Professor Hans Noodles

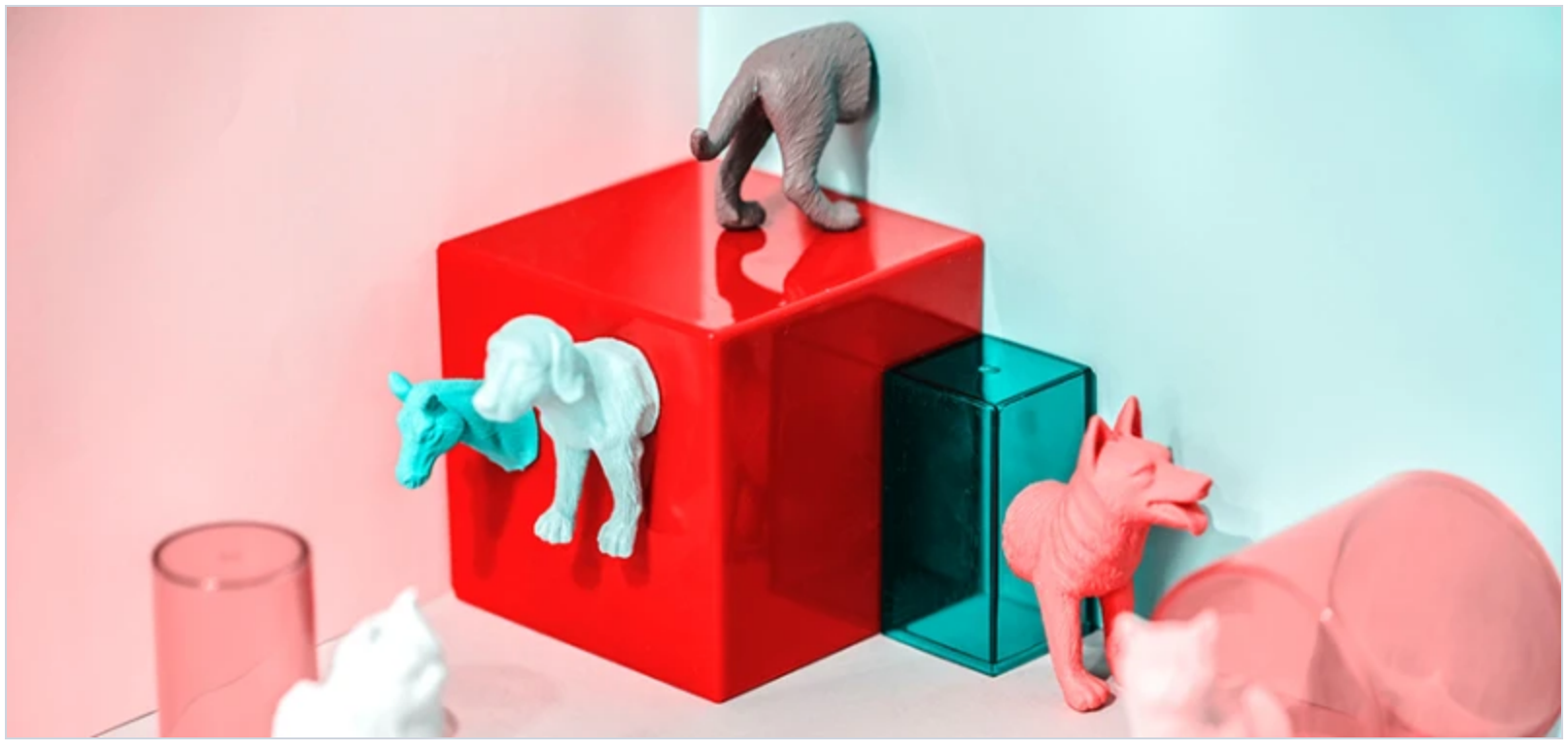
41 уровень

24.07.2019 44269 35

Wildcards в generics

Привет! Продолжаем изучать тему дженериков.

Ты уже обладаешь солидным багажом знаний о них из предыдущих лекций (об использовании varargs при работе с дженериками и о стирании типов), но одну важную тему мы пока не рассматривали — **wildcards**.



