Coleções a Tipos Genéricos em Java

SCC0604 - Programação Orientada a Objetos

Prof. Fernando V. Paulovich
http://www.icmc.usp.br/~paulovic
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

8 de novembro de 2010

Sumário

Sumário

O que são coleções na linguagem Java?

- São classes voltadas para estruturas de dados, pertencentes ao pacote **java.util**

O que são coleções na linguagem Java?

- São classes voltadas para estruturas de dados, pertencentes ao pacote **java.util**
  - Conjuntos, Conjuntos ordenados, Mapas, Mapas Ordenados e listas.

O que são coleções na linguagem Java?

- São classes voltadas para estruturas de dados, pertencentes ao pacote **java.util**
  - Conjuntos, Conjuntos ordenados, Mapas, Mapas Ordenados e listas.
- É largamente utilizado em programas: quem não precisa de um vetor, uma lista ou conjunto?

O que são coleções na linguagem Java?

- São classes voltadas para estruturas de dados, pertencentes ao pacote **java.util**
  - Conjuntos, Conjuntos ordenados, Mapas, Mapas Ordenados e listas.
- É largamente utilizado em programas: quem não precisa de um vetor, uma lista ou conjunto?
- Melhora a performance e a qualidade: evita "reinventar" a roda

O que são coleções na linguagem Java?

- São classes voltadas para estruturas de dados, pertencentes ao pacote **java.util**
  - Conjuntos, Conjuntos ordenados, Mapas, Mapas Ordenados e listas.
- É largamente utilizado em programas: quem não precisa de um vetor, uma lista ou conjunto?
- Melhora a performance e a qualidade: evita "reinventar" a roda
- Estimula o reuso

O que são coleções na linguagem Java?

- São classes voltadas para estruturas de dados, pertencentes ao pacote **java.util**
  - Conjuntos, Conjuntos ordenados, Mapas, Mapas Ordenados e listas.
- É largamente utilizado em programas: quem não precisa de um vetor, uma lista ou conjunto?
- Melhora a performance e a qualidade: evita "reinventar" a roda
- Estimula o reuso
- São conhecidos também como conjuntos

Organização em Interfaces e Classes

- O conjunto de classes java para coleções (**Java Collection Framework**) é composto de várias interfaces, e classes concretas
- Existem três interfaces principais: **Collection**, **Map** e **Queue**; que são estruturas de dados do tipo coleções, mapas e filas respectivamente
- Caso você precisar de instanciar algum conjunto, use as classes concretas

Organização em Interfaces e Classes

Hierarquia

- Collection (I)
  - Set (I)
    - HashSet (C)
    - Sorted Set (I) – TreeSet(C)
  - List (I)
    - ArrayList (C)
    - Vector (C)
    - LinkedList (C)
  - Queue (I)
    - ...
- Map
  - HashMap (C)
  - Hashtable (C)
  - SortedMap (I)
    - TreeMap (C)

(I) - interface; (C) - classe concreta

Definindo cada uma das implementações I

Coleções

- **Collection**: interface que define operações comuns de coleções. Esta interface possui duas sub-interfaces - **Set** (Conjunto) e **List** (Lista)

Conjuntos

- **Set**: implementação de **Collection**, que modela um conjunto de elementos únicos
- **HashSet**: implementação de **Set**, modela conjuntos não ordenados
- **TreeSet**: implementação de **SortedSet**, modela conjuntos ordenados

Definindo cada uma das implementações II

Listas

- **List**: modela listas de dados, onde os elementos (repetidos ou não) estão ordenados
- **ArrayList**: usa métodos não-sincronizados e **Vector** utiliza métodos sincronizados (**synchronized**)
- **Vector**: é apropriado para uso em *multithread*, porém é mais lento que **ArrayList**
- **LinkedList**: é uma Lista, onde os elementos estão ligados. Tem uma inserção e deleção muito mais rápidos que **ArrayList** e **Vector**

Definindo cada uma das implementações III

Mapas

- **Map**: modela mapeamentos entre chaves não-repetidas a valores
- **HashMap**: subclasse de **Map**, modela mapas não classificados, com métodos não-sincronizados
- **Hashtable**: subclasse de **Map**, modela mapas não classificados, com métodos sincronizados
- **SortedMap**: modela mapas classificados

