**Tipos genéricos**

Um *tipo genérico* é uma classe genérica ou interface que é parametrizada sobre tipos. A classe Box a seguir será modificada para demonstrar o conceito.

**Uma classe de caixa simples**

Comece examinando uma classe Box não genérica que opera em objetos de qualquer tipo. Ele só precisa fornecer dois métodos: set, que adiciona um objeto à caixa, e get, que o recupera:

public class Box {

private Object object;

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

public Object get() { return object; }

}

Como seus métodos aceitam ou retornam um Objeto, você é livre para passar o que quiser, desde que não seja um dos tipos primitivos. Não há como verificar, em tempo de compilação, como a classe é usada. Uma parte do código pode colocar um Integer na caixa e esperar obter Integers fora dele, enquanto outra parte do código pode passar por engano em uma String, resultando em um erro de tempo de execução.

**Uma versão genérica da classe Box**

Uma *classe genérica* é definida com o seguinte formato:

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

A seção de parâmetro de tipo, delimitada por colchetes angulares (<>), segue o nome da classe. Ele especifica os *parâmetros* de tipo (também chamados de *variáveis de tipo*) T1, T2, ..., e Tn.

Para atualizar a classe Box para usar genéricos, crie uma declaração de *tipo genérica* alterando o código "public class Box" para "public class Box<T>". Isso introduz a variável de tipo, T, que pode ser usada em qualquer lugar dentro da classe.

Com essa mudança, a classe Box se torna:

/\*\*

\* 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; }

}

Como você pode ver, todas as ocorrências de Object são substituídas por T. Uma variável de tipo pode ser qualquer tipo **não primitivo** que você especificar: qualquer tipo de classe, qualquer tipo de interface, qualquer tipo de matriz ou até mesmo outra variável de tipo.

Essa mesma técnica pode ser aplicada para criar interfaces genéricas.

**Convenções de nomenclatura de parâmetros de tipo**

Por convenção, os nomes dos parâmetros de tipo são letras maiúsculas simples. Isso contrasta fortemente com as convenções de [nomenclatura](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html#naming) de variáveis que você já conhece, e com uma boa razão: sem essa convenção, seria difícil dizer a diferença entre uma variável de tipo e uma classe comum ou nome de interface.

Os nomes de parâmetros de tipo mais usados são:

* E - Elemento (utilizado extensivamente pelo Java Collections Framework)
* K - Chave
* N - Número
* T - Tipo
* V - Valor
* S, U, V etc. - 2º, 3º, 4º tipos

Você verá esses nomes usados em toda a API Java SE e no restante desta lição.

**Invocando e instanciando um tipo genérico**

Para fazer referência à classe Box genérica de dentro do seu código, você deve executar uma *chamada de tipo genérico*, que substitui T por algum valor concreto, como Integer:

Box<Integer> integerBox;

Você pode pensar em uma invocação de tipo genérico como sendo semelhante a uma chamada de método comum, mas em vez de passar um argumento para um método, você está passando um *argumento de tipo* — Integer neste caso — para a própria classe Box.

**Terminologia do parâmetro de tipo e do argumento de tipo:** Muitos desenvolvedores usam os termos "parâmetro de tipo" e "argumento de tipo" alternadamente, mas esses termos não são os mesmos. Ao codificar, fornece-se argumentos de tipo para criar um tipo parametrizado. Portanto, o T em Foo<T> é um parâmetro de tipo e o String em Foo<String> f é um argumento de tipo. Esta lição observa essa definição ao usar esses termos.

Como qualquer outra declaração de variável, esse código não cria um novo objeto Box. Ele simplesmente declara que integerBox manterá uma referência a uma "Caixa de Inteiro", que é como Box<Integer> é lido.

Uma invocação de um tipo genérico é geralmente conhecida como um *tipo parametrizado*.

Para instanciar essa classe, use a nova palavra-chave, como de costume, mas coloque <Integer> entre o nome da classe e o parêntese:

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

**O Diamante**

No Java SE 7 e posterior, é possível substituir os argumentos de tipo necessários para invocar o construtor de uma classe genérica por um conjunto vazio de argumentos de tipo (<>), desde que o compilador possa determinar, ou inferir, os argumentos de tipo do contexto. Este par de colchetes angulares, <>, é informalmente chamado *de diamante*. Por exemplo, você pode criar uma instância de Box<Integer> com a seguinte instrução:

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

Para obter mais informações sobre notação de diamante e inferência de tipo, consulte [Inferência de tipo](https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html).

**Vários parâmetros de tipo**

Como mencionado anteriormente, uma classe genérica pode ter vários parâmetros de tipo. Por exemplo, a classe genérica OrderedPair, que implementa a interface genérica Pair:

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; }

}

As instruções a seguir criam duas instanciações da classe OrderedPair:

Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);

Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");

O código, novo OrderedPair<String, Integer>, instancia K como uma String e V como um Integer. Portanto, os tipos de parâmetro do construtor OrderedPair são String e Integer, respectivamente. Devido ao [autoboxing](https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html" \t "_top), é válido passar um String e um int para a classe.

