

# Técnicas de Programação Avançada

*TCC-00175*

*Profs.: Anselmo Montenegro*

*[www.ic.uff.br/~anselmo](http://www.ic.uff.br/~anselmo)*

Conteúdo: Tipos Genéricos



Baseado em <http://docs.oracle.com/javase/tutorial/java/generics/>



Introdução

Tipos Genéricos do Java

Vantagens do uso de tipos genéricos

Declaração de Tipos Genéricos

Exemplo de Tipos Genéricos

Convenções sobre nomes de tipos de parâmetros

Invocação de Tipos Genéricos

Instanciação de Tipos Genéricos

Métodos Genéricos



## Invocação de Métodos Genéricos



É uma melhoria no sistema de tipos que foi introduzida na J2SE 5.0 que permite um **método ou tipo operar sobre objetos de vários tipos diferentes**

Permite **segurança de tipos em tempo de compilação**

Adiciona segurança de tipos em tempo de compilação ao Framework de Coleções **eliminando a necessidade *de casting***



Tipos genéricos permitem que (classes e interfaces) sejam usadas como parâmetros na definição de classes, interfaces e métodos.

De forma análoga aos parâmetros formais usados nas declarações de métodos, os parâmetros de tipo proveem formas de utilizar o mesmo código com diferentes entradas

A entrada para parâmetros formais são valores enquanto que para os parâmetros de tipo são tipos de dados



Checagem de tipos mais forte em tempo de compilação

Eliminação de conversão de tipos (casts)

Permite a programação de algoritmos genéricos seguros em relação a tipagem e legíveis



Um **tipo genérico** é uma **classe ou interface parametrizada sobre tipos**

Uma classe genérica é definida da seguinte forma:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

A seção dos parâmetros de tipo é delimitada pelos símbolos (<>) e vem após o nome da classe.

T1, T2, ..., and Tn podem ser quaisquer **tipos não primitivos** incluindo classes, interfaces, arrays e etc.



### Classe não genérica

```
public class Box {  
  
    private Object object;  
  
    public void set(Object object) {  
        this.object = object;  
    }  
  
    public Object get() {  
        return object;  
    }  
  
}
```

### Versão genérica da mesma classe

```
/** * Generic version of the Box class.  
 * @param <T> the type of the value being  
boxed  
 */  
public class Box<T> { // T stands for "Type"  
  
    private T t;  
  
    public void set(T t) {  
        this.t = t;  
    }  
  
    public T get() {  
        return t;  
    }  
  
}
```





Por convenção, **parâmetros de tipos são nomeados por letras maiúsculas únicas**

O uso desta convenção é para distinguir parâmetros de tipo de nomes de variáveis, classes e interfaces



Nomes mais comuns:

E - Element (usado extensivamente no *Java Collections Framework*)

K - Key

N - Number

T - Type

V - Value

S,U,V etc. – segundo, terceiro e quanto tipos de uma lista de tipos de parâmetros



Para referenciar e usar uma classe genérica no código é necessário fazer uma **invocação de tipo genérico**

A invocação de tipo genérico **substitui um parâmetro de tipo, por exemplo, T com algum valor concreto**

Exemplo: `Box<Integer> integerBox;`

Uma invocação de um tipo genérico é análoga a uma invocação de um método; ao invés de passarmos um argumento para um método, passamos um **argumento de tipo**



A invocação é apenas uma declaração e não cria um novo objeto

Apenas indica que a variável irá conter uma referência para tal tipo declarado que no caso é um **tipo parametrizado**

Para instanciar um classe, usamos a palavra **new**, e colocamos o **argumento de tipo no lugar do parâmetro de tipo**

```
Box<Integer> integerBox = new Box<Integer>();
```



No Java SE 7 e versões posteriores é possível substituir os tipos de argumentos requeridos na invocação de um construtor de uma classe genérica usando um conjunto vazio (<>)

Isto pode ser feito desde que o compilador seja capaz de inferir o tipo pelo contexto

<> é informalmente denominado por “diamante” (*diamond*).

Exemplo: `Box<Integer> integerBox = new Box<>();`



**Métodos genéricos** são métodos que definem e utilizam seus próprios parâmetros de tipo.

O **escopo do parâmetro de tipo é limitado ao método** em que é declarado.

São permitidos métodos estáticos, não-estáticos e construtores de classes genéricos.



