Оглавление

Предыстория дженериков

Дженерики в Java появились в 5 версии языка. До их появления задачи в Java решались довольно муторно.

Возьмем для примера задачу, в которой нам надо каждый раз в массиве находить минимальный элемент, но тип элементов нам заранее не известен.

Можно написать свой метод для каждого типа или, например, взять обобщенный тип Object. Во втором случае мы бы решили проблему копипаста, но в целом это решение не очень неудобно. Во-первых, надо помнить объекты какого типа мы храним, чтобы не положить туда чего-нибудь лишнего. Во-вторых, при извлечении элементов нужно явно приводить к нужному типу (ошибка проявится только во время исполнения программы).

Неудобства этого подхода побудили разработчиков Java добавить возможность параметризации классов, интерфейсов и методов каким-то типом. Т. е. вместо конкретного типа можно объявить и затем использовать некоторую переменную, в качестве которой будет подставлен любой тип, удовлетворяющий условиям.

Преимущества дженериков

· совершает проверку типов на уровне компиляции;

· не требует приведения;

· предоставляет возможность программистам реализовывать общие алгоритмы.

Стирание типов

Стирание типов — суть заключается в том, что внутри класса не хранится никакой информации о типе-параметре. Эта информация доступна только на этапе компиляции и стирается (становится недоступной) в runtime.

Ограничения дженериков

· параметризация возможна только для ссылочных типов;
· внутри параметризованного класса или метода нельзя создавать экземпляр или массив T, не работает проверка instanceof (не скомпилируется), а также приведение типа к T ничего не сделает (скомпилируется).

Это позволяет, например, реализовать трюк 19 по превращению проверяемого исключения в непроверяемое. Дело в том, что исключение делятся на проверяемые и непроверяемые только с токи зрения компилятора, а виртуальная машина их не различает. Для нее все исключения

являются непроверяемыми. То есть она прекрасно может бросить Exeption, проверяемый Exeption из метода, где он не задекларирован. Но, конечно, пользоваться этим хаком не следует. Это просто иллюстрация особенности дженериков.

```
 1. java.io.IOExeption;
 2.
 3. public class Hack {
 4.
 5.     public static void main(String[] args) {
 6.             throwAsUchecked(
 7.             new IOExeption());
 8.     }
 9.
10.     private static void throwAsUncheked(
11.             Exception e) {
12.         Hack.<RuntimeException>genericTrow(
13.                 e);
14.     }
15.
16.     private static <T extends Trowable>
17.             void genericThrow(Exception e)
18.             throws T {
19.         throw (T) e;
20.     }
21. }
22.
23.
24.
25.
26.
```

Синтаксис дженериков

После имени класса идут угловые скобки с именами дженерик-параметров. Если их несколько, то они будут перечислены через запятую. При желании мы можем объявить

ограничение на тип параметра, используя выражение `extends` и имя класса или интерфейса. Можно даже потребовать, чтобы параметр реализовывал несколько интерфейсов, перечислив их через &.

В теле класса этот дженерик параметр может использоваться практически в любом месте, где и обычно имя типа. Можно объявить поле или возвращаемое значение метода, или параметр метода, или локальную переменную. Дженерик-параметр, заданный на уровне класса, используется для параметризации экземпляров, поэтому не доступен в статических полях и методах.

Статический, да и не статический метод, можно параметризовать отдельно от класса, объявив дженерик-параметры в угловых скобках после модификаторов, но перед именем возвращаемого типа. В данном случае параметр тоже называется T, но это T совсем другое, не то, что указано в заголовке класса.

```
 1.  package java.util;
 2.
 3.  public final class Optional<T> {
 4.
 5.      private final T value;
 6.
 7.      private Optional(T value) {
 8.          this.value = Objects.requireNonNull(
 9.                  value);
10.      }
```

```
11.
12.     public static <T> Optional<T> of(
13.             T value) {
14.         return new Optional<>(value);
15.     }
16.
17.     public T get() {
18.         if (value == null) {
19.             throw new
20.                     NoSuchElementException(
21.                     "No value present");
22.         }
23.         return value;
24.     }
25.     // ...
26. }
```

Компилятор не создает специализированные версии классов, а создает единственную максимально общую версию класса и в байт-коде получается класс 3, где вместо T везде подставлен тип Object.

Т. е. компилятор сам на себя возлагает обязанность по проверке новых вкладываемых элеменота (на пригодность типа), а также неявное приведении элемент при извлечении.

Если не указать тип (пропустить угловые скобки), то класс будет работать как с Object.

diamond оператор

```
1. List<String> list = new ArrayList<>();
```

Пустые скобочки `<>` называются ***diamond оператором***. Здесь можно было бы явно написать `String`. А можно этого не делать,

и тогда компилятор сам подставит сюда параметры, взятые из типа переменной, куда мы присваиваем значение. diamond оператор работает только вместе с `new`.

