Paradigmas de Linguagens de Programação

Prof. Sergio D. Zorzo
Departamento de Computação - UFSCar
1º semestre / 2013
Aula 9

Material adaptado do Prof. Daniel Lucredio

Sobrescrita de métodos

Sobrescrita de métodos

- Segunda característica “marcante” da POO
  - Quando a subclasse altera o comportamento herdado da superclasse
- Em termos de código:
  - Método da superclasse é redefinido na subclasse
  - Mesmo cabeçalho (nome, tipo de retorno, parâmetros e exceções)
  - Corpo diferente

Sobrescrita de métodos

- Ex:

```
class Animal {
  public void examinar() {
    System.out.println("Examinando animal");
  }
}
class Cachorro extends Animal {
  public void examinar() {
    System.out.println("Examinando cachorro");
  }
}
...
Cachorro c = new Cachorro();
c.examinar();
```

Sobrescrita de métodos

- Ex:

```
class Animal {
  public void examinar() {
    System.out.println("Examinando animal");
  }
}
class Cachorro extends Animal {
  public void examinar() {
    System.out.println("Examinando cachorro");
  }
}
class Gato extends Animal {
  public void examinar() {
    System.out.println("Examinando gato");
  }
}
...
Cachorro c = new Cachorro();
c.examinar();
Animal a = c;
a.examinar();
```

Sobrescrita de métodos

```
examinar(Animal[] animais) {
  for(Animal a: animais) {
    a.examinar();
  }
}
Animal[] lista = { new Cachorro(), new
 Cachorro(), new Gato(), new Gato(),
 new Cachorro() };
examinar(lista);
```

Classes e métodos abstratos

Classes e métodos abstratos

- Em uma hierarquia de classes, pode haver comportamentos que só fazem sentido a partir de uma determinada classe da hierarquia
- Ex:

```
class FiguraGeometrica {
  public double calcularArea() {
    ... // O que colocar aqui?
  }
}
class Quadrado extends FiguraGeometrica {
  double lado;
  public double calcularArea() {
    return lado * lado;
  }
}
class Circulo extends FiguraGeometrica {
  double raio;
  public double calcularArea() {
    return Math.PI * raio * raio;
  }
}
```

Classes e métodos abstratos

- Solução:

```
abstract class FiguraGeometrica {
  public abstract double calcularArea();
}
class Quadrado extends FiguraGeometrica {
  double lado;
  public double calcularArea() {
    return lado * lado;
  }
}
class Circulo extends FiguraGeometrica {
  double raio;
  public double calcularArea() {
    return Math.PI * raio * raio;
  }
}
```

Classes e métodos abstratos

- Resultado
  - Torna-se impossível criar instâncias da classe FiguraGeometrica
  - Faz sentido, o que iria acontecer ao chamarmos calcularArea()?
  - Mas ainda é possível usar essa classe como um tipo

```
Quadrado q = new Quadrado();
q.lado = 3;
FiguraGeometrica fg = q;
double areaQuadrado = fg.calcularArea();
```

Classes e métodos abstratos

- Exemplo mais real

```
double areaTotal(FiguraGeometrica[]
  figuras) {
   double area = 0;
   for(FiguraGeometrica fg : figuras) {
     area += fg.calcularArea();
   }
}
...
FiguraGeometrica[] figs = { new
  Quadrado(), new Circulo(), ... };
double at = areaTotal(figs);
```

Classes e métodos abstratos

- Alguns pontos:
  - Se uma classe tem um método abstrato, ela também deve ser declarada como abstrata
  - Uma classe pode ter todos os seus métodos abstratos
  - Se uma subclasse herda de uma superclasse abstrata, ela deve OBRIGATORIAMENTE implementar os métodos abstratos
    - Ou também ser declarada como abstrata
  - A assinatura do método que implementa o método abstrato deve ser mantida

Interfaces

Interfaces

- Primeiro, vamos analisar do ponto de vista do código
- Uma interface é uma classe abstrata, com TODOS os métodos abstratos
  - Mas sem precisar usar a palavra “abstract”
- Uma interface pode ter atributos
  - Todos são automaticamente “public static final”
    - Ou seja, “constantes”
  - Todos devem ser inicializados na própria declaração

Interfaces

- Exs:

```
interface Mensuravel {
  public double calcularTamanho();
}
interface Compravel {
  public void comprar();
}
interface Emprestavel extends Compravel {
  public void emprestar();
  public void devolver();
}
```

Interfaces

- Uma classe pode implementar (e não estender) zero, uma ou mais interfaces
  - Nesse caso, ela deve OBRIGATORIAMENTE implementar todos os métodos da interface
    - Ou ser declarada como abstrata
- Ex:

```
class Caixa implements Mensuravel {
   double largura, altura, comprimento;
   public double calcularTamanho() {
      return largura * altura * comprimento;
   }
}
```

Interfaces

- Ex:

```
class Lavadora implements Compravel, Mensuravel {
  double largura, altura, comprimento;
  double precoSemDesconto;
  double desconto;
  public double calcularTamanho() {
    return largura * altura * comprimento;
  }
  public void comprar() {
    double precoReal = precoSemDesconto -
                       precoSemDesconto * desconto;
    // efetuar a compra
}
```

Interfaces

- Utilidade
  - Definir “facetas” para classes
  - Atribuir uma característica comportamental a uma classe
  - Permite reduzir o acoplamento, através de uma interface comum entre duas classes
    - Essa interface transfere a necessidade de conhecimento de uma classe com relação a outra
  - Em outras palavras:
    - A precisa usar B
    - Ao invés de usar B diretamente
    - Fazer A usar uma interface I
    - E fazer B implementar a interface I

Interfaces

- Normalmente utilizamos palavras terminadas em “ável”
  - Mensurável, Comprável, Emprestável
- Ex:

```
public interface Ordenavel {
    public enum Relacao { MAIOR,
                          MENOR, IGUAL };
    public Relacao compara(Ordenavel outro);
}

...
```

Interfaces

```
void ordena(Ordenavel[] lista) {
  for (int i = 0; i < lista.length; i++) {
    for (int j = i+1; j < lista.length; j++) {
      if (lista[i].compara(lista[j]) ==
                              Ordenavel.Relacao.MAIOR) {
        Ordenavel temp = lista[i];
        lista[i] = lista[j];
        lista[j] = temp;
      }
    }
  }
}
```

Interfaces

```
public class Animal implements Ordenavel {
    int peso;

    public Relacao compara(Ordenavel outro) {
        Animal outroAnimal = (Animal)outro;
        if(peso < outroAnimal.peso)
            return Relacao.MENOR;
        if(peso > outroAnimal.peso)
            return Relacao.MAIOR;
        return Relacao.IGUAL;
    }
}
```

Interfaces

```
...
Animal[] animais = {new Animal(),
                    new Animal(),
                    new Animal(),
                    new Animal(),
                    new Animal()};
animais[0].peso = 2;
animais[1].peso = 1;
animais[2].peso = 10;
animais[3].peso = 5;
animais[4].peso = 1;
ordena(animais);
for(Animal a : animais) {
  System.out.println(a.peso);
}
```

