Programação em Java com J2SE 5.0

Tipos genéricos

Helder da Rocha

Julho 2005

O que são os Genéricos?

- Recurso introduzido no Java 5.0
  - Sintaxe que permite restringir os tipos de dados aceitos por referências genéricas
  - Permite verificação de tipo em tempo de compilação!
- Exemplo: coleções
  - Permitem agrupar objetos de diferentes tipos, devido à sua interface genérica (baseada em Object)

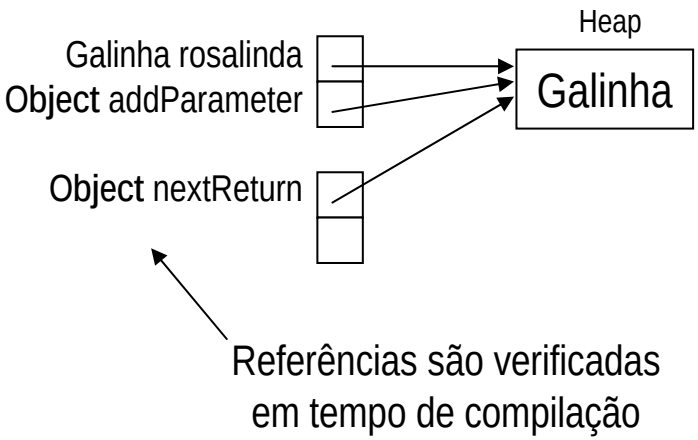
```
interface List {
    public void add(Object item);
    public Object get(int index);
    public Iterator iterator();
    ...
}

interface Iterator {
    public Object next();
    ...
}
```

Vantagem: aceita qualquer tipo

Desvantagem: retorna tipo genérico e requer downcasting que só é verificado em tempo de execução

O Problema

- Antes dos genéricos, era preciso fazer coerção (cast) para usar os dados recuperados de coleções



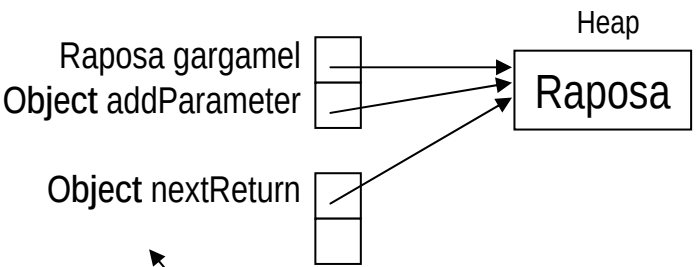

```
Fazenda fazenda = Fazenda.getInstance();

Galinha g1 = new Galinha("Chocagilda");
Galinha g2 = new Galinha("Cocotalva");
// galinhas entram no galinheiro
g1.entrarNoGalinheiro(fazenda.getGalinheiro());
g2.entrarNoGalinheiro(fazenda.getGalinheiro());

Fazendeiro ze = new Fazendeiro(fazenda);
Iterator galinhas = ze.pegarGalinhas();
while(galinhas.hasNext()) {
    Galinha galinha = (Galinha) galinhas.next();
}
```

```
class Fazenda {
    ...
  private List galinheiro =
       new ArrayList();
  public List getGalinheiro() {
    return galinheiro;
  }
}
```

```
class Galinha {
    public void entrarNoGalinheiro(List galinheiro) {
        galinheiro.add(this);
    }
}
```

O Problema (2)

- E havia o risco do programador cometer um erro (já que não há verificação de tipos de objeto na compilação)



Referências estão corretas:
Trata-se de um Object!
Mas não dá para fazer cast de para uma referência Galinha

```
g1.entrarNoGalinheiro(fazenda.getGalinheiro());
g2.entrarNoGalinheiro(fazenda.getGalinheiro());
...
// entra uma raposa!!!
Raposa galius = new Raposa("Galius");
galius.assaltar (fazenda.getGalinheiro());

Fazendeiro ze = new Fazendeiro(fazenda);
Iterator galinhas = ze.pegarGalinhas();
while(galinhas.hasNext()) {
    Galinha galinha =
      (Galinha) galinhas.next();
}
```

