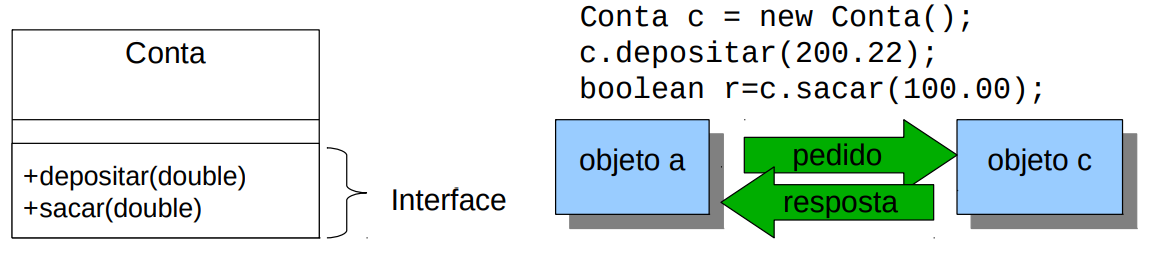
## Polimorfismo

* Interface de um objeto: conjunto de pedidos que podem ser feitos a ele, ou seja, os métodos que definem o comportamento dele
  + Enviando mensagem a um objeto:

<nome\_objeto>.<nome\_metodo>(<args>\*);

Propriedades de uma Interface:

* Em Java: interface é como uma classe 100% abstrata;
* Interface Java é equivalente a tipo (conforme GoF)‏;
* Tipo pode ser entendido como um subconjunto de mensagens que um objeto pode receber;
* É um contrato entre o código cliente (que usa a classe)‏ e a classe que implementa a interface Java;
* É uma declaração formal do contrato por meio de assinatura de métodos sem implementação;
* Várias classes, mesmo que não relacionadas, podem implementar o mesmo tipo (interface Java);
* Posso definir uma interface Java personalizada, ou usar uma pré-definida na biblioteca Java;
* Definindo um classe como implementando um tipo interface Java:

<modificador>\* class <nome> implements <nome\_interface>\* { }

* Todos os métodos definidos na interface Java devem ser implementados pela classe concreta;
* Uma classe pode implementar várias interfaces Java

public class I1 extends Applet implements Runnable, Estable, Sortable, Observable {...}.

* Sintaxe de Declaração:

<modificador>\* interface <nome> {

<protótipo\_método>\*

}

**Exemplo:**

public interface Voador {

public void decolar();

public void aterrizar();

public void voar();

}

public class Aviao implements Voador {

public void decolar() {

/\* implementação \*/

}

public void aterrizar(){

/\* implementação \*/

} public void voar() {

/\* implementação \*/

}

}

* Implementando várias interfaces

interface IServico {

public boolean iniciar();

public boolean parar();

}

interface IMonitor {

public int nivel();

public float temperatura();