Interfaces

```
public class Produto implements Ordenavel {
  double preco;

  public Relacao compara(Ordenavel outro) {
    Produto outroProduto = (Produto)outro;
    if(preco < outroProduto.preco)
      return Relacao.MENOR;
    if(preco > outroProduto.preco)
      return Relacao.MAIOR;
    return Relacao.IGUAL;
  }
}
```

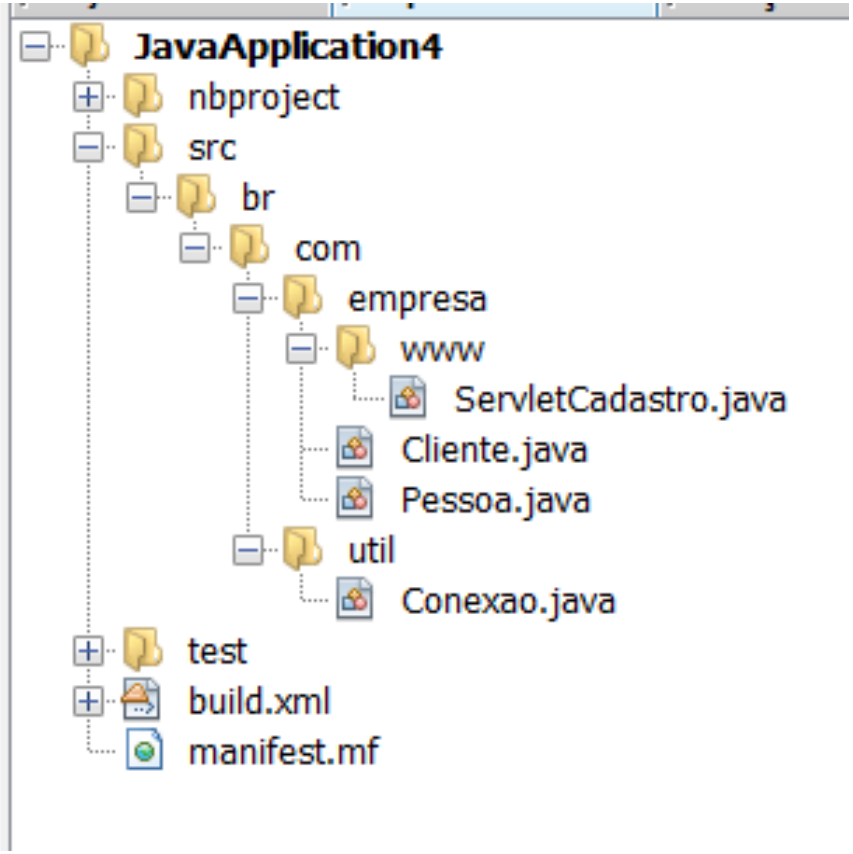
Pacotes

Pacotes

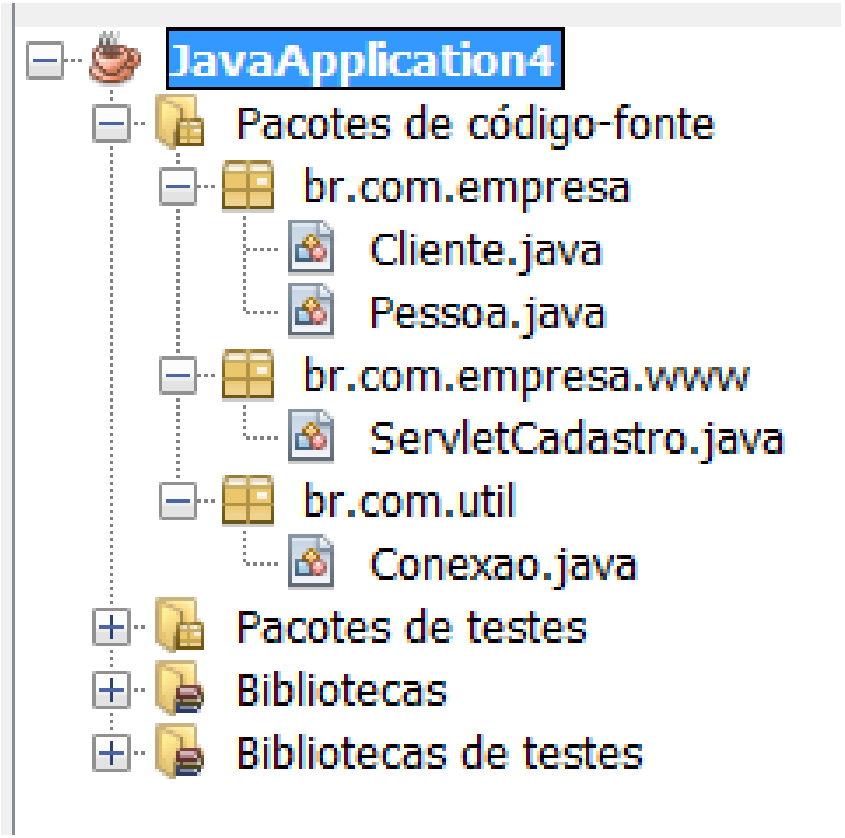
- Forma de organização das classes
  - Imita a estrutura de diretórios do sistema de arquivos
    - Com ponto ao invés de barra
    - A partir de uma pasta inicial (contexto)
- Permite agrupar classes relacionadas
- Pacotes podem ser organizados em uma hierarquia
  - Pacotes → subpacotes → subsubpacotes

