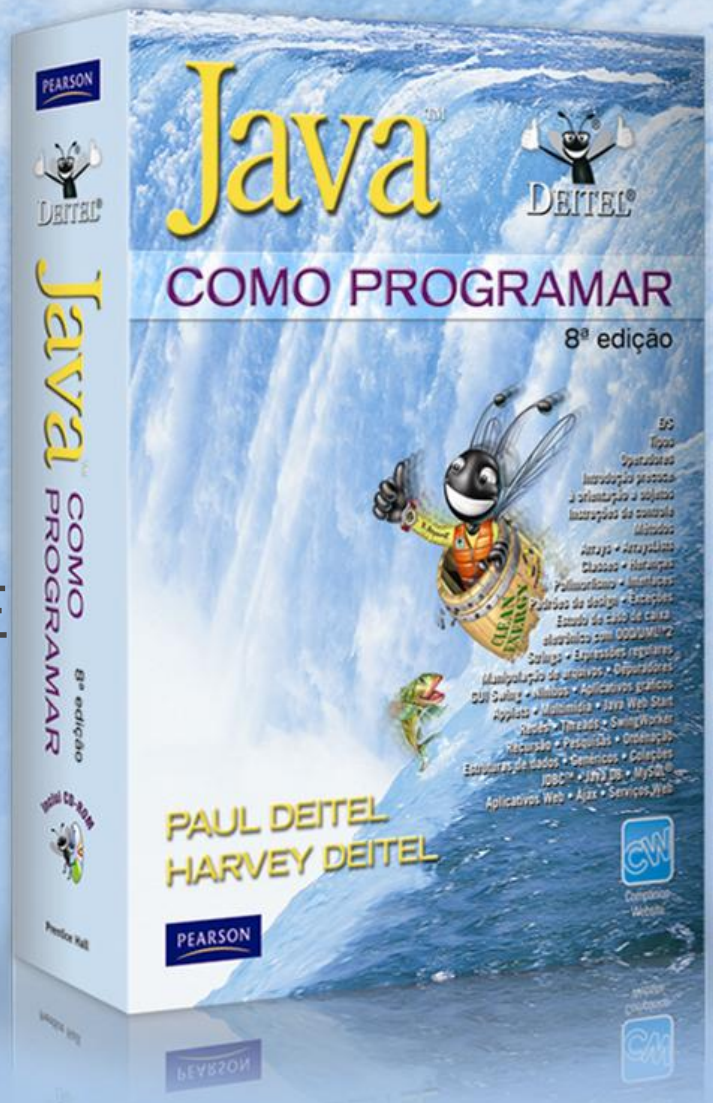
Capítulo 21
Classes e métodos genéricos

Java Como Programar, 8/E



COMO PROGRAMAR

OBJETIVOS

Neste capítulo, você aprenderá:

- A criar métodos genéricos que realizam tarefas idênticas em argumentos de diferentes tipos.
- A criar uma classe `Stack` genérica que pode ser utilizada para armazenar objetos de qualquer tipo de classe ou interface.
- A entender como sobrecarregar métodos genéricos com métodos não genéricos ou com outros métodos genéricos.
- A entender tipos brutos e como eles ajudam a alcançar a retrocompatibilidade.
- A utilizar curingas quando informações precisas de tipo sobre um parâmetro não são requeridas no corpo de método.
- A identificar o relacionamento entre herança e genéricos.

21.1 Introdução

- **Métodos genéricos** e **classes genéricas** (e interfaces) permitem especificar, com uma única declaração de método, um conjunto de métodos relacionados ou, com uma única declaração de classe, um conjunto de tipos relacionados, respectivamente.
- Os genéricos também fornecem segurança de tipo em tempo de compilação que permite capturar tipos inválidos em tempo de compilação.

Observação de engenharia de software 21.1

Métodos e classes genéricas estão entre as capacidades mais poderosas do Java para reutilização de software com segurança de tipo em tempo de compilação.

21.2 Motivação para métodos genéricos

- Métodos sobrecarregados são frequentemente utilizados para realizar operações semelhantes em tipos diferentes de dados.
- Estude cada método `printArray`.
- Observe que o tipo de elemento do array de tipos aparece no cabeçalho de cada método e no cabeçalho da instrução `for`.
- Se fôssemos substituir os tipos de elementos em cada método com um nome genérico — `T` por convenção — então todos os três métodos se pareceriam com aquele na Figura 21.2.


```
1  // Figura 21.1: OverloadedMethods.java
2  // Imprimindo elementos do array com métodos sobrecarregados.
3  public class OverloadedMethods
4  {
5     public static void main( String[] args )
6     {
7        // cria arrays de Integer, Double e Character
8        Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
9        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
10       Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
11
12       System.out.println( "Array integerArray contains:" );
13       printArray( integerArray ); // passa um array de Integer
14       System.out.println( "\nArray doubleArray contains:" );
15       printArray( doubleArray ); // passa um array Double
16       System.out.println( "\nArray characterArray contains:" );
17       printArray( characterArray ); // passa um array de Character
18    } // fim de main
19
```

