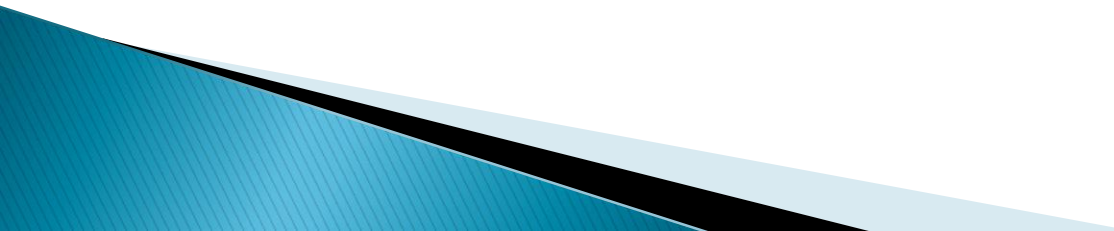


# Genéricos – continuação

*Profa. Thienne Johnson*

*EACH/USP*

# Conteúdo

- ▶ Declarando classes usando generics
  - ▶ Métodos que usam um parâmetro de tipo como tipo de retorno
- 

# Declarando de classes usando generics

# Declarando Classe Utilizando *Generics*

## ▶ Declaração da classe *BasicGeneric*:

```
class BasicGeneric<A>
```

- Contém o parâmetro de tipo: <A>
- Indica que a classe declarada é uma classe *Generics*
- Classe não trabalha com nenhuma referência a um tipo específico

## ▶ Declaração do atributo:

```
private A data;
```

- O atributo *data* é de um tipo *Generic* e depende do tipo de dado com que o objeto *BasicGeneric* for desenvolvido para trabalhar

# Declarando uma instância da classe

- ▶ Deve ser especificado o tipo de referência com qual se vai trabalhar

```
BasicGeneric<String> basicGeneric = new  
    BasicGeneric<String>(data1);
```

```
BasicGeneric<Integer> basicGeneric =new  
    BasicGeneric<Integer>(data1);
```

# Declaração de método

- ▶ Declaração do método *getData*:

```
public A getData() {  
    return data;  
}
```

# Instâncias da classe *BasicGeneric*

- ▶ “presa” ao tipo *String*:

```
BasicGeneric<String> basicGeneric =  
new  
BasicGeneric<String>(data1);  
String data2 = basicGeneric.getData();
```

- ▶ “presa” ao tipo *Integer*:

```
BasicGeneric<Integer> basicGeneric =  
new  
BasicGeneric<Integer>(data1);  
Integer data2 =  
basicGeneric.getData();
```

# *Generics Limitados*

- ▶ Limitando as instanciações de tipo de uma classe:
  - Utilize a palavra-chave *extends no parâmetro de tipo*

```
class ClassName <ParameterName extends  
    ParentClass>
```



# *Generics*: Limitação "Primitiva"

- ▶ Tipos *Generics* do Java são restritos a tipos de referência (objetos) e não funcionarão com tipos primitivos

```
BasicGeneric<int> basicGeneric = new  
BasicGeneric<int>(data1);
```

- ▶ Solução:
  - Encapsular tipos primitivos antes de usá-los
  - Utilizar tipos encapsuladores (*wrapper types*) como argumentos para um tipo *Generics*

# Métodos que usam um parâmetro de tipo como tipo de retorno

# Métodos que usam um parâmetro de tipo como tipo de retorno

- ▶ O método generico `maximum` determina e retorna o maior de 3 argumentos do mesmo tipo.
- ▶ O operador relacional `>` não pode ser usado com tipos de referência, mas é possível comparar 2 objetos da mesma classe se a classe implementar a interface genérica `interface Comparable<T>` (pacote `java.lang`).
- ▶
  - Todos as classes empacotadoras implementam esta interface.

# Métodos que usam um parâmetro de tipo como tipo de retorno(cont.)

- ▶ Objetos `Comparable<T>` tem um método `compareTo`.
  - O método retorna 0 se os objetos são iguais, um inteiro negativo se `object1` é menor do que `object2` ou um inteiro positivo se `object1` é maior do que `object2`.
- ▶ Um benefício de implementar a interface `Comparable<T>` é que objetos `Comparable<T>` podem ser usados com os métodos de ordenar e procurar objetos da classe `Collections` (package `java.util`).

```

1 // Fig. 21.5: MaximumTest.java
2 // Generic method maximum returns the largest of three objects.
3
4 public class MaximumTest
5 {
6     public static void main( String[] args )
7     {
8         System.out.printf( "Maximum of %d, %d and %d is %d\n\n", 3, 4, 5,
9             maximum( 3, 4, 5 ) );
10        System.out.printf( "Maximum of %.1f, %.1f and %.1f is %.1f\n\n",
11            6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );
12        System.out.printf( "Maximum of %s, %s and %s is %s\n", "pear",
13            "apple", "orange", maximum( "pear", "apple", "orange" ) );
14    } // end main
15
16    // determines the largest of three Comparable objects
17    public static < T extends Comparable< T > > T maximum( T x, T y, T z )
18    {
19        T max = x; // assume x is initially the largest
20
21        if ( y.compareTo( max ) > 0 )
22            max = y; // y is the largest so far
23    }

```

← Only Comparable objects can be used with this method

**Fig. 21.5** | Generic method maximum with an upper bound on its type parameter.  
(Part I of 2.)

```
24     if ( z.compareTo( max ) > 0 )
25         max = z; // z is the largest
26
27     return max; // returns the largest object
28 } // end method maximum
29 } // end class MaximumTest
```

Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear

**Fig. 21.5** | Generic method `maximum` with an upper bound on its type parameter.  
(Part 2 of 2.)

## Métodos que usam um parâmetro de tipo como tipo de retorno (cont.)

- ▶ A seção de parâmetro de tipo especifica que `T` estende `Comparable<T>`
  - Somente objetos das classes que implementam a interface `Comparable<T>` podem usar este método.
- ▶ `Comparable` é conhecido como o limite superior do parâmetro de tipo.
  - Por padrão, `Object` é o limite superior.



