CAPÍTULO 4

Vetores

"A melhor maneira de fugir do seu problema é resolvê-lo"
— Robert Anthony

Vamos implementar uma Lista para resolver o problema da listagem de alunos. Lembrando que a interface da Lista já foi definida no capítulo de armazenamento sequencial, seguem as operações que devemos implementar:

1. Adiciona um dado aluno no fim da Lista.
2. Adiciona um dado aluno em uma dada posição.
3. Pega o aluno de dada posição.
4. Remove o aluno de dada posição.
5. Verifica se um dado aluno está armazenado.
6. Informa o número de alunos armazenados.

Ainda falta definir como os alunos serão armazenados. Como queremos manter muitos alunos vamos alocar um grande espaço de memória sequencial com capacidade para guardar uma certa quantidade de alunos, talvez 100 alunos por enquanto seja razoável.

Para facilitar o acesso aos alunos, dividiremos o espaço de memória alocado em pequenos pedaços idênticos. Cada pedaço armazenará um aluno. Além disso, vamos indexar (numerar) os pequenos pedaços para ser fácil recuperar um aluno.

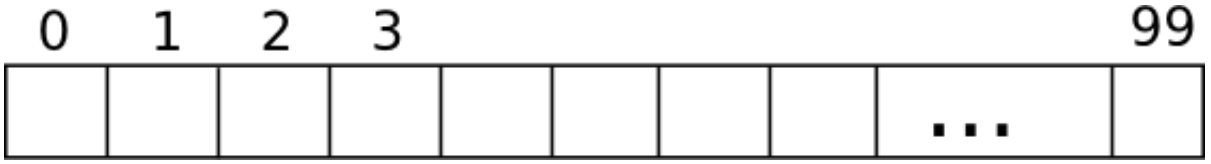

Figura 4.9: Array

Praticamente todas as linguagens de programação têm um recurso similar ao que descrevemos acima. Em Java, este recurso é chamado de **Array**.

Um array é uma porção de memória fixa e sequencial dividida em pedaços idênticos indexados a partir do `0`. Em cada posição do array, podemos guardar um aluno. Na verdade, cada posição pode guardar uma **referência** para um objeto do tipo `Aluno`.

A capacidade de um array é fixa e deve ser informada no momento da criação do array. Não é possível redimensionar um array em Java, teremos de contornar isso mais adiante.

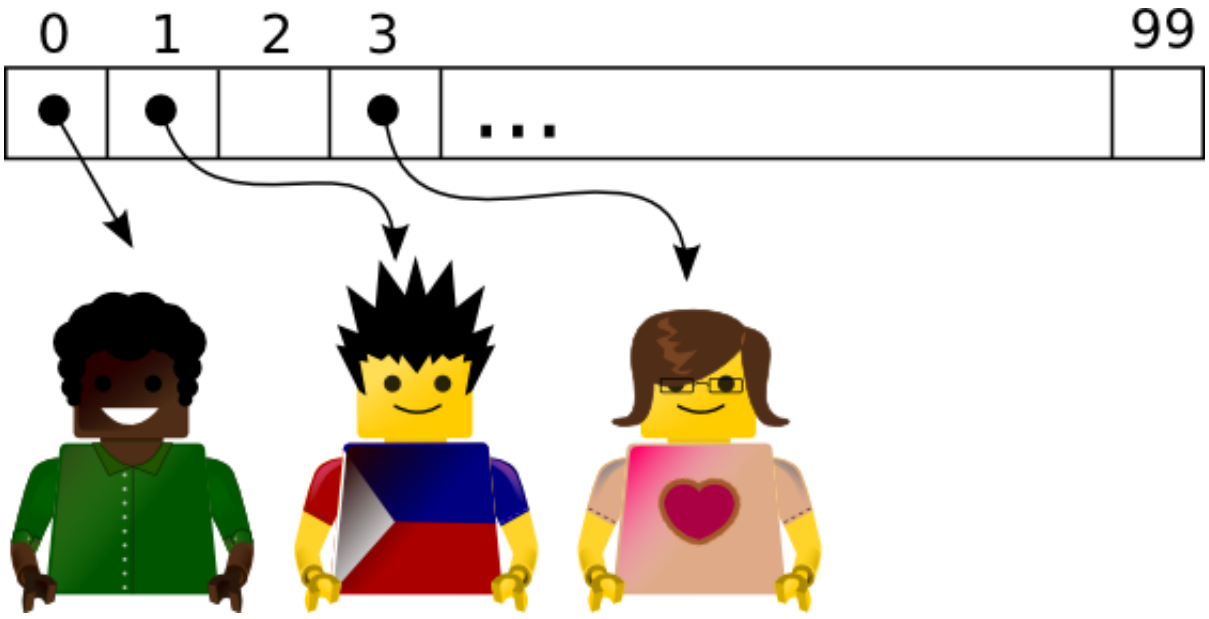

Figura 4.10: Array de Referências

Uma Lista implementada com array muitas vezes é denominada **Vetor**. Então criaremos uma classe chamada `Vetor` que armazena os alunos em array e tem todas as operações de uma Lista, encapsulando o acesso a esta Array.

```
public class Vetor {

  // Declarando e Inicializando um array de Aluno com capacidade 100.
  private Aluno[] alunos = new Aluno[100];

  public void adiciona(Aluno aluno) {
    // implementação
  }

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

```
    // implementação
  }

  public Aluno pega(int posicao) {
    // implementação
  }

  public void remove(int posicao) {
    // implementação
  }

  public boolean contem(Aluno aluno) {
    // implementação
  }

  public int tamanho() {
    // implementação
  }

  public String toString() {
    return Arrays.toString(alunos);
  }
}
```

O código do `Vetor` acima não compila porque alguns de seus métodos não são **void**, obrigando a você retornar alguma coisa de um certo tipo. Se você quiser fazer com que o código acima compile, adicione alguns **return**s apropriados, como **0**, **null**s e **false**s.

4.1 – Os testes primeiro

Como temos a interface do Vetor já bem definida, podemos criar alguns testes antes mesmo de começar a implementar o código. Criaremos testes para cada uma das operações.

Adiciona no fim da lista

Teste:

```
public class TesteAdicionaNoFim {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();

    a1.setNome("João");
    a2.setNome("José");

    Vetor lista = new Vetor();

    lista.adiciona(a1);
```

```
    lista.adiciona(a2);

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

Saída:

```
[João, José]
```

Adiciona em uma dada posição

Teste:

```
public class TesteAdicionaPorPosicao {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();
    Aluno a3 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");
    a3.setNome("Ana");

    Vetor lista = new Vetor();

    lista.adiciona(a1);
    lista.adiciona(0, a2);
    lista.adiciona(1, a3);

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

Saída:

```
[Paulo, Ana, Rafael]
```

Pegar um aluno por posição

Teste:

```
public class TestePegaPorPosicao {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");

    Vetor lista = new Vetor();

    lista.adiciona(a1);
```

```
    lista.adiciona(a2);

    Aluno aluno1 = lista.pega(0);
    Aluno aluno2 = lista.pega(1);

    System.out.println(aluno1);
    System.out.println(aluno2);
  }
}
```

Saída:

```
Rafael

Paulo
```

Remover um aluno por posição

Teste:

```
public class TesteRemovePorPosicao {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");

    Vetor lista = new Vetor();

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

    lista.remove(0);

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

Saída:

```
[Paulo]
```

Verificar se a lista contem um dado aluno

Teste:

```
public class TesteContemAluno {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();

    a1.setNome("Rafael");
```

```
    a2.setNome("Paulo");

    Vetor lista = new Vetor();

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

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

    System.out.println(lista.contem(a2));

    Aluno aluno = new Aluno();
    aluno.setNome("Ana");

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

Saída:

```
true

true

false
```

Informar o tamanho da lista

Teste:

```
public class TesteTamanhoLista {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();
    Aluno a3 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");

    Vetor lista = new Vetor();

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

    System.out.println(lista.tamanho());

    lista.adiciona(a3);

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

Saída:

```
2
```

Estes testes podem ser rodados a medida que preenchemos nosso `Vetor` com sua respectiva implementação. Em uma aplicação profissional Java, criaríamos **testes de unidade**, utilizando bibliotecas auxiliares, como JUnit ou TestNG, para facilitar a escrita destes mesmos testes.

