# TIPOS GENÉRICOS

# REFERÊNCIA


A elaboração deste material foi baseada no livro:
Schildt, Herbert. Java para Iniciantes. 6° Edição. Bookman, 2015.

# FUNDAMENTOS

- Os genéricos adicionaram um elemento de sintaxe totalmente novo e causaram mudanças em muitas das classes e métodos da API principal.
- A sua inclusão basicamente reformulou a natureza de Java.
- Termo genéricos refere-se a tipos parametrizados.
- Os tipos parametrizados são importantes porque permitem criar classes, interfaces e métodos em que o tipo de dado usado é especificado como parâmetro.
  - Uma classe, interface ou método que opera sobre um parâmetro de tipo é chamado de genérico, como em classe genérica ou método genérico.
- Uma vantagem importante do código genérico é que ele funciona automaticamente com o tipo de dado passado para seu parâmetro de tipo.
- Os genéricos propiciam segurança de tipos, pois tornam as coerções automáticas e implícitas. Ou seja, expandem a habilidade de reutilizar código e permitem fazê-lo de maneira segura e confiável.

# EXEMPLO DE UMA CLASSE GENÉRICA SIMPLES

```
// Classe genérica simples.
// Aqui, T é um parâmetro de tipo que
// será substituído pelo tipo real quando
// um objeto de tipo Gen for criado.
class Gen<T> {
  T ob; // declara um objeto de tipo T

  // Passa para o construtor uma
  // referência a um objeto de tipo T
  Gen(T o) {
    ob = o;
  }

  // Retorna ob.
  T getob() {
    return ob;
  }

  // Exibe o tipo de T.
  void showType() {
    System.out.println("Type of T is " +
                       ob.getClass().getName());
    }
  }
```

Declara uma classe genérica. **T** é o parâmetro de tipo genérico.

Créditos: Schildt, Herbert. Java para Iniciantes. 6° Edição. Bookman, 2015

```
// Demonstra a classe genérica.
class GenDemo {
  public static void main(String args[]) {
    // Cria uma referência Gen para Integers.
    Gen<Integer> iOb;

    // Cria um objeto Gen<Integer> e atribui sua
    // referência a iOb. Observe o uso do autoboxing no
    // encapsulamento do valor 88 dentro de um objeto Integer.
    iOb = new Gen<Integer>(88);

    // Exibe o tipo de dado usado por iOb.
    iOb.showType();

    // Obtém o valor de iOb. Observe
    // que nenhuma coerção é necessária.
    int v = iOb.getob();
    System.out.println("value: " + v);

    System.out.println();

    // Cria um objeto Gen para Strings.
    Gen<String> strOb = new Gen<String>("Generics Test");

    // Exibe o tipo de dado usado por strOb.
    strOb.showType();

    // Obtém o valor de strOb. Novamente, observe
    // que nenhuma coerção é necessária.
    String str = strOb.getob();
    System.out.println("value: " + str);
  }
}
```

Cria uma referência a um objeto de tipo **Gen<Integer>**.

Instancia um objeto de tipo **Gen<Integer>**.

Cria uma referência e um objeto de tipo **Gen<String>**.

Créditos: Schildt, Herbert. Java para Iniciantes. 6° Edição. Bookman, 2015

# GENÉRICOS SÓ FUNCIONAM COM TIPOS DE REFERÊNCIA

- Na declaração de uma instância de um tipo genérico, o argumento de tipo passado para o parâmetro de tipo deve ser um tipo de referência.
- Não pode-se utilizar um tipo primitivo, como int ou char.
- Por exemplo, com Gen, no exemplo anterior, é possível passar qualquer tipo de classe para T, mas você não pode passar um tipo primitivo para T. Logo, a declaração a seguir é inválida:
  - **Gen<int> intOb = new Gen<int>(53); → Erro, não pode usar um tipo primitivo**
- Certamente, não poder especificar um tipo primitivo não é uma restrição grave, porque pode-se utilizar os encapsuladores de tipos para encapsular um tipo primitivo.

