СБЕРБАНК ТЕХНОЛОГИИ

Generics

12.02.18

Цели

- Ознакомиться с Java Generics
- Изучить правила использования
- Использование масок (Wildcards)
- Применить на практике свои знания о Java Generics

Обобщённое программирование — это такой подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания.

Generics введены с версии Java 1.5

generics (дженерики) или <<контейнеры типа T>> — подмножество обобщенного программирования.

До generics

```
List strList = new ArrayList();
strList.add("some text");

//ОК, хотя коллекция предназначалась для хранения строк!
strList.add(new Integer(0));
String str = (String)strList.get(0);

//Ошибка приведения типов во время выполнения
(ClassCastException) Integer i = (Integer)strList.get(0);
```

С использованием generics

```
List<String> strList = new ArrayList<String>();
strList.add("some text");
strList.add(new Integer()); // сообщение об ошибке компилятора
String str = strList.get(0);
Integer i = strList.get(0); // сообщение об ошибке компилятора
```

В чём профит:

- Проверка типов – на этапе компиляции
- Отсутствие приведения типов

«Алмазный» синтаксис Diamonds

```
Pair<Integer, String> pair = new Pair<Integer, String>(6, "Apr");
```

В Java 1.7 введен “diamond” operator <>, с помощью которого можно опустить параметры типа:

```
Pair<Integer, String> pair = new Pair<>(6, "Apr");
```

Параметризация класса

```
public class Test <T>  {

    private T param;

    public Test(T param) {
        this.param = param;
    }
    public T getParam() {
        return this.param;
    }
}
```

Параметризация класса

```
class PairOfT<T> {
    T object1;
    T object2;

    PairOfT(T one, T two) {
        object1 = one;
        object2 = two;
    }

    public T getFirst() {
        return object1;
    }

    public T getSecond() {
        return object2;
    }
}
```

Параметризация класса

```
class Pair <T1, T2> {
    T1 object1;
    T2 object2;

    Pair(T1 one, T2 two) {
        object1 = one;
        object2 = two;
    }

    public T1 getFirst() {
        return object1;
    }

    public T2 getSecond() {
        return object2;
    }
}
```

Ограничение переменной типа

```
public class Box <T extends Comparable & Serializable>{
    private T content;

    Box(R content) {this.content = content; }
    T getContent() {return first; }
    void setContent(T first) {this.first = first;}
}
```

После type erasure:

```
public class Box {
    private Comparable content;

    Box(Comparable content) {this.content = content; }
    Comparable getContent() {return first; }
    void setContent(Comparable first) {this.first = first;}
}
```

Параметризация методов **(Generic methods)**

Универсальными могут быть кроме классов также и методы

```
public static <T> void fill(List<T> list, T val) {…}

public <T> T getData(T data) { return data; }
```

Принятые наименования:

- **E** — элемент (Element, обширно используется Java Collections Framework)
- **K** — Ключ, **V** — Значение
- **N** — Число
- **T** — Тип
- **S, U** и т. п. — 2-й, 3-й типы

Возможное именование – как и названия классов.

Может пример уже посмотрим?

Выведение типов — это возможность компилятора Java автоматически определять аргументы типа на основе контекста, чтобы вызов получился возможным.

Алгоритм выведения типов определяет типы аргументов и, если есть, тип, в который присваивается результат или в котором возвращается результат. Далее алгоритм пытается найти наиболее конкретный тип, который работает со всеми аргументами.

Целевой тип выражения — это тип данных, который компилятор Java ожидает в зависимости от того, в каком месте находится выражение.

```
List<String> list = Collections.emptyList();
```

Вопрос по использованию generics

какая (-ие) из нижеприведённых строк откомпилируется без проблем?