O desenvolvimento dirigido a testes (Test Driven Development, TDD) é uma prática que ganha cada vez mais força, onde escreveríamos primeiro os testes das nossas classes, antes mesmo de começar a escrever a sua classe. O intuito disso é que você acaba apenas criando as classes e os métodos que realmente necessita, e eles já estão testados! O design da classe também costuma sair mais simples, pois uma classe com muitas dependências e acoplamento é difícil ser testada.

4.2 - Operações em vetores

Na seqüência, implementaremos cada uma das operações de uma Lista.

4.3 - Adicionar no fim da Lista

Esta operação receberá um aluno e vai guardá-lo no fim da Lista. Então, precisamos saber onde é o fim da Lista. A dificuldade em descobrir a última posição depende de como os alunos serão armazenados no array.

Há duas possibilidades: ou guardamos os alunos compactados a esquerda do array ou não. No primeiro caso, será bem mais fácil achar a última posição da Lista. Além disso, o índice do array será o mesmo índice da Lista.

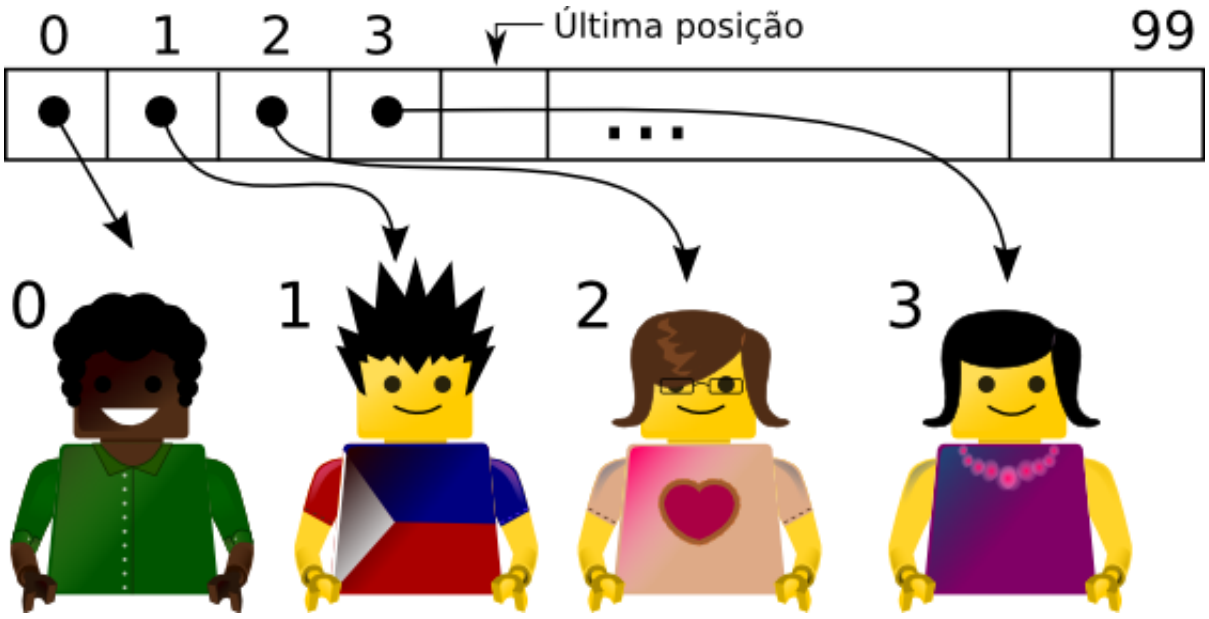

Figura 4.1: Array compactado a esquerda

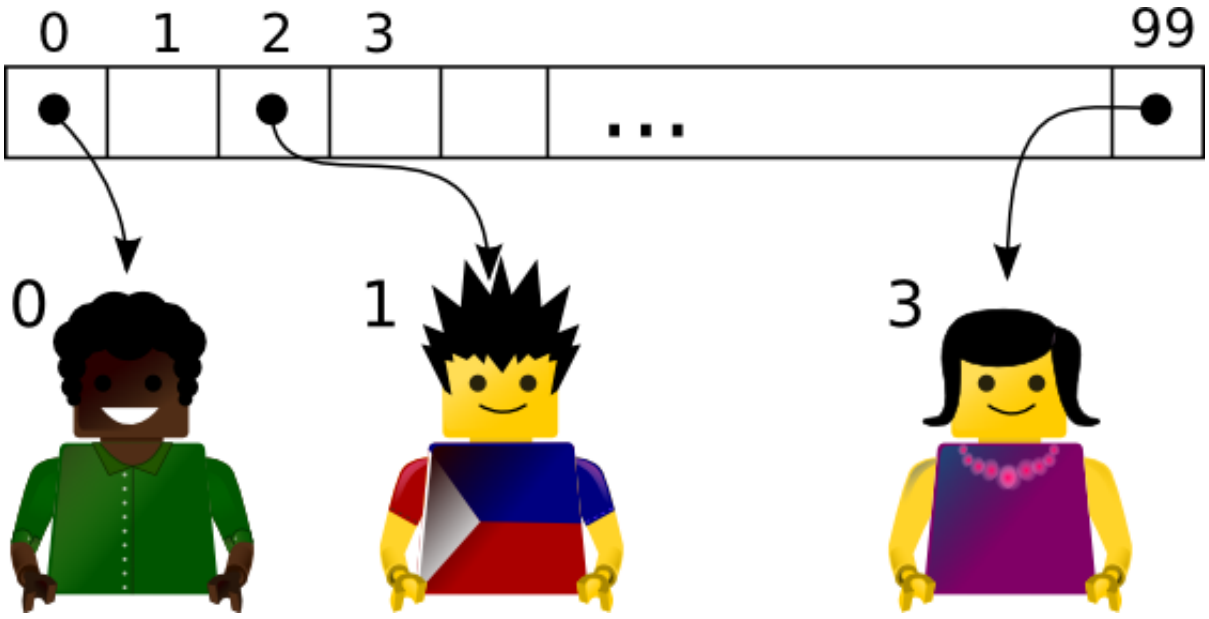

Figura 4.2: Array não compactado a esquerda

Então, vamos definir que sempre os alunos serão armazenados compactados a esquerda no array, sem "buracos", que além de tudo economiza espaço.

