



Vetores

Mário Meireles Teixeira
mario@deinf.ufma.br

1



Coleções de tamanho fixo

- Às vezes, o tamanho máximo de uma coleção pode ser pré-determinado. Vetores são um caso especial de **coleção**.
- Linguagens de programação frequentemente oferecem um tipo de coleção de tamanho fixo: um *array* (vetor).
- Arrays Java podem armazenar valores de objetos ou de tipo primitivo.
- Arrays utilizam uma sintaxe especial.

2

2

Declaração de Vetores

- Estrutura de dados que permite agrupar variáveis de um mesmo tipo
- Pode-se declarar vetores de qualquer tipo, primitivo ou de objeto

```
public class Qualquer {  
    int x, y;  
    int i[];           // como nas linguagens C e C++  
    Ponto p[];  
}
```

```
int[] i, j, k;  // forma transitiva  
Ponto[] p;
```

3

3

Criação de Vetores (1/3)

- Em Java um vetor é um objeto, mesmo quando for composto por tipos primitivos
- Quando um vetor é criado, ele possui “métodos” e campos de dados como qualquer outro objeto

4

4

Criação de Vetores (2/3)

- A criação de vetores se dá da mesma forma que a criação (instanciação) de objetos:

```
int[] num;  
Ponto[] p;  
num = new int[20];  
p = new Ponto[5];           // índice de 0 a 4
```

- Outra maneira de declarar seria:

```
Ponto[] p = new Ponto[5];   // Ponto[i] == null  
int[] num = new int[20];   // num[i] == 0
```

5

5

Criação de Vetores (3/3)

- Os objetos da classe **Ponto** devem ser instanciados separadamente:

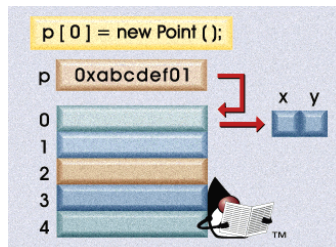
```
Ponto[] p = new Ponto[5];  
  
for (int i=0; i < p.length; i++) {  
    p[i] = new Ponto();  
}
```

6

6

Inicialização de Vetores

- Situação após a atribuição `p[0] = new Point();`



7

7

Inicialização Abreviada

- Exemplo: Strings

```
String[] cores = {"verde", "azul", "vermelho"};
```

- equivale a:

```
String[] cores = new String[3];
```

```
cores [0] = "verde";
```

```
cores [1] = "azul";
```

```
cores [2] = "vermelho";
```

8

8

Tamanho de um vetor

- Se **a** é um identificador de um vetor, **a.length** fornece o seu tamanho
- O método a seguir imprime um array de inteiros de tamanho arbitrário:

```
static void imprimir(int[] a) {  
    for (int i=0; i < a.length; i++)  
        System.out.println (a[i]);  
}
```

9

9

Percorrendo um vetor usando 'foreach'

- O Java possui outra sintaxe para percorrer arrays, mais intuitiva:

```
static void imprimir(int[] vet) {  
    for (int elem : vet)  
        System.out.println(elem);  
}
```

Não é mais necessário o campo **length** para percorrer o vetor

10

10

Exemplo: InitArray

// Deitel - Fig. 7.2: InitArray.java

```
public class InitArray
{
    public static void main( String args[] )
    {
        int array[];
        array = new int[ 10 ];
        System.out.printf( "%s%8s\n", "Index", "Value" );

        for ( int counter = 0; counter < array.length; counter++ )
            System.out.printf( "%5d%8d\n", counter, array[ counter ] );
    }
}
```

11

11

Exemplo: InitArray (2)

// Fig. 7.3: InitArray.java

// Inicializando os elementos do array

```
public class InitArray
{
    public static void main( String args[] )
    {
        int array[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

        System.out.printf( "%s%8s\n", "Index", "Value" );

        for ( int counter = 0; counter < array.length; counter++ )
            System.out.printf( "%5d%8d\n", counter, array[ counter ] );
    }
}
```

12

12

// Fig. 7.6: BarChart.java

```
public class BarChart
{
    public static void main( String args[] )
    {
        int array[] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };

        System.out.println( "Grade distribution:" );
        for ( int counter = 0; counter < array.length; counter++ )
        {
            // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
            if ( counter == 10 )
                System.out.printf( "%5d: ", 100 );
            else
                System.out.printf( "%02d-%02d: ", counter * 10,
                                    counter * 10 + 9 );

            for ( int stars = 0; stars < array[ counter ]; stars++ )
                System.out.print( "*" );

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

13

// Fig. 7.8: StudentPoll.java