Pacotes

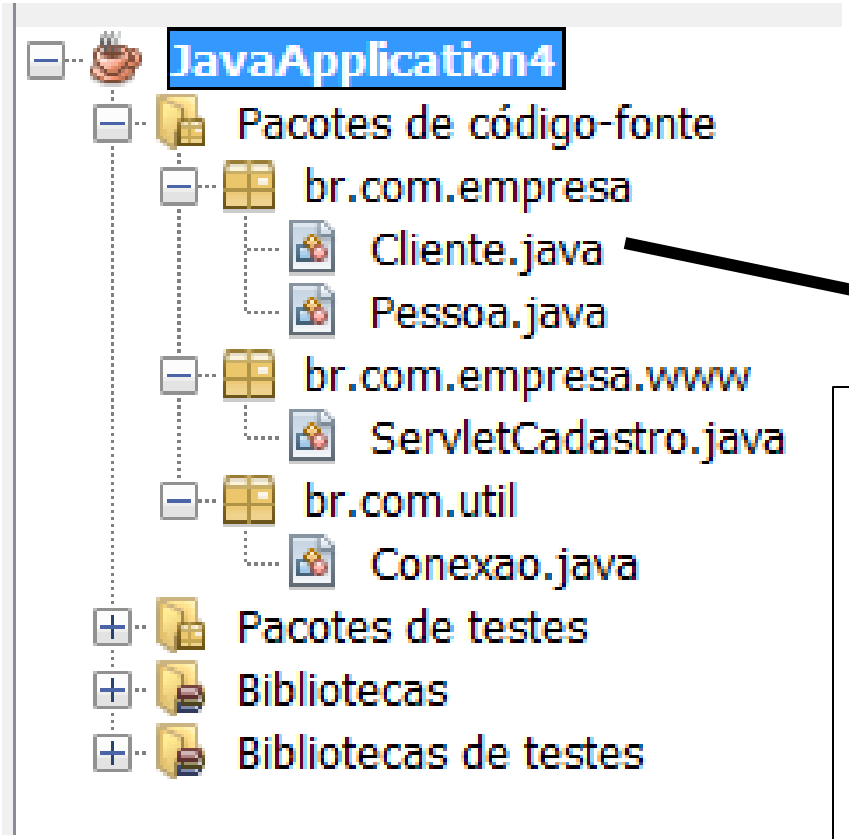
Estrutura real de diretórios



Estrutura de pacotes



Pacotes




```
package br.com.empresa;


public class Cliente {


}
```

Pacotes

- Para utilizar classes do mesmo pacote
  - Basta utilizar normalmente
- Ex:

```
package br.com.empresa;

public class Cliente {
    void realizarAluguel() {
        Pessoa fiador = new Pessoa();
        // Classe Pessoa está no mesmo pacote que Cliente
    }
}
```

Pacotes

- Para utilizar classes de outro pacote
  - Deve-se utilizar o nome completo
    - Mesmo que esteja em um subpacote
- Ex:

```
package br.com.empresa;


public class Cliente {
 void realizarAluguel() {
  Pessoa fiador = new Pessoa();
  br.com.util.Conexao c = new
                          br.com.util.Conexao();
  br.com.empresa.www.ServletCadastro sc = new
                  br.com.empresa.www.ServletCadastro();
    }
}
```

Pacotes

- Para utilizar classes de outro pacote
  - Pode-se também adicionar uma cláusula “import” no início do arquivo
- Ex:

```
package br.com.empresa;

import br.com.empresa.www.ServletCadastro;
import br.com.util.Conexao;

public class Cliente {
    void realizarAluguel() {
        Pessoa fiador = new Pessoa();
        Conexao c = new Conexao();
        ServletCadastro sc = new ServletCadastro();
    }
}
```

Pacotes

- Para utilizar classes de outro pacote
  - Pode-se também adicionar uma cláusula “import” com “*” para importar todas as classes de um pacote
    - Mas não é recomendado, pois dessa forma as dependências não ficam explícitas
  - Obs: “*” não faz importação de subpacotes!
- Ex:

```
package br.com.empresa;
import br.com.empresa.www.*;
import br.com.util.*;
public class Cliente {
    void realizarAluguel() {
        Pessoa fiador = new Pessoa();
        Conexao c = new Conexao();
        ServletCadastro sc = new ServletCadastro();
    }
}
```

Pacotes

- Modificador default (sem modificador)
  - Estende a visibilidade para todo o pacote
    - Ou seja, classes do mesmo pacote conseguem acessar atributos e métodos sem modificador
- Modificador protected
  - Estende a visibilidade default para as subclasses
    - Ou seja, classes do mesmo pacote conseguem acessar atributos e métodos marcados com “protected”, além das subclasses, ainda que estejam em outro pacote

Pacotes

- A declaração de pacotes deve ser a primeira instrução de um arquivo
- Não é recomendável criar classes sem pacotes
  - Apesar de ser possível
- Não é possível importar duas classes com o mesmo nome
  - Ex:
    - import java.util.Date
    - import java.sql.Date
  - Neste caso, uma delas deve ser acessada com o nome completo

Tratamento de exceções

Tratamento de exceções

- Pode-se pensar em um determinado programa/subprograma em termos de fluxos
  - Fluxo normal (quando tudo “dá certo”)
  - Fluxos alternativos (quando algo difere do fluxo normal)
- Maneira natural de enxergar um algoritmo
  - Claro, podem existir desvios de fluxo dentro do fluxo normal
- Como desviar entre fluxo normal/alternativo?
  - If, while, for, switch...
- Problema:
  - Misturar fluxo normal com alternativos reduz a legibilidade, manutenibilidade, flexibilidade, etc.

Tratamento de exceções

- Em JAVA (e outras linguagens) existe uma forma de fazer isso
- Ex:

```
class EntradaSaida {
   public void criarArquivo(String nome) {
      File f = new File(name);
      f.createNewFile(); // pode acontecer
                         // uma exceção aqui
      ... outros comandos
   }
}
```

Tratamento de exceções

```
class EntradaSaida {
  public void criarArquivo(String nome) {
    try {
      File f = new File(name);
      f.createNewFile();
      ... outros comandos
    } catch (IOException ioe) {
      ... código de tratamento de erro
    }
  }
}
```

Tratamento de exceções

```
class EntradaSaida {
  public void criarArquivo(String nome) throws
  IOException {
    File f = new File(name);
    f.createNewFile();
    ... outros comandos
  }
}
...
EntradaSaida es = new EntradaSaida();
try {
  es.criarArquivo("bla");
} catch (IOException ioe) {
  ... código para tratar exceção
}
```

Tratamento de exceções

- Podem haver múltiplas exceções
- Ex:

```
try {
   int[] a = new int[3];
   a[10] = 5;
   int i = 10;
   int j = 0;
} catch(ArithmeticException ae) {
   ...
} catch(ArrayIndexOutOfBoundsException aioobe) {
   ...
}
```

Tratamento de exceções

- Bloco finally
  - Executado sempre
    - Caso dê tudo certo, após o término da execução do try
    - Caso haja uma exceção, após o tratamento da mesma
    - Obs: mesmo que houver um comando “return”, o bloco finally será executado

Tratamento de exceções

```
try {
  int[] a = new int[3];
  a[10] = 5;
  int i = 10;
  int j = 0;
  return;
} catch(ArithmeticException ae) {
  ...
} catch(ArrayIndexOutOfBoundsException aioobe) {
  ...
} finally {
  ...
}
```

Tratamento de exceções

- O compilador JAVA obriga o tratamento de algumas exceções
  - Exceções do tipo CHECKED
  - Classe Exception (e subclasses)
- Outras podem ser tratadas ou não
  - Exceções UNCHECKED
  - Classe RuntimeException (e subclasses)