}

class Reator implements IServico,IMonitor {

public boolean iniciar() {

/\* implementação \*/

}

public boolean parar() {

/\* implementação \*/

}

public int nivel() {

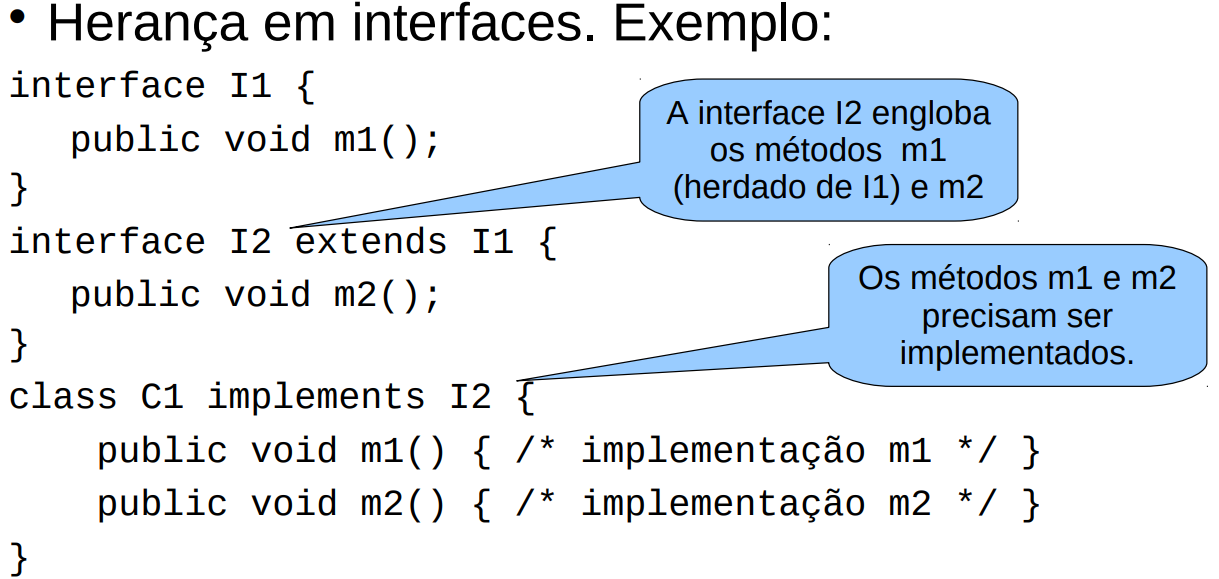
/\* implementação \*/

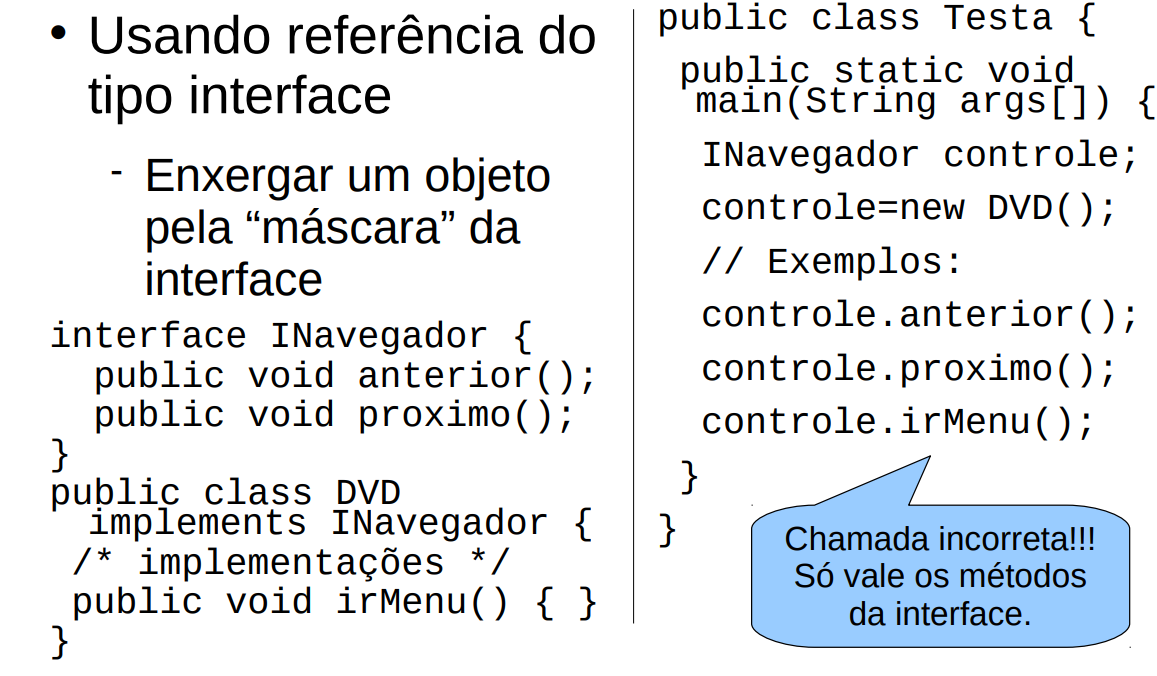
}

public float temperatura(){

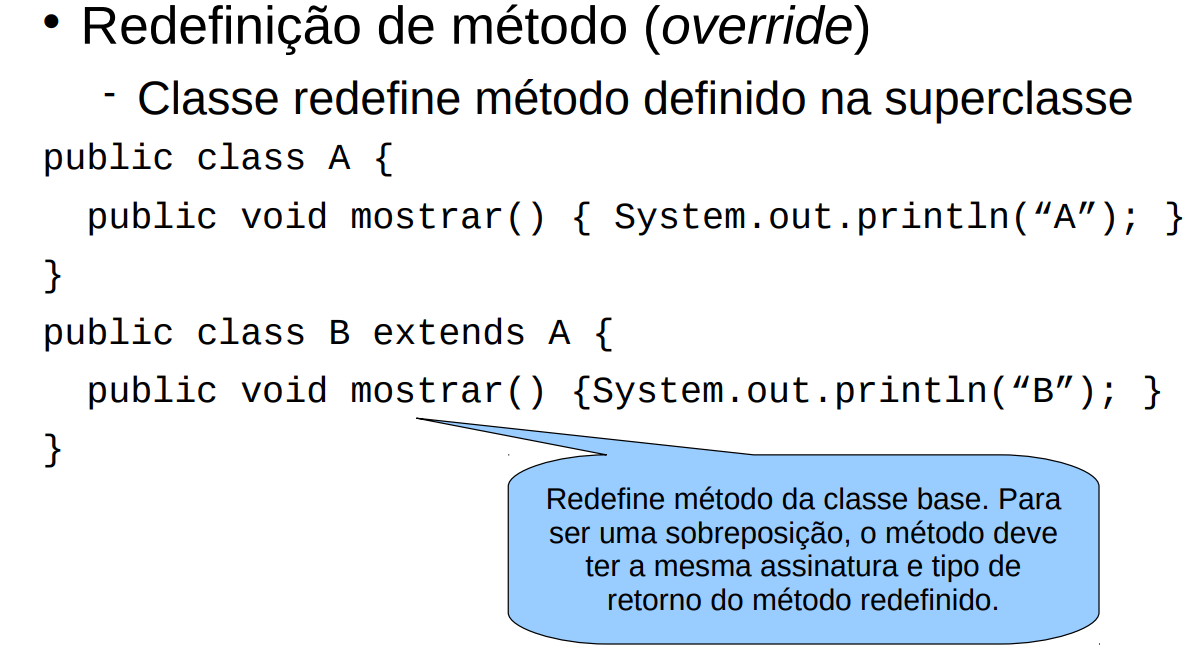
/\* implementação \*/

}

}



## Polimorfismo

* Significado: diferentes formas;
* Em POO: dois objetos, pertencentes a mesma classe base, podem se comportar de maneira diferente recebendo a mesma mensagem;
* Maneiras de implementar polimorfismo
  + Por implementação de método abstrato;
  + Por redefinição (override ou sobreposição)‏ de método herdado.
* Instrução qualificadora super

public class A {

public void mostrar() { System.out.println(“A”); }

}

public class B extends A {

public void mostrar() {

super.mostrar();

System.out.println(“B”); }

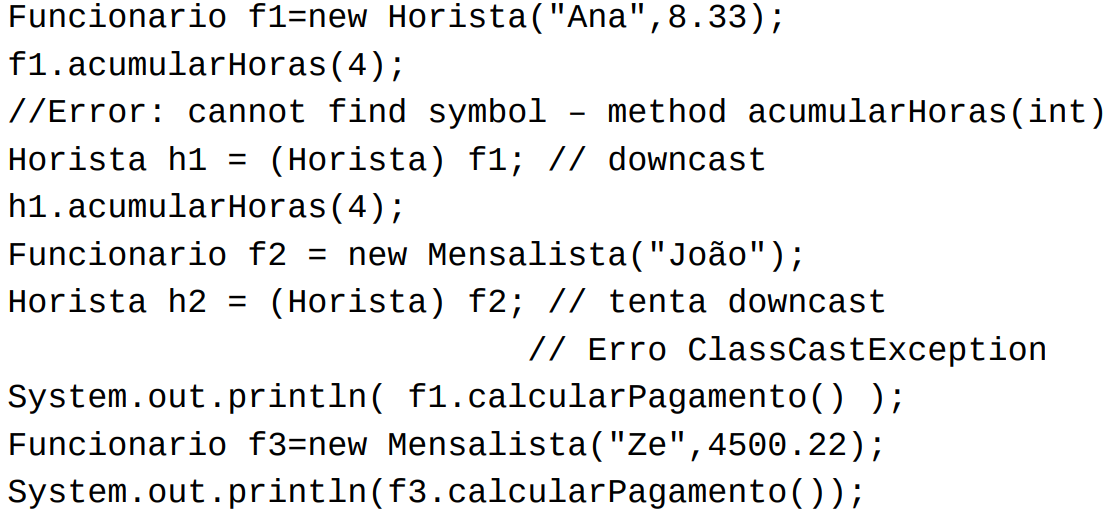
}

Nota: O super() serve para chamar o construtor da superclasse.

* Classe Abstrata
  + Define uma classe de objetos cuja implementação não conhecemos integralmente;
  + Diferente das classes concretas, não pode haver instaciamento direto desse tipo de classe;
  + Serve como um molde para que sejam definidas classes concretas, que devem completar a implementação ausente na abstrata;
  + De forma análoga, um método abstrato serve apenas para definir uma mensagem na interface, mas não sabemos a implementação.
* Referência polimórfica: consiste em utilizar o tipo de uma das superclasses na hierarquia para referenciar um objeto de uma subclasse.