Figura 21.1 | Imprimindo elementos do array com métodos sobrecarregados. (Parte 1 de 3.)


```
20      // método printArray para imprimir um array de Integer
21      public static void printArray( Integer[] inputArray )
22      {
23         // exibe elementos do array
24         for ( Integer element : inputArray )
25            System.out.printf( "%s ", element );
26
27         System.out.println();
28      } // fim do método printArray
29
30      // método printArray para imprimir um array de Double
31      public static void printArray( Double[] inputArray )
32      {
33         // exibe elementos do array
34         for ( Double element : inputArray )
35            System.out.printf( "%s ", element );
36
37         System.out.println();
38      } // fim do método printArray
39
```

Figura 21.1 | Imprimindo elementos do array com métodos sobrecarregados. (Parte 2 de 3.)


```
40       // método printArray para imprimir um array de Character
41       public static void printArray( Character[] inputArray )
42       {
43          // exibe elementos do array
44          for ( Character element : inputArray )
45             System.out.printf( "%s ", element );
46
47          System.out.println();
48       } // fim do método printArray
49    } // fim da classe OverloadedMethods
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Figura 21.1 | Imprimindo elementos do array com métodos sobrecarregados. (Parte 3 de 3.)


```
1   public static void printArray( T[] inputArray )
2   {
3      // exibe elementos do array
4      for ( T element : inputArray)
5         System.out.printf( "%s ", element );
6
7      System.out.println();
8   } // fim do método printArray
```

Figura 21.2 | Método `printArray` em que nomes de tipos reais são substituídos pelo nome genérico `T` por convenção.

21.3 Métodos genéricos: implementação e tradução em tempo de compilação

- Se as operações realizadas por vários métodos sobrecarregados forem idênticas para cada tipo de argumento, os métodos sobrecarregados podem ser codificados mais compacta e convenientemente com um método genérico.
- Você pode escrever uma única declaração de método genérico que pode ser chamada com argumentos de tipos diferentes.
- Com base nos tipos dos argumentos passados para o método genérico, o compilador trata cada chamada de método apropriadamente.
- A linha 22 inicia a declaração do método `printArray`.


```
1  // Figura 21.3: GenericMethodTest.java
2  // Imprimindo elementos do array com o método genérico printArray.
3
4  public class GenericMethodTest
5  {
6     public static void main( String[] args )
7     {
8        // cria arrays de Integer, Double e Character
9        Integer[] intArray = { 1, 2, 3, 4, 5, 6 };
10       Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
11       Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
12
13       System.out.println( "Array integerArray contains:" );
14       printArray( integerArray ); // passa um array de Integer
15       System.out.println( "\nArray doubleArray contains:" );
16       printArray( doubleArray ); // passa um array Double
17       System.out.println( "\nArray characterArray contains:" );
18       printArray( characterArray ); // passa um array de Character
19    } // fim de main
20
```

Figura 21.3 | Imprimindo elementos do array com o método genérico `printArray`. (Parte 1 de 2.)

```
21      // método genérico printArray
22      public static < T > void printArray( T[] inputArray )
23      {
24         // exibe elementos do array
25         for ( T element : inputArray )
26            System.out.printf( "%s ", element );
27
28         System.out.println();
29      } // fim do método printArray
30   } // fim da classe GenericMethodTest
```

Qualquer coisa pode ser impressa como uma `String`, portanto, um método genérico não é realmente necessário aqui

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Figura 21.3 | Imprimindo elementos do array com o método genérico `printArray`. (Parte 2 de 2.)

- Todas declarações de método genérico têm uma **seção de parâmetro de tipo** delimitada por **colchetes angulares** (< e >) que precede o tipo de retorno do método (< T > nesse exemplo).
- Cada seção de parâmetro de tipo contém um ou mais **parâmetros de tipos** (também chamados **parâmetros de tipo formais**), separados por vírgulas.
- Um parâmetro de tipo, também conhecido como **variável de tipo**, é um identificador que especifica o nome de um tipo genérico.
- Pode ser utilizado para declarar o tipo de retorno, tipos de parâmetros e tipos de variáveis locais em um método genérico, e atuam como marcadores de lugar para os tipos dos argumentos passados ao método genérico (**argumentos de tipos reais**).
- O corpo de um método genérico é declarado como o de qualquer outro método.
- Parâmetros de tipo podem representar somente tipos por referência — não tipos primitivos.

Erro comum de programação 21.1

Ao declarar um método genérico, não conseguir colocar uma seção de parâmetro de tipo antes do tipo de retorno de um método é um erro de sintaxe — o compilador não entenderá os nomes do parâmetro de tipo quando eles forem encontrados no método.

Boa prática de programação 21.1

É recomendável que parâmetros de tipos sejam especificados como letras maiúsculas individuais. Tipicamente, um parâmetro de tipo que representa o tipo de um elemento do array (ou outra coleção) é nomeado T.

Erro comum de programação 21.2

Se o compilador não puder encontrar uma correspondência entre uma chamada de método e uma declaração de método genérico ou não genérico, ocorrerá um erro de compilação.

Erro comum de programação 21.3

Se o compilador não encontrar uma declaração de método que corresponda exatamente a uma chamada de método, mas encontrar dois ou mais métodos genéricos que podem satisfazer a chamada de método, ocorrerá um erro de compilação.

- Quando o compilador traduz o método genérico `printArray` em bytecodes Java, ele remove a seção de parâmetro de tipo e substitui os parâmetros de tipo por tipos reais.
- Esse processo é conhecido como **erasure.**
- Por padrão, todos os tipos genéricos são substituídos pelo tipo `Object`.
- Assim, a versão compilada do método `printArray` aparece como mostrada na Figura 21.4 — há somente uma cópia desse código utilizada para todas as chamadas a `printArray` no exemplo.