FALHA! Há uma raposa no meio das galinhas

[java] Exception in thread "main" java.lang.ClassCastException: Raposa

```
class Raposa {
    public void assaltar(List lugar) {
        lugar.add(this);
    }
}
```

Correto! List tem add(Object) e Raposa é um Object

Solução sem usar genéricos

- Typesafe collection: padrão de design
  - Apenas Galinhas são aceitas no galinheiro
  - Para recuperar objetos, não é necessário usar cast!
  - Mas Galinheiro não é List,
  - E GalinhaIterator não é Iterator

```
import java.util.*;
public class Galinheiro {
  private List galinhas;
  public Galinheiro(List list) {
    galinhas = list;
  }
  public GalinhaIterator iterator() {
    return new GalinhaIterator(galinhas.iterator());
  }
  public Galinha get(int idx) {
    return (Galinha) galinhas.get(idx);
  }
  public void add(Galinha g) {
    galinhas.add(g);
  }
}
```

```
import java.util.*;
public class GalinhaIterator {
  private Iterator iterator;
  GalinhaIterator(Iterator it) {
    iterator = it;
  }
  public boolean hasNext() {
    return iterator.hasNext();
  }
  public Galinha next() {
    return (Galinha)iterator.next();
  }
}
```

Não compila!

Não requer cast!

```
Galinheiro g =
  new Galinheiro(new ArrayList());
g.add(new Galinha("Frida"));
// g.add(new Raposa("Galius"));
Galinha frida = g.get(0);
```

Solução com genéricos (1)

- Genéricos permitem associar um tipo à referência
  - Restringe tipos permitidos ao passado como parâmetro
  - Permite verificação em tempo de compilação

Parâmetro de tipo
(type parameter)

```
class Fazenda {
    ...
  private List<Galinha> galinheiro = new ArrayList<Galinha>();

  public List<Galinha> getGalinheiro() {
    return galinheiro;
  }
}
```

Isto não é um ArrayList de Object
mas um ArrayList de Galinha

```
class Galinha {
    public void entrarNoGalinheiro(List<Galinha> galinheiro) {
        galinheiro.add(this);
    }
}
```

- Sintaxe básica: **`TipoGenerico<TipoEspecifico>`**

Solução com genéricos (2)

- Agora não precisamos mais do cast

```
Fazendeiro ze = new Fazendeiro(fazenda);
Iterator<Galinha> galinhas = ze.pegarGalinhas();
while(galinhas.hasNext()) {
    Galinha galinha = galinhas.next(); // sem cast!!!
}
```

- Raposas também não entram mais
  - Precisam especificar tipo de lugar. O compilador avisa (warning) sobre uso de objetos inseguros:

```
[javac] Note: C:\usr\projects\Fragmentos.java uses
        unchecked or unsafe operations.
```

  - Conserta-se a Raposa:

```
class Raposa {
  public void assaltar(List<Object> lugar) { lugar.add(this); }
}
```

Se a Raposa fosse esperta, declararia como <?>

mas mesmo assim não iria funcionar!

  - O **compilador** acusa o erro!

```
[javac] C:\usr\projects\Fragmentos.java:16:
        assaltar(java.util.List<java.lang.Object>) in Raposa
        cannot be applied to (java.util.List<Galinha>)
```

Nova interface java.util.Collection

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

Definição de genéricos

- Sintaxe de **declaração**: trecho de java.util.List

```
public interface List<E> extends Collection<E> {
    void add(E element);
    E get(int index);
    Iterator<E> iterator();
    boolean addAll(Collection<? extends E> c);
    <T> T[] toArray(T[] a);
}
```

  - Convenção: defina parâmetros de tipo com uma letra, de caixa-alta (ex: **T** de tipo, **V** de valor, **E** de Elemento, **K** de key)