Como mencionado em [The Diamond](https://docs.oracle.com/javase/tutorial/java/generics/types.html#diamond), como um compilador Java pode inferir os tipos K e V a partir da declaração OrderedPair<String, Integer>, essas instruções podem ser encurtadas usando a notação diamond:

OrderedPair<String, Integer> p1 = new OrderedPair**<>**("Even", 8); 🡨passando por parâmetro

OrderedPair<String, String> p2 = new OrderedPair**<>**("hello", "world");

Para criar uma interface genérica, siga as mesmas convenções que para criar uma classe genérica.

**Tipos parametrizados**

Você também pode substituir um parâmetro type (ou seja, K ou V) por um tipo parametrizado (ou seja, List<String>). Por exemplo, usando o exemplo OrderedPair<K, V>:

OrderedPair<String, **Box<Integer>**> p = new OrderedPair<>("primes", new Box<Integer>(...));

# Tipos brutos = Raw Type

Um tipo bruto é o nome de uma classe genérica ou interface sem quaisquer argumentos de tipo. Por exemplo, dada a classe Box genérica:

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...

}

Para criar um tipo parametrizado de Box<T>, forneça um argumento de tipo real para o parâmetro de tipo formal T:

Box<Integer> intBox = new Box<>();

Se o argumento de tipo real for omitido, você criará um tipo bruto de Box<T>:

Box rawBox = new Box();

Portanto, Box é o tipo bruto do tipo genérico Box<T>. No entanto, uma classe não genérica ou tipo de interface não é um tipo bruto.

Os tipos brutos aparecem no código herdado porque muitas classes de API (como as classes Collections) não eram genéricas antes do JDK 5.0. Ao usar tipos brutos, você essencialmente obtém um comportamento pré-genérico — uma caixa fornece objetoss. Para compatibilidade com versões anteriores, a atribuição de um tipo parametrizado ao seu tipo bruto é permitida:

Box<String> stringBox = new Box<>();

Box rawBox = stringBox; // OK

Mas se você atribuir um tipo bruto a um tipo parametrizado, receberá um aviso:

Box rawBox = new Box(); // rawBox is a raw type of Box<T>

Box<Integer> intBox = rawBox; // warning: unchecked conversion

Você também receberá um aviso se usar um tipo bruto para invocar métodos genéricos definidos no tipo genérico correspondente:

Box<String> stringBox = new Box<>();

Box rawBox = stringBox;

rawBox.set(8); // warning: unchecked invocation to set(T)

O aviso mostra que os tipos brutos ignoram as verificações de tipo genéricas, adiando a captura de código não seguro para tempo de execução. Portanto, você deve evitar o uso de tipos brutos.

A seção [Type Erasure](https://docs.oracle.com/javase/tutorial/java/generics/erasure.html" \t "_top) tem mais informações sobre como o compilador Java usa tipos brutos.

## Mensagens de erro não verificadas

Como mencionado anteriormente, ao misturar código herdado com código genérico, você pode encontrar mensagens de aviso semelhantes à seguinte:

Note: Example.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

Isso pode acontecer ao usar uma API mais antiga que opera em tipos brutos, conforme mostrado no exemplo a seguir:

public class WarningDemo {

public static void main(String[] args){

Box<Integer> bi;

bi = createBox();

}

static Box createBox(){

return new Box();

}

}

O termo "não marcado" significa que o compilador não dispõe de informações de tipo suficientes para efectuar todas as verificações de tipo necessárias para garantir a segurança do tipo. O aviso "desmarcado" está desabilitado, por padrão, embora o compilador dê uma dica. Para ver todos os avisos "desmarcados", recompile com -Xlint:unchecked.

Recompilar o exemplo anterior com -Xlint:unchecked revela as seguintes informações adicionais:

WarningDemo.java:4: warning: [unchecked] unchecked conversion

found : Box

required: Box<java.lang.Integer>

bi = createBox();

^

1 warning

Para desativar completamente os avisos não verificados, use o sinalizador -Xlint:-uncheck. A anotação @SuppressWarnings("não marcada") suprime avisos não verificados. Se você não estiver familiarizado com a sintaxe, consulte [Anotações](https://docs.oracle.com/javase/tutorial/java/annotations/index.html).@SuppressWarnings

# Métodos genéricos

Métodos genéricos são métodos que introduzem seus próprios parâmetros de tipo. Isso é semelhante a declarar um tipo genérico, mas o escopo do parâmetro type é limitado ao método em que ele é declarado. Métodos genéricos estáticos e não estáticos são permitidos, bem como construtores de classe genéricos.

A sintaxe de um método genérico inclui uma lista de parâmetros de tipo, dentro de colchetes angulares, que aparece antes do tipo de retorno do método. Para métodos genéricos estáticos, a seção de parâmetro type deve aparecer antes do tipo de retorno do método.