```
1   public static void printArray(Object [] inputArray )
2   {
3      // exibe elementos do array
4      for (Object element : inputArray )
5         System.out.printf( "%s ", element );
6
7      System.out.println();
8   } // fim do método printArray
```

Figura 21.4 | O método genérico `printArray` depois de a erasure ser realizada pelo compilador.

21.4 Questões adicionais da tradução em tempo de compilação: métodos que utilizam um parâmetro de tipo como o tipo de retorno

- O método genérico `maximum` determina e retorna o maior de seus três argumentos do mesmo tipo.
- O operador relacional > não pode ser utilizado com tipos de referência, mas é possível comparar dois objetos da mesma classe se essa classe implementa a **interface** genérica **Comparable<T>** (pacote `java.lang`).
- Todas as classes empacotadoras de tipo para tipos primitivos implementam essa interface.
- Como ocorre com classes genéricas, **interfaces genéricas** permitem especificar, com uma única declaração de interface, um conjunto de tipos relacionados.

- Os objetos `Comparable<T>` têm um **método compareTo.**
- O método deve retornar 0 se os objetos forem iguais, um número inteiro negativo se `object1` for menor que `object2` ou um número inteiro positivo se `object1` for maior que `object2`.
- Um benefício da implementação da interface `Comparable<T>` é que objetos `Comparable<T>` podem ser utilizados com os métodos de classificação e pesquisa da classe `Collections` (pacote `java.util`).

```
1  // Figura 21.5: MaximumTest.java
2  // O método genérico maximum retorna o maior dos três objetos.
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    } // fim de main
15
16    // determina o maior dos três objetos Comparable
17    public static < T extends Comparable< T > > T maximum( T x, T y, T z )
18    {
19       T max = x; // supõe que x é inicialmente o maior
20
21       if ( y.compareTo( max ) > 0 )
22          max = y; // y é o maior até agora
23
```

Somente objetos `Comparable` podem ser utilizados com esse método

Figura 21.5 | Método genérico `maximum` com um limite superior no seu parâmetro de tipo. (Parte 1 de 2.)


```
24          if ( z.compareTo( max ) > 0 )
25             max = z; // z é o maior
26
27          return max; // retorna o maior objeto
28       } // fim do método Maximum
29    } // fim da classe 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
```

Figura 21.5 | Método genérico `maximum` com um limite superior no seu parâmetro de tipo. (Parte 2 de 2.)

- A seção de parâmetro de tipo especifica que `T` estende `Comparable<T>` — somente objetos das classes que implementam a interface `Comparable<T>` podem ser utilizados com esse método.
- `Comparable` é conhecido como o **limite superior** do parâmetro de tipo.
- Por padrão, `Object` é o limite superior.
- Declarações do parâmetro de tipo que limitam o parâmetro sempre utilizam a palavra-chave `extends` independentemente de o parâmetro de tipo estender uma classe ou implementar uma interface.
- A restrição à utilização de objetos `Comparable<T>` é importante, pois nem todos os objetos podem ser comparados.

- Quando o compilador traduz o método genérico `maximum` em bytecodes Java, ele utiliza erasure para substituir os parâmetros de tipo por tipos reais.
- Todos os parâmetros de tipo são substituídos pelo limite superior do parâmetro de tipo, que é especificado na seção do parâmetro de tipo.
- Quando o compilador substitui as informações do parâmetro de tipo pelo tipo do limite superior na declaração do método, ele também insere operações explícitas de coerção na frente de cada chamada de método para assegurar que o valor retornado é do tipo esperado pelo chamador.

