SCC0504 - Programação Orientada a Objetos

Generics

Prof.: Leonardo Tórtoro Pereira

leonardop@usp.br

Generics

Generics[1]

- ➔ *Generics* permitem tipos (classes e interfaces) a serem parâmetros na definição de classes, interfaces e métodos
- ➔ É possível reutilizar códigos com diferentes entradas
- ➔ Elimina a necessidade de *casts*
- ➔ Possibilita implementar algoritmos genéricos, que funcionam para diferentes tipos, fáceis de ler, podem ser customizados e são de tipagem segura

Generics[1]

- ➔ Um tipo genérico é uma classe ou interface que é parametrizada por tipos
- ➔ Vamos ver um exemplo sem e com generalização

Generics[1]

```
public class Box {
    private Object object;

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

Generics[1]

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Generics[1]

- ➔ Podemos ter mais de um tipo como parâmetro!
- ➔ A classe de par ordenados faz isso:

Generics[2]

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;
    public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
    }
    public K getKey()    { return key; }
    public V getValue() { return value; }
}
```

Generics[1]

- Com isso, podemos instanciar novos pares ordenados com os seguintes comandos

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
OrderedPair<String, Box<Integer>> p =
            new OrderedPair<>("primes", new Box<Integer>(...));
```

- Note que o tipo pode ser um tipo parametrizado!

Generics[1]

- Existem algumas convenções de nomes para parâmetros de tipos:
- E - Element (usado no Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2º, 3º, 4º tipos

Métodos Genéricos

Métodos Genéricos [3]

- São métodos que introduzem seus próprios parâmetros
- Similar a tipos genéricos, mas o escopo do parâmetro é limitado ao método
  - É possível fazer métodos estáticos, não-estáticos e construtores
- Antes do tipo de retorno é preciso colocar uma lista de parâmetros de tipo, entre “<>”

Métodos Genéricos [3]

```
public class Util {
    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());
    }
}
```

➔ Exemplo de uso:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.<Integer, String>compare(p1, p2);
```

➔ É possível omitir os tipos

```
boolean same = Util.compare(p1, p2);
```

Generics[2]

```
public class GenericMethodTest {
   public static <E> void printArray( E[] inputArray )   {
      for ( E element : inputArray )
         System.out.printf( "%s ", element );
      System.out.println();
   } // fim do método printArray
   public static void main( String args[] )    {
      Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
      Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
      System.out.println( "Array integerArray contains:" );
      printArray( integerArray ); // passa um array de Integers
      System.out.println( "\nArray doubleArray contains:" );
      printArray( doubleArray ); // passa um array Doubles
      System.out.println( "\nArray characterArray contains:" );
      printArray( characterArray ); // passa um array de Characters
   } // fim de main
} // fim da classe GenericMethodTest
```

Generics[4]

➔ Vamos ver um exemplo de pilha genérica!

Limitantes de Parâmetros

Limitantes de Parâmetros [5]

- Às vezes queremos restringir os tipos que podem ser usados como argumentos para um tipo parametrizado
  - Por exemplo, um método que opera com números pode querer aceitar apenas instâncias de *Number* e suas subclasses
- Para isso existem os parâmetros de tipo limitado (*bounded type parameters*)

Limitantes de Parâmetros [5]

- Para isso, na declaração do nome do parâmetro de tipo, coloque depois dele a palavra-chave *extends*, seguida de seu *limitante superior*
  - Isso é diferente do uso de *extends* de **herança**!
- É possível também invocar métodos definidos pela classe limitante!