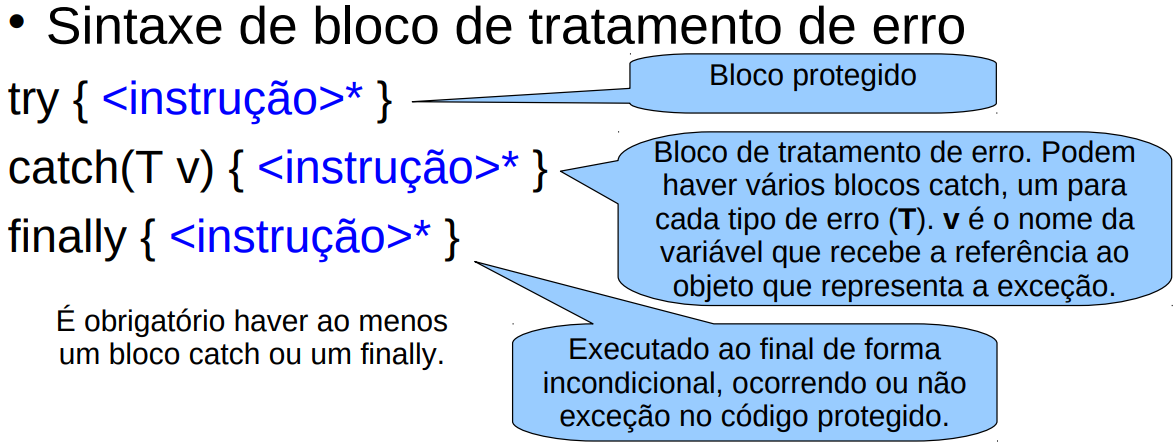
− Funcionario f1=new Horista(“Ana”,8.33)‏;

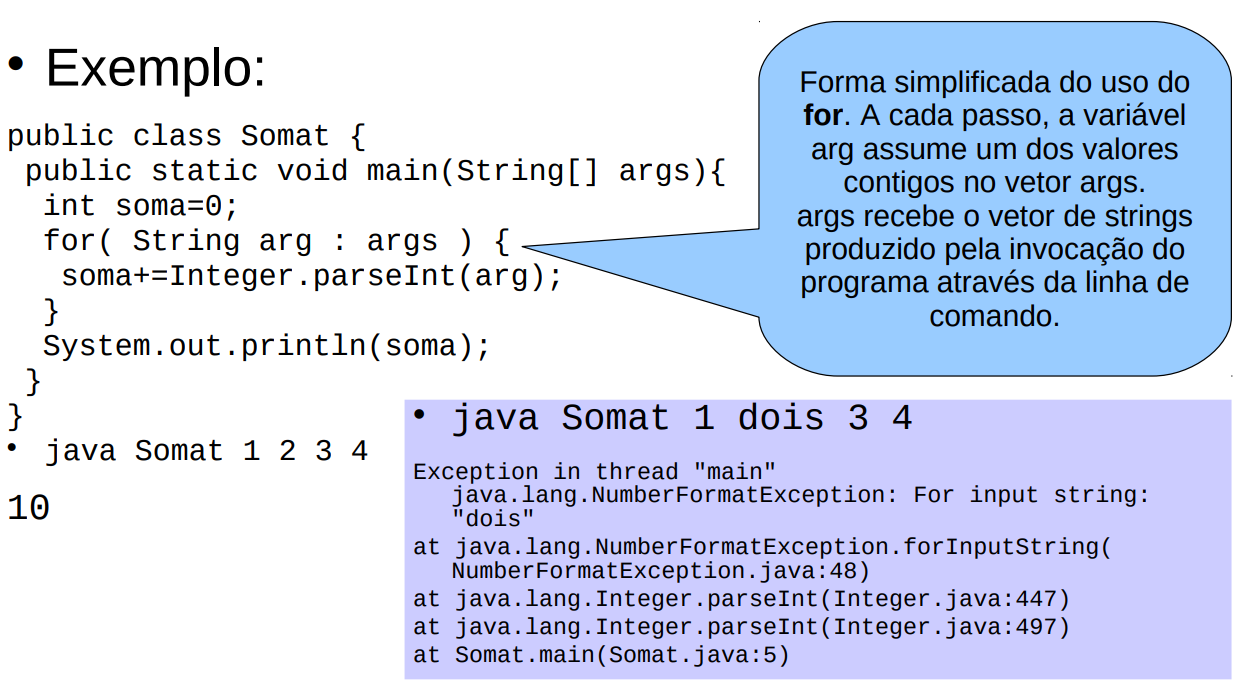
* A interface acessível através da referência polimórfica é a mesma da classe utilizada, não sendo possível usar de forma direta a interface das subclasses.
* Para acessar a interface da subclasses, temos que fazer um casting (mais especificamente downcast).