Para achar a última posição da Lista ou a primeira posição vazia basta percorrer o array da esquerda para a direita até achar um valor **null** (lembrando que os arrays do Java guardam referências, e o valor padrão para referências é **null**). Achado a primeira posição vazia é só guardar o aluno nela.

Para percorrer o array usaremos um laço. Perceba que usamos o controlador de laço **break** para parar o **for** quando o aluno já foi adicionado.

```
public class Vetor {

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

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

Neste ponto já seria interessante testar com o `TesteAdicionaNoFim`.

Fazendo uma análise simples deste método, é fácil ver que quanto mais alunos forem inseridos pior será o desempenho deste método. Por exemplo, se a Lista tem 50 alunos, o laço vai rodar 51 vezes para achar a primeira posição vazia.

Já que o consumo de tempo deste método piora proporcionalmente na medida que o número de elementos que existem na Lista aumenta, dizemos que o consumo é **linear**.

Será que tem alguma maneira de melhorar este consumo de tempo?

Uma pequena modificação é capaz de melhorar muito o desempenho do `adiciona(Aluno)`. Perceba que percorremos o array somente para procurar a primeira posição vazia. Mas isso é realmente necessário?

Vamos lembrar que o nosso array está compactado a esquerda então o índice da primeira posição vazia é igual a quantidade de elementos. Logo, se guardarmos a quantidade de elementos em algum lugar então no momento de adicionar um aluno já saberíamos qual é o índice da primeira posição vazia.

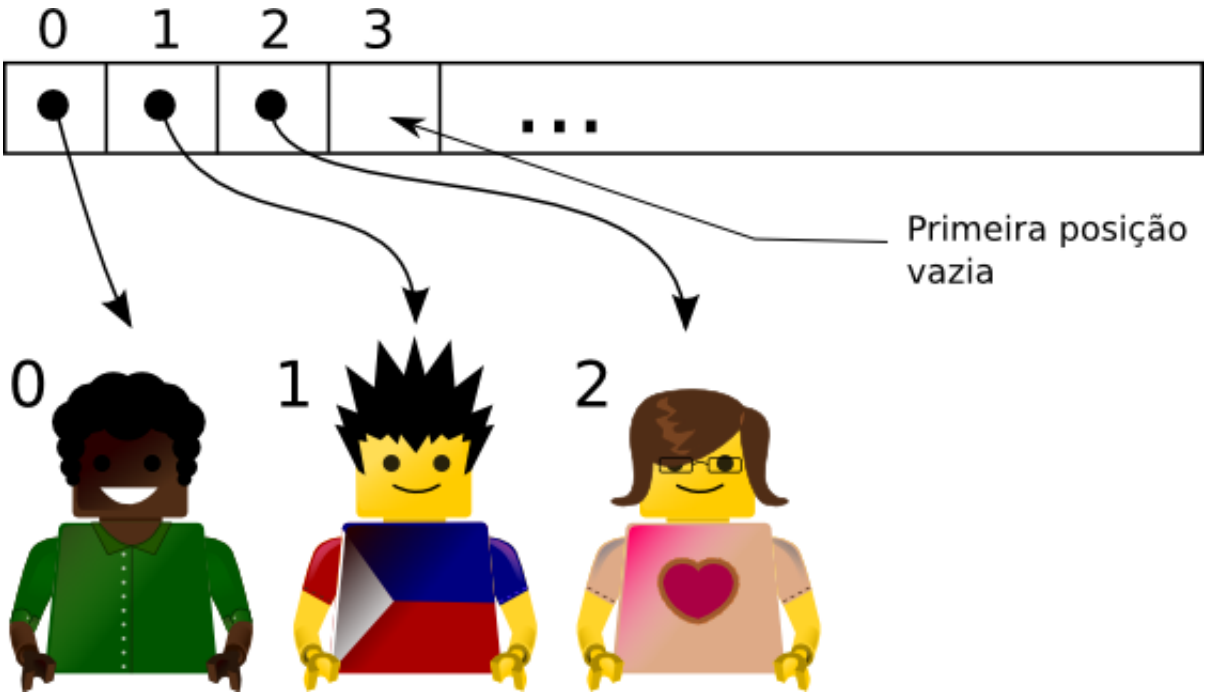

Figura 4.3: Quantidade de elementos = Índice da primeira posição vazia

Para guardar a quantidade de alunos existentes na Lista definiremos um atributo na classe `Vetor` do tipo `int`.

```
public class Vetor {

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

  private int totalDeAlunos = 0;

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

Agora, o consumo de tempo do método é sempre o mesmo não importa quantos alunos estejam armazenados. Neste caso, dizemos que o consumo é **constante**.

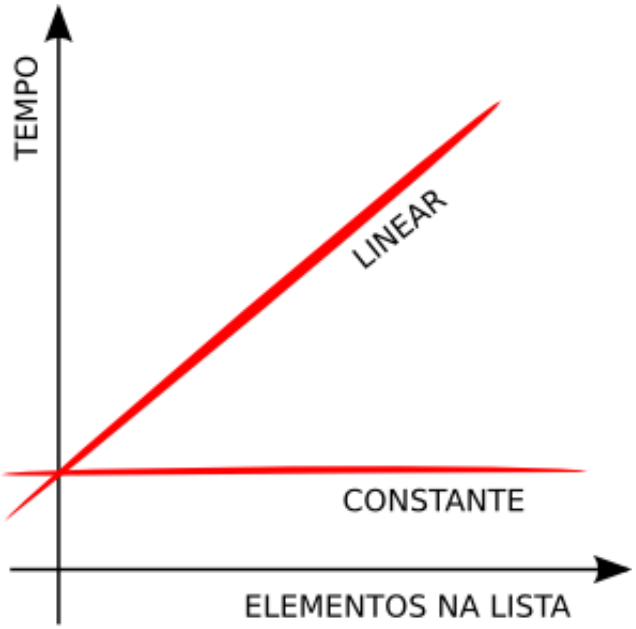

Figura 4.4: Consumo Linear VS Consumo Constante

Um ponto interessante de enxergar aqui é que modificamos nossa **implementação** sem alterar nossa **interface**, e conseguimos deixar o método `adiciona` mais rápido. Com isso, se alguém já estivesse usando nossa classe `Vetor` antiga, poderia substituir pela nova sem alterar nenhuma outra linha de código. Se o acesso a nossa array fosse público, teríamos problemas nos códigos das pessoas que estão usando a nossa classe. Graças ao encapsulamento e a boa definição de nossa interface, evitamos ter de reescrever uma quantidade grande de código.

Para verificar se o método continua funcionando devemos executar novamente o `TesteAdicionaNoFim`.