A sintaxe de um método genérico inclui o parâmetro de tipo englobado por `<>` e deve aparecer antes do tipo de retorno do método

Em métodos estáticos, a seção de parâmetros de tipo deve vir antes do retorno do método



```
public class Util {  
    // Generic static method  
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) { return  
        p1.getKey().equals(p2.getKey()) && p1.getValue().equals(p2.getValue());  
    }  
}
```

```
public class Pair<K, V> {  
    private K key; private V value;  
    // Generic constructor  
    public Pair(K key, V value) {  
        this.key = key; this.value = value;  
    }  
  
    // Generic methods  
    public void setKey(K key) { this.key = key; }  
    public void setValue(V value) { this.value = value; }  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```





## Sintaxe para a invocação de método genérico

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.<Integer, String>compare(p1, p2);
```

## Usando inferência de tipos:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.compare(p1, p2);
```

O tipo dos argumentos em um parâmetro de tipo pode ser restringido quando necessário.

Exemplo: um método que opera sobre números somente deve aceitar instâncias de `Number` ou de suas subclasses.

Para declarar um parâmetro de tipo restrito (*bounded type parameter*) basta listar o nome do parâmetro de tipo seguido da palavra *extends* seguida da restrição superior (*upper bound*)

```
public class Box<T> {
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
    public <U extends Number> 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.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

Tipos restritos permitem a invocação de métodos dentro do limite da restrição

Exemplo:

```
public class NaturalNumber<T extends Integer> {  
    private T n;  
  
    public NaturalNumber(T n) { this.n = n; }  
    public boolean isEven() {  
        return n.intValue() % 2 == 0;  
    } // ...  
}
```



Parâmetros de tipo podem ter restrições múltiplas.

Exemplo: `<T extends B1 & B2 & B3>`

Uma variável de tipo com restrições múltiplas e subtipo de todos os tipos listados na restrição.

Classes devem ser especificadas primeiro na listagem declarativa.



```
Class A { /* ... */ }  
interface B { /* ... */ }  
interface C { /* ... */ }  
  
class D <T extends A & B & C> { /* ... */ }  
  
class D <T extends B & A & C> { /* ... */ } // compile-time error
```



Exercício: Escreva um método genérico que conte o número de elementos em um array `T[]` que sejam maiores que um elemento `elem`



Exercício: Escreva um método genérico que conte o número de elementos em um array `T[]` que sejam maiores que um elemento `elem`

```
public static <T> int countGreaterThan(T[] anArray, T elem) { int count = 0;
    for (T e : anArray)
        if (e > elem)
            ++count;
    return count;
}
```





Exercício: Escreva um método genérico que conte o número de elementos em um array T[] que sejam maiores que um elemento elem

```
public static <T> int countGreaterThan(T[] anArray, T elem) { int count = 0;
    for (T e : anArray)
        if (e > elem) // compiler error
            ++count;
    return count;
}
```



```
public interface Comparable<T> {
    public int compareTo(T o);
}

public static <T extends Comparable<T>> int countGreaterThan( T[] anArray, T
elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem))
            ++count;
    return count;
}
```

Tipos genéricos permitem a **abstração** sobre tipos

O exemplos mais comum de uso de genéricos em Java é a hierarquia de Coleções

## Uso típico de Containers

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

Problema: o compilador somente pode garantir que um Object seja retornado pelo método next() do iterator

Para assegurar que a **atribuição seja segura** é necessário uso *casting*

Como evitar o uso do *casting*?

Solução: **usar os tipos genéricos do Java**

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



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

A declaração da variável `myIntList` especifica que ela não é simplesmente do tipo `List`, mas do tipo `List of Integer`, representada como `List<Integer>`

`List<E>` é uma interface genérica que toma um **parâmetro de tipo** (*type parameter*) `E`

O **argumento real** do parâmetro de tipo deve ser especificado na criação de uma instância da lista (`Integer`)

Com a introdução de tipos genéricos não é mais necessário o uso do *casting*

Mas as diferenças não param ai

Com **tipos genéricos** o compilador pode checar a corretude do programa em **tempo de compilação**



A declaração de `myIntList` como `List<Integer>` é uma afirmação que será sempre verdade

Por outro lado, o *casting* apenas nos diz o que o programador acha que é verdade em um ponto do código

Isso favorece a **robustez e legibilidade** do código

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



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

## Exemplo de definição de um tipo genérico simples

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

Os elementos entre *brackets* são os **parâmetros de tipos formais**

Na invocação de um tipo parametrizado, todas as ocorrências do parâmetro formal são substituídas pelo argumento real (Integer)

Entretanto, `List<Integer>` não corresponde ao código abaixo

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



Isto significa que **não há nenhum tipo de expansão**

Uma declaração de tipo genérico é **compilada uma única vez** e gera somente um único arquivo de classe como uma declaração de classe ou interface normal

