Введение в программирование на Java

Лекция 13. Дженерики (часть 1). Базовые коллекции.

Виталий Олегович Афанасьев

28 апреля 2025

Проблема

Рассмотрим класс `IntStack` — стек чисел типа `int`.

```
1 public class IntStack {
2     private int[] values;
3     private int size;
4
5     public void push(int value) { ... }
6     public int pop() { ... }
7     public int size() { ... }
8 }
```

Что делать, если мы хотим реализовать стек из строк? Или же стек вещественных чисел?

Замена конкретного типа на Object (1)

Мы могли бы заменить конкретный тип на `Object`.

Т.к. `Object` является родительским для любого класса, то это должно сработать?

```
1 public class MyStack {
2     private Object[] values;
3     private int size;
4
5     public void push(Object value) { ... }
6     public Object pop() { ... }
7     public int size() { ... }
8 }
```

Замена конкретного типа на Object (2)

Но возникает проблема с тем, что компилятор перестаёт проверять корректность операций.

```
1  MyStack stack = new MyStack();
2
3
4
5
6
7
8
9
10
```

Замена конкретного типа на Object (2)

Но возникает проблема с тем, что компилятор перестаёт проверять корректность операций.

```
1  MyStack stack = new MyStack();
2  stack.push("Hello");
3
4
5
6
7
8
9
10
```

Замена конкретного типа на Object (2)

Но возникает проблема с тем, что компилятор перестаёт проверять корректность операций.

```
1  MyStack stack = new MyStack();
2  stack.push("Hello");
3  stack.push("World");
4
5
6
7
8
9
10
```

Замена конкретного типа на Object (2)

Но возникает проблема с тем, что компилятор перестаёт проверять корректность операций.

```
1  MyStack stack = new MyStack();
2  stack.push("Hello");
3  stack.push("World");
4
5  String result = stack.pop(); // Ошибка компиляции: pop возвращает Object
6
7
8
9
10
```

Замена конкретного типа на Object (2)

Но возникает проблема с тем, что компилятор перестаёт проверять корректность операций.

```
1  MyStack stack = new MyStack();
2  stack.push("Hello");
3  stack.push("World");
4
5  String result = stack.pop(); // Ошибка компиляции: pop возвращает Object
6  String result = (String) stack.pop(); // OK
7
8
9
10
```

Замена конкретного типа на Object (2)

Но возникает проблема с тем, что компилятор перестаёт проверять корректность операций.

```
1  MyStack stack = new MyStack();
2  stack.push("Hello");
3  stack.push("World");
4
5  String result = stack.pop(); // Ошибка компиляции: pop возвращает Object
6  String result = (String) stack.pop(); // OK
7
8  stack.push(42);
9
10
```

Замена конкретного типа на Object (2)

Но возникает проблема с тем, что компилятор перестаёт проверять корректность операций.

```
1  MyStack stack = new MyStack();
2  stack.push("Hello");
3  stack.push("World");
4
5  String result = stack.pop(); // Ошибка компиляции: pop возвращает Object
6  String result = (String) stack.pop(); // OK
7
8  stack.push(42);
9  stack.push(1.5);
10
```

Замена конкретного типа на Object (2)

Но возникает проблема с тем, что компилятор перестаёт проверять корректность операций.

```
1  MyStack stack = new MyStack();
2  stack.push("Hello");
3  stack.push("World");
4
5  String result = stack.pop(); // Ошибка компиляции: pop возвращает Object
6  String result = (String) stack.pop(); // OK
7
8  stack.push(42);
9  stack.push(1.5);
10 stack.push(true);
```

Обобщённые классы (1)

Java позволяет работать с обобщёнными типами.

```
1 public class MyStack<T> {
2     private T[] values;
3     private int size;
4
5     public void push(T value) { ... }
6     public T pop() { ... }
7     public int size() { ... }
8 }
```

Тип `T` здесь выступает некоторой заглушкой — про него ничего неизвестно, но методы `push` и `pop` гарантированно работают с одним и тем же типом.

`T` называется **типовым параметром**.

