Урок 7.
Параметризация и обобщенное программирование (Generics)

Гарбузов Николай
Donriver Inc.

DonRiver

Agenda

- Смысл обобщенного программирования
- Метасимвольные аргументы (wildcards)
- Ограниченные типы
- Обобщенные методы
- Иерархии обобщенных классов
- Стирание (erasure)
- Выведение типов и обобщения
- Сравнение обобщенных классов и их приведение
- Ограничения обобщений

Смысл обобщенного программирования

```
interface  Comparator {
        int compare (Object o1, Object o2);
}
```

Смысл обобщенного программирования

```
interface  Comparator {
        int compare (Object o1, Object o2);
}
```

```
Comparator intComp = new Comparator () {
    int compare (Object o1, Object o2) {
        // validate o1, o2 types, convert to Integer
        return i1.compareTo(i2);
    }
}

intComp.compare(new Integer(1), new Integer(3));
```

Смысл обобщенного программирования

```
interface  Comparator {
        int compare (Object o1, Object o2);
}
```

```
Comparator intComp = new Comparator () {
    int compare (Object o1, Object o2) {
        // validate o1, o2 types, convert to Integer
        return i1.compareTo(i2);
    }
}

intComp.compare(new Integer(1), new Integer(3));
```

Смысл обобщенного программирования

- Обобщения введены в Java для повышения безопасности типов
- Упростили процесс преобразования типов

```
interface  Comparator<T> {
        int compare (T o1, T o2);
}
```

Смысл обобщенного программирования

- Обобщения введены в Java для повышения безопасности типов
- Упростили процесс преобразования типов

```
interface  Comparator<T> {
        int compare (T o1, T o2);
}
```

```
Comparator<Integer> intComp = new Comparator<Integer> () {...};
intComp.compare (1,3);

Comparator<Person> persComp = new Comparator<>() {...};
persComp.compare (p1,p2);
```

Смысл обобщенного программирования

- Обобщения введены в Java для повышения безопасности типов
- Упростили процесс преобразования типов

```
interface  Comparator<T> {
        int compare (T o1, T o2);
}
```

```
Comparator<Integer> intComp = new Comparator<Integer> () {...};
intComp.compare (1,3);

Comparator<Person> persComp = new Comparator<>() {...};
persComp.compare (p1,p2);
```

Метасимвольные аргументы (wildcards)

```
interface  Stack<T> {
        void add (T o1);
        T remove();
        int compareTo(Stack<T> st);
}
```

Метасимвольные аргументы (wildcards)

```
interface  Stack<T> {
        void add (T o1);
        T remove();
        int compareTo(Stack<T> st);
}
```

```
Stack<Integer> intStack = new Stack<Integer> () {...};
Stack<Long> longStack = new Stack<Long> () {...};

intStack.compareTo(longStack);
```

Метасимвольные аргументы (wildcards)

```
interface  Stack<T> {
        void add (T o1);
        T remove();
        int compareTo(Stack<?> st);
}
```

```
Stack<Integer> intStack = new Stack<Integer> () {...};
Stack<Long> longStack = new Stack<Long> () {...};

intStack.compareTo(longStack);
```

Метасимвольные аргументы (wildcards)

- Используется <?>
- Обозначает некоторый тип, который не известен на этапе компиляции

Метасимвольные аргументы (wildcards)

- Используется <?>
- Обозначает некоторый тип, который не известен на этапе компиляции

- Запрещает операцию записи (присваивания) к типу, определенному через wildcard!

Ограниченные типы

```
interface  Calculator<T> {
        T sum (T o1, T o2);
}
```

Ограниченные типы

```
interface  Calculator<T> {
        T sum (T o1, T o2);
}
```

Не любой тип можно суммировать!

Ограниченные типы

```
interface  Calculator<T extends Number> {
        T sum (T o1, T o2);
}
```

```
Calculator<Integer> calc = new Calculator<Integer> () {...};
calc.sum(1,2);
```

Ограниченные типы

```
interface  Calculator<T extends Number> {
        T sum (T o1, T o2);
}
```

```
Calculator<Integer> calc = new Calculator<Integer> () {...};
calc.sum(1,2);

Calculator<String> numStack = new Calculator<String> () {...};
```

Ограниченные типы

- Для ограничения обобщенного типа сверху или снизу можно использовать доп. конструкцию ограничения

? extends SomeClass

? super SomeClass

Ограниченные типы

- Для ограничения обобщенного типа сверху или снизу можно использовать доп. конструкцию ограничения