4.4 - O método toString() para o Vetor

Vamos reescrever o método `toString()` para visualizar facilmente o conteúdo da Lista. Utilizamos a classe `StringBuilder` para construir a String que mostrará os elementos da Lista.

```
public String toString() {
  if (this.totalDeAlunos == 0) {
    return "[]";
  }

  StringBuilder builder = new StringBuilder();
  builder.append("[");
```

```
    for (int i = 0; i < this.totalDeAlunos - 1; i++) {
      builder.append(this.alunos[i]);
      builder.append(", ");
    }

    builder.append(this.alunos[this.totalDeAlunos - 1]);
    builder.append("]");

    return builder.toString();
}
```

4.5 – Informar o tamanho da Lista

Esta operação ficou muito simples de ser implementada porque a classe `Vetor` tem um atributo que guarda a quantidade de alunos armazenados. Então, basta devolver o valor do `totalDeAlunos`. Perceba que o consumo de tempo será constante: não há laços.

```
public class Vetor {

  ...
  private int totalDeAlunos = 0;

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

Se não tivéssemos criado o atributo `totalDeAlunos` o método `tamanho()` teria que fazer um laço para percorrer o array inteiro e contar quantas posições estão ocupadas. Ou seja, o desempenho seria linear que é muito pior que constante.

Não podemos esquecer de rodar o teste para o tamanho da Lista.

4.6 – Verificar se um aluno está presente no vetor

Nesta operação, precisamos comparar o aluno dado com todos os alunos existentes na Lista. Para implementar esta funcionalidade faremos um laço.

```
public class Vetor {

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

  private int totalDeAlunos = 0;

  ...
  public boolean contem(Aluno aluno) {
    for (int i = 0; i < this.alunos.length; i++) {
      if (aluno == this.alunos[i]) {
        return true;
      }
    }
    return false;
  }
}
```

Neste método, se o aluno procurado for encontrado então o valor **true** é devolvido. Se a array acabar e o aluno não for encontrado, significa que ele não está armazenado logo o método deve devolver falso. A capacidade do array é obtida pelo atributo `length`.

O nosso método é ineficiente quando a Lista tem poucos elementos. Perceba que ele sempre percorre o array todo. Não é necessário percorrer o array inteiro basta percorrer as posições ocupadas, ou seja, o laço tem que ir até a última posição ocupada. Nós podemos obter a última posição ocupada através do atributo `totalDeAlunos`.

```
public class Vetor {

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

  private int totalDeAlunos = 0;

  ...
  public boolean contem(Aluno aluno) {
    for (int i = 0; i < this.totalDeAlunos; i++) {
      if (aluno == this.alunos[i]) {
        return true;
      }
    }
    return false;
  }
}
```

Nós estamos comparando os alunos com o operador `==`. Este operador compara o conteúdo das variáveis. No Java, as variáveis de tipos não primitivos, como o tipo `Aluno`, guardam referências para objetos. Então, na verdade, estamos comparando as referências e não os objetos propriamente.

Para comparar objetos devemos usar o método `equals(Object)`. Lembrando que rescrevemos este método para considerar que dois objetos do tipo `Aluno` são iguais quando os seus atributos `nome` são iguais.

```
public class Vetor {

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

  private int totalDeAlunos = 0;

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

Aqui deveríamos executar o teste do `contem`.

4.7 – Pegar o aluno de uma dada posição do array

Esta operação parece bem simples, ela deve simplesmente acessar e devolver o aluno da posição desejada.

```
public class Vetor {

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

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

Mas não é tão simples assim: esquecemos de considerar a possibilidade do usuário pedir por uma posição inválida. Se desconsiderarmos esta possibilidade vamos correr o risco de acessar uma posição vazia ou inexistente. Então, antes de acessar a posição, vamos verificar se ela está ocupada.

Será criado o método `posicaoOcupada(int)` que devolve verdadeiro se a posição estiver ocupada, e falso caso contrário. Uma posição é válida se ela pertence ao intervalo fechado [0, this.totalDeAlunos - 1].

```
public class Vetor {
  private Aluno[] alunos = new Aluno[100];

  private int totalDeAlunos = 0;

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

A maneira que o método `posicaoOcupada(int)` foi implementado é bem interessante pois ela não usa `if`s.

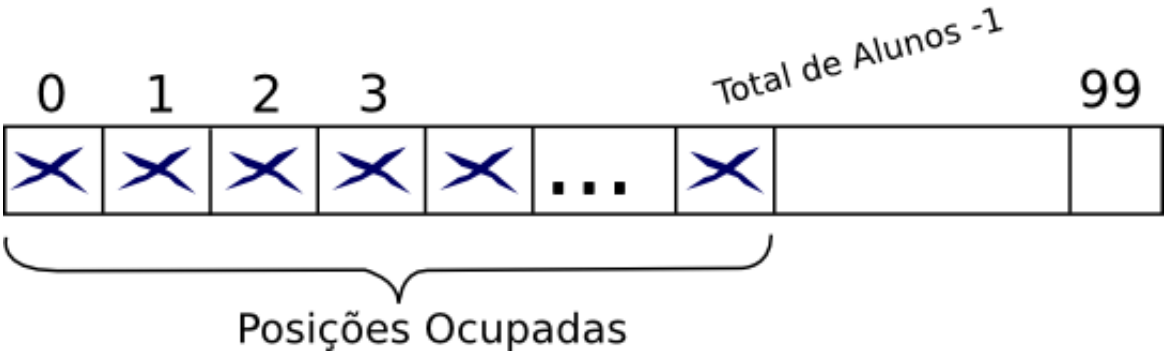

Figura 4.5: Posições ocupadas

É importante observar que o método `posicaOcupada(int)` deve ser privado pois não deve ser acessado por quem está usando a classe `Vetor`. Desta forma, o sistema fica mais encapsulado.

Agora, o método `pega(int)` pode invocar o `posicaoOcupada(int)` para saber se a posição está ocupada ou não. Caso a posição seja válida, o `pega(int)` devolve o aluno correspondente e caso contrário, ele lança uma exceção (optamos por `IllegalArgumentException`).

As exceções são uma maneira de informar para quem chamou um método que algo aconteceu de maneira diferente da comum.

```
public class Vetor {
  private Aluno[] alunos = new Aluno[100];

  private int totalDeAlunos = 0;
```

```
  ...
  public Aluno pega(int posicao) {
    if (!this.posicaoOcupada(posicao)) {
      throw new IllegalArgumentException("Posição inválida");
    }
    return this.alunos[posicao];
  }

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

Feito o método devemos testá-lo com o teste definido anteriormente.

4.8 – Adicionar um aluno em uma determinada posição do array

A operação de adicionar um aluno em uma determinada posição é mais delicada. Primeiro, precisamos verificar se a posição faz sentido ou não. Só podemos adicionar um aluno em alguma posição que já estava ocupada ou na primeira posição vazia da Lista.

Para verificar se podemos adicionar um aluno em uma dada posição devemos testar se a posição está no intervalo `[0, this.totalDeAlunos]`. Vamos criar um método para isolar esta verificação.