```
public class StudentPoll
{
    public static void main( String args[] )
    {
        int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
                            10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
                            4, 8, 6, 8, 10 };

        int frequency[] = new int[ 11 ];
        for ( int answer = 0; answer < responses.length; answer++ )
            ++frequency[ responses[ answer ] ];

        System.out.printf( "%s%10s\n", "Rating", "Frequency" );

        for ( int rating = 1; rating < frequency.length; rating++ )
            System.out.printf( "%6d%10d\n", rating, frequency[ rating ] );
    }
}
```

14

14

Atribuição de vetores

```
public class TestaVetor {  
    public static void main(String [] args) {  
        int [] vetor1, vetor2;  
        int vetor3[] = { 1,2,3,4,5,6,7,8,9,10 };  
        vetor1 = vetor3;  
  
        for(int i = 0; i < vetor1.length; i++) {  
            System.out.println("Elem. " + i +  
                               " igual a " + vetor1[i]);  
        }  
    }  
}
```

Elem. 0 igual a 1
Elem. 1 igual a 2
Elem. 2 igual a 3
Elem. 3 igual a 4
Elem. 4 igual a 5
Elem. 5 igual a 6
Elem. 6 igual a 7
Elem. 7 igual a 8
Elem. 8 igual a 9
Elem. 9 igual a 10

15

15

Argumentos de linha de comando

```
public class Args {  
    public static void main(String [] args) {  
        for (int i=0; i < args.length; i++)  
            System.out.println("args[" + i + "] = " +  
                               args[i]);  
    }  
}
```

Para passar o parâmetro utilizamos o comando:

java Args parametro1 parametro2 parametroN

Saída:
args[0] = parametro1
args[1] = parametro2
args[2] = parametroN

16

16

Passando vetores como parâmetros

// Fig. 7.13: PassArray.java

```
public class PassArray
{
    public static void main( String args[] )
    {
        int array[] = { 1, 2, 3, 4, 5 };
        // output original array elements
        for ( int value : array ) System.out.printf( " %d", value );
        modifyArray( array ); // pass array reference

        // output modified array elements
        for ( int value : array ) System.out.printf( " %d", value );

        System.out.println("\nantes de modifyElement: " + array[ 3 ] );
        modifyElement( array[ 3 ] );
        System.out.println("depois de modifyElement: "+ array[ 3 ] );
    }
    // métodos omitidos (próximo slide)
}
```

Parâmetros em Java são sempre passados *por valor*

17

PassArray: métodos auxiliares

```
public class PassArray
{
    // método main() omitido

    // multiply each array element by 2
    public static void modifyArray( int array2[] )
    {
        for ( int counter = 0; counter < array2.length; counter++ )
            array2[ counter ] *= 2;
    }

    // multiply argument by 2
    public static void modifyElement( int element )
    {
        element *= 2;
        System.out.println("dentro de modifyElement: "+ element );
    }
}
```

18

18

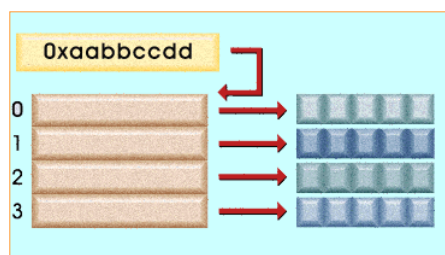
Retornando vetores como parâmetro

```
public class Loteria {  
    public static void main(String[] args) {  
        int[] aposta = getDezenas();  
        for(int i=0; i < aposta.length; i++)  
            System.out.print(aposta[i] + " ");  
    }  
    public static int[] getDezenas() {  
        int[] dezenas = new int[6];  
        for (int i = 0; i < dezenas.length; i++) {  
            dezenas[i] = (int)Math.ceil((Math.random()*50));  
        }  
        return dezenas;  
    }  
}
```

19

Vetores Multidimensionais

- Java não suporta vetores multidimensionais diretamente, mas como um vetor pode ser declarado como tendo qualquer tipo, pode-se criar vetores de vetores (matrizes)



20

20

Matrizes

- Exemplo

