Menu

RandomDev

De tudo um pouco sobre o mundo do desenvolvimento de software.

JAVA • GENERICS

Generics indo além - Wildcards

POR ROBSON BITTENCOURT 02/02/2016 0 COMMENTS TWEET LIKE +1

EDITE ESTE POST

No meu último post apresentei alguns conceitos básicos sobre *Generics*. Vale reforçar que o conteúdo que será apresentado aqui, foi visto no curso da Pluralsight sobre Generics. Não, não estou fazendo propaganda, é que o curso é bom mesmo, recomendo :)

Variância

Antes de explicar o que são *Wildcards* é necessário mostrar alguns detalhes sobre *arrays* em Java. Variância é um conceito que se relaciona a como a herança entre classes se comporta na declaração de variáveis. Não vou me aprofundar neste assunto, mas este link traz uma ótima explicação sobre.

Por hora precisamos saber que *arrays* em Java são covariantes, vamos entender o porquê.

```
public class Animal {}

public class Cat extends Animal {}

public class Dog extends Animal {}
```

```
Cat[] cats = new Cat[2];

Animal[] animals = cats;
```

Como pode-se perceber, é possível declarar um *array* com o tipo da classe filha e atribuir à um *array* do tipo da classe pai. Isso é conhecido como covariância, e traz alguns problemas como este:

```
Cat[] cats = new Cat[2];

Animal[] animals = cats;

animals[0] = new Dog(); //compila pois Dog é um Animal

// out - Exception in thread "main" java.lang.ArrayStoreException:
// com.generics.lab.Dog at
// com.generics.lab.GenericTest.main(GenericTest.java:12)
```

O *array* é um tipo primitivo especial, não possui uma classe nem implementa uma interface, mas possui uma estrutura, e esta conhece o tipo que deve ser possível armazenar, por isso a exceção no exemplo foi lançada. Isso nos leva a perceber que *arrays* não são muito seguros, por isso é recomendado o uso de listas.

As listas ao contrário dos *arrays* são invariantes e só podem ser atribuídas a variáveis com o mesmo tipo genérico.

```
List<Dog> dogs = new ArrayList<>();
List<Animal> animals = dogs; // não compila apesar de Dog ser um Animal

List<Dog> otherDogs = dogs; // compila
```

Isso torna o uso de listas mais seguro que *arrays*. Caso as listas não fossem invariantes poderiamos fazer algo desse tipo:

```
List<Dog> dogs = new ArrayList<>();
List<Animal> animals = dogs; // não compila é somente um exemplo

animals.add(new Cat());
Dog dog = dogs.get(0); // nada de bom iria acontecer aqui
```

Wildcards

Mas e no caso de precisarmos de uma lista com elementos de vários tipos os quais temos domínio. Por exemplo, queremos criar um método que retorne a espécie dos animais de uma lista. Como observamos, se utilizarmos o tipo da classe pai, só poderemos trabalhar com listas deste tipo, o que nos leva a ter que criar uma método para cada classe filha.

```
// considere que getSpecies é um método da classe Animal

public void printAllDogSpecies(List<Dog> dogs) {
   for (Dog dog : dogs) {
      System.out.println(dog.getSpecies());
   }
}

public void printAllCatSpecies(List<Cat> cats) {
   for (Cat cat : cats) {
      System.out.println(cat.getSpecies());
   }
}
```

Cheira mal certo? Se houvessem cem implementações de Animal, teríamos este código replicado uma centena de vezes. Felizmente temos como resolver isso utilizando *Wildcards*.

Upper Bounded Wildcards

Wildcard é o nome dado ao identificador *?* em códigos genéricos. Ele representa um tipo desconhecido, e pode ser utilizado em algumas situações como um tipo de parâmetro ou uma

variável local.

Existem algumas formas de utilizar o *Wildcard*, a primeira que iremos observar é o *Upper Bounded Wildcard*. Vamos resolver o problema anterior utilizando este tipo de *Wildcard*:

```
public void printAllSpecies(List<? extends Animal> animals) {
   for (Animal animal : animals) {
      System.out.println(animal.getSpecies());
   }
}
```

Pronto agora temos um método que pode ser utilizado para qualquer classe filha de Animal. A sintaxe que utilizamos diz que pode-se passar no parâmetro qualquer lista onde o tipo genérico seja uma classe que extenda Animal.

Observando-se a sintaxe, podemos entender o porque do nome *Upper Bounded Wildcard*. Ao utilizarmos `<? extends Animal>` Animal é a classe de hierarquia mais alta (*upper*) aceita como parâmetro.

Você pode estar se perguntando, qual é a diferença entre utilizar o `? extends` ou `T extends`. O *?* é utilizado onde já é esperado um tipo genérico, por exemplo na interface List. Quando uma classe genérica é criada não podemos utilizar o *Wildcard* deve-se utilizar o identificador de tipo genérico.

