

Java

Tipos Genéricos

Programação III
José Luis Oliveira; Carlos Costa

Motivações

- Quando os programas aumentam de dimensão é possível começarmos a ter métodos que executam operações similares com diferentes tipos de dados
- O que há de “errado” com o seguinte bloco de código?

```
String add (String a, String b) { return a + b; }  
int      add (int a, int b)      { return a + b; }  
double   add (double a, double b) { return a + b; }
```

```
public static void main(String[] args) {  
    System.out.println(add(Gene, ricos));  
    System.out.println(add(2, 5));  
    System.out.println(add(3.2, 5.6));  
}
```

- Nada?

O código está correcto mas definimos estaticamente operações repetidas.

Motivações

```
class Node {  
    Object value;  
    Node next;  
  
    Node( Object o ) {  
        value = o;  
        next = null;  
    }  
}
```

```
Car c = new Car( ... );  
Node n = new Node( c );
```

...

```
Vehicle v = ( Vehicle ) n.value;
```

OK

```
Movie m = new UniversalMovie( "ET" );  
Node n = new Node( m );
```

...

```
Vehicle v = ( Vehicle ) n.value;
```

Run-Time Error

Down-Cast e Runtime Error

O que são Genéricos?

- Uma forma de Polimorfismo Paramétrico
- Estruturas e Algoritmos são implementados uma única vez, mas utilizados com diferentes tipos de dados
- Dizemos que:
 - Os **Tipos de dados** também são um **Parâmetro**
- Genéricos aplicados a:
 - Métodos
 - Classes
 - Interfaces
- Introduzidos em JAVA na versão 5
- Em C++, designam-se por *templates*

Classes Genéricas

Exemplo: Conjunto Genérico

Declaração

```
class Conjunto<T> {  
    T[] c;;  
    //...  
}
```

Nota Importante:

O tipo parametrizado (T), não pode ser instanciado com um tipo primitivo.

Utilização

```
Conjunto<Pessoa> c1 = new Conjunto<Pessoa>(..);  
Conjunto<Jogador> c2 = new Conjunto<Jogador>(..);  
Conjunto<Integer> c3 = new Conjunto<Integer>(..);
```

Classes Genéricas

Exemplo: Uma Pilha Genérica

Sem Genéricos

```
class Stack {  
    void push(Object o)  
    { ... }  
    Object pop() { ... }  
    ...}  
  
String s = "Hello";  
Stack st = new Stack();  
...  
st.push(s);  
...  
s = (String) st.pop();
```



E se estivermos errados?



Runtime Error

Utilizando Genéricos

```
class Stack<T> {  
    void push(T a) { ... }  
    T pop() { ... }  
    ...}  
  
String s = "Hello";  
Stack<String> st =  
    new Stack<String>();  
  
st.push(s);  
...  
s = st.pop();
```

OK

Genéricos

```
class Node< T > {  
    T value;  
    Node< T > next;  
  
    Node( T t ) {  
        value = t;  
        next = null;  
    }  
}
```

```
Car c = new Car( ... );  
Node<Car> node = new Node<Car>( c );  
  
...  
  
Car c2 = node.value;
```

OK

```
Movie m = new ActionMovie( ... );  
  
...  
  
Node<Car> node = new Node<Car>( m );
```

Compiler Error

**Detecção de Erros em
Compilação**

Genéricos - Processamento

- Etapas de processamento de Genéricos em JAVA
 - **Check**: Verificação da correcta utilização de tipos
 - **Erase**: Remove toda a informação “generic type”
 - **Compile**: Geração do byte-code
- Este processo denomina-se como:
 - **Type Erasure**

Genéricos - Check

- Declaração

```
class Foo <T> {  
    void method(T arg) ;  
};
```

- Utilização

```
Foo<Bar> fb = new Foo<Bar>;  
fb.method(aBar) ;                // OK  
fb.method(not_a_Bar) ;           // Compile Error
```

- O Compilador garante que o *arg* é do mesmo tipo <Bar>

Genéricos - Erase

- Cada parâmetro definido como genérico é substituído por um `java.lang.Object`
- Os *casts* “Object -> Tipo Concreto” são automaticamente introduzidos pelo compilador.

```
class choice <T>  
{ public T best ( T a , T b ) {..} }
```

É substituído por:

```
class choice  
{ public Object best ( Object a, Object b ) {..} }
```

Genéricos em Classes

```
public class Stack_Generic<T> {
    private class Node<E> {
        E val;
        Node<E> next;
        Node(E v, Node<E> n) {
            val = v;
            next = n;
        }
    }

    private Node<T> top = null;

    public boolean empty( ) {
        return top == null;
    }

    public T pop( ) {
        T result = top.val;
        top = top.next;
        return result;
    }

    public void push(T v) {
        top = new Node<T>(v, top);
    }
}
```

```
public class TestStack {

    public static Figura randFig(int max) {
        Switch ((int) (Math.random() * (max))) {
            default:
            case 0:
                return new Circulo (1,3, 1.2);
            case 1:
                return new Quadrado(3,4, 2);
            case 2:
                return new Rectangulo(1,1, 5,6);
        }
    }

    public static void main(String[] args) {
        Stack_Generic<Figura> stk =
            new Stack_Generic<Figura>();

        for (int i=0; i<10; i++)
            stk.push(randFig(3));

        for (int i=0; i<10; i++)
            System.out.println(stk.pop());
    }
}
```

Genéricos em Métodos

- Declaração

```
public <T> T add (T a, T b) {  
    return a + b;  
}
```

Compilará?