# TIPOS GENÉRICOS DIFEREM DE ACORDO COM SEUS ARGUMENTOS DE TIPO

- Uma referência de uma versão específica de um tipo genérico não tem compatibilidade de tipo com outra versão do mesmo tipo genérico.
- Por exemplo, supondo um objeto **strOb** seja do tipo **Gen<String>** e **iOb** seja do tipo **Gen<Integer>**, a linha de código abaixo está errada e não será compilada:
  - **iOb = strOb; → Errado!**
- Ainda que tanto **iOb** quanto **strOb** sejam de tipo **Gen<T>**, são referências a tipos diferentes porque seus argumentos de tipo diferem. Isso faz parte da maneira como os genéricos adicionam segurança de tipos e evitam erros.

# CLASSE GENÉRICA COM DOIS PARÂMETROS DE TIPO

- Você pode declarar mais de um parâmetro de tipo em um tipo genérico. Para especificar dois ou mais parâmetros de tipo, usa-se uma lista separada por vírgulas. Exemplo:

```
// Classe genérica simples com
// dois parâmetros de tipos: T e V.
class TwoGen<T, V> {
  T ob1;
  V ob2;

  // Passa para o construtor referências
  // a objetos de tipo T e V.
  TwoGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }

  // Exibe os tipos de T e V.
  void showTypes() {
    System.out.println("Type of T is " +
                       ob1.getClass().getName());

    System.out.println("Type of V is " +
                       ob2.getClass().getName());
  }

  T getob1() {
    return ob1;
  }

  V getob2() {
    return ob2;
  }
}
```

Usa dois parâmetros de tipo.

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

```
// Demonstra TwoGen.
class SimpGen {
  public static void main(String args[]) {

    TwoGen<Integer, String> tgObj =
      new TwoGen<Integer, String>(88, "Generics");

    // Exibe os tipos.
    tgObj.showTypes();

    // Obtém e exibe valores.
    int v = tgObj.getob1();

    System.out.println("value: " + v);

    String str = tgObj.getob2();
    System.out.println("value: " + str);
  }
}
```

Aqui, **Integer** é passado para **T** e **String** é passado para **V**.

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

# TIPOS LIMITADOS

- No caso dos parâmetros de tipo poder serem substituídos por qualquer tipo de classe, em muitos casos, é bom.
- Contudo, as vezes é útil limitar os tipos que podem ser passados para um parâmetro de tipo.
- Por exemplo: supõe-se a necessidade de criar uma classe genérica que armazene um valor numérico para executar várias funções matemáticas. É possível usar a classe para calcular valores para qualquer tipo de número, inclusive **Integer**, **Float** e **Double**.
- No exemplo da classe ao lado, a classe não será compilada, porque os dois métodos gerarão erros devido pressupor que haverá um cálculo com números.

```
// NumericFns tenta (sem sucesso) criar uma
// classe genérica que possa executar várias
// funções numéricas, como calcular o recíproco ou o
// componente fracionário, dado qualquer tipo de número.
class NumericFns<T> {
  T num;

  // Passa para o construtor uma referência
  // a um objeto numérico.
  NumericFns(T n) {
    num = n;
  }

  // Retorna o recíproco.
  double reciprocal() {
    return 1 / num.doubleValue(); // Erro!
  }

  // Retorna o componente fracionário.
  double fraction() {
    return num.doubleValue() - num.intValue(); // Erro!
  }

  // ...
}
```

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

# TIPOS LIMITADOS

- Para tratar essas situações, Java fornece os tipos limitados. Na especificação de um parâmetro de tipo, deve-se criar um limite superior declarando a superclasse da qual todos os argumentos de tipo devem derivar.
- Usa-se a cláusula **extends** na especificação do parâmetro de tipo.
  - **<T extends superclasse>**
- Essa sintaxe especifica que **T** só pode ser substituído pela superclasse, ou por subclasses da superclasse. Logo, superclasse define um limite superior no qual ela também se inclui.
- Pode-se usar um limite superior para corrigir a classe **NumericFns** mostrada anteriormente especificando **Number** como o limite.

```
// Nesta versão de NumericFns, o argumento de
// tipo de T deve ser Number ou uma classe
// derivada de Number.
class NumericFns<T extends Number> {
  T num;

  // Passa para o construtor uma referência
  // a um objeto numérico.
  NumericFns(T n) {
    num = n;
  }

  // Retorna o recíproco.
  double reciprocal() {
    return 1 / num.doubleValue();
  }

  // Retorna o componente fracionário.
  double fraction() {
    return num.doubleValue() - num.intValue();
  }

  // ...
}
```