Fila

- **Queue**: é uma interface, que modela filas, não será tratada aqui

Comparando com *array*

- Não confunda array (criados com []) com as classes **ArrayList** ou **Vector**

array	**ArrayList**
não tem dimensão dinâmica	tem dimensão dinâmica
suporta tipos primitivos	não suporta tipos primitivos diretamente
não é uma classe	é uma classe
não suporta métodos	suporta métodos
possui atributo **length**	não possui atributo **length**

Exemplo comparativo entre *array* e ArrayList

```
1  public class ComparandoArrayEArrayList {
2    public static void main(String[] args) {
3      // inicialização:
4      String[] ola1 = new String[3];
5      ArrayList<String> ola2 = new ArrayList<String>();
6
7      // atribuição
8      ola1[0] = "o"; ola1[1] = "l"; ola1[2] = "a";
9      ola2.add("o"); ola2.add("l"); ola2.add("a"); ola2.add("!");
10
11     // percorrendo com for
12     for (String s: ola1) {
13       System.out.print(s);
14     }
15
16     System.out.println();
17
18     for (String s: ola2) {
19       System.out.print(s);
20     }
21   }
22 }
```

Interface Collection

- A interface **Collection** define várias operações básicas e operações entre coleções

```
1  public interface Collection<E> extends Iterable<E> {
2    // Operações Básicas
3    int size();
4    boolean isEmpty();
5    boolean contains(Object element);
6    boolean add(E element);
7    boolean remove(Object element);
8    Iterator<E> iterator();
9
10   // Operações em massa
11   boolean containsAll(Collection<?> c);
12   boolean addAll(Collection<? extends E> c);
13   boolean removeAll(Collection<?> c);
14   boolean retainAll(Collection<?> c);
15   void clear();
16
17   // Operações de array
18   Object[] toArray();
19   <T> T[] toArray(T[] a);
20 }
```

Interface Iterator

- Uma interface bastante importante é a interface **Iterator**, que permite navegar (iterar) pelos vários elementos de uma coleção
- Note que a interface **Iterator** é bastante simples
  - um **Iterator** é obtido da própria coleção através do método **iterator()** e depois podemos navegar pela coleção por meio dos métodos **hasNext()**, **next()** e **remove()**

```
1 public interface Iterator<E> {
2   boolean hasNext();
3   E next();
4   void remove();
5 }
```

Exemplo simples

TreeSet(C)

```
1  public class TesteArray {
2    public static void main(String[] args) {
3      ArrayList<String> a1 = new ArrayList<String>();
4      ArrayList<String> a2 = new ArrayList<String>();
5      a1.add("a");
6      a1.add("b");
7      a1.add("c");
8      a2.add("d");
9      a2.add("e");
10     a2.add("f");
11
12     Iterator i1 = a1.iterator();
13     while (i1.hasNext()) {
14       System.out.println(i1.next());
15     }
16
17     Iterator i2 = a2.iterator();
18     while (i2.hasNext()) {
19       System.out.println(i2.next());
20     }
21   }
22 }
```

Sumário

Generics

- Incluídos na Versão 1.5 do Java

Generics

- Incluídos na Versão 1.5 do Java
- Problemas com *casting* (conversão de tipos)

Generics

- Incluídos na Versão 1.5 do Java
- Problemas com *casting* (conversão de tipos)
  - E um perigo em potencial para uma **ClassCastException**

Generics

- Incluídos na Versão 1.5 do Java
- Problemas com *casting* (conversão de tipos)
  - E um perigo em potencial para uma **ClassCastException**
  - Torna o código fonte mais poluídos

Generics

- Incluídos na Versão 1.5 do Java
- Problemas com *casting* (conversão de tipos)
  - E um perigo em potencial para uma **ClassCastException**
  - Torna o código fonte mais poluídos
  - Menos legíveis

Generics

- Incluídos na Versão 1.5 do Java
- Problemas com *casting* (conversão de tipos)
  - E um perigo em potencial para uma **ClassCastException**
  - Torna o código fonte mais poluídos
  - Menos legíveis
  - Destrói benefícios de uma linguagem com tipos fortemente definidos

Generics

- Por que *generics*?