Имена параметров

- E — элемент (обычно для коллекций);
- K — ключ;
- V — значение;
- N — номер;
- T — тип 1-го уровня;
- S, U, V и т. д. — типы 2-го, 3-го, 4-го уровней.

Принцип PECS

Если метод имеет аргументы с параметризованным типом (например, `Collection` или `Predicate`), то в случае, если аргумент — производитель (producer), нужно использовать `? extends T`, а если аргумент — потребитель (consumer), нужно использовать `? super T`.

Производитель и потребитель, кто это такие? Очень просто: если метод читает данные из аргумента, то этот аргумент — производитель, а если метод передает данные в аргумент, то аргумент является потребителем. Важно заметить, что определяя

производителя или потребителя, мы рассматриваем только данные типа T.

Разное

это технический термин, обозначающий набор свойств языка позволяющих определять и использовать обобщенные типы и методы. Обобщенные типы или методы отличаются от обычных тем, что имеют типизированные параметры.

Примером использования обобщенных типов может служить Java Collection Framework. Так, класс LinkedList<E> - типичный обобщенный тип. Он содержит параметр E, который представляет тип элементов, которые будут храниться в коллекции. Создание объектов обобщенных типов происходит посредством замены параметризированных типов реальными типами данных. Вместо того, чтобы просто использовать LinkedList, ничего не говоря о типе элемента в списке, предлагается использовать точное указание типа LinkedList<String>, LinkedList<Integer> и т.п.

Raw type - это имя интерфейса без указания параметризованного типа:

```
List list = new ArrayList(); // raw type
```

```
List<Integer> listIntgrs = new
ArrayList<>(); // parameterized type
```

Отдельно надо обсудить поведение Generic совместно с наследованием. Вспомним, что у нас есть базовый класс java.lang.Number и его наследник java.lang.Integer.

```
1.  Number number = new Integer(1);
2.  Number[] numberArray = new Integer[10];
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
```

Мы можем спокойно присвоить объект типа Integer в переменную типа Number.
Или массив Integer в массив Number. Однако с Generic ситуация другая.

```
Optional<Integer> optionalInt =
        Optional.of(1);
Optional<Number> optionalNumber =
        optionalInt;
```

Optional<Integer> нельзя присвоить переменную Optional<Number>. Это не скомпилируется. С точки зрения компилятора эти типы несовместимы. Это запрещено из следующих соображений.

```
Optional<Integer> optionalInt =
        Optional.of(1);
Optional<Number> optionalNumber =
        optionalInt;
optionalNumber.set(new BigDecimal("3.14"));
```

```
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
```

Давайте представим, что в Optional есть метод set(), заменяющий объект внутри контейнера. Тогда, присвоив Optional<Integer> в Optional<Number> мы могли бы следующим шагом заменить объект внутри Optional<Integer> на какой-нибудь BigDecimal и тем самым нарушить ограничение, заданное при создании контейнера. Кстати, в случае с массивами, виртуальная машина защищает нас от подобной ошибки.

```
1.  numberArray[0] = new BigDecimal(...)
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
```

```
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
```

Если мы попытаемся положить в массив, созданный как массив Integer, значение другого типа, несовместимого, например, BigDecimal, то виртуальная машина бросит исключение ArrayStoreException.

Такая несовместимость Generic-типов, конечно, не очень удобно. Рассмотрим эти неудобства более подробно на примере методов ifPresent() и orElseGet() из класса Optional. Я сейчас привел объявление этих методов в наивном простом варианте, который не совсем соответствует тому, что написано в стандартной библиотеке, но сейчас мне это нужно для демонстрации проблемы.

```
1.  package java.util;
2.  public final class Optional<T> {
3.      // ...
4.      public void ifPresent(
5.              Consumer<T> comsumer) {
6.          if (value != null)
7.              consumer.accept(value);
```

```
    }
    // ...
}

Optional<CharSequence> opt;
Consumer<Object> cons;
opt.ifPresent(cons);
```

Итак, давайте представим, что у нас есть экземпляр Optional, параметризованный интерфейсом CharSequence. Метод ifPresent принимает экземпляр Consumer. Consumer — это функциональный интерфейс с единственным методом accept(), принимающим один параметр указанного типа. Предположим, у нас откуда-то есть Consumer, параметризованный Object, то есть его метод accept() с радостью примет абсолютно любой объект, в том числе и CharSequence. Однако система типов Java запретит нам вызвать метод Optional ifPresent() с этим Consumer из-за несовместимости типов.

```
package java.util;
public final class Optional<T> {
```

```
    // ...
    public T orElseGet(Supplier<T> other) {
        return value != null ? value :
                other.get();
    }
    // ...
}

Optional<CharSequence> opt;
Supplier<String> sup;
opt.orElseGet(sup);
```

Аналогичная проблема с методом orElseGet(). Этот метод принимает Supplier. Supplier — это функциональный интерфейс с единственным методом get(), возвращающим значение, указанного типа. Предположим, у нас откуда-то есть Supplier от String. Однако, Java запретит нам вызвать метод Optional orElseGet() от этого Supplier из-за того же несоответствия типов String и CharSequence. Хотя по смыслу такая операция абсолютно корректна и должна быть разрешена.

```
package java.util;