Tipos genéricos do Java são diferentes de Templates em C++



*Parâmetros de tipo* são análogos aos parâmetros comuns

Parâmetros formais de métodos e construtores descrevem os tipos de valores sobre os quais eles operam

Similarmente uma declaração genérica tem parâmetros de tipos formais



Quando um método é invocado os **parâmetros formais** são substituídos pelos **argumentos reais**

Similarmente na invocação de um genérico o **parâmetro de tipo formal** é substituído pelo **argumento de tipo real**



Deve-se tomar muito cuidado ao **considerar subtipos e genéricos** em uma mesma construção

Considere o exemplo abaixo

```
List<String> ls = new ArrayList<String>(); // 1
List<Object> lo = ls; // 2
lo.add(new Object()); // 3
String s = ls.get(0); // 4: Attempts to assign an
// Object to a String!
```





Questão: uma `List<String>` é um subtipo de uma `List<Object>`?

```
List<String> ls = new ArrayList<String>(); // 1
List<Object> lo = ls; // 2
lo.add(new Object()); // 3
String s = ls.get(0); // 4: Attempts to assign an
// Object to a String!
```



Se for verdade então:

É possível atribuir uma referência de **ls** para **lo** (2)

É possível adicionar um Object a um objeto que é uma lista de Strings (3)

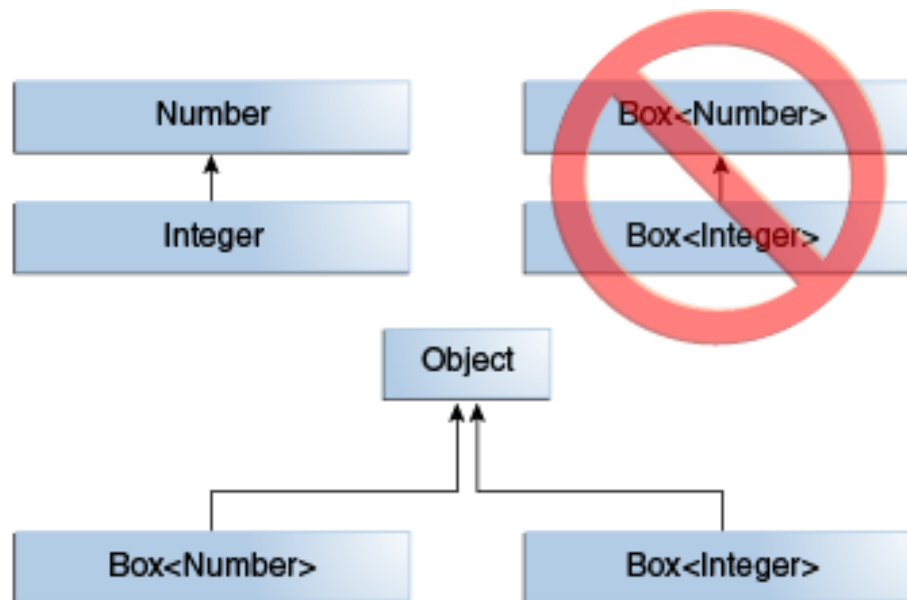
Mas não será possível atribuir um objeto de **List<String>** para uma variável de tipo String (4)

```
List<String> ls = new ArrayList<String>(); // 1
List<Object> lo = ls; // 2
lo.add(new Object()); // 3
String s = ls.get(0); // 4: Attempts to assign an
// Object to a String!
```



Conclusão:

Sejam  $A$  e  $B$  classes (ou interfaces) ,  $B \subseteq A$  e  $G\langle \rangle$   
uma classe genérica, então não é verdade que  
 $G\langle B \rangle \subseteq G\langle A \rangle$





Considere o problema de imprimir todos os elementos de uma coleção

Solução sem usar genéricos

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



## Solução com genéricos

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

Problema????



## Solução com genéricos

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

Problema: `Collection<Object>` não é um supertipo de todas as coleções possíveis



Qual o supertipo de todas as coleções possíveis?

Resposta: `Collection<?>`

`Collection<?>` é denominada coleção de desconhecidos

`?` é um símbolo denominado *wildcard*





Um wildcard (?) representa um tipo desconhecido.