```
public class Vetor {
  private Aluno[] alunos = new Aluno[100];

  private int totalDeAlunos = 0;

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

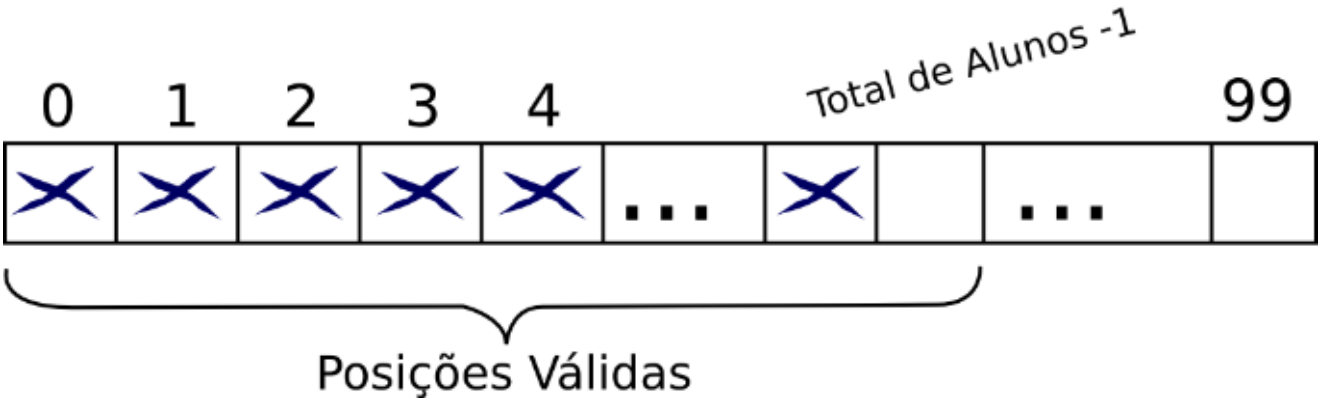

Figura 4.6: Posições válidas

Após verificar se podemos adicionar o aluno na posição dada, devemos tomar cuidado para não colocar um aluno sobre outro. É preciso deslocar todos os alunos a "direita" da posição onde vamos inserir uma vez para a "frente". Isso abrirá um espaço para guardar a referência para o aluno novo.

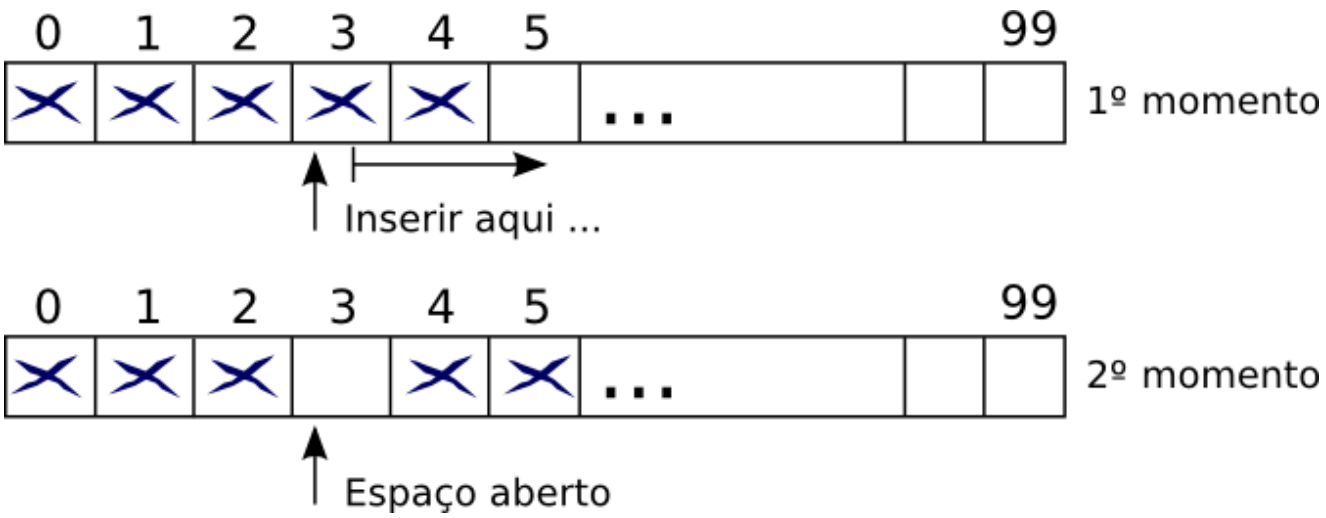

Figura 4.7: Deslocamento para a direita

```
public class Vetor {
  private Aluno[] alunos = new Aluno[100];

  private int totalDeAlunos = 0;

  ...
  public void adiciona(int posicao, Aluno aluno) {
    if (!this.posicaoValida(posicao)) {
      throw new IllegalArgumentException("Posição inválida");
    }

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

    this.alunos[posicao] = aluno;
    this.totalDeAlunos++;
  }

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

Quanto este método consome de tempo? Depende! Se for a última posição, consumirá tempo constante. No caso de ser a primeira posição, ele terá de deslocar **todos** os elementos para a direita, consumindo tempo linear em relação

ao número de elementos distantes. É comum calcularmos o tempo consumido de um método pensando sempre no pior caso, então diremos que o método que adiciona em qualquer posição de nosso Vetor consome tempo linear.

Agora é um ótimo momento para testar. Podemos rodar o teste de adicionar por posição.

4.9 - Remover um aluno de uma dada posição

Inicialmente, precisamos verificar se a posição está ocupada ou não. Afinal, não faz sentido remover algo que não existe. Para saber se a posição está ocupada ou não podemos usar o método `posicaoOcupada(int)`.

Se a posição estiver ocupada então podemos remover o aluno. Além disso, precisamos deslocar os alunos que estavam a direita daquele que removemos uma vez para esquerda para fechar o "buraco" aberto pela remoção.

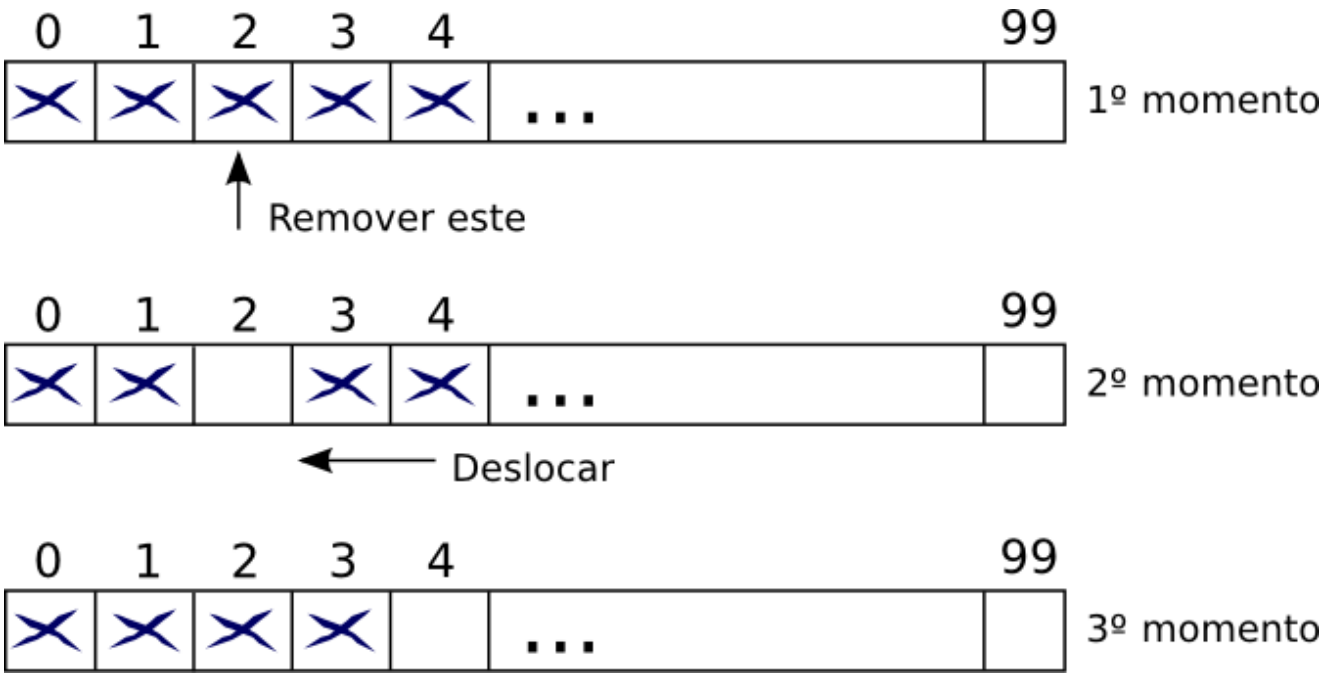

Figura 4.8: Deslocamento para a esquerda

