**Обобщение типа данных, generic**

Начиная с Java 5 появились новые возможности для программирования, к которым следует отнести поддержку обобщенного программирования, названная в Java **generic**. Эта возможность позволяет создавать более статически типизированный код. Соответственно, программы становятся более надежными и проще в отладке.

**generic** являются аналогией с конструкцией "Шаблонов"(template) в С++. Ожидалось, что дженерики Java будут похожи на шаблоны C++. На деле оказалось, что различия между generic'ами Java и шаблонами С++ довольно велики. В основном generic в Java получился проще, чем их C++-аналог, однако он не является упрощенной версией шаблонов C++ и имеют ряд значительных отличий. Так, в языке появилось несколько новых концепций, касающихся generic'ов – это маски и ограничения.

Рассмотрим 2 примера без использования и с использованием **generic**. Пример без использования *generic* с приведением типа (**java casting**):

List integerList = new LinkedList();

integerList.add(new Integer(0));

Integer x = (Integer) integerList.iterator().next();

В данном примере программист знает тип данных, размещамый в List'e. Тем не менее, необходимо обратить особое внимание на **приведение типа** ("java casting"). Компилятор может лишь гарантировать, что метод *next()* вернёт *Object*, но чтобы обеспечить присвоение переменной типа Integer правильным и безопасным, требуется *java casting*. Приведение типа не исключает возможности появления ошибки "Runtime Error" из-за невнимательности разработчика.

Возникает вопрос: "Как с этим бороться? Каким образом зарезервировать **List** для определенного типа данных?". Данную проблему решают дженерики **generic**. В следующем примере используется *generic* без приведения типов.

1. List<Integer> integerList = new LinkedList<Integer>();

2. integerList.add(new Integer(0));

3. Integer x = integerList.iterator().next();

Обратите внимание на объявления типа для переменной integerList, которое указывает на то, что это не просто произвольный List, а List<Integer>. Кроме этого теперь **java casting** выполняется автоматически.

В примере вместо приведения к Integer, был определен тип списка List. В этом заключается существенное отличие, и компилятор может проверить данный тип на корректность во время компиляции во всем коде. Эффект от **generic** особенно проявляется в крупных проектах: он улучшает читаемость и надежность кода в целом.

**Свойства Generics**

* Строгая типизация.
* Единая реализация.
* Отсутствие информации о типе.

**Объявление generic-класса**

Объявить generic-класс совсем несложно. Пример такого объявления :

package test;

class GenericSample<T>

{

private T value;

public GenericSample(T value) {

this.value = value;

}

public String toString() {

return "{" + value + "}";

}

public T getValue() {

return value;

}

}

Пример использования generic-класса GenericSample :

class TestSample

{

public static void main(String[] args)

{

GenericSample<Integer> value1;

value1 = new GenericSample<Integer>(new Integer(10));

System.out.println(value1);

// Ошибки нет

Integer intValue1 = value1.getValue();

GenericSample<String> value2;

value2 = new GenericSample<String>("Hello world");

System.out.println(value2);

// Здесь возникает ошибка несоответствия типа

Integer intValue2 = value2.getValue();

}

}

**Проблемы реализации generic**

**1. Wildcard**

Рассмотрим процедуру dump, которой в качестве параметров передается Collection<Object> для вывода значений в консоль.

void dump(Collection<Object> c) {

for (Iterator<Object> i = c.iterator(); i.hasNext();){

Object o = i.next();

System.out.println(o);

}

}

List<Object> l; dump(l); // ОК

List<Integer> l; dump(l); // Ошибка

При передаче списка данных с целочисленным типом возникает ошибка. В этом примере список List<Integer> нельзя передавать в качестве параметра в dump, так как он не является подтипом List<Object>.