```
1   public static Comparable maximum(Comparable x, Comparable y, Comparable z)
2   {
3      Comparable max = x; // supõe que x é inicialmente o maior
4
5      if ( y.compareTo( max ) > 0 )
6         max = y; // y é o maior até agora
7
8      if ( z.compareTo( max ) > 0 )
9         max = z; // z é o maior
10
11     return max; // retorna o maior objeto
12  } // fim do método Maximum
```

Figura 21.6 | O método genérico `maximum` depois de a erasure ser realizada pelo compilador.

21.5 Sobrecarregando métodos genéricos

- Um método genérico pode ser sobrecarregado.
- Uma classe pode fornecer dois ou mais métodos genéricos que especificam o mesmo nome de método, mas diferentes parâmetros de método.
- Um método genérico também pode ser sobrecarregado por métodos não genéricos.
- Quando o compilador encontra uma chamada de método, ele procura a declaração de método que corresponde mais precisamente ao nome de método e aos tipos de argumentos especificados na chamada.

21.6 Classes genéricas

- O conceito de uma estrutura de dados, como uma pilha, pode ser entendido independentemente do tipo de elemento que ela manipula.
- Classes genéricas fornecem um meio de descrever o conceito de uma pilha (ou de qualquer outra classe) de uma maneira independente do tipo.
- Essas classes são conhecidas como **classes parametrizadas** ou **tipos parametrizados** porque aceitam um ou mais parâmetros.

```
1  // Figura 21.7: Stack.java
2  // Declaração da classe genérica Stack.
3  import java.util.ArrayList;
4
5  public class Stack< T >
6  {
7     private ArrayList< T > elements; // ArrayList armazena elementos de pilha
8
9     // construtor sem argumento cria uma pilha do tamanho padrão
10    public Stack()
11    {
12       this( 10 ); // tamanho padrão da pilha
13    } // fim do construtor sem argumentos da classe Stack
14
15    // construtor cria uma pilha com o número especificado de elementos
16    public Stack( int capacity )
17    {
18       int initCapacity = capacity > 0 ? capacity : 10; // valida
19       elements = new ArrayList< T >( initCapacity ); // cria a ArrayList
20    } // fim do construtor Stack de um argumento
21
```

Figura 21.7 | Declaração da classe genérica Stack. (Parte 1 de 2.)


```
22       // insere o elemento na pilha
23       public void push( T pushValue )
24       {
25          elements.add( pushValue ); // insere pushValue na Stack
26       } // fim do método push
27
28       // retorna o elemento superior se não estiver vazia; do contrário lança uma EmptyStackException
29       public T pop()
30       {
31          if ( elements.isEmpty() ) // se a pilha estiver vazia
32             throw new EmptyStackException( "Stack is empty, cannot pop" );
33
34          // remove e retorna o elemento superior da Stack
35          return elements.remove( elements.size() - 1 );
36       } // fim do método pop
37    } // fim da classe Stack < T >
```

Figura 21.7 | Declaração da classe genérica `Stack`. (Parte 2 de 2.)

```
1  // Figura 21.8: EmptyStackException.java
2  // Declaração da classe EmptyStackException.
3  public class EmptyStackException extends RuntimeException
4  {
5     // construtor sem argumento
6     public EmptyStackException()
7     {
8        this( "Stack is empty" );
9     } // fim do construtor sem argumentos de EmptyStackException
10
11    // construtor de um argumento
12    public EmptyStackException( String message )
13    {
14       super( message );
15    } // fim do construtor de EmptyStackException de um argumento
16 } // fim da classe EmptyStackException
```

Figura 21.8 | Declaração de classe `EmptyStackException`.


```
1  // Figura 21.9: Stacktest.java
2  // Programa de teste da classe genérica Stack.
3
4  public class StackTest
5  {
6     public static void main( String[] args )
7     {
8        double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
9        int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10
11       // Cria uma Stack< Double > e uma Stack< Integer
12       Stack< Double > doubleStack = new Stack< Double >( 5 );
13       Stack< Integer > integerStack = new Stack< Integer >();
14
15       // insere os elementos de doubleElements em doubleStack
16       testPushDouble( doubleStack, doubleElements );
17       testPopDouble( doubleStack ); // remove de doubleStack
18
19       // insere os elementos de integerElements em integerStack
20       testPushInteger( integerStack, integerElements );
21       testPopInteger( integerStack ); // remove de integerStack
22    } // fim de main
```

Figura 21.9 | Programa de teste da classe genérica `Stack`. (Parte 1 de 5.)

