9

CLASSES GENÉRICAS

Paradigmas de Programação

LEI - ISEP

Luiz Faria, adaptado de Donald W. Smith (TechNeTrain.com)

Objetivos

- Compreender o objetivo da programação genérica
- Implementar classes e métodos genéricos
- Explicar a execução de métodos genéricos na máquina virtual
- Descrever as limitações da programação genérica em Java

2

Conteúdos

- Classes Genéricas e Parâmetros de Tipo
- Implementação de Tipos Genéricos
- Métodos Genéricos
- Imposição de restrições aos Parâmetros de Tipo
- *Type Erasure* (*Técnica de Apagamento*)

3

Classes Genéricas e Parâmetros de Tipo

- **Programação genérica:** criação de blocos de código que podem ser usados com tipos diferentes
  - Em Java, pode ser feito com parâmetros de Tipo ou com herança
  - Exemplo de parâmetro de tipo: `ArrayList<String>`)
  - Programação genérica por herança: utilização de referências do tipo `Object` para referenciar qualquer tipo de objecto

4

Classes Genéricas e Parâmetros de Tipo

- *Classe genérica*: declarada com um ou mais parâmetros de tipo
- Exemplo: A biblioteca Java declara a classe `ArrayList<E>`
  - `E` é a variável tipo que define o tipo de elemento
  - Esta mesma variável é depois usada na declaração de métodos:

```
public void add(E element)
public E get(int index)
```

5

Parâmetros de Tipo (1)

- Parâmetros de tipo podem ser instanciados com uma classe ou um tipo de interface:

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

- Não é possível usar um tipo primitivo com variável de tipo:

```
ArrayList<double> // Errado!
```

- Em alternativa deve usar-se a *wrapper class* correspondente:

```
ArrayList<Double>
```

6

Parâmetros de Tipo (2)

- O tipo fornecido substitui a variável de tipo na declaração da classe
- Exemplo: `add` em `ArrayList<BankAccount>` vê a variável de tipo `E` substituída por `BankAccount`:

```
public void add(BankAccount element)
```

- No caso da utilização de herança teríamos a seguinte situação:

```
public void add(Object element)
```

7

Parâmetros de Tipo: Vantagens

- Tornam o código genérico mais robusto/seguro e mais legível
  - Não é possível adicionar uma `String` a um `ArrayList<BankAccount>`
  - No entanto, no caso da utilização de herança já seria possível adicionar uma `String` a uma estrutura destinada a armazenar contas bancárias
- Simplicidade
  - Não é necessário proceder a *Type Casting* para extrair os objetos das coleções

8

Parâmetros de Tipo: Aumento da Robustez

Considerando que `ListaLigada` usa herança

```
ArrayList<BankAccount> accounts1 =
   new ArrayList<BankAccount>();
ListaLigada accounts2 = new ListaLigada();
// Deve armazenar objetos do tipo BankAccount
accounts1.add("my savings");
// Erro durante a compilação
accounts2.add("my savings");
// Não detetado durante a compilação
 . . .
 BankAccount account = (BankAccount) accounts2.getFirst();
// Erro de run-time
```

9

Implementação de Tipos Genéricos

- Exemplo: classe genérica simples que armazena pares de objetos

```
Pair<String, Integer> result =
      new Pair<String, Integer>("Harry Morgan", 1729);
```

- Os métodos `getFirst` e `getSecond` obtêm o primeiro e segundo valores do par:

```
String name = result.getFirst();
Integer number = result.getSecond();
```

10

Implementação de Tipos Genéricos

- Exemplo de utilização: devolução de dois valores em simultâneo (o método devolve um objeto do tipo `Pair`)
- A classe genérica `Pair` requer dois parâmetros de tipo, um para cada tipo de elemento rodeado por < e >:

```
public class Pair<T, S>
```

11

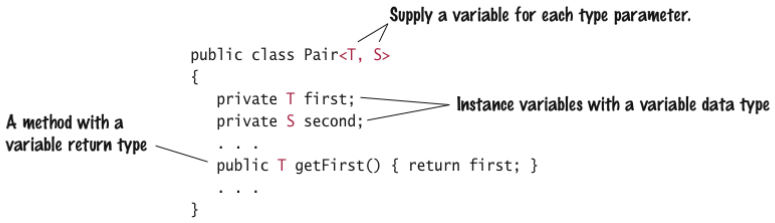
Sugestões de Nomes para as Variáveis

Variável de tipo	Significado
`E`	Tipo de elementos de uma coleção
`K`	Tipo da chave (Key) num Map
`V`	Tipo do Valor num Map
`T`	Tipo em geral
`S, U`	Tipos em geral (adicionais)