Это очень важная фишка дженериков. Настолько, что мы выделили для нее отдельную лекцию! Впрочем, ничего сложного в wildcards нет, в этом ты сейчас убедишься :)

Давай рассмотрим пример:

```
1  public class Main {
2
3     public static void main(String[] args) {
4
5         String str = new String("Test!");
6         // никаких проблем
7         Object obj = str;
8
9         List<String> strings = new ArrayList<String>();
10        // ошибка компиляции!
11        List<Object> objects = strings;
```

Что же тут происходит?

Мы видим две очень похожие ситуации. В первой из них мы пытаемся привести объект `String` к типу `Object`. Никаких проблем с этим не возникает, все работает как надо.

Но вот во второй ситуации компилятор выдает ошибку. Хотя, казалось бы, мы делаем то же самое. Просто теперь мы используем коллекцию из нескольких объектов.

Но почему возникает ошибка? Какая, по сути, разница — приводим мы один объект `String` к типу `Object` или 20 объектов?

Между **объектом** и **коллекцией объектов есть важное различие**.

Если класс `B` является наследником класса `A`, то `Collection<B>` при этом — не наследник `Collection<A>`.

Именно по этой причине мы не смогли привести наш `List<String>` к `List<Object>`. **`String` является наследником `Object`, но `List<String>` не является наследником `List<Object>`.**

Интуитивно это выглядит не очень логично. Почему именно таким принципом руководствовались создатели языка?

Давай представим, что здесь компилятор не выдавал бы нам ошибку:

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

В этом случае, мы бы могли, например, сделать следующее:

```
objects.add(new Object());
String s = strings.get(0);
```

Поскольку компилятор не выдал нам ошибок и позволил создать ссылку `List<Object> object` на коллекцию строк `strings`, можем добавить в `strings` не строку, а просто любой объект `Object`!

Таким образом, **мы лишились гарантии того, что в нашей коллекции находятся только указанные в дженерике объекты `String`**. То есть, мы потеряли главное преимущество дженериков — типобезопасность.

И раз компилятор позволил нам все это сделать, значит, мы получим ошибку только во время исполнения программы, что всегда намного хуже, чем ошибка компиляции.

Чтобы предотвратить такие ситуации, компилятор выдает нам ошибку:

```
// ошибка компиляции
List<Object> objects = strings;
```

...и напоминает, что `List<String>` — не наследник `List<Object>`.

Это железное правило работы дженериков, и его нужно обязательно помнить при их использовании.

Поехали дальше.

Допустим, у нас есть небольшая иерархия классов:

```java
public class Animal {

   public void feed() {

       System.out.println("Animal.feed()");
   }
}

public class Pet extends Animal {

   public void call() {

       System.out.println("Pet.call()");
   }
}

public class Cat extends Pet {

   public void meow() {

       System.out.println("Cat.meow()");
   }
}
```

Во главе иерархии стоят просто Животные: от них наследуются Домашние Животные. Домашние Животные делятся на 2 типа — Собаки и Кошки.

А теперь представь, что нам нужно создать простой метод `iterateAnimals()`. Метод должен принимать коллекцию любых животных (`Animal`, `Pet`, `Cat`, `Dog`), перебирать все элементы, и каждый раз выводить что-нибудь в консоль.

Давай попробуем написать такой метод:

```java
public static void iterateAnimals(Collection<Animal> animals) {

   for(Animal animal: animals) {

       System.out.println("Еще один шаг в цикле пройден!");
   }
}
```

Казалось бы, задача решена!

Однако, как мы недавно выяснили, `List<Cat>`, `List<Dog>` или `List<Pet>` не являются наследниками `List<Animal>`!

Поэтому при попытке вызвать метод `iterateAnimals()` со списком котиков мы получим ошибку компилятора:

```java
import java.util.*;

public class Main3 {


   public static void iterateAnimals(Collection<Animal> animals) {
```

```
9
10              System.out.println("Еще один шаг в цикле пройден!");
11          }
12      }
13
14      public static void main(String[] args) {
15
16
17          List<Cat> cats = new ArrayList<>();
18          cats.add(new Cat());
19          cats.add(new Cat());
20          cats.add(new Cat());
21          cats.add(new Cat());
22
23          //ошибка компилятора!
24          iterateAnimals(cats);
25      }
26  }
```

Ситуация выглядит не очень хорошо для нас! Получается, нам придется писать отдельные методы для перебора всех видов животных?

На самом деле нет, не придется :) И в этом нам как раз помогут **wildcards**!

