

# Использование параметризуемых классов в J2SE 5

Денис Цыплаков

В данной статье рассматривается пример использования параметризуемых классов (generics) в рамках платформы J2SE 5.

## 1. Введение

Одним из нововведений платформы J2SE 5 являются параметризуемые классы (generics): **JSR-014**. Параметризуемые классы не добавляют никакой новой функциональности собственно Java машине. Они облегчают написание программ, позволяя обнаруживать ошибки определенного класса на стадии компиляции.

### 1.1. Небольшой пример

Вот пример ошибочного кода (надо понимать, что это пример, в реальной жизни все несколько иначе, но суть не меняется):

```
//Создаем список в котором мы будем хранить значения типа Long
List lst = new ArrayList();
lst.add(new Long(1));
lst.add("и каким-то хитрым образом сюда затесалась строка");
lst.add(new Long(2));
...
//Теперь читаем из списка
for ( int i = 0; i < lst.size(); i++)
{
    //Здесь во время исполнения на втором шаге
```

```
//будет ошибка приведения типов
Long l = (Long)lst.get(i);
doSomethingWithLong(l);
}
```

Какой способ борьбы предлагает нам J2SE 5? Вот такой:

```
//Создаем список в котором мы будем хранить значения типа Long
//При этом класс ArrayList и интерфейс List в J2SE 5, параметризуемые,
//это значит, что мы можем в коде декларировать значения какого типа,
//будут храниться в списке.
List<Long> lst = new ArrayList<Long> ();

lst.add(new Long(1));

//Вот в этой строке компилятор даст ошибку.
lst.add("и каким-то хитрым образом сюда затесалась строка");

lst.add(new Long(2));
...
//Теперь читаем из списка
//Можно конечно читать как раньше.
for ( int i = 0; i < lst.size(); i++)
{
    //Обратите внимание, здесь приведение типов не нужно.
    Long l = lst.get(i);
    doSomethingWithLong(l);
}

//Но можно и с помощью нового оператора for
for ( Long l : lst )
{
    doSomethingWithLong(l);
}
//Лично мне этот способ более симпатичен.
```

Полное описание синтаксиса, каждый желающий легко может найти в спецификации Java SE 5. Но давайте лучше рассмотрим живой пример.

## 2. Постановка задачи

## Использование параметризуемых классов в J2SE 5

Итак постановка задачи: разработать интерфейс структуры данных позволяющий хранить объекты в виде очереди (я в курсе, что J2SE 5 содержит стандартный интерфейс `Queue` и ряд классов его реализующих. Но для тестовой задачи у него слишком большая функциональность. Для демонстрации использования параметризуемых классов нам нужно что-то простое) и несколько классов, реализующих данный интерфейс:

- Класс хранящий объекты очереди в памяти.
- Класс хранящий данные очереди в сериализованном виде в файлах в определенном каталоге файловой системы.

### 3. Шаг решения 1

Сначала объявим интерфейс

<b>Файл</b>	ru/yandex/lc/gdm/LiteQueue.java
<b>Описание</b>	Интерфейс "легкой" очереди. Интерфейс очень простой. В очередь можно помещать объекты, можно их оттуда извлекать и можно узнать есть ли в очереди хоть один элемент.

```
package ru.yandex.lc.gdm;

//Интерфейс параметризуется одним типом, мы называем этот параметр ElementType
//Имя как можно догадаться может быть любым.
public interface LiteQueue <ElementType>
{
    /**
     * Поместить элемент в очередь.
     * @param element элемент, который следует поместить
     */
    public void push(ElementType element);

    /**
     * Взять следующий элемент очереди.
     *
     * @return следующий элемент очереди. Если очередь пуста - возвращает null.
     */
}
```

```
public ElementType pop();

/**
 * Проверить, не пуста ли очередь.
 * @return true если очередь пуста.
 */
public boolean isEmpty();
}
```

## 4. Шаг решения 2

Теперь опишем класс реализующий интерфейс `LiteQueue`, хранящий данные очереди в памяти, в виде однонаправленного списка.

<b>Файл</b>	ru/yandex/lc/gdm/MemoryQueue.java
<b>Описание</b>	Класс реализующий интерфейс <code>LiteQueue</code> , хранящий данные очереди в памяти, в виде однонаправленного списка.

```
package ru.yandex.lc.gdm;

//Класс параметризуется типом ClassElementType
public class MemoryQueue <ClassElementType>
    implements LiteQueue<ClassElementType>
{
    /**
     * С головы берем элементы.
     */
    private Container head = null;

    /**
     * С хвоста элементы добавляем.
     */
    private Container tail = null;

    public void push(ClassElementType classElementType)
    {
        if (tail == null)
        {
```

## Использование параметризуемых классов в J2SE 5

```
        tail = new Container();
        head = tail;
        tail.data = classElementType;
    }
    else
    {
        Container c = new Container();
        tail.next = c;
        c.data = classElementType;
        tail = c;
    }
}

public ClassElementType pop()
{
    if (head == null)
    {
        return null;
    }
    Container c = head;
    head = head.next;
    if (head == null)
    {
        tail = null;
    }
    return c.data;
}

public boolean isEmpty()
{
    return head == null;
}

/**
 * Класс - контейнер для элементов очереди. Обратите внимание - класс не
 * параметризован, но мы можем использовать ClassElementType т.к. класс
 * Container не статический.
 */
class Container
{
```

```
/**
 * Собственно данные хранящиеся в контейнере
 */
ClassElementType data;

/**
 * Ссылка на следующий элемент.
 */
Container next = null;
}
}
```

### 5. Шаг решения 3

И, наконец, опишем класс, хранящий данные очереди, в сериализованной форме, в каталоге файловой системы. Тут нас ждет одна проблема. Мы будем хранить объекты очереди в сериализованной форме, следовательно эти объекты должны реализовывать интерфейс `Serializable`. Причем желательно, чтобы проверка на соответствие типа элемента контейнера интерфейсу `Serializable`, производилась на стадии компиляции. Синтаксис параметризуемых классов позволяет это описать.

<b>Файл</b>	ru/yandex/lc/gdm/FileQueue.