Limitantes de Parâmetros [5]

```
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!
    }
}
```

Limitantes de Parâmetros [5]

- É possível ter múltiplos limitantes
  - Mas apenas 1 pode ser uma classe
    - Caso tenha uma classe, ela deve ser a primeira

```
class D <T extends A & B & C> { /* ... */ }
```

- No caso, **A** é uma classe e **B** e **C** são interfaces

Limitantes de Parâmetros [6]

```
public class MaximumTest {
    public static < T extends Comparable <T> > T maximum( T x, T y, T z ){
        T max = x; // supõe que x é inicialmente o maior
        if ( y.compareTo( max ) > 0 )
            max = y; // y é o maior até agora
        if ( z.compareTo( max ) > 0 )
            max = z; // z é o maior
        return max; // retorna o maior objeto
    } // fim do mÃ©todo Maximum
    public static void main( String args[] ) {
        System.out.printf( "Maximum of %d, %d and %d is %d\n\n", 3, 4, 5,
                maximum( 3, 4, 5 ));
        System.out.printf( "Maximum of %.1f, %.1f and %.1f is %.1f\n\n",
                6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));
        System.out.printf( "Maximum of %s, %s and %s is %s\n", "pear",
                "apple", "orange", maximum( "pear", "apple", "orange" ));
    } // fim de main
} // fim da classe MaximumTest
```

Subtipos

Subtipos [7]

- Já sabemos que é possível atribuir objetos de um tipo para outro desde que sejam compatíveis
  - Desde que o objeto X “seja um” tipo de objeto Y
    - *Integer* “é um” *Number*
- O mesmo funciona com *generics*
  - Se uma *Box* for *Number*, podemos adicionar *Doubles* nela

Subtipos [7]

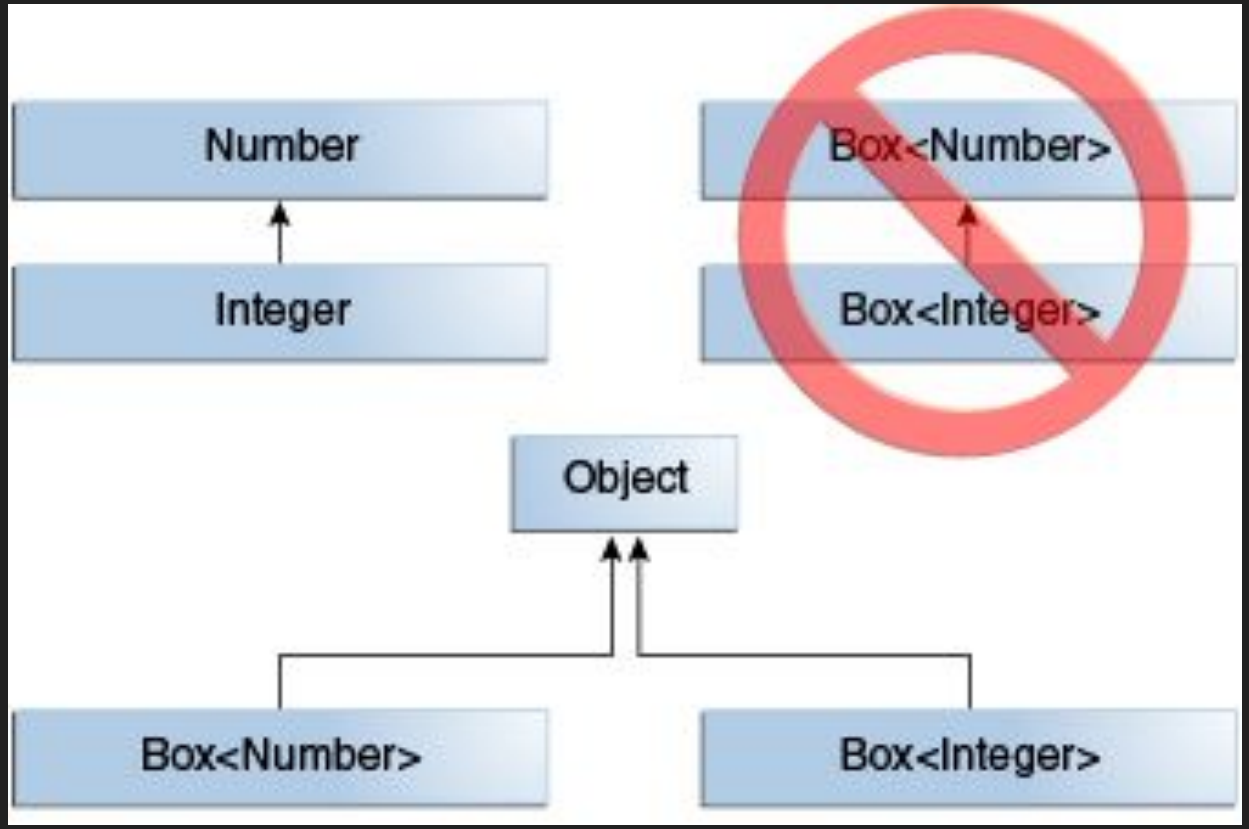
- Ok... mas e se tivermos o seguinte método:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