Tratamento de exceções

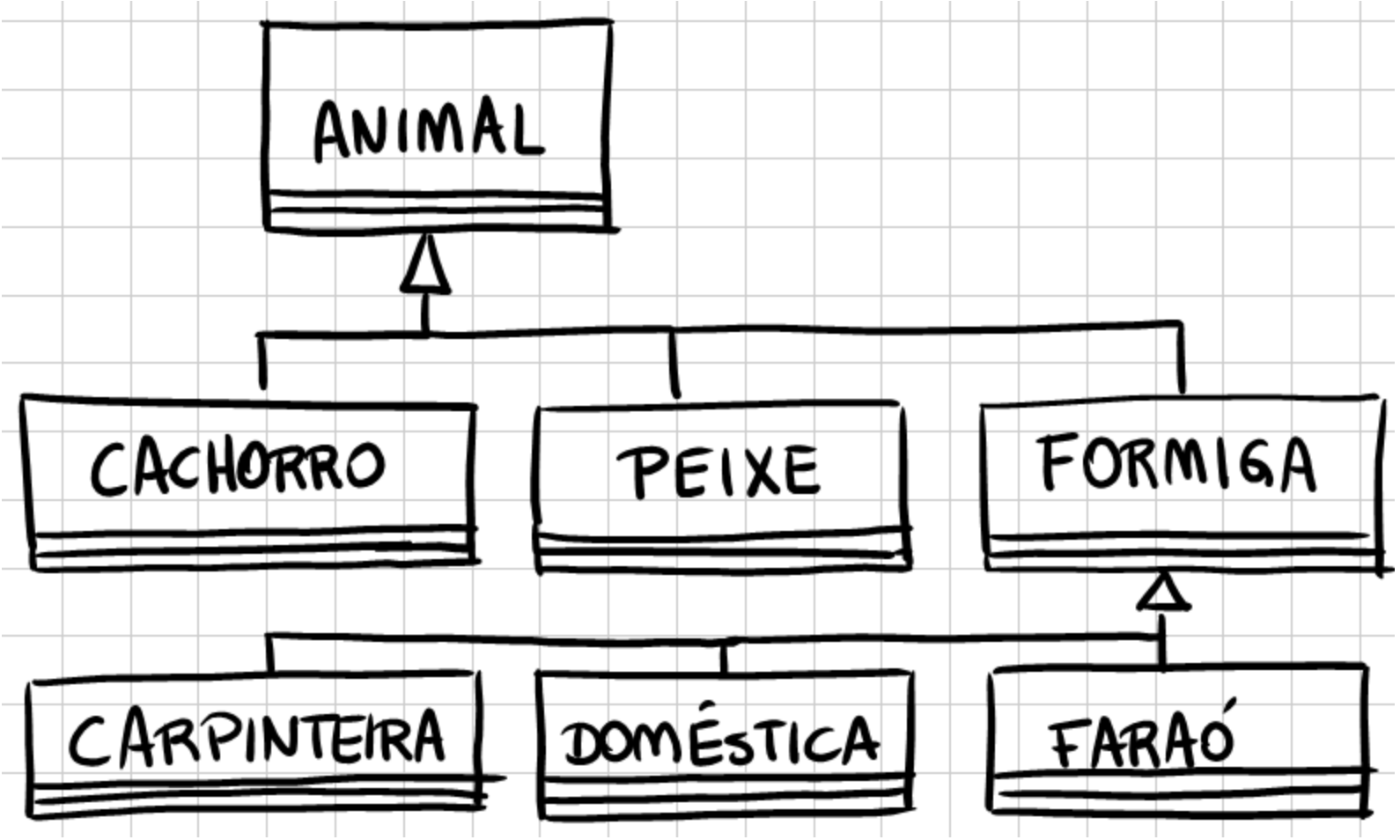
- As exceções declaradas com “throws” fazem parte da assinatura do método
  - Ou seja, devem ser repetidas caso o método seja sobrescrito em uma herança ou implementação de interface

Tipos genéricos

Tipos Genéricos

- Uma das mudanças mais esperadas da linguagem Java
  - Java 5
- Recurso para facilitar a escrita e leitura de código
- Duas formas:
  - Usuários de APIs, bibliotecas e frameworks são auxiliados por formas mais simples de utilização
  - Desenvolvedores de APIs, bibliotecas e frameworks tem um recurso a mais de modelagem de classes que flexibiliza com segurança o seu modelo

Tipos Genéricos



Tipos Genéricos

```
1.  Animal a1 = new Cachorro();
2.  Animal a2 = new Formiga();
3.  Animal a3 = new Carpinteira();
4.  Formiga f1 = new Formiga();
5.  Formiga f2 = new Carpinteira();
6.  Formiga f3 = new Animal();
7.  Peixe p1 = new Peixe();
8.  Peixe p2 = new Animal();
9.  Peixe p3 = new Cachorro();
```

Tipos Genéricos

```
public void cadastrar(Animal a) {
   // Código para fazer cadastro
}


...
Animal a1 = new Animal();
Cachorro c1 = new Cachorro();
cadastrar(a1); // OK
cadastrar(c1); // OK
```

Tipos Genéricos

```
public void vacinar(Cachorro c) {
   // Código para vacinar
}


...
Animal a1 = new Animal();
Cachorro c1 = new Cachorro();
vacinar(a1); // Erro!
vacinar(c1); // OK
```

Herança

- Um objeto de um tipo mais genérico consegue “armazenar” um tipo mais específico
- Com isso, é possível construir classes e métodos genéricos, que se aplicam a diferentes tipos de objetos

Herança

- Uma vez que você usou um tipo genérico para “guardar” um tipo específico, o seu tipo verdadeiro é “esquecido” pelo Java
  - Você precisa lembrá-lo, explicitamente
    - Animal a1 = new Cachorro();
    - Cachorro c1 = (Cachorro)a1;
  - Ou, se você também esqueceu, pode testar em tempo de execução
    - Animal a1 = new Cachorro();
    - if(a1 instanceof Cachorro) return true;

Herança

```
1.  Cachorro c1 = new Cachorro()
2.  Animal a1 = c1;
3.  Cachorro c2 = c1;
4.  Cachorro c3 = a1;
5.  Cachorro c4 = (Cachorro) a1;
5.  if(a1 instanceof Cachorro)
6.     System.out.println(“X”);
7.  if(a1 instanceof Animal)
8.     System.out.println(“Y”);
9.  if(c1 instanceof Animal)
10.    System.out.println(“Z”);
```

Herança

- Apesar de flexível e genérico, exige que você fique “lembrando” o Java sobre os tipos corretos
- Precisa fazer casting ()
- Precisa testar o tipo (instanceof)

Herança

- Ou seja, somente a capacidade de generalização da herança não garante 100% de segurança
  - Ainda exige algum trabalho
- Para isso, existem os tipos genéricos

Tipos genéricos

- O desenvolvedor de uma classe pode usar um tipo sem dizer exatamente qual
- O utilizador de uma classe diz o tipo que deseja usar

```
class C <T> {
    // aqui dentro posso usar T como
    // se fosse um tipo específico
}
```

Tipos genéricos

- É possível definir múltiplos tipos genéricos

```
class Caixa<T,E,X> {
   // uso T, E e X como se fossem
   // tipos específicos
}
...
Caixa<Integer,String,Integer> c1 =
   new Caixa<Integer,String,Integer>();
```