```
23
24       // testa o método push com a pilha de doubles
25       private static void testPushDouble(
26          Stack< Double > stack, double[] values )
27       {
28          System.out.println( "\nPushing elements onto doubleStack" );
29
30          // insere elementos na Stack
31          for ( double value : values )
32          {
33             System.out.printf( "%.1f ", value );
34             stack.push( value ); // insere em doubleStack
35          } // for final
36       } // fim do método testPushDouble
37
38       // testa o método pop com a pilha de doubles
39       private static void testPopDouble( Stack< Double > stack )
40       {
41          // remove elementos da pilha
42          try
43          {
44             System.out.println( "\nPopping elements from doubleStack" );
45             double popValue; // armazena o elemento removido da pilha
46
```

Figura 21.9 | Programa de teste da classe genérica `Stack`. (Parte 2 de 5.)

```
47            // remove todos os elementos da Stack
48            while ( true )
49            {
50               popValue = stack.pop(); // remove de doubleStack
51               System.out.printf( "%.1f ", popValue );
52            } // fim do while
53         } // fim do try
54         catch( EmptyStackException emptyStackException )
55         {
56            System.err.println();
57            emptyStackException.printStackTrace();
58         } // fim da captura de EmptyStackException
59      } // fim do método testPopDouble
60
61      // testa o método push com a pilha de integers
62      private static void testPushInteger(
63         Stack< Integer > stack, int[] values )
64      {
65         System.out.println( "\nPushing elements onto integerStack" );
66
67         // insere elementos na Stack
68         for ( int value : values )
69         {
```

Figura 21.9 | Programa de teste da classe genérica `Stack`. (Parte 3 de 5.)

```
70              System.out.printf( "%d ", value );
71              stack.push( value ); // insere em integerStack
72           } // for final
73        } // fim do método testPushInteger
74
75        // testa o método pop com a pilha de integers
76        private static void testPopInteger( Stack< Integer > stack )
77        {
78           // remove os elementos da pilha
79           try
80           {
81              System.out.println( "\nPopping elements from integerStack" );
82              int popValue; // armazena o elemento removido da pilha
83
84              // remove todos os elementos da Stack
85              while ( true )
86              {
87                 popValue = stack.pop(); // remove de intStack
88                 System.out.printf( "%d ", popValue );
89              } // fim do while
90           } // fim do try
91           catch( EmptyStackException emptyStackException )
92           {
```

Figura 21.9 | Programa de teste da classe genérica `Stack`. (Parte 4 de 5.)


```
93             System.err.println();
94             emptyStackException.printStackTrace();
95          } // fim da captura de EmptyStackException
96       } // fim do método testPopInteger
97    } // fim da classe StackTest
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest.testPopDouble(StackTest.java:50)
        at StackTest.main(StackTest.java:17)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest.testPopInteger(StackTest.java:87)
    at StackTest.main(StackTest.java:21)
```

Figura 21.9 | Programa de teste da classe genérica `Stack`. (Parte 5 de 5.)

- O código nos métodos `testPushDouble` e `testPushInteger` do exemplo anterior é quase idêntico a inserir valores em uma `Stack<Double>` ou uma `Stack<Integer>`, respectivamente, e o código nos métodos `testPopDouble` e `testPopInteger` é quase idêntico a remover valores de uma `Stack<Double>` ou uma `Stack<Integer>`, respectivamente.
- Isso representa outra oportunidade de utilizar métodos genéricos.


```
1  // Figura 21.10: StackTest2.java
2  // Passando objetos Stack genéricos para métodos genéricos.
3  public class StackTest2
4  {
5     public static void main( String[] args )
6     {
7        Double [] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
8        Integer [] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
9
10       // Cria uma Stack< Double > e uma Stack< Integer
11       Stack< Double > doubleStack = new Stack< Double >( 5 );
12       Stack< Integer > integerStack = new Stack< Integer >();
13
14       // insere os elementos de doubleElements em doubleStack
15       testPush( "doubleStack", doubleStack, doubleElements );
16       testPop( "doubleStack", doubleStack ); // remove de doubleStack
17
18       // insere os elementos de integerElements em integerStack
19       testPush( "integerStack", integerStack, integerElements );
20       testPop( "integerStack", integerStack ); // remove de integerStack
21    } // fim de main
22
```

Figura 21.10 | Passando objetos `Stack` genéricos para métodos genéricos. (Parte 1 de 4.)

```
23      // método genérico testPush insere elementos em uma Stack
24      public static < T > void testPush( String name , Stack< T > stack,
25         T[] elements )
26      {
27         System.out.printf( "\nPushing elements onto %s\n", name );
28
29         // insere elementos na Stack
30         for ( T element : elements )
31         {
32            System.out.printf( "%s ", element );
33            stack.push( element ); // insere o elemento na pilha
34         } // for final
35      } // fim do método testPush
36
37      // método genérico testPop remove elementos de uma Stack
38      public static < T > void testPop( String name, Stack< T > stack )
39      {
40         // remove elementos da pilha
41         try
42         {
43            System.out.printf( "\nPopping elements from %s\n", name );
44            T popValue; // armazena o elemento removido da pilha
45
```

Figura 21.10 | Passando objetos `Stack` genéricos para métodos genéricos. (Parte 2 de 4.)