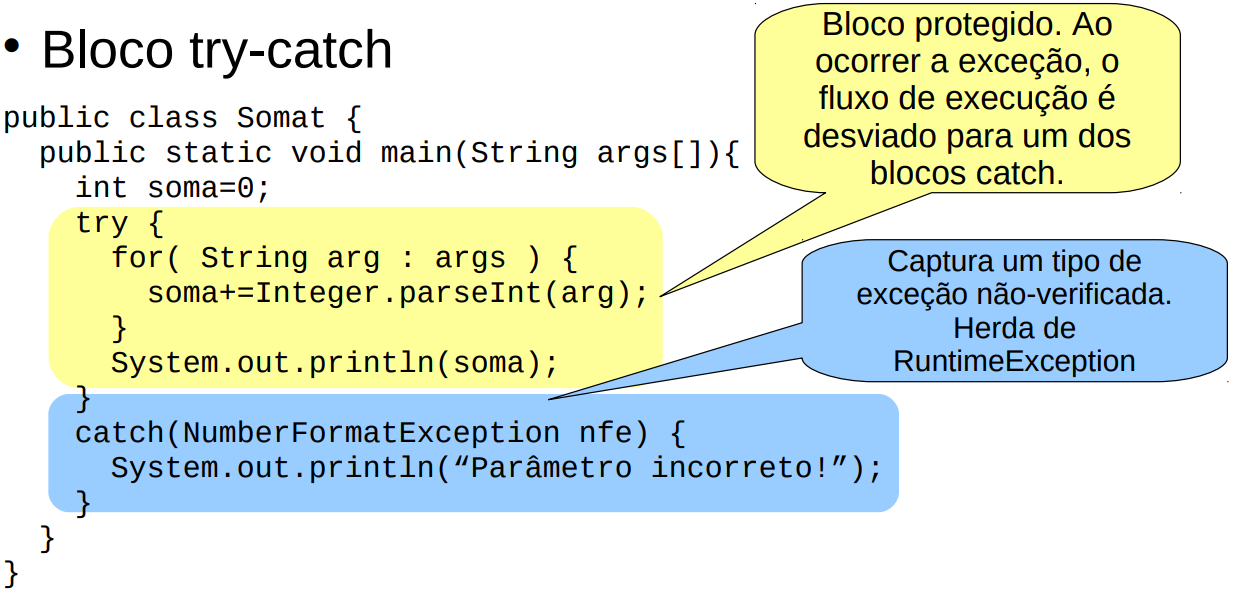


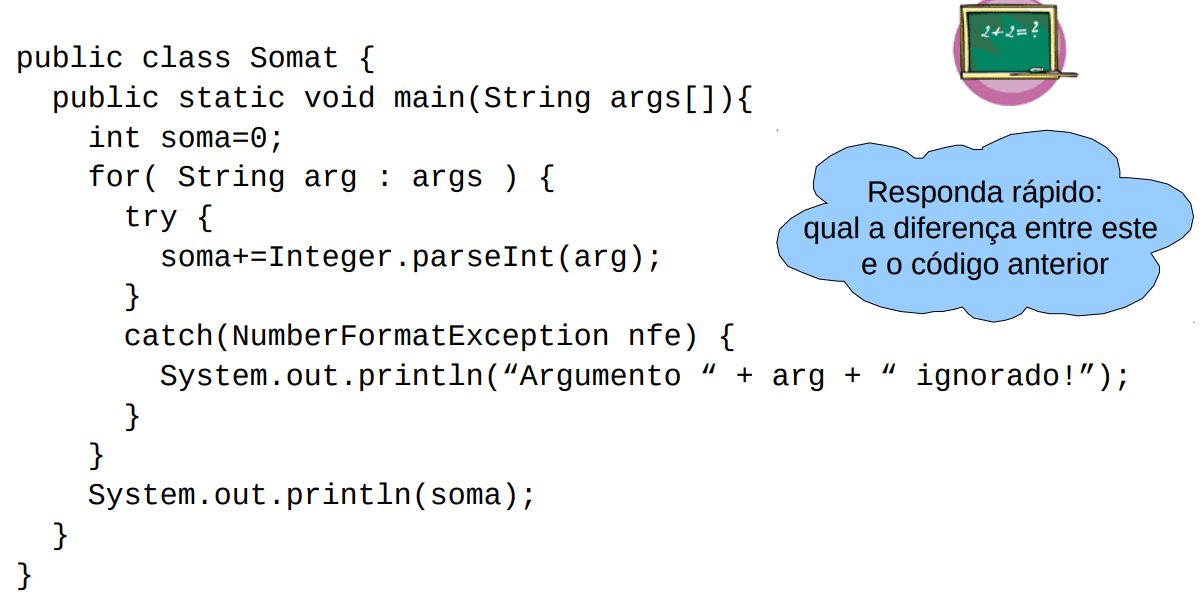
### Tratamento de Exceções

* **Exceção**: situação que foge ao cenário típico durante a execução de um código
  + Verificadas: exceções que devem obrigatoriamente ser tratadas (que herdam diretamente de Exception);
  + Não verificadas: não é obrigatório tratar. Para alguns tipos de erros, não existe tratamento típico, podendo causar a parada brusca da execução do código (Error e ‏(RuntimeException).
* A documentação da API Java cita as exceções que podem ser disparadas por um método. Ver exemplo da classe Integer.parseInt









* Tratando várias classes de erro:

try {

/\* código que pode disparar uma ou mais exceções \*/

}

catch (MinhaExcecao e1) {

/\* Código a ser executado se a exceção for da classe

MinhaExcecao \*/

}

catch (MinhaOutraExcecao e2) {

/\* Código a ser executado se a exceção for da classe

MinhaOutraExcecao \*/

}

catch (Exception e3) {

/\* Qualquer outra exceção é tratada aqui, uma vez que

todas as outras exceções são herdadas de Exception \*/

}

* Mecanismo da pilha de execução:
  + Se uma exceção é disparada e não existe tratamento (try..catch)‏, a exceção é jogada para o método chamador, e assim sucessivamente;
  + Se a exceção voltar ao método main e não for tratada também nele, o programa aborta;
  + A pilha de execução (stackTrace)‏ é mostrada no console na ocorrência de erro fatal.
* Cláusula finally
  + Opcional e apenas um por bloco try..catch
  + É sempre executada, ocorrendo ou não exceção

try {

abrirTorneira();

iniciarAquecimento();

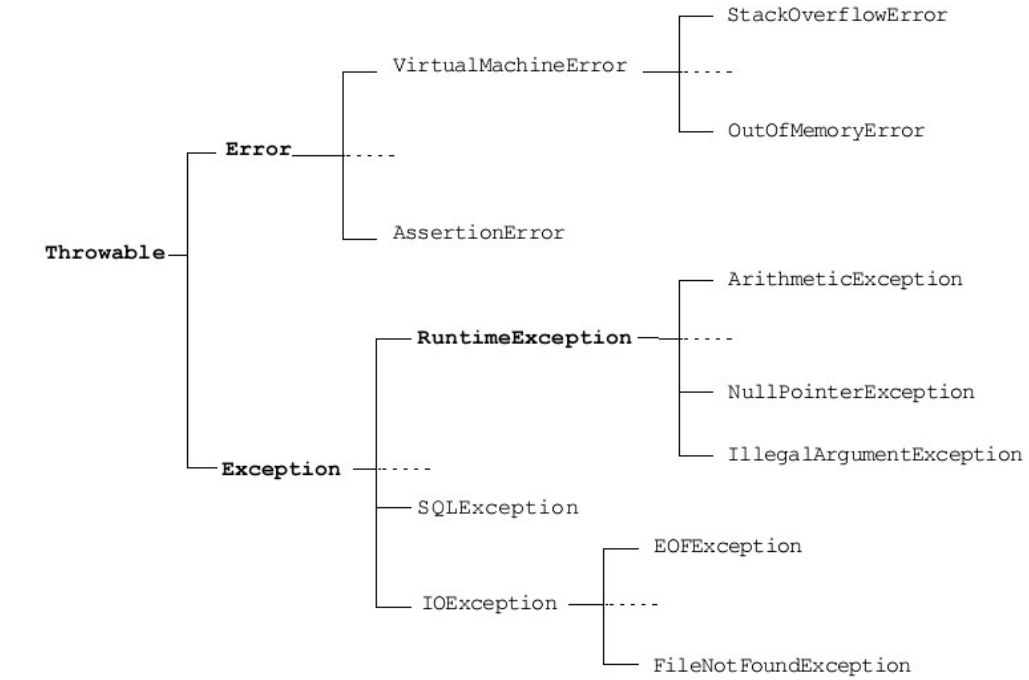
} catch(SuperAquecimentoException e) {

exibirProblema(e);