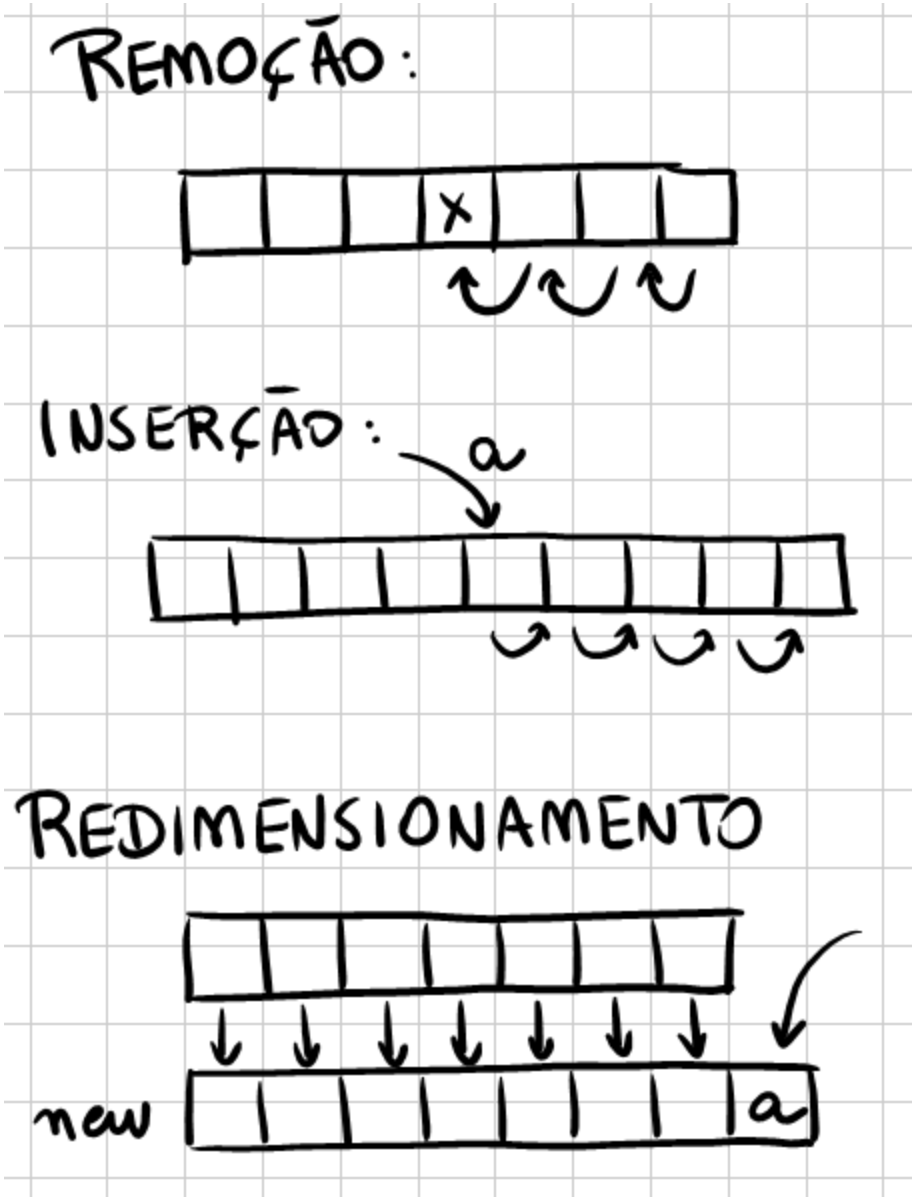
Coleções

Antes...

- Array é uma lista de variáveis do mesmo tipo
- Estrutura de dados do tipo coleção
- Tamanho é fixo (definido na inicialização)
- Variáveis do tipo array são na verdade referências
  - Ao contrário de variáveis de tipos primitivos
  - Ou seja, podem ser null

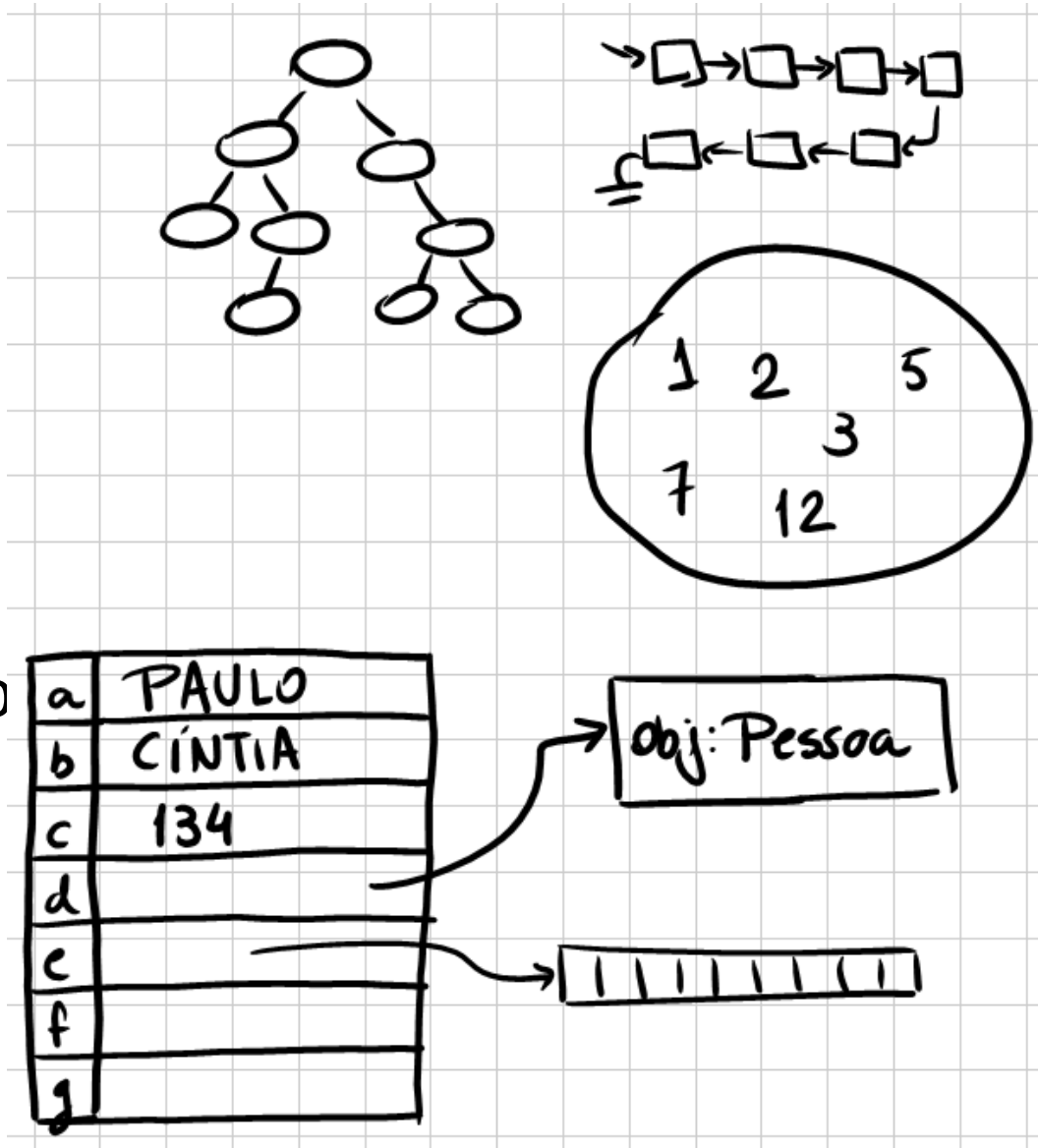
Arrays

- Na prática, arrays são limitados
- Muitas operações exigem trabalho extra



Outras estruturas

- Úteis em diversas situações
- Poderiam ser implementadas com arrays, mas daria muito trabalho