A classe Util inclui um método genérico, compare, que compara dois objetos Pair:

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());

}

}

public class Pair<K, V> {

private K key;

private V value;

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

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; }

}

A sintaxe completa para invocar esse método seria:

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

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

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

O tipo foi explicitamente fornecido, conforme mostrado em negrito. Geralmente, isso pode ser deixado de fora e o compilador irá inferir o tipo que é necessário:

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

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

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

Esse recurso, conhecido como inferência de tipo, permite que você invoque um método genérico como um método comum, sem especificar um tipo entre colchetes angulares. Este tópico é discutido mais detalhadamente na seção a seguir, [Inferência de tipo](https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html).

# Parâmetros de tipo limitado

Pode haver momentos em que você deseja restringir os tipos que podem ser usados como argumentos de tipo em um tipo parametrizado. Por exemplo, um método que opera em números pode querer aceitar apenas instâncias de ou suas subclasses. É para isso que servem *os parâmetros de tipo limitado*.Number

Para declarar um parâmetro de tipo limitado, liste o nome do parâmetro type, seguido da palavra-chave, seguido de seu *limite superior*, que neste exemplo é . Note que, neste contexto, é usado em um sentido geral para significar "estende" (como em classes) ou "implementa" (como em interfaces).extendsNumberextends

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!**

}

}

Ao modificar nosso método genérico para incluir esse parâmetro de tipo limitado, a compilação agora falhará, já que nossa invocação de ainda inclui um :inspectString

Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot

be applied to (java.lang.String)

integerBox.inspect("10");

^

1 error

Além de limitar os tipos que você pode usar para instanciar um tipo genérico, os parâmetros de tipo limitado permitem que você invoque métodos definidos nos limites:

public class NaturalNumber<T extends Integer> {

private T n;

public NaturalNumber(T n) { this.n = n; }

public boolean isEven() {

return **n.intValue()** % 2 == 0;

}

// ...

}

O método isEven invoca o método intValue definido na classe Integer até n.

## Vários limites

O exemplo anterior ilustra o uso de um parâmetro type com um único limite, mas um parâmetro type pode ter vários limites:

<T extends B1 & B2 & B3>

Uma variável de tipo com vários limites é um subtipo de todos os tipos listados no limite. Se um dos limites for uma classe, ele deve ser especificado primeiro. Por exemplo:

Class A { /\* ... \*/ }

interface B { /\* ... \*/ }

interface C { /\* ... \*/ }

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

Se o limite A não for especificado primeiro, você obterá um erro em tempo de compilação:

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

# Métodos genéricos e parâmetros de tipo delimitado

Parâmetros de tipo limitado são fundamentais para a implementação de algoritmos genéricos. Considere o método a seguir que conta o número de elementos em uma matriz T[] que são maiores do que um elemento especificado 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;

}

A implementação do método é direta, mas não compila porque o operador maior que (>) se aplica apenas a tipos primitivos como short, int, double, long, float, byte e char. Não é possível usar o operador > para comparar objetos. Para corrigir o problema, use um parâmetro de tipo delimitado pela interface Comparable<T>:

public interface Comparable<T> {

public int compareTo(T o);

}

O código resultante será:

public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {

int count = 0;

for (T e : anArray)

if (e.compareTo(elem) > 0)

++count;

return count;

}

# Genéricos, herança e subtipos

Como você já sabe, é possível atribuir um objeto de um tipo a um objeto de outro tipo, desde que os tipos sejam compatíveis. Por exemplo, você pode atribuir um Integer a um Object, já que Object é um dos supertipos de Integer:

Object someObject = new Object();

Integer someInteger = new Integer(10);

someObject = someInteger; // OK

Na terminologia orientada a objetos, isso é chamado de uma relação "é um". Como um Integer é um tipo de Object, a atribuição é permitida. Mas Integer também é um tipo de Número, então o seguinte código também é válido:

public void someMethod(Number n) { /\* ... \*/ }

someMethod(new Integer(10)); // OK

someMethod(new Double(10.1)); // OK

O mesmo acontece com os genéricos. Você pode executar uma chamada de tipo genérico, passando Number como seu argumento de tipo, e qualquer invocação subsequente de add será permitida se o argumento for compatível com Number:

Box<Number> box = new Box<Number>();

box.add(new Integer(10)); // OK

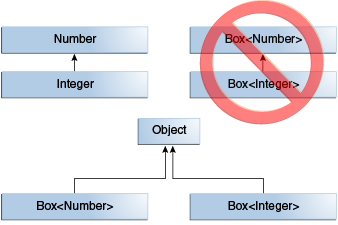
box.add(new Double(10.1)); // OK

Agora considere o seguinte método:

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

Que tipo de argumento aceita? Ao observar sua assinatura, você pode ver que ela aceita um único argumento cujo tipo é Box<Number>. Mas o que isso significa? Você tem permissão para passar em Box<Integer> ou Box<Double>, como você poderia esperar? A resposta é "não", porque Box<Integer> e Box<Double> não são subtipos de Box<Number>.

