03

Armazenamento sequêncial e Vetores

Capítulo 2 - Armazenamento sequencial e Vetores

Com o *Eclipse* aberto, vamos começar o nosso primeiro projeto de estrutura de dados. Como exemplo, estaremos trabalhando com uma universidade e precisamos guardar e recuperar alunos. Dado um aluno, vamos adicioná-lo no fim ou no meio de uma lista, removê-lo, achá-lo a partir de seu número, e assim por diante.

O primeiro passo é modelar a Classe "Aluno". Para isso criamos um novo projeto "ed" e, dentro dele, essa Classe:

```
package ed;

public class Aluno {

}
```

Guardaremos o nome do aluno, que receberemos no próprio Construtor da Classe:

```
public class Aluno {

    private String nome;

    public Aluno(String nome) {
        this.nome = nome;
    }

}
```

Criemos o *getter*:

```
public String getNome() {
    return nome;
}
```

Vamos também implementar os métodos "equals" e "toString", os quais serão muito importantes:

```
@Override
public boolean equals(Object arg0) {
    // TODOAuto-generated method stub
    return super.equals(arg0);
}

@Override
public String toString() {
    // TODOAuto-generated method stub
    return super.toString();
}
```

O "equals" é o método que serve para comparar dois objetos, no caso alunos. Faremos um *casting* do *object* para aluno:

```
@Override
public boolean equals(Object obj) {
    Aluno outro = (Aluno) obj;
    return outro.getNome().equals(this.nome);
}
```

O "toString" retorna o nome do aluno:

```
@Override
public String toString() {
    return nome;
}
```

A primeira estrutura de dados que veremos é o ***Armazenamento Sequencial***. A ideia dessa estrutura é armazenar um aluno atrás do outro. Teremos um conjunto de espaços (*Array*), sendo que: o primeiro aluno fica no primeiro espaço, o segundo aluno no segundo espaço, e assim por diante.

Vamos criar uma nova Classe, chamada "Vetor", na qual implementaremos a estrutura de armazenamento sequencial:

```
package ed;

public class Vetor {

}
```

Precisamos inserir um *array* com 100 posições e implementar os métodos dos comportamentos desse *array*:

```
import java.util.Arrays;

public class Vetor {

    private Aluno[] alunos = new Aluno[100];

    public void adiciona(Aluno aluno) {
        //recebe um aluno
    }

    public Aluno pega(int posicao) {
        //recebe uma posição e devolve o aluno
        return null;
    }

    public void remove(int posicao) {
        //remove pela posição
    }

    public boolean contem(Aluno aluno) {
        //descobre se o aluno está ou não na lista
        return false;
    }
```

```
    public int tamanho() {
        //devolve a quantidade de alunos
        return 0;
    }

    public String toString() {
        //facilitará na impressão
        return Arrays.toString(alunos);
    }
}
```

Os *return*'s foram já inseridos para já podermos compilar o código. Antes de implementar os comportamentos, iremos escrever o método *main* para testar o Vetor, antes mesmo do código existir.

Para isso criaremos a Classe "VetorTeste":

```
package ed;

public class VetorTeste {

    public static void main(String[] args) {

    }
}
```

Método *Adiciona*

O primeiro método que testaremos é o "Adiciona", com dois alunos:

```
    public static void main(String[] args) {
        Aluno a1 = new Aluno("Joao");
        Aluno a2 = new Aluno("Jose");

        Vetor lista = new Vetor();

        lista.adiciona(a1);
        lista.adiciona(a2);

        System.out.println(lista);
    }
```

Ao rodar o programa, ele retorna:

```
[null, null, null, null, null...]
```

Serão 100 *null*'s, então o método ädiciona" está funcionando. Vamos implementá-lo. A ideia é percorrer todo o *array* e, assim que encontrar uma posição nula, o aluno da vez é armazenado nela:

```
    public void adiciona(Aluno aluno) {
        for(int i = 0; i < alunos.length; i++) {
            if(alunos[i] == null) {
                alunos[i] = aluno;
                break;
            }
        }
    }
```