```
- int mat [][] = new int [4][] ;  
- mat[0] = new int [5] ;  
- mat[1] = new int [5] ;  
- mat[2] = new int [5] ;  
- mat[3] = new int [5] ;
```

```
// m[0][0] == 1; m[0][1] == 2  
- int m[][] = { {1,2}, {0,-3} };
```

- A sintaxe a seguir **não** é válida

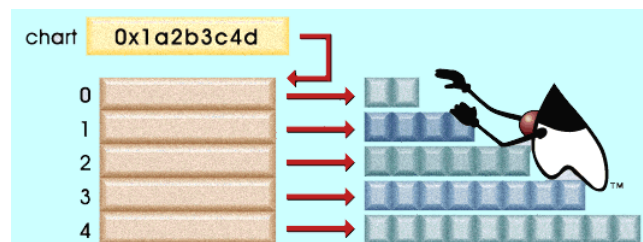
```
- int mat [][]= new int [][][4];
```

21

21

Matrizes

- Cada elemento da submatriz deve ser construído separadamente. Portanto, é possível criar matrizes não retangulares



22

22

“Matrizes”

- Exemplo

```
- int mat [][] = new int [5][] ;  
- mat[0] = new int [2] ;  
- mat[1] = new int [4] ;  
- mat[2] = new int [6] ;  
- mat[3] = new int [8] ;  
- mat[3] = new int [10] ;
```

- Java fornece um atalho para criar matrizes bidimensionais retangulares

```
- int mat [][] = new int [4][5] ;
```

23

23

imprimeArray()

```
public static void imprimeArray( int array[][] )  
{  
    for ( int linha = 0; linha < array.length; linha++ )  
    {  
        for ( int col = 0; col < array[ linha ].length; col++ )  
            System.out.print( array[ linha ][ col ] + " " );  
  
        System.out.println();  
    }  
}
```

24

24

imprimeArray(): 'foreach'

```
public static void imprimeArray( int mat[][] )
{
    for (int[] linha : mat) {
        for (int elem : linha)
            System.out.print(elem + " ");
        System.out.println();
    }
}
```

25

25

Exercício

- Implemente uma classe que receba um conjunto de cinco notas de alunos numa disciplina (P1, P2, P3, P4, FINAL) e escreva um programa para testá-la. Armazene os nomes dos alunos e suas notas num array bidimensional
- Seu programa de teste deve imprimir a relação de alunos e suas notas, a média de cada aluno e sua situação. Deve, ainda, calcular a média geral da turma e mostrar um gráfico de barras com a distribuição das notas

```
media = (P1 + P2 + P3) / 3;
Se (media >= 7) aprovado;
senão Substitua menor nota por P4 ;

Se (media >= 7) aprovado;
senão media2 = (media + FINAL) / 2;

Se (media2 >= 6) aprovado;
senão reprovado;
```

26

26

java.util.Arrays

- Classe utilitária com diversos métodos estáticos para manipulação de vetores
- Principais métodos
 - void Arrays.sort(vetor)
 - Usa Quicksort para tipos primitivos; Mergesort para objetos
 - boolean Arrays.equals(vetor1, vetor2)
 - int Arrays.binarySearch(vetor, chave)
 - void Arrays.fill(vetor, valor)

27

27

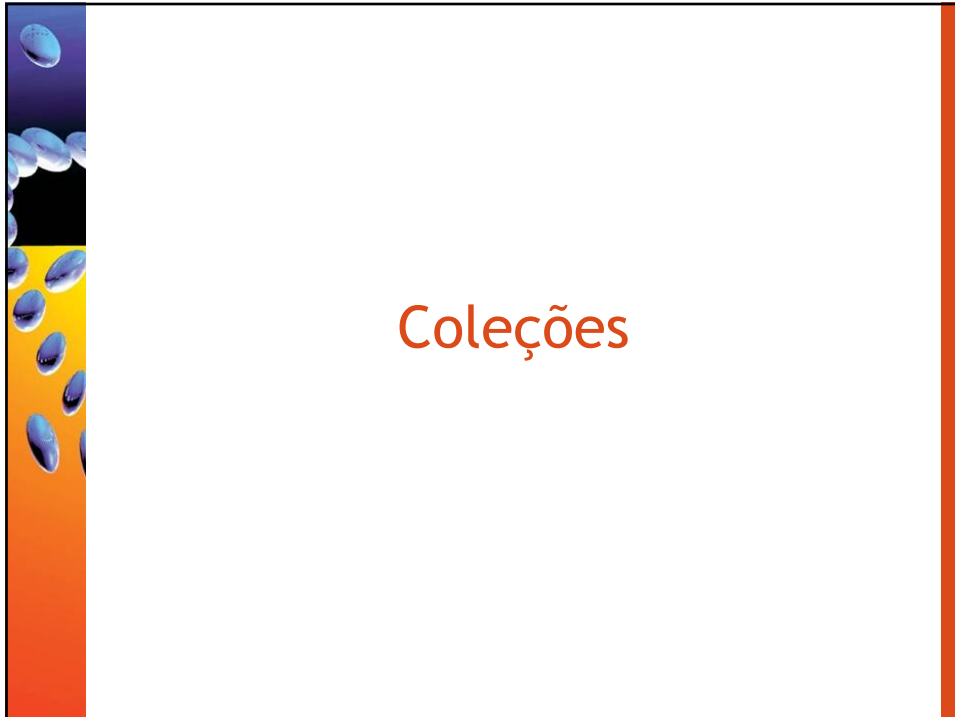
```
public class Cadastro {
    public static void main(String[] args) {
        String[] cadastro = {"maria", "joao", "pedro"};