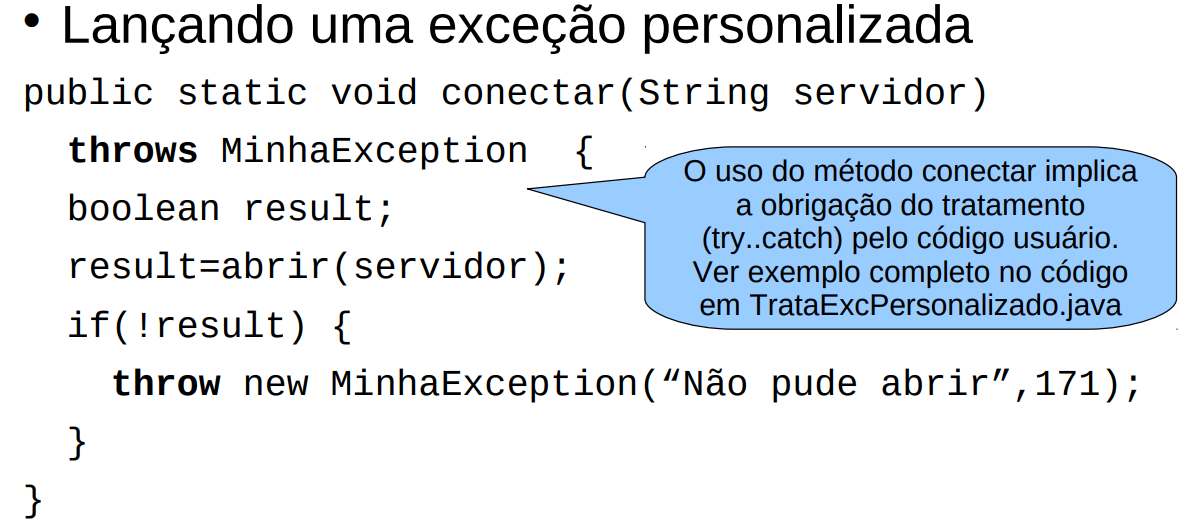
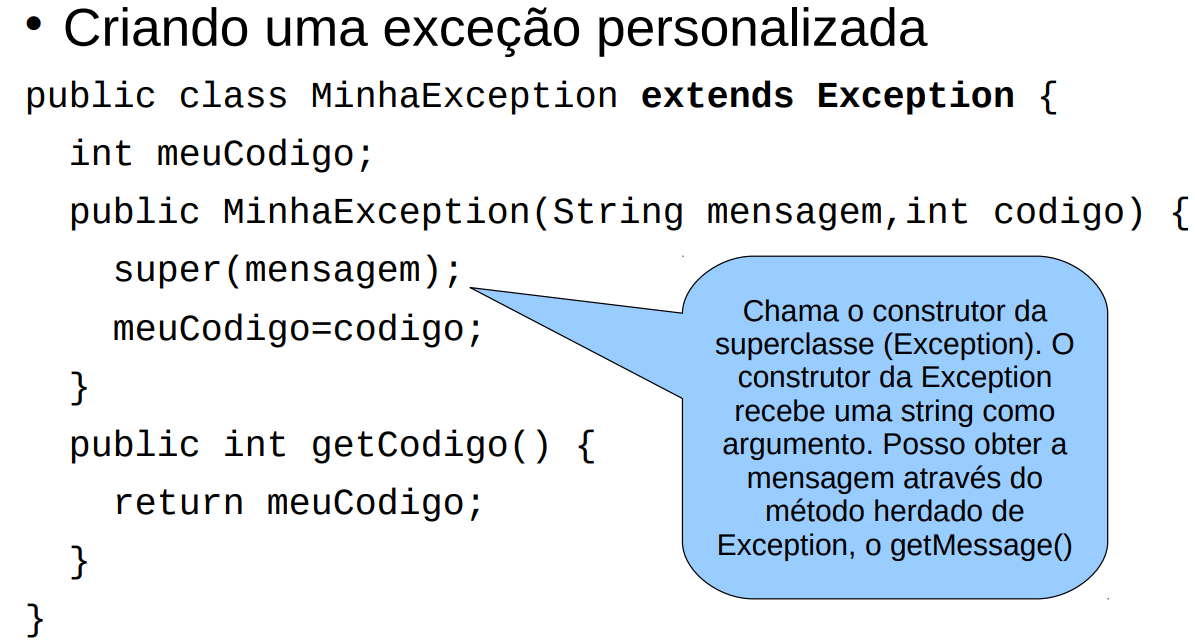
} finally {

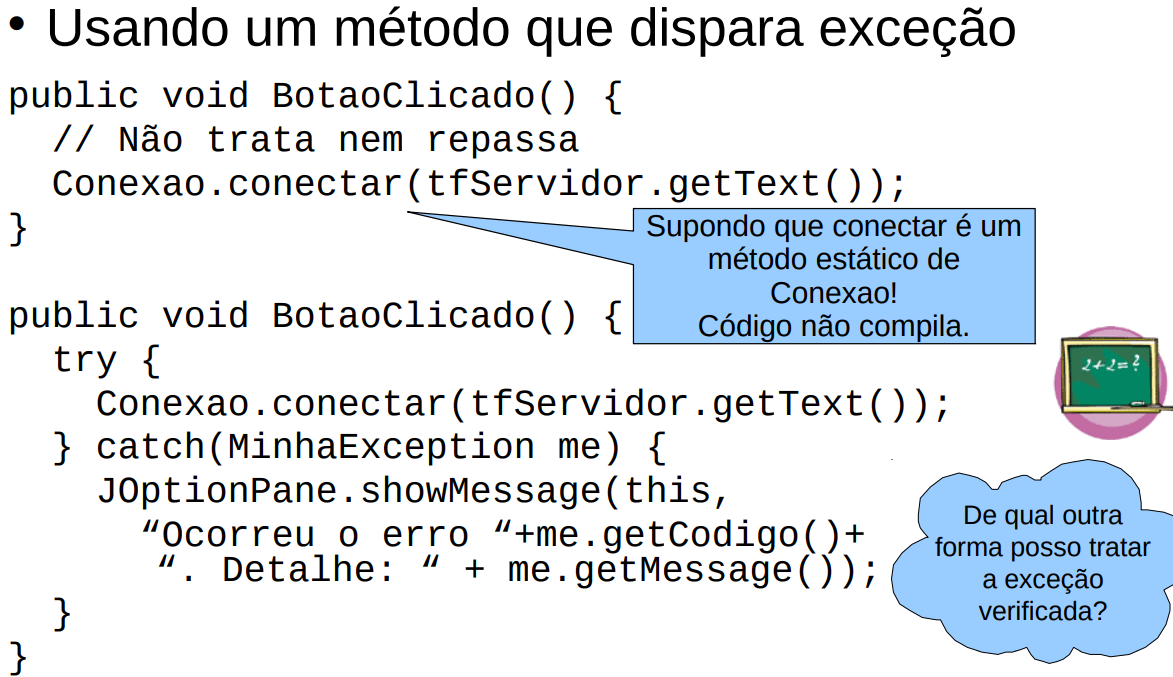
fecharTorneira();

}

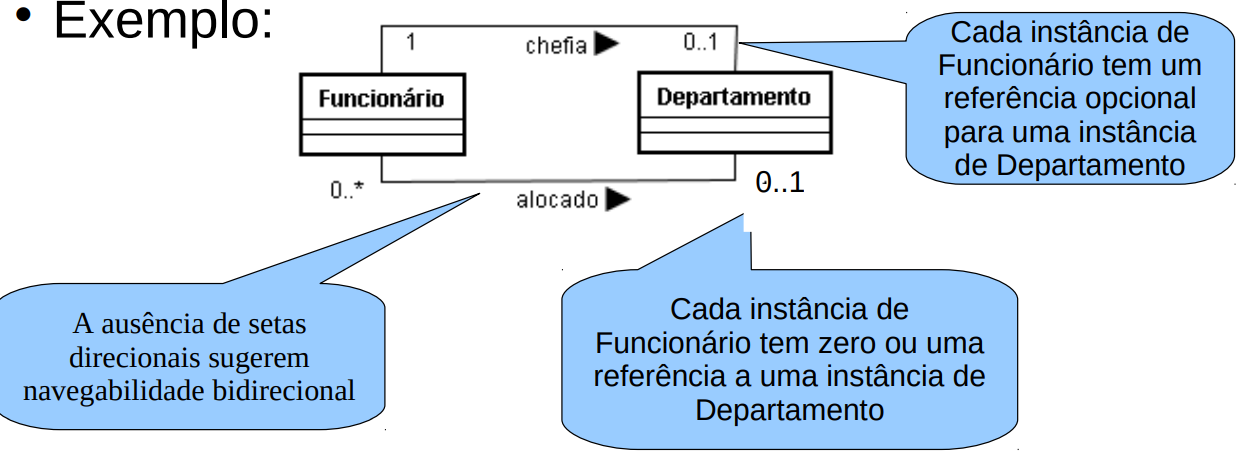
* Categorias de Exceção:
* Exceções mais comuns
  + NullPointerException, FileNotFoundException, NumberFormatException, ArithmeticException, SecurityException.
* Mecanismo trata ou repassa
  + Compilador força tratamento de exceções verificadas;
  + Claúsula throws: especifica as exceções que um método pode lançar e força que o código chamador contenha tratamento para cada exceção declarada