Обобщённые классы (2)

При использовании обобщённых классов типовым параметрам должны быть присвоены конкретные типы (они называются **типовыми аргументами**).

```
1 MyStack<String> stack = new MyStack<String>();
2
3
4
5
6
7
8
9
```

Обобщённые классы (2)

При использовании обобщённых классов типовым параметрам должны быть присвоены конкретные типы (они называются **типовыми аргументами**).

```
1 MyStack<String> stack = new MyStack<String>();
2 stack.push("Hello");
3
4
5
6
7
8
9
```

Обобщённые классы (2)

При использовании обобщённых классов типовым параметрам должны быть присвоены конкретные типы (они называются **типовыми аргументами**).

```
1 MyStack<String> stack = new MyStack<String>();
2 stack.push("Hello");
3 stack.push("World");
4
5
6
7
8
9
```

Обобщённые классы (2)

При использовании обобщённых классов типовым параметрам должны быть присвоены конкретные типы (они называются **типовыми аргументами**).

```
1 MyStack<String> stack = new MyStack<String>();
2 stack.push("Hello");
3 stack.push("World");
4
5 String result = stack.pop(); // OK
6
7
8
9
```

Обобщённые классы (2)

При использовании обобщённых классов типовым параметрам должны быть присвоены конкретные типы (они называются **типовыми аргументами**).

```
1 MyStack<String> stack = new MyStack<String>();
2 stack.push("Hello");
3 stack.push("World");
4
5 String result = stack.pop(); // OK
6
7 stack.push(42); // Ошибка компиляции: ожидался String
8
9
```

Обобщённые классы (2)

При использовании обобщённых классов типовым параметрам должны быть присвоены конкретные типы (они называются **типовыми аргументами**).

```
1 MyStack<String> stack = new MyStack<String>();
2 stack.push("Hello");
3 stack.push("World");
4
5 String result = stack.pop(); // OK
6
7 stack.push(42); // Ошибка компиляции: ожидался String
8 stack.push(1.5); // Ошибка компиляции: ожидался String
9
```

Обобщённые классы (2)

При использовании обобщённых классов типовым параметрам должны быть присвоены конкретные типы (они называются **типовыми аргументами**).

```
1 MyStack<String> stack = new MyStack<String>();
2 stack.push("Hello");
3 stack.push("World");
4
5 String result = stack.pop(); // OK
6
7 stack.push(42); // Ошибка компиляции: ожидался String
8 stack.push(1.5); // Ошибка компиляции: ожидался String
9 stack.push(true); // Ошибка компиляции: ожидался String
```

Обобщённые классы (3)

Зачастую типовые аргументы при вызове конструкторов можно опускать — компилятор их выведет автоматически.

При этом используется т.н. "diamond operator" — <>.

```
1 MyStack<String> stack = new MyStack<String>();
2 MyStack<String> stack = new MyStack<>();
```

Несколько generic-параметров

```java
public class Pair<T, U> {
    private final T first;
    private final U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
        return second;
    }
}
```

```java
Pair<String, Integer> pair = new Pair<>("Something", 42);
```

Generic-методы (1)

Обобщённым можно делать не весь класс, но и один конкретный метод.

```
1  public class Example {
2      public static <T> boolean contains(T[] arr, T elem) {
3          for (int i = 0; i < arr.length; ++i) {
4              if (arr[i].equals(elem)) {
5                  return true;
6              }
7          }
8          return false;
9      }
10 }
```

Generic-методы (2)

```
1 public class Example {
2     public static <T> boolean contains(T[] arr, T elem) { ... }
3
4     public static boolean example(String[] values) {
5         return Example.<String>contains(values, "Something");
6     }
7 }
```

Generic-методы (3)

Как и в случае с конструкторами, компилятор в большинстве случаев автоматически выводит типовые аргументы методов.

```
1 public class Example {
2     public static <T> boolean contains(T[] arr, T elem) { ... }
3
4     public static boolean example(String[] values) {
5         // Эквивалентно Example.<String>contains
6         return Example.contains(values, "Something");
7     }
8 }
```

Типовые параметры в static-методах (1)

Типовые параметры классов получают конкретные значения только при создании экземпляров. Поэтому типовые параметры имеют смысл только для не-static-членов класса.