Проблема состоит в том что данная реализация кода не эффективна, так как Collection<Object> не является полностью родительской коллекцией всех остальных коллекций, грубо говоря Collection<Object> имеет ограничения. Для решения этой проблемы используется **Wildcard** ("?"), который не имеет ограничения в использовании, то есть имеет соответствие с любым типом, и в этом его плюсы. И теперь, мы можем вызвать это с любым типом коллекции.

void dump(Collection<?> c) {

for (Iterator<?> i = c.iterator(); i.hasNext();) {

Object o = i.next();

System.out.println(o);

}

}

**2. Bounded Wildcard**

Рассмотрим процедуру draw, которая рисует фигуры, наследующие свойства родителя Shape. Допустим у Shape есть наследник Circle, и его необходимо "изобразить".

void draw(List<Shape> c) {

for (Iterator<Shape> i = c.iterator(); i.hasNext();){

Shape s = i.next();

s.draw();

}

}

List<Shape> l; draw(l); // ОК

List<Circle> l; draw(l); // Ошибка

Возникла ошибка, связанная с несовместимостью типов. В предложенном решении необходимо определить тип и его подтипы. Это есть так называемое "ограничение сверху". Для этого нужно вместо <Shape> определить <? extends Shape>.

void draw(List<? extends Shape> c) {

for (Iterator<? extends Shape> i = c.iterator();

i.hasNext(); ) {

Shape s = i.next();

s.draw();

}

}

Использование <? super Cycle> позволяет использовать тип Cycle и всех его предков вполоть до Object.

**3. Generic метод**

Определим процедуру addAll, которая в качестве параметров получает массив данных Object[] и переносит его в коллекцию Collection<?>

void addAll(Object[] a, Collection<?> c) {

for (int i = 0; i < a.length; i++) {

c.add(a[i]);

}

}

addAll(new String[10], new ArrayList<String>());

addAll(new Object[10], new ArrayList<Object>());

addAll(new Object[10], new ArrayList<String>()); // error

addAll(new String[10], new ArrayList<Object>()); // error

Ошибки, возникающие в последних строках связаны с тем, что нельзя просто вставить Object в коллекции неизвестного типа. Способ решения этой проблемы является использование "**generic метода**". Для этого перед методом нужно объявить <T> и использовать его.

<T> void addAll(T[] a, Collection<T> c) {

for (int i = 0; i < a.length; i++) {

c.add(a[i]);

}

}

Но все равно после выполнение останется ошибка в третьей строчке :

addAll(new Object[10], new ArrayList<String>()); // Ошибка

Допустим имеется функция, которая находит ближайший объект к точке Glyph из заданной коллекции. Glyph – это базовый тип, и может иметься неограниченное количество потомков этого типа. Также может иметься неограниченное количество коллекций, хранящих элементы, тип которых соответствует одному из этих потомков. Хотелось бы, чтобы функция могла работать со всеми подобными коллекциями, и возвращала элемент, тип которого совпадал бы с типом элемента коллекции, а не приводился к Glyph. Следующий пример не очень удачный:

<T> T findNearest(Collection<T> glyphs, int x, int y) {

...

}

Функция выглядит неплохо, но, тем не менее, не лишена недостатков. Получается так, что функции можно передать коллекцию любого типа. Это усложняет реализацию функции, порождая необходимость проверки типа элемента. Будет гораздо лучше написать так:

<T extends Glyph> T findNearest(Collection<T> glyphs,

int x, int y) {

...

}

Теперь все встает на свои места, и в функцию можно передать только коллекцию, элементы которой реализуют интерфейс Glyph. **generic** сделал свое дело, код получился более типобезопасным.

**4. Generic-классы**

Наследование можно применять и для параметров **generic**-классов:

class <T extends Glyph> GlyphsContainter

{

...

public void addGlyph(T glyph){...}

}

Как в методах, так и в классах можно задать более одного базового интерфейса, который должен реализовывать **generic**-параметр. Это делается при помощи следующего синтаксиса:

class <T extends Glyph & MoveableGlyph> MoveableGlyphsContainter

{

...

public void addGlyph(T glyph){...}

}

В данном примере **generic**-параметр должен реализовывать не только интерфейс Glyph, но и MoveableGlyph. Ограничений на количество интерфейсов, которые должен реализовывать переданный тип, нет. Но в класс можно передать только один, т.к. в Java нет множественного наследования. Типы в этом списке могут быть generic-типами, но ни один конкретный интерфейс не может появляться в списке более одного раза, даже с разными параметрами:

interface Bar<T> {...}

interface Bar1 {...}

public class Foo<T extends Bar<T> & Bar1> {...} // ok

public class Foo<T extends Bar<T> & Bar<Object> & Bar1> {...} // ошибка

**5. Bounded type argument**

Метод копирования из одной коллекции в другую

<T> void addAll(Collection<T> c, Collection<T> c2) {

for (Iterator<T> i = c.iterator(); i.hasNext(); ) {

T o = i.next();

c2.add(o);

}

}

addAll(new AL<Integer>(), new AL<Integer>());

addAll(new AL<Integer>(), new AL<Object>()); //Ошибка

Проблема в том, что две коллекции могут быть разных типов (несовместимость generic-типов). Для таких случаев был придуман **Bounded type argument**. Он нужен если метод, который мы разрабатываем, использовал бы определенный тип данных. Для этого нужно ввести <N extends M> (N принимает только значения M). Также можно корректно писать <T extends A & B & C>. (Принимает значения нескольких переменных).