Nesse caso, o argumento de tipo deve ser **Number** ou uma subclasse de **Number**.

Créditos: Schildt, Herbert. Java para Iniciantes. 6° Edição. Bookman, 2015

# ARGUMENTOS CURINGAS

- Mesmo sendo útil, às vezes a segurança de tipos pode invalidar construções perfeitamente aceitáveis.
- Para criar um método genérico, deve-se usar outro recurso dos genéricos Java: o argumento curinga. O argumento curinga é especificado pelo símbolo ? e representa um tipo desconhecido

```
// Usa um curinga.
class NumericFns<T extends Number> {
  T num;

  // Passa para o construtor uma referência
  // a um objeto numérico.
  NumericFns(T n) {
    num = n;
  }

  // Retorna o recíproco.
  double reciprocal() {
    return 1 / num.doubleValue();
  }

  // Retorna o componente fracionário.
  double fraction() {
    return num.doubleValue() - num.intValue();
  }

  // Determina se os valores absolutos de
  // dois objetos são iguais.
  boolean absEqual(NumericFns<?> ob) {
    if(Math.abs(num.doubleValue()) ==
          Math.abs(ob.num.doubleValue())) return true;

    return false;
  }

  // ...
}
```

```
// Demonstra um curinga.
class WildcardDemo {
  public static void main(String args[]) {

    NumericFns<Integer> iOb =
                  new NumericFns<Integer>(6);

    NumericFns<Double> dOb =
                  new NumericFns<Double>(-6.0);

    NumericFns<Long> lOb =
                  new NumericFns<Long>(5L);

    System.out.println("Testing iOb and dOb.");
    if(iOb.absEqual(dOb))
      System.out.println("Absolute values are equal.");
    else
      System.out.println("Absolute values differ.");

    System.out.println();

    System.out.println("Testing iOb and lOb.");
    if(iOb.absEqual(lOb))
      System.out.println("Absolute values are equal.");
    else
      System.out.println("Absolute values differ.");

  }
}
```

Nesta chamada, o tipo curinga equivale a **Double**.

Nesta chamada, o curinga equivale a **Long**.

Créditos: Schildt, Herbert. Java para Iniciantes. 6° Edição. Bookman, 2015

# CURINGAS LIMITADOS

- Os argumentos curingas podem ser limitados de maneira semelhante aos parâmetros de tipo. Um curinga limitado é particularmente importante quando estamos criando um método projetado para operar somente com objetos que sejam subclasses de uma superclasse específica. Como exemplo, considere o conjunto de classes a seguir:

```
class A {
  // ...
}

class B extends A {
  // ...
}

class C extends A {
  // ...
}

// Observe que D NÃO estende A.
class D {
  // ...
}

// Classe genérica simples.
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }
}
```

```
class UseBoundedWildcard {
  // Aqui, o símbolo ? equivalerá a A ou a
  // qualquer tipo de classe que estenda A.
  static void test(Gen<? extends A> o) {   <-- Usa um curinga limitado.
    // ...
  }

  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();

    Gen<A> w = new Gen<A>(a);
    Gen<B> w2 = new Gen<B>(b);
    Gen<C> w3 = new Gen<C>(c);
    Gen<D> w4 = new Gen<D>(d);

    // Estas chamadas a test() estão corretas.
    test(w);
    test(w2);
    test(w3);   <-- Essas chamadas são válidas porque w, w2 e w3 são subclasses de A.

    // Não pode chamar test() com w4 porque
    // ele não é um objeto de uma classe que
    // herde A.
//    test(w4); // Error!   <-- Não é válido porque w4 não é subclasse de A.
  }
}
```