Este é um mal-entendido comum quando se trata de programação com genéricos, mas é um conceito importante para aprender.



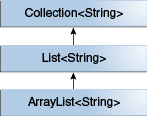
Box<Integer> não é um subtipo de Box<Number> mesmo que Integer seja um subtipo de Number.

**Nota:** Dado dois tipos concretos A e B (por exemplo, Número e Inteiro), MyClass<A> não tem relação com MyClass<B>, independentemente de A e B estarem ou não relacionados. O pai comum de MyClass<A> e MyClass<B> é Object.  
  
Para obter informações sobre como criar uma relação semelhante a um subtipo entre duas classes genéricas quando os parâmetros de tipo estão relacionados, consulte [Curingas e subtipagem](https://docs.oracle.com/javase/tutorial/java/generics/subtyping.html).

## Classes genéricas e subtipagem

Você pode subdigitar uma classe ou interface genérica estendendo-a ou implementando-a. A relação entre os parâmetros de tipo de uma classe ou interface e os parâmetros de tipo de outra são determinados pelas cláusulas extend e implements.

Usando as classes Collections como exemplo, ArrayList<E> implementa List<E> e List<E> estende Collection<E>. Portanto, ArrayList<String> é um subtipo de List<String>, que é um subtipo de Collection<String>. Contanto que você não varie o argumento type, a relação de subtyping é preservada entre os tipos.



Uma hierarquia de Coleções de exemplo

Agora imagine que queremos definir nossa própria interface de lista, PayloadList, que associa um valor opcional do tipo genérico P a cada elemento. Sua declaração pode ser assim:

interface PayloadList<E,P> extends List<E> {

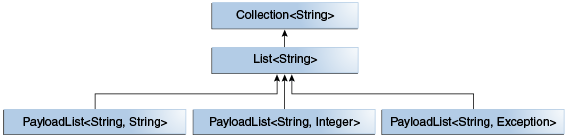
void setPayload(int index, P val);

...

}

As seguintes parametrizações de PayloadList são subtipos de List<String>:

* PayloadList<String,String>
* PayloadList<String,Inteiro>
* PayloadList<String,Exceção>



Uma hierarquia PayloadList de exemplo

# Inferência de tipo

A inferência de tipo é a capacidade de um compilador Java de examinar cada invocação de método e declaração correspondente para determinar o argumento de tipo (ou argumentos) que tornam a invocação aplicável. O algoritmo de inferência determina os tipos dos argumentos e, se disponível, o tipo que o resultado está sendo atribuído ou retornado. Finalmente, o algoritmo de inferência tenta encontrar o tipo mais específico que funciona com todos os argumentos.

Para ilustrar esse último ponto, no exemplo a seguir, a inferência determina que o segundo argumento que está sendo passado para o método pick é do tipo Serializable:

static <T> T pick(T a1, T a2) { return a2; }

Serializable s = pick("d", new ArrayList<String>());

## Inferência de Tipos e Métodos Genéricos

[Os Métodos Genéricos](https://docs.oracle.com/javase/tutorial/java/generics/methods.html) introduziram a inferência de tipos, que permite invocar um método genérico como faria com um método comum, sem especificar um tipo entre colchetes angulares. Considere o exemplo a seguir, [BoxDemo](https://docs.oracle.com/javase/tutorial/java/generics/examples/BoxDemo.java" \t "_blank), que requer a classe [Box](https://docs.oracle.com/javase/tutorial/java/generics/examples/Box.java):

public class BoxDemo {

public static <U> void addBox(U u,

java.util.List<Box<U>> boxes) { // Caixa de caixas, // ou lista de caixas

Box<U> box = new Box<>();

box.set(u);

boxes.add(box);

}

public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {

int counter = 0;

for (Box<U> box: boxes) {

U boxContents = box.get();

System.out.println("Box #" + counter + " contains [" +

boxContents.toString() + "]");

counter++;

}

}

public static void main(String[] args) {

java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =

new java.util.ArrayList<>();

BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);

BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);

BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);

BoxDemo.outputBoxes(listOfIntegerBoxes);

}

}

A seguir está a saída deste exemplo:

Box #0 contains [10]

Box #1 contains [20]

Box #2 contains [30]

O método genérico define um parâmetro de tipo chamado . Geralmente, um compilador Java pode inferir os parâmetros de tipo de uma chamada de método genérico. Consequentemente, na maioria dos casos, você não precisa especificá-los. Por exemplo, para invocar o método genérico, você pode especificar o parâmetro type com uma testemunha de tipo da seguinte maneira: addBoxUaddBox

BoxDemo.**<Integer>**addBox(Integer.valueOf(10), listOfIntegerBoxes);

