Dia 23 - Generecis

Conteúdo

1. Introdução

Os Genéricos Java foram introduzidos no JDK 5.0 com o objetivo de reduzir os bugs e adicionar uma camada extra de abstração sobre os tipos.

Este artigo é uma introdução rápida aos Genéricos em Java, o objetivo por trás deles e como eles podem ser usados para melhorar a qualidade do nosso código.

2. A necessidade de genéricos

Vamos imaginar um cenário onde queremos criar uma lista em Java para armazenar o Integer; podemos ser tentados a escrever:

```
List list = new LinkedList();
list.add(new Integer(1));
Integer i = list.iterator().next();
```

Surpreendentemente, o compilador vai reclamar da última linha. Ele não sabe que tipo de dados é devolvido. O compilador exigirá um elenco explícito:

```
Integer i = (Integer) list.iterator.next();
```

Dia 23 - Generecis

Não há contrato que garanta que o tipo de retorno da lista seja um Inteiro. A lista definida pode conter qualquer objeto. Só sabemos que estamos recuperando uma lista inspecionando o contexto. Ao olhar para os tipos, ele só pode garantir que é um Objeto, portanto, requer um elenco explícito para garantir que o tipo é seguro.

Este elenco pode ser irritante, sabemos que o tipo de dados nesta lista é um Inteiro. O elenco também está bagunçando nosso código. Pode causar erros de tempo de execução relacionados ao tipo se um programador cometer um erro com o elenco explícito.

Seria muito mais fácil se os programadores pudessem expressar sua intenção de usar tipos específicos e o compilador pudesse garantir a correção desse tipo. Esta é a ideia central por trás dos genéricos.

Vamos modificar a primeira linha do trecho de código anterior para:

List<Integer> list = new LinkedList<>();

Ao adicionar o operador de diamante <> contendo o tipo, reduzimos a especialização desta lista apenas ao tipo Integer, ou seja, especificamos o tipo que será mantido dentro da lista. O compilador pode impor o tipo na hora da compilação.

Em programas pequenos, isso pode parecer uma adição trivial, no entanto, em programas maiores, isso pode adicionar robustez significativa e torna o programa mais fácil de ler.

3. Métodos Genéricos

Métodos genéricos são aqueles métodos que são escritos com uma única declaração de método e podem ser chamados com argumentos de diferentes tipos. O compilador garantirá a correção de qualquer tipo que for usado. Estas são algumas propriedades de métodos genéricos:

Os métodos genéricos têm um parâmetro de tipo (o operador de diamante que inclui o tipo) antes do tipo de devolução da declaração do método

Parâmetros de tipo podem ser limitados (os limites são explicados mais tarde no artigo)

Métodos genéricos podem ter diferentes parâmetros de tipo separados por vírgulas na assinatura do método

Corpo de método para um método genérico é como um método normal

Um exemplo de definição de um método genérico para converter uma matriz em uma lista:

Dia 23 - Generecis

```
public <T> List<T> fromArrayToList(T[] a) {
   return Arrays.stream(a).collect(Collectors.toList());
}
```

No exemplo anterior, o <T> na assinatura do método implica que o método estará lidando com o tipo Tgenérico . Isso é necessário mesmo que o método esteja retornando vazio.

Como mencionado acima, o método pode lidar com mais de um tipo genérico, onde este é o caso, todos os tipos genéricos devem ser adicionados à assinatura do método, por exemplo, se quisermos modificar o método acima para lidar com o tipo T e o tipo G,ele deve ser escrito assim:

```
public static <T, G> List<G> fromArrayToList(T[] a, Function<T, G> mapperFunction) {
   return Arrays.stream(a)
     .map(mapperFunction)
     .collect(Collectors.toList());
}
```

No exemplo anterior, o <T> na assinatura do método implica que o método estará lidando com o tipo Tgenérico . Isso é necessário mesmo que o método esteja retornando vazio.

