Professor Hans Noodles

41 уровень

15.07.2019 28384 25

Стирание типов

Статья из группы Java Developer

43181 участник

Вы в группе

Привет!

Мы продолжаем серию лекций о дженериках. Ранее мы разобрались в общих чертах, что это такое, и зачем нужно. Сегодня поговорим о некоторых особенностях дженериков и рассмотрим некоторые подводные камни при работе с ними. Поехали!

В прошлой лекции мы говорили о разнице между **Generic Types** и **Raw Types**. Если ты забыл, Raw Type — это класс-дженерик, из которого удалили его тип.

```
1  List list = new ArrayList();
```

Вот пример. Здесь мы не указываем, какого именно типа объекты будут помещаться в наш `List`.

Если попытаться создать такой `List` и добавить в него какие-то объекты мы увидим в IDEa предупреждение:

```
“Unchecked call to add(E) as a member of raw type of java.util.List”.
```

Но также мы говорили о том, что дженерики появились только в версии языка Java 5. К моменту ее выхода программисты успели написать кучу кода с использованием Raw Types, и чтобы он не перестал работать, возможность создания и работы

НАЧАТЬ ОБУЧЕНИЕ

Однако эта проблема оказалась гораздо обширнее.

Java-код, как ты знаешь, преобразуется в специальный байт-код, который потом выполняется виртуальной машиной Java.

И если бы в процессе перевода мы помещали в байт-код информацию о типах-параметрах, это сломало бы весь ранее написанный код, ведь до Java 5 никаких типов-параметров не существовало!

При работе с дженериками есть одна очень важная особенность, о которой необходимо помнить. Она называется “стирание типов” (type erasure).

Ее суть заключается в том, что внутри класса не хранится никакой информации о его типе-параметре.

Эта информация доступна только на этапе компиляции и стирается (становится недоступной) в runtime.

Если ты попытаешься положить объект не того типа в свой `List<String>`, компилятор выдаст ошибку. Этого как раз и добивались создатели языка, создавая дженерики — проверки на этапе компиляции.

Но когда весь написанный тобой Java-код превратится в байт-код, в нем не будет информации о типах-параметрах.

Внутри байт-кода твой список `List<Cat>` cats не будет отличаться от `List<String>` strings. В байт-коде ничто не будет говорить о том, что `cats` — это список объектов `Cat`. Информация об этом сотрется во время компиляции, и в байт код попадет только информация о том, что у тебя в программе есть некий список `List<Object> cats`.

Давай посмотрим как это работает:

```
1  public class TestClass<T> {
2
3     private T value1;
4     private T value2;
5
6     public void printValues() {
7         System.out.println(value1);
8         System.out.println(value2);
9     }
10
11    public static <T> TestClass<T> createAndAdd2Values(Object o1, Object o2) {
12        TestClass<T> result = new TestClass<>();
13        result.value1 = (T) o1;
14        result.value2 = (T) o2;
15        return result;
16    }
17
18    public static void main(String[] args) {
19        Double d = 22.111;
20        String s = "Test String";
21        TestClass<Integer> test = createAndAdd2Values(d, s);
22        test.printValues();
23    }
24 }
```

Мы создали собственный дженерик-класс `TestClass`.

Он довольно прост: по сути это небольшая “коллекция” на 2 объекта, которые помещаются туда сразу же при создании

НАЧАТЬ ОБУЧЕНИЕ

При выполнении метода `createAndAdd2Values()` должно произойти приведение двух переданных объектов `Object a` и `Object b` к нашему типу `T`, после чего они будут добавлены в объект `TestClass`.

В методе `main()` мы создаем `TestClass<Integer>`, то есть в качестве `T` у нас будет `Integer`.

Но при этом в метод `createAndAdd2Values()` мы передаем число `Double` и объект `String`.

Как ты думаешь, сработает ли наша программа? Ведь в качестве типа-параметра мы указали `Integer`, а `String` точно нельзя привести к `Integer`!

Давай запустим метод `main()` и проверим.

Вывод в консоль:

22.111
Test String

Неожиданный результат! Почему такое произошло?

Именно из-за стирания типов.

Во время компиляции кода информация о типе-параметре `Integer` нашего объекта `TestClass<Integer> test` стерлась.

Он превратился в `TestClass<Object> test`.