```
46              // remove todos os elementos da Stack
47              while ( true )
48              {
49                 popValue = stack.pop();
50                 System.out.printf( "%s ", popValue );
51              } // fim do while
52           } // fim do try
53           catch( EmptyStackException emptyStackException )
54           {
55              System.out.println();
56              emptyStackException.printStackTrace();
57           } // fim da captura de EmptyStackException
58        } // fim do método testPop
59     } // fim da classe StackTest2
```

Figura 21.10 | Passando objetos `Stack` genéricos para métodos genéricos. (Parte 3 de 4.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest2.testPop(StackTest2.java:50)
        at StackTest2.main(StackTest2.java:17)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest2.testPop(StackTest2.java:50)
    at StackTest2.main(StackTest2.java:21
```

Figura 21.10 | Passando objetos `Stack` genéricos para métodos genéricos. (Parte 4 de 4.)

21.7 Tipos brutos

- Também é possível instanciar uma classe genérica `Stack` sem especificar um argumento de tipo, como a seguir:

```
// nenhum argumento de tipo especificado
Stack objectStack = new Stack( 5 );
```

- Dizemos que `objectStack` tem um **tipo bruto**.
- O compilador utiliza implicitamente o tipo `Object` por toda a classe genérica para cada argumento de tipo.
- A instrução precedente cria uma `Stack` que pode armazenar objetos de qualquer tipo.
- Isso é importante para retrocompatibilidade com versões anteriores do Java.
- Operações de tipos brutos são perigosas e podem levar a exceções.


```
1  // Figura 21.11: RawTypeTest.java
2  // Programa de teste de tipos brutos.
3  public class RawTypeTest
4  {
5     public static void main( String[] args )
6     {
7        Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
8        Integer[] integerElements = {  1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
9
10       // Pilha de tipos brutos atribuídos à classe Stack da variável de tipos brutos
11       Stack rawTypeStack1 = new Stack( 5 );
12
13       // Stack< Double > atribuído à Stack da variável de tipos brutos
14       Stack rawTypeStack2 = new Stack< Double >( 5 );
15
16       // Pilha de tipos crus atribuídos à variável Stack< Integer >
17       Stack< Integer > integerStack = new Stack( 10 );
18
19       testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
20       testPop( "rawTypeStack1", rawTypeStack1 );
21       testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
22       testPop( "rawTypeStack2", rawTypeStack2 );
23       testPush( "integerStack", integerStack, integerElements );
24       testPop( "integerStack", integerStack );
25    } // fim de main
```

Figura 21.11 | Programa de teste de tipos brutos. (Parte 1 de 4.)

```
26
27       // método genérico insere elementos na pilha
28       public static < T > void testPush( String name, Stack< T > stack,
29          T[] elements )
30       {
31          System.out.printf( "\nPushing elements onto %s\n", name );
32
33          // insere elementos na Stack
34          for ( T element : elements )
35          {
36             System.out.printf( "%s ", element );
37             stack.push( element ); // insere o elemento na pilha
38          } // for final
39       } // fim do método testPush
40
41       // método genérico testPop remove elementos da pilha
42       public static < T > void testPop( String name, Stack< T > stack )
43       {
44          // remove elementos da pilha
45          try
46          {
47             System.out.printf( "\nPopping elements from %s\n", name );
48             T popValue; // armazena o elemento removido da pilha
49
```

Figura 21.11 | Programa de teste de tipos brutos. (Parte 2 de 4.)

```
50             // remove elementos da Stack
51             while ( true )
52             {
53                popValue = stack.pop(); // remove da pilha
54                System.out.printf( "%s ", popValue );
55             } // fim do while
56          } // fim do try
57          catch( EmptyStackException emptyStackException )
58          {
59             System.out.println();
60             emptyStackException.printStackTrace();
61          } // fim da captura de EmptyStackException
62       } // fim do método testPop
63    } // fim da classe RawTypeTest
```

```
Pushing elements onto rawTypeStack1
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack1
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:20)
```

Figura 21.11 | Programa de teste de tipos brutos. (Parte 3 de 4.)


```
Pushing elements onto rawTypeStack2
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack2
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:22)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:24)
```

Figura 21.11 | Programa de teste de tipos brutos. (Parte 4 de 4.)