12

Declaração de uma Classe Genérica

Syntax *accessSpecifier* `class` *GenericClassName*`<`*TypeVariable*$_1$`,` *TypeVariable*$_2$`, . . .>`
`{`
instance variables
constructors
methods
`}`



13

Pair.java

```
1  /**
2     Esta classe agrega um par de elementos de tipos diferentes.
3  */
4  public class Pair<T, S>
5  {
6     private T first;
7     private S second;
8
9     /**
10       constrói um pair contendo dois elementos recebidos.
11       @param firstElement primeiro elemento
12       @param secondElement segundo elemento
13    */
14    public Pair(T firstElement, S secondElement)
15    {
16       first = firstElement;
17       second = secondElement;
18    }
19
```

Continua

14

Pair.java (cont.)

```
20      /**
21          Obtém o primeiro elemento deste par.
22          @return o primeiro elemento
23      */
24      public T getFirst() { return first; }
25
26      /**
27          Obtém o segundo elemento deste par.
28          @return o  segundo elemento
29      */
30      public S getSecond() { return second; }
31
32      public String toString() {
            return "(" + first + ", " + second + ")"; }
33  }
```

15

PairDemo.java

```
1   public class PairDemo
2   {
3      public static void main(String[] args)
4      {
5         String[] names = { "Tom", "Diana", "Harry" };
6         Pair<String, Integer> result = firstContaining(names, "a");
7         System.out.println(result.getFirst());
8         System.out.println("Expected: Diana");
9         System.out.println(result.getSecond());
10        System.out.println("Expected: 1");
11     }
12
```

Continua

16

PairDemo.java (cont.)

```
13      /**
14         Obtém a primeira String contendo uma dada string, juntamente
15         com o seu indíce.
16         @param strings um array de strings
17         @param sub uma string
18         @return um pair (strings[i], i) onde strings[i] é a primeira
19         strings[i] contendo str, ou um pair (null, -1) se não há
20         correspondência.
21      */
22      public static Pair<String, Integer> firstContaining(
23         String[] strings, String sub)
24      {
25         for (int i = 0; i < strings.length; i++)
26         {
27            if (strings[i].contains(sub))
28            {
29               return new Pair<String, Integer>(strings[i], i);
30            }
31         }
32         return new Pair<String, Integer>(null, -1);
33      }
34   }
```

Continua

17

PairDemo.java (cont.)

Execução:

```
Diana
Expected: Diana
1
Expected: 1
```

18

Métodos Genéricos

- **Método Genérico:** método com um parâmetro de tipo
- Pode ser definido no interior de classes não genéricas
- Exemplo: Pretende-se declarar um método capaz de imprimir o conteúdo de um *array* de qualquer tipo:

```
public class ArrayUtil
{
   /**
      Imprime todos os elementos contidos num array.
      @param a: o array a imprimir
   */
   public static <T> void print(T[] a)
   {
      . . .
   }
   . . .
}
```

19

Métodos Genéricos (2)

- Para implementar um método genérico será mais simples começarmos por implementar um método concreto (por exemplo, um método que imprima todos os elementos contidos num *array* de *strings*):

```
public class ArrayUtil
{
   public static void print(String[] a)
   {
      for (String e : a)
      {
         System.out.print(e + " ");
      }
      System.out.println();
   }
   . . .
}
```

20

Métodos Genéricos (3)

- Para transformar o método num método genérico:
  - Substituir `String` por um parâmetro de tipo, por exemplo `E`, para definir o tipo do elemento
  - Incluir os parâmetros de tipo entre os modificadores de acesso do método e o tipo de retorno do método

```
public static <E> void print(E[] a)
{
   for (E e : a)
   {
      System.out.print(e + " ");
   }
   System.out.println();
}
```

21

Métodos Genéricos (4)

- Na invocação de um método genérico, não é necessário instanciar os parâmetros de tipo; exemplo:

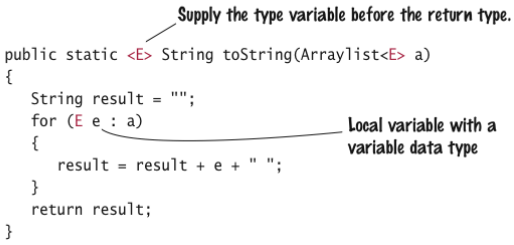
```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

  - O compilador deduz que `E` é `Rectangle`
- É possível definir métodos genéricos que não sejam estáticos, assim como construtores
- É possível ter métodos genéricos dentro de classes genéricas
- Não é possível substituir as variáveis de tipo por tipos primitivos
  - Exemplo: não é possível usar o método genérico `print` para imprimir um *array* do tipo `int[]`

22

Declaração de um Método Genérico

Syntax *modifiers* <*TypeVariable*$_1$, *TypeVariable*$_2$, . . .> *returnType methodName*(*parameters*)
{
body
}




