1

Coleções em Java – *Collections*
Genéricos em Java - *Generics*

2

Interface Collection

- O trecho seguinte mostra a interface Collection.

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

3

Interface Collection

- **Todas as Classes que a implementam:**
  - AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector
  - Muitas delas são classes abstratas – com implementações *default* – "stubs"
  - Se a classe concreta não implementar algum método dispara UnsupportedOperationException

4

Atravesando Coleções

- Duas formas de atravessar coleções:
  - com a construção for-each e
  - usando Iterators.
- **Construção for-each:** Permite atravessar de forma **concisa** uma coleção ou array usando uma repetição **for**.
- Ex.: código que usa for-each para escrever cada elemento de uma coleção em uma linha separada:

```
for (Object o : collection)
        System.out.println(o);
```

5

Atravesando Coleções

- **Iterator:** objeto que possibilita atravessar uma coleção e, se desejado, remover elementos da mesma de forma seletiva.
- **Obtém-**se um Iterator para uma collection chamando seu método ***iterator ().***
- Método ***hasNext*** → retorna true se a iteração tiver mais elementos
- Método ***next*** → retorna o próximo elemento na iteração
- Método ***remove*** → remove o último elemento que foi retornado por ***next*** da Coleção subjacente.
- **Interface Iterator**:

```
public interface Iterator<E> {
   boolean hasNext();
    E next();
    void remove(); }
```

6

Atravesando Coleções

- Use Iterator em vez de for-each quando precisar:
  - Remover o elemento corrente
    - for-each esconde o iterator, então vc não pode chamar remove.
  - Fazer uma Iteração sobre múltiplas coleções em paralelo.
- Ex: método que usa um Iterator para filtrar uma Collection arbitrária - atravessa a coleção removendo elementos específicos

```
static void filter(Collection<?> c) {
for (Iterator<?> it = c.iterator(); it.hasNext(); )
 if (!cond(it.next()))
       it.remove(); }
```

- Trecho de código polimórfico
  - Funciona para *qualquer* Collection independente de sua implementação

Interface Collection X Classe Collections

7

- **Interface Collection**: contém operações coletivas (*bulk*)
  - Adicionar, limpar, comparar e reter objetos
- **Classe Collections**: Fornece métodos estáticos que manipulam coleções
  - operam sobre ou retornam coleções
  - contém algoritmos polimórficos que operam sobre coleções
  - seus métodos disparam uma NullPointerException se as coleções fornecidas para eles forem nulas

Algoritmos

8

- A Framework Collections fornece um conjunto de **algoritmos**
  - Implementados como métodos `static` da classe **Collections**
    - Algoritmos sobre `Listas`
      - `sort`
      - `binarySearch`
      - `reverse`
      - `shuffle`
      - `fill`
      - `copy`
    - Algoritmos sobre `Collection`
      - `min`
      - `max`

Algoritmo `sort`

9

- `sort`
  - Ordena elementos de `List`as
    - Ordem determinada pela ordem *natural* do tipo dos elementos
    - Relativamente rápido

```
1  // Sort1.java
2  // usando algoritmo sort.
3  import java.util.*;
4
5  public class Sort1 {
6     private static final String suits[] =
7        { "Hearts", "Diamonds", "Clubs", "Spades" };
8
9     // apresenta os elmentos do array
10    public void printElements()
11    {
12       // cria ArrayList
13       List list = new ArrayList( Arrays.asList( suits ) );
14
15       // escreve a lista
16       System.out.println( "Unsorted array elements:\n" + list );
17
18       Collections.sort( list ); // ordena Arra
19
20       // escreve a lista
21       System.out.println( "Sorted array elemen
22    }
23
```

Usa o método sor `Collections` para `ArrayList`

classe java.util.Arrays: contém métodos para manipular arrays e um factory que permite arrays serem vistos como listas

```
24    public static void main( String args[] )
25    {
26       new Sort1().printElements();
27    }
28
29 } // fim da classe Sort1
```

```
Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted array elements:
[Clubs, Diamonds, Hearts, Spades]
```