- Utilização Pretendida

```
int c = add(8, 3);  
double d = add (4.5, 6.9);  
String str = add("gene" , "ricos");
```

Podemos aplicar o
operador + ao tipo Object?

Não.

Não.

Genéricos - Métodos

```
public <T extends Number> T add (T a, T b) {  
    return a + b;  
}
```

- Impusemos um limite superior ao tipo passado para o método genérico
- Neste caso estamos a dizer que o tipo passado será um subtipo de `java.lang.Number`

- **E Agora?**

Podemos aplicar o
operador + ?

E o tipo de Retorno.
Correcto?

Não.

Genéricos - Métodos

```
public static <T extends Number> Double add (T a, T b) {  
    return a.doubleValue() + b.doubleValue();  
}
```

```
public static <T> String sumToString (List<T> aList){  
    StringBuilder aBuffer = new StringBuilder();  
    for (T x : aList)  
        aBuffer.append (x.toString());  
    return aBuffer.toString();  
}
```

add: 5.2

sumToString: 223344

```
public static void main(String[] args) {  
    System.out.println("add: " + add (2.2, 3));  
    List<Integer> myList = new ArrayList<Integer>();  
    myList.add(22); myList.add(33); myList.add(44);  
    System.out.println("sumToString : " +  
        sumToString(myList));  
}
```

Genéricos e Subtipos

- Já sabemos que:
 - Uma referência do tipo T pode apontar para uma instância da classe T ou para uma instância de um subtipo de T.
- Como se conjuga **Polimorfismo** com **Tipos Genéricos**?

```
public static void main(String[] args) {  
    LinkedList<Figura> list = new LinkedList<Figura>();  
    LinkedList<Quadrado> list2 = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(3,4, 2);
```

```
    list.add(q);  
    list2.add(q);
```

OK

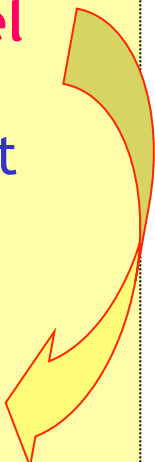
```
    LinkedList<Quadrado> list3 = list;  
    LinkedList<Figura> list4 = list2;
```

Compile-Time Error

Porquê ?

Genéricos e Subtipos

```
LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
LinkedList<Figura> list2 = list;           // Imaginando que é possível  
  
Figura f = new Figura(..);                // Supondo que não é abstract  
  
list2.add(f);  
  
Quadrado q = list.get(0);                  // Runtime ERROR!!!!
```



Uma Figura não é um Quadrado

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

Genéricos e Subtipos

```
public static void main( String[ ] args ) {  
    LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(3,4, 2);  
    list.add(q);  
    list.add(q);  
  
    print(list);  
}
```

Compile-Time Error

```
public static void print( LinkedList<Figura> listOfFig ) {  
    Iterator it = listOfFig.iterator();  
    while( it.hasNext())  
        System.out.println( it.next() );  
}
```

Questão: Como permitir que, tendo um argumento tipo `LinkedList` de `Figura`, se possa aceitar uma `LinkedList` de `Figura` mas também um dos seus subtipos?

Genéricos - Wildcards


- **Resposta:** “Utilizando Wildcards”
- Bounded wildcards
 - `< ? extends class-T >`
subclass de class-T, incluindo class-T
 - `< ? extends class-T & interface-E >`
subclass de class-T e int-E, incl. class-T e int-E
 - `< ? super class-T >`
superclass de class-T, incluindo class-T
- Unbounded wildcards
 - `< ? extends Object >`
subclass de Object, i.e. qualquer tipo
 - `< ? >`
Semelhante a `< ? extends Object >`

Genéricos e Subtipos

```
public static void main( String[ ] args ) {  
    LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(3,4, 2);  
    list.add(q);  
    list.add(q);  
  
    print(list);  
}  
  
public static void print( LinkedList<? extends Figura> listOfFig ) {  
    Iterator it = listOfFig.iterator();  
    while( it.hasNext())  
        System.out.println( it.next() );  
}
```