Generics

- Por que *generics*?
  - Permite que uma única classe trabalhe com uma grande variedade de tipos

Generics

- Por que *generics*?
  - Permite que uma única classe trabalhe com uma grande variedade de tipos
  - E uma forma natural de eliminar a necessidade de se fazer *cast*

Generics

- Por que *generics*?
  - Permite que uma única classe trabalhe com uma grande variedade de tipos
  - E uma forma natural de eliminar a necessidade de se fazer *cast*
  - Preserva benefícios da checagem de tipos

Problema com *Casting*

- Existem dois tipos de *casting*
  - Transformar um subtipo em um supertipo (*upercasting*)
  - Transformar um supertipo em um subtipo (*downcasting*)

```
1 ArrayList strings = new ArrayList();
2
3 strings.add("1"); //upcasting (transforma String em Object)
4
5 String msg = (String)strings.get(0); //downcasting (transforma Object ↩
      em String)
6 Integer i = (Integer)strings.get(0); //erro de downcasting
```

Problema com *Casting*

- Existem dois tipos de *casting*
  - Transformar um subtipo em um supertipo (*upercasting*)
  - Transformar um supertipo em um subtipo (*downcasting*)

```
1  ArrayList strings = new ArrayList();
2
3  strings.add("1"); //upcasting (transforma String em Object)
4
5  String msg = (String)strings.get(0); //downcasting (transforma Object ↩
       em String)
6  Integer i = (Integer)strings.get(0); //erro de downcasting
```

- Atribuir outra coisa além de **String** a saída do método **get** causará um erro de *downcasting*

Problema com *Casting*

- Para evitar que usos indevidos de *casting* possam trazer problemas em tempo de execução, a versão *Generics* da classe **ArrayList** deve ser usada

Problema com *Casting*

- Para evitar que usos indevidos de *casting* possam trazer problemas em tempo de execução, a versão *Generics* da classe **ArrayList** deve ser usada
- A partir da versão 1.5, Java fornece aos usuários versões que utilizam *Generics* para todas as classes **Collection**

Problema com *Casting*

- O seguinte código evita problemas de *downcasting*/*upcasting* em tempo de execução

```
1  ArrayList<String> strings = new ArrayList<String>();
2
3  strings.add("1"); //não precisa de upcasting
4
5  String msg = strings.get(0); //não precisa de downcasting
6  Integer i = (Integer)strings.get(0); //erro em tempo de compilação
```

Declarando Classe Utilizando *Generics*

- Para criar uma classe que use tipos genéricos declaro **class BasicGeneric<A>**

Declarando Classe Utilizando *Generics*

- Para criar uma classe que use tipos genéricos declaro **class BasicGeneric<A>**
- Isso define que minha classe contém um tipo genérico <A>

Declarando Classe Utilizando *Generics*

- Para criar uma classe que use tipos genéricos declaro **class BasicGeneric<A>**
- Isso define que minha classe contém um tipo genérico <A>
- Nessa classe, um atributo poderia ser declarado como **private A data**

Declarando Classe Utilizando *Generics*

- Para criar uma classe que use tipos genéricos declaro **class BasicGeneric<A>**
- Isso define que minha classe contém um tipo genérico <A>
- Nessa classe, um atributo poderia ser declarado como **private A data**
- O atributo **data** é de um tipo *Generic* e depende do tipo de dado com que um objeto *BasicGeneric* for desenvolvido para trabalhar

Declarando Classe Utilizando *Generics*

- Para criar uma classe que use tipos genéricos declaro **class BasicGeneric<A>**
- Isso define que minha classe contém um tipo genérico <A>
- Nessa classe, um atributo poderia ser declarado como **private A data**
- O atributo **data** é de um tipo *Generic* e depende do tipo de dado com que um objeto *BasicGeneric* for desenvolvido para trabalhar

Declarando Classe Utilizando *Generics*

- Para criar uma classe que use tipos genéricos declaro **class BasicGeneric<A>**
- Isso define que minha classe contém um tipo genérico <A>
- Nessa classe, um atributo poderia ser declarado como **private A data**
- O atributo **data** é de um tipo *Generic* e depende do tipo de dado com que um objeto *BasicGeneric* for desenvolvido para trabalhar

```
1 public class BasicGeneric<A> {
2   private A data;
3   ...
4 }
```