Наши параметры `Double` и `String` без проблем преобразовались в `Object` (а не в `Integer`, как мы того ожидали!) и спокойно добавились в `TestClass`.

Вот еще один простой, но очень показательный пример стирания типов:

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class Main {
5
6     private class Cat {
7
8     }
9
10    public static void main(String[] args) {
11
12        List<String> strings = new ArrayList<>();
13        List<Integer> numbers = new ArrayList<>();
14        List<Cat> cats = new ArrayList<>();
15
16        System.out.println(strings.getClass() == numbers.getClass());
17        System.out.println(numbers.getClass() == cats.getClass());
18
19    }
20 }
```

Вывод в консоль:

НАЧАТЬ ОБУЧЕНИЕ

Казалось бы, мы создали коллекции с тремя разными типами-параметрами — `String`, `Integer`, и созданный нами класс `Cat`.

Но во время преобразования в байт-код все три списка превратились в `List<Object>`, поэтому при выполнении программа говорит нам, что во всех трех случаях у нас используется один и тот же класс.

Стирание типов при работе с массивами и дженериками

Есть один очень важный момент, который необходимо четко понимать при работе с массивами и дженериками (например, `List`). Также его стоит учитывать при выборе структуры данных для твоей программы.

Дженерики подвержены стиранию типов. Информация о типе-параметре недоступна во время выполнения программы.

В отличие от них, массивы знают и могут использовать информацию о своем типе данных во время выполнения программы.

Попытка поместить в массив значение неверного типа приведет к исключению:

```
public class Main2 {

   public static void main(String[] args) {

       Object x[] = new String[3];
       x[0] = new Integer(222);
   }
}
```

Вывод в консоль:

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
```

Из-за того, что между массивами и дженериками есть такая большая разница, у них могут возникнуть проблемы с совместимостью.

Прежде всего, ты не можешь создать массив объектов-дженериков или даже просто типизированный массив.

Звучит немного непонятно?

Давай рассмотрим наглядно.

К примеру, ты не сможешь сделать в Java ничего из этого:

```
new List<T>[]
new List<String>[]
new T[]
```

Если мы попытаемся создать массив списков `List<String>`, получим ошибку компиляции generic array creation:

```
import java.util.List;

```

НАЧАТЬ ОБУЧЕНИЕ

```
5     public static void main(String[] args) {
6
7         //ошибка компиляции! Generic array creation
8         List<String>[] stringLists = new List<String>[1];
9     }
10 }
```

Но для чего это сделано? Почему создание таких массивов запрещено? Это все — для обеспечения типобезопасности.

Если бы компилятор позволял нам создавать такие массивы из объектов-дженериков, мы могли бы заработать кучу проблем.

Вот простой пример из книги Джошуа Блоха “Effective Java”:

```
1  public static void main(String[] args) {
2
3     List<String>[] stringLists = new List<String>[1];  //  (1)
4     List<Integer> intList = Arrays.asList(42, 65, 44);  //  (2)
5     Object[] objects = stringLists;  //  (3)
6     objects[0] = intList;  //  (4)
7     String s = stringLists[0].get(0);  //  (5)
8  }
```

Давай представим, что создание массива `List<String>[] stringLists` было бы разрешено, и компилятор бы не ругался.

Вот каких дел мы могли бы наворотить в этом случае:

В строке 1 мы создаем массив листов `List<String>[] stringLists`.
Наш массив вмещает в себя один `List<String>`.

В строке 2 мы создаем список чисел `List<Integer>`.

В строке 3 мы присваиваем наш массив `List<String>[]` в переменную `Object[] objects`. Язык Java позволяет это делать: в массив объектов `X` можно помещать и объекты `X`, и объекты всех дочерних классов `X`. Соответственно, в массив `Objects` можно поместить вообще все что угодно.

В строке 4 мы подменяем единственный элемент массива `objects (List<String>)` на список `List<Integer>`.

В результате мы поместили `List<Integer>` в наш массив, который предназначался только для хранения `List<String>`!

С ошибкой же мы столкнемся только когда код дойдет до строки 5. Во время выполнения программы будет выброшено исключение `ClassCastException`.

Поэтому запрет на создание таких массивов и был введен в язык Java — это позволяет нам избегать подобных ситуаций.

Научитесь программировать с нуля с JavaRush:
1200 задач, автопроверка решения и стиля кода
НАЧАТЬ ОБУЧЕНИЕ

Как можно обойти стирание типов?