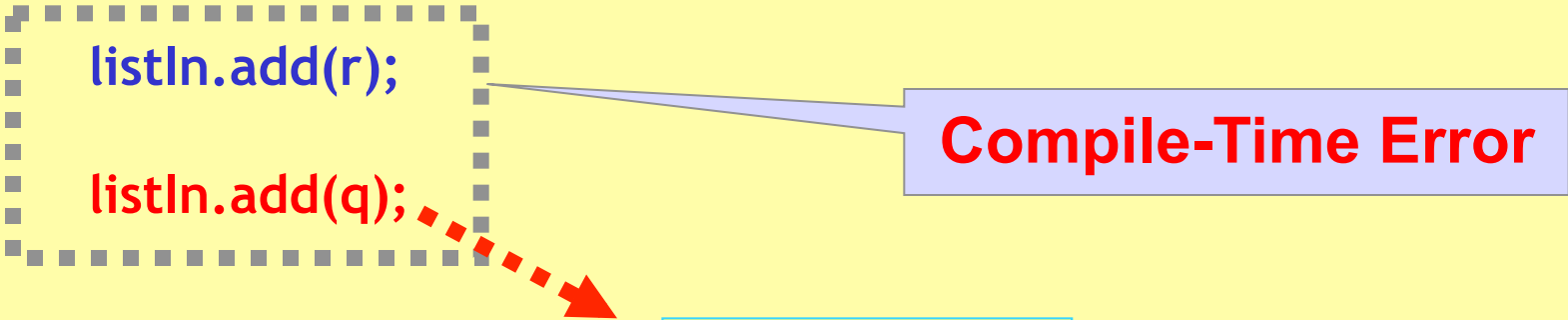
Genéricos - Wildcards

```
public static void main( String[ ] args ) {  
    LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(1,2, 5.5);  
    list.add(q);  
    addMore(list);  
}
```



OK

```
public static void addMore( LinkedList<? extends Figura> listIn ) {  
    Rectangulo r = new Rectangulo(3,4, 2, 3);  
    Quadrado q = new Quadrado(1,2, 5.5);  
  
    listIn.add(r);  
    listIn.add(q);  
}
```



Compile-Time Error

Porquê ?

Genéricos - Wildcards

- É possível ainda especificar um parâmetro tipo genérico que *extend/implement* uma *class/interface*
- Para ambos os casos é utilizada a keyword **extend**

```
public class Xpto <T extends Serializable &  
    Comparable<T>> {..}
```

- Tipo Genérico deve implementar Serializable e Comparable

Genéricos

- **Excepções**: Também podemos passar um parâmetro genérico que é um subtipo de **Exception**

```
public class Employer <T extends Employee,  
                        X extends Exception>{  
    public double calculatePay (T employee) throws X {};  
}
```

- **Arrays**: Não é possível criar um array de tipos genéricos

```
T[] array = new T[MAX];
```

Compile-Time Error

```
T[] newArray = (T[]) new Object[MAX];
```

Exemplos

```
/**
 * This version introduces a generic method.
 */
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U> void inspect(U u) {
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}
```

T: java.lang.Integer

U: java.lang.String

Exemplos

```
public class Box<T> {  
    ...  
    public static <U> void fillBoxes(U u, List<Box<U>> boxes) {  
        for (Box<U> box : boxes) {  
            box.add(u);  
        }  
    }  
    ...  
}
```

```
Crayon red = ...;  
List<Box<Crayon>> crayonBoxes = ...;
```

```
Box.fillBoxes(red, crayonBoxes);
```


Java 7 - The Diamond

- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set (<>) as long as the compiler can determine the type arguments from the context.
 - This pair of angle brackets, <>, is informally called the *diamond*.
- Exemplo:

```
Box<Integer> integerBox = new Box<>();
```
- For example, consider the following variable declaration:

```
Map<String, List<String>> myMap =  
    new HashMap<String, List<String>>();
```
- In Java SE 7 and later, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>();
```

Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters.
 - This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason:
 - Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
- The most commonly used type parameter names are:
 - E - Element (used extensively by the Java Collections Framework)
 - K - Key
 - N - Number
 - T - Type
 - V - Value

Tipos genéricos em Java - Resumo

- Conseguimos:
 - eliminar necessidade de coerção explícita (cast)
 - aumentar robustez: verificação estática de tipo
 - aumentar legibilidade
- Não há múltiplas versões do código
 - declaração é compilada para todos os tipos
 - parâmetros formais possuem tipo genérico
 - na invocação, os tipos dos parâmetros actuais são substituídos pelos tipos dos formais

C++ versus Java

C++

- Templates - Mecanismo Estático
- Compilador é responsável por todo o trabalho necessário à multi instanciação de código genérico.
- Compilador cria uma cópia separada do código de cada instância.

JAVA

- Uma declaração de tipo genérico é compilada e gera uma classe Java comum, como outra qualquer.
- Todas as instâncias do código genérico partilham o mesmo código em *run-time*.
- Um parâmetro tipo genérico (T em Java), tem comportamento idêntico a um `java.lang.Object` sem necessidade de utilizar *cast*.

Sumário

- Tipos Genéricos
- Motivações
- Processamento
- Wildcards e Sub-tipos
- Classe e Métodos Genéricos