



Обобщенное программирование

Croc Java School




Логика обработки данных некоторых классов не требует дополнительной информации о типе этих данных.

Такие классы как правило реализуют хранение/упорядочивание/поиск и их можно воспринимать как контейнеры.



Pair


```
public class Pair {  
    private final Object first;  
    private final Object second;  
  
    public Pair(Object first, Object second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public Object getFirst() {  
        return first;  
    }  
  
    public Object getSecond() {  
        return second;  
    }  
}
```



Логика класса не накладывает ограничения на типы данных его полей.

```
Pair adaBirthday = new Pair(  
    "Ada Lovelace",  
    LocalDate.of(1815, 12, 10));
```

```
Pair usdBuysell = new Pair(  
    new BigDecimal("70.75"),  
    new BigDecimal("71.70"));
```



Но в текущем виде использовать класс не очень удобно из-за необходимости приведения типов полей класса к более специфицированным типам данных

```
Pair adaBirthday = new Pair(  
    "Ada Lovelace",  
    LocalDate.of(1815, 12, 10));
```

```
DayOfWeek dayOfWeek = ((LocalDate)adaBirthday.getSecond())  
    .getDayOfWeek();
```

Вдобавок можем получить RuntimeException, если ошиблись с типом. Нет проверки на этапе компиляции.



Удобным было бы поведение, при котором

- методы возвращали значения конкретных типов, указанных в момент создания объекта
- не возникало runtime-ошибок приведения типов

```
Pair adaBirthday = new Pair(  
    "Ada Lovelace",  
    LocalDate.of(1815, 12, 10));
```

```
adaBirthday.getFirst(); // хотим String  
adaBirthday.getSecond(); // хотим LocalDate
```

—

Дженерики



Параметризованные типы и методы

В Java 5 появились дженерики (generic classes, generic methods)

- Расширение системы типов
- Позволяют классам оперировать данными разных типов без нарушения безопасности типов на этапе компиляции
- Реализованы с помощью “стирания типов” — обобщения на этапе компиляции заменяются конкретными типами. Поддержка на уровне языка, а не платформы



Параметризованные типы

Определение параметризованного класса:

```
class Name<T1, T2, ..., Tn> {  
    // ...  
}
```

Инстанцирование параметризованного класса:

```
Name<String, Integer, ..., Object> instance = new Name<>();
```

В определении класса используем параметры без указания конкретных типов, при создании объектов задаем специфицируем типы в <...>.



Вывод типов (тайп инференс)

Diamond operator <> позволяет не дублировать спецификацию параметров класса в определении переменной и вызове конструктора.

Вместо

```
Name<String, Integer, ..., Object> instance = new Name<String, Integer, ..., Object>();
```

Пишем

```
Name<String, Integer, ..., Object> instance = new Name<>();
```



Значения параметров

Параметры типа могут быть специфицированы любым ссылочным типом. Примитивы с дженериками использовать нельзя

```
Generic<Integer> ofReference;
```

```
Generic<int[]> ofArray;
```

```
Generic<Closeable> ofInterface;
```

```
Generic<Generic<Object>> ofOtherGeneric;
```

```
Generic<int> ofPrimitive; // error: Type argument cannot be of primitive type
```



Generic Pair


```
public class Pair<T1, T2> {  
    private final T1 first;  
    private final T2 second;  
  
    public Pair(T1 first, T2 second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public T1 getFirst() {  
        return first;  
    }  
  
    public T2 getSecond() {  
        return second;  
    }  
}
```



Явных приведений типов больше не требуется

```
Pair<String, LocalDate> adaBirthday = new Pair<>(  
    "Ada Lovelace",  
    LocalDate.of(1815, 12, 10));
```

```
DayOfWeek dayOfWeek = adaBirthday.getSecond()  
    .getDayOfWeek();
```



```
Pair<BigDecimal, BigDecimal> usdBuySell = new Pair<>(  
    new BigDecimal("70.75"),  
    new BigDecimal("71.70"));
```

```
BigDecimal diff = usdBuySell.getSecond()  
    .subtract(usdBuySell.getFirst());
```



Соглашения по названиям параметров


E	Элемент коллекции
K	Ключ (уникальный идентификатор)
N	Численный тип
T	Произвольный тип
V	Значение



Параметризованные методы

Параметры можно задавать не только для классов, но и для отдельных методов

```
<T1, T2, ..., Tn> void method() {  
    // ...  
}
```

```
public static <T1, T2> Pair<T1, T2> makePair(T1 first, T2 second) {  
    return new Pair<>(first, second);  
}
```

Использование (типы специфицируем перед названием метода)

```
Pair<String, String> pair = Pair.<String, String>makePair("a", "b");
```

Вывод типов работает и с методами тоже

```
Pair<String, String> pair = Pair.makePair("a", "b");
```

Ограничения (bounds)



Ограничение типов параметров

Область значений параметров можно ограничить одним или несколькими базовыми типами. T должен быть наследником Number

```
<T extends Number>
```

T должен реализовывать оба интерфейса: Iterator и Closeable

```
<T extends Iterator & Closeable>
```

Если в списке ограничений один из типов класс, то он всегда должен стоять первым в перечислении



Использование ограничений

```
public static <T extends Comparable<T>> T max(T... values) {  
    T max = null;  
    for (T value : values) {  
        if (max == null || max.compareTo(value) < 0)  
            max = value;  
    }  
    return max;  
}
```

—

<?>



В качестве значения параметра может использоваться wildcard-тип, определяемый знаком вопроса и означающий “любой тип”.

```
public static void print(Pair<?, ?> pair) {  
    System.out.println(pair.first.toString()  
        + "-" + pair.second.toString());  
}
```

Если в пример `Pair<?, ?>` заменить на `Pair<Object, Object>`, то метод нельзя будет применить к, например, `Pair<String, Integer>`, так как `Pair<String, Integer>` не наследник `Pair<Object, Object>`



Ограничения для <?>

К wildcard-типам могут быть применены ограничения не только “сверху”, но и “снизу”

`<? extends Number>`

? — наследник Number

`<? super Integer>`

? — родитель Integer

Raw-типы



Конструктор без <>

Параметризованный тип можно использовать без указания параметров.

Параметризованный класс без спецификации параметров называется **raw-классом**

```
Pair rawPair = new Pair("Zzzz..", 17);
```

Raw- и параметризованные типы можно неявно приводить между собой, но безопасность типов при этом теряется, поэтому raw-типы стоит стараться не использовать

```
Pair<String, Integer> pair = new Pair<>("Zzzz..", 17);  
rawPair = pair; // ok  
pair = rawPair; // warning: unchecked assignment
```



Heap pollution

Ситуации, когда “задекларированное” значение типа параметра не соответствует его значению в рантайме, называется heap pollution.

```
Pair pair = new Pair<Integer, Integer>(1, 2); // ok
Pair<String, String> polluted = pair; // warning, but ok
String first = polluted.getFirst(); // ClassCastException
```

Дженерики в сочетании с varargs потенциальный источник подобных ситуаций



Иерархия типов

Number — базовый класс для Integer

```
Integer value = 1;  
Number base = value;
```

Optional<Integer> не наследник Optional<Number>

```
Optional<Integer> genericValue = Optional.of(1);  
Optional<Number> genericBase = genericValue; // error
```

Но Optional<? extends Integer> наследник Optional<? extends Number>

```
Optional<? extends Integer> genericValue = Optional.of(1);  
Optional<? extends Number> genericBase = genericValue; // ok
```

Ограничения (limitations)



Что нельзя делать с параметрами типов и методов

- Использовать в качестве типов статических полей классов
- Создавать экземпляры (но есть `Class<T>.newInstance()`)
- Создавать массивы
- Применять оператор преобразования типов ()
- Применять оператор `instanceof`



Пара слов про “стирание” типов

- Типы параметров классов и методов на этапе компиляции заменяются конкретными типами: Object или типам ограничения. На уровне байткода никаких дженериков уже нет.
- Компилятор добавляет преобразования типов, когда базовый тип должен быть “сужен” до более специфицированного наследника.
- В некоторых случаях он может добавлять bridge-методы, гарантирующие корректную реализацию принципов ООП (сохраняющие полиморфизм, который может пострадать из-за удаления информации о типах)