static-члены не могут использовать типовые параметры класса.

```
1 public class Pair<T, U> {
2     public Pair(T first, U second) { ... }
3
4     // Ошибка компиляции: T и U неизвестны
5     public static Pair<T, U> of(T first, U second) {
6         return new Pair<>(first, second);
7     }
8 }
```

Типовые параметры в static-методах (2)

Решение: указывать у static-методов свои собственные типовые параметры.

```
1 public class Pair<T, U> {
2     public Pair(T first, U second) { ... }
3
4     public static <T1, T2> Pair<T1, T2> of(T1 first, T2 second) {
5         return new Pair<T1, T2>(first, second);
6     }
7 }
```

Примитивные типы как типовые аргументы

В качестве типовых аргументов можно использовать только ссылочные типы.

```
1 MyStack<String> stringStack = new MyStack<>(); // OK
2 MyStack<Object> objectStack = new MyStack<>(); // OK
3
4 MyStack<int> intStack = new MyStack<>(); // Ошибка компиляции
```

Wrapper-типы

Для каждого примитивного типа существует ссылочный тип-обёртка (wrapper):

- `int` — `Integer`
- `byte` — `Byte`
- `short` — `Short`
- `long` — `Long`
- `double` — `Double`
- `float` — `Float`
- `char` — `Character`
- `boolean` — `Boolean`
- `void` — `Void`

```
1 MyStack<Integer> intStack = new MyStack<>();
2 intStack.push(42);
3 int top = intStack.pop();
```

Autoboxing и unboxing

Между примитивным и wrapper-типом происходят автоматические приведения типов:

- Autoboxing — приведение из примитивного в wrapper-тип.
- Unboxing — приведение из wrapper-типа в примитивный.

```
1 int first = 42;
2 Integer second = first; // OK: Autoboxing
3 int third = second; // OK: Unboxing
```

Важно: autoboxing и unboxing — не бесплатные операции. По возможности их нужно избегать.