Rodando novamente o teste, ele retorna:

```
[Joao, Jose, null, null, null...]
```

Agora os dois alunos foram inseridos no *array*. Mas perceba que o algoritmo que implementamos não é muito performático, pois quanto maior o número de alunos inseridos no *array*, mais demorado será o método, uma vez que o laço irá percorrer várias vezes os espasços já preenchidos. Vamos tentar melhorá-lo para que não fique dependente da quantidade de elementos na lista.

```
    private Aluno[] alunos = new Aluno[100];
    private int totalDeAlunos = 0;

    public void adiciona(Aluno aluno) {
        this.alunos[totalDeAlunos] = aluno;
        totalDeAlunos++;
    }
```

Método *Tamanho*

O próximo método fácil de implementar é o "Tamanho":

```
    public int tamanho() {
        return totalDeAlunos;
    }
```

Vamos acrescentar no método *main*:

```
        System.out.println(lista.tamanho());
        lista.adiciona(a1);
        System.out.println(lista.tamanho());
        lista.adiciona(a2);
        System.out.println(lista.tamanho());
```

Ele retornará:

```
0
1
2
[Joao, Jose, null, null, null...]
```

A cada iteração ele retorna o tamanho da lista de alunos preenchida.

Método *Contém*

Vamos implementar o método "Contém". Queremos "perguntar" para a lista se um aluno específico está ou não nela.

```
    public boolean contem(Aluno aluno) {

        for(int i = 0; i < totalDeAlunos; i++) {
            if(aluno.equals(alunos[i])) {
                return true;
            }
        }
        return false;
    }
```

Para testar o "*true*", adicionamos no *main:

```
System.out.println(lista.contem(a1));
```

Rodando:

```
0
1
2
[Joao, Jose, null, null, null...]
true
```

Para testar o "*false*" criamos um aluno que não será adicionado na lista:

```
Aluno a3 = new Aluno("Danilo");
System.out.println(lista.contem(a3));
```

Rodando:

```
0
1
2
[Joao, Jose, null, null, null...]
true
false
```

Método *Pega*

Para implementar este método - que nos retorna o nome do aluno na posição que perguntamos - fazemos

```
    public Aluno pega(int posicao) {
        return alunos[posicao];
    }
```

Lembre-se que nosso *array* possui 100 posições. O que aconteceria se perguntássemos sobre o aluno na posição 200? Vamos testar pelo *main*:

```
Aluno x = lista.pega(1);
System.out.println(x);
```

O programa retorna o "Jose", pois é ele que está na posição de número 1. Se escolhermos a posição 200, o programa retorna um erro com a mensagem"*ArrayIndexOutOfBounds*", ou seja, estamos tentando acessar uma posição do *array* que não existe.

Vamos começar a pensar na validação dos dados que vamos passar para o programa. Precisamos que ele retorne, por exemplo, uma mensagem mais amigável, ao invés de um erro. Criaremos um método auxiliar que irá dizer se uma determinada posição está ocupada ou não:

```
private boolean posicaoOcupada(int posicao) {
    posicao >= 0 && posicao < totalDeAlunos;
}
```

No método "pega":

```
    public Aluno pega(int posicao) {

        if(!posicaoOcupada(posicao)) {
            throw new IllegalArgumentException("posiçao invalida");
        }

        return alunos[posicao];
    }
```

Essa parte é muito importante pois é nossa responsabilidade a implementação da estrutura para garantir que ela trate bem qualquer dado errado passado pelo usuário.