Мы решим задачу в рамках одного простого метода, используя вот такую конструкцию:

```
1  public static void iterateAnimals(Collection<? extends Animal> animals) {
2
3     for(Animal animal: animals) {
4
5         System.out.println("Еще один шаг в цикле пройден!");
6     }
7  }
```

Это и есть wildcard. Точнее, это первый из нескольких типов wildcard — “**extends**” (другое название — **Upper Bounded Wildcards**).

О чем нам говорит эта конструкция? Это значит, что метод принимает на вход коллекцию объектов класса `Animal` либо объектов любого класса-наследника `Animal (? extends Animal)`.

Иными словами, метод может принять на вход коллекцию `Animal`, `Pet`, `Dog` или `Cat` — без разницы.

Давай убедимся что это работает:

```
1  public static void main(String[] args) {
2
3     List<Animal> animals = new ArrayList<>();
4     animals.add(new Animal());
5     animals.add(new Animal());
6
7     List<Pet> pets = new ArrayList<>();
8     pets.add(new Pet());
```

```
11      List<Cat> cats = new ArrayList<>();
12      cats.add(new Cat());
13      cats.add(new Cat());
14
15      List<Dog> dogs = new ArrayList<>();
16      dogs.add(new Dog());
17      dogs.add(new Dog());
18
19      iterateAnimals(animals);
20      iterateAnimals(pets);
21      iterateAnimals(cats);
22      iterateAnimals(dogs);
23  }
```

Вывод в консоль:

```
Еще один шаг в цикле пройден!
Еще один шаг в цикле пройден!
Еще один шаг в цикле пройден!
Еще один шаг в цикле пройден!
Еще один шаг в цикле пройден!
Еще один шаг в цикле пройден!
Еще один шаг в цикле пройден!
Еще один шаг в цикле пройден!
```

Мы создали в общей сложности 4 коллекции и 8 объектов, и в консоли ровно 8 записей. Все отлично работает! :)

Wildcard позволил нам легко уместить нужную логику с привязкой к конкретным типам в один метод. Мы избавились от необходимости писать отдельный метод для каждого вида животных. Представь, сколько методов у нас было бы, если бы наше приложение использовалось в зоопарке или ветеринарной клинике :)

А теперь давай рассмотрим другую ситуацию.

Наша иерархия наследования останется неизменной: класс верхнего уровня `Animal`, чуть ниже — класс домашних животных `Pet`, а на следующем уровне — `Cat` и `Dog`.

Теперь тебе нужно переписать метод `iretateAnimals()` таким образом, чтобы он мог работать с любым типом животных, **кроме собак**.

То есть он должен принимать на вход `Collection<Animal>`, `Collection<Pet>` или `Collection<Cat>`, но не должен работать с `Collection<Dog>`.

Как мы можем этого добиться?

Кажется, перед нами опять замаячила перспектива писать отдельный метод для каждого типа :/

А сделать это можно очень просто! Здесь нам снова придут на помощь wildcards. Но на этот раз мы воспользуемся другим типом — “**super**” (другое название — **Lower Bounded Wildcards**).

```
public static void iterateAnimals(Collection<? super Cat> animals) {

   for(int i = 0; i < animals.size(); i++) {

       System.out.println("Еще один шаг в цикле пройден!");
   }
}
```

Здесь принцип похож.

Конструкция `<? super Cat>` говорит компилятору, что метод `iterateAnimals()` может принимать на вход коллекцию объектов класса `Cat` либо любого другого класса-предка `Cat`.

Под это описание в нашем случае подходят сам класс `Cat`, его предок — `Pets`, и предок предка — `Animal`.

Класс `Dog` не вписывается в это ограничение, и поэтому попытка использовать метод со списком `List<Dog>` приведет к ошибке компиляции:

```
public static void main(String[] args) {

   List<Animal> animals = new ArrayList<>();
   animals.add(new Animal());
   animals.add(new Animal());

   List<Pet> pets = new ArrayList<>();
   pets.add(new Pet());
   pets.add(new Pet());

   List<Cat> cats = new ArrayList<>();
   cats.add(new Cat());
   cats.add(new Cat());

   List<Dog> dogs = new ArrayList<>();
   dogs.add(new Dog());
   dogs.add(new Dog());

   iterateAnimals(animals);
   iterateAnimals(pets);
   iterateAnimals(cats);

   //ошибка компиляции!
   iterateAnimals(dogs);
}
```

Наша задача решена, и снова wildcards оказались крайне полезными :)

На этом лекция подошла к концу.