```
public class Vetor {
  private Aluno[] alunos = new Aluno[100];

  private int totalDeAlunos = 0;

  ...
  public void remove(int posicao) {
    if (!this.posicaoOcupada(posicao)) {
      throw new IllegalArgumentException("Posição inválida");
    }
    for (int i = posicao; i < this.totalDeAlunos - 1; i++) {
      this.alunos[i] = this.alunos[i + 1];
    }
    this.totalDeAlunos--;
  }

  private boolean posicaoOcupada(int posicao) {
    return posicao < this.alunos.length && posicao >= 0;
  }
}
```

Você sabe dizer quanto tempo este método consome?

E agora falta executar o teste para esta operação.

4.10 – Alocação Dinâmica

Há um grande problema na implementação apresentada até o momento. Imagine que o vetor já contenha 100 alunos. Ao adicionar mais um aluno ocorreria um erro pois o array foi criado com 100 posições e o método `adiciona()` tentaria inserir um aluno em uma posição que não existe: o java lançará um exceção.

Para resolver isso, podemos tentar inicializar o array com um tamanho maior. Isso não resolveria problema. Por exemplo, se o tamanho da array fosse 200 em vez de 100, no momento que fosse inserido o aluno número 201, ocorreria novamente um erro.

Uma abordagem mais eficiente seria cada vez que o array ficar cheio alguma providência seja tomada, como, por exemplo, dobrar o tamanho dele.

Vamos criar um método que tem como tarefa verificar se o array está cheio. Caso estiver cheio, ele criará um novo array com o dobro do tamanho do antigo e moverá os alunos do array antigo para o novo.

```
public class Vetor {
  private Aluno[] alunos = new Aluno[100];

  private int totalDeAlunos = 0;

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

O risco de tentar adicionar um aluno sem ter posição disponível só ocorre, evidentemente, nos métodos de adicionar aluno. Então, para evitar este problema, vamos verificar se existe espaço disponível antes de adicionar um aluno.

```
public class Vetor {
  private Aluno[] alunos = new Aluno[100];

  private int totalDeAlunos = 0;

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

  public void adiciona(int posicao, Aluno aluno) {
    this.garantaEspaco();
    if (!this.posicaoValida(posicao)) {
      throw new IllegalArgumentException("Posição inválida");
    }

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

    this.alunos[posicao] = aluno;
    this.totalDeAlunos++;
  }
}
```

O método `garantaEspaco()` só é útil dentro da classe `Vetor`, ou seja, não deve ser disponibilizado para o usuário então ele deve ser um método **private**.

4.11 – Generalização

A implementação de vetor feita até agora funciona muito bem para armazenar alunos. Porém, não serve para armazenar nenhum outro tipo de objeto. Nossa estrutura de dados está muito atrelada ao tipo de dado que ela armazena (Alunos). Se amanhã ou depois precisarmos de uma Lista de carro ou uma Lista de computador teríamos que implementar novamente o Vetor.

Em vez de colocarmos um array de `Aluno` na classe `Vetor` vamos colocar um array de `Object`. Assim, estamos generalizando a nossa estrutura de dados. Desta forma, poderemos armazenar qualquer tipo de objeto.

```
public class Vetor {

  private Object[] objetos = new Object[100];

  private int totalDeObjetos = 0;

  public void adiciona(Object objeto) {
    this.garantaEspaco();
    this.objetos[this.totalDeObjetos] = objeto;
    this.totalDeObjetos++;
  }

  public void adiciona(int posicao, Aluno aluno) {
    this.garantaEspaco();
    if (!this.posicaoValida(posicao)) {
      throw new IllegalArgumentException("Posição inválida");
    }

    for (int i = this.totalDeObjetos - 1; i >= posicao; i--) {
      this.objetos[i + 1] = this.objetos[i];
    }

    this.objetos[posicao] = aluno;
    this.totalDeObjetos++;
  }

  public Object pega(int posicao) {
    if (!this.posicaoOcupada(posicao)) {
      throw new IllegalArgumentException("Posição inválida");
    }
    return this.objetos[posicao];
  }

  public void remove(int posicao) {
    if (!this.posicaoOcupada(posicao)) {
      throw new IllegalArgumentException("Posição inválida");
    }
    for (int i = posicao; i < this.totalDeObjetos - 1; i++) {
      this.objetos[i] = this.objetos[i + 1];
```

```
    }
    this.totalDeObjetos--;
  }

  public boolean contem(Aluno aluno) {
    for (int i = 0; i < this.totalDeObjetos; i++) {
      if (aluno.equals(this.objetos[i])) {
        return true;
      }
    }
    return false;
  }

  public int tamanho() {
    return this.totalDeObjetos;
  }

  private boolean posicaoOcupada(int posicao) {
    return posicao >= 0 && posicao < this.totalDeObjetos;
  }

  private boolean posicaoValida(int posicao) {
    return posicao >= 0 && posicao <= this.totalDeObjetos;
  }

  private void garantaEspaco() {
    if (this.totalDeObjetos == this.objetos.length) {
      Object[] novaArray = new Object[this.objetos.length * 2];
      for (int i = 0; i < this.objetos.length; i++) {
        novaArray[i] = this.objetos[i];
      }
      this.objetos = novaArray;
    }
  }
}
```

No Java todas as classes herdam, diretamente ou indiretamente, da classe `Object`. Então, um objeto de qualquer tipo pode ser referenciado com uma variável do tipo `Object`. Este conceito de referenciar um mesmo objeto de várias maneiras (`Aluno` ou `Object`) é chamado de **polimorfismo**.

O que ganhamos com esta generalização foi um forte reaproveitamento da classe `Vetor`, porém na hora do uso perdemos a segurança da tipagem do java. Isso acarretará na necessidade do uso de **casting**.

```
Vetor lista = new Vetor();

// Inserindo uma String
lista.adiciona("Joao Silva");

// Fazendo um casting de String para Aluno. Erro de EXECUÇÃO.
Aluno aluno = (Aluno) lista.pega(0);
```

Existe uma outra maneira de fazer a mesma classe sem essa desvantagem de usar castings, que é criar uma classe parametrizada, um recurso existente no Java a partir da versão 5.