void metodoA() throws IOException, MinhaExcecao {...}

O código que chama metodoA deve estar num bloco try..catch ou então repassar (throws)‏ a exceção.



## Delegação e Composição

* Composição:
  + Trata da separação de classes, tornando-as coesas, e utilizadas através de relacionamentos TODO-PARTE.
  + Estudo de caso: modelar a classe Teclado contida numa classe Computador?
* Delegação:
  + Consiste em delegar a responsabilidade de uma operação para os objetos relacionados.
* Associação:
  + Representa um conjunto de relacionamentos que existem entre objetos de classes diferentes.

public class Funcionario {

private int matricula;

private String nome;

private Departamento alocado;

private Departamento chefia;

public Funcionario(int mat, String nome, Departamento depto) {

matricula=mat;

this.nome=nome;

alocado=depto;

chefia=null; // explícito

}

}

public class Departamento {

private int codDepto;

private String nome;

private Funcionario chefe;

private Funcionario[] aloca;

public Departamento(int codDepto, String nome, Funcionario chefe) {

this.codDepto=codDepto;

this.nome=nome;

this.chefe=chefe;

aloca=null;

}

public void alocarFunc(Funcionario func) {

// adiciona func no vetor aloca

}

}

## Array x Coleções

* Array:
  + Tamanho estático, manipulação elemento-a-elemento, por índice.
* Coleções:
  + Tamanho dinâmico, diferentes maneiras de manipular os elementos;
  + Objeto que agrupa múltiplos objetos-elemento dentro de si;
  + Usados para armazenar, recuperar e manipular dados, bem como transmiti-los entre objetos;
  + Java suporta coleções de Object (pré-generics, pré-java 5)‏ e de um tipo especializado (usando Generics)‏.

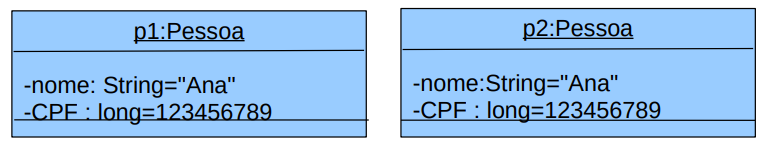
## Classe Object

* boolean equals(Object o)‏: define se dois objetos são equivalentes em significado.
  + Exemplo:

Integer i1=new Integer(7)‏ ;Integer i2=new Integer(7)‏ ;String s="";

if(i1==i2)‏ s += "i1==i2 sao iguais! "; else s += "i1!=i2! ";

if(i1.equals(i2)‏(‏ s += "i1 equals i2! "; else s += "i1 not equals i2! ";

* Quando se cria uma classe, deve-se definir o que será determinante na equivalência (sobreposição do equals)‏, caso contrário objetos diferentes serão sempre considerados diferentes. Exemplo: classe Pessoa.
* Quando se deve sobrepor o equals?
  + Quando o objeto é usado como chave de busca numa das coleções (Sets).

Exemplo na classe Pessoa

public boolean equals(Object o) {

if ( (o instanceof Pessoa) && ((Pessoa)o).getCPF() == this.getCPF() ) {

return true;

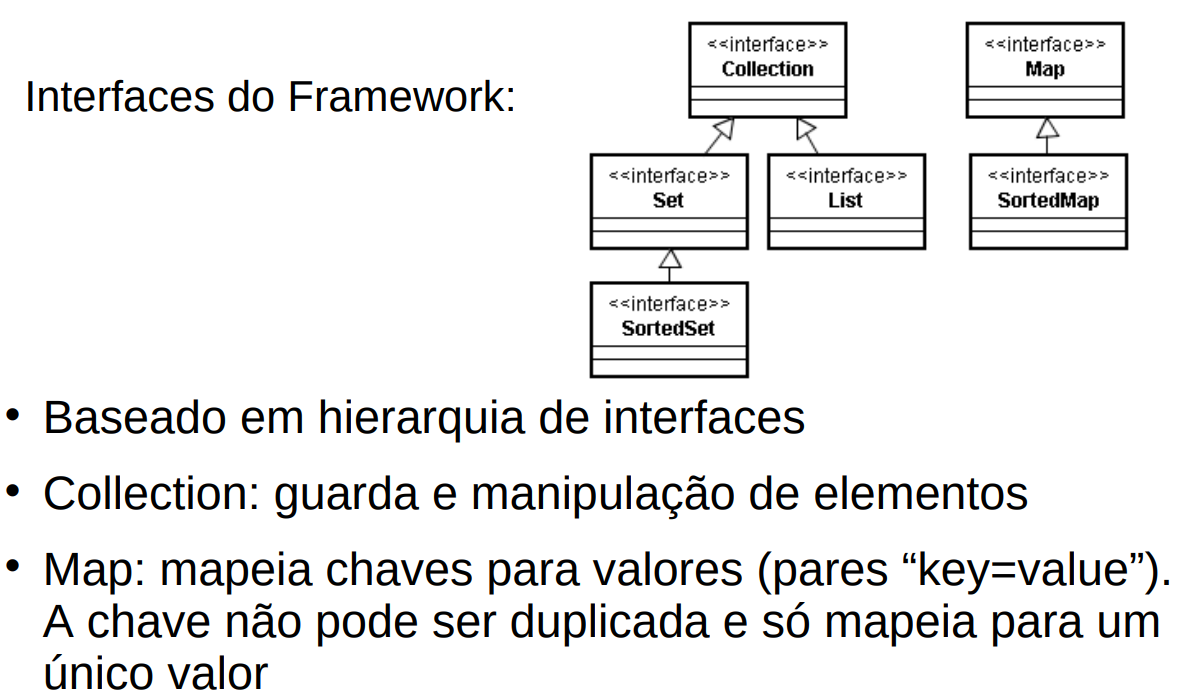
} else {

return false;

