Marvinorez 19 июля 2018

# Пришел, увидел, обобщил: погружаемся в Java Generics

Java Generics — это одно из самых значительных изменений за всю историю языка Java. «Дженерики», доступные с Java 5, сделали использование Java Collection Framework проще, удобнее и безопаснее. Ошибки, связанные с некорректным использованием типов, теперь обнаруживаются на этапе компиляции. Да и сам язык Java стал еще безопаснее. Несмотря на кажущуюся простоту обобщенных типов, многие разработчики сталкиваются с трудностями при их использовании. В этом посте я расскажу об особенностях работы с Java Generics, чтобы этих трудностей у вас было поменьше. Пригодится, если вы не гуру в дженериках, и поможет избежать много трудностей при погружении в тему.


## Работа с коллекциями

Предположим, банку нужно подсчитать сумму сбережений на счетах клиентов. До появления «дженериков» метод вычисления суммы выглядел так:

```
public long getSum(List accounts) {
    long sum = 0;

    for (int i = 0, n = accounts.size(); i < n; i++) {
        Object account = accounts.get(i);
        if (account instanceof Account) {
```

```
            sum += ((Account) account).getAmount();
        }
    }

    return sum;
}
```

Мы итерировались, пробегались по списку аккаунтов и проверяли, действительно ли элемент из этого списка является экземпляром класса `Account` — то есть счетом пользователя. Выполняли приведение типа нашего объекта класса `Account` и метод `getAmount`, который возвращал сумму на этом счете. Дальше все это суммировали и возвращали итоговую сумму. Требовалось выполнить два действия:

```
if (account instanceof Account) { // (1)
```

```
sum += ((Account) account).getAmount(); // (2)
```

Если не сделать проверку (`instanceof`) на принадлежность к классу `Account`, то на втором этапе возможен `ClassCastException` – то есть аварийное завершение программы. Поэтому такая проверка была обязательной.

С появлением Generics необходимость в проверке и приведении типа отпала:

```
public long getSum2(List<Account> accounts) {
    long sum = 0;

    for (Account account : accounts) {
```

```
            sum += account.getAmount();
        }

        return sum;
    }
```

Теперь метод

```
getSum2(List<Account> accounts)
```

принимает в качестве аргументов только список объектов класса `Account`. Это ограничение указано в самом методе, в его сигнатуре, программист просто не может передать никакой другой список — только список клиентских счетов.

Нам не нужно выполнять проверку типа элементов из этого списка: она подразумевается описанием типа у параметра метода

```
List<Account> accounts
```

(можно прочитать как `список объектов класса Account`). И компилятор выдаст ошибку, если что-то пойдет не так — то есть если кто-то попробует передать в этот метод список объектов, отличных от класса `Account`.

Во второй строчке проверки необходимость тоже отпадала. Если потребуется, приведение типов (`casting`) будет сделано на этапе компиляции.

## Принцип подстановки

Принцип подстановки Барбары Лисков – специфичное определение подтипа в объектно-ориентированном программировании. Идея Лисков о «подтипе» дает определение понятия

замещения: если `S` является подтипом `T`, тогда объекты типа `T` в программе могут быть замещены объектами типа `S` без каких-либо изменений желательных свойств этой программы.

| **Тип** | **Подтип** |
|---|---|
| *Number* | *Integer* |
| *List*<E> | *ArrayList*<E> |
| *Collection*<E> | *List*<E> |
| *Iterable*<E> | *Collection*<E> |

*Примеры отношения тип/подтип*

Вот пример использования принципа подстановки в Java:

```
Number n = Integer.valueOf(42);
List<Number> aList = new ArrayList<>();
Collection<Number> aCollection = aList;
Iterable<Number> iterable = aCollection;
```

`Integer` является подтипом `Number`, следовательно, переменной `n` типа `Number` можно присвоить значение, которое возвращает метод `Integer.valueOf(42)`.