- Podemos passar Box <Integer> ?
  - Não!
  - Ele não é um subtipo de Box <Number>

Subtipos [7]

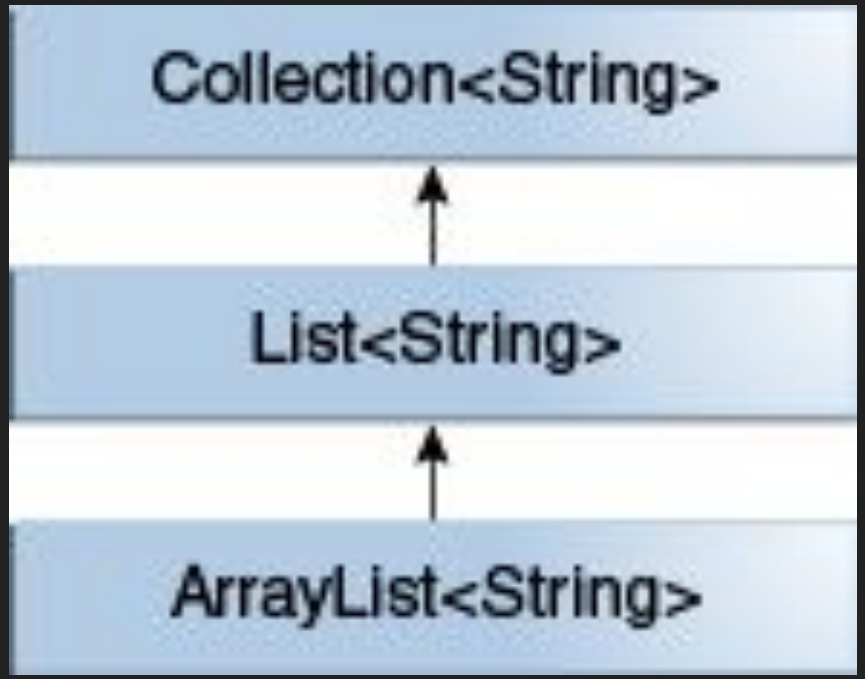
- Dados dois tipos *A* e *B*, *MyClass<A>* **não tem relação** com *MyClass<B>*, **independente da relação entre** *A* e *B*
- O único pai comum de *MyClass<A>* e *MyClass<B>* é *object*


Number
Integer
Box<Number>
Box<Integer>
Object
Box<Number>
Box<Integer>

Fonte: [7]

Subtipos [7]

- A relação entre subtipos é preservada entre os tipos desde que o argumento de tipo não seja modificado
  - *ArrayList<E>* implementa *List<E>*
  - *List<E>* herda de *Collection<E>*
  - Portanto
    - *ArrayList<String>* é subtipo de *List<String>*
    - *List<String>* é subtipo de *Collection<String>*



