atribuídos de modo estático. Foi o que fizemos no primeiro exemplo acima, declarando o vetor **notas** com as notas de cinco alunos. Nesse caso o espaço suficiente para as notas de cinco alunos foi reservado e as notas foram guardadas nas respectivas posições do vetor.

Entretanto, nem sempre é tão fácil assim. Em geral, estaremos interessados em trabalhar com vetores muito maiores, e cujos elementos sejam provenientes de outras fontes, que variam com o tempo. Assim, seremos obrigados a seguir os passos acima.

Eis mais alguns exemplos de vetores e matrizes:

 // 12 primeiros termos da seqüência de Fibonacci:

long Fibonacci[] = {1,1,2,3,5,8,13,34,55,89,144};

// Tabela de sen(n\*pi/6), n = 0,1,2,...5

float seno[] = {0.0000,0.5000,0.8660,1.0000,0.8660,0.5000};

// Tabela de log(1+n), n = 0,2...99:

double tlog[] = new double[100];

for(int n = 0; n < 100; n++) tlog[i] = Math.log(1+n);

// Matriz dos coeficientes

double a[][] = {{1,2,3}, {0,1,3}, {0,0,-1}};

Matrizes bidimensionais são muito comuns e facilmente compreendemos: uma planilha de Excel é um exemplo. Matrizes tridimensionais também visualizamos com facilidade: na forma de um cubo (como o cubo mágico). Mas a partir daí começamos a ter dificuldade em pensar no formato concreto de 4, 5 ou 6 dimensões. Mas se necessário, nossas matrizes em Java podem ter quantas dimensões quisermos.

Isto é possível pois tecnicamente em Java uma matriz é um vetor de vetores. Por isso é possível criar uma matriz de qualquer dimensão (3, 4, 5,...) - apesar de ser difícil de enxergar ou imaginar. No exemplo

double[][] pesos = new double[3][5];

pesos[0][2] = 10.4;

pesos[2][1] = 3.1;

Podemos imaginar essa disposição das linhas e colunas com seus valores:

Tabela, Calendário

Descrição gerada automaticamente

Mas internamente é uma estrutura assim que armazena os valores:

Diagrama

Descrição gerada automaticamente

2        ArrayList

Array redimensionável implementando a interface List. Implementa todas as operações opcionais de uma lista e permite qualquer elemento, incluindo null.

Os métodos básicos da classe ArrayList() em Java estão listados na tabela a seguir:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| **add** (objeto) | Adiciona um objeto na lista |
| **add** (pos, objeto) | Adiciona um objeto na lista na posição indicada por pos |
| **clear**() | Elimina todos os objetos da lista |
| boolean **contains** (objeto) | Informa se um determinado objeto está na lista |
| objeto **get** (pos) | Retorna um objeto de uma determinada posição da lista |
| boolean **isEmpty**() | Informa se a lista está vazia |
| **remove** (objeto) | Retira o objeto da lista |
| objeto **remove** (pos) | Retira o objeto de uma determinada posição da lista |
| int **size**() | Retorna o tamanho da lista |

2.1   Declarando um ArrayList

Um atributo de uma classe ou uma variável local a um método pode ser definido como um ArrayList da seguinte forma:

ArrayList lista;

Toda lista deve ser criada, assim como qualquer objeto. A criação de um objeto lista é feita da seguinte forma.

lista = new ArrayList();

Como todas as coleções, uma lista pode armazenar objetos de qualquer tipo, pois as coleções trabalham com os objetos como sendo *Object* – a classe mais básica do Java. Em uma mesma lista é possível ter instâncias de pessoas, carros, livros, etc. Mas isto pode não ser muito conveniente.

Então é desejável especificar um tipo único de objeto a ser inserido na lista, facilitando a sua manipulação.

ArrayList<Classe> lista;

Desta forma, a lista terá apenas objetos que sejam do tipo da classe especificada.

Exemplo:

ArrayList<Pessoa> listaPessoas;

No exemplo apresentado o atributo listaPessoas, que é um ArrayList, poderá armazenar apenas objetos da classe Pessoa. Para criar a lista também se especifica o tipo de objeto

listaPessoas = new ArrayList<Pessoa>();

2.2   Percorrendo um ArrayList

Há três formas de percorrer um ArrayList:

* Através de um laço *for*, controlado pelo tamanho (*size()*), buscando os objetos pela sua posição (*get(int)*);
* Por um iterador (*Iterator*);
* Por um laço *for-each*.

Na primeira forma, utiliza-se os métodos da própria classe ArrayList para controlar o laço (quantidade de elementos na lista) e para buscar cada objeto.

for (int cont=0; cont < nomeLista.**size()**; contador++) {

    objeto = (ClasseDoObjeto)nomeLista.**get**(cont);

*[comandos da repetição]*

}

Exemplo:

Pessoa alguem;

for (int i=0; i<listaPessoas.size(); i++) {

    alguem = listaPessoas.get(i);

    System.out.println (alguem.getNome());

    System.out.println (alguem.getIdade());

}

O exemplo acima escreve o nome e a idade de todas as pessoas de uma lista de pessoas.

Na segunda forma, é usada a infraestrutura do *framework Collections*, através de um iterador. O funcionamento é simples: pede-se ao ArrayList o objeto de iteração capaz de percorrer sequencialmente todos os elementos da lista. Através dos métodos deste objeto é feita a busca dos elementos.