}

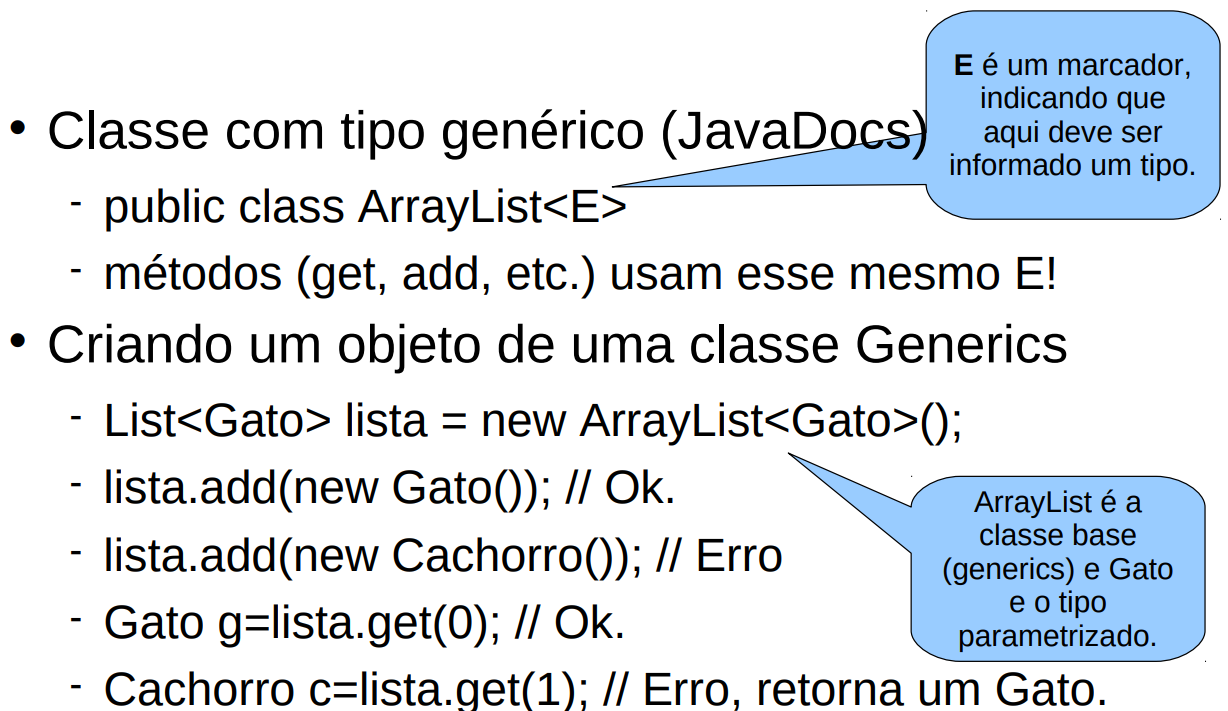
}

* int hashCode()‏: retorna um valor inteiro que é o hash do objeto, para uso com coleções com esse recurso;
  + Não é um número único para cada objeto. Vários objetos diferentes (!equals)‏ podem ter o mesmo código de hash
  + Objetos iguais devem retornar o mesmo hash.
* Framework de Coleções Java
  + Biblioteca Java de classes e interfaces reutilizáveis para manipulação de coleções (em java.util)‏;
  + Reduz o esforço de programação, fornecendo qualidade, velocidade e interoperabilidade.
* As coleções só armazenam tipos de referência, não os primitivos. Alternativa: os wrappers.
* Interfaces com duas hierarquias distintas: Collection e Map.



* Interface Collection (principais métodos)‏
  + Métodos: int size()‏, boolean isEmpty()‏, boolean contains(T)‏, boolean add(T)‏, boolean remove(T)‏, Iterator iterator()‏, boolean containsAll(Collection)‏, boolean addAll(Collection)‏, boolean removeAll(Colection)‏, clear()‏, T[] toArray()‏,
  + Interface Iterator: boolean hasNext()‏, T next()‏, ‏ remove ()

## Breve introdução ao Generics

* Denominação em Java para uma classe que pode receber um ou mais tipos parametrizados;
* Com o tipo específico, posso garantir que somente objetos que passem no teste do É-UM possam ser manipulados pela classe, evitando erros em tempo de execução.
  + Ou você quer que um Gato possa entrar na sua lista de Cachorros?
* 99% do tempo usamos Generics apenas nas classes de coleção do Java.
* Existe muito mais por trás do Generics
  + Não há polimorfismo para o tipo parametrizado, só para o tipo base

List lista=new ArrayList()‏ ;// Erro

* Um método por usar wildcard
  + public void metodo(List);
  + pode receber qualquer subtipo de Animal, por exemplo um List ou um List, mas só para leitura.
* Coleção List
  + objetos-elemento armazenados de forma ordenada;
  + Implementações
    - ArrayList
    - LinkedList

Exemplo:

List<Integer> al=new ArrayList<Integer>();

al.add(10); al.add(30); al.add(20);

for(Integer item:al) System.out.print(item+"-");

System.out.print(al.get(1));

* Interface List
  + Collection indexada, que permite elementos duplicados;
  + Permite acessar os elementos de forma indexada: T get(int)‏, T set(int,T)‏, void add(int,T)‏, T remove(int)‏, boolean addAll(int,Collection)‏, int indexOf(T)‏, int lastIndexOf(T)‏, ListIterator listIterator()‏, ListIterator listIterator(int)‏, List subList(int,int)‏
* Coleção Set
  + Não pode ter objetos-elemento duplicados
    - O método equals de um objeto já armazenado retorna true tendo como parâmetro o objeto sendo inserido.
  + Implementações:
    - HashSet: iteração entre os elementos de forma não ordenada;
    - LinkedHashSet: iteração entre os elementos de forma ordenada, pela ordem de inserção;
    - TreeSet: armazenamento, busca, e iteração ocorrem em um ordem especificada (ordem natural ou personalizada).
* Exemplo usando o HashSet

public class Main {

public static void main(String[] args) {

Set<String> unicos=new HashSet<String>();

Set<String> dups=new HashSet<String>();

for(String valor:args)

if(!unicos.add(valor)) dups.add(valor);

unicos.removeAll(dups);