Declarando Classe Utilizando *Generics*

- Na instanciação de um objeto da classe deve ser especificado o tipo que será genérico

```
1 BasicGeneric<String> basicGeneric1 = new BasicGeneric<String>();
2 BasicGeneric<Integer> basicGeneric2 = new BasicGeneric<Integer>();
```

Declarando Classe Utilizando Generics

- Declaração do método getData

```
1  public class BasicGeneric<A> {
2    private A data;
3
4    ...
5
6    public A getData() {
7      return data;
8    }
9  }
```

Declarando Classe Utilizando Generics

- Instâncias da classe BasicGeneric "presa" ao tipo String

```
1 BasicGeneric<String> basicGeneric = new BasicGeneric<String>();
2 String data = basicGeneric.getData(); //não precisa de casting
```

- Instâncias da classe BasicGeneric "presa" ao tipo Integer

```
1 BasicGeneric<Integer> basicGeneric = new BasicGeneric<Integer>();
2 Integer data = basicGeneric.getData(); //não precisa de casting
```

Generics: Limitação "Primitiva"

- Tipos *Generics* do Java são restritos a tipos de referência (objetos) e não funcionarão com tipos primitivos

```
1 BasicGeneric<int> basicGeneric = new BasicGeneric<int>(data1);
```

Generics: Limitação "Primitiva"

- Tipos *Generics* do Java são restritos a tipos de referência (objetos) e não funcionarão com tipos primitivos

```
1 BasicGeneric<int> basicGeneric = new BasicGeneric<int>(data1);
```

Generics: Limitação "Primitiva"

- Tipos *Generics* do Java são restritos a tipos de referência (objetos) e não funcionarão com tipos primitivos

```
1 BasicGeneric<int> basicGeneric = new BasicGeneric<int>(data1);
```

- Solução:

Generics: Limitação "Primitiva"

- Tipos *Generics* do Java são restritos a tipos de referência (objetos) e não funcionarão com tipos primitivos

```
1 BasicGeneric<int> basicGeneric = new BasicGeneric<int>(data1);
```

- Solução:
  - Encapsular tipos primitivos antes de usá-los

Generics: Limitação "Primitiva"

- Tipos *Generics* do Java são restritos a tipos de referência (objetos) e não funcionarão com tipos primitivos

```
1 BasicGeneric<int> basicGeneric = new BasicGeneric<int>(data1);
```

- Solução:
  - Encapsular tipos primitivos antes de usá-los
  - Utilizar tipos encapsuladores (*wrapper types*) como argumentos para um tipo *Generics* (**Integer**, **Float**, etc.)

Generics Limitados

No exemplo anterior:

- Os parâmetros de tipo da classe **BasicGeneric** podem ser de qualquer tipo de dado de referência (**Object**)

Generics Limitados

No exemplo anterior:

- Os parâmetros de tipo da classe **BasicGeneric** podem ser de qualquer tipo de dado de referência (**Object**)

Generics Limitados

No exemplo anterior:

- Os parâmetros de tipo da classe **BasicGeneric** podem ser de qualquer tipo de dado de referência (**Object**)

- E possível restringir os tipos em potencial usados em instanciações de uma classe que utiliza *Generics*

Generics Limitados

No exemplo anterior:

- Os parâmetros de tipo da classe **BasicGeneric** podem ser de qualquer tipo de dado de referência (**Object**)

- E possível restringir os tipos em potencial usados em instanciações de uma classe que utiliza *Generics*
- Para limitar as instanciações a um certo tipo, declara-se o tipo genérico como estendendo (extends) esse certo tipo

Generics Limitados

No exemplo anterior:

- Os parâmetros de tipo da classe **BasicGeneric** podem ser de qualquer tipo de dado de referência (**Object**)

- E possível restringir os tipos em potencial usados em instanciações de uma classe que utiliza *Generics*
- Para limitar as instanciações a um certo tipo, declara-se o tipo genérico como estendendo (extends) esse certo tipo

Generics Limitados

No exemplo anterior:

- Os parâmetros de tipo da classe **BasicGeneric** podem ser de qualquer tipo de dado de referência (**Object**)

- E possível restringir os tipos em potencial usados em instanciações de uma classe que utiliza *Generics*
- Para limitar as instanciações a um certo tipo, declara-se o tipo genérico como estendendo (extends) esse certo tipo

```
1 public class ClassName <ParameterName extends ParentClass> {
2 ...
3 }
```