1. List<Integer> list = **new** List<Integer>();
2. List<Integer> list = **new** ArrayList<Integer>();
3. List<Number> list = **new** ArrayList<Integer>();
4. List<Integer> list = **new** ArrayList<Number>();

Wildcards (Маски)

```
//Ошибка компиляции (нарушение типобезопасности)
List<Number> intList = new ArrayList<Integer>();
```

Wildcards (Маски)

```
//Этот код скомпилируется
List<?> intList = new ArrayList<Integer>();
```

Однако при попытке

```
intList.add(new Integer(10));
```

Получим ошибку компиляции

При использовании маски <?> мы не знаем, какого типа аргумент может быть передан.

```
List<? super Integer> intList = new ArrayList<Integer>();
intList.add(new Integer(10));
```

ограниченные маски (Bounded wildcards)

<? extends T> - Ограничение сверху: тип T или любого из подтипов T

<? super T> - Ограничение снизу: от T или любого супертипа T (вплоть до Object)

PECS (Producer Extends Consumer Super) (Джошуа Блох) aka Get and Put principle.

<? super T>

```
class Test {
…
public static <T> void addValue(List<? super T> dest, T value) {
      list.add(value);
}
…
}
```

Использование

```
List<Number> list = new ArrayList<>();
Test.<Number> addValue(list, new Integer(1));
Test.<Number> addValue(list, new Double(2));
```

Пример <? super T> и <? extends T>

```
public class CollectionsUtil {
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        for (int i=0; i<src.size(); i++)  {
            dest.set(i, src.get(i));
        }
    }
}
```

Сырые типы / **raw types**

Сырой тип (raw type) — это имя обобщённого класса или интерфейса без аргументов типа (type arguments).

Обобщенный тип:

```
Box<Shoes> boxOfShoes = new Box<>();
```

Сырой тип:

```
Box boxOfShoes = new Box();
```

Стирание типов / **Type Erasure**

Стирание типов - процесс приведения обобщенного класса к сырому.

- заменяются переменные типов **первым ограничением** или, если ограничения нет – типом **Object**.
- вставляются касты (**cast**) где необходимо
- генерируются **bridge-методы** для сохранения полиморфизма.

- дженерики отсутствуют в run-time

Стирание типов: bridge-методы

До type erasure:

```
public class Node<T> {
    public T data;
    public void setData(T data){this.data = data;}
}
public class IntNode extends Node<Integer> {
    public void setData(Integer data){super.setData(data);}
}
```

После type erasure:

```
public class Node {
    public Object data;
    public void setData(Object data){this.data = data; }
}
public class IntNode extends Node {
    public void setData(Integer data){
        super.setData(data);
    }
}
```

А что, что-то не так?

Стирание типов: bridge-методы

```
public class Node {
    public Object data;
    public void setData(Object data) { this.data = data; }
}
```

Дочерний класс более не переопределяет родительский метод.
Сигнатуры методов не совпадают!

```
public class IntNode extends Node {
    public void setData(Integer data) {
        super.setData(data);
    }

    // Генерируемый bridge-method
    public void setData(Object data) {
        setData((Integer) data);
    }
}
```

Reifiable тип, информация о котором полностью доступна во время выполнения программы.

- примитивы
- необобщенные типы
- сырые типы
- вызовы с неограниченным wildcard (`List<?> list`)
- массивы reifiable типов

Non-reifiable (невосстанавливаемые) типы - это типы, в которых информация удалялась во время компиляции стиранием типов - вызовы обобщенных типов, которые не определены как неограниченные wildcards.

Примерами невосстанавливаемых типов являются List <String> и List <Number>; JVM не может определить разницу между этими типами во время выполнения.

Это накладывает ограничения на использование дженериков…

Невозможно:

- инициализировать дженерики примитивными типами
- создать экземпляры параметров типа
- объявить статические поля с дженериками
- использовать приведение типов или **instanceof** с параметризованными типами
- создать массивы параметризованных типов
- работать с исключениями параметризованных типов
- перегрузить метод, в котором формальные параметры типов каждой перегрузки стираются к одинаковым сырым типам

Задание для самоконтроля

Необходимо написать свой LinkedList
Методы:

- add(E e)
- add(int index, E element)
- E get(int index)
- E remove(int index)
- Iterator<E> iterator()

с использованием wildcards:

- boolean addAll(Collection c)
- boolean copy(Collection c)

ПРИЛОЖЕНИЯ