- A Figura 21.12 mostra as mensagens de alerta geradas pelo compilador quando o arquivo `RawTypeTest.java` (Figura 21.11) é compilado com a opção `-Xlint:unchecked`, que fornece informações adicionais sobre operações potencialmente perigosas em código que utiliza genéricos.


```
RawTypeTest.java:17: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Integer>
      Stack< Integer > integerStack = new Stack( 10 );
                                      ^
RawTypeTest.java:19: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Double>
      testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
                                 ^
RawTypeTest.java:19: warning: [unchecked] unchecked method invocation:
<T>testPush(java.lang.String,Stack<T>,T[]) in RawTypeTest is applied to
(java.lang.String,Stack,java.lang.Double[])
      testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
               ^
RawTypeTest.java:20: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<T>
      testPop( "rawTypeStack1", rawTypeStack1 );
                                ^
RawTypeTest.java:20: warning: [unchecked] unchecked method invocation:
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
      testPop( "rawTypeStack1", rawTypeStack1 );
              ^
```

Figura 21.12 | Mensagens de alerta do compilador. (Parte 1 de 2.)


```
RawTypeTest.java:21: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Double>
      testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
                                 ^
RawTypeTest.java:21: warning: [unchecked] unchecked method invocation:
<T>testPush(java.lang.String,Stack<T>,T[]) in RawTypeTest is applied to
(java.lang.String,Stack,java.lang.Double[])
      testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
              ^
RawTypeTest.java:22: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<T>
      testPop( "rawTypeStack2", rawTypeStack2 );
                                ^
RawTypeTest.java:22: warning: [unchecked] unchecked method invocation:
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
      testPop( "rawTypeStack2", rawTypeStack2 );
             ^
9 warnings
```

Figura 21.12 | Mensagens de alerta do compilador. (Parte 2 de 2.)

21.8 Curingas em métodos que aceitam parâmetros de tipo

- Nessa seção, apresentamos um conceito poderoso sobre genéricos conhecido como **curingas.**
- Suponha que você quer implementar um método genérico `sum` que soma os números em uma `ArrayList`.
- Você começaria inserindo os números na coleção.
- Os números passariam por um autoboxing como objetos das classes empacotadoras de tipo — qualquer valor `int` passaria por um autoboxing como um objeto `Integer` e qualquer valor `double` passaria por um autoboxing como um objeto `Double`.
- Gostaríamos de poder somar todos os números na `ArrayList` independentemente dos seus tipos.
- Por essa razão, declararemos a `ArrayList` com o argumento de tipo `Number`, que é a superclasse de `Integer` e `Double`.
- Além disso, o método `sum` receberá um parâmetro do tipo `ArrayList<Number>` e somará seus elementos.

```
1  // Figura 21.13: TotalNumbers.java
2  // Somando os números em uma ArrayList<Number>.
3  import java.util.ArrayList;
4
5  public class TotalNumbers
6  {
7     public static void main( String[] args )
8     {
9        // cria, inicializa e gera saída de ArrayList de números contendo
10       // tanto Integers como Doubles e então exibe o total dos elementos
11       Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
12       ArrayList< Number > numberList = new ArrayList< Number >();
13
14       for ( Number element : numbers )
15          numberList.add( element ); // insere cada número na numberList
16
17       System.out.printf( "numberList contains: %s\n", numberList );
18       System.out.printf( "Total of the elements in numberList: %.1f\n",
19          sum( numberList ) );
20    } // fim de main
21
```

Figura 21.13 | Somando os números em uma `ArrayList<Number>`. (Parte 1 de 2.)


```
22      // calcula o total de elementos em ArrayList
23      public static double sum( ArrayList< Number > list )
24      {
25         double total = 0; // inicializa o total
26
27         // calcula a soma
28         for ( Number element : list )
29            total += element.doubleValue();
30
31         return total;
32      } // fim do método sum
33   } // fim da classe TotalNumbers
```

Não pode receber `ArrayLists` que especificam argumentos de tipo que são subclasses de `Number`

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Figura 21.13 | Somando os números em uma `ArrayList<Number>`. (Parte 2 de 2.)

- No método `sum`:
- A instrução `for` atribui cada `Number` na `ArrayList` à variável `element` e utiliza o **método Number doubleValue** para obter o valor primitivo subjacente de `Number` como um valor `double`.
- O resultado é adicionado a `total`.
- Quando o loop termina, o método retorna o `total`.

- Considerando-se que o método `sum` pode somar os elementos de uma `ArrayList` de `Number`s, você poderia supor que o método também funcionaria para `ArrayList`s que contêm elementos de somente um tipo numérico, como `ArrayList<Integer>`.
- A classe modificada `TotalNumbers` para criar uma `ArrayList` de `Integer`s e passá-la para o método `sum`.
- Ao compilar o programa, o compilador emite a mensagem de erro a seguir:
  - `sum(java.util.ArrayList<java.lang.Number>) em TotalNumbersErrors não pode ser aplicado a (java.util.ArrayList<java.lang.Integer>)`
- Embora `Number` seja a superclasse de `Integer`, o compilador não considera o tipo parametrizado `ArrayList<Number>` como uma superclasse de `ArrayList<Integer>`.
- Se fosse, então, toda operação que realizássemos em `ArrayList<Number>` também funcionaria em uma `ArrayList<Integer>`.