Generics Limitados

- Permite uma checagem estática de tipos adicional

Generics Limitados

- Permite uma checagem estática de tipos adicional
  - Garante que toda instanciação de um tipo *Generic* respeita as restrições que atribuímos a ele

Generics Limitados

- Permite uma checagem estática de tipos adicional
  - Garante que toda instanciação de um tipo *Generic* respeita as restrições que atribuímos a ele
  - Pode-se chamar, de forma segura, qualquer método encontrado no tipo estendido

Generics Limitados

- Permite uma checagem estática de tipos adicional
  - Garante que toda instanciação de um tipo *Generic* respeita as restrições que atribuímos a ele
  - Pode-se chamar, de forma segura, qualquer método encontrado no tipo estendido
- Quando não existir um limite explicito no parâmetro genérico

Generics Limitados

- Permite uma checagem estática de tipos adicional
  - Garante que toda instanciação de um tipo *Generic* respeita as restrições que atribuímos a ele
  - Pode-se chamar, de forma segura, qualquer método encontrado no tipo estendido
- Quando não existir um limite explicito no parâmetro genérico
  - O limite padrão é **Object**, portanto o que se pode usar são os métodos de **Object**

Sumário

A Classe **Class**

Identificação de Tipo em Tempo de Execução

- Enquanto um **programa está executando** a JVM está sempre mantendo o que é chamado de **identificação de tipo** em tempo de execução **de todos os objetos**

A Classe **Class**

Identificação de Tipo em Tempo de Execução

- Enquanto um **programa está executando** a JVM está sempre mantendo o que é chamado de **identificação de tipo** em tempo de execução **de todos os objetos**
- Essa informação pode ser acessada usando a classe **Class**. Essa classe pode ser obtida via método **getClass()** da classe **Object**

A Classe **Class**

```
1 public class Principal {
2   public static void main(String[] args) {
3     Gerente emp = new Gerente("CHEFE", 10);
4     Class c = emp.getClass();
5     System.out.println(c.getName());
6   }
7 }
```

A Classe **Class**

- Pode-se obter o objeto **Class** de duas maneiras:

A Classe **Class**

- Pode-se obter o objeto **Class** de duas maneiras:
  - Perguntando a um objeto pelo seu objeto **Class** correspondente

A Classe **Class**

- Pode-se obter o objeto **Class** de duas maneiras:
  - Perguntando a um objeto pelo seu objeto **Class** correspondente
  - Perguntando qual é o objeto **Class** que corresponde a um certa string usando o método **forName()**

A Classe **Class**

```
1  public class Principal {
2    public static void main(String[] args) {
3      try {
4        Class c = Class.forName("heranca.Gerente");
5        System.out.println(c.getName());
6      } catch(ClassNotFoundException e) {
7        e.printStackTrace();
8      }
9    }
10 }
```

A Classe **Class**

- Um método útil é o que permite criar um objeto qualquer a partir de um objeto **Class**

A Classe **Class**

- Um método útil é o que permite criar um objeto qualquer a partir de um objeto **Class**
- Isso pode ser feito através do método **newInstance()**

A Classe **Class**

- Um método útil é o que permite criar um objeto qualquer a partir de um objeto **Class**
- Isso pode ser feito através do método **newInstance()**
- O método **newInstance()** chama o construtor padrão (sem argumentos) para criar o objeto

A Classe **Class**

```
1  public class Principal {
2    public static void main(String[] args) {
3      try {
4        Gerente g = (Gerente)Class.forName("heranca.Gerente").↩
             newInstance();
5
6        System.out.println(g);
7      } catch(Exception e) {
8        e.printStackTrace();
9      }
10   }
11 }
```

A Classe **Class**

```
1  public class Gerente extends Empregado {
2    ....
3
4    public Gerente(){
5      this("Gerente",0);
6    }
7  }
```

Reflexão

- É possível se utilizar o mecanismo de reflexão Java para descobrir os métodos, construtores e atributos que uma classe oferece, mesmo está classe já estando compilada

Reflexão

- É possível se utilizar o mecanismo de reflexão Java para descobrir os métodos, construtores e atributos que uma classe oferece, mesmo está classe já estando compilada
- Maiores informações: seção Reflexão, livro Core Java 2 Volume I : Fundamentos, pág. 183