## Ковариантность, контравариантность и инвариантность

Сначала немного теории. Ковариантность — это сохранение иерархии наследования исходных типов в производных типах в том же порядке. Например, если *Кошка* — это подтип *Животные*, то *Множество<Кошки>* — это подтип *Множество<Животные>*. Следовательно, с учетом принципа подстановки можно выполнить такое присваивание:

*Множество<Животные> = Множество<Кошки>*

Контравариантность — это обращение иерархии исходных типов на противоположную в производных типах. Например, если *Кошка* — это подтип `Животные`, то *Множество<Животные>* — это подтип *Множество<Кошки>*. Следовательно, с учетом принципа подстановки можно выполнить такое присваивание:

*Множество<Кошки> = Множество<Животные>*

Инвариантность — отсутствие наследования между производными типами. Если *Кошка* — это подтип *Животные*, то *Множество<Кошки>* не является подтипом *Множество<Животные>* и *Множество<Животные>* не является подтипом *Множество<Кошки>*.

**Массивы в Java ковариантны**. Тип `S[]` является подтипом `T[]`, если `S` — подтип `T`. Пример присваивания:

```
String[] strings = new String[] {"a", "b", "c"};
Object[] arr = strings;
```

Мы присвоили ссылку на массив строк переменной `arr`, тип которой – «`массив объектов`». Если бы массивы не были ковариантными, нам бы это сделать не удалось. Java позволяет это сделать, программа скомпилируется и выполнится без ошибок.

```
arr[0] = 42; // ArrayStoreException. Проблема обнаружилась на этапе выполнения программ
```

Но если мы попытаемся изменить содержимое массива через переменную `arr` и запишем туда число 42, то получим `ArrayStoreException` на этапе выполнения программы, поскольку 42 является не строкой, а числом. В этом недостаток ковариантности массивов Java: мы не можем выполнить проверки на этапе компиляции, и что-то может сломаться уже в рантайме.

**«Дженерики» инвариантны.** Приведем пример:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<Number> nums = ints; // compile-time error. Проблема обнаружилась на этапе компиля
nums.set(2, 3.14);
assert ints.toString().equals("[1, 2, 3.14]");
```

Если взять список целых чисел, то он не будет являться ни подтипом типа `Number`, ни каким-либо другим подтипом. Он является только подтипом самого себя. То есть `List <Integer>` — это `List<Integer>` и ничего больше. Компилятор позаботится о том, чтобы переменная `ints`, объявленная как список объектов класса *Integer*, содержала только объекты класса `Integer` и ничего кроме них. На этапе компиляции производится проверка, и у нас в рантайме уже ничего не упадет.

## Wildcards

Всегда ли Generics инварианты? Нет. Приведу примеры:

```
List<Integer> ints = new ArrayList<Integer>();
List<? extends Number> nums = ints;
```

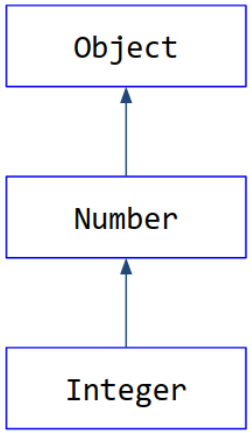
Это ковариантность. `List<Integer>` — подтип `List<? extends Number>`

```
List<Number> nums = new ArrayList<Number>();
List<? super Integer> ints = nums;
```

Это контравариантность. `List<Number>` является подтипом `List<? super Integer>`.

Запись вида "`? extends ...`" или "`? super ...`" — называется wildcard или символом подстановки, с верхней границей (`extends`) или с нижней границей (`super`). `List<? extends Number>` может содержать объекты, класс которых является `Number` или наследуется от `Number`. `List<? super Number>` может содержать объекты, класс которых `Number` или у которых `Number` является наследником (супертип от `Number`).

| | |
|---|---|
| | extends *B* — символ подстановки с указанием верхней границы<br>super *B* — символ подстановки с указанием нижней границы<br>где *B* — представляет собой границу<br><br>Запись вида $T_2$ <= $T_1$ означает, что набор типов описываемых $T_2$ является подмножеством набора типов описываемых $T_1$<br><br>т.е.<br>Number <=? extends Object |




? extends Number <=? extends Object

и

? super Object <=? super Number

*Более математическая интерпретация темы*

Пара задачек для проверки знаний:

1. Почему в примере ниже compile-time error? Какое значение можно добавить в список `nums`?

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(3.14); // compile-time error
```