- Sintaxe de **uso**

```
List<Number> numeros =
   new ArrayList<Number>();
numeros.add(new Integer(5));
numeros.add(new Double(3.14));
Number pi = numeros.get(1);

Collection<Long> longs =
   new HashSet<Long>();
longs.add(123L);
longs.add(0xBABEL);
```

```
boolean success = numeros.addAll(longs);

Number[] dummy = {};
Number[] array = numeros.toArray(dummy);

for (Number n: array) {
    System.out.println(n);
}
```

Compilação de genéricos

- Uma declaração de tipo genérico é compilada e gera uma classe Java comum, como outra qualquer
  - Não é um template! Não há geração especial de código. Informação de generics serve para compilação e é apagada no bytecode final
  - Não aumenta o tamanho do código: veja os arquivos .class gerados e compare: são iguais aos anteriores.
- Genéricos funcionam como estruturas que aceitam parâmetros em tempo de execução
  - Uma razoável analogia é comparar com a passagem de argumentos para parâmetros em métodos

Uma boa analogia

- Em métodos
  - Na **declaração** parâmetros formais de valor descrevem os valores que o método pode aceitar;
  - Quando um método é chamado, os argumentos passados (valores e referências) substituem os parâmetros formais
- Em genéricos:
  - Na **declaração**, parâmetros formais de tipo descrevem os tipos que o genérico pode representar;
  - Quando um tipo genérico é usado, os argumentos passados (tipos) substituem os parâmetros formais

Métodos		Tipos Genéricos

`interface Exemplo {`
`  int metodo(int x);`
`}`
	Declaração	`class ArrayList<E>`
		`  implements List<E> {`
		`... }`

`Exemplo e = ...` `int r = e.metodo(5);`	Uso	`List<Integer> lista = new` `  ArrayList<Integer>();`

Hierarquia de subtipos

Se **X** é um subtipo de **Y**, e **G** um tipo genérico, **não** é verdade que **G<X>** é um subtipo de **G<Y>**

- Exemplo: o código abaixo é ilegal

```
List<Galinha> galinhas = new ArrayList<Galinha>();
galinhas.add(new Galinha("Cocogilda"));
List<Object> objetos = galinhas; // ILEGAL!
```

Causa erro de compilação

- Ah, mas com arrays não seria!

```
Galinha[] vg = { new Galinha("Cocogilda") };
Object[] vo = vg;
```

- Sim, mas arrays são imutáveis; coleções não!
  - Problemas poderiam acontecer se coleção crescesse

```
objetos.add(new Raposa()); // ok, pois Raposa é Object
Galinha gilda = galinhas.get(0); // Raposa fantasiada
```

Flexibilidade com curingas

- De acordo com a regra anterior:
  - Collection<Object> não é supertipo de Collection<Integer>
  - Collection<Object> só aceita componentes Object!
- O supertipo genérico verdadeiro é **Collection<?>**
  - **?** é o **"tipo desconhecido"**
  - O tipo Collection<?> portanto é uma **Collection do desconhecido**
- Exemplo: o método abaixo aceita coleções que contenham qualquer tipo

```
void printCollection(Collection<?> c) { ... }
```

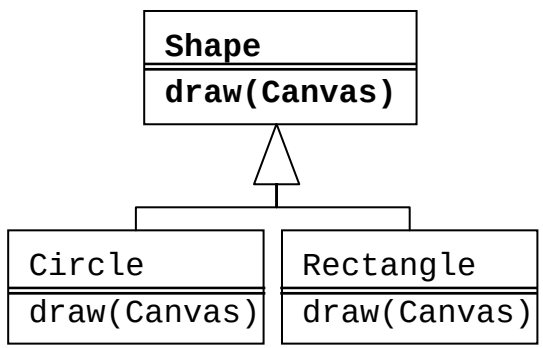
- Limitações
  - Coleções de tipos desconhecidos podem ter seus componentes lidos, mas não pode receber novos componentes

```
c.add(new Object()); // dentro do método acima, é ilegal
Object obj = c.get(0); // é legal, assumindo que o elemento 0 exista
```