`Sort1.java`

```
1  // Sort2.java
2  // Usando um objeto Comparator com o algoritmo sort.
3  import java.util.*;
4
5  public class Sort2 {
6     private static final String suits[] =
7        { "Hearts", "Diamonds", "Clubs", "Spades" };
8
9     // escreve elements da Lista
10    public void printElements()
11    {
12       List list = Arrays.asList( suits ); // cria List
13
14       // escreve elements da Lista
15       System.out.println( "Unsorted array elements:\n" + list );
16
17       // ordena descendente usando um comparador
18       Collections.sort( list, Collections.reverseOrder() );
19
20       // escreve elements da Lista
21       System.out.println( "Sorted list elements:\n" + list );
22    }
23
```

Método `reverseOrder` da classe `Collections` retorna um objeto `Comparator` que representa a ordem reversa da collection

Método `sort` da classe `Collections` pode usar um `Comparator` para ordenar uma `List`

```
24    public static void main( String args[] )
25    {
26       new Sort2().printElements();
27    }
28
29 } // end class Sort2
```

Sort2.java

```
Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted list elements:
[Spades, Hearts, Diamonds, Clubs]
```


Conjuntos - Sets

- `HashSet`
  - Armazena elementos em um hashtable
- `TreeSet`
  - Armazena elementos em uma árvore

Exemplo: `TreeSet`

- Método:

  public SortedSet<E> **headSet**(E toElement)
  - Retorna uma visão da porção desse conjunto cujos elementos são estritamente menores que **toElement**.
  - Mudanças no conjunto retornado são refletidas no conjunto original e vice-versa;
  - O conjunto retornado irá disparar uma IllegalArgumentException se o usuário tentar inserir um elemento maior do que ou igual a toElement.

```
1  //  SortedSetTest.java
2  // Usa TreeSet e SortedSet.
3  import java.util.*;
4
5  public class SortedSetTest {
6     private static final String names[] = { "yellow", "green",
7        "black", "tan", "grey", "white", "orange", "red", "green" };
8
9     // cria um conjunto ordenado com TreeSet e o manipula
10    public SortedSetTest()
11    {
12       SortedSet tree = new TreeSet( Arrays.asList( names ) );
13
14       System.out.println( "set: " );
15       printSet( tree );
16
17       // obtém headSet baseado em "orange"
18       System.out.print( "\nheadSet (\"orange\"):  " );
19       printSet( tree.headSet( "orange" ) );
20
21       // obtém tailSet baseado em "orange"
22       System.out.print( "tailSet (\"orange\"):  " );
23       printSet( tree.tailSet( "orange" ) );
24
25       // obtém primeiro e último elementos
26       System.out.println( "first: " + tree.first() );
27       System.out.println( "last : " + tree.last() );
28    }
29
```

Cria `TreeSet` a partir do array `names`

Usa o método `headSet` de TreeSet para obter um sub-conjunto **menor que** "orange"

Usa o método `tailSet` de TreeSet para obter um sub-conjunto de elementos **maiores ou iguais a** "orange"

Métodos `first` e `last` obtêm os menor e maior elementos do `TreeSet`, respectivamente

```
30    // apresenta o conjunto
31    private void printSet( SortedSet set )
32    {
33       Iterator iterator = set.iterator();
34
35       while ( iterator.hasNext() )
36          System.out.print( iterator.next() + " " );
37
38       System.out.println();
39    }
40
41    public static void main( String args[] )
42    {
43       new SortedSetTest();
44    }
45
46 } // end class SortedSetTest
```

Usa `Iterator` para atravessar `HashSet` e escreve os valores

```
set:
black green grey orange red tan white yellow

headSet ("orange"):  black green grey
tailSet ("orange"):  orange red tan white yellow
first: black
last : yellow
```