Como alternativa, se você omitir a testemunha de tipo, um compilador Java inferirá automaticamente (a partir dos argumentos do método) que o parâmetro type é :Integer

BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);

## Inferência de tipo e instanciação de classes genéricas

Você pode substituir os argumentos de tipo necessários para invocar o construtor de uma classe genérica por um conjunto vazio de parâmetros de tipo (), desde que o compilador possa inferir os argumentos de tipo do contexto. Este par de colchetes angulares é informalmente chamado [de diamante](https://docs.oracle.com/javase/tutorial/java/generics/types.html#diamond).<>

Por exemplo, considere a seguinte declaração de variável:

Map<String, List<String>> myMap = new HashMap<String, List<String>>();

Você pode substituir o tipo parametrizado do construtor por um conjunto vazio de parâmetros de tipo (<>):

Map<String, List<String>> myMap = new HashMap<>();

Observe que para aproveitar a inferência de tipo durante a instanciação de classe genérica, você deve usar o diamante. No exemplo a seguir, o compilador gera um aviso de conversão não verificado porque o construtor se refere ao tipo bruto, não ao tipo:HashMap()HashMapMap<String, List<String>>

Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning

## Inferência de tipos e construtores genéricos de classes genéricas e não genéricas

Observe que os construtores podem ser genéricos (em outras palavras, declarar seus próprios parâmetros de tipo formais) em classes genéricas e não genéricas. Considere o seguinte exemplo:

class MyClass<X> {

<T> MyClass(T t) {

// ...

}

}

Considere a seguinte instanciação da classe:MyClass

new MyClass<Integer>("")

Essa instrução cria uma instância do tipo parametrizado ; A instrução especifica explicitamente o tipo para o parâmetro de tipo formal, , da classe genérica . Observe que o construtor para essa classe genérica contém um parâmetro de tipo formal, . O compilador infere o tipo para o parâmetro de tipo formal, , do construtor dessa classe genérica (porque o parâmetro real desse construtor é um objeto).MyClass<Integer>IntegerXMyClass<X>TStringTString

Compiladores de releases anteriores ao Java SE 7 são capazes de inferir os parâmetros de tipo reais de construtores genéricos, semelhantes aos métodos genéricos. No entanto, compiladores em Java SE 7 e posterior podem inferir os parâmetros de tipo reais da classe genérica que está sendo instanciada se você usar o diamante (). Considere o seguinte exemplo:<>

MyClass<Integer> myObject = new MyClass<>("");

Neste exemplo, o compilador infere o tipo para o parâmetro de tipo formal, , da classe genérica . Ele infere o tipo para o parâmetro de tipo formal, , do construtor dessa classe genérica.IntegerXMyClass<X>StringT

**Nota:** É importante observar que o algoritmo de inferência usa apenas argumentos de invocação, tipos de destino e, possivelmente, um tipo de retorno esperado óbvio para inferir tipos. O algoritmo de inferência não usa resultados posteriores no programa.

## Tipos de destino

O compilador Java aproveita a digitação de destino para inferir os parâmetros de tipo de uma chamada de método genérico. O tipo de destino de uma expressão é o tipo de dados que o compilador Java espera, dependendo de onde a expressão aparece. Considere o método , que é declarado da seguinte forma:Collections.emptyList

static <T> List<T> emptyList();

Considere a seguinte declaração de atribuição:

List<String> listOne = Collections.emptyList();

Esta declaração está esperando uma instância de ; Esse tipo de dados é o tipo de destino. Como o método retorna um valor de tipo , o compilador infere que o argumento type deve ser o valor . Isso funciona em Java SE 7 e 8. Como alternativa, você pode usar um tipo testemunha e especificar o valor da seguinte maneira:List<String>emptyListList<T>TStringT

List<String> listOne = Collections.<String>emptyList();

No entanto, isso não é necessário neste contexto. Mas era necessário em outros contextos. Considere o seguinte método:

void processStringList(List<String> stringList) {

// process stringList

}

Suponha que você queira invocar o método com uma lista vazia. No Java SE 7, a instrução a seguir não compila:processStringList

processStringList(Collections.emptyList());

O compilador Java SE 7 gera uma mensagem de erro semelhante à seguinte:

List<Object> cannot be converted to List<String>

O compilador requer um valor para o argumento type, portanto, ele começa com o valor . Consequentemente, a invocação de retorna um valor do tipo , que é incompatível com o método . Assim, no Java SE 7, você deve especificar o valor do valor do argumento type da seguinte maneira:TObjectCollections.emptyListList<Object>processStringList

processStringList(Collections.<String>emptyList());

Isso não é mais necessário no Java SE 8. A noção do que é um tipo de destino foi expandida para incluir argumentos de método, como o argumento para o método . Nesse caso, requer um argumento do tipo . O método retorna um valor de , portanto, usando o tipo de destino de , o compilador infere que o argumento type tem um valor de . Assim, no Java SE 8, a seguinte instrução compila:processStringListprocessStringListList<String>Collections.emptyListList<T>List<String>TString

processStringList(Collections.emptyList());

Consulte [Digitação de destino](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#target-typing) em [expressões do Lambda](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) para obter mais informações.