**Ответ**

Если контейнер объявлен с wildcard `? extends`, то можно только читать значения. В список нельзя ничего добавить, кроме `null`. Для того чтобы добавить объект в список нам нужен другой тип wildcard — `? super`

2. Почему нельзя получить элемент из списка ниже?

```
public static <T> T getFirst(List<? super T> list) {
   return list.get(0); // compile-time error
}
```

**Ответ**

Нельзя прочитать элемент из контейнера с wildcard `? super`, кроме объекта класса `Object`

```
public static <T> Object getFirst(List<? super T> list) {
   return list.get(0);
}
```

## The Get and Put Principle или PECS (Producer Extends Consumer Super)

Особенность wildcard с верхней и нижней границей дает дополнительные фичи, связанные с безопасным использованием типов. Из одного типа переменных можно только читать, в другой — только вписывать (исключением является возможность записать `null` для `extends` и прочитать `Object` для `super`). Чтобы было легче запомнить, когда какой wildcard использовать, существует принцип PECS — Producer Extends Consumer Super.

- Если мы объявили *wildcard c extends*, то это *producer*. Он только «продюсирует», предоставляет элемент из контейнера, а сам ничего не принимает.
- Если же мы объявили *wildcard c super* — то это *consumer*. Он только принимает, а предоставить ничего не может.

Рассмотрим использование Wildcard и принципа PECS на примере метода copy в классе java.util.Collections.

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
…
}
```

Метод осуществляет копирование элементов из исходного списка `src` в список `dest`. `src` — объявлен с wildcard `? extends` и является продюсером, а `dest` — объявлен с wildcard `? super` и является потребителем. Учитывая ковариантность и контравариантность wildcard, можно скопировать элементы из списка `ints` в список `nums`:

```
List<Number> nums = Arrays.<Number>asList(4.1F, 0.2F);
List<Integer> ints = Arrays.asList(1,2);
Collections.copy(nums, ints);
```

Если же мы по ошибке перепутаем параметры метода copy и попытаемся выполнить копирование из списка `nums` в список `ints`, то компилятор не позволит нам это сделать:

```
Collections.copy(ints, nums); // Compile-time error
```

### <?> и Raw типы

Ниже приведен wildcard с неограниченным символом подстановки. Мы просто ставим `<?>`, без ключевых слов `super` или `extends`:

```
static void printCollection(Collection<?> c) {
   // a wildcard collection
   for (Object o : c) {
       System.out.println(o);
   }
}
```

На самом деле такой «неограниченный» wildcard все-таки ограничен, сверху. `Collection<?>` — это тоже символ подстановки, как и "`? extends Object`". Запись вида `Collection<?>` равносильна `Collection<? extends Object>` , а значит — коллекция может содержать объекты любого класса, так как все классы в Java наследуются от `Object` – поэтому подстановка называется неограниченной.

Если мы опустим указание типа, например, как здесь:

```
ArrayList arrayList = new ArrayList();
```

то, говорят, что `ArrayList` — это `Raw` тип параметризованного *ArrayList<T>*. Используя Raw типы, мы возвращаемся в эру до дженериков и сознательно отказываемся от всех фич, присущих параметризованным типам.

Если мы попытаемся вызвать параметризованный метода у Raw типа, то компилятор выдаст нам предупреждение «Unchecked call». Если мы попытаемся выполнить присваивание ссылки на параметризованный тип Raw типу, то компилятор выдаст предупреждение «Unchecked assignment». Игнорирование этих предупреждений, как мы увидим позже, может привести к ошибкам во время выполнения нашего приложения.