        System.out.println("Vetor não ordenado");
        for (int i = 0; i < cadastro.length; i++) {
            System.out.println(cadastro[i]);
        }
        // ordena o cadastro
        Arrays.sort(cadastro);

        // imprime o cadastro ordenado
        System.out.println("Vetor Ordenado");
        for (int i = 0; i < cadastro.length; i++) {
            System.out.println(cadastro[i]);
        }
    }
}
```

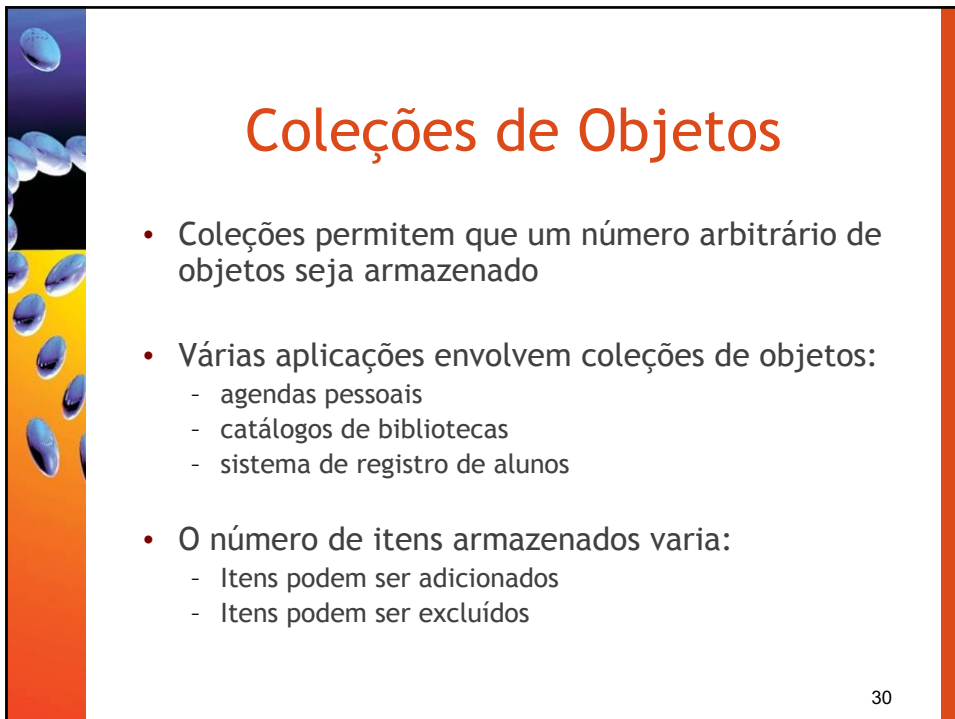
28

28



Coleções

29



Coleções de Objetos

- Coleções permitem que um número arbitrário de objetos seja armazenado
- Várias aplicações envolvem coleções de objetos:
 - agendas pessoais
 - catálogos de bibliotecas
 - sistema de registro de alunos
- O número de itens armazenados varia:
 - Itens podem ser adicionados
 - Itens podem ser excluídos

30

30

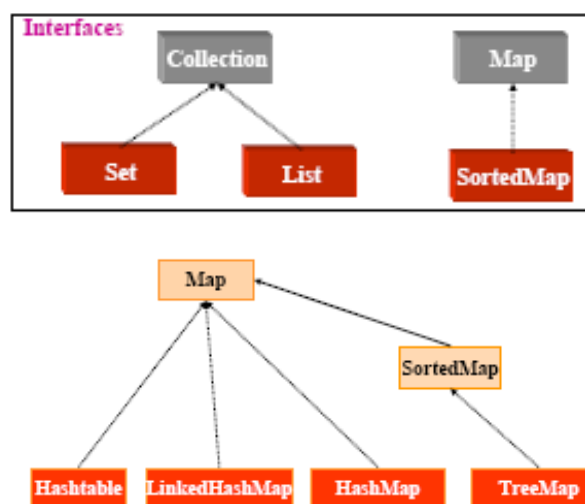
Tipos de Coleções

- Existem dois tipos de coleções em Java: as que implementam a Interface **Collection** e os mapas, que implementam a interface **Map**
- A interface Collection possui duas subinterfaces:
 - **List** - representa uma lista de itens
 - **Set** - representa um conjunto (os itens não podem se repetir)
- A interface **Map** implementa uma tabela Hash, guarda compostos de *chaves + valor*
 - **SortedMap** - mapa ordenado

31

31

Tipos de Coleções



32

32



Listas

33

33



Listas

- Uma lista (**List**) é uma coleção de elementos dispostos em ordem linear, onde cada elemento tem um antecessor (exceto o primeiro) e um sucessor (exceto o último)
 - Normalmente, implementada como "Array" (**Vector**, **ArrayList**) ou "Lista Encadeada" (**LinkedList**)
 - Todas as três implementações são ordenadas (pode-se visitar todos os elementos em uma ordem não aleatória)
 - Uma lista pode ser mantida classificada ou não

34

34



Bibliotecas de classes

- Coleções de classes úteis
- Não temos de escrever tudo a partir do zero
- O Java chama suas bibliotecas de *pacotes*
- Agrupar objetos é um requisito recorrente
 - O pacote `java.util` contém as classes para fazer isso, p.ex., a classe `ArrayList`

35

35



ArrayList: Bloco de notas

- Notas podem ser armazenadas
- Notas individuais podem ser visualizadas
- Não há um limite para o número de notas
- Ela informará quantas observações estão armazenadas

36

36