Como mencionado acima, o método pode lidar com mais de um tipo genérico, onde este é o caso, todos os tipos genéricos devem ser adicionados à assinatura do método, por exemplo, se quisermos modificar o método acima para lidar com o tipo T e o tipo G,ele deve ser escrito assim:

```
public static <T, G> List<G> fromArrayToList(T[] a, Function<T, G> mapperFunction) {
   return Arrays.stream(a)
     .map(mapperFunction)
     .collect(Collectors.toList());
}
```

Estamos passando uma função que converte uma matriz com os elementos do tipo T para listar com elementos do tipo G. Um exemplo seria converter o Integer à sua representação string:

```
@Test
```

```
public void givenArrayOfIntegers_thanListOfStringReturnedOK() {
   Integer[] intArray = {1, 2, 3, 4, 5};
   List<String> stringList
     = Generics.fromArrayToList(intArray, Object::toString);

   assertThat(stringList, hasItems("1", "2", "3", "4", "5"));
}
```

Vale ressaltar que a recomendação da Oracle é usar uma letra maiúscula para representar um tipo genérico e escolher uma letra mais descritiva para representar tipos formais, por exemplo, em Java Collections T é usado para tipo, K para chave, V para valor.

3.1. Genéricos Delimitados

Como mencionado anteriormente, os parâmetros de tipo podem ser limitados. Limitado significa "restrito", podemos restringir tipos que podem ser aceitos por um método.

Por exemplo, podemos especificar que um método aceita um tipo e todas as suas subclasses (limite superior) ou um tipo de todas as suas superclasses (limite inferior).

Para declarar um tipo superior limitado, usamos a palavra-chave se estende após o tipo seguido pelo limite superior que queremos usar. Por exemplo:

```
public <T extends Number> List<T> fromArrayToList(T[] a) {
   ...
}
```

A palavra-chave é usada aqui para significar que o tipo T estende o limite superior em caso de uma classe ou implementa um limite superior no caso de uma interface.

3.2. Múltiplos Limites

Um tipo também pode ter vários limites superiores da seguinte forma:

```
<T extends Number & Comparable>
```

Se um dos tipos que são estendidos por T é uma classe (ou seja, Número), ele deve ser colocado em primeiro lugar na lista de limites. Caso contrário, causará um erro de tempo de compilação.

Dia 23 - Generecis

4. Usando curingas com genéricos

Curingas são representados pelo ponto de interrogação em Java "?" e eles são usados para se referir a um tipo desconhecido. Curingas são particularmente úteis ao usar genéricos e podem ser usados como um tipo de parâmetro, mas primeiro, há uma nota importante a considerar.

Sabe-se que Object é o supertipo de todas as classes Java, no entanto, uma coleção de Objeto não é o supertipo de qualquer coleção.

Por exemplo, um List<Object> não é o supertipo de List<String> e atribuir uma variável do tipo List<Object> a uma variável do tipo List<String> causará um erro no compilador. Isso é para evitar possíveis conflitos que podem acontecer se adicionarmos tipos heterogêneos à mesma coleção.

A mesma regra se aplica a qualquer coleção de um tipo e seus subtipos. Considere este exemplo:

```
public static void paintAllBuildings(List<Building> buildings) {
   buildings.forEach(Building::paint);
}
```

se imaginarmos um subtipo de Construção,por exemplo, uma Casa,não podemos usar esse método com uma lista de Casa,mesmo que House seja um subtipo de Construção. Se precisarmos usar este método com o tipo Building e todos os seus subtipos, então o curinga limitado pode fazer a magia:

```
public static void paintAllBuildings(List<? extends Building> buildings) {
   ...
}
```

Agora, este método funcionará com o tipo Building e todos os seus subtipos. Isso é chamado de curinga de limites superiores onde o tipo Edifício é o limite superior.

Curingas também podem ser especificados com um limite inferior, onde o tipo desconhecido deve ser um supertipo do tipo especificado. Limites inferiores podem ser especificados usando a super palavra-chave seguida pelo tipo específico, por exemplo, <? super T> significa tipo desconhecido que é uma superclasse de T (= T e todos os seus pais).

Dia 23 - Generecis

5. Eliminação do tipo

Genéricos foram adicionados ao Java para garantir a segurança do tipo e para garantir que os genéricos não causassem sobrecarga no tempo de execução, o compilador aplica um processo chamado eliminação de tipo em genéricos na hora da compilação.