Mapas

- `Map`
  - Associa chaves a valores
  - Não pode conter chaves duplicadas
    - mapeamento *um-para-um*

```
1  //  WordTypeCount.java
2  // Conta o numero de ocorrências de cada palavra em uma string
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.*;
6  import javax.swing.*;
7
8  public class WordTypeCount extends JFrame {
9     private JTextArea inputField;
10    private JLabel prompt;
11    private JTextArea display;
12    private JButton goButton;
13
14    private Map map;
15
16    public WordTypeCount()
17    {
18       super( "Word Type Count" );
19       inputField = new JTextArea( 3, 20 );
20
21       map = new HashMap();
22
23       goButton = new JButton( "Go" );
24       goButton.addActionListener(
25
```

MapTest.java

```
26          new ActionListener() { // inner class
27
28             public void actionPerformed( ActionEvent event )
29             {
30                createMap();
31                display.setText( createOutput() );
32             }
33
34          } // fim da inner class
35
36       ); // fim da chamada a addActionListener
37
38       prompt = new JLabel( "Enter a string:" );
39       display = new JTextArea( 15, 20 );
40       display.setEditable( false );
41
42       JScrollPane displayScrollPane = new JScrollPane( display );
43
44       // adiciona componentes a GUI
45       Container container = getContentPane();
46       container.setLayout( new FlowLayout() );
47       container.add( prompt );
48       container.add( inputField );
49       container.add( goButton );
50       container.add( displayScrollPane );
51
```

MapTest.java

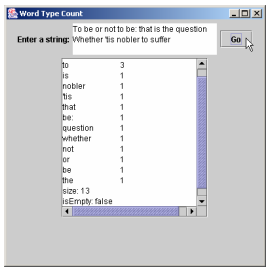
```
52          setSize( 400, 400 );
53          show();
54
55       } // fim do construtor
56
57       // cria map a partir da entrada do usuário
58       private void createMap()
59       {
60          String input = inputField.getText();
61          StringTokenizer tokenizer = new StringTokenizer( input );
62
63           while ( tokenizer.hasMoreTokens() ) {
64             String word = tokenizer.nextToken().toLowerCase();//obtém palavra
65
66             // se o mapa contém a palavra
67             if ( map.containsKey( word ) ) {
68
69                Integer count = (Integer) map.get( word ); // obtém valor
70
71                // incrementa valor
72                map.put( word, new Integer( count.intValue() + 1 ) );
73             }
74             else //caso contrário adiciona palavra com valor de 1 no mapa
75                map.put( word, new Integer( 1 ) );
76
77           } // fim do while
78
79       } // fim do createMap
```

```
80
81    // cria string contendo map values
82    private String createOutput() {
83       StringBuffer output = new StringBuffer( "" );
84       Iterator keys = map.keySet().iterator();
85
86       // iteração pelas chaves
87       while ( keys.hasNext() ) {
88          Object currentKey = keys.next();
89
90          // apresenta os pares chave-valor
91          output.append( currentKey + "\t" +
92                         map.get( currentKey ) + "\n" );
93       }
94
95       output.append( "size: " + map.size() + "\n" );
96       output.append( "isEmpty: " + map.isEmpty() + "\n" );
97
98       return output.toString();
99
100   } // fim de createOutput
101
```

Retorna uma visão na forma de um Set das chaves contidas no map

```
102   public static void main( String args[] )
103   {
104      WordTypeCount application = new WordTypeCount();
105      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
106   }
107
108 } // fim da classe WordTypeCount
```

MapTest.java



Genéricos em Java - *Generics*

Introdução

- JDK 1.5 → várias novas extensões a Linguagem Java
  - Uma dessas é a introdução de **genéricos - *generics***.
- Construções similares em outras linguagens
  - Em especial templates C++

- **Generics** → permite *abstrair-se* sobre os **tipos** das variáveis
  - Exemplos mais comuns → tipos *container*, como a hierarquia Collection
- Uso típico:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```


Introdução

- **Cast** na linha 3 um tanto incômodo
  - Tipicamente, o **programador sabe** que tipo de dado foi colocado em uma lista específica
  - Contudo, o **cast é essencial** – o compilador só garante que um **Object** será retornado pelo iterator
  - Para garantir que a atribuição para uma variável do tipo Integer é *type safe*, o cast é requerido
  - Cast "bagunça" o código
    - e também introduz a possibilidade de um erro em tempo de execução
      - já que o programador pode se enganar (e o compilador não terá como verificar)
- **E se** os programadores pudessem realmente expressar sua intenção e marcar uma lista como sendo restrita a conter um **tipo de dado específico** ?