```
import java.util.ArrayList;

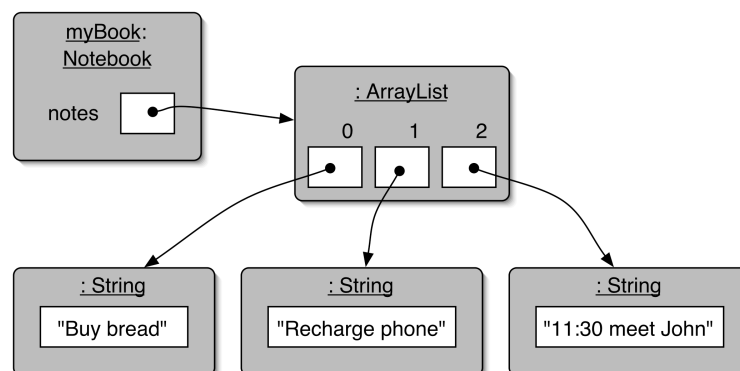
public class Notebook
{
    // Campo para um número arbitrário de notas
    private ArrayList notes;

    // Realiza qualquer inicialização
    // necessária para o notebook
    public Notebook()
    {
        notes = new ArrayList();
    }
    ...
}
```

37

37

Numeração de índice



38

38

Recursos da classe ArrayList

- Ela aumenta a capacidade interna conforme necessário
- Mantém uma contagem privada (método de acesso `size()`)
- Mantém os objetos em ordem de inserção
- Os principais métodos `ArrayList` são `add`, `get`, `remove` e `size`
- Os detalhes sobre como tudo isso é feito são ocultados do programador (encapsulamento)

39

39

Utilizando a coleção