Coleções

- Todas essas estruturas tem uma característica em comum
  - São COLEÇÕES de valores
    - Ou melhor dizendo: objetos
- Java possui um framework
  - Diversos tipos de coleções
  - MUITAS funcionalidades já prontas
    - Ordenação, busca, redimensionamento, otimização, transformação de um tipo para outro, etc

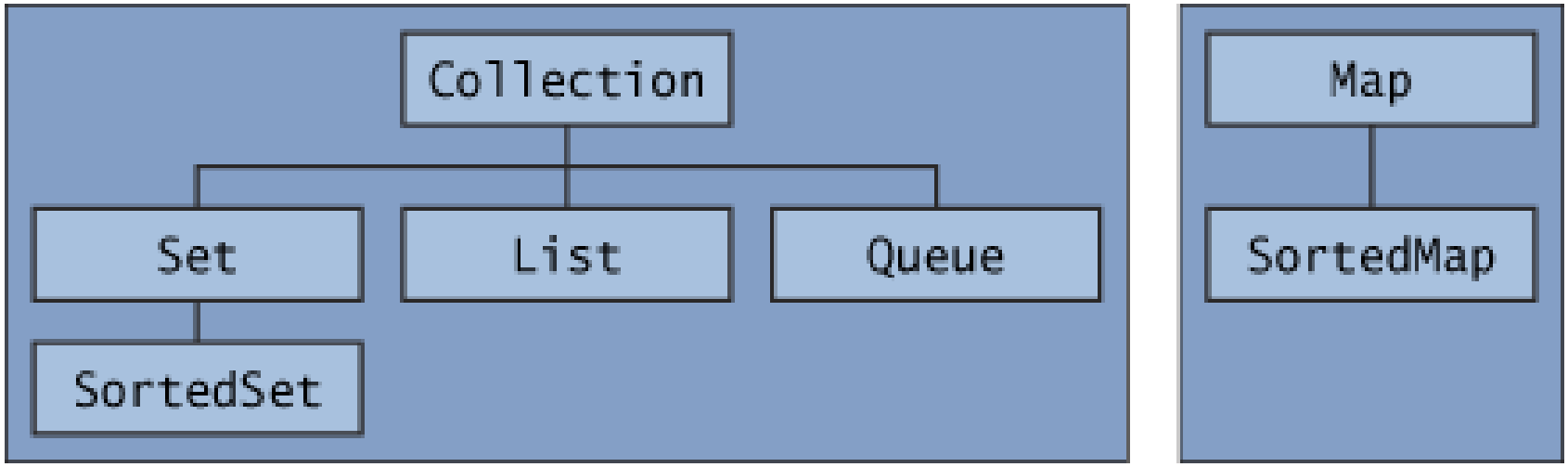
Benefícios

- Reduz esforço de programação
- Aumenta qualidade e produtividade
- Interoperabilidade
- Reduz esforço para aprender e desenvolver novas APIs
- Promove reutilização de software

java.util.Collections

- O framework é composto por:
  - Interfaces
  - Implementações
    - Algumas implementações legadas
  - Utilitários
    - Para arrays e coleções

Interfaces



Implementações

- As interfaces apenas representam tipos de coleções genéricos e as operações permitidas
- Ao criar uma nova coleção (new), é necessário definir uma implementação
- Dependendo da implementação:
  - Pode haver duplicação ou não
  - Elementos são ordenados ou não
  - Permite inserção de elemento null ou não

Implementações

Interfaces	**Implementações**				
	Hash table	**Array redimensionável**	**Árvore**	**Lista ligada**	**Hash table + lista ligada**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList (Vector)		LinkedList	
Queue					
Map	HashMap (Hashtable)		TreeMap		LinkedHashMap

Polimorfismo

- Uso de herança e polimorfismo
- Permite escrever código mais genérico e flexível
  - Sempre que possível, deve-se utilizar variáveis dos tipos mais genéricos possíveis
- Exs:

```
void folhaPagto(List funcionarios) { /* OK! */ }
void folhaPagto(ArrayList funcionarios) { /* Não! */ }
```

  - Assim, pode-se trocar de implementação sem causar muito impacto no código
    - Manutenção facilitada!

Coleções

- Toda coleção do framework faz uso de tipos genéricos
- Dessa forma, é possível (fortemente recomendado) declarar coleções especificando um tipo
- Ex:

```
List<String> l = new ArrayList<String>();
l.add("Eu sou uma string");
String s = l.get(13);
```

java.util.Collection

- Tipo mais genérico de coleção
  - Um pouco diferente de Map
- Características básicas de agrupamento de objetos
- Não se sabe se é ordenado ou não
- Não se sabe se permite duplicação ou não
- A implementação pode ser um conjunto, uma lista, uma fila...

Percorrendo coleções

- Utilizando for-each

```
for(String s:colecaoStrings) {
    System.out.println(s);
}
```

- Utilizando Iterator

```
Iterator<String> i = colecao.iterator();
while(i.hasNext()) {
  String s = i.next();
   System.out.println(s);
}
```

java.util.Collection

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);
  boolean remove(Object element);
  Iterator<E> iterator();
  containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c);
  boolean removeAll(Collection<?> c);
  boolean retainAll(Collection<?> c);
  void clear();
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

Modificando coleções

- Métodos add() e remove()
  - Funcionamento depende da implementação
  - Ex:
    - em um conjunto, add verifica se não há duplicatas
    - em uma lista, add insere no final
    - em uma fila, remove elimina o primeiro elemento

Interfaces mais específicas

- List, Set, Queue e Map
- Veremos cada uma em detalhes, já exemplificando com implementações

java.util.List

- Coleção **ordenada** de elementos
  - Definição necessária:
    - Ordered = ordenada
    - Sorted = classificada
- Ou seja, os elementos de uma lista estão em sequência, um após o outro
  - Mas não necessariamente classificada (Alfabeticamente, por exemplo)
- Permite duplicação de elementos

java.util.List

- Além das operações básicas “herdadas” de java.util.Collection
  - Define operações que consideram a ordenação dos elementos
  - Acesso posicional
  - Busca
  - Iteração específica de lista
  - Intervalos

java.util.List

```
public interface List<E> extends Collection<E> {
  E get(int index);
  E set(int index, E element);
  boolean add(E element);
  void add(int index, E element);
  E remove(int index);
  boolean addAll(int index, Collection<? extends E> c);
  int indexOf(Object o);
  int lastIndexOf(Object o);
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);
  List<E> subList(int from, int to);
}
```