Iterator nomeIterador = nomeLista.**iterator()**;

while (nomeIterador.**hasNext()**) {

    objeto =  nomeIterador.**next()**;

*[comandos da repetição]*

}

Exemplo:

Iterator<Pessoa> it = listaPessoas.iterator();

Pessoa alguem;

while (it.hasNext()) {

    alguem = it.next();

    System.out.println (alguem.getNome());

    System.out.println (alguem.getIdade());

}

Na terceira forma, o Java oferece uma estrutura de repetição baseado no *for* (*for-each* ou para cada), onde se deve especificar a lista que se deseja percorrer, e um objeto que representa o elemento corrente da lista.

for (ClasseDoObjeto nomeObjeto : nomeLista) {

*[comandos da repetição]*

}

Exemplo:

for (Pessoa alguem: listaPessoas) {

    System.out.println (alguem.getNome());

    System.out.println (alguem.getIdade());

}

3        HashMap

Implementação de mapeamento, ou seja, uma coleção que tem mapeado chaves para valores (objetos). Através de tabela hash, é a melhor implementação da interface Map neste pacote. Provê todas as operações opcionais de Map , permitindo valores  null e chave null .

Os métodos básicos da classe HashMap() em Java estão listados na tabela a seguir:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| **clear**() | Elimina todos os objetos da coleção |
| boolean **containsKey** (chave) | Retorna true caso exista a chave especificada |
| boolean **containsValue** (objeto) | Informa se um determinado objeto está mapeado na coleção |
| objeto **get** (chave) | Retorna um objeto mapeado de acordo com a chave especificada |
| objeto**put** (chave,objeto) | Insere um objeto na coleção, associando a chave ao objeto |
| objeto**remove** (chave) | Retira o objeto da lista, de acordo com a chave |
| int **size**() | Retorna a quantidade de objetos inseridos |

3.1   Declarando um HashMap

Um atributo de uma classe ou uma variável local a um método pode ser definido como um HashMap da seguinte forma:

HashMap tabela;

Todo mapeamento deve ser criado, assim como qualquer objeto. A criação de um objeto de mapeamento é feita da seguinte forma.

tabela = new HashMap();

Como todas as coleções, um mapeamento também pode armazenar objetos de qualquer tipo (*Object*). Novamente é desejável especificar um tipo único de objeto a ser inserido no mapeamento, bem como o tipo de sua chave, facilitando a sua manipulação.

HashMap<ClasseChave, ClasseInserido> tabela;

Desta forma, o mapeamento terá apenas objetos que sejam do tipo da classe especificada, com chaves da classe especificada.

Exemplo:

HashMap<String,Pessoa> tabela;

No exemplo apresentado o *handler* tabela, que é um HashMap, poderá armazenar apenas objetos da classe Pessoa, tendo como chave objetos de String. Para criar o mapeamento também é necessário especificar o tipo de objeto:

tabela = new HashMap<String,Pessoa>();

3.2   Armazenando e buscando objetos

Para inserir um objeto no mapeamento, deve-se usar o método put (chave, objeto).

Exemplo:

Pessoa p = new Pessoa(“João da Silva”, “Rua dos Andradas”, “999.123.878-00”);

tabela.put(“999.123.878-00”, p);

O exemplo acima coloca o objeto de Pessoa (instanciado na 1ª linha) no mapeamento (*HashMap*) tabela, utilizando como chave um objeto de String cujo valor representa o CPF da pessoa.

Desta maneira, quando quiser buscar alguma pessoa dentro do mapeamento, deverá se ter o valor de seu CPF. Utilizando o método get(chave) se obterá o objeto procurado. Caso a chave não seja localizada, null é retornado. Exemplo:

Pessoa pessoa;

pessoa = tabela.get(“123.456.789.10”);

if (pessoa == null)

System.out.println(“Não encontrado”);

else

pessoa.getNome();

3.3   Percorrendo um HashMap

HashMap não foram feitos para ser percorridos sequencialmente (como listas). Como Map não estende Collection, não é previsto em Map uma varredura dos valores contidos no mapeamento. Porém é possível através do método *values()* ter uma visão de coleção dos valores (objetos) mapeados e assim percorrê-los sequencialmente através de um iterador.

Iterator nomeIterador = nomeMapeamento.**values()**.**iterator()**;

while (nomeIterador.**hasNext()**) {

    objeto = (ClasseDoObjeto) nomeIterador.**next()**;

*[comandos da repetição]*

}

Exemplo:

Pessoa alguem;

Iterator<Pessoa> it = tabela.values().iterator();

while (it.hasNext()) {

    alguem = it.next();

    System.out.println (alguem.getNome());

    System.out.println (alguem.getIdade());

}

Outra maneira também é pelo  *for* (*for-each* ou para cada), baseado em uma estrutura temporária criada pelo método *values():*

for (ClasseDoObjeto nomeObjeto : nomeMapeamento.**values()**) {

*[comandos da repetição]*

}

Exemplo:

for (Pessoa alguem: tabela.values()) {

    System.out.println (alguem.getNome());

    System.out.println (alguem.getIdade());

}

Um aspecto importante a ressaltar é que esta visão de coleção criada a partir do método values() não garante nenhuma ordem dos objetos a serem percorridos, nem a ordem das chaves nem a ordem de inserção dos objetos.