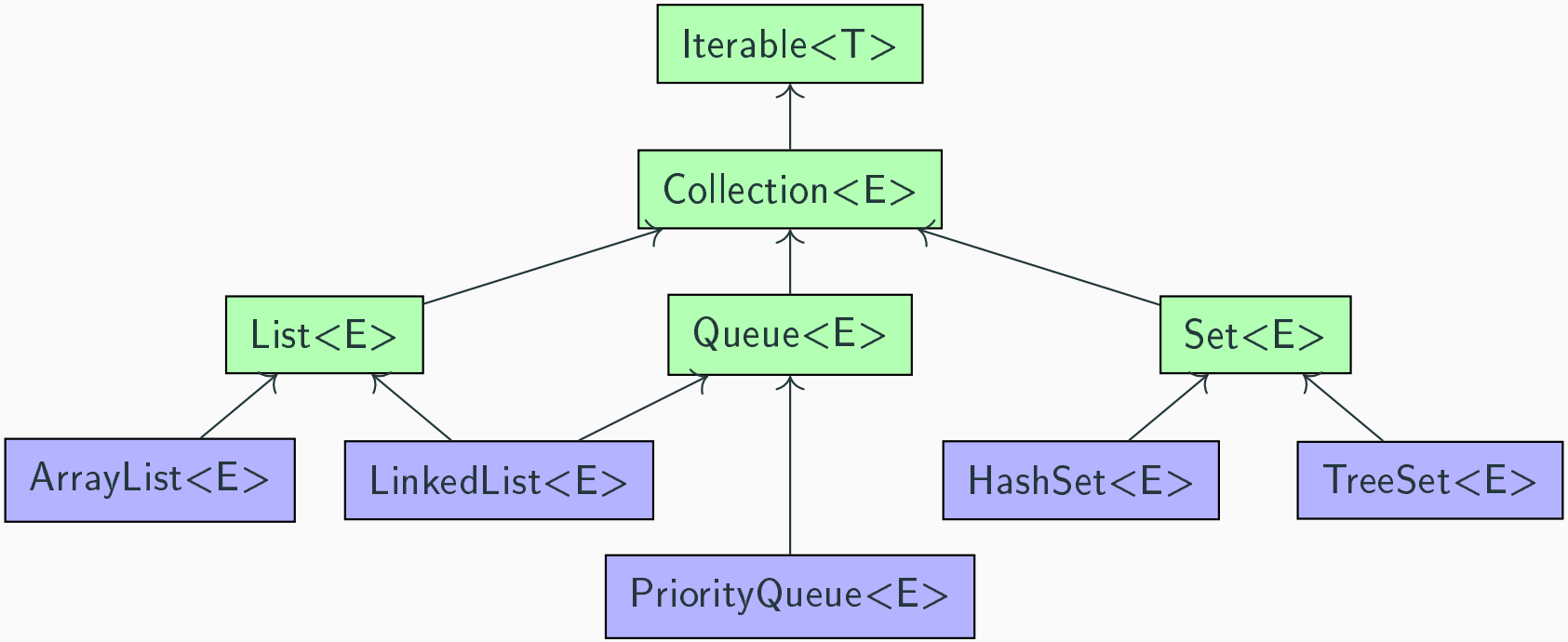
NPE при unboxing

Wrapper-типы допускают значение `null`, а примитивные — нет.

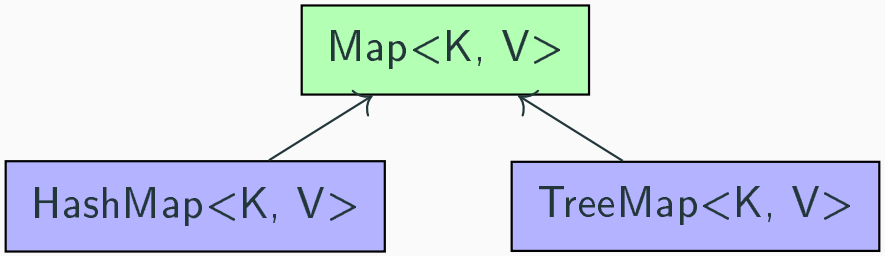
Если переменная wrapper-типа имеет значение `null`, то при unboxing возникнет `NullPointerException`.

```
1 Integer first = null;
2 int second = first; // Компилируется, но возникает NPE при выполнении
```

Collection API (1)



Collection API (2)



Операции со списками

```
1 List<String> list = new ArrayList<>();
2
3
4
5
6
7
8
9
```

Операции со списками

```
1 List<String> list = new ArrayList<>();
2 list.add("first")
3
4
5
6
7
8
9
```

Операции со списками

```
1 List<String> list = new ArrayList<>();
2 list.add("first")
3 list.add("second");
4
5
6
7
8
9
```

Операции со списками

```
1 List<String> list = new ArrayList<>();
2 list.add("first")
3 list.add("second");
4 String at0 = list.get(0); // "first"
5
6
7
8
9
```

Операции со списками

```
1 List<String> list = new ArrayList<>();
2 list.add("first")
3 list.add("second");
4 String at0 = list.get(0); // "first"
5 int size = list.size(); // 2
6
7
8
9
```

Операции со списками

```
1 List<String> list = new ArrayList<>();
2 list.add("first")
3 list.add("second");
4 String at0 = list.get(0); // "first"
5 int size = list.size(); // 2
6 boolean isEmpty = list.isEmpty(); // false
7
8
9
```

Операции со списками

```
1 List<String> list = new ArrayList<>();
2 list.add("first")
3 list.add("second");
4 String at0 = list.get(0); // "first"
5 int size = list.size(); // 2
6 boolean isEmpty = list.isEmpty(); // false
7 boolean containsSecond = list.contains("second"); // true
8
9
```

Операции со списками

```
1 List<String> list = new ArrayList<>();
2 list.add("first")
3 list.add("second");
4 String at0 = list.get(0); // "first"
5 int size = list.size(); // 2
6 boolean isEmpty = list.isEmpty(); // false
7 boolean containsSecond = list.contains("second"); // true
8 list.remove(0);
9
```

Операции со списками

```java
List<String> list = new ArrayList<>();
list.add("first")
list.add("second");
String at0 = list.get(0); // "first"
int size = list.size(); // 2
boolean isEmpty = list.isEmpty(); // false
boolean containsSecond = list.contains("second"); // true
list.remove(0);
at0 = list.get(0); // "second"
```

Цикл for-each

Если тип реализует интерфейс `Iterable`, то для него становится доступен цикл for-each.

```
1 List<String> list = new ArrayList<>();
2 list.add("first");
3 list.add("second");
4 list.add("third");
5 for (String item : list) {
6     System.out.println(item); // first second third
7 }
```

Кроме того, данный цикл можно использовать и для массивов.