Uso de List

- Parecido com array

```
String[] a = {“Alo”, “Mundo”, “!”};
a[1] = “Pessoal”;
String b = a[2];

List<String> l = new ArrayList<String>();
l.add(“Alo”); l.add(“Pessoal”); l.add(“!”);
l.set(1, “Pessoal”);
String c = l.get(2);
```

Uso de List

- Não tem uma inicialização “simples” como no caso de arrays
  - É necessário inserir um a um os elementos
    - Ou não!!! (Aguardem ...)
  - Ou utilizar um laço (for, while, etc)
- Método add permite inserir elementos em uma posição específica
  - A implementação irá movimentar o restante para acomodar o novo elemento

Uso de List

- Cuidado especial: remoção
  - Primeira opção: remove(int index)
    - Mais simples, remove o elemento na posição especificada
      - Se for um array, ajusta as posições para não deixar uma posição vazia
      - Se for uma lista ligada, apenas redireciona as referências

Uso de List

- Cuidado especial: remoção
  - Segunda opção: remove(Object o)
    - Faz uma busca pelo objeto, removendo quando o encontrar
    - A busca utiliza o critério definido no método equals da classe daquele objeto
    - Ex: String tem método equals implementado
    - Se quiser utilizar uma classe customizada, é necessário implementar o método equals adequadamente

Uso de List

- Cuidado especial: remoção
  - Opções anteriores não funciona enquanto a lista estiver sendo percorrida
  - Terceira opção: iterator.remove()
    - Remove o elemento sendo atualmente percorrido
    - Faz os ajustes necessários

Implementações de List

- ArrayList
  - Tem um array como base
- LinkedList
  - Baseado em lista ligada
  - Características de fila e pilha
    - Aliás, LinkedList implementa List e Queue ao mesmo tempo

Implementações de List

- Na prática usaremos ArrayList na maioria dos casos
- Permite especificar o tamanho inicial da lista (que é 10, por default)
- Deve-se especificar o tipo
- Ex:

```
List<String> l = new ArrayList<String>(20);
```

java.util.Set

- Noção matemática de conjunto
- Coleção de objetos onde
  - A ordem não importa
  - Não há duplicação
- Não contém nenhuma operação além das operações “herdadas” de Collection
  - add, remove, size, contains, isEmpty, etc...

Uso de Set

- Semelhante ao uso de lista
  - Porém, sem índices

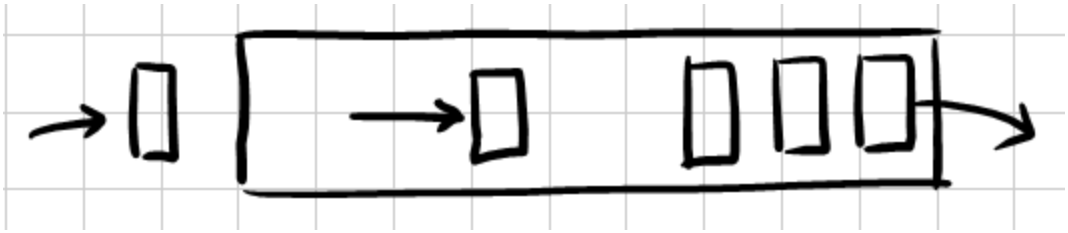
```
Set<String> conj = new HashSet<String>();
conj.add(“Alo”);
conj.add(“Pessoal”);
conj.add(“!”);
conj.add(“Pessoal”); // não vai inserir
for(String s: conj) { ... }
```

Implementações de Set

- HashSet
  - Baseada em tabela hash
  - Melhor desempenho
- TreeSet
  - Baseada em árvore (red-black)
- LinkedHashSet
  - Usa uma tabela hash e uma lista ligada
- TreeSet e LinkedHashSet classificam os elementos

java.util.Queue

- Estrutura de fila



- Ordenação é relevante
- Permite duplicação
- Primeiro que entra é o primeiro que sai
  - FIFO – First In First Out

java.util.Queue

- Além das operações “herdadas” de Collection, define operações específicas de fila

```
public interface Queue<E> extends Collection<E> {
  boolean offer(E e); // equivalente ao add.
                      // Inclui no final da fila
  E peek();           // Dá uma “olhadinha” no
                      // início da fila
  E poll();           // Remove (e retorna) o
                      // objeto do início da fila
}
```

Uso de Queue

- Parecido com lista

```
Queue<String> q = new LinkedList<String>();
q.offer(“Alo”);
q.offer(“Pessoal”);
q.offer(“!”);
System.out.println(q.peek()); // imprime “Alo”
System.out.println(q.poll()); // imprime “Alo”
System.out.println(q.poll()); // imprime “Pessoal”
System.out.println(q.poll()); // imprime “!”
```

java.util.Deque

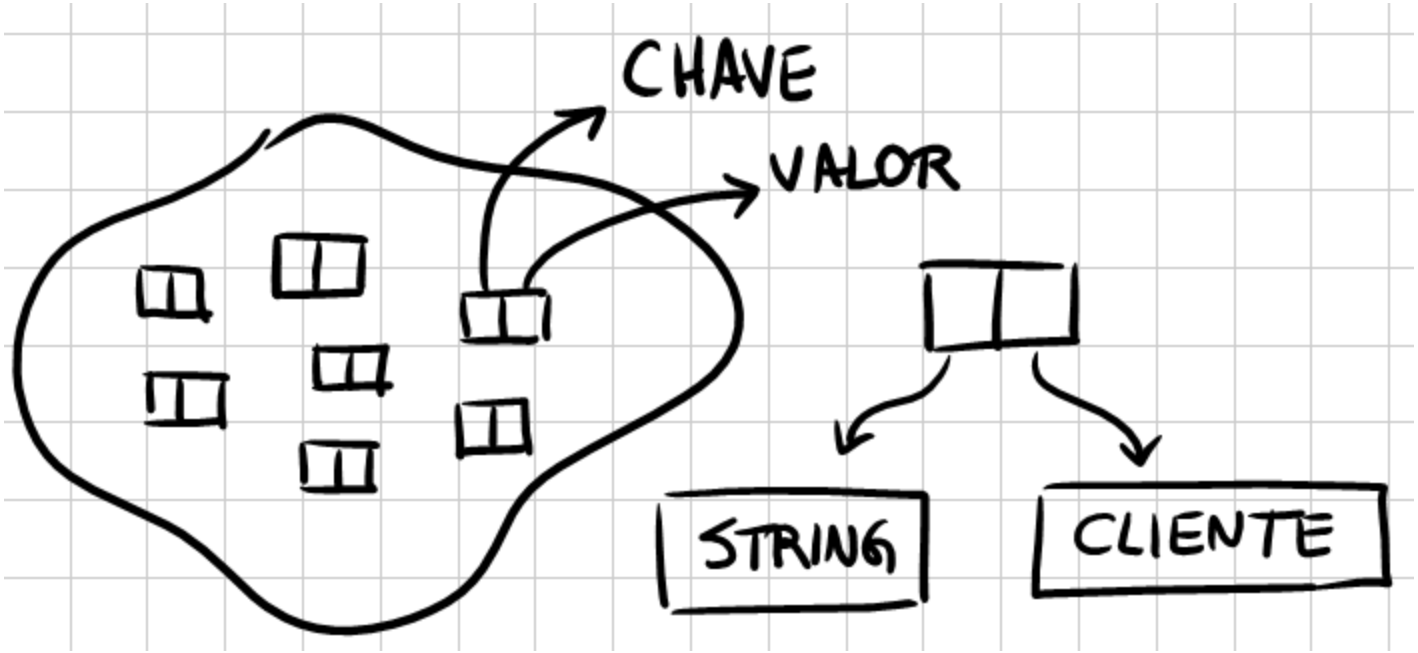
- Mesmo conceito de fila, porém é possível adicionar/remover elementos dos dois lados
  - Também conhecido com “fila de dois lados”
- As implementações de Queue normalmente também implementam Deque