IDÉIA CENTRAL POR TRÁS DOS GENÉRICOS !

Introdução

- Versão do fragmento anterior usando generics:

```
List<Integer> myIntList = new LinkedList<Integer> (); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

- Note: a **declaração de tipo** para a variável myIntList
  - especifica que essa **NÃo** é uma lista (List) arbitrária,
  - mas sim uma **List de Integer**, denotada por List<Integer>
- Dizemos que List é uma **interface genérica** que obtém um *parâmetro tipo* - nesse caso, Integer.
- Também especificamos um parâmetro tipo quando formos criar o objeto lista

O cast da linha 3 não existe mais !

Introdução

- **Argumento**: só se conseguiu mover a bagunça para outro local do código !
  - Em vez de um cast para Integer na linha 3, temos Integer como um *parâmetro tipo* na linha 1.
- Entretanto, há uma grande diferença !
  - O **compilador agora pode verificar** a corretude do tipo em **tempo de compilação**
    - Quando falamos que myIntList é declarada com tipo List<Integer>, isso diz algo sobre a variável myIntList, **que é verdade** sempre onde quer que ela seja usada, e o compilador irá garantir isso
  - Em contraste: **cast** nos diz algo que o **programador pensa** que é verdade em um único ponto do código
- Efeito bruto (especialmente em programas grandes):
  - aumentar a **legibilidade e robustez**

Definindo Genéricos Simples

- Fragmento das definições das interfaces List e Iterator no pacote java.util:

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator(); }

public interface Iterator<E> {
  E next();
  boolean hasNext();}
```

- **Trecho entre colchetes**: corresponde as declarações dos **parâmetros formais *tipo*** das interfaces List e Iterator.
- **Na invocação**, todas as ocorrências do parâmetro formal tipo (**E** nesse caso) são substituídas pelo argumento real tipo (no caso do exemplo, **Integer**).

Definindo Genéricos Simples

- Pode-se imaginar que List<Integer> denota uma versão de List onde E foi uniformemente substituído por Integer:

```
public interface IntegerList {
  void add(Integer x)
  Iterator<Integer> iterator();
}
```

Intuição útil porém enganosa !

- **Útil**: porque o tipo parametrizado List<Integer> de fato possui métodos que parecem com essa expansão
- **Enganosa**: porque a declaração de um genérico nunca é de fato expandida dessa forma
  - Não há múltiplas cópias do código: nem na fonte, nem no binário, no disco ou em memória
  - Uma declaração de tipo genérico é compilada **apenas uma vez**, e torna-se um **único arquivo** .class
    - da mesma forma que uma classe ou uma interface comuns

31

Definindo Genéricos Simples

- Parâmetros tipos são análogos aos parâmetros ordinários usados em métodos ou construtores
  - Da mesma forma que um método tem valores de parâmetros formais, que descrevem os tipos de valores que eles operam, uma declaração generic tem parâmetros tipo formais
  - Quando um método é invocado, argumentos reais substituem os parâmetros formais, e o corpo do método é avaliado

32

Type Erasure

- Genéricos são implementados por *type erasure*: a informação do tipo do genérico está presente apenas em tempo de compilação
  - após isso é **apagada** pelo compilador
- Quando um tipo genérico é instanciado, o compilador o transforma aplicando a técnica de *type erasure*
  - O compiladore remove todas as informações relacionadas ao parâmetro tipo e a argumentos tipo dentro de uma classe ou método
- Type erasure permite a aplicações Java que usam genéricos manter compatibilidade binária com bibliotecas e aplicativos Java criados antes dos genéricos
-