```
public static <E> String toString(Arraylist<E> a)
{
   String result = "";
   for (E e : a)
   {
      result = result + e + " ";
   }
   return result;
}
```


Restrições Aplicadas aos Parâmetros de Tipo

- Os parâmetros de tipo de um tipo genérico podem ser limitados
- O limite de um parâmetro de tipo restringe os tipos que podem ser usados como argumento

```
class Pair<A extends Animal,
           B extends Integer> {
  …
}
```

```
class Pair<A extends Comparable<A> & Cloneable,
           B extends Comparable<B> & Cloneable>
implements Comparable<Pair<A,B>>, Cloneable {
 …
}
```

Restrições Aplicadas ao Parâmetros de Tipo

- Os parâmetros de tipo de um tipo genérico podem ser limitados
- O limite de um parâmetro de tipo restringe os tipos que podem ser usados como argumento

```
public static <E extends Comparable<E>>
   E min(ArrayList<E> objects)
{
   E smallest = objects.get(0);
   for (int i = 1; i < objects.size(); i++)
   {
      E obj = objects.get(i);
      if (obj.compareTo(smallest) < 0)
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

25

Restrições Aplicadas ao Parâmetros de Tipo

- Por vezes, é necessário definir dois ou mais limites; exemplo:

```
<E extends Comparable<E> & Measurable>
```

  - `extends`, quando aplicado a parâmetros de tipo assume o significado "extends ou implements"
- Os limites podem ser classes ou interfaces
- Um parâmetro de tipo pode ser instanciado com uma classe ou interface
- Uma lista de limites é constituída por **uma** classe e/ou **várias** interfaces

```
<TypeParameter extends Class &
   Interface1 & ... & InterfaceN>
```

...

26

Genéricos e Herança

- Se `SavingsAccount` é uma subclasse de `BankAccount`, será `ArrayList<SavingsAccount>` uma subclasse de `ArrayList<BankAccount>`? Não há qualquer relação entre `ArrayList<SavingsAccount>` e `ArrayList<BankAccount>`

```
ArrayList<SavingsAccount> savingsAccounts = new ArrayList<SavingsAccount>();
ArrayList<BankAccount> bankAccounts = savingsAccounts;
// Não é legal, mas vamos supor que seria
BankAccount harrysChecking = new CheckingAccount();
// CheckingAccount é outra subclasse de BankAccount
bankAccounts.add(harrysChecking);//OK:é possível adic. um objeto BankAccount
```

- Mas `bankAccounts` e `savingsAccounts` referem-se ao mesmo *array list* – Se a atribuição fosse legal, seria possível inserir uma `CheckingAccount` num `ArrayList<SavingsAccount>`

Restrições Aplicadas ao Parâmetros de Tipo

- Tal como na declaração de classes, é possível restringir o tipo de parâmetros num método. Um método que recebe uma lista de `Vehicles` e retorna o mais rápido da lista pode ter o seguinte tipo:

```
<T extends Vehicle> T getFastest(List<T> list) { }
```

- É possível passar como argumento uma lista de qualquer tipo de veículos: `List<Car>, List<Motorcycle>, List<Vehicle>`
- No caso seguinte o comportamento seria diferente:

```
Vehicle getFastest2(List<Vehicle> list) { }
```

- O argumento do método `getFastest2` tem que ser exatamente `List<Vehicle>`, e não `List<Car>`, porque `List<Car>` não é um subtipo de `List<Vehicle>`

Tipos *Wildcard*

- ? representa um qualquer tipo

```
public void printPair(Pair<?,?> pair) {
   System.out.println("(" + pair.getFirst() +
   "," + pair.getSecond() + ")");
}

Pair<?,?> limit = new Pair<String,Long>("maximum",1024L);

printPair(limit);
```

Tipos *Wildcard*

Designação	Sintaxe	Significado
Wildcard com limite inferior	`? extends B`	Qualquer subtipo de `B`
Wildcard com limite superior	`? super B`	Qualquer supertipo de `B`
Wildcard não limitado	`?`	Qualquer tipo

30

Métodos Genéricos vs. *Wildcards*

- Podemos implementar com métodos genéricos o que se implementa com *wildcards*

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

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

Métodos Genéricos vs. *Wildcards*

- Consideremos a seguinte declaração:

```
<T extends Vehicle> int totalFuel(List<T> list) { }
```

- O parâmetro T aparece apenas uma vez na assinatura do método, num argumento. Consideremos também que o corpo do método não usa o nome T. Neste caso é possível utilizar uma sintaxe alternativa, designada *wildcard* (?):

```
int totalFuel(List<? extends Vehicle> list) { }
```

- As duas assinaturas anteriores para `totalFuel` são equivalentes. O significado de `<? extends Vehicle>` é: o parâmetro de tipo pode ser instanciado com qualquer tipo desde que seja uma subclasse de `Vehicle`.