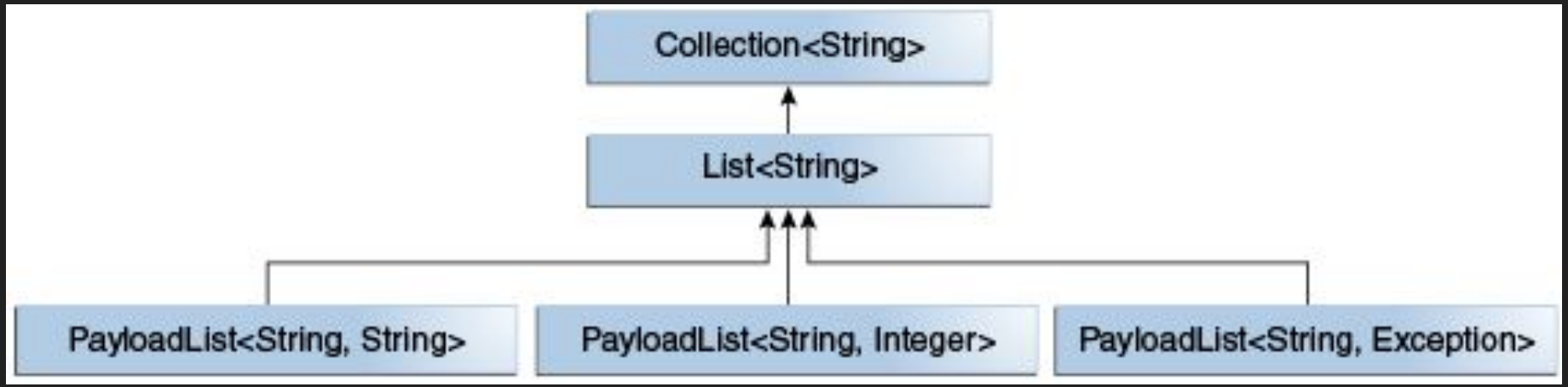
Fonte: [7]

Subtipos [7]

- Se quisermos fazer nossa própria interface de lista, associando um valor opcional do tipo genérico *P* para cada elemento:

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...}
```

- Essas parametrizações são subtipos de *List<String>*:
  - `PayloadList<String,String>`
  - `PayloadList<String,Integer>`
  - `PayloadList<String,Exception>`


Collection<String>
List<String>
PayloadList<String, String>
PayloadList<String, Integer>
PayloadList<String, Exception>

Fonte: [7]

Generics[4]

- Vamos ver alguns exemplos
  - http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/SubTypes.java
  - http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/SubTypes2.java
  - http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/Generico1.java

Wild Cards

Wild Cards [12]

- Ao usarmos o símbolo *?* em genéricos, representamos um tipo desconhecido
- Pode ser usado como tipo de parâmetro, campo ou variável local
  - Às vezes como tipo de retorno, mas não é boa prática
- Ele pode ser usado como limite superior, inferior ou sem limites

Wild Cards [12]

- Limite superior
  - Relaxa restrições da variável
  - Colocar *?* antes de *extends*, seguido pelo limitante superior

```
public static double sumOfList(List<? extends Number> list)
```

- Isso permite usar o método para listas de *Number*, *Integer*, *Double*, etc.
  - O que é mais genérico que *List<Number>*

Wild Cards [12]

```
public static double sumOfList(List<? extends Number> list){
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}