# Curingas

No código genérico, o ponto de interrogação (?), chamado de curinga, representa um tipo desconhecido. O curinga pode ser usado em uma variedade de situações: como o tipo de um parâmetro, campo ou variável local; às vezes como um tipo de retorno (embora seja melhor prática de programação para ser mais específico). O curinga nunca é usado como um argumento de tipo para uma chamada de método genérico, uma criação de instância de classe genérica ou um supertipo.

As seções a seguir discutem curingas com mais detalhes, incluindo curingas com limite superior, curingas com limite inferior e captura de curinga.

# Curingas delimitados superiores

Você pode usar um curinga com limite superior para relaxar as restrições em uma variável. Por exemplo, digamos que você queira escrever um método que funcione em List<Integer>, List<Double> e List<Number>; Você pode conseguir isso usando um curinga com limite superior.

Para declarar um curinga de limite superior, use o caractere curinga ('?'), seguido da palavra-chave extend, seguida de seu limite superior. Note que, neste contexto, extend é usado em um sentido geral para significar "estende" (como em classes) ou "implementa" (como em interfaces).

Para escrever o método que funciona em listas de Number e os subtipos de Number, como Integer, Double, e Float, você especificaria List<? estende Number>. O termo Lista<Número> é mais restritivo do que Lista<? estende Número> porque o primeiro corresponde apenas a uma lista do tipo Número, enquanto o segundo corresponde a uma lista do tipo Número ou a qualquer uma de suas subclasses.

Considere o seguinte método de processo:

public static void process(List**<? extends Foo>** list) { /\* ... \*/ }

O curinga de limite superior, <?, estende Foo>, onde Foo é qualquer tipo, combina Foo e qualquer subtipo de Foo. O método de processo pode acessar os elementos da lista como tipo Foo:

public static void process(List<? extends Foo> list) {

for (Foo elem : list) {

// ...

}

}

Na cláusula foreach, a variável elem itera sobre cada elemento da lista. Qualquer método definido na classe Foo agora pode ser usado no elem.

O método sumOfList retorna a soma dos números em uma lista:

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

double s = 0.0;

for (Number n : list)

s += n.doubleValue();

return s;

}

O código a seguir, usando uma lista de objetos Integer, imprime soma = 6,0:

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

System.out.println("sum = " + sumOfList(li));

Uma lista de valores Double pode usar o mesmo método sumOfList. O código a seguir imprime soma = 7,0:

List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);

System.out.println("sum = " + sumOfList(ld));

# Curingas ilimitados

O tipo curinga ilimitado é especificado usando o caractere curinga (?), por exemplo, List<?>. Isso é chamado de lista de tipo desconhecido. Há dois cenários em que um curinga ilimitado é uma abordagem útil:

* Se você estiver escrevendo um método que pode ser implementado usando a funcionalidade fornecida na classe Object.
* Quando o código está usando métodos na classe genérica que não dependem do parâmetro type. Por exemplo, List.size ou List.clear. Na verdade, Class<?> é tão frequentemente usado porque a maioria dos métodos em Class<T> não dependem de T.

Considere o seguinte método, printList:

public static void printList(List<Object> list) {

for (Object elem : list)

System.out.println(elem + " ");

System.out.println();

}

O objetivo de printList é imprimir uma lista de qualquer tipo, mas não consegue atingir esse objetivo — ele imprime apenas uma lista de instâncias de Object; ele não pode imprimir List<Integer>, List<String>, List<Double> e assim por diante, porque eles não são subtipos de List<Object>. Para escrever um método printList genérico, use List<?>:

public static void printList(List<?> list) {

for (Object elem: list)

System.out.print(elem + " ");

System.out.println();

}

Como para qualquer tipo concreto A, List<A> é um subtipo de List<?>, você pode usar printList para imprimir uma lista de qualquer tipo:

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

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

printList(li);

printList(ls);

**Nota:** O método [Arrays.asList](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html" \l "asList-T...-" \t "_blank) é usado em exemplos ao longo desta lição. Esse método de fábrica estático converte a matriz especificada e retorna uma lista de tamanho fixo.

É importante notar que List<Object> e List<?> não são a mesma coisa. Você pode inserir um Objeto, ou qualquer subtipo de Objeto, em uma Lista<Objeto>. Mas você só pode inserir null em uma List<?>. A seção Diretrizes para uso de curinga tem mais informações sobre como determinar que tipo de [curinga](https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html), se houver, deve ser usado em uma determinada situação.

# Curingas delimitados superiores

Você pode usar um curinga com limite superior para relaxar as restrições em uma variável. Por exemplo, digamos que você queira escrever um método que funcione em List<Integer>, List<Double> e List<Number>; Você pode conseguir isso usando um curinga com limite superior.