Também podemos utilizar em uma classe genérica que possui um método genérico com tipo diferente do da classe. Além disso, também é útil como *sintax sugar* na assinatura de métodos.

```
public <T extends Animal> void printAllSpecies(List<T> animals) {
   for (Animal animal : animals) {
      System.out.println(animal.getSpecies());
   }
}

// como não utilizamos o T no corpo do método podemos utilizar o Wildcard
```

```
public void printAllSpecies(List<? extends Animal> animals) {
   for (Animal animal : animals) {
      Sytslem.out.println(animal.getSpecies());
   }
}
```

Lower Bounded Wildcards

Semelhante a forma anterior, o *Lower Bounded Wildcards* possui uma sintaxe semelhante, porém no lugar da palavra *extends* utilizamos *super*.

```
public void loadNewAnimals(List<? super Animal> animals) {
   Animal animal;

   while ((animal = searchNewAnimals()) != null) {
      animals.add(animal);
   }
}
```

Neste caso podemos passar como argumento uma lista com tipo genérico igual a *Animal* ou uma de suas classes pai. Ou seja *Animal* é a classe mais baixa (lower) da hierarquia que pode ser utilizada.

Unbounded Wildcards

A última forma é utilizada quando precisamos de um tipo genérico mas não nos importamos com qual.

```
Class<?> animal = Class.forName("Animal");
```

O *Wildcard* sozinho é um *sintaxe sugar* para `<? extends Object>`

Quando utilizar cada forma?

Existem recomendações que nos dizem quando utilizar cada uma das formas de *Wildcards*. **Lembrando que devemos utilizá-los somente quando o tipo concreto é muito restritivo para a lógica do método.**

O *Upper Bounded* deve ser utilizado quando a variável for "de entrada" (in). Dizemos que a variável é de entrada quando ela fornece dados à um método, ou seja será lida.

```
public void printAllSizes(List<? extends Animal> animals) {
   for (Animal animal : animals) {
      Sytslem.out.println(animal.getSize());
   }
}
```

Essa regra nos dá segurança, pois como a variável `animals` será lida dentro do `for`, só é seguro aceitarmos uma instância de *Animal* ou uma de subclasses.

O *Lower Bounded* é o caso contrário. Devemos utilizá-lo quando a variável for de saída (out), ou seja, uma variável que terá dados escritos nela, para utilização posterior.

```
public void createDefaultAnimalList(List<? super Animal> animals) {
   animals.add(new Dog());
   animals.add(new Cat());
}
```

Como o método adiciona objetos da classe *Animal* em uma lista, o tipo genérico da lista deve ser *Animal* ou uma de suas superclasses.

O *Unbounded* é o que tem aplicação mais limitada. Sua sintaxe `<?>` é um *sintaxe sugar* para `<? extends Object>`.

```
public static void printList(List<?> list) {
    for (Object elem: list) {
        System.out.print(elem + " ");
    }
    System.out.println();
}
```

Podemos utilizá-lo quando o método usa funcionalidades da classe Object no parâmetro, ou quando a implementação não depende do tipo genérico.

Erros comuns

Muitas pessoas se confundem ao utilizar *Wildcards* com listas, devido ao fato de acreditarem que o tipo genérico se refere ao elementos que podem ser inseridos na mesma.

É importante ressaltar que o tipo genérico define o tipo que será utilizado na classe ou método, ou seja uma vez definido ele é único. Vamos ver um exemplo utilizando o *Unbounded Wildcard*.

```
List<?> objects = new ArrayList<>();

// não compila
objects.add(new Object());
objects.add(new Dog());
```

Isso acontece pois `<?>` é a mesma coisa que `<? extends Object>`. Por isso nenhum valor pode ser inserido nesta lista (a não ser null).

O exemplo também ressalta que não devemos pensar no `<?>` como sendo a mesma coisa que a classe *Object*, pois com ela o mesmo exemplo seria válido.

```
List<Object> objects = new ArrayList<>();

// compila
```

```
objects.add(new Object());
objects.add(new Dog());
```

Conclusão

Como vimos, o conhecimento sobre o uso correto dos *Wildcards* nos possibilita escrevermos códigos mais flexíveis. É um assunto um pouco complexo, alguns pontos ainda ficaram de fora deste post, por isso deixo algumas referências que utilizei que podem servir como complemento. No post anterior, havia dito que falaria sobre *Raw Types* e *Erasure*, mas como esse ficou grande, vai ter que ficar para o próximo.

Referências

Java Fudamentals: Generic

Variância

Documentação sobre Generics

Guidelines for Wildcard Use

Java Generics FAQs - Programming With Java Generics

Previous Next