Что ж, стирание типов мы изучили. Давай попробуем обмануть систему! :)

НАЧАТЬ ОБУЧЕНИЕ

У нас есть класс-дженерик `TestClass<T>`. Нам нужно создать в нем метод `createNewT()`, который будет создавать и возвращать новый объект типа `T`.

Но ведь это невозможно сделать, так? Вся информация о типе `T` будет стерта во время компиляции, и в процессе работы программы мы не сможем узнать, объект какого именно типа нам нужно создать.

На самом деле, есть один хитрый способ.

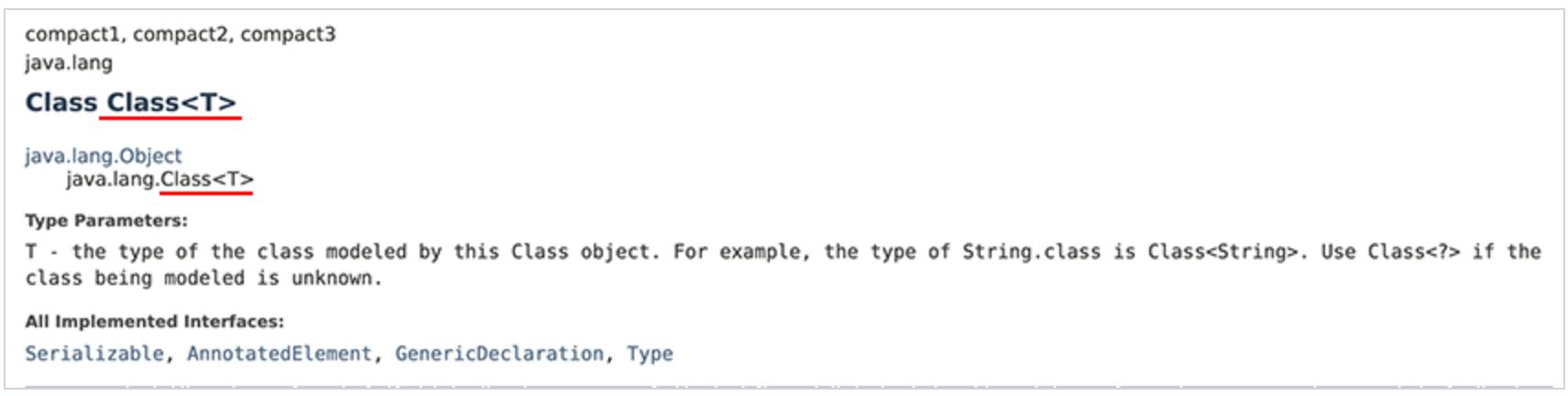
Ты наверняка помнишь, что в Java есть класс `Class`. Используя его, мы можем получить класс любого нашего объекта:

```
public class Main2 {

   public static void main(String[] args) {

       Class classInt = Integer.class;
       Class classString = String.class;

       System.out.println(classInt);
       System.out.println(classString);
   }
}
```

Вывод в консоль:

```
class java.lang.Integer
class java.lang.String
```

Но вот одна особенность, о которой мы не говорили. В документации Oracle ты увидишь, что класс Class — это дженерик!


compact1, compact2, compact3
java.lang

Class Class<T>

java.lang.Object
    java.lang.Class<T>

Type Parameters:

T - the type of the class modeled by this Class object. For example, the type of String.class is Class<String>. Use Class<?> if the class being modeled is unknown.

All Implemented Interfaces:

Serializable, AnnotatedElement, GenericDeclaration, Type

В документации написано: “T — это тип класса, моделируемого этим объектом Class”.

Если перевести это с языка документации на человеческий, это означает, что классом для объекта `Integer.class` является не просто `Class`, а `Class<Integer>`. Типом объекта `string.class` является не просто `Class`, `Class<String>`, и т.д.

Если все еще непонятно, попробуй добавить тип-параметр к предыдущему примеру:

```
public class Main2 {

   public static void main(String[] args) {

       Class<Integer> classInt = Integer.class;
```

НАЧАТЬ ОБУЧЕНИЕ

```
8
9
10          Class<String> classString = String.class;
11          //ошибка компиляции!
12          Class<Double> classString2 = String.class;
13      }
14  }
```

И вот теперь, используя это знание, мы можем обойти стирание типов и решить нашу задачу!