```
ArrayList<String> strings = new ArrayList<>();
ArrayList arrayList = new ArrayList();
arrayList = strings; // Ok
strings = arrayList; // Unchecked assignment
arrayList.add(1); //unchecked call
```

## Wildcard Capture

Попробуем теперь реализовать метод, выполняющий перестановку элементов списка в обратном порядке.

```
public static void reverse(List<?> list);

// Ошибка!
public static void reverse(List<?> list) {
  List<Object> tmp = new ArrayList<Object>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1)); // compile-time error
  }
}
```

Ошибка компиляции возникла, потому что в методе `reverse` в качестве аргумента принимается список с неограниченным символом подстановки `<?>` .
`<?>` означает то же что и `<? extends Object>`. Следовательно, согласно принципу PECS, `list` – это `producer`. А `producer` только продюсирует элементы. А мы в цикле `for` вызываем метод `set()`, т.е. пытаемся записать в `list`. И поэтому упираемся в защиту Java, что не позволяет установить какое-то значение по индексу.

Что делать? Нам поможет паттерн `Wildcard Capture`. Здесь мы создаем обобщенный метод `rev`. Он объявлен с переменной типа `T`. Этот метод принимает список типов `T`, и мы можем сделать сет.

```
public static void reverse(List<?> list) {
  rev(list);
}

private static <T> void rev(List<T> list) {
  List<T> tmp = new ArrayList<T>(list);
  for (int i = 0; i < list.size(); i++) {
    list.set(i, tmp.get(list.size()-i-1));
  }
}
```

Теперь у нас все скомпилируется. Здесь произошел захват символа подстановки (wildcard capture). При вызове метода `reverse(List<?> list)` в качестве аргумента передается список каких-то объектов (например, строк или целых чисел). Если мы можем захватить тип этих объектов и присвоить его переменной типа `X`, то можем заключить, что `T` является `X`.

Более подробно о `Wildcard Capture` можно прочитать здесь и здесь.

### Вывод

Если необходимо читать из контейнера, то используйте wildcard с верхней границей "`? extends`". Если необходимо писать в контейнер, то используйте wildcard с нижней границей "`? super`". Не используйте wildcard, если нужно производить и запись, и чтение.

Не используйте `Raw` типы! Если аргумент типа не определен, то используйте wildcard `<?>`.

## Переменные типа

Когда мы записываем при объявлении класса или метода идентификатор в угловых скобках, например `<T>` или `<E>`, то создаем *переменную типа*. Переменная типа — это неквалифицированный идентификатор, который можно использовать в качестве типа в теле класса или метода. Переменная типа может быть ограничена сверху.

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {
  T candidate = coll.iterator().next();
  for (T elt : coll) {
    if (candidate.compareTo(elt) < 0) candidate = elt;
  }
  return candidate;
}
```

В этом примере выражение `T extends Comparable<T>` определяет `T` (переменную типа), ограниченную сверху типом `Comparable<T>`. В отличие от wildcard, переменные типа могут быть ограничены только сверху (только `extends`). Нельзя записать `super`. Кроме того, в этом примере `T` зависит от самого себя, это называется `recursive bound` — рекурсивная граница.

Вот еще пример из класса Enum:

```
public abstract class Enum<E extends Enum<E>>implements Comparable<E>, Serializable
```

Здесь класс Enum параметризован типом E, который является подтипом от `Enum<E>`.