```
public class Notebook
{
    private ArrayList notes;
    ...

    public void storeNote(String
note)
    {
        notes.add(note);

    }

    public int numberOfNotes()
    {
        return notes.size();

    }

    ...
}
```

Adicionando uma
nova nota

Retornando o número
de notas (*delegação*)

40

40

Recuperando um item da coleção

```
public void showNote(int noteNumber)
{
    if(noteNumber < 0) {
        // Este não é um número de nota válido.
    }
    else if(noteNumber < numberOfNotes()) {
        System.out.println(notes.get(noteNumber));
    }
    else {
        // Este não é um número de nota válido.
    }
}
```

41

41

Removendo um item da coleção

```
public void removeNote(int noteNumber)
{
    if(noteNumber < 0) {
        // Não é um índice de nota válido
    }
    else if(noteNumber < numberOfNotes()) {
        notes.remove(noteNumber);
    }
    else {
        // Não é um índice de nota válido
    }
}
```

Após a remoção, os itens à direita do que foi removido são deslocados uma posição para a esquerda, alterando seus índices.

42

42

Percorrendo uma coleção

```
/**
 * Lista todas as notas no bloco de notas.
 */
public void listNotes()
{
    int index = 0;
    while(index < notes.size()) {
        System.out.println(notes.get(index));
        index++;
    }
}
```

43

43

Objetos Iterator

java.util.Iterator

Retorna um objeto
Iterator

```
Iterator it = myCollection.iterator();
while(it.hasNext()) {
    chame it.next() para obter o próximo objeto
    faça algo com esse objeto
}
```

```
public void listNotes()
{
    Iterator it = notes.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
}
```


44

44



Exemplo: classe Auction

45



```
public class Auction // POO Java – Barnes, Cap. 4
{
    private ArrayList lots;
    private int nextLotNumber;
    public Auction() {
        lots = new ArrayList();
        nextLotNumber = 1;
    }

    public void enterLot(String description) {
        lots.add(new Lot(nextLotNumber, description));
        nextLotNumber++;
    }

    public void showLots() {
        Iterator it = lots.iterator();
        while(it.hasNext()) {
            Lot lot = (Lot) it.next();
            System.out.println(lot.getNumber() + ": " + lot.getDescription());
            Bid highestBid = lot.getHighestBid();
            if(highestBid != null) {
                System.out.println("    Bid: " + highestBid.getValue());
            } else {
                System.out.println("    (No bid)");
            }
        }
    }
}
```

46

```

public class Auction
{
    ...

    public Lot getLot(int number)
    {
        if((number >= 1) && (number < nextLotNumber)) {
            Lot selectedLot = (Lot) lots.get(number-1);
            if(selectedLot.getNumber() != number) {
                System.out.println("Internal error: " +
                    "Wrong lot returned. " +
                    "Number: " + number);
            }
            return selectedLot;
        }
        else {
            System.out.println("Lot number: " + number +
                " does not exist.");
            return null;
        }
    }
    ...
}

```

47

47

```

public class Lot
{
    private final int number;
    private String description;
    private Bid highestBid;

    public Lot(int number, String description) {
        this.number = number;
        this.description = description;
    }

    public void bidFor(Person bidder, long value) {
        if((highestBid == null) ||
            (highestBid.getValue() < value)) {
            setHighestBid(new Bid(bidder, value));
        }
        else {
            System.out.println("Lot number: " + getNumber() + " (" +
                getDescription()
                + ") " + " already has a bid of: " + highestBid.getValue());
        }
    }
    ...
}

```

48

48



Mapas

49

49



Mapas

- Mapas (**Maps**) são um tipo de coleção que armazenam pares de objetos, do tipo **chave/valor**
- Utiliza-se o objeto chave para pesquisar o objeto valor
- Um mapa é organizado de forma a tornar as consultas rápidas em um sentido (**chave -> valor**)
- Tanto a chave quanto o valor são objetos
- Chaves são unívocas (Set) / Valores podem ser duplicados (Collection)

50

50



Mapas

- Diferentes de **ArrayList**, que armazena apenas um objeto em cada entrada e recupera os objetos segundo seu índice (posição) na lista
- Exemplo de mapas: lista telefônica, cadastro de correntistas de um banco, cadastro de funcionários

51

51



Interface Map - principais métodos

- **Object put(Object key, Object value)**
 - associa uma chave a um valor no mapa. Se a chave já existir, o novo valor substitui o anterior e é retornado o valor antigo
 - corresponde ao método **add(Object obj)** de Collection (ArrayList é um subtipo de Collection)
- **Object get(Object key)**
 - recupera um objeto associado a uma chave; retorna null caso a chave não exista
 - corresponde ao método **Object get(int pos)** de Collection

52

52

HashMap

- Fornece um conjunto **Map** não-ordenado
 - Quando se precisar de um mapa e não se estiver preocupado com a ordem dos elementos, então **HashMap** é a melhor opção
 - **TreeMap** mantém o mapa ordenado pela chave, mas impõe uma certa sobrecarga às operações de inserção e remoção

53

53

Utilizando mapas

- Um map com strings como chaves e valores:

:HashMap	
"Charles Nguyen"	"(531) 9392 487"
"Lisa Jones"	"(402) 4536 4674"
"William H. Smith"	"(998) 5488 0123"

54

54

Utilizando mapas