## Métodos que usam um parâmetro de tipo como tipo de retorno(cont.)

- ▶ Quando o compilador traduz o método `maximum` em Java bytecodes, ele faz *erasure* para trocar os parâmetros de tipo com tipos reais.
- ▶ Todos os parâmetros de tipo são trocados pelo *limite superior* do parâmetro de tipo.
- ▶ Quando o compilador substitui a informação do parâmetro de tipo pelo tipo *limite superior* na declaração de método, ele insere também as operações de *cast* na frente de cada método chamado para realizar *erasure*.



---

```
1 public static Comparable maximum(Comparable x, Comparable y, Comparable z)
2 {
3     Comparable max = x; // assume x is initially the largest
4
5     if ( y.compareTo( max ) > 0 )
6         max = y; // y is the largest so far
7
8     if ( z.compareTo( max ) > 0 )
9         max = z; // z is the largest
10
11     return max; // returns the largest object
12 }
```

---

**Fig. 21.6** | Generic method `maximum` after erasure is performed by the compiler.

# Exemplos



```

public class OverloadedMethods
{
    // method printArray to print Integer array
    public static void printArray( Integer[] inputArray )
    {
        // display array elements
        for ( Integer element : inputArray )
            System.out.printf( "%s ", element );

        System.out.println();
    } // end method printArray

    // method printArray to print Double array
    public static void printArray( Double[] inputArray )
    {
        // display array elements
        for ( Double element : inputArray )
            System.out.printf( "%s ", element );

        System.out.println();
    } // end method printArray

    // method printArray to print Character array
    public static void printArray( Character[] inputArray )
    {
        // display array elements
        for ( Character element : inputArray )
            System.out.printf( "%s ", element );

        System.out.println();
    } // end method printArray

    public static void main( String args[] )
    {
        // create arrays of Integer, Double and Character
        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( "\nArray integerArray contains:" );
        printArray( integerArray ); // pass an Integer array
        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array
        System.out.println( "\nArray characterArray contains:" );

```

```
// Fig. 18.3: GenericMethodTest.java
// Using generic methods to print array of different types.
```

```
public class GenericMethodTest
{
    // generic method printArray
    public static < E > void printArray( E[] inputArray )
    {
        // display array elements
        for ( E element : inputArray )
            System.out.printf( "%s ", element );

        System.out.println();
    } // end method printArray

    public static void main( String args[] )
    {
        // create arrays of Integer, Double and Character
        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 ); // pass an Integer array
        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array
        System.out.println( "\nArray characterArray contains:" );
        printArray( characterArray ); // pass a Character array
    } // end main
} // end class GenericMethodTest
```

// Fig. 18.5: MaximumTest.java

// Generic method maximum returns the largest of three objects.

```
public class MaximumTest
{
    // determines the largest of three Comparable objects
    public static < T extends Comparable< T > > T maximum( T x, T y, T z )
    {
        T max = x; // assume x is initially the largest

        if ( y.compareTo( max ) > 0 )
            max = y; // y is the largest so far

        if ( z.compareTo( max ) > 0 )
            max = z; // z is the largest

        return max; // returns the largest object
    } // end method 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" ) );
    } // end main
} // end class MaximumTest
```

# Exercício 1

Considere as seguintes classes:

```
public class AnimalHouse<E> {  
    private E animal;  
    public void setAnimal(E x) {  
        animal = x;  
    }  
    public E getAnimal() {  
        return animal;  
    }  
}  
  
public class Animal{}  
public class Cat extends Animal {}  
public class Dog extends Animal {}
```

Para os seguintes trechos de código, identifique se o código acima falha para compilar, compila com warning, gera um erro de runtime, ou nenhum dos anteriores (compila e executa sem problema)

- a. **AnimalHouse<Animal> house = new AnimalHouse<Cat>();**
- b. **AnimalHouse<Dog> house = new AnimalHouse<Animal>();**
- c. **AnimalHouse<?> house = new AnimalHouse<Cat>(); house.setAnimal(new Cat());**
- d. **AnimalHouse house = new AnimalHouse(); house.setAnimal(new Dog());**

# Exercício 2

- ▶ Para o seguinte código, escreva outra versão usando Genericos

```
import java.util.List;
import java.util.ArrayList;
public class Library {
    private List resources = new ArrayList();
    public void addMedia(Media x) {
        resources.add(x);
    }
    public Media retrieveLast() {
        int size = resources.size();
        if (size > 0) { return (Media)resources.get(size - 1);}
        return null;
    }
}
interface Media {}
interface Book extends Media {}
interface Video extends Media {}
interface Newspaper extends Media {}
```