```

```
3.  public final class Optional<T> {
4.
5.      private final T value;
6.
7.      public void ifPresent(
8.              Consumer<? super T> comsumer) {
9.          if (value != null)
10.                 consumer.accept(value);
11.     }
12.
13.     public T orElseGet(Supplier<? extends T>
14.             other) {
15.         return value != null ? value :
16.                 other.get();
17.     }
18.
19.     // ...
20. }
21.
22.
23.
24.
25.
26.
```

К счастью, в Java все-таки есть способ договориться с компилятором. Можно использовать Generic-типы не просто с T, а с некоторым выражением маской относительно T. Выглядит это <? super T> и <? extends T>. В методе ifPresent мы говорим, что принимаем Consumer, принимающего объект любого супертипа T. А в методе orElseGet() мы говорим, что принимаем Supplier любого подтипа T. При этом сам тип T тоже считается и своим супертипом и своим подтипом. Эвристическое правило здесь такое: если вы собираетесь

получать объекты откуда-то, то используйте <? extends T>, а если отдавать куда-то, то <? super T>.

```
Optional<?> optional = Optional.of(1);
```

В тех случаях, когда нам абсолютно все равно чем параметризован Generic, можно просто написать ?. Это означает, что принимается любое значение Generic-параметра.

```
Object value1 = optional.get();
```

```
 6.
 7.
 8.
 9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
```

При этом, когда мы попробуем получить из такого Optional значение типа T, то оно будет возвращаться как Object.

```
 1. Object value2 = optional.orElse(2);
 2.
 3.
 4.
 5.
 6.
 7.
 8.
 9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
```

```
19.
20.
21.
22.
23.
24.
25.
26.
```

А методы, принимающий параметры типа T, вызвать будет невозможно. Когда компилятор не видит конкретное значение дженерик параметра T, то он не может проконтролировать совместимость типов и откажется это компилировать.

Optional

Зачем нужен `Optional`? Любая переменная и так умеет хранить ссылку на один объект или значение `null`.

Две причины:

· В программе по типу переменной мы не можем определить допускает ли логика программы значение `null`, т. е. надо ли делать проверку на `null`.
Эта неопределенность источник большого количества `NPE`. `Optional` решает данную проблему на уровне типов.

· Позволяет писать код без `if`.

```
1.  Optional<String> baz = Optional.of("baz");
2.  baz.ifPresent(System.out::println);
3.
4.  if (s != null) {
5.      System.out.println(s);
```

```
6. }
```

Реализация и методы

`empty()` возвращает пустой `Optional`, не содержащий ссылку на объект.

```
1. Optional<String> foo = Optional.empty();
```

`of()` возвращает `Optional`, содержащий указанный объект. При этом аргумент не может быть `null`, иначе исключение.

```
1. Optional<String> bar = Optional.of("bar");
```

`ofNullable()` возвращает `Optional`, содержащий указанный объект, если там не null. А если `null`, то возвращается пустой `Optional`.

```
1. Optional<String> baz = Optional.ofNullable("baz");
2.
```

Компилятор видит значение какого типа мы передаем в параметризованный фабричный метод и отдает нам экземпляр Optional, параметризованный этим типом. В данных случаях String. В случае метода empty(), который не принимает параметров, компилятор просто подгоняет возвращаемое значение к нужному типу.

```
1. Optional<CharSequence> optionalCharSequence =
2.         Optional.<CharSequence>ofNullable(
               "baz");
```

Если мы хотим получить из строчки, то есть String, экземпляр Optional,

параметризованный интерфейсом CharSequence, то надо явно указать на это компилятору. Иначе эта строчка бы не скомпилировалась, так как типы Optional от CharSequence и Optional от String не совместимы между собой. Подробнее об этом мы еще поговорим.

```
1. Optional<String> newOptional =
2.         new Optional<>("foobar");
```

Если бы конструктор Optional был публичным, то экземпляр можно было бы создать так. Эти пустые скобочки <> называются diamond-оператор. Здесь можно было бы явно написать String. А можно этого не делать, и тогда компилятор сам в уме подставит сюда параметры, взятые из типа переменной, куда мы присваиваем значение. diamond-оператор работает только вместе с new.

```
1.  package java.util;
2.
3.  public final class Optional<T> {
4.
5.      private final T value;
6.
7.      private Optional(T value) {
8.          this.value = Objects.requireNonNull(
9.                  value);
10.     }
11.
12.     public static <T> Optional<T> of(
13.             T value) {
14.         return new Optional<>(value);
15.     }
```

```
16.
17.     public T get() {
18.         if (value == null) {
19.             throw new
20.                     NoSuchElementException(
21.                     "No value present");
22.         }
23.         return value;
24.     }
25.     // ...
26. }
```