```
public class ListaTelefonica {  
  
    public static void main(String[] args) {  
        HashMap agenda = new HashMap();  
  
        agenda.put("Luís Felipe", "(75) 3392 4587");  
        agenda.put("Luciana Serpa", "(98) 5536 4674");  
        agenda.put("Carlos Augusto", "(21) 2488 0123");  
  
        // Imprime os nomes/telefones  
        System.out.println(agenda);  
        // Obtém entrada do mapa  
        String number = (String) agenda.get("Luciana Serpa");  
        System.out.println("Numero encontrado: " + number);  
        // Altera valor associado à chave  
        agenda.put("Luís Felipe", "(98) 3222 2323");  
        System.out.println(agenda);  
        // Remove entrada do mapa  
        agenda.remove("Carlos Augusto");  
        System.out.println(agenda);  
    }  
}
```

55

55

Collections utilizando Generics

- **Collection<E>**
 - O parâmetro E parametriza o tipo de elemento armazenado na coleção

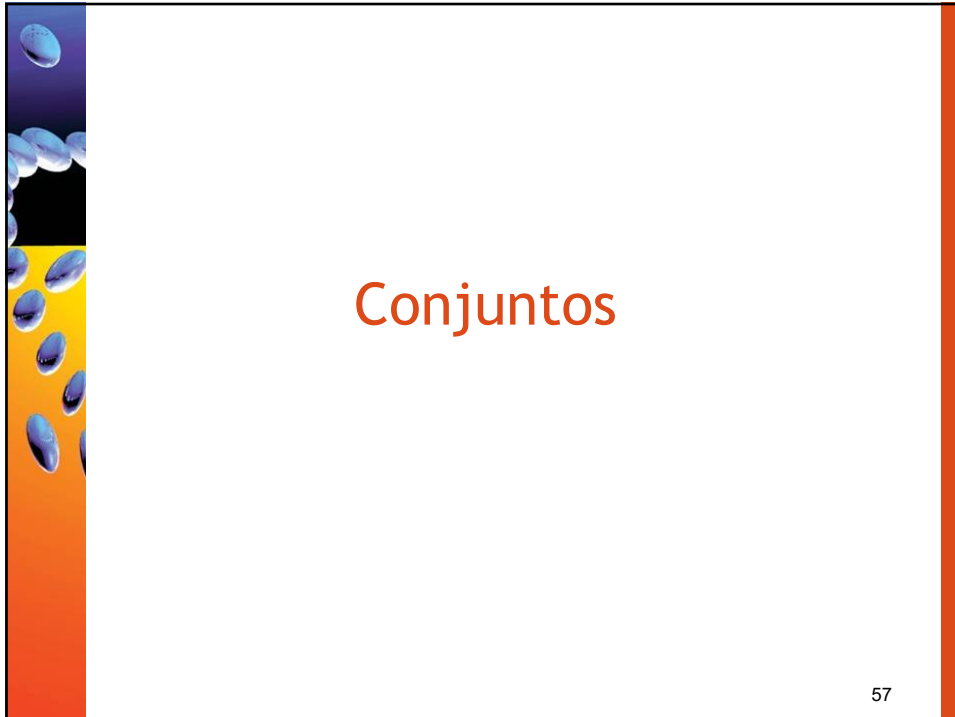
```
Collection<Conta> agencia = new ArrayList<Conta>();  
agencia.add(new Conta("1234-5"));  
agencia.add(new Conta("9876-6"));  
agencia.add(new Cliente("João"));  
Iterator<Conta> it = agencia.iterator();  
while(it.hasNext()) {  
    Conta c = it.next();  
    c.creditar(500);  
}
```

Causa erro de compilação

Não é mais necessário fazer o cast

56

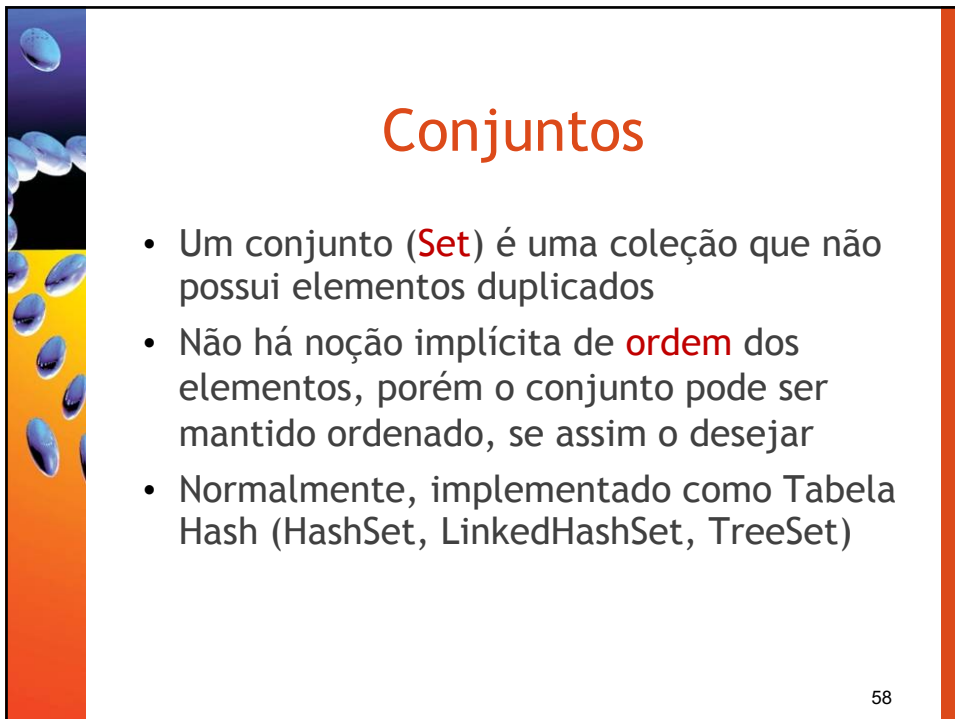
56



Conjuntos

57

57



Conjuntos