Попробуем получить информацию о типе-параметре. Его роль будет играть класс `MySecretClass`:

```
1   public class MySecretClass {
2
3       public MySecretClass() {
4
5           System.out.println("Объект секретного класса успешно создан!");
6       }
7   }
```

А вот как мы используем на практике наше решение:

```
1   public class TestClass<T> {
2
3       Class<T> typeParameterClass;
4
5       public TestClass(Class<T> typeParameterClass) {
6           this.typeParameterClass = typeParameterClass;
7       }
8
9       public T createNewT() throws IllegalAccessException, InstantiationException {
10          T t = typeParameterClass.newInstance();
11          return t;
12      }
13
14      public static void main(String[] args) throws InstantiationException, IllegalAccessException {
15
16          TestClass<MySecretClass> testString = new TestClass<>(MySecretClass.class);
17          MySecretClass secret = testString.createNewT();
18
19      }
20  }
```

Вывод в консоль:

```
Объект секретного класса успешно создан!
```

Мы просто передали нужный класс-параметр в конструктор нашего класса-дженерика:

НАЧАТЬ ОБУЧЕНИЕ

Благодаря этому мы сохранили информацию о типе-параметре и уберегли ее от стирания. В итоге мы смогли создать объект `T`! :)

На этом сегодняшняя лекция подходит к концу.

О стирании типов всегда необходимо помнить при работе с дженериками. Выглядит это дело не очень удобно, но нужно понимать — дженерики не были частью языка Java при его создании. Это позже прикрученная возможность, которая помогает нам создавать типизированные коллекции и отлавливать ошибки на этапе компиляции.

В некоторых других языках, где дженерики появлялись с первой версии, стирание типов отсутствует (например, в C#).

Впрочем, мы не закончили изучение дженериков! На следующей лекции ты познакомишься с еще несколькими особенностями работы с ними. А пока было бы неплохо решить пару задач! :)

+118

Комментарии (25) популярные новые старые

JavaCoder

Введите текст комментария

Руслан Уровень 42 31 мая, 12:39

В этом примере:
List<String> strings = new ArrayList<>();
List<Cat> cats = new ArrayList<>();
System.out.println(strings.getClass() == cats.getClass()); == true.
Но strings.getClass и cats.getClass будут равны ArrayList. Т.е. мы проверяем, равен ли ArrayList ему же, а не String и Cat. Или я чего-то не понял?

Ответить 0

Марат Гарипов Уровень 48, Россия 24 июня, 23:36

Насколько я понимаю, ArrayList<> = это тип объекта, который присваивается переменной как раз таки класса List с параметрами <String> или <Cat>, и после компиляции программы из-за стирания типов эти параметры и пропадают, а остаются только List'ы, которые, соответственно, равны

Ответить 0

Серега Батенин Уровень 16, Москва, Россия 3 апреля, 19:17

Как говорит одна моя знакомая "Очень интересно, но нифига не понятно". Вот и мне пока мало что понятно(

Ответить +5

Мирослав Уровень 27, Тбилиси, Грузия 12 мая, 11:28

)))

Ответить +1

LuneFox инженер по сопровождению в **BIFIT** EXPERT 26 января, 17:25

Мне кажется, или почти все лекции от профессора заканчиваются фразой "А пока было бы неплохо решить пару задач!", даже если следующей лекцией идёт не решение задач?)

Ответить +2

Igor Java/Kotlin Developer 12 июня 2021, 15:17

НАЧАТЬ ОБУЧЕНИЕ

Anonymous #3068853 Уровень 3 — 28 мая, 02:49

А, ну тогда ясно

Ответить +1

zdRusty Уровень 36, Оренбург, Российская Федерация — 21 февраля 2021, 13:32

Сколько, интересно, еще появится костылей для поддержки обратной совместимости по мере прикручивания новых фич.
Избавиться от них можно лишь написав с нуля какую-нибудь Java 2.0. Только тогда все равно придется поддерживать терабайты старого кода, на мертвом (в этом случае) языке. Ситуация, похоже, печальная.

Ответить +14

Денис Уровень 29, Харьков, Украина — 25 марта 2021, 21:46

как же я с тобой согласен...