Implementações de Queue

- LinkedList
  - Lista ligada
- PriorityQueue
  - Estrutura heap (min-heap)
  - A ordem é definida conforme algum critério de classificação

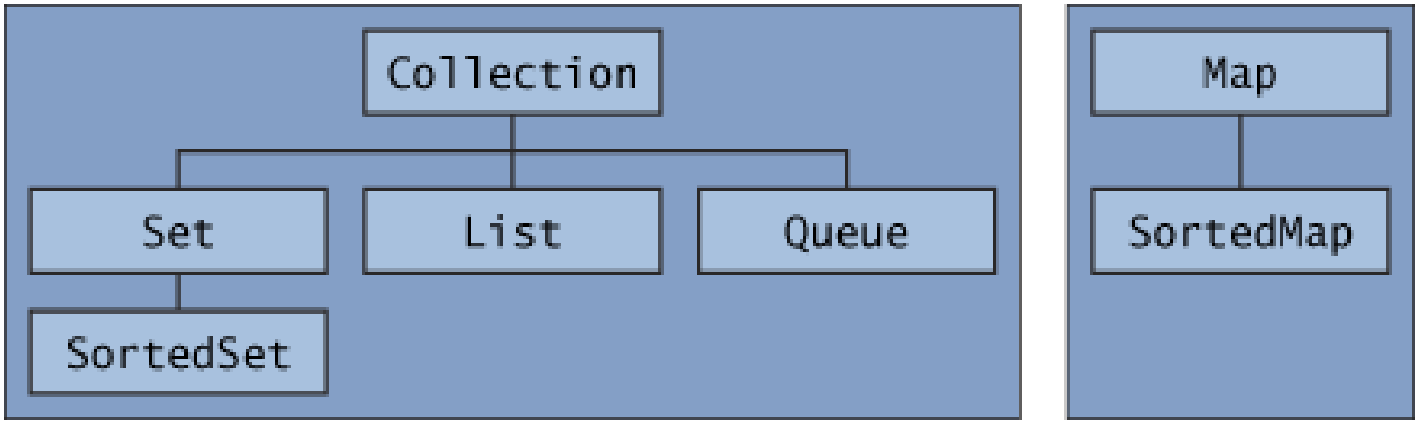
java.util.Map

- Estrutura tipo Dicionário
- Conjunto de pares (chave,valor)



java.util.Map

- Possui métodos para lidar com coleções de objetos
  - Mas não é uma subinterface de Collection
  - Cada objeto tem uma chave associada, normalmente usada para busca eficiente



java.util.Map

```
public interface Map<K,V> {
  V put(K key, V value);
  V get(Object key);
  V remove(Object key);
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  int size();
  boolean isEmpty();
  void putAll(Map<? extends K, ? extends V> m);
  void clear();
  // ... (continua)
```

java.util.Map

```
  // ... (continuação)
  public Set<K> keySet();
  public Collection<V> values();
  public Set<Map.Entry<K,V>> entrySet();

  public interface Entry {
      K getKey();
      V getValue();
      V setValue(V value);
  }
}
```

Uso de Map

- Sempre associando um valor a uma chave
  - Inserção e remoção
- Ambos devem ser do tipo Object

```
Map<Integer,String> meses = new HashMap<Integer,String>();
meses.put(1, “Janeiro”); meses.put(2, “Fevereiro”); ...
System.out.println(meses.get(2)); // imprime “Fevereiro”
System.out.remove(3); // remove o mês de Março
```

Uso de Map

- É possível percorrer um Map obtendo uma referência para o conjunto de chaves:

```
Map<Integer,String> meses = new HashMap<Integer,String>();
meses.put(1, “Janeiro”); meses.put(2, “Fevereiro”); ...
```

```
Set<Integer> numerosMeses = meses.keySet();
for(Integer i:numerosMeses) {
        System.out.println(“Mês "+i
}
```

A ordem não será necessariamente a mesma da inserção

Uso de Map

- É possível percorrer um Map obtendo uma referência para uma coleção de valores:

```
Map<Integer,String> meses = new HashMap<Integer,String>();
meses.put(1, "Janeiro"); meses.put(2, "Fevereiro"); ...

  Collection<String> nomesMeses = meses.values();
  for(String s:nomesMeses) {
       System.out.println("Mês "+s);
  }
```

A ordem não será necessariamente a mesma da inserção

Uso de Map

- É possível percorrer um Map obtendo uma referência para um conjunto de pares (Entry) chave/valor:

```
Map<Integer,String> meses = new HashMap<Integer,String>();
meses.put(1, "Janeiro"); meses.put(2, "Fevereiro"); ...

Set<Entry<Integer,String>> pMeses = meses.entrySet();
for(Entry<Integer,String> e:pMeses) {
  System.out.println("Num: "+e.getKey());
  System.out.println("Nome: "+e.getValue());
}
```

A ordem não será necessariamente a mesma da inserção

Implementações de Map

- HashMap
  - Utiliza uma tabela Hash
  - Busca rápida, mas sem ordenação durante a iteração
- TreeMap
  - Utiliza uma árvore
  - Permite ordenação das chaves durante iteração
- LinkedHashMap
  - Utiliza uma lista ligada
  - Bom desempenho, com ordenação das chaves com base na ordem de inserção

Utilitários

- Existem classes com operações genéricas sobre coleções
- java.util.Arrays
- java.util.Collections
- Métodos de ordenação, embaralhamento, busca, etc...

Arrays vs Listas com Generics

- Lembram do exemplo onde o compilador Java deixou colocarmos um gato em um vetor de cachorros?
- Vejam agora, na demonstração, o que acontece com tipos genéricos

Resumo

Resumo

- Vimos em detalhes como a plataforma Java oferece suporte ao paradigma orientado a objetos
- Existem outras linguagens que suportam OO, mas o Java é uma das mais “puristas”
  - Em alguns casos, o Java é mais rigoroso (ex: tratamento de exceções obrigatório)
  - Mas com isso, tem-se uma maior confiabilidade

Fim