Impondo limites

- Pode-se impor limites aos curingas
- Limites superiores: **`<? extends T>`**
  - Aceita T e todos os seus descendentes
  - Ex: se T for Collection, aceita List, Set, ArrayList, etc.
- Limites inferiores: **`<? super T>`**
  - Aceita T e todos os seus ascendentes
  - Ex: se T for ArrayList, aceita List, Collection, Object
- Exemplo: método que aceita qualquer **Shape**

```
class Canvas {
    public void drawAll(List<? extends Shape> shapes) {
        for(Shape s: shapes) {
            s.draw(this);
        }
    }
}
```



Variações

- Tipos genéricos podem receber mais argumentos
- Exemplo: `Map<K, V>`
  - **K** = key, **V** = value
  - Exemplo:

    ```
    Map<String, ? extends Employee>
    ```

    usa uma **String** como chave para elementos que podem ser quaisquer subclasses de **Employee**

Métodos genéricos

- Usados quando existem **dependências** entre o tipo de retorno e os parâmetros
- Exemplo de sintaxe

```
mods <T1, T2> tipoRetorno nome(Item<T1>, T2) {...}
```

- Exemplo
  - declaração

```
static <T> void arrayToCollection(T[] arr, Collection<T> col) {
    for (T obj: arr) {
        col.add(obj);
    }
}
```

  - uso

```
Collection<Number> nums = new ArrayList<Number>();
arrayToCollection(array, nums);
```

Curingas vs. métodos genéricos

- Pode-se implementar com métodos genéricos o que se implementa com curingas. As implementações abaixo são equivalentes
  - Collection original

```
interface Collection<E> {
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
}
```

  - Collection implementada com método genérico

```
interface Collection<E> {
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
}
```

- Questão:
  - Quando usar métodos genéricos?
  - Quando usar curingas?
- Só use métodos genéricos quando não for possível usar curingas
  - Se não existir dependência entre parâmetros e/ou tipos de retorno, deve-se preferir curingas

Genéricos estão para ficar

- Genéricos estão em toda a API Java
  - Aprenda a usá-los. Não tente fugir deles!
  - **Familiarize-se** com a sintaxe (vai ficar mais simples com o tempo e o costume) e evite abusar dela: quando algo puder ser feito de duas formas prefira sempre a mais simples
- Duas implementações equivalentes
  - Qual você escolheria?

a)
```
public static <T> void copy(List<T> dest, List<? extends T> src) {}
```

b)
```
public static <T, S extends T> void copy(List<T> dest, List<S> src) {}
```

- Familiarize-se com a sintaxe. Use generics já!
  - 99% das aplicações resolvem tudo com sintaxe simples
  - Qual é o tipo de history?

```
static List<List<? extends Shape>> history =
   new ArrayList<List<? extends Shape>>();
```

Código legado em genéricos

- Como usar APIs e frameworks em código antigo, no seu código novo que usa genéricos?
  - Não é preciso fazer nada de especial. Basta tratar o código como se fosse pré-Java 5.0
- Exemplo: pode-se passar para um método que aceita **`Classe`**, qualquer **`Classe<Tipo>`**
  - Classe é um **tipo cru** (raw type). É parecido com **`Classe<?>`** mas não causa nenhuma verificação de precisão de tipos
  - É inseguro, portanto o compilador imprime um aviso (warning) que não pode garantir a precisão do código
  - Tipos crus existem apenas para garantir a compatibilidade reversa

Genéricos em código legado

- Como usar APIs novas (com genéricos) em código antigo?
  - Isto já acontece se você compila seu código antigo que usa collections no Java 5.0
  - Você pode passar tipos crus para tipos genéricos checados sem causar erro de compilação
  - O compilador não tem como garantir a precisão dos tipos e avisa com um *warning*.
- Se **todo** o seu código usar genéricos, toda a verificação de tipos de referência será feita em tempo de compilação e não haverá erro de *cast*