O tipo de eliminação remove todos os parâmetros de tipo e substitui-o por seus limites ou por Objeto se o parâmetro de tipo não for vinculado. Assim, o bytecode após a compilação contém apenas classes, interfaces e métodos normais, garantindo assim que nenhum novo tipo seja produzido. O fundição adequado também é aplicado ao tipo Objeto na hora da compilação.

Este é um exemplo de eliminação do tipo:

```
public <T> List<T> genericMethod(List<T> list) {
   return list.stream().collect(Collectors.toList());
}
```

Com o apagamento do tipo, o tipo T não vinculado é substituído por Object da seguinte forma:

```
// for illustration
public List<Object> withErasure(List<Object> list) {
   return list.stream().collect(Collectors.toList());
}

// which in practice results in
public List withErasure(List list) {
   return list.stream().collect(Collectors.toList());
}
```

Se o tipo estiver limitado, o tipo será substituído pelo limite na hora do compilado:

```
public <T extends Building> void genericMethod(T t) {
   ...
}
```

mudaria após a compilação:

Dia 23 - Generecis

```
public void genericMethod(Building t) {
   ...
}
```

6. Genéricos e Tipos de Dados Primitivos

Uma restrição de genéricos em Java é que o parâmetro do tipo não pode ser um tipo primitivo.

Por exemplo, o seguinte não compila:

```
List<int> list = new ArrayList<>();
list.add(17);
```

Para entender por que os tipos de dados primitivos não funcionam, vamos lembrar que os genéricos são um recurso de tempo de compilação,o que significa que o parâmetro do tipo é apagado e todos os tipos genéricos são implementados como objetotipo .

Como exemplo, vamos olhar para o método de adoção de uma lista:

```
List<Integer> list = new ArrayList<>();
list.add(17);
```

A assinatura do método de adoção é:

```
boolean add(E e);
```

E será compilado para:

```
boolean add(Object e);
```

Portanto, os parâmetros de tipo devem ser conversíveis para Objeto. Como os tipos primitivos não estendem o objeto,não podemos usá-los como parâmetros de tipo.

No entanto, java fornece tipos encaixotados para primitivos, juntamente com autoboxing e unboxing para desembrulhá-los:

```
Integer a = 17;
int b = a;
```

Dia 23 - Generecis

Então, se quisermos criar uma lista que possa conter inteiros, podemos usar o invólucro:

```
List<Integer> list = new ArrayList<>();
list.add(17);
int first = list.get(0);
```

O código compilado será o equivalente a:

```
List list = new ArrayList<>();
list.add(Integer.valueOf(17));
int first = ((Integer) list.get(0)).intValue();
```

Versões futuras de Java podem permitir tipos de dados primitivos para genéricos. O Projeto Valhalla visa melhorar a forma como os genéricos são tratados. A ideia é implementar a especialização de genéricos conforme descrito na JEP 218.

Video

---

https://www.youtube.com/watch?v=DgdA5oXr2ko&ab_channel=DevDojo

EXERCICIO

---

1) Dados as declarações de importação adequadas e:

13. TreeSet <> s = novo TreeSet <> ();

14. TreeSet <> subs = novo TreeSet <> ();

15. s.add ("a"); s.add ("b"); s.add ("c"); s.add ("d"); s.add ("e");

16

17. subs = (TreeSet) s.subSet ("b", verdadeiro, "d", verdadeiro);

18. s.add ("g");

19. s.pollFirst ();

20. s.pollFirst ();

21. s.add ("c2");

22. System.out.println (s.size () + "" + subs.size ());

Quais são verdadeiras? (Escolha todas as opções aplicáveis.)

A. O tamanho de s é 4

B. O tamanho de s é 5

Dia 23 - Generecis

C. O tamanho de s é 7

D. O tamanho dos subs é 1

E. O tamanho dos subs é 2

2) Dado:

```
public static void main (String [] args) {
// INSERIR DECLARAÇÃO AQUI
para (int i = 0; i <= 10; i ++) {
List <Integer> row = new ArrayList <Integer> ();
para (int j = 0; j <= 10; j ++)
row.add (i * j);
table.add (linha);
}
para (List <Integer> linha: tabela)
System.out.println (linha);
}
```

Quais instruções podem ser inseridas em // INSERT DECLARATION HERE para permitir que este código

compilar e executar? (Escolha todas as opções aplicáveis.)