Para declarar um curinga de limite superior, use o caractere curinga ('?'), seguido da palavra-chave extend, seguida de seu limite superior. Note que, neste contexto, extend é usado em um sentido geral para significar "estende" (como em classes) ou "implementa" (como em interfaces).

Para escrever o método que funciona em listas de Number e os subtipos de Number, como Integer, Double, e Float, você especificaria List<? estende Number>. O termo Lista<Número> é mais restritivo do que Lista<? estende Número> porque o primeiro corresponde apenas a uma lista do tipo Número, enquanto o segundo corresponde a uma lista do tipo Número ou a qualquer uma de suas subclasses.

Considere o seguinte método de processo:

public static void process(List**<? extends Foo>** list) { /\* ... \*/ }

O curinga de limite superior, <?, estende Foo>, onde Foo é qualquer tipo, combina Foo e qualquer subtipo de Foo. O método de processo pode acessar os elementos da lista como tipo Foo:

public static void process(List<? extends Foo> list) {

for (Foo elem : list) {

// ...

}

}

Na cláusula foreach, a variável elem itera sobre cada elemento da lista. Qualquer método definido na classe Foo agora pode ser usado no elem.

O método sumOfList retorna a soma dos números em uma lista:

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

double s = 0.0;

for (Number n : list)

s += n.doubleValue();

return s;

}

O código a seguir, usando uma lista de objetos Integer, imprime soma = 6,0:

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

System.out.println("sum = " + sumOfList(li));

Uma lista de valores Double pode usar o mesmo método sumOfList. O código a seguir imprime soma = 7,0:

List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);

System.out.println("sum = " + sumOfList(ld));

# Curingas com limite inferior

A seção [Curingas Delimitados Superiores mostra que um curinga](https://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html) delimitado superior restringe o tipo desconhecido a ser um tipo específico ou um subtipo desse tipo e é representado usando a palavra-chave extend. De maneira semelhante, um curinga com limite inferior restringe o tipo desconhecido a ser um tipo específico ou um supertipo desse tipo.

Um curinga com limite inferior é expresso usando o caractere curinga ('?'), seguido pela palavra-chave super, seguido por seu limite inferior: <? super A>.

**Nota:** Você pode especificar um limite superior para um curinga ou pode especificar um limite inferior, mas não pode especificar ambos.

Digamos que você queira escrever um método que coloque objetos Integer em uma lista. Para maximizar a flexibilidade, você gostaria que o método funcionasse em List<Integer>, List<Number> e List<Object> — qualquer coisa que possa conter valores Integer.

Para escrever o método que funciona em listas de Inteiro e os supertipos de Inteiro, como Inteiro, Número e Objeto, você especificaria List<? super Integer>. O termo List<Integer> é mais restritivo do que List<? **super** Integer> porque **o primeiro corresponde a uma lista do tipo Integer somente,** enquanto o segundo corresponde a uma lista de qualquer tipo que seja um supertipo de Integer.

O código a seguir adiciona os números de 1 a 10 ao final de uma lista:

public static void addNumbers(List<? super Integer> list) {

for (int i = 1; i <= 10; i++) {

list.add(i);

}

}

A seção [Diretrizes para uso de](https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html) curinga fornece orientação sobre quando usar curingas com limite superior e quando usar curingas com limite inferior.

# Curingas e subtipagem

Conforme descrito em [Genéricos, Herança e Subtipos](https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html), classes ou interfaces genéricas não estão relacionadas simplesmente porque há uma relação entre seus tipos. No entanto, você pode usar curingas para criar uma relação entre classes ou interfaces genéricas.

Dadas as seguintes duas classes regulares (não genéricas):

class A { /\* ... \*/ }

class B extends A { /\* ... \*/ }

Seria razoável escrever o seguinte código:

B b = new B();

A a = b;

Este exemplo mostra que a herança de classes regulares segue esta regra de subtipagem: classe B é um subtipo de classe A se B estende A. Esta regra não se aplica a tipos genéricos:

List<B> lb = new ArrayList<>();

List<A> la = lb; // compile-time error

Dado que Integer é um subtipo de Número, qual é a relação entre List<Integer> e List<Number>?

Diagrama

Descrição gerada automaticamente

O pai comum é List<?>.

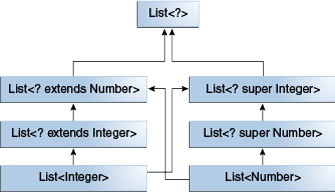
Embora Integer seja um subtipo de Número, Lista<Inteiro> não é um subtipo de Lista<Número> e, na verdade, esses dois tipos não estão relacionados. O pai comum de List<Number> e List<Integer> é List<?>.