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

Tipos Genéricos

# MÉTODOS GENÉRICOS

- Os métodos de uma classe genérica podem fazer uso do parâmetro de tipo da classe e, portanto, são automaticamente genéricos de acordo com o parâmetro de tipo.
- Pode-se declarar um método genérico que use um ou mais parâmetros de tipo exclusivamente seus.
- Pode-se criar um método genérico embutido em uma classe não genérica.
- Exemplo:
- O programa ao lado declara uma classe não genérica chamada **GenericMethodDemo** e um método genérico estático dentro dessa classe chamado **arraysEqual( )**. Esse método determina se dois arrays contêm os mesmos elementos, na mesma ordem. Pode ser usado para comparar dois arrays, sejam eles quais forem, contanto que sejam de tipos iguais ou compatíveis e seus elementos sejam comparáveis.
- No método **arraysEqual**, os parâmetros de tipo são declarados antes do tipo de retorno do método. **T** estende **Comparable<T>**. **Comparable** é uma interface declarada em **java.lang**. Uma classe que implementa **Comparable** define objetos que podem ser ordenados. Logo, usar um limite superior **Comparable** assegura que **arraysEqual( )** só possa ser usado com objetos que possam ser comparados. **Comparable** é genérica e seu parâmetro de tipo especifica o tipo dos objetos que ela compara.

```
// Demonstra um método genérico simples.
class GenericMethodDemo {

  // Determina se o conteúdo de dois arrays é igual.
  static <T extends Comparable<T>, V extends T> boolean
    arraysEqual(T[] x, V[] y) {          <-- Método genérico
    // Se o tamanho dos arrays for diferente, os arrays também serão.
    if(x.length != y.length) return false;

    for(int i=0; i < x.length; i++)
      if(!x[i].equals(y[i])) return false; // os arrays são diferentes

    return true; // os conteúdos dos arrays são equivalentes
  }

  public static void main(String args[]) {

    Integer nums[] = { 1, 2, 3, 4, 5 };
    Integer nums2[] = { 1, 2, 3, 4, 5 };
    Integer nums3[] = { 1, 2, 7, 4, 5 };
    Integer nums4[] = { 1, 2, 7, 4, 5, 6 };

    if(arraysEqual(nums, nums))          <-- Os argumentos de tipo de T e V são determinados implicitamente quando o método é chamado.
      System.out.println("nums equals nums");

    if(arraysEqual(nums, nums2))
      System.out.println("nums equals nums2");

    if(arraysEqual(nums, nums3))
      System.out.println("nums equals nums3");

    if(arraysEqual(nums, nums4))
      System.out.println("nums equals nums4");

    // Cria um array de Doubles
    Double dvals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

    // Essa parte não será compilada, porque nums
    // e dvals não são do mesmo tipo.
//    if(arraysEqual(nums, dvals))
//      System.out.println("nums equals dvals");
  }
}
```

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

# CONSTRUTORES GENÉRICOS

- Um construtor pode ser genérico, mesmo se sua classe não o seja.
- Por exemplo, no programa a seguir, a classe **Summation** não é genérica, mas seu construtor é genérico.

```
// Usa um construtor genérico.
class Summation {
  private int sum;

  <T extends Number> Summation(T arg) {   <-- Construtor genérico
    sum = 0;

    for(int i=0; i <= arg.intValue(); i++)
      sum += i;
  }

  int getSum() {
    return sum;
  }
}

class GenConsDemo {
  public static void main(String args[]) {
    Summation ob = new Summation(4.0);

    System.out.println("Summation of 4.0 is " +
                       ob.getSum());
  }
}
```