4.12 – API do Java

Na biblioteca do Java, há uma classe que implementa a estrutura de dados que foi vista neste capítulo, esta classe chama-se **ArrayList** e será testada pelo código abaixo.

```
public class Teste {

  public static void main(String[] args) {

    ArrayList vetor = new ArrayList();

    Aluno aluno1 = new Aluno();
    Aluno aluno2 = new Aluno();
    Aluno aluno3 = new Aluno();

    vetor.add(aluno1);
    vetor.add(aluno2);
    vetor.add(0, aluno3);

    int tamanho = vetor.size();

    if (tamanho != 3) {
      System.out.println("Erro: O tamanho da lista está errado.");
    }

    if (!vetor.contains(aluno1)) {
      System.out
          .println("Erro: Não achou um aluno que deveria estar na lista");
    }
```

```
    vetor.remove(0);
    tamanho = vetor.size();

    if (tamanho != 2) {
      System.out.println("Erro: O tamanho da lista está errado.");
    }

    if (vetor.contains(aluno3)) {
      System.out
          .println("Erro: Achou um aluno que não deveria estar na lista");
    }
  }
}
```

A classe `Vector` é muito similar a `ArrayList`, a grande diferença é que `ArrayList` não é segura para ser compartilhada entre várias threads simultaneamente sem o devido cuidado. Dizemos que `Vector` é thread safe, mas isso tem um custo, e é por isso que evitamos usar `Vector` e preferimos usar `ArrayList` sempre que possível.

Para evitar fazer casting de objetos, podemos utilizar o recurso de generics do Java 5. A utilização de generics é bem simples, a gente deve informar que o nossa Lista vai guardar alunos. Isso é feito como mostra o código a seguir:

```
public class Teste {
  public static void main(String[] args) {

    ArrayList vetorSemGenerics = new ArrayList();
    ArrayList<Aluno> vetorComGenerics = new ArrayList<Aluno>();

    Aluno aluno = new Aluno();

    vetorSemGenerics.add(aluno);
    vetorComGenerics.add(aluno);

    Aluno a1 = (Aluno) vetorSemGenerics.get(0);
    Aluno a2 = vetorComGenerics.get(0);
  }
}
```

Com o generics temos uma segurança em tempo de compilação em relação a tipagem dos objetos. Se tentarmos adicionar um objeto que não é do tipo `Aluno` um erro de **compilação** acontecerá.

```
ArrayList<Aluno> vetorComGenerics = new ArrayList<Aluno>();
vetorComGenerics.add("Rafael"); // erro de compilação
```

Qual a vantagem de um erro de compilação sobre um erro de execução? O de execução acontece quando o usuário está do lado do computador. O de compilação acontece quando o programador está no computador.

4.13 – Exercícios: Vetores

1. Crie a classe `Vetor` no pacote **br.com.caelum.ed.vetores** com as assinaturas dos métodos vistos neste capítulo e com um atributo do tipo array de Aluno inicializado com `100000` posições.

```
package br.com.caelum.ed.vetores;

import br.com.caelum.ed.Aluno;

public class Vetor {

  // Declarando e Inicializando um array de Aluno com capacidade 100.
  private Aluno[] alunos = new Aluno[100000];

  public void adiciona(Aluno aluno) {
    // implementação
  }

  public void adiciona(int posicao, Aluno aluno) {
    // implementação
  }

  public Aluno pega(int posicao) {
    // implementação
    return null;
  }

  public void remove(int posicao) {
    // implementação
  }

  public boolean contem(Aluno aluno) {
    // implementação
    return false;
  }

  public int tamanho() {
    // implementação
    return 0;
  }
}
```

2. Escreva os testes de unidade vistos neste capítulo. Coloque os testes no pacote **br.com.caelum.ed.vetores.testes**.

Teste:

```
public class TesteAdicionaNoFim {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();
```

```
    a1.setNome("Rafael");
    a2.setNome("Paulo");

    Vetor lista = new Vetor();

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

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

Saída:

```
[Rafael, Paulo]
```

Teste:

```
public class TesteAdicionaPorPosicao {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();
    Aluno a3 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");
    a3.setNome("Ana");

    Vetor lista = new Vetor();

    lista.adiciona(a1);
    lista.adiciona(0, a2);
    lista.adiciona(1, a3);

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

Saída:

```
[Paulo, Ana, Rafael]
```

Teste:

```
public class TestePegaPorPosicao {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");

    Vetor lista = new Vetor();
```

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

    Aluno aluno1 = lista.pega(0);
    Aluno aluno2 = lista.pega(1);

    System.out.println(aluno1);
    System.out.println(aluno2);
  }
}
```

Saída:

```
Rafael

Paulo
```

Teste:

```
public class TesteRemovePorPosicao {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");

    Vetor lista = new Vetor();

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

    lista.remove(0);

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

Saída:

```
[Paulo]
```

Teste:

```
public class TesteContemAluno {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");

    Vetor lista = new Vetor();

    lista.adiciona(a1);
```

```
    lista.adiciona(a2);

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

    System.out.println(lista.contem(a2));

    Aluno aluno = new Aluno();
    aluno.setNome("Ana");

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

Saída:

```
true

true

false
```

Teste:

```
public class TesteTamanhoLista {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();
    Aluno a2 = new Aluno();
    Aluno a3 = new Aluno();

    a1.setNome("Rafael");
    a2.setNome("Paulo");

    Vetor lista = new Vetor();

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

    System.out.println(lista.tamanho());

    lista.adiciona(a3);

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

Saída:

```
2

3
```

3. Uma vez definida a interface do Vetor poderíamos programar os testes. Para os testes vamos reescrever o `toString()`.

```
public String toString() {
```

```
  if (this.totalDeAlunos == 0) {
    return "[]";
  }

  StringBuilder builder = new StringBuilder();
  builder.append("[");

  for (int i = 0; i < this.totalDeAlunos - 1; i++) {
    builder.append(this.alunos[i]);
    builder.append(", ");
  }

  builder.append(this.alunos[this.totalDeAlunos - 1]);
  builder.append("]");

  return builder.toString();
}
```

4. Implemente o método `adiciona(Aluno)` da primeira maneira vista neste capítulo.

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

Faça um teste para calcular o tempo gasto. Este teste deve adicionar `100000` alunos. Execute e marque o tempo.

```
package br.com.caelum.ed.vetores;

import br.com.caelum.ed.Aluno;

public class TesteLinearVSConstante {
  public static void main(String[] args) {
    Vetor vetor = new Vetor();
    long inicio = System.currentTimeMillis();
    for (int i = 1; i < 100000; i++) {
      Aluno aluno = new Aluno();
      vetor.adiciona(aluno);
    }
    long fim = System.currentTimeMillis();
    double tempo = (fim - inicio) / 1000.0;
    System.out.println("Tempo em segundos = " + tempo);
  }
}
```

Implemente o método `adiciona(Aluno)` da segunda maneira vista neste capítulo. Não esqueça de acrescentar o atributo `totalDeAlunos`.

```
public class Vetor {

  private Aluno[] alunos = new Aluno[100000];

  private int totalDeAlunos = 0;

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

Execute a classe `TesteLinearVSConstante` e veja o tempo agora. A diferença de tempo é bem considerável.

5. Implemente o método `tamanho()` na classe `Vetor` como visto neste capítulo.

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

Execute o teste apropriado feito anteriormente.

6. Implemente o método `contem()` na classe `Vetor` da primeira maneira mostrada neste capítulo.