### Multiple bounds (множественные ограничения)

`Multiple Bounds` – множественные ограничения. Записывается через символ "`&`", то есть мы говорим, что тип, представленный переменной типа `T`, должен быть ограничен сверху классом `Object` и интерфейсом `Comparable`.

```
<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

Запись `Object & Comparable<? super T&gt` образует тип пересечения `Multiple Bounds`. Первое ограничение — в данном случае `Object` — используется для `erasure`, процесса затирания типов. Его выполняет компилятор на этапе компиляции.

### Вывод

Переменная типа может быть ограничена только сверху одним или несколькими типами. В случае множественного ограничения левая граница (первое ограничение) используется в процессе затирания (Type Erasure).

## Type Erasure

Type Erasure представляет собой отображение типов (возможно, включая параметризованные типы и переменные типа) на типы, которые никогда не являются параметризованными типами или переменными типами. Мы записываем затирание типа `T` как `|T|`.

Отображение затирания определяется следующим образом:

- Затиранием параметризованного типа *G*<*T1*,...,*Tn*> является |*G*|
- Затиранием вложенного типа *T.C* является |*T*|.*C*
- Затиранием типа массива *T[]* является |*T*|[]
- Затиранием переменной типа является затирание ее левой границы
- Затиранием любого иного типа является сам этот тип

В процессе выполнения Type Erasure (затирания типов) компилятор производит следующие действия:

- добавляет приведение типов для обеспечения type safety, если это необходимо
- генерирует Bridge методы для сохранения полиморфизма

| **T (Тип)** | **\|T\| (Затирание типа)** |
|---|---|
| *List< Integer>, List< String>, List< List< String>>* | *List* |
| *List< Integer>[]* | *List[]* |
| *List* | *List* |

| | |
|---|---|
| *int* | *int* |
| *Integer* | *Integer* |
| *<T extends Comparable<T>>* | *Comparable* |
| *<T extends Object & Comparable<? super T>>* | *Object* |
| *LinkedCollection<E>.Node* | *LinkedCollection.Node* |

Эта таблица показывает, во что превращаются разные типы в процессе затирания, Type Erasure.

На скриншоте ниже два примера программы:

```
public class Name implements Comparable<Name> {
    private final String value;
    public Name(String value) {
        this.value = value;
    }
    @Override
    public String toString() {
        return "value='" + value + '\'';
    }
    @Override
    public int compareTo(Name o) {
        return value.compareTo(o.value);
    }
    public int compareTo(Object o) {
        return value.compareTo(((Name)o).value);
    }
}
```

```
public class Name implements Comparable {
    private final String value;
    public Name(String value) {
        this.value = value;
    }
    @Override
    public String toString() {
        return "value='" + value + '\'';
    }
    public int compareTo(Name o) {
        return value.compareTo(o.value);
    }
    @Override
    public int compareTo(Object o) {
        return value.compareTo(((Name)o).value);
    }
}
```

Разница между ними в том, что слева происходит compile-time error, а справа все компилируется

без ошибок. Почему?

**Ответ**

В Java два разных метода не могут иметь одну и ту же сигнатуру. В процессе Type Erasure компилятор добавит bridge-метод `public int compareTo(Object o)`. Но в классе уже содержится метод с такой сигнатурой, что и вызовет ошибку во время компиляции.

Скомпилируем класс Name, удалив метод `compareTo(Object o)`, и посмотрим на получившийся байткод с помощью javap:

```
# javap Name.class
Compiled from "Name.java"
public class ru.sberbank.training.generics.Name implements java.lang.Comparable<ru.s
  public ru.sberbank.training.generics.Name(java.lang.String);
  public java.lang.String toString();
  public int compareTo(ru.sberbank.training.generics.Name);
  public int compareTo(java.lang.Object);
}
```

Видим, что класс содержит метод `int compareTo(java.lang.Object)` , хотя мы его удалили из исходного кода. Это и есть bridge метод, который добавил компилятор.

## Reifiable типы

В Java мы говорим, что тип является `reifiable`, если информация о нем полностью доступна во время выполнения программы. В reifiable типы входят:

- Примитивные типы (*int*, *long*, *boolean*)
- Непараметризованные (необобщенные) типы (*String*, *Integer*)
- Параметризованные типы, параметры которых представлены в виде unbounded wildcard (неограниченных символов подстановки) (*List<?>*, *Collection<?>*)
- *Raw* (несформированные) типы (*List*, *ArrayList*)
- Массивы, компоненты которых — Reifiable типы (*int[]*, *Number[]*, *List<?>[]*, *List[*)

Почему информация об одних типах доступна, а о других нет? Дело в том, что из-за процесса затирания типов компилятором информация о некоторых типах может быть потеряна. Если она потерялась, то такой тип будет уже не reifiable. То есть она во время выполнения недоступна. Если доступна – соответственно, reifiable.

Решение не делать все обобщенные типы доступными во время выполнения — это одно из наиболее важных и противоречивых проектных решений в системе типов Java. Так сделали, в первую очередь, для совместимости с существующим кодом. За миграционную совместимость пришлось платить — полная доступность системы обобщенных типов во время выполнения невозможна.

Какие типы не являются reifiable:

- Переменная типа (*T*)
- Параметризованный тип с указанным типом параметра (*List<Number> ArrayList<String>*, *List<List<String>>*)
- Параметризованный тип с указанной верхней или нижней границей (*List<? extends Number>, Comparable<? super String>*). Но здесь стоит оговориться: *List<? extends Object>* — **не** reifiable, а *List<?>* — reifiable

И еще одна задачка. Почему в примере ниже нельзя создать параметризованный Exception?

```
class MyException<T> extends Exception {
   T t;
}
```

**Ответ**

Каждое catch выражение в try-catch проверяет тип полученного исключения во время выполнения программы (что равносильно instanceof), соответственно, тип должен быть Reifiable. Поэтому Throwable и его подтипы не могут быть параметризованы.

```
class MyException<T> extends Exception {// Generic class may not extend `java.lang.T
   T t;
}
```

## Unchecked Warnings

Компиляция нашего приложения может выдать так называемый `Unchecked Warning` — предупреждение о том, что компилятор не смог корректно определить уровень безопасности использования наших типов. Это не ошибка, а предупреждение, так что его можно пропустить. Но

желательно все-так исправить, чтобы избежать проблем в будущем.

## Heap Pollution

Как мы упомянули ранее, присваивание ссылки на Raw тип переменной параметризованного типа, приводит к предупреждению «Unchecked assignment». Если мы проигнорируем его, то возможна ситуация под названием "`Heap Pollution`" (загрязнение кучи). Вот пример:

```
static List<String> t() {
   List l = new ArrayList<Number>();
   l.add(1);
   List<String> ls = l; // (1)
   ls.add("");
   return ls;
}
```

В строке (1) компилятор предупреждает об «Unchecked assignment».

Нужно привести и другой пример «загрязнения кучи» — когда у нас используются параметризованные объекты. Кусок кода ниже наглядно показывает, что недопустимо использовать параметризованные типы в качестве аргументов метода с использованием `Varargs`. В данном случае параметр метода m – это `List<String>...`, т.е. фактически, массив элементов типа `List<String>`. Учитывая правило отображения типов при затирании, тип `stringLists` превращается в массив raw списков (`List[]`), т.е. можно выполнить присваивание `Object[] array = stringLists;` и после записать в `array` объект, отличный от списка строк (1), что вызовет `ClassCastException` в строке (2).