A. List <List <Integer>> table = new List <List <Integer>> ();

B. List <List <Integer>> tabela = new ArrayList <List <Integer>> ();

C. List <List <Integer>> table = new ArrayList <ArrayList <Integer>> ();

D. List <List, Integer> table = new List <List, Integer> ();

E. List <List, Integer> table = new ArrayList <List, Integer> ();

3) Quais afirmações são verdadeiras sobre a comparação de duas instâncias da mesma classe, dado que o

Os métodos equals () e hashCode () foram substituídos corretamente? (Escolha todas as opções aplicáveis.)

A. Se o método equals () retornar verdadeiro, a comparação hashCode () == pode retornar falso

B. Se o método equals () retornar falso, a comparação hashCode () == pode retornar verdadeiro

Dia 23 - Generecis

C. Se a comparação hashCode () == retornar verdadeiro, o método equals () deve retornar verdadeiro

D. Se a comparação hashCode () == retorna null, o método equals () pode retornar verdadeiro

Prova

---

1) Dado:

```
public static void before () {
 Set set = new TreeSet ();
 set.add ("2");
 set.add (3);
 set.add ("1");
 Iterator it = set.iterator ();
 while (it.hasNext ())
 System.out.print (it.next () + "");
}
```

Quais afirmações são verdadeiras?

A. O método before () imprimirá 1 2

B. O método before () imprimirá 1 2 3

C. O método before () imprimirá três números, mas a ordem não pode ser determinada

D. O método before () lançará uma exceção em tempo de execução

2) Dado:

```
import java.util. *;
class MapEQ {
 public static void main (String [] args) {
 Map <ToDos, String> m = novo HashMap <ToDos, String> ();
 ToDos t1 = novos ToDos ("segunda-feira");
 ToDos t2 = novos ToDos ("segunda-feira");
 ToDos t3 = novos ToDos ("terça-feira");
 m.put (t1, "doLaundry");
 m.put (t2, "contas a pagar");
 m.put (t3, "cleanAttic");
```

```
 System.out.println (m.size ());
}}
classe ToDos {
 Dia da corda;
 ToDos (String d) {day = d; }
 public boolean equals (Object o) {
 return ((ToDos) o) .day == this.day;
 }
 // public int hashCode () {return 9; }
}
```

Qual é correto? (Escolha todas as opções aplicáveis.)

A. Como o código está, ele não irá compilar

B. Como o código está, a saída será 2

C. Como o código está, a saída será 3

D. Se o método hashCode () não for comentado, a saída será 3

3) Dado:

```
12. public class AccountManager {
13. private Map accountTotals = new HashMap ();
14. fundo privado de aposentadoria;
15
16. public int getBalance (String accountName) {
17. Número inteiro total = (Número inteiro) accountTotals.get (accountName);
18. if (total == null)
19. total = Integer.valueOf (0);
20. retornar total.intValue ();
21.}
23. public void setBalance (String accountName, int amount) {
24. accountTotals.put (accountName, Integer.valueOf (amount));
25.}}
```

Dia 23 - Generecis

Esta classe deve ser atualizada para fazer uso de tipos genéricos apropriados, sem mudanças no comportamento

(para melhor ou pior). Qual dessas etapas pode ser executada? (Escolha três.)

A. Substitua a linha 13 por

private Map <String, int> accountTotals = new HashMap <String, int> ();

B. Substitua a linha 13 por

private Map <String, Integer> accountTotals = new HashMap <String, Integer> ();

C. Substitua a linha 13 por

Mapa privado <String <Integer>> accountTotals = new HashMap <String <Integer>> ();

D. Substitua as linhas 17-20 por

int total = accountTotals.get (accountName);

se (total == nulo) total = 0;

retorno total;

se (total == nulo) total = 0;

retorno total;

F. Substitua as linhas 17-20 por

return accountTotals.get (accountName);