Créditos: Schildt, Herbert. Java para Iniciantes. 6° Edição. Bookman, 2015

# INTERFACES GENÉRICAS

- Uma interface pode ser genérica. As interfaces genéricas são especificadas como as classes genéricas. A seguir, um exemplo em que é criada uma interface chamada **Containment**, que pode ser implementada por classes que armazenem um ou mais valores. Também declara um método chamado **contains( )** que determina se um valor especificado está contido no objeto chamador.

```
// Exemplo de interface genérica.

// Uma interface genérica que lida com armazenamento.
// Esta interface requer que a classe usuária
// tenha um ou mais valores.
interface Containment<T> {
  // O método contains() verifica se um item
  // especificado está contido dentro de um
  // objeto que implementa Containment.
  boolean contains(T o);
}

// Implementa Containment usando um array
// para armazenar os valores.
class MyClass<T> implements Containment<T> {
  T[] arrayRef;

  MyClass(T[] o) {
    arrayRef = o;
  }

  // Implementa contains()
  public boolean contains(T o) {
    for(T x : arrayRef)
      if(x.equals(o)) return true;
    return false;
  }
}
```

Interface genérica.

Toda classe que implemente uma interface genérica também deve ser genérica.

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

```
class GenIFDemo {
  public static void main(String args[]) {
    Integer x[] = { 1, 2, 3 };

    MyClass<Integer> ob = new MyClass<Integer>(x);

    if(ob.contains(2))
      System.out.println("2 is in ob");
    else
      System.out.println("2 is NOT in ob");

    if(ob.contains(5))
      System.out.println("5 is in ob");
    else
      System.out.println("5 is NOT in ob");

    // A parte a seguir não é válida porque ob
    // é um objeto Containment de tipo Integer e
    // 9.25 é um valor Double.
//  if(ob.contains(9.25)) // Inválido!
//    System.out.println("9.25 is in ob");
  }
}
```

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

# ALGUMAS RESTRIÇÕES DOS GENÉRICOS

Há algumas restrições que deve-se atentar ao usar genéricos. Por exemplo:

## Parâmetros de tipos não podem ser instanciados

- Não é possível criar uma instância de um parâmetro de tipo. Por exemplo, considere a classe a seguir:

```
// Não é possível criar uma instância de T.
class Gen<T> {
  T ob;
  Gen() {
    ob = new T(); // Inválido!!!
  }
}
```

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

## Restrições aos membros estáticos

- Nenhum membro **static** pode usar um parâmetro de tipo declarado pela classe externa. Por exemplo, os dois membros **static** dessa classe não são válidos:

```
class Wrong<T> {
  // Errado, não há variáveis estáticas de tipo T.
  static T ob;

  // Errado, nenhum método estático pode usar T.
  static T getob() {
    return ob;
  }
}
```

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015

# ALGUMAS RESTRIÇÕES DOS GENÉRICOS

Há algumas restrições que deve-se atentar ao usar genéricos, como restrições a exceções genéricas. Por exemplo:

## Restrições aos arrays genéricos

- Há duas restrições importantes dos genéricos aplicáveis aos **arrays**. Em primeiro lugar, não pode instanciar um **array** cujo tipo do elemento seja um parâmetro de tipo. Em segundo lugar, não pode criar um **array** de referências genéricas específicas de um tipo.

## Restrições a exceções genéricas

- Uma classe genérica não pode estender **Throwable**. Ou seja, pode-se criar classes de exceção genéricas.

**Exemplo**

```
// Genéricos e arrays.
class Gen<T extends Number> {
  T ob;

  T vals[]; // Correto

  Gen(T o, T[] nums) {
    ob = o;

    // Esta instrução não é válida.
//  vals = new T[10]; // não pode criar um array de tipo T

    // Mas esta instrução está correta.
    vals = nums; // É correto atribuir referências de um array
existente
  }
}

class GenArrays {
  public static void main(String args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };

    Gen<Integer> iOb = new Gen<Integer>(50, n);

    // Não pode criar um array de referências genéricas específicas de
um tipo..
    // Gen<Integer> gens[] = new Gen<Integer>[10]; // Errado!

    // Isso é correto.
    Gen<?> gens[] = new Gen<?>[10]; // Correto
  }
}
```

Créditos: Schildt, Herbert. Java para Iniciantes. 6º Edição. Bookman, 2015