```
static void m(List<String>... stringLists) {
   Object[] array = stringLists;
   List<Integer> tmpList = Arrays.asList(42);
   array[0] = tmpList; // (1)
   String s = stringLists[0].get(0); // (2)
}
```

Рассмотрим еще один пример:

```
ArrayList<String> strings = new ArrayList<>();
ArrayList arrayList = new ArrayList();
arrayList = strings; // (1) Ok
arrayList.add(1); // (2) unchecked call
```

Java разрешает выполнить присваивание в строке (1). Это необходимо для обеспечения обратной совместимости. Но если мы попытаемся выполнить метод `add` в строке (2), то получим предупреждение `Unchecked call` — компилятор предупреждает нас о возможной ошибке. В самом деле, мы же пытаемся в список строк добавить целое число.

## Reflection

Хотя при компиляции параметризованные типы подвергаются процедуре стирания (type erasure), кое-какую информацию мы можем получить с помощью Reflection.

- Все reifiable доступны через механизм Reflection

- Информация о типе полей класса, параметров методов и возвращаемых ими значений доступна через Reflection.

Если мы хотим через Reflection получить информацию о типе объекта и этот тип не `Reifiable`, то у нас ничего не получится. Но, если, например, этот объект нам вернул какой-то метод, то мы можем получить тип возвращаемого этим методом значения:

```
java.lang.reflect.Method.getGenericReturnType()
```

С появлением Generics класс `java.lang.Class` стал параметризованным. Рассмотрим вот этот код:

```
List<Integer> ints = new ArrayList<Integer>();
Class<? extends List> k = ints.getClass();
assert k == ArrayList.class;
```

Переменная `ints` имеет тип `List<Integer>` и она содержит ссылку на объект типа `ArrayList<Integer>`. Тогда `ints.getClass()` вернёт объект типа `Class<ArrayLis>`, так как `List<Integer>` затирается в `List`. Объект типа `Class<ArrayList>` можно присвоить переменной `k` типа `Class<? extends List>`, согласно ковариантности символов подстановки? `extends`. А `ArrayList.class` возвращает объект типа `Class<ArrayList>`.

### Вывод

Если информация о типе доступна во время выполнения программы, то такой тип называется Reifiable. К Reifiable типам относятся: примитивные типы, непараметризованные типы, параметризованные типы с неограниченным символом подстановки, Raw типы и массивы,

элементы которых являются reifiable.

Игнорирование Unchecked Warnings может привести к «загрязнению кучи» и ошибкам во время выполнения программы.

Reflection не позволяет получить информацию о типе объекта, если он не Reifiable. Но Reflection позволяет получить информацию о типе возвращаемого методом значения, о типе аргументов метода и о типе полей класса.

## Type Inference

Термин можно перевести как «Вывод типа». Это возможность компилятора определять (выводить) тип из контекста. Вот пример кода:

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

С появлением даймонд-оператора в Java 7 мы можем не указывать тип у `ArrayList`:

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

Компилятор выведет тип `ArrayList` из контекста – `List<Integer>`. Этот процесс и называется `type inference`.

В Java 8 сильно усовершенствовали механизм выведения типа благодаря JEP 101.
В общем случае процесс получения информации о неизвестных типах именуется выводом типа Type Inference. На верхнем уровне вывод типа можно разделить на три процесса:

- Приведение (reduction)

- Объединение (incorporation)
- Разрешение (resolution)

Эти процессы тесно взаимодействуют: приведение может запустить объединение, объединение может привести к дальнейшему приведению, а разрешение — к дальнейшему объединению. Детальное описание механизма выведения типа доступно в спецификации языка, где ему посвящена целая глава. Мы же вернемся к JEP 101 и рассмотрим какие цели он преследовал.

Предположим у нас есть вот такой класс, который описывает связный список:

```
class List<E> {
   static <Z> List<Z> nil() { ... };
   static <Z> List<Z> cons(Z head, List<Z> tail) { ... };
   E head() { ... }
}
```

Результат обобщенного метода `List.nil()` может быть выведен из правой части:

```
List<String> ls = List.nil();
```

Механизм выбора типа компилятором показывает, что аргумент типа для вызова `List.nil()` действительно `String` — это работает в JDK 7, все хорошо.

Выглядит разумно, что компилятор также должен иметь возможность вывести тип, когда результат такого вызова обобщенного метода передается другому методу в качестве аргумента, например:

```
List.cons(42, List.nil()); //error: expected List<Integer>, found List<Object>
```

В JDK 7 мы получили бы compile-time error. А в JDK 8 скомпилируется. Это и есть первая часть JEP-101, его первая цель — вывод типа в позиции аргумента. Единственная возможность осуществить это в версиях до JDK 8 — использовать явный аргумент типа при вызове обобщенного метода:

```
List.cons(42, List.<Integer>nil());
```

Вторая часть JEP-101 говорит о том, что неплохо бы выводить тип в цепочке вызовов обобщенных методов, например:

```
String s = List.nil().head(); //error: expected String, found Object
```

Но данная задача не решена до сих пор, и вряд ли в ближайшее время появится такая функция. Возможно, в будущих версиях JDK необходимость в этом исчезнет, но пока нужно указывать аргументы вручную:

```
String s = List.<String>nil().head();
```

После выхода JEP 101 на StackOverflow появилось множество вопросов по теме. Программисты спрашивают, почему код, который выполнялся на 7-й версии, на 8-й выполняется иначе – или вообще не компилируется? Вот пример такого кода:

```
class Test {
   static void m(Object o) {
       System.out.println("one");
```

```
    }

    static void m(String[] o) {
        System.out.println("two");
    }

    static <T> T g() {
        return null;
    }

    public static void main(String[] args) {
        m(g());
    }
}
```

Посмотрим на байт-код после компиляции на JDK1.8:

```
  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: invokestatic  #6                  // Method g:()Ljava/lang/Object;
         3: checkcast     #7                  // class "[Ljava/lang/String;"
         6: invokestatic  #8                  // Method m:([Ljava/lang/String;)V
         9: return
      LineNumberTable:
        line 15: 0
        line 16: 9
```

Инструкция под номером 0 выполняет вызов метода `g:()Ljava/lang/Object;` Метод возвращает `java.lang.Object`. Далее, инструкция 3 производит приведение типа («кастинг») объекта, полученного на предыдущем шаге к типу массива `java.lang.String`, и инструкция 6 выполняет метод `m:([Ljava/lang/String;)`, что и напечатает в консоли «two».

А теперь байт-код после компиляции на JDK1.7 – то есть на Java 7:

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=1, args_size=1
       0: invokestatic  #6                  // Method g:()Ljava/lang/Object;
       3: invokestatic  #7                  // Method m:(Ljava/lang/Object;)V
       6: return
    LineNumberTable:
      line 15: 0
      line 16: 6
```

Мы видим, что здесь нет инструкции `checkcast`, которую добавила Java 8, так что вызовется метод `m:(Ljava/lang/Object;)`, а в консоли напечатается «one». `Checkcast` – результат нового выведения типа, который был усовершенствован в  Java 8.

Чтобы избежать таких проблем, Oracle выпустил руководство по переходу с JDK1.7 на JDK 1.8 в котором описаны проблемы, которые могут возникнуть при переходе на новую версию Java, и то, как эти проблемы можно решить.

Например если вы хотите, чтобы в коде выше после компиляции на Java 8 все работало так же, как и на Java 7, сделайте приведение типа вручную:

```
public static void main(String[] args) {
   m((Object)g());
}
```

## Заключение

На этом мой рассказ о Java Generics подходит к концу. Вот другие источники, которые помогут вам в освоении темы:


- Naftalin, Maurice; Wadler, Philip. Java Generics and Collections. O'Reilly Media. ISBN-13: 978-0596527754
- https://docs.oracle.com/javase/specs/jls/se8/html/index.html
- Язык программирования Java SE 8. Addison-Wesley. ISBN: 978-5-8459-1875-8
- Bloch, Joshua. Effective Java. Third Edition. Addison-Wesley. ISBN-13: 978-0-13-468599-1


Пост является кратким пересказом одноименного доклада, на котором мы разбираем особенности работы с Java Generics.

**Теги:** Сбербанк, Java, generics