- Um conjunto (**Set**) é uma coleção que não possui elementos duplicados
- Não há noção implícita de **ordem** dos elementos, porém o conjunto pode ser mantido ordenado, se assim o desejar
- Normalmente, implementado como Tabela Hash (HashSet, LinkedHashSet, TreeSet)

58

58

Utilizando conjuntos

```
import java.util.HashSet;
import java.util.Iterator;

HashSet mySet = new HashSet();

mySet.add("um");
mySet.add("dois");
mySet.add("três");

Iterator it = mySet.iterator();
while(it.hasNext()) {
    System.out.println(it.next());}
```

59

59

Resumo

- O uso de coleções em Java é bem semelhante, mesmo para tipos de coleções diferentes. As diferenças estão no comportamento de cada coleção
- Lista: elementos inseridos em ordem; acesso por índice; duplicatas permitidas
- Mapa: armazena pares de objetos (chave, valor); possibilita pesquisas rápidas utilizando a chave
- Set: não mantém qualquer ordem específica dos elementos; não admite duplicatas

60

60

Strings

- Visite o link:

http://www.tutorialspoint.com/java/java_strings.htm

61

61

Tokenizando strings

```
public HashSet getInput()
{
    System.out.print("> ");
    String inputLine = readInputLine().trim().toLowerCase();

    StringTokenizer tokenizer = new StringTokenizer(inputLine);
    HashSet words = new HashSet();

    while(tokenizer.hasMoreTokens()) {
        words.add(tokenizer.nextToken());
    }
    return words;
}
```

62

62