Equivalência

- O que você acha que o trecho seguinte imprime?

```
List <String> strings = new ArrayList<String>();
List <Integer> integers = new ArrayList<Integer>();

System.out.println(strings.getClass().getName());
System.out.println(integers.getClass().getName());
System.out.println(strings.getClass() == integers.getClass());
```

```
java.util.ArrayList
java.util.ArrayList
true
```

- O motivo
  - Todas as instâncias de uma classe genérica tem a **mesma classe** em tempo de execução, indepentemente dos parâmetros de tipo passados
- Uma classe não é mais sinônimo de tipo
  - Uma classe genérica pode ter muitos tipos

Casts e instanceof

- Como classes de tipos diferentes são equivalentes, não faz sentido usar cast ou **instanceof** com tipos genéricos
- O seguinte trecho é ilegal (causa erro de compilação)

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ... }
```

- O trecho abaixo causa aviso (warnings do compilador) já que a informação de cast não é útil para o sistema de tempo de execução

```
Collection <String> cstr = (Collection<String>) cs;
```

- Casts com variáveis de tipo são inúteis, uma vez que elas não existem em tempo de execução (causa warning)

```
<T> T badCast(T t, Object o) {
    return (T)o;
}
```

Arrays

- O tipo dos componentes de um array de tipos genéricos não pode ser uma variável de tipo
- O seguinte trecho é ilegal

```
List<String>[] lsa = new List<String>[10]; //  ERRO!!
```

- O seguinte trecho é permitido, porém causa um aviso do compilador (pode causar erro de tempo de execução)

```
List<String>[] lsa = new List<?>[10]; // advertência!
String s = lsa[0].get(0); // possível erro runtime
```

- O seguinte trecho é legal, porém para extrair um String do array será preciso usar cast

```
List<?>[] lsa = new List<?>[10]; // ok!
String s = (String) lsa[0].get(0);
```

java.lang.Class

- No Java 1.5, java.lang.Class é genérico!
- Pode-se fazer

```
Class<String> c = String.class;
Class<Pessoa> pc =
          Class.forName("Pessoa");
Employee e = new Employee();
Class<Employee> = e.getClass();
```

- O método **newInstance()** agora retorna um objeto do tipo da classe onde é chamado

```
Pessoa joao = pc.newInstance();
```

Como começar a usar genéricos

- Explore a documentação
  - Veja principalmente o pacote de coleções de Java: reveja as classes que você já usa e as novas assinaturas dos métodos
- Aplique genéricos no seu código
  - Crie novos programas usando genéricos
  - Altere programas existentes gradualmente
  - Compile (javac) usando a opção –Xlint para identificar trechos obsoletos
    - No Ant, use

```
<javac ...>
   <compilerarg line="-Xlint" ...>
```

Fontes de pesquisa

[1] Gilad Bracha, "Generics in the Java Programming Language" a.k.a "The Generics Tutorial", Julho de 2004.

- Distribuído com a documentação Java. A maior parte desta apresentação foi construído com exemplos desse tutorial.

[2] James Gosling, Bill Joy, Guy Steele e Gilad Bracha, "The Java Language Specification, 3rd Edition". Addison-Wesley, 2005 (disponível em java.sun.com).

- Tópicos de interesse: 4.4 Type Variables, 4.5 Parameterized Types, 4.6 Type Erasure, 4.7 Reifiable Types, 4.8 Raw Types, 4.9 Intersection Types, 8.1.2 Generic Classes and Type Parameters, 8.4.4 Generic Methods, 8.8.4 Generic Constructors, 9.1.2 Generic Interfaces (e vários outros)

[3] Angelika Langer, "The Generics FAQ". 2004-2005

- www.langer.camelot.de/GenericsFAQ/JavaGenericsFAQ.html
- Contém uma investigação detalhada sobre os aspectos mais obscuros de Java Generics

[4] Bruce Eckel, "Templates vs. Generics", 08/2004

- mindview.net/WebLog/log-0061
- Discussão explorando as diferenças entre C++ Templates e Java Generics, procurando soluções para os principais problemas