List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li)); //6.0

List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld)); //7.0
```

Wild Cards [12]

- Sem Limitante *?*
  - Tipo desconhecido (*unknow*)
  - Escrita de métodos que usam funcionalidades da classe *object*
  - Usando métodos na classe genérica que não dependem do tipo do parâmetro
    - Ex: *List.size* ou *List.clear*

Wild Cards [12]

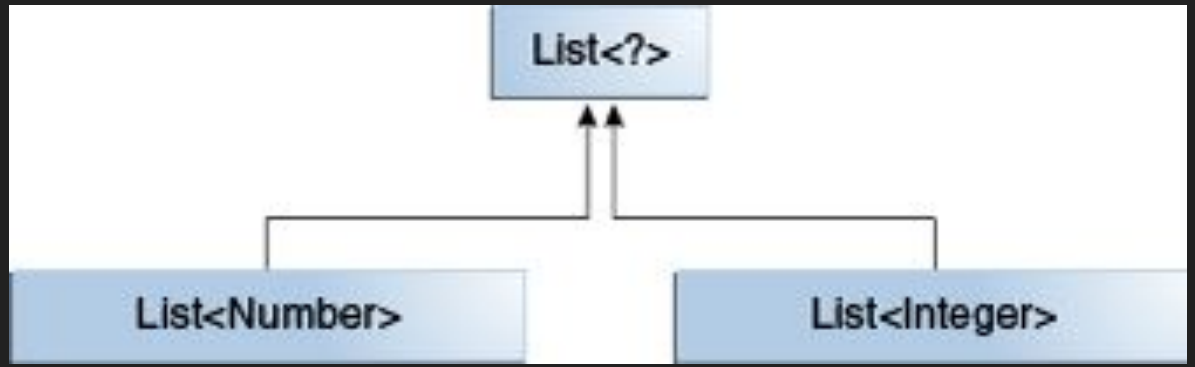
```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
} //Printa elementos de qualquer tipo de lista


List<Integer> li = Arrays.asList(1, 2, 3);

List<String>  ls = Arrays.asList("one", "two", "three");

printList(li);