33

Type Erasure

- Por exemplo: Box<String> é transformada para o tipo Box – chamado o tipo "cru" - *raw type*
  - um raw type é o nome de uma classe ou interface genérica sem qualquer argumento tipo
  - Isso significa que não é possível descobrir em tempo de execução qual tipo de Object uma classe genérica está usando
  - As seguintes operações NÃO são possíveis:

```
public class MyClass<E> {
    public static void myMethod(Object item)
    { if (item instanceof E) { //Compiler error ... }
     E item2 = new E(); //Compiler error
    E[] iArray = new E[10]; //Compiler error
    E obj = (E)new Object(); //Unchecked cast warning } }
```

- As operações em negrito não fazem sentido em tempo de execução
  - Porque o compilador remove todas as informações sobre o argumento tipo real
- Única razão de existência de Type erasure → código novo continuar a interfacear com código legado
  - Usar um raw type por qualquer outra razão é má prática de programação !

34

Genéricos e Subtipos

- O trecho abaixo é legal?

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

  - A linha 1 certamente é legal
  - Parte capciosa da questão → linha 2
    - Ela levanta a questão: ***Uma Lista de String é uma Lista de Object ?***

- Olhemos as próximas linhas:

```
lo.add(new Object()); // 3
String s = ls.get(0); // 4: tentativa de atribuir um Object para uma String!
```

35

Genéricos e Subtipos

- No exemplo, fizemos com que **lo** fosse um **alias** de **ls**.
- Acessando **ls -** lista de String **-** através do seu alias **lo**, nós podemos inserir **objetos** arbitrários a lista
  - Resultado → ls não armazena mais apenas Strings
    - quando tentamos obter algo dela, podemos ter uma surpresa
- O compilador Java obviamente impede isso: linha 2 causa um erro em tempo de compilação
- Em geral:
  - se **Foo** é um subtipo (sub-classe ou sub-interface) de **Bar**, e
  - G é alguma declaração de tipo generic,
  - NÃO é verdade que G<Foo> é um subtipo de G<Bar>
- Princípio que vai contra nossa intuição !
  - o problema de tal intuição é que ela assume que **coleções não mudam**

36

Wildcards

- **Seja o problema:** construir uma *rotina* que escreva todos os elementos existentes em uma coleção (Collection)
- Versões anteriores de Java:

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (k = 0; k < c.size(); k++) {
    System.out.println(i.next());
 }}
```

- Tentativa ingênua de escrever usando generics:

```
void printCollection(Collection<Object> c) {
        for (Object e : c) { System.out.println(e);
}}
```

Wildcards

37

- **Problema**: nova versão muito menos útil do que antiga
  - Código anterior → poderia ser chamado com qualquer tipo de coleção como parâmetro
  - Novo → apenas recebe coleções de Objetos (Collection<Object>)
    - que, como visto, NÃO é super-tipo de todos os tipos de coleções !
- Então, qual seria o super-tipo de todos os tipos de coleções ?
  - Deve-se usar Collection<?> (coleção de *desconhecido*)
    - isto é, uma coleção cujo tipo de elemento combina com tudo (qualquer coisa)
  - Isso é chamado um *wildcard type*

Wildcards

38

- Pode-se então escrever:

```
void printCollection(Collection<?> c) {
        for (Object e : c)
          { System.out.println(e);}}
```

- E chamá-lo com qualquer tipo de coleção
- Note que dentro de printCollection(), ainda podemos ler elementos de ***c*** e lhes atribuir o tipo ***Object***
  - Isso é sempre seguro, uma vez que qualquer que seja o tipo real da coleção, ela de fato contém objetos

Wildcards

39

- Contudo, **NÃO é seguro adicionar** objetos arbitrários a coleção:

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // erro em tempo de compilação
```