Wildcards podem ser usados nas seguintes situações:

- Como o tipo de um parâmetro;
- Como o tipo de um campo;
- Como o tipo de uma variável local;
- Como um tipo de retorno de um método (prática não recomendada)



Wildcards jamais devem ser usados como argumentos de tipos em:

- Invocações de métodos genéricos;
- Criação de classes genéricas;
- Um supertipo;



## Solução para imprimir uma coleção usando genéricos

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

É importante obter objetos Object da coleção

É seguro porque todo objeto em uma coleção qualquer é um Object



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

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

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



Não é seguro entretanto atribuir objetos a uma coleção de desconhecidos

```
Collection<?> c = new ArrayList<String>(); c.add(new Object()); // Compile time error
```

Motivo: óbvio. Se não sabemos o tipo da coleção como podemos adicionar um Object?



É possível entretanto obter um Object de um Collection<?> ou um outro tipo genérico

```
Collection<?> c = new ArrayList<String>();  
Object o = c.get();
```



```
public abstract class Shape {
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { ... }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) { ... }
}

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



```
public abstract class Shape {
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { ... }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) { ... }
}

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





```
public void addRectangle(List<? extends Shape> shapes) {  
    shapes.add(0, new Rectangle()); // Compile-time error!  
}
```



```
static void fromArrayToCollection(Object[] a,  
Collection<?> c) {  
    for (Object o : a)  
        c.add(o); // compile-time error  
}
```

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



```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co); // T inferred to be Object

String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();

fromArrayToCollection(sa, cs); // T inferred to be String
fromArrayToCollection(sa, co); // T inferred to be 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 inferred to be Number
fromArrayToCollection(fa, cn); // T inferred to be Number
fromArrayToCollection(na, cn); // T inferred to be Number
fromArrayToCollection(na, co); // T inferred to be Object
fromArrayToCollection(na, cs); // compile-time error
```



```
interface Sink<T> {
    flush(T t);
}

public static <T> T writeAll(Collection<T> coll, Sink<T> snk)
{
    T last;
    for (T t : coll) {
        last = t;
        snk.flush(last);
    }
    return last;
}

Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s); // Illegal call.
```



```
interface Sink<T> {
    flush(T t);
}

public static <T> T writeAll(Collection<? extends T>,
Sink<T>) {
    T last;
    for (T t : coll) {
        last = t;
        snk.flush(last);
    }
    return last;
}

Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s); // Call is OK, but wrong return
type.
```



# Tipos Genéricos

## *Mais exemplos de uso de genéricos*

```
interface Sink<T> {
    flush(T t);
}

public static <T> T writeAll(Collection<T>, Sink<? super T>)
{
    T last;
    for (T t : coll) {
        last = t;
        snk.flush(last);
    }
    return last;
}

Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s); // OK!!!
```

1. Não é possível instanciar tipos genéricos com tipos primitivos
2. Não é possível criar instâncias de tipos parametrizados
3. Não é possível declarar campos estáticos cujos tipos sejam parametrizados
4. Não é possível usar *casts* ou *instanceof* com tipos parametrizados
5. Não é possível criar arrays de tipos parametrizados
6. Não é possível criar, usar `Catch` ou usar `Throw` em objetos de tipos parametrizados
7. Não é possível sobrecarregar um método cujo parâmetro formal leva ao mesmo tipo cru (Raw Type) durante o processo de apagamento de tipo (Type Erasure)



# Tipos Genéricos

## Exercícios

(obtidos da página da oracle  
<http://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html>)

1 - Write a generic method to count the number of elements in a collection that have a specific property (for example, odd integers, prime numbers, palindromes).

2- Will the following class compile? If not, why?

```
public final class Algorithm {  
    public static T max(T x, T y) { return x > y ? x : y; }  
}
```

3 - Write a generic method to exchange the positions of two different elements in an array.

8 - Write a generic method to find the maximal element in the range [begin, end) of a list (difícil).





# Tipos Genéricos

## Exercícios

(obtidos da página da oracle  
<http://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html>)

10 - Given the following classes:

```
class Shape { /* ... */ }  
class Circle extends Shape { /* ... */ }  
class Rectangle extends Shape { /* ... */ }  
class Node<T> { /* ... */ }
```

Will the following code compile? If not, why?

```
Node<Circle> nc = new Node<>();  
Node<Shape> ns = nc;
```



# Tipos Genéricos

## Exercícios

(obtidos da página da oracle  
<http://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html>)

11 - Consider this class:

```
class Node<T> implements Comparable<T> {  
    public int compareTo(T obj) { /* ... */ }  
    // ...  
}
```

Will the following code compile? If not, why?

```
Node<String> node = new Node<>();  
Comparable<String> comp = node;
```



- 1- Escreva uma implementação em Java para uma árvore binária com as operações de inserção e busca.
- 2- Estenda o código do exercício de mineração de documentos para uso de documentos genéricos.