- Во входящих аргументов используют верхнюю границу (extends)

```
class LowBounded {
    int average (Collection<? extends Number> col) {
        Number n = col.get(0);
        Integer i = col.get(1);
        // do calculation
        return average;
    }
}
```

Ограниченные типы

- Для ограничения обобщенного типа сверху или снизу можно использовать доп. конструкцию ограничения

- Во входящих аргументов используют верхнюю границу (extends)

```
class UpperBounded {
    int average (Collection<? extends Number> col) {
        Number n = col.get(0);
        Integer i = col.get(1);
        col.add(new Integer(1));
        // do calculation
        return average;
    }
}
```

Ограниченные типы

- Для ограничения обобщенного типа сверху или снизу можно использовать доп. конструкцию ограничения

- Во входящих аргументов используют верхнюю границу (extends)
- Для выходящих аргументов используют нижнюю границу (super)

```
class LowBounded <T> {
    void fill (Collection<? super T> col, T t, int count) {
        for (int i=0; i < count; i++){
            col.add(t);
        }
    }
}
```

Обобщенные методы

- Можно использовать обобщения на уровне методов в обычном классе

```
class LowBounded {
        <T> void fill (Collection<? super T> col, T t, int count) {
                for (int i=0; i < count; i++){
                        col.add(t);
                }
        }
}
```

Иерархии обобщенных классов

- Можно создавать иерархии классов, в которых применяются обобщения

```
class A {}

class B <T, U extends Comparable<U>> extends A{}

class C <T> extends B<T, Integer> {}

class D extends C <String> {}
```

Иерархии обобщенных классов

- Можно создавать иерархии классов, в которых применяются обобщения

```
class A {}

class B <T, U extends Comparable<U>> extends A{}

class C <T> extends B<T, Integer> {}

class D extends C <String> {}
```

Иерархии обобщенных классов

- Можно создавать иерархии классов, в которых применяются обобщения

```
class A {}

class B <T, U extends Comparable<U>> extends A{}

class C <T> extends B<T, Integer> {}

class D extends C <String> {}
```

Иерархии обобщенных классов

- Можно создавать иерархии классов, в которых применяются обобщения

```
class A {}

class B <T, U extends Comparable<U>> extends A{}

class C <T> extends B<T, Integer> {}

class D extends C <String> {}
```

Иерархии обобщенных классов

- Можно создавать иерархии классов, в которых применяются обобщения
- Можно переопределять обобщенные методы!

```
class A {}

class B <T, U extends Comparable<U>> extends A{}

class C <T> extends B<T, Integer> {}

class D extends C <String> {}
```

Стирание (erasure)

- При компиляции все обобщения стираются и заменяются на Object
- Добавляются операторы приведения типов

Стирание (erasure)

- При компиляции все обобщения стираются и заменяются на Object
- Добавляются операторы приведения типов

Необходимо для обратной совместимости!

Выведение типов и обобщения

- Выведение типа позволяет избежать дублирования типов обобщения после оператора new

```
Stack<Integer> lst = new Stack<Integer>();
```

Выведение типов

- Выведение типа позволяет избежать дублирования типов обобщения после оператора new

```
Stack<Integer> lst = new Stack<Integer>();

Stack<Integer> lst = new Stack<>();
```

Выведение типов

- Выведение типа позволяет не указывать тип обобщения при вызове обобщенного метода (позволяет вызывать обобщенный метод как обычный)

Выведение типов

- Выведение типа позволяет не указывать тип обобщения при вызове обобщенного метода (позволяет вызывать обобщенный метод как обычный)

```
class Calculator {
      static <T extends Number> T average(List<T> lst){};
}

Integer i = Calculator.<Integer>average(intList);
```

Выведение типов

- Выведение типа позволяет не указывать тип обобщения при вызове обобщенного метода (позволяет вызывать обобщенный метод как обычный)

```
class Calculator {
      static <T extends Number> T average(List<T> lst){};
}

Integer i = Calculator.<Integer>average(intList);

Integer i = Calculator.average(intList);
```

Сравнение обобщенных классов и их приведение

- Оператор instanceOf игнорирует обобщения из-за стирания типов!
- Можно приводить класс к обобщенному классу

Ограничения обобщений

- Невозможно узнать тип обобщения в рантайме
- Нельзя объявлять статическое поле типа обобщения
- Обобщенный класс не может расширять Throwable

Home Work

Создать собственную реализацию

- стека(FILO),
- односвязного списка (FIFO)

с использованием обобщений (без использования стандартных коллекций)