printList(ls);
```

Wild Cards [12]

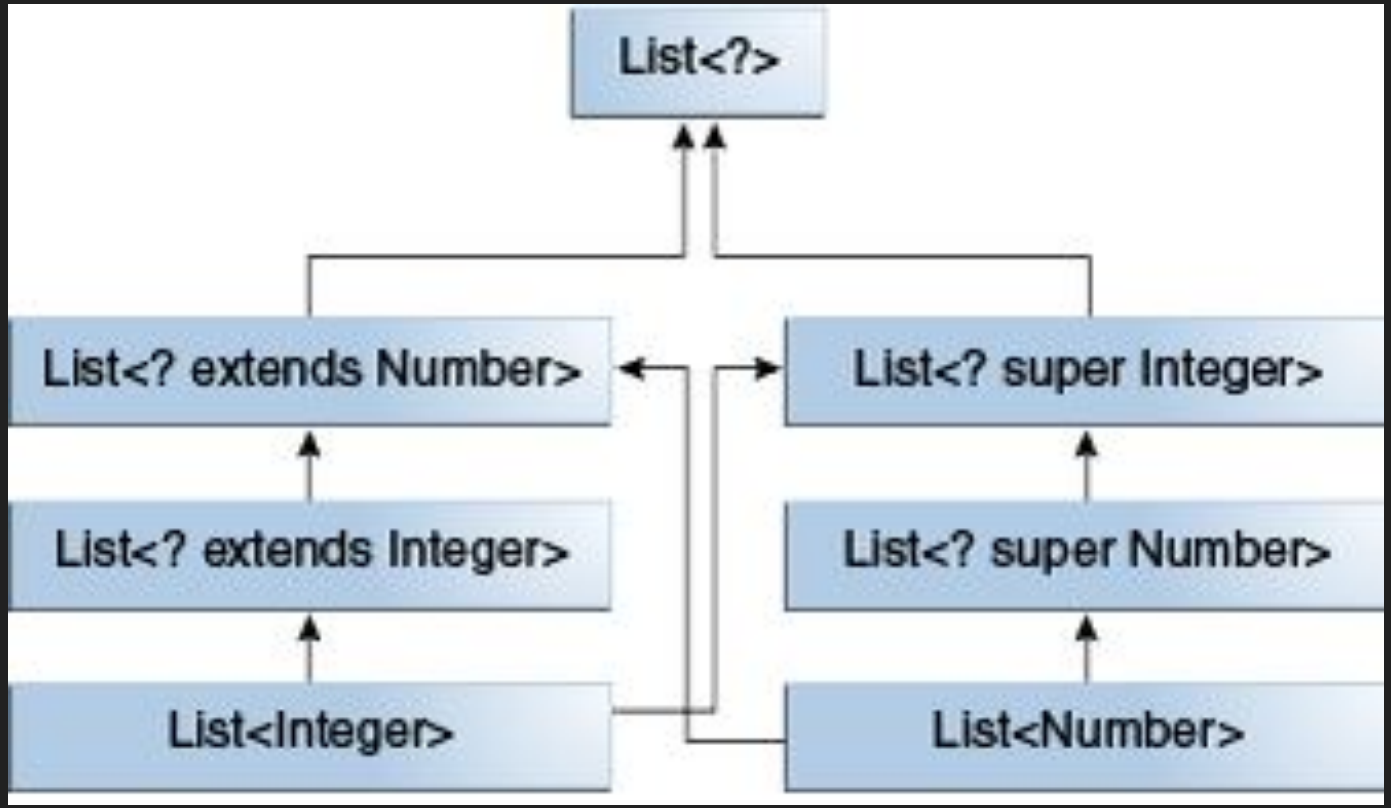
- Limite inferior
  - Restring tipo para ser do tipo passado ou um *super* dele
  - Usa-se *?* seguido de *super* e o limitante inferior
- Por exemplo:
  - Queremos fazer uma lista que aceite qualquer classe que armazene valores *Integer*

Wild Cards [12]

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```



Fonte: [12]


List<?>
List<? extends Number>
List<? super Integer>
List<? extends Integer>
List<? super Number>
List<Integer>
List<Number>

Fonte: [12]

Quando usar *Wild Cards*

Wild Cards [12]

- ➔ É uma variável “*in*” ?
  - ◆ Fornece dados ao código
  - ◆ Imagine um método de cópia de 2 argumentos
    - copy(src, dest)
    - *src* fornece dados para serem copiados “*in*”
- ➔ É uma variável “*out*” ?
  - ◆ Armazena dados para serem usados em outro lugar
  - ◆ *dest* do exemplo anterior

Wild Cards [12]

- “*in*” é definida com um **wildcard de limitante superior**
- “*out*” é definida com um **wildcard de limitante inferior**
- Se “*in*” pode ser acessada por métodos definidos na classe objeto, usar **wildcard sem limitante**
- Se o código precisa acessar a variável como “*in*” e “*out*”, **não usar wildcard**

Wild Cards [12]

- Vamos ver alguns exemplos
  - http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/Generico2.java
  - http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/Generico3.java
  - http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/NoWildcard.java
  - http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/WildcardDemo.java
  - http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/SumNumbers.java

Genéricos em Outras Linguagens

Métodos Genéricos [17, 18, 19]

- ➔ É importante lembrar que a grande maioria dos conceitos vistos até agora com aplicações em Java também são aplicáveis à C++ e C#
- ➔ Em Python o conceito de Generics não é relevante, uma vez que ela é uma linguagem de tipagem dinâmica!
- ➔ As referências indicadas (17 a 19) são bons pontos de partida para entender essas propriedades das linguagens citadas

Referências

1. https://docs.oracle.com/javase/tutorial/java/generics/why.html
2. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/GenericMethodTest.java
3. https://docs.oracle.com/javase/tutorial/java/generics/methods.html
4. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/pilha/pilha.zip
5. https://docs.oracle.com/javase/tutorial/java/generics/bounded.html
6. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/MaximumTest.java
7. https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html
8. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/SubTypes.java
9. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/SubTypes2.java
10. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/Generico1.java
11. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/Generico2.java

Referências

12. https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html
13. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/Generico3.java
14. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/NoWildcard.java
15. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/WildcardDemo.java
16. http://www.lcad.icmc.usp.br/~jbatista/sce537/src/prog13/gen2/SumNumbers.java
17. https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/generics/
18. https://www.geeksforgeeks.org/generics-in-c/
19. https://medium.com/@sergiocosta/generics-em-python-sobrecarga-inclus%C3%A3o-e-outros-tipos-de-polimorfismo-b1d59185f89e