Ответить 0

Pineapple Уровень 45, Абакан, Россия — 18 августа 2021, 10:04

ну судя по всему C# это и есть java 2.0

Ответить 0

profitroll Software Developer в **Алроса** — 14 сентября 2021, 16:52

Так-то текущая реализация - это и есть Java2))

Ответить 0

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

А если в C# прикручиваются новые фичи, не происходит той же ситуации с обратной совместимостью? В каком языке новые фичи вводятся без поддержки обратной совместимости старых программ? (Это вопрос настоящий, а не риторический, если что.)

Ответить 0

Андрей Овчаренко Уровень 41, Москва — 24 февраля, 22:25

Поговаривают дотнет вполне себе иногда забивает на обратную совместимость

Ответить 0

Owpk Уровень 36, Иркутск, Россия — 9 января 2021, 08:52

/ Комментарий удален */*

Ответить 0

funbiscuit Уровень 41, Россия — 18 января 2021, 13:15

Потому что вызов test.value2.getClass() заменяется компилятором на

```
((Integer) test.value2).getClass()
```

Ответить 0

Pig Man Главная свинья в **Свинарнике** — 12 февраля 2021, 23:02

Прекрасный пример стирания типа комментария

Ответить +6

Leonid Java Developer в **ProgForce** EXPERT — 8 июня 2020, 17:19

Пример из книги Джошуа Блоха “Effective Java”, как мне кажется, показывает не проблему дженериков и стирания типов, а проблему того, что массив любого типа мы можем присвоить в переменную Object[] objects. Ипотом его поменять. В результате я могу замутить точно такой же код приводящий к Exeption без дженериков:

```
public static void main(String[] args) {
    String[] stringLists = new String[1];  //  (1)

    Object[] objects = stringLists;  //  (3)
    objects[0] = 1;  //  (4)
    String s = stringLists[0];  //  (5)
}
```

В колекциях, кстати, эта проблема массивов решена благодаря именно дженерикам:

```
List<String> strings = new ArrayList<String>();
// ошибка компиляции!
List<Object> objects = strings;
```

Ответить +5

НАЧАТЬ ОБУЧЕНИЕ

В предыдущей статье про использование "varargs и дженериков" упоминается эта проблема. Там она называется "загрязнение кучи
"

Ответить

0

Сергей З. Уровень 27, Москва

3 июня 2020, 13:42

/ Комментарий удален */*

Ответить

0

Leonid Java Developer в **ProgForce** EXPERT

8 июня 2020, 17:14

В смысле "как"? Мы же внутри класса находимся.

Ответить

0

Dzmitry Mikhalka Java Developer

1 мая 2020, 18:57

имя переменной TestClass<MySecretClass> **testString** в последнем примере сбивает с толку :) а так отличная статья

Ответить

0

Simon Bashkirov Уровень 40

25 марта 2020, 12:37

Есть еще один обходной маневр против стирания типов: clazz.getGenericSuperclass()
Хотим мы сделать фабрику, которая выдаёт инстансы чего угодно.
Пишем

```
//Делаем фабрику абстрактной
// т.к. этот финт корректно работает только при использовании её наследников
abstract class Factory<T> {
    public T createInstance() {
        // получаем класс наследника
        Class clazz = this.getClass();
        // получаем его супер класс, т.е. нашу фабрику Factory<T>
        Type type = clazz.getGenericSuperclass();
        //Здесь JVM почему-то знает конкретный тип-параметр
        ParameterizedType parameterizedType = (ParameterizedType) type;
        Type[] args = parameterizedType.getActualTypeArguments();
        //Первый аргумент - и есть класс параметра конкретного наследника фабрики
        Class argClass = (Class) args[0];
        try {
            //Здесь варнинг, игнорируем его
            T instance = (T) argClass.newInstance();
            return instance;
        } catch (InstantiationException | IllegalAccessException ex) {
            return null;
        }
    }
}

В клиентском коде:
 // создаём анонимного наследника нашей фабрики с указанным конкретным типом.
Factory<JPanel> panelFactory = new Factory<JPanel>(){};
//создали панельку, вуаля
JPanel m = f.createInstance();
```

Ответить

+4

Interstellar Java Developer в **EPAM** EXPERT

10 июля 2020, 18:17

В конце наверное не f.createInstance(), а panelFactory.createInstance()

Ответить

0

Показать еще комментарии

RUSH

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.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский

"Программистами не рождаются" © 2022 JavaRush