Métodos Genéricos vs. *Wildcards*

- Questão:
  - Quando usar métodos genéricos?
  - Quando usar *wildcards*?
- Se não existir dependência entre parâmetros e/ou tipos de retorno, devemos optar pela utilização de *wildcards*

Type Erasure

- A máquina virtual apaga os parâmetros de tipo, substituindo-os pelos seus limites `Object`s
- Por exemplo, a classe genérica `Pair<T, S>` é transformada na seguinte classe:

```
public class Pair
{
   private Object first;
   private Object second;

   public Pair(Object firstElement, Object secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public Object getFirst() { return first; }
   public Object getSecond() { return second; }
}
```

34

Type Erasure (2)

- O mesmo processo é aplicado aos métodos genéricos:

```
public static Measurable min(Measurable [] objects)
{
   Measurable smallest = objects[0];
   for (int i = 1; i < objects.length; i++)
   {
      Measurable obj = objects[i];
      if (obj.getMeasure() < smallest.getMeasure())
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

35

Type Erasure (3)

- O conhecimento acerca do mecanismo de *Type Erasure* permite perceber as limitações dos genéricos
- Por exemplo, não é possível instanciar objetos de um tipo genérico
- O exemplo seguinte, que tenta preencher um *array* com objetos por defeito, não funcionará

```
public static <E> void fillWithDefaults(E[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new E(); // ERRO
}
```

- A aplicação do *Type Erasure* permite perceber porquê:

```
public static void fillWithDefaults(Object[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new Object(); // Sem utilidade
}
```

36

Type Erasure (4)

- Não é possível construir um *array* de tipo genérico:

```
public class Stack<E>
{
   private E[] elements;
   . . .
   public Stack()
   {
      elements = new E[MAX_SIZE]; // Erro
   }
}
```

- Isto porque a expressão de construção do *array* `new E[]` será alterada para `new Object[]`

37

Type Erasure (5)

- A alternativa passa por usar uma coleção, como um *array list*:

```
public class Stack<E>
{
   private ArrayList<E> elements;
   . . .
   public Stack()
   {
      elements = new ArrayList<E>(); // Correto
   }
}
```

38

Type Erasure (6)

- Outra alternativa passa por usar um array de `Object`s e efetuar uma conversão quando se faz uma operação sobre o *array*:

```
public class Stack<E> {
  private Object[] elements;
  private int currentSize;
  ...
  public Stack() {
    elements = new Object[MAX_SIZE]; // Correto
  }
  ...
  public E pop() {
    size--;
    return (E) elements[currentSize];
  }
}
```

O *cast* (`E`) gera um aviso uma vez que a conversão não pode ser verificada durante a compilação

39

Sumário: Classes Genéricas e Parâmetros de Tipo

- Em Java, a programação genérica pode ser obtida por aplicação de herança ou de parâmetros de tipo
- Uma classe genérica tem um ou mais parâmetros de tipo
- Parâmetros de tipo podem ser instanciados por classes ou por interfaces
- Os parâmetros de tipo tornam o código genérico mais robusto e de mais fácil leitura

40

Sumário: Classes Genéricas e Interfaces

- As variáveis de tipo de uma classe genérica aparecem após o nome da classe e são rodeadas por < >
- Podemos usar parâmetros de tipo para tipos de variáveis de instância genéricas, parâmetros de métodos e valores de retorno de métodos

41

Sumário: Métodos Genéricos

- Um método genérico é um método com um parâmetro de tipo
- Os parâmetros de tipo de um método genérico são colocados entre os modificadores do método e o tipo de retorno do método
- Na invocação de um método genérico, não é necessário instanciar os parâmetros de tipo

42

Sumário: Restrições nos Parâmetros de Tipo

- Os parâmetros de tipo podem ser restringidos através da definição de limites

43

Sumário: *Type Erasure*

- A máquina virtual "apaga" os parâmetros de tipo, substituindo-os pelos seus limites (se definidos) ou por `Object`s.
- Não é possível construir objetos ou *arrays* de um tipo genérico

44