Outro método *Adiciona*

Vamos implementar um outro método que, diferentemente do "adiciona" que já vimos, insere um aluno em qualquer posição do *array*:

```
public void adiciona(int posicao, Aluno aluno) {

}
```

Vamos pensar como construir esse método. Imaginemos, no nosso *array* de 100, que as primeiras dez posições já estão preenchidas. Queremos inserir um aluno na terceira posição:

Para isso arrastamos todos os alunos da terceira posicão em diante para a direita e colocamos aquele aluno no buraco que ficou:

Então fazemos:

```
public void adiciona(int posicao, Aluno aluno) {

    for(int i = totalDeAlunos - 1; i >= posicao; i-=1) {
        alunos[i+1] = alunos[i];
    }
    alunos[posicao] = aluno;
    totalDeAlunos++;
}
```

Vamos testá-lo adicionando um aluno:

```
lista.adiciona(1, a3);
System.out.println(lista);
```

Ao que o programa retorna:

```
[Jose, Danilo, Jose, null, null...]
```

De fato, o aluno Danilo foi para a posição 1 empurrando todos para a direita. Porém, da mesma forma que o "pega", precisamos de uma validação:

```
private boolean posicaoValida(int posicao) {
    return posicao >= 0 && posicao <= totalDeAlunos;
}
```

E no método:

```
public void adiciona(int posicao, Aluno aluno) {
```

```
        if(!posicaoValida(posicao)) {
            throw new IllegalArgumentException("posicao invalida");
        }
        for(int i = totalDeAlunos - 1; i >= posicao; i-=1) {
            alunos[i+1] = alunos[i];
        }
        alunos[posicao] = aluno;
        totalDeAlunos++;
    }
```

O método *Remove*

O nosso próximo desafio é o método "remove", que será parecido com o "adiciona", porém pensando inversamente: retiramos o aluno da posição *n* e empurramos para a esquerda todos aqueles que vinham depois dele:

```
public void remove(int posicao) {
    for(int i = posicao; i < this.totalDeAlunos; i++) {
        this.alunos[i] = this.alunos[i+1];
    }
    totalDeAlunos--;
}
```

Testando:

```
lista.remove(1);
System.out.println(lista);
```

Antes estava assim:

```
[Jose, Danilo, Jose, null, null...]
```

E agora:

```
[Joao, Jose, null, null, null...]
```

Redimensionando o *array*

Já implementamos os principais métodos do nosso Vetor. Porém perceba que o tamanho do *array* é constante, valendo 100. Nós queremos que ele seja mutável de acordo com o número de alunos.

Em Java não consrguimos mudar o tamanho de um *array*, então teremos que criar um novo maior e copiar tudo que está no antigo para este. Criemos o método "*garanteEspaço*":

```
private void garanteEspaco() {
    if(totalDeAlunos == alunos.length) {
        Aluno[] novoArray = new Aluno[alunos.length*2];
        for(int i = 0; i < alunos.length; i++) {
            novoArray[i] = alunos[i];
```

```
            }
            this.alunos= novoArray;
        }

    }
```

Falta agora o invocarmos nos dois métodos "adiciona":

```
    public void adiciona(Aluno aluno) {
        this.garanteEspaco();
        ...
    }

    public void adiciona(int posicao, Aluno aluno) {
        this.garanteEspaco();
        ...
    }
```

Agora se adicionarmos mais elementos do que o tamanho do velho *array*, ele será redimencionado em um novo *array*.

para testar essa implementação vamos criar um laço no *main* que adicionará 300 alunos:

```
    for(int i = 0; i < 300; i++) {
        Aluno y = new Aluno("Joao" + i);
        lista.adiciona(y);
    }
    System.out.println(lista);
```

O programa, de fato, retornará uma lista de 300 elementos:

```
    [Joao, Jose, Joao 0, Joao 1, Joao 2, Joao 3...]
```

Nesse exemplo, perceba que houve dois redimensionamentos:

1. Quando passou de 100, dobrando o *array* para 200 posições;
2. Quando passou de 200, dobrando o *array* para 400 posições (tendo 100 delas valores *null*).

O *Array List*

O Java já tem uma implementação de Vetor, é a Classe conhecida por "*ArrayList*". Ela é bem parecida com tudo o que fizemos até agora e funciona como um Armazenamento Sequencial, possuindo os métodos implementados nesta aula:

```
    ArrayList<Aluno> listaDoJava = new ArrayList<Aluno>();
```

Apesar dela existir e facilitar nossa vida, foi importante aprendermos como e o que implementar para criarmos uma estrutura de dados.