- Como não sabemos a que tipo de elemento ***c*** se refere, não podemos adicionar objetos a ela
- Método add() recebe argumentos de tipo E (tipo de elemento da coleção)
- Quando o parâmetro tipo real é ? ele denota um tipo ***desconhecido***
  - Qualquer parâmetro que se passe para add teria que ser um sub-tipo desse tipo desconhecido.
  - Como não sabemos que tipo é, não podemos passar nada
  - Única exceção: null - membro de qualquer tipo

Wildcards

40

- Por outro lado, dada uma lista List<?>, nós **podemos** chamar get() e fazer uso do resultado.
- O tipo resultante é desconhecido, porém nós sempre sabemos que **ele é um objeto**
  - Portanto é seguro atribuir o resultado de get() a uma variável do tipo Object ou passá-lo para um parâmetro onde o tipo Object é esperado

Subtipos X Atribuições de elementos a Coleções

41

- Diferença entre subtipos de genéricos e herança de classes/interfaces
  Vimos que o trecho abaixo é ilegal:

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

  Uma lista de Strings NÃO é uma lista de Objects

- se **Foo** é um subtipo (sub-classe ou sub-interface) de **Bar**, e
  - G é alguma declaração de tipo generic,
  - NÃO é verdade que G<Foo> é um subtipo de G<Bar>
- Da mesma forma: o método

```
void printCollection(Collection<Object> c) {
        for (Object e : c) { System.out.println(e);
}}
```

- Só pode receber como parâmetros Coleções de Objetos
- PORÉM, PARA ATRIBUIÇÃO DE ELEMENTOS A COLEÇÕES → valem os princípios tradicionais de herança
  Uma variável do tipo Strings É um Object

Subtipos X Atribuições de elementos a Coleções

42

```
import java.util.*;
public class Teste {
   public static void main(String[] args) {
      List<Object> s = new LinkedList<Object>();
     String uma_s = new String ("objeto string");
     Object um_obj = new Object ();
     Integer um_int = new Integer(1);
     s.add("string");
     s.add (uma_s);
     s.add (um_obj);
     s.add (um_int);
      System.out.println("lista: " + s);

   }
}
```

Métodos Genéricos

43

- **Seja o problema:** escrever um método que recebe um array de objetos e uma coleção e coloca todos os objetos do array na coleção
- **Primeira tentativa:**

```
static void fromArrayToCollection(Object[] a, Collection<?> c)
{ for (Object o : a) {
    c.add(o); // erro em tempo de compilação
}}
```

- **Erro de iniciantes**: tentar usar Collection<Object> como o tipo de parâmetro da coleção
- Usar Collection<?> não vai funcionar também
  - NÃO se pode adicionar objetos em uma coleção de tipo desconhecido

- Forma de lidar com esses problemas → usar **métodos genéricos**

Métodos Genéricos

44

- Assim como declaração de tipos, **declarações de métodos** podem ser genéricas – isto é, parametrizadas por um ou mais ***parâmetros tipo*** (*type parameters*)

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c)
        { for (T o : a)
        { c.add(o); // correto
        }}
```

- Pode-se chamar esse método com qualquer tipo de coleção cujo tipo do elemento seja um super-tipo do tipo de elemento do array

Métodos Genéricos

45

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T inferido como sendo Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T inferido como sendo String
fromArrayToCollection(sa, co);// T inferido como sendo Object
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn);// T inferido como sendo Number
fromArrayToCollection(fa, cn);// T inferido como sendo Number
fromArrayToCollection(na, cn);// T inferido como sendo Number
fromArrayToCollection(na, co);// T inferido como sendo Object
fromArrayToCollection(na, cs);// erro em tempo de compilação
```

Métodos Genéricos

46

- **Observação**: **não é preciso** passar um argumento tipo real para um método genérico.
  - O compilador infere o argumento tipo, baseado nos tipos dos argumentos reais
  - Ele em geral irá inferir o argumento **tipo mais específico que fará com que a chamada fique correta**