А сейчас — самое время вернуться к задачам!

Успехов в обучении! :)

+149

Комментарии (35) популярные новые старые

JavaCoder

Введите текст комментария

Алексей Медынцев Уровень 1, Russian Federation 12 августа, 13:29

>> Ситуация выглядит не очень хорошо для нас! Получается, нам придется писать отдельные методы для перебора всех видов животных?

А если использовать полиморфизм?

Вместо:

```
List<Cat> cats = new ArrayList<>();
cats.add(new Cat());
cats.add(new Cat());
cats.add(new Cat());
cats.add(new Cat());
```

Написать:

```
List<Animal> cats = new ArrayList<>();
cats.add(new Cat());
cats.add(new Cat());
cats.add(new Cat());
cats.add(new Cat());
```

И не нужно писать отдельные методы для перебора всех видов животных

Ответить 0

Maksim Tatarintsev Уровень 37, Москва 18 июля, 14:48

Кому не очень понятно про <? extends A> и <? super A>, предлагаю скопировать пример ниже в IDEA и рассмотреть допустимые значения:

```
import java.util.ArrayList;
import java.util.List;

public class Test {
    List list;
    List list1;
    static class C{}
    static class A extends C{}
    static class B extends A{}
    static class D extends B{}

    public static void main(String[] args) {
        Test test = new Test();

        List<A> listA = new ArrayList<>();
        List<B> listB = new ArrayList<>();
        List<C> listC = new ArrayList<>();
        List<D> listD = new ArrayList<>();

        test.setList(listA);
```

НАЧАТЬ ОБУЧЕНИЕ

```
24
25          test.setList1(listA);
26          test.setList1(listB);
27          test.setList1(listD);
28          test.setList1(listC);
29      }
30
31
32      void setList(List<? extends A> spisok){
33          this.list = spisok;
34      }
35      void setList1(List<? super A> spisok){
36          this.list1 = spisok;
37      }
38  }
```

Ответить +1

Maksim Tatarintsev Уровень 37, Москва 18 июля, 14:51

В данном случае метод setList -относится к <? extends A>, а метод setList1 - <? super A>:
в нем видно что super - это нижняя граница (включающая текущий класс и его родителей), а extends - это верхняя граница(тоже включающая текущий класс и его наследников)

Ответить +1

Андрей Безладнов Уровень 15, Тольятти, Russian Federation 19 августа, 11:28

Слегка по лучше уложился материал, спасибо)

Ответить 0

Anonymous #3145979 Уровень 1, Ukraine 20 августа, 19:15

Мне так понятнее немного

```
import java.util.ArrayList;
import java.util.List;

public class Wildcards {
    List list;
    List list1;
    static class A{}
    static class B extends A{}
    static class C extends B{}
    static class D extends C{}

    public static void main(String[] args) {
        Wildcards test = new Wildcards();

        List<A> listA = new ArrayList<>();
        List<B> listB = new ArrayList<>();
        List<C> listC = new ArrayList<>();
        List<D> listD = new ArrayList<>();

        test.eXtends(listA);
        test.eXtends(listB);
        test.eXtends(listC);
        test.eXtends(listD);

        test.sUper(listA);
        test.sUper(listB);
        test.sUper(listC);
        test.sUper(listD);
    }


    void eXtends(List<? extends B> spisok){
        this.list = spisok;
    }
    void sUper(List<? super B> spisok){
        this.list1 = spisok;
    }
}
```

Ответить 0

Anonymous #3068853 Уровень 3 28 мая, 04:43

Вопрос к первому примеру. Перепишем его так:

```
1  public class Main {
2
3      public static void main(String[] args) {
4
5          String str = new String("Test!");
6          // никаких проблем
```

```
10          List objects = strings;
11          objects.add(new Object());
12      }
13  }
```

Тут никакой ошибки не возникает, только предупреждения. Почему компилятор не запрещает так делать, чтобы обеспечить типобезопасность?

Ответить 0

Алексей Уровень 32, Москва, Россия 28 мая, 20:03

Как я понял, как раз в случае использования raw List, как в Вашем случае:
List objects = strings;
типобезопасность не гарантируется.
Idea пишет: "Unchecked call to 'add(E)' as a member of raw type 'java.util.List'"
На stackoverflow пишут: "Этот синтаксис остался от Java 1.4 и более ранних версий. Он по-прежнему компилируется в целях обратной совместимости."