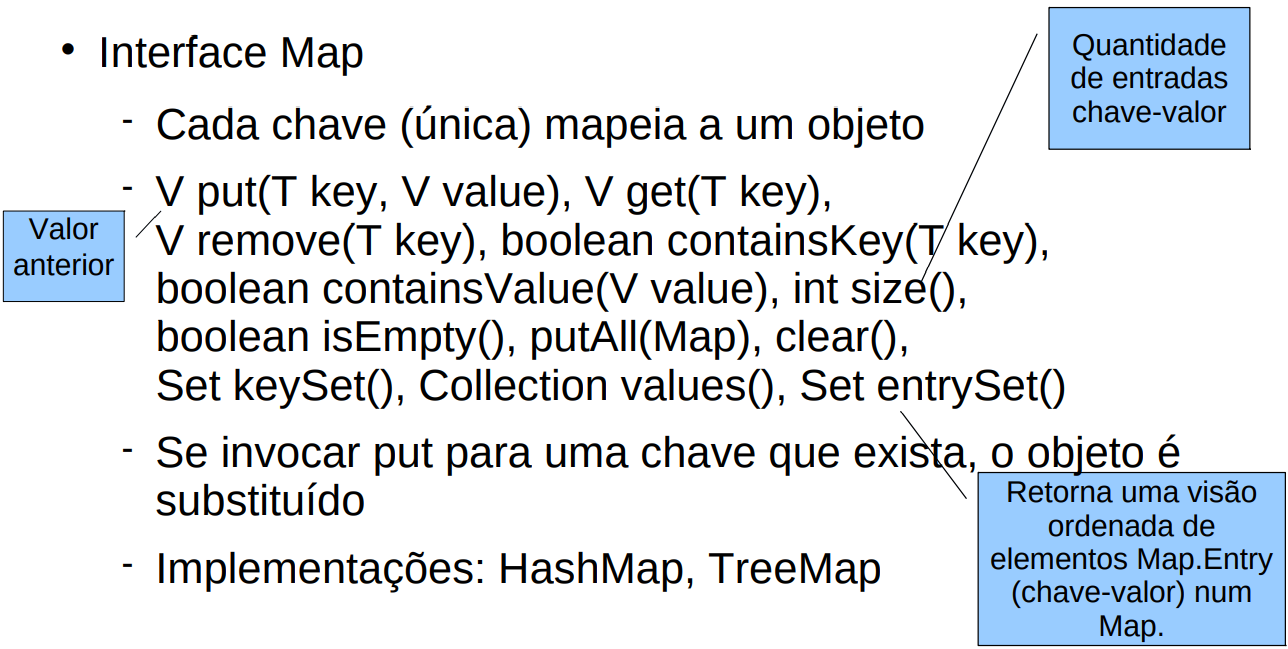
System.out.println(unicos);

System.out.println(dups);

}

}

* Navegação usando a interface ListIterator
  + Métodos: boolean hasNext()‏, T next()‏, boolean hasPrevious()‏, T previous()‏, int nextIndex()‏, int previousIndex()‏, remove()‏, set(T)‏, add(T)
  + Implementações: ArrayList



import java.util.\*;

public class TestaTreeMap {

public static void main(String[] args) {

Map<String,Integer> m=new TreeMap<String,Integer>();

for(int i=0;i<args.length;i++) {

Integer freq=m.get(args[i]);

m.put(args[i],freq==null ? 1 : ++freq );

}

System.out.println(m.size()+" tokens distintos.");

Set<String> s=m.keySet();

for(String str:s) {

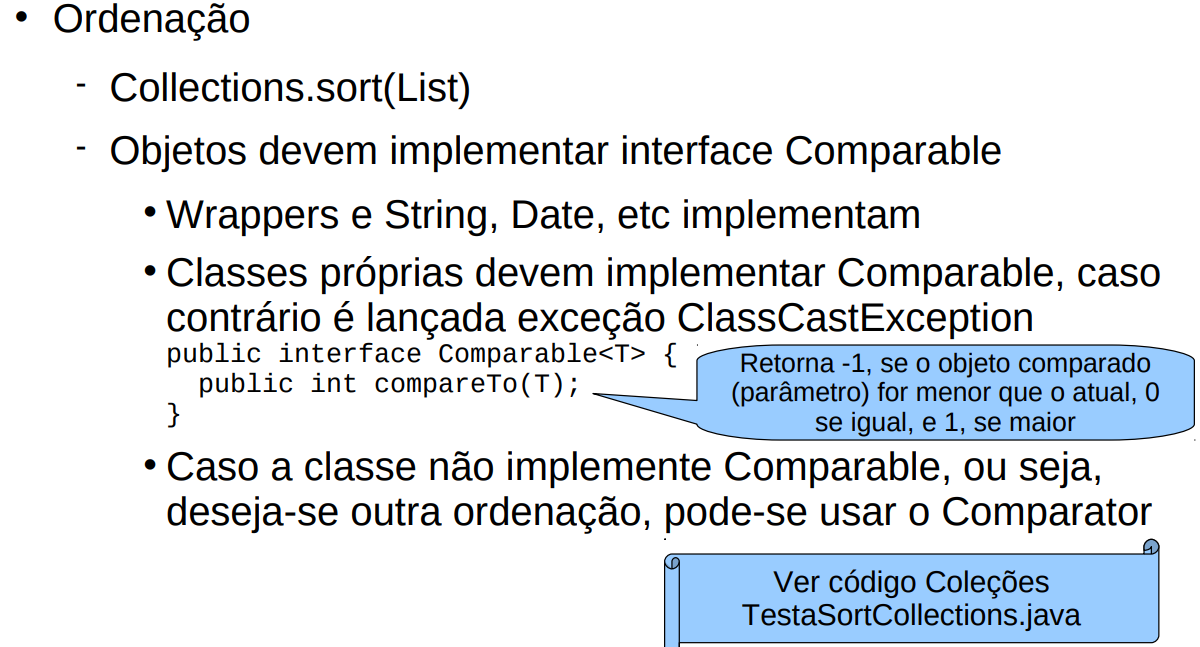
System.out.println(str+":"+m.get(str));

}

}

}

// invocado com:

// $ java TestaTreeMap ana pedro ana augusto pedro ana leo

* Interface SortedSet (extensão de Set)
  + Elementos ordenados por ordem padrão ou definido em um Comparator;
  + Deve-se usar a implementação TreeSet;
  + SortedSet subSet(T deElemento,T ateElemento);
  + SortedSet headSet(T e)‏, SortedSet tailSet(T e);
  + T first()‏, T last()‏, Comparator comparator()‏;
  + O Iterator (de Set)‏ varre o em ordem;
  + O toArray()‏ retorna elementos ordenados.
* Interface SortedMap (extensão de Map)
  + Elementos ordenados pela chave na ordem padrão ou definido em um Comparator;
  + Deve-se usar a implementação TreeMap;
  + SortMap subSet(T deElemento,T ateElemento)‏;
  + SortMap headSet(T e)‏, SortMap tailSet(T e);
  + T first()‏, T last()‏, Comparator comparator();
* Detalhes das Implementações das coleções
  + Vetor redimensionável (implementa List)‏
    - ArrayList;
    - LinkedList (usa lista duplamente encadeada).
* Usam tabela de Hash
  + HashSet (Interface Set)‏ e HashMap (interface Map);
  + Exigem sobreposição do equals e hashCode
* Árvore Rubro-Negra:
  + TreeSet e TreeMap (interface Set e Map, respectivamente)‏;
  + Exigem que os elementos sejam comparáveis.
* ArrayList x LinkedList
  + ArrayList: acesso posicional mais rápido;
  + LinkedList: melhor quando tem inserções/remoções no meio da lista.
* HashSet x TreeSet
  + HashSet: melhor performance em todos os casos;
  + TreeSet: ordenação. Para uso com SortedSet.
* HashMap x TreeMap
  + HashMap: geralmente a melhor;
  + TreeMap: para uso com SortedMap.
* Classe Collections
  + .sort(List)
  + .sort(List,Comparator)‏
  + .shuffle(List)
  + .reverse(List)‏
  + .fill(List,T)
  + .copy(List dest,List ori)
  + .binarySearch(List,T)‏
  + .min(Collection)‏
  + .max(Collection)