Para criar uma relação entre essas classes para que o código possa acessar os métodos de Number por meio dos elementos de List<Integer>, use um curinga delimitado superior:

List<? extends Integer> intList = new ArrayList<>();

List<? extends Number> numList = intList; // OK. List<? extends Integer> is a subtype of List<? extends Number>

Como Integer é um subtipo de Number e numList é uma lista de objetos Number, agora existe uma relação entre intList (uma lista de objetos Integer) e numList. O diagrama a seguir mostra as relações entre várias classes List declaradas com curingas delimitados superior e inferior.



Uma hierarquia de várias declarações de classe List genéricas.

A seção Diretrizes para uso de curinga tem mais informações sobre as ramificações do [uso de curingas](https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html) com limite superior e inferior.

# Métodos auxiliares e de captura curinga

Em alguns casos, o compilador infere o tipo de um curinga. Por exemplo, uma lista pode ser definida como List<?> mas, ao avaliar uma expressão, o compilador infere um tipo específico do código. Esse cenário é conhecido como captura curinga.

Na maioria das vezes, você não precisa se preocupar com a captura de curinga, exceto quando você vê uma mensagem de erro que contém a frase "captura de".

O exemplo [WildcardError](https://docs.oracle.com/javase/tutorial/java/generics/examples/WildcardError.java" \t "_blank) produz um erro de captura quando compilado:

import java.util.List;

public class WildcardError {

void foo(List<?> i) {

i.set(0, i.get(0));

}

}

Neste exemplo, o compilador processa o parâmetro i input como sendo do tipo Object. Quando o método foo invoca [List.set(int, E),](https://docs.oracle.com/javase/8/docs/api/java/util/List.html" \l "set-int-E-" \t "_blank) o compilador não é capaz de confirmar o tipo de objeto que está sendo inserido na lista e um erro é produzido. Quando esse tipo de erro ocorre, normalmente significa que o compilador acredita que você está atribuindo o tipo errado a uma variável. Os genéricos foram adicionados à linguagem Java por esse motivo — para impor a segurança de tipo em tempo de compilação.

O exemplo WildcardError gera o seguinte erro quando compilado pela implementação javac do JDK 7 da Oracle:

WildcardError.java:6: error: method set in interface List<E> cannot be applied to given types;

i.set(0, i.get(0));

^

required: int,CAP#1

found: int,Object

reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Object from capture of ?

1 error

Neste exemplo, o código está tentando executar uma operação segura, portanto, como você pode contornar o erro do compilador? **Você pode corrigi-lo escrevendo um método auxiliar privado que captura o curinga.** Nesse caso, você pode contornar o problema criando o método auxiliar privado, fooHelper, conforme mostrado em [WildcardFixed](https://docs.oracle.com/javase/tutorial/java/generics/examples/WildcardFixed.java" \t "_blank):

public class WildcardFixed {

void foo(List<?> i) {

fooHelper(i);

}

// Helper method created so that the wildcard can be captured

// through type inference.

**private <T> void fooHelper(List<T> l) {**

**l.set(0, l.get(0));**

**}**

}

Graças ao método auxiliar, o compilador usa inferência para determinar que T é CAP#1, a variável de captura, na invocação. O exemplo agora é compilado com êxito.

Por convenção, os métodos auxiliares são geralmente chamados de originalMethodNameHelper.

Agora considere um exemplo mais complexo, [WildcardErrorBad](https://docs.oracle.com/javase/tutorial/java/generics/examples/WildcardErrorBad.java" \t "_blank):

import java.util.List;

public class WildcardErrorBad {

void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {

Number temp = l1.get(0);

l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,

// got a CAP#2 extends Number;

// same bound, but different types

l2.set(0, temp); // expected a CAP#1 extends Number,

// got a Number

}

}

Neste exemplo, o código está tentando uma operação não segura. Por exemplo, considere a seguinte invocação do método swapFirst:

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

List<Double> ld = Arrays.asList(10.10, 20.20, 30.30);

swapFirst(li, ld);

Embora List<Integer> e List<Double> preencham os critérios de List<? estende Number>, é claramente incorreto pegar um item de uma lista de valores inteiros e tentar colocá-lo em uma lista de valores duplos.

A compilação do código com o compilador javac JDK da Oracle produz o seguinte erro:

WildcardErrorBad.java:7: error: method set in interface List<E> cannot be applied to given types;

l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,

^

required: int,CAP#1

found: int,Number

reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Number from capture of ? extends Number

WildcardErrorBad.java:10: error: method set in interface List<E> cannot be applied to given types;

l2.set(0, temp); // expected a CAP#1 extends Number,

^

required: int,CAP#1

found: int,Number

reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Number from capture of ? extends Number

WildcardErrorBad.java:15: error: method set in interface List<E> cannot be applied to given types;

i.set(0, i.get(0));

^

required: int,CAP#1

found: int,Object

reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Object from capture of ?

3 errors

Não há nenhum método auxiliar para contornar o problema, porque o código está fundamentalmente errado: é claramente incorreto pegar um item de uma lista de valores inteiros e tentar colocá-lo em uma lista de valores duplos.