<M, N extends M> void addAll(Collection<N> c, Collection<M> c2)

{

for (Iterator<N> i = c.iterator(); i.hasNext(); ) {

N o = i.next();

c2.add(o);

}

}

**6. Lower bounded wildcard**

Метод нахождения максимума в коллекции

<T extends Comparable<T>>

T max(Collection<T> c) {

…

}

Пример использования :

List<Integer> il; Integer I = max(il);

class Test implements Comparable<Object> {…}

List<Test> tl; Test t = max(tl); // Ошибка

<T extends Comparable<T>> обозначает что Т обязан реализовывать интерфейс Comparable<T>.

Ошибка возникает из-за того, что Test реализует интерфейс Comparable<Object>. Решение этой проблемы - **Lower bounded wildcard** ("Ограничение снизу"). Суть заключается в том, что необходимо реализовывать метод не только для Т, но и для его супертипов (родительских типов). Например:

List<T super Integer> list;

Теперь можно заполнить List<Integer>, List<Number> или List<Object>.

<T extends Comparable<? super T>>

T max(Collection<T> c) {

…

}

**6. Wildcard Capture**

Реализация метода Swap в List<?>

void swap(List<?> list, int i, int j) {

list.set(i, list.get(j)); // Ошибка

}

Проблема заключается в том, что метод List.set() не может работать с List<?>, так как ему не известен тип List. Для решение этой проблемы используют **Wildcard Capture** (или "Capture helpers"), т.е. обманываем компилятор. Напишем еще один метод с параметризованной переменной и будем его использовать внутри нашего метода.

void swap(List<?> list, int i, int j) {

swapImpl(list, i, j);

}

<T> void swapImpl(List<T> list, int i, int j) {

T temp = list.get(i);

list.set(i, list.get(j));

list.set(j, temp);

}

**Ограничения generic**

Невозможно создать массив generic'ов :

Collection<T> c;

T[] ta;

new T[10]; // Ошибка

Невозможно создать массив generic-классов :

new ArrayList<List<Integer>>();

List<?>[] la = new List<?>[10]; // Ошибка !!

**Преобразование типов**

В **Generics** также можно манипулировать с информацией, хранящийся в переменных.

// Уничтожение информации о типе

List l = new ArrayList<String>();

// Добавление информации о типе

List<String> l = (List<String>) new ArrayList();

List<String> l1 = new ArrayList();

**Наследование исключений в generic'ах**

Возможность использовать параметр **generic-класса** или метода в **throws** позволяет при описании абстрактного метода не ограничивать разработчика, использующего класс или интерфейс, конкретным типом исключения. Но использовать тип, заданный в качестве параметра, в catch-выражениях нельзя.

Кроме того, можно сгенерировать исключение, тип которого задается **generic-параметром**, но экземпляр должен быть создан извне. Это ограничение порождается одним из ограничений Java generic'ов - нельзя создать объект, используя оператор new, тип которого является параметром generic'а.

abstract class Processor <T extends Throwable>

{

abstract void process() throws T; // ok

void doWork()

{

try {

process();

} catch (T e) {

// ошибка времени компиляции

}

}

void doThrow(T except) throws T

{

throw except; // ok

}

}

Необходимо добавить, что тип, переданный в качестве параметра, должен обязательно быть наследником **Throwable**.

Таким образом, **generic**-и в Java получились проще и внесли несколько интересных концепций, таких как маски (wildcard) и ограничения, которые, добавили удобство при работе и помогли решить проблемы. Но, как и любое усложнение языка, эти нововведения затрудняют его понимание и изучение. Появление **generic**-ов сделало язык Java более выразительным и строгим; такие изменения только на пользу.

[Наверх](http://java-online.ru/java-generic.xhtml#top)