Ответить +1

LuneFox инженер по сопровождению в **BIFIT** EXPERT 26 января, 17:44

```
1  public static void iterateAnimals(Collection<? super Cat> animals) { ... }
```

Получается, что этим же методом я смогу итерировать даже просто список любых Object. А как сделать, чтобы можно было итерировать только то, что ниже Animal и выше Cat?

Ответить +7

Alexander Tsaregorodtsev Уровень 35, Новосибирск, Russian Federation 9 апреля, 18:47

Тоже интересует этот вопрос!

Ответить +1

Anonymous #2436575 Android Developer в **AllPets** 4 июля, 09:51

Это невозможно сделать в соответствии с Oracle Docs :
Можно указать верхнюю границу Wildcards или нижнюю границу, но нельзя указать и то, и другое.

Отдельно могу сказать, что если вам требуется такое ограничение, то с вашей иерархией наследования что то не так.

Ответить 0

Karoshi Уровень 13, Новосибирск, Russian Federation 18 декабря 2021, 14:42

Так и что такое wildcard?

Ответить +3

AlinaAlina Уровень 35, Санкт-Петербург 6 февраля, 00:45

Это дженерик любого типа - <?>

Ответить +5

Shaman_2010 Уровень 22, Санкт-Петербург, Russian Federation 12 ноября 2021, 15:57

Не совсем понятно в чем разница между:

```
1  public static void iterateAnimals(Collection<? extends Animal> animals) {
2     for(Animal animal: animals) {
3         System.out.println("Еще один шаг в цикле пройден!");
4     }
5  }
```

и

```
1  public static <T extends Animal> void iterateAnimals(Collection<T> animals) {
2     for(T animal: animals) {
3         System.out.println("Еще один шаг в цикле пройден!");
4     }
5  }
```

Ответить +4

Андрей Шляхтович Уровень 18 19 ноября 2021, 02:24

В верхнем случае метод примет аргументом только с типом, унаследованным от Animal, а нижний - коллекцию с любым вложенным типом.

Если в коде много методов и каждый принимает свой тип, то из-за того, что IDE(IDEA, к примеру) не выдаст ошибки на стадии присваивания аргумента (то есть при указании конкретной

Shaman_2010 Уровень 22, Санкт-Петербург, Russian Federation — 21 ноября 2021, 13:59

Спасибо за ответ, но во втором случае разве с любым?
Разве

```
1  <T extends Animal>
```

так же не ограничивает только типом, унаследованным от Animal?

Ответить +4

SchlechtGut Уровень 51, Москва — 3 декабря 2021, 15:37

idea все пропускает, поъоже, что это две идентичные записи

Ответить +4

Максим Уровень 37 — 30 марта 2021, 15:05

Наконец то просветление... вроде

Ответить +2

Алексей Серов Уровень 27 — 23 сентября 2020, 17:57

То есть он должен принимать на вход Collection<Animal>, Collection<Pet> **или Collection<Car>**, но не должен работать с Collection<Dog>.
Cat...

Ответить +4

Хорс Уровень 41, Харьков — 29 августа 2020, 12:03

В иерархии классов упущен класс Dog.
Он, на самом деле потомок Pet и находится на одном уровне с классом Cat. - тогда статья становится понятней

Ответить +5

Pig Man Главная свинья в **Свинарнике** — 12 февраля 2021, 20:41

Мне кажется, у 10167 просмотревших, у тебя единственного возникли проблемы с этим моментом..

Ответить +5

Сергей Уровень 36, Москва — 4 ноября 2021, 21:31

У меня тоже возникла тревога ..и я хотел ее забить.

Ответить 0

alex Уровень 41, Россия — 30 апреля 2020, 12:10

а если у нас есть три класса extends Pet:
Cat
Dog
mouse
как с помощью этой конструкции сделать, чтобы мы принимали Cat и Dog, а mouse нет

Ответить +2

сергей александрович Backend Developer — 6 августа 2020, 22:51

? super Cat & Dog

Ответить +4

Арман Уровень 37, Самара, Россия — 8 апреля 2021, 23:22

А разве можно классы перечислять через &?
После & можно писать только интерфейсы

Ответить 0

Александр Черенков Уровень 37, Бердск, Россия — 22 апреля 2021, 13:47

не делайте mouse наследником pet, что в общем логично

Ответить +1

Показать еще комментарии

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.