```
public boolean contem(Aluno aluno) {
  for (int i = 0; i < this.alunos.length; i++) {
    if (aluno == this.alunos[i]) {
      return true;
    }
  }
  return false;
}
```

Verifique este método com o teste do contem aluno. Faça outro teste para calcular o tempo gasto.

```
package br.com.caelum.ed.vetores;

import br.com.caelum.ed.Aluno;

public class TesteDeTempoDoContem {
  public static void main(String[] args) {
    Vetor vetor = new Vetor();
    long inicio = System.currentTimeMillis();

    // Adicionado 100000 alunos e
    // Verificando se eles foram realmente adicionados.
    for (int i = 1; i < 30000; i++) {
      Aluno aluno = new Aluno();
      vetor.adiciona(aluno);
      if(!vetor.contem(aluno)){
```

```
        System.out.println("Erro: aluno "
          + aluno + " não foi adicionado.");
      }
    }

    // Verificando se o Vetor não encontra alunos não adicionados.
    for (int i = 1; i < 30000; i++) {
      Aluno aluno = new Aluno();
      if(vetor.contem(aluno)){
        System.out.println("Erro: aluno "
           + aluno + " foi adicionado.");
      }
    }

    long fim = System.currentTimeMillis();
    double tempo = (fim - inicio) / 1000.0;
    System.out.println("Tempo = " + tempo);
  }
}
```

Implemente o método `contem()` na classe `Vetor` da segunda maneira mostrada neste capítulo.

```
public boolean contem(Aluno aluno) {
  for (int i = 0; i < this.totalDeAlunos; i++) {
    if (aluno == this.alunos[i]) {
      return true;
    }
  }
  return false;
}
```

Execute o teste novamente e veja a diferença de tempo.

7. Implemente o método `pega(int)` na classe `Vetor` da primeira maneira vista neste capítulo.

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

Teste este método! Depois faça um teste pegando uma posição ocupada, uma vazia e uma que não existe.

```
public class Teste {
  public static void main(String[] args) {
    Aluno a1 = new Aluno();

    a1.setNome("Rafael");

    Vetor1 lista = new Vetor1();

    lista.adiciona(a1);
```

```
    lista.pega(0);

    lista.pega(1);

    lista.pega(100000000);
  }
}
```

Este teste deve gerar um erro somente na última linha.

Implemente o método `pega(int)` na classe `Vetor` da segunda maneira vista neste capítulo. Não esqueça de implementar o método `posicaoOcupada(int)` também.

```
public Aluno pega(int posicao) {
  if (!this.posicaoOcupada(posicao)) {
    throw new IllegalArgumentException("Posição inválida");
  }
  return this.alunos[posicao];
}

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

Rode o teste novamente e agora veja que o erro já ocorre no segundo "pega". Este é o resultado esperado pois ao pegar uma posição não ocupada deve ocorrer erro.

8. Implemente o método `adiciona(int, Aluno)` na classe `Vetor`. Não esqueça do método `posicaoValida(int)`.

```
public void adiciona(int posicao, Aluno aluno) {
  if (!this.posicaoValida(posicao)) {
    throw new IllegalArgumentException("Posição inválida");
  }

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

  this.alunos[posicao] = aluno;
  this.totalDeAlunos++;
}

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

Rode o teste adequado!

9. Implemente o método `remove(int)` na classe `Vetor`.

```
public void remove(int posicao) {
  if (!this.posicaoOcupada(posicao)) {
```

```
    throw new IllegalArgumentException("Posição inválida");
  }
  for (int i = posicao; i < this.totalDeAlunos - 1; i++) {
    this.alunos[i] = this.alunos[i + 1];
  }
  this.totalDeAlunos--;
}
```

Faça o teste do remove para verificar se tudo deu certo.

10. Execute o seguinte teste:

```
package br.com.caelum.ed.vetores;

import br.com.caelum.ed.Aluno;

public class TesteEstoura {
  public static void main(String[] args) {
    Vetor vetor = new Vetor();
    for (int i = 0; i < 100001; i++) {
      Aluno aluno = new Aluno();
      vetor.adiciona(aluno);
    }
  }
}
```

Um erro ocorre pois a capacidade do Vetor estoura.

Implemente o método `garantaEspaco()` na classe `Vetor` para evitar o problema de estourar a capacidade.

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

Não esqueça de invocar o método `garantaEspaco()` dentro dos dois métodos de adicionar alunos.

Execute a classe `TesteEstoura` mais uma vez.

11. Implemente a classe `Vetor` para objetos (Genérico) com os métodos e atributos vistos neste capítulo na seção de generalização.

12. Utilize a classe `Vetor` que foi implementada nos exercícios anteriores e as classes da API do Java `Vector` ou `ArrayList`. Para saber os métodos das classes da

API, utilize a documentação do Java (http://java.sun.com/j2se/1.5.0/docs/api/index.html).

1. Crie um vetor usando a classe `Vetor`; guarde nele 1000 alunos;

2. Imprima o tamanho do vetor antes e depois de adicionar os 1000 alunos.

3. Crie um vetor usando a classe `Vector`; passe os alunos do vetor anterior para este novo vetor; imprima o tamanho do novo vetor antes e depois de adicionar os alunos.

4. Crie um vetor usando a classe \classe{ArrayList}; passe os alunos do vetor criado no item anterior para este novo vetor; imprima o tamanho do novo vetor antes e depois de adicionar os alunos.

4.14 – Exercícios opcionais

1. Use o recurso de generics do Java 5 e crie um Vetor usando a classe `ArrayList` para guardar objetos do tipo `String`. Teste adicionar neste vetor algumas `Strings`; e também tente adicionar `Alunos` (O que acontecerá quando você tentar adicionar alunos?). Retire elementos deste vetor (você precisa fazer casting para `String` nestes elementos?).

2. Acrescente uma operação na Lista, para isso, implemente um novo método. A nova operação deve remover da Lista todas as ocorrências de um elemento que é passado com parâmetro. Não esqueça de rearranjar os elementos do vetor após a remoção.

```
public void remove(Object objeto) {
  // implementação
}
```

3. Acrescente uma operação na Lista, para isso, implemente um novo método. A nova operação deve limpar a lista, ou seja, remover todos os elementos.

```
public void clear() {
  // implementação
}
```

4. Acrescente uma operação na Lista, para isso, implemente um novo método. A nova operação deve procurar o índice da primeira ocorrência de um elemento passado como parâmetro.

```
public void indexOf(Object objeto) {
  // implementação
```

```
}
```

5. Acrescente uma operação na Lista, para isso, implemente um novo método. A nova operação deve procurar o índice da última ocorrência de um elemento passado como parâmetro.

```
public void lastIndexOf(Object objeto) {
  // implementação
}
```

6. (Desafio) Pesquise sobre análise amortizada para saber porque é mais eficiente dobrar a capacidade da array quando o vetor fica cheio. A outra opção seria incrementar. Por exemplo, quando acaba o espaço na array, criaríamos uma com o mesmo tamanho + 10. É incrível a diferença de uma estratégia e de outra, ao você tentar adicionar uma enorme quantidade de elementos.

CAPÍTULO ANTERIOR:

Armazenamento Sequencial

PRÓXIMO CAPÍTULO:

Listas Ligadas

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter