# Programação orientada a objetos em Java

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

# Conteúdo

# Introdução

# Introdução

- Programação baseada na composição e interação entre objetos
  - POO surgiu com a linguagem Simula (1965), usada para criar simulações
  - Composição de objetos e suas interações também é uma boa abstração para muitos outros domínios além da simulação

# Introdução

- Os objetos podem representar coisas reais ou abstratas
- Os objetos têm
  - Estado (atributos, assim como uma estrutura em C)
  - Comportamento (métodos)

# Introdução

- Em geral, o estado de um objeto não pode ser manipulado diretamente, apenas através dos métodos do objeto
    - Este conceito é chamado de encapsulamento e é um dos pilares da POO

# Introdução

- Objetos em Java são instanciados a partir de classes usando o operador `new`
- O operador `new` invoca o construtor especificado
- O construtor é um método especial responsável por inicializar o objeto em um estado consistente
- Cada classe tem ao menos um construtor

# Uso de classes existentes

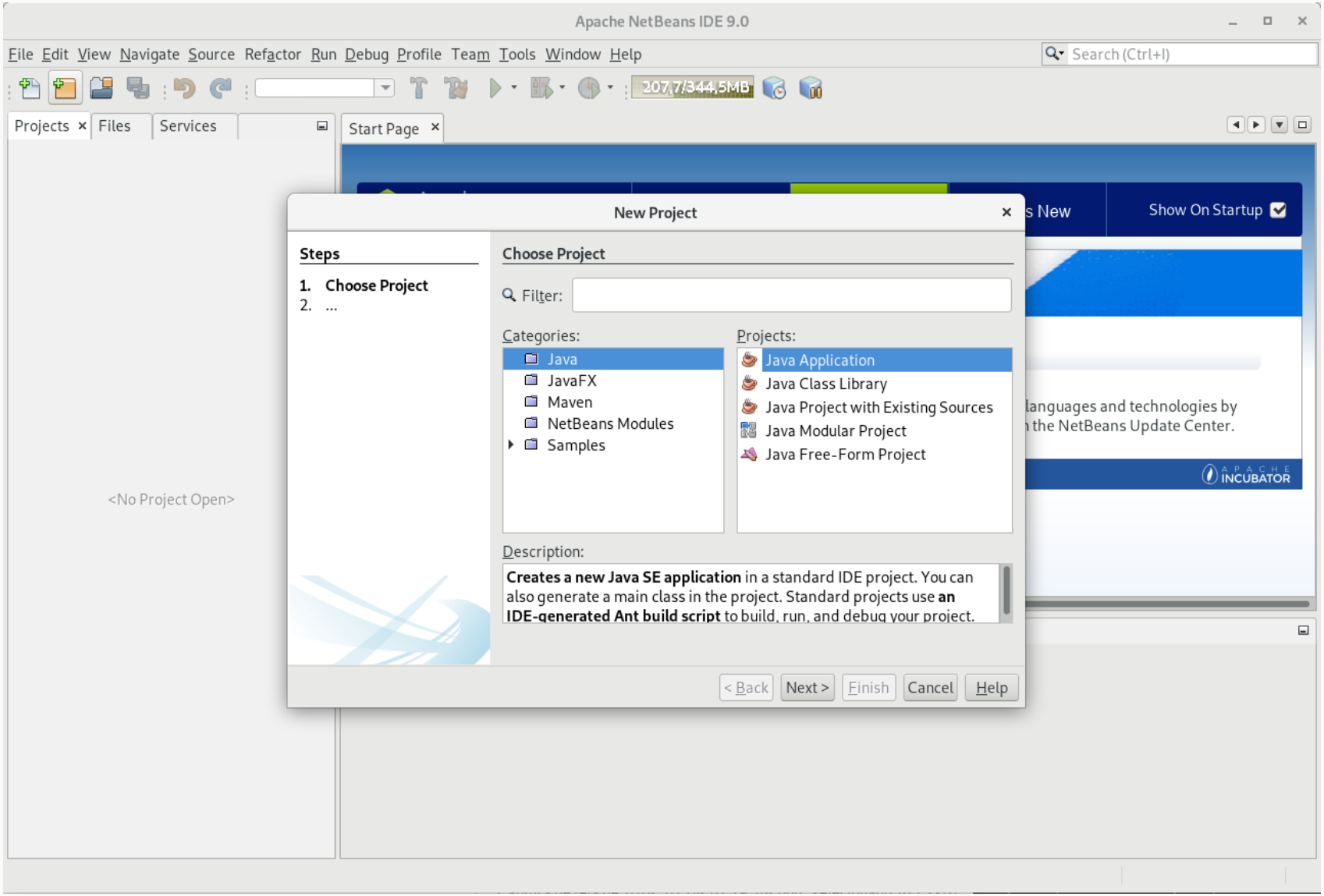
# Uso de classes existentes

- Java oferece uma biblioteca com milhares de classes, vamos ver alguns exemplos
- Vamos criar um projeto no Netbeans e criar um classe de teste para cada exemplo

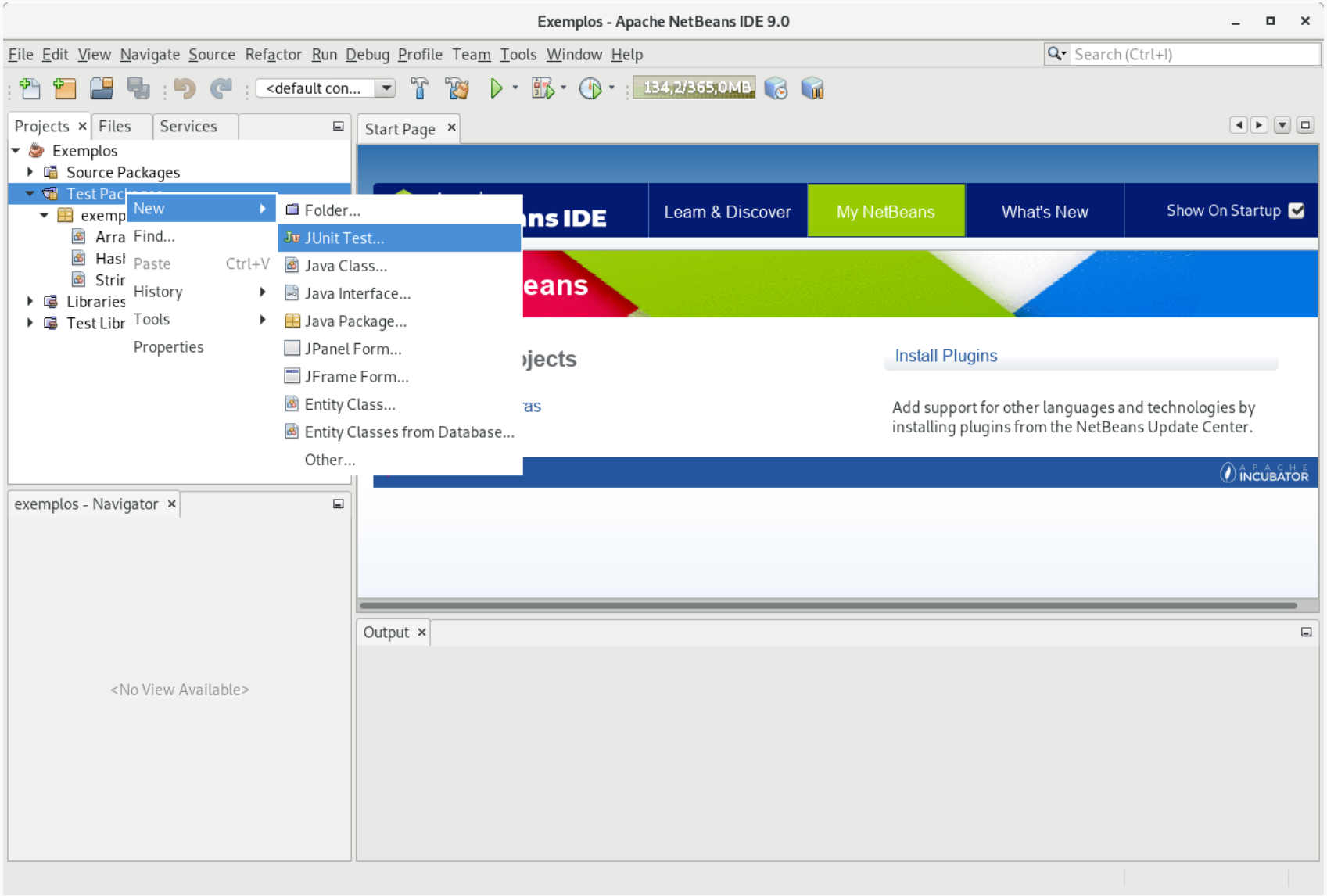
```
import org.junit.Test;
import static org.junit.Assert.*;

public class NomeDaClasseTest {
    @Test
    public void testNomeDoMetodo() {
        // código do teste
    }
}
```

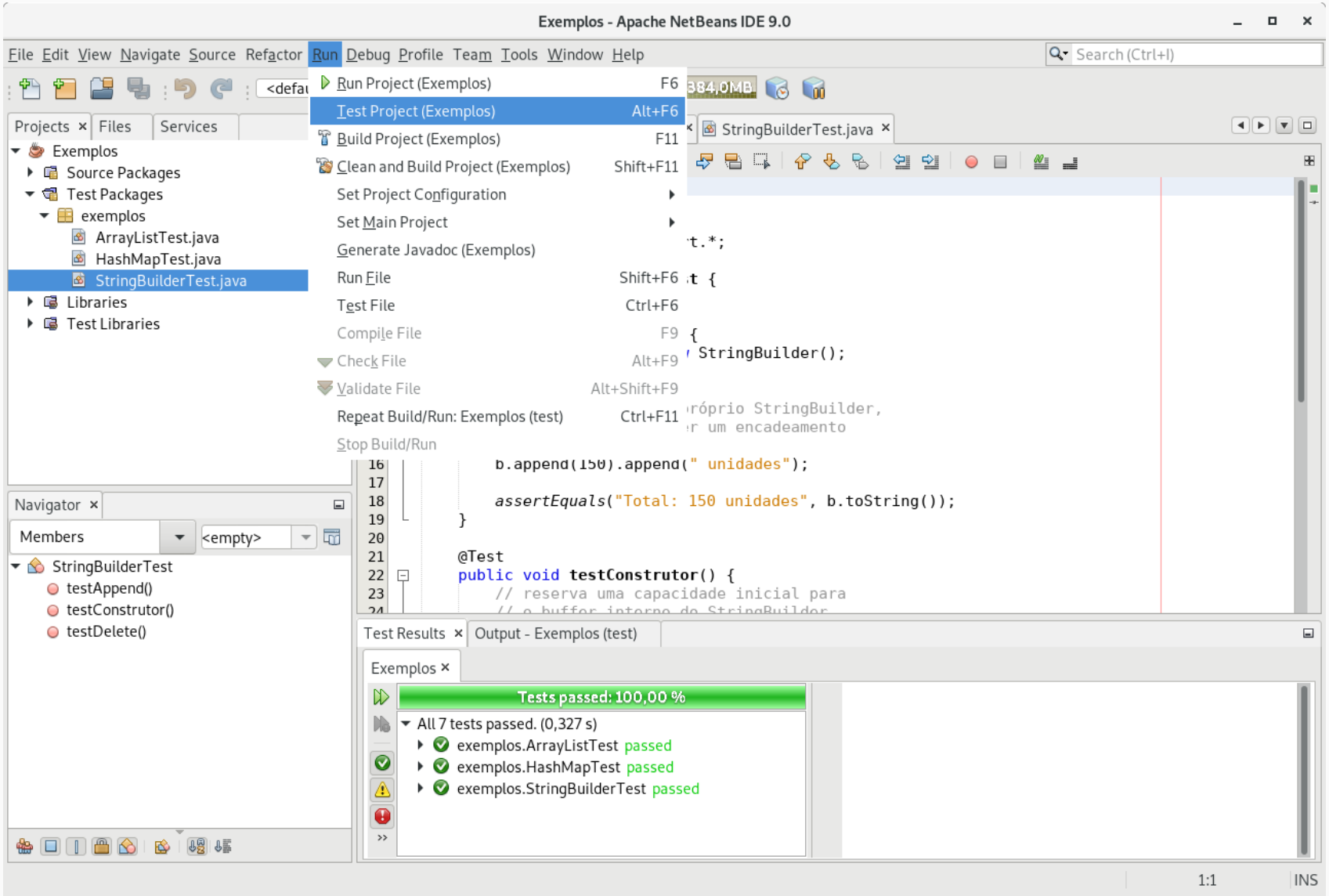
# Uso de classes existentes



# Uso de classes existentes



# Uso de classes existentes



# StringBuilder

- As strings em Java são imutáveis
- Muitas operações de edição de string com o exemplo abaixo podem ser ineficiente

```
String s = "Total: " + 150 + " unidades";
```

- Para trabalhar de forma eficiente com edição de strings, usamos a classe `StringBuilder`

# StringBuilder

```java
@Test
public void testAppend() {
    StringBuilder b = new StringBuilder();
    b.append("Total: ");

    // append retorna o próprio StringBuilder,
    // então podemos fazer um encadeamento
    // de chamadas
    b.append(150).append(" unidades");

    assertEquals("Total: 150 unidades", b.toString());
}
```

# StringBuilder

```java
@Test
public void testConstrutor() {
    // reserva uma capacidade inicial para
    // o buffer interno do StringBuilder
    StringBuilder b = new StringBuilder(2500);
    assertTrue(b.capacity() >= 2500);
}
```

```java
@Test
public void testDelete() {
    StringBuilder b = new StringBuilder(
        "Construtor que recebe uma string inicial"
    );
    b.delete(10, 32);
    assertEquals("Construtor inicial", b.toString());
}
```

Outros métodos

| Método | Descrição |
|---|---|
| `indexOf` | Retorna o índice da primeira ocorrência de uma string |
| `insert` | Insere a representação em string de um objeto em uma determinada posição |
| `deleteCharAt` | Remove um caractere de uma determinada posição |
| `substring` | Devolve uma string que contém os caracteres em um determinado intervalo |

# ArrayList

- Uma lista implementada com um arranjo que aumenta e diminui de tamanho conforme a necessidade
- Usado como alternativa para um arranjo quando a quantidade de elementos do arranjo pode variar

# ArrayList

```java
@Test
public void testAdd() {
    ArrayList<Integer> lista = new ArrayList<>();
    assertEquals(0, lista.size());
    lista.add(20);
    lista.add(10);
    lista.add(20);
    assertEquals(3, lista.size());
    // apenas assertEquals(20, lista.get(0)) não compila
    // devido a confusão entre o tipo primitivo int
    // e o tipo referência Integer...
    assertEquals(new Integer(20), lista.get(0));
    assertEquals(10, lista.get(1).intValue());
    assertEquals(20, lista.get(2).intValue());
}
```

```java
@Test
public void testSort() {
    ArrayList<Integer> lista = new ArrayList<>();
    lista.add(10);
    lista.add(2);
    lista.add(-4);
    lista.add(3);
    // ordena segundo a ordem natural dos inteiros
    lista.sort(null);
    assertArrayEquals(
        new Integer[] {-4, 2, 3, 10},
        lista.toArray(new Integer[4])
    );
}
```

Outros métodos

| Método | Descrição |
|---|---|
| `contains` | Verifica se um determinado elemento está na lista |
| `isEmpty` | Devolve verdadeiro se a lista está vazia, falso caso contrário |
| `set` | Substitui o elemento em uma determinada posição |
| `remove` | Remove a primeira ocorrência de um elemento na lista, se ele estiver presente |

# HashMap

- Um dicionário (mapeamento entre chave e valor) implementado com uma tabela hash

```java
@Test
public void testPut() {
    HashMap<Integer, String> map = new HashMap();
    map.put(10, "verde");
    assertEquals("verde", map.get(10));
    map.put(2, "amarelo");
    assertEquals("verde", map.get(10));
    assertEquals("amarelo", map.get(2));
    // substitui o antigo valor
    map.put(10, "vermelho");
    assertEquals("vermelho", map.get(10));
    assertEquals("amarelo", map.get(2));
}
```

```java
@Test
public void testRemove() {
    HashMap<Integer, String> map = new HashMap();

    map.put(10, "verde");
    map.put(2, "amarelo");
    map.put(7, "vermelho");
    map.remove(2);

    assertEquals("verde", map.get(10));
    assertEquals("vermelho", map.get(7));
    assertNull(map.get(2));
}
```

Outros métodos

| Método | Descrição |
|---|---|
| `containsKey` | Devolve verdadeiro se o `HashMap` contém uma determinada chave, falso caso contrário |
| `clear` | Remove todos os itens do `HashMap` |

# Outras classes

| Classe | Descrição |
|---|---|
| `Arrays` | Contém um conjunto de métodos estáticos que trabalham com arranjos |
| `Collections` | Contém um conjunto de métodos estáticos que trabalham com coleções |
| `HashSet` | Um conjunto implementado com uma tabela hash |
| `Scanner` | Um scanner que pode extrair tipos primitivos e strings de texto (incluindo arquivos) |
| `FileReader` | Leitura de arquivos texto |
| `FileWriter` | Criação e escrita de arquivos texto |

# Uso de classes existentes

- Podemos fazer algumas observações sobre o uso de classes existentes
  - Precisamos entender o propósito de cada classe
  - Mas não precisamos entender os detalhes da implementação
    - Uma ideia da implementação pode ser útil, ex: tempo de execução de `ArrayList.contains` é linear ao passo que `HashSet.contains` é constante
  - O estado de um objeto de uma classe é manipulado indiretamente através dos métodos

# Criação de novas classes

# Criação de novas classes

Escreva um programa que leia um arquivo texto e imprima na tela as $n$ palavras mais frequentes no arquivo e o número de ocorrências destas palavras. O nome do arquivo e o valor de $n$ devem ser passados como argumento na linha de comando.

# Criando novas classes

- Entrada
  - Classes `FileReader` e `Scanner` para leitura do arquivo e identificação das palavras
- Saída
  - Impressão simples usando `System.out`
- Processamento
  - Vamos definir uma classe para fazer a contagem das palavras

# Criando novas classes

- Antes de definir uma classe, podemos pensar em como gostaríamos de usar a classe se ela já existisse

```
Contagem contagem = new Contagem();

for (String palavra: palavras) {
    contagem.adiciona(palavra)
}

String[] maisFrequentes = contagem.getMaisFrequentes(n);
for (String palavra: maisFrequentes) {
    int frequencia = contagem.getFrequencia(palavra)
    ...
}
```

# Criando novas classes

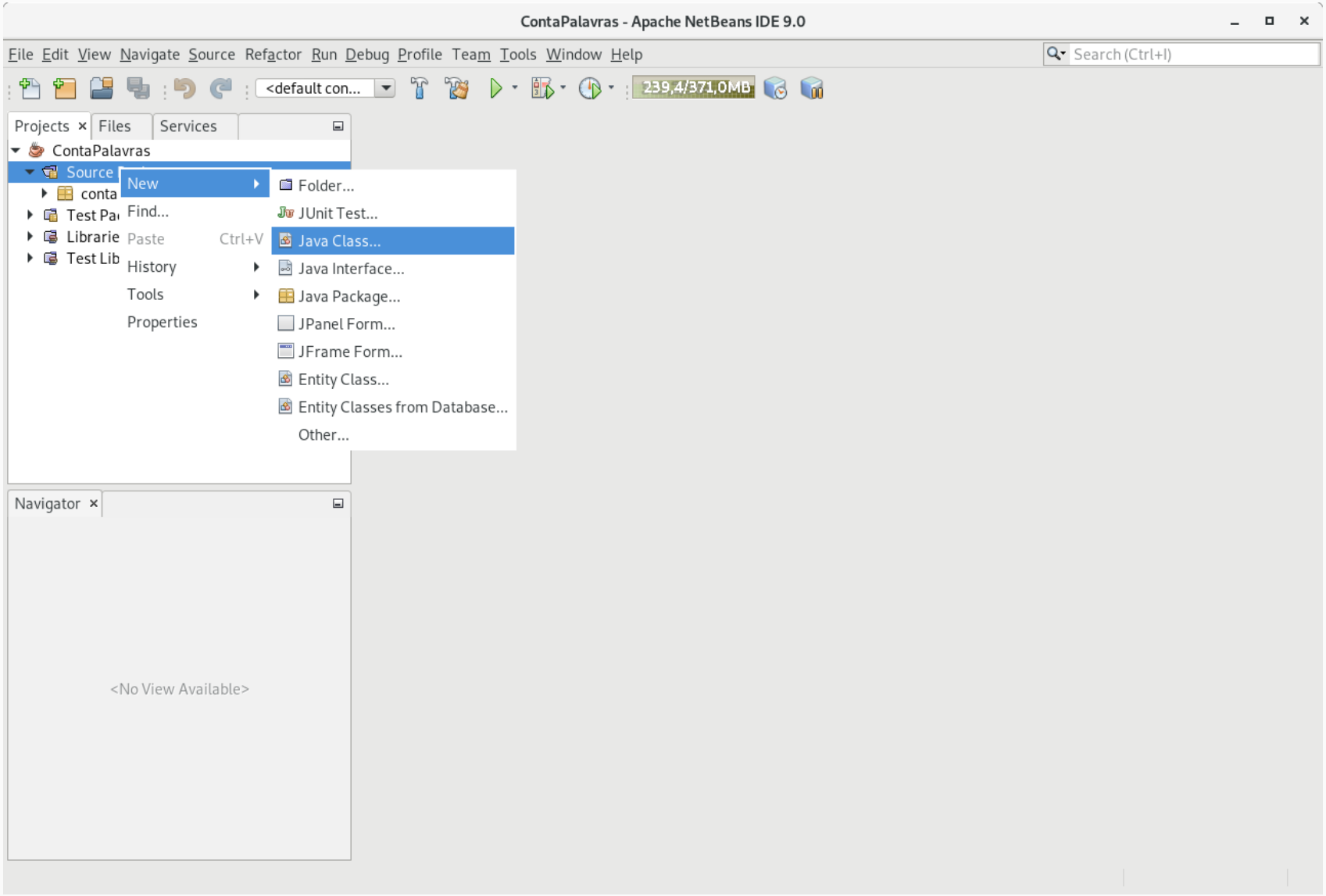
- Esboço da classe

```
public class Contagem {
    public void adiciona(String palavra);
    public int getFrequencia(String palavra);
    public String[] getMaisFrequentes(int quantidade);
}
```

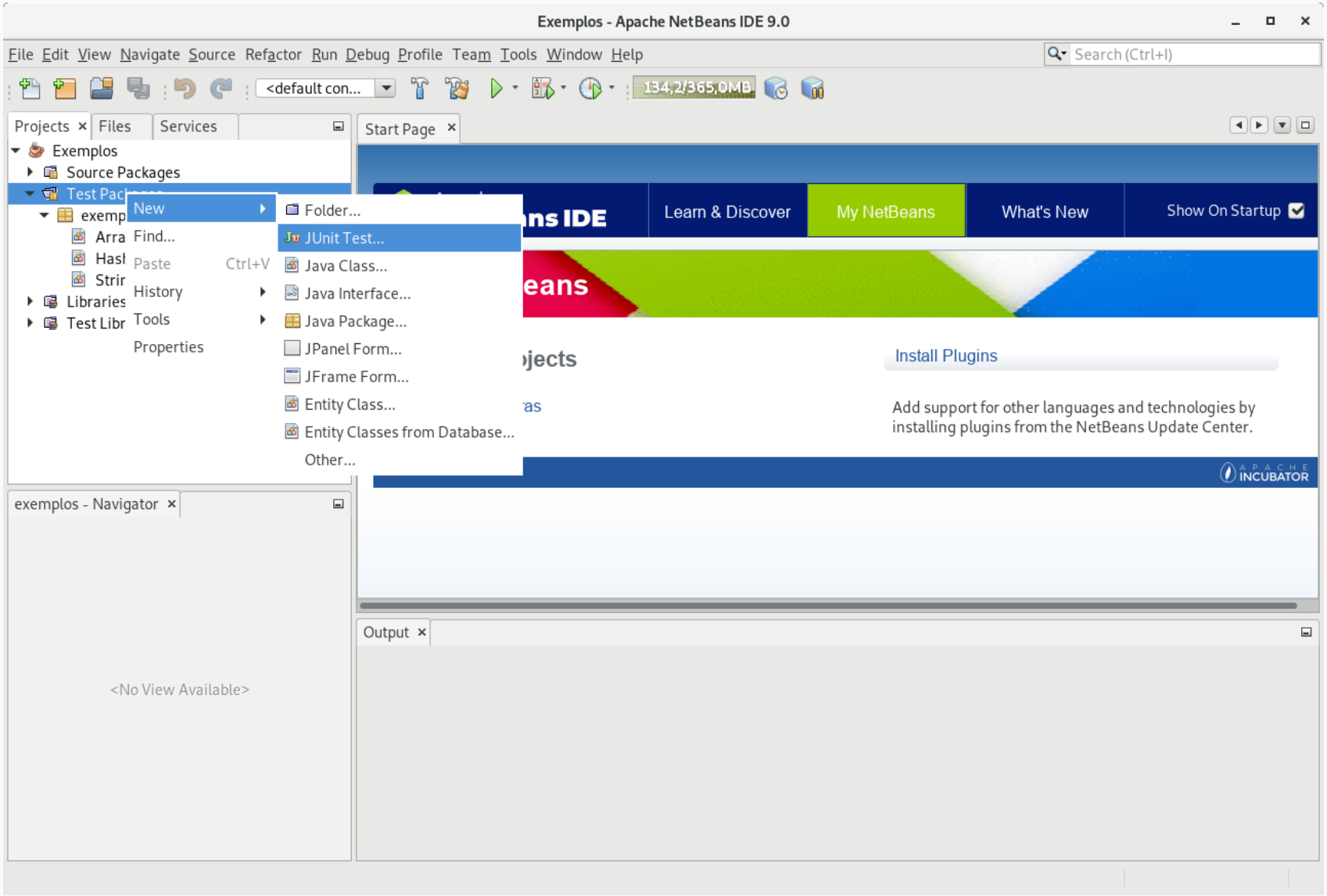
# Criando novas classes

- Vamos criar uma implementação simples que armazena as palavras e as frequências em um `ArrayList`
- Vamos escrever os testes antes da implementação!

# Criando novas classes



# Criando novas classes



# Criando novas classes

```java
@Test
public void testAdicionaFreq() {
    Contagem contagem = new Contagem();
    assertEquals(0, contagem.getFrequencia("casa"));
    assertEquals(0, contagem.getFrequencia("carro"));

    contagem.adiciona("casa");
    assertEquals(1, contagem.getFrequencia("casa"));
    assertEquals(0, contagem.getFrequencia("carro"));

    contagem.adiciona("casa");
    assertEquals(2, contagem.getFrequencia("casa"));
    assertEquals(0, contagem.getFrequencia("carro"));

    contagem.adiciona("carro");
    assertEquals(2, contagem.getFrequencia("casa"));
    assertEquals(1, contagem.getFrequencia("carro"));
}
```

# Criando novas classes

```
import java.util.ArrayList;

public class Contagem {
    private final ArrayList<PalavraFreq> tabela;

    // Construtor, método chamado pelo operador new
    // para inicializar o objeto
    public Contagem() {
        tabela = new ArrayList();
    }
}

class PalavraFreq {
    String palavra;
    int freq;
}
```

# Criando novas classes

```
public class Contagem {
    public void adiciona(String palavra) {
        PalavraFreq entrada = null;
        for (PalavraFreq pf : tabela) {
            if (pf.palavra.equals(palavra)) {
                entrada = pf;
                break;
            }
        }
        if (entrada != null) {
            entrada.freq++;
        } else {
            tabela.add(new PalavraFreq(palavra));
        }
    }
}
```

# Criando novas classes

```
public class Contagem {
    public int getFrequencia(String palavra) {
        PalavraFreq entrada = null;
        for (PalavraFreq pf : tabela) {
            if (pf.palavra.equals(palavra)) {
                entrada = pf;
                break;
            }
        }
        if (entrada != null) {
            return entrada.freq;
        } else {
            return 0;
        }
    }
}
```

# Criando novas classes

- Os testes funcionam, hora de refatorar!
- Vamos extrair o código comum entre `adiciona` e `getFrequencia`

# Criando novas classes

```
public class Contagem {
    public void adiciona(String palavra) {
        PalavraFreq entrada = getEntrada(palavra);
        if (entrada != null) {
            entrada.freq++;
        } else {
            tabela.add(new PalavraFreq(palavra));
        }
    }
    public int getFrequencia(String palavra) {
        PalavraFreq entrada = getEntrada(palavra);
        if (entrada == null) {
            return 0;
        } else {
            return entrada.freq;
        }
    }
    private PalavraFreq getEntrada(String palavra) {
        for (PalavraFreq entrada : tabela) {
            if (entrada.palavra.equals(palavra)) {
                return entrada;
            }
        }
        return null;
    }
}
```

# Criando novas classes

```
@Test
public void testGetMaisFrequentes() {
    Contagem contagem = new Contagem();
    contagem.adiciona("casa");
    contagem.adiciona("carro");
    contagem.adiciona("banana");
    contagem.adiciona("casa");
    contagem.adiciona("casa");
    contagem.adiciona("banana");
    assertArrayEquals(
            new String[]{"casa"},
            contagem.getMaisFrequentes(1));
    assertArrayEquals(
            new String[]{"casa", "banana"},
            contagem.getMaisFrequentes(2));
    assertArrayEquals(
            new String[]{"casa", "banana", "carro"},
            contagem.getMaisFrequentes(3));
    // E se quantidade > número de elementos ?
}
```

# Criando novas classes

```
public class Contagem {
    public String[] getMaisFrequentes(int quantidade) {
        tabela.sort((a, b) -> Integer.compare(b.freq, a.freq));
        String[] mais = new String[quantidade];
        for (int i = 0; i < quantidade; i++) {
            mais[i] = tabela.get(i).palavra;
        }
        return mais;
    }
}
```

# Criando novas classes

- A classe `Contagem` está pronta, vamos escrever a classe principal

# Criando novas classes

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) throws FileNotFoundException {
        String arquivo = args[0];
        int quantidade = Integer.parseInt(args[1]);
        Scanner scanner = new Scanner(new FileReader(arquivo));
        // use qualquer sequência de caracteres que não
        // sejam letras como delimitador
        scanner.useDelimiter("[^\\p{IsLetter}]+");
        Contagem contagem = new Contagem();
        while (scanner.hasNext()) {
            contagem.adiciona(scanner.next());
        }
        for (String palavra : contagem.getMaisFrequentes(quantidade)) {
            System.out.println(palavra + " " + contagem.getFrequencia(palavra));
        }
    }
}
```

# Criando novas classes

- Executando o programa na linha de comando

```
$ ant jar # ou Crtl+F11 no Netbeans
$ time java -cp dist/ContaPalavras.jar moby-dick.txt 5
the 13885
of 6648
and 6095
a 4631
to 4617

real    0m5,712s
user    0m6,441s
sys     0m0,089s
```

# Criando novas classes

- Parece muito lento para um arquivo com 215136 palavras...
- O problema é que a implementação de `Cotagem.adiciona` é linear no número de palavras distintas
- No pior caso, uma sequência de adições de palavras é quadrático
- Vamos criar uma implementação alternativa
- Queremos que o código funcione com as duas implementações, desta forma poderemos avaliar qual é mais eficiente

# Interfaces

# Interfaces

- Uma classe define uma coleção de métodos
- A coleção de métodos de uma classe é a interface que é usada para se comunicar com objetos do tipo da classe
- Em Java é possível separar a interface da implementação, desta forma, várias classes podem implementar a mesma interface

# Interfaces

- Java define muitas interfaces na biblioteca padrão que são implementadas por diferentes classe, por exemplo
  - `Collection` é uma interface que representa qualquer coleção de valores
    - Define métodos para adicionar e remover valores da coleção, criar arranjos com os itens da coleção, etc
  - Muitas classes implementam a interface `Collection`
    - `ArrayList`, `LinkedList`, `TreeSet`, `HashSet`, etc

# Interfaces

- A uma variável definida com um tipo que é uma interface pode ser atribuída instâncias de qualquer classe que implemente aquela interface

```
Collection<Integer> c = null;

if (...) { c = new ArrayList<>(); } else { c = new TreeSet<>(); }

c.add(20);
...
```

- Todos os métodos de c são vinculado dinamicamente, pois o tipo concreto de c é determinado em tempo de execução
  - Esta vinculação dinâmica de método é chamada de polimorfismo e é outro pilar da POO

# Interfaces

```java
import java.util.ArrayList;
import java.util.Collection;
import java.util.TreeSet;

public class X {
    public static void main(String[] args) {
        // instâncias de qualquer classe que
        // implemente a interface Collection
        // pode ser passada como parâmetro
        // para o método cores
        cores(new ArrayList<>());
        cores(new TreeSet<>());
    }

    static void cores(Collection<String> c) {
        c.add("amarelo");
        c.add("verde");
        c.add("azul");
        System.out.println("= "
            + c.getClass().getCanonicalName()
            + " =");
        System.out.println("Total de cores: " + c.size());
        for (String cor : c) {
            System.out.println(cor);
        }
    }
}
```

Resultado da execução:

```
= java.util.ArrayList =
Total de cores: 3
amarelo
verde
azul
= java.util.TreeSet =
Total de cores: 3
amarelo
azul
verde
```

# Interfaces

- Vamos definir uma interface para `Contagem`

```
public interface Contagem {
    void adiciona(String palavra);
    int getFrequencia(String palavra);
    String[] getMaisFrequentes(int quantidade);
}
```

# Interfaces

- Vamos renomear a nossa antiga classe `Contagem` para `ContagemArrayList` e implementar a nova interface `Contagem`

```
public class ContagemArrayList implements Contagem {
    ...
}
```

# Interfaces

- Vamos criar uma nova implementação de `Contagem` usando um `HashMap`

```
public class ContagemHashMap implements Contagem {
    private final HashMap<String, Integer> frequencia;

    ...
}
```

# Interfaces

- Como `ContagemArrayList` e `ContagemHashMap` implementam a interface `Contagem`, elas podem ser usadas em qualquer código que dependa de uma `Contagem`

# Interfaces

- Modificamos o método `main` para instanciar a classe que implementa `Contagem` de acordo como um argumento para o programa (o restante do código do método `main` permanece inalterado)

```
String impl = args[0];
...
Contagem contagem = null;
switch (impl) {
    case "array":
        contagem = new ContagemArrayList();
        break;
    case "map":
        contagem = new ContagemHashMap();
        break;
    default:
        System.out.println("Argumento impl inválido: " + impl);
        System.exit(1);
}
```

# Interfaces

Agora podemos executar o programa com cada uma das duas implementações de `Contagem` e verificar qual é mais eficiente

```
$ time java -cp dist/ContaPalavras.jar\
    array moby-dick.txt 5
the 13885
of 6648
and 6095
a 4631
to 4617

real    0m5,712s
user    0m6,441s
sys     0m0,089s
```

```
$ time java -cp dist/ContaPalavras.jar\
    map moby-dick.txt 5
the 13885
of 6648
and 6095
a 4631
to 4617

real    0m0,690s
user    0m1,459s
sys     0m0,089s
```

# Herança

# Herança

- Como a classe `ContagemHashMap` é testada?
- Da mesma forma que a classe `ContagemArrayList`

# Herança

- Antes nós "extraímos" a interface de contagem para permitir mais de uma implementação
- Agora nós temos uma implementação de teste e queremos especializar para duas situações
  - Testar a classe `ContagemArrayList`
  - Testar a classe `ContagemHashMap`
- A especialização através de herança é o terceiro pilar da POO

# Herança

- Considere a classe `ContagemTest` (versão quando a classe `ContagemArrayList` chamava apenas `Contagem`)

```
public class ContagemTest {
    @Test
    public void testAdicionaFreq() {
        Contagem contagem = new Contagem();
        ...
    }

    @Test
    public void testGetMaisFrequentes() {
        Contagem contagem = new Contagem();
        ...
    }
}
```

- Enquanto o corpo dos métodos de teste não dependem da classe que está sendo testada, a instanciação de `contagem` depende. Precisamos "especializar" a instanciação para cada uma das classe que queremos testar.

# Herança

- Vamos criar um método abstrato (sem corpo) para instanciar a classe contagem que será testada
  - Uma classe com pelo menos um método abstrato é considerada abstrata
  - Uma classe abstrata não pode ser instanciada
- Este método será especializado para cada caso usando herança

# Herança

```
public abstract class ContagemTest {
    abstract Contagem criaContagem();

    @Test
    public void testAdicionaFreq() {
        Contagem contagem = criaContagem();
        ...
    }

    @Test
    public void testGetMaisFrequentes() {
        Contagem contagem = criaContagem;
        ...
    }
}
```

# Herança

- Para especializar o método `criaContagem` criamos uma subclasse
  - A classe que está sendo especializada é chamada de superclasse ou classe mãe
- Uma subclasse é como a classe que ela especializa, mas pode definir novos campos e métodos e sobrescrever métodos existentes
  - Todos os métodos e campos existentes na superclasse também estão presentes na subclasse

# Herança

```
public class ContagemArrayListTest extends ContagemTest {
    @Override
    Contagem criaContagem() {
        return new ContagemArrayList();
    }
}
```

`ContagemArrayListTest` é como a classe `ContagemTest`, mas ela define uma implementação específica para `criaContagem`. Quando o testador for executar os testes de `ContagemArrayListTest` ele vai executar o corpo dos métodos `testAdicionaFreq` e `testGetMaisFrequentes` definidos em `ContagemTest` mas o corpo de `criaContagem` definido em `ContagemArrayListTest`.

# Herança

```
public class ContagemHashMapTest extends ContagemTest {
    @Override
    Contagem createContagem() {
        return new ContagemHashMap();
    }
}
```

`ContagemHashMapTest` é como a classe `ContagemTest`, mas ela define uma implementação específica para `criaContagem`. Quando o testador for executar os testes de `ContagemHashMapTest` ele vai executar o corpo dos métodos `testAdicionaFreq` e `testGetMaisFrequentes` definidos em `ContagemTest` mas o corpo de `criaContagem` definido em `ContagemHashMaptTest`.

# Referências

# Referências

- Object-Oriented Programming Concepts
- Classes and Objects
- Interfaces and Inheritance