- Para criar uma versão mais flexível do método `sum` que possa somar os elementos de qualquer `ArrayList` contendo elementos de qualquer subclasse de `Number`, utilizamos **argumentos do tipo curinga.**
- Os curingas permitem especificar parâmetros de método, valores de retorno, variáveis ou campos e assim por diante, que atuam como supertipos ou subtipos de tipos parametrizados.
- Na Figura 21.14, o parâmetro do método `sum` é declarado na linha 50 com o tipo:
  - `ArrayList< ? estende Number >`
- Um argumento do tipo curinga é indicado por um ponto de interrogação (?), que por si só representa um "tipo desconhecido".
- Nesse caso, o curinga estende a classe `Number`, o que significa que o curinga tem um limite superior de `Number`.
- Portanto, o argumento de tipo desconhecido deve ser `Number` ou uma subclasse de `Number`.


```
1  // Figura 21.14: WildcardTest.java
2  // Programa de teste de curinga.
3  import java.util.ArrayList;
4
5  public class WildcardTest
6  {
7     public static void main( String[] args )
8     {
9        // cria, inicializa e gera saída de ArrayList de Integers, então
10       // exibe o total dos elementos
11       Integer[] integers = { 1, 2, 3, 4, 5 };
12       ArrayList< Integer > integerList = new ArrayList< Integer >();
13
14       // insere elementos na integerList
15       for ( Integer element : integers )
16          integerList.add( element );
17
18       System.out.printf( "integerList contains: %s\n", integerList );
19       System.out.printf( "Total of the elements in integerList: %.0f\n\n",
20          sum( integerList ) );
21
22       // cria, inicializa e gera saída do ArrayList de Doubles, então
23       // exibe o total dos elementos
24       Double[] doubles = { 1.1, 3.3, 5.5 };
```

Figura 21.14 | Programa de teste de curinga genérico. (Parte 1 de 3.)

```
25          ArrayList< Double > doubleList = new ArrayList< Double >();
26
27          // insere elementos na doubleList
28          for ( Double element : doubles )
29             doubleList.add( element );
30
31          System.out.printf( "doubleList contains: %s\n", doubleList );
32          System.out.printf( "Total of the elements in doubleList: %.1f\n\n",
33             sum( doubleList ) );
34
35          // cria, inicializa e gera saída de ArrayList de números contendo
36          // tanto Integers como Doubles e então exibe o total dos elementos
37          Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
38          ArrayList< Number > numberList = new ArrayList< Number >();
39
40          // insere elementos na numberList
41          for ( Number element : numbers )
42             numberList.add( element );
43
44          System.out.printf( "numberList contains: %s\n", numberList );
45          System.out.printf( "Total of the elements in numberList: %.1f\n",
46             sum( numberList ) );
47       } // fim de main
```

Figura 21.14 | Programa de teste de curinga genérico. (Parte 2 de 3.)

```
48
49       // soma os elementos; utilizando um curinga no parâmetro ArrayList
50       public static double sum( ArrayList< ? extends Number > list )
51       {
52          double total = 0; // inicializa o total
53
54          // calcula a soma
55          for ( Number element : list )
56             total += element.doubleValue();
57
58          return total;
59       } // fim do método sum
60    } // fim da classe WildcardTest
```

O método agora pode receber ArrayLists de qualquer subclasse Number

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15

doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Figura 21.14 | Programa de teste de curinga genérico. (Parte 3 de 3.)

- Como o curinga (`?`) no cabeçalho do método não especifica um nome de parâmetro de tipo, você não pode utilizá-lo como um nome de tipo por todo o corpo do método (isto é, não pode substituir `Number` por `?` na linha 55).
- Você pode, porém, declarar o método `sum` dessa maneira:

```
public static <T estende Number> double
   sum( ArrayList< T > list )
```

- Isso permite ao método receber uma `ArrayList` que contém elementos de qualquer subclasse `Number`.
- Você pode então utilizar o parâmetro de tipo `T` por todo o corpo do método.
- Se o curinga for especificado sem um limite superior, somente os métodos do tipo `Object` podem ser invocados nos valores do tipo curinga.
- Além disso, métodos que utilizam curingas nos seus argumentos de tipo do parâmetro não podem ser utilizados para adicionar elementos a uma coleção referenciada pelo parâmetro.

Erro comum de programação 21.4

Utilizar um curinga na seção de parâmetro de tipo de um método ou utilizar um curinga como um tipo explícito de uma variável no corpo do método é um erro de sintaxe.