Операции с Map

Тип `Map` (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2
3
4
5
6
7
8
9
10
```

Операции с Map

Тип `Map` (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2  weekdayToNumber.put("Monday", 1);
3
4
5
6
7
8
9
10
```

Операции с Map

Тип `Map` (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2  weekdayToNumber.put("Monday", 1);
3  weekdayToNumber.put("Tuesday", 2);
4
5
6
7
8
9
10
```

Операции с Map

Тип `Map` (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2  weekdayToNumber.put("Monday", 1);
3  weekdayToNumber.put("Tuesday", 2);
4  ...
5
6
7
8
9
10
```

Операции с Map

Тип `Map` (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2  weekdayToNumber.put("Monday", 1);
3  weekdayToNumber.put("Tuesday", 2);
4  ...
5
6  int tuesday = weekdayToNumber.get("Tuesday"); // 2
7
8
9
10
```

Операции с Map

Тип `Map` (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2  weekdayToNumber.put("Monday", 1);
3  weekdayToNumber.put("Tuesday", 2);
4  ...
5
6  int tuesday = weekdayToNumber.get("Tuesday"); // 2
7  int size = weekdayToNumber.size(); // 7
8
9
10
```

Операции с Map

Тип Map (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2  weekdayToNumber.put("Monday", 1);
3  weekdayToNumber.put("Tuesday", 2);
4  ...
5
6  int tuesday = weekdayToNumber.get("Tuesday"); // 2
7  int size = weekdayToNumber.size(); // 7
8  boolean isEmpty = weekdayToNumber.isEmpty(); // false
9
10
```

Операции с Map

Тип `Map` (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2  weekdayToNumber.put("Monday", 1);
3  weekdayToNumber.put("Tuesday", 2);
4  ...
5
6  int tuesday = weekdayToNumber.get("Tuesday"); // 2
7  int size = weekdayToNumber.size(); // 7
8  boolean isEmpty = weekdayToNumber.isEmpty(); // false
9  boolean containsJanuary = weekdayToNumber.contains("January"); // false
10
```

Операции с Map

Тип `Map` (ассоциативный массив) хранит отображение объектов-ключей в объекты-значения.

Одному ключу соответствует максимум одно значение.

```
1  Map<String, Integer> weekdayToNumber = new HashMap<>();
2  weekdayToNumber.put("Monday", 1);
3  weekdayToNumber.put("Tuesday", 2);
4  ...
5
6  int tuesday = weekdayToNumber.get("Tuesday"); // 2
7  int size = weekdayToNumber.size(); // 7
8  boolean isEmpty = weekdayToNumber.isEmpty(); // false
9  boolean containsJanuary = weekdayToNumber.contains("January"); // false
10 Integer january = weekdayToNumber.get("January"); // null
```

Использование собственных типов как ключей в HashMap (1)

Чтобы заставить `HashMap` работать для своих типов ключей, необходимо переопределить методы `equals` и `hashCode` для этих типов.

Законы:

- Если `X.equals(Y)` возвращает `true`, то `X.hashCode()` должен быть равен `Y.hashCode()`.
- Если же объекты не равны, то их хэш-код может совпадать, а может и не совпадать.
- Если объект не менялся, то `hashCode` должен возвращать один и тот же результат при нескольких вызовах.

Использование собственных типов как ключей в HashMap (2)

Стандартная практика: использовать в equals и hashCode одни и те же поля.

```
1  public class User {
2      private final String firstName;
3      private final String lastName;
4      @Override
5      public boolean equals(Object o) {
6          if (this == o) return true;
7          if (o == null) return false;
8          if (getClass() != o.getClass()) return false;
9          User other = (User) o;
10         return firstName.equals(other.firstName)
11                 && lastName.equals(other.lastName);
12     }
13     @Override
14     public int hashCode() {
15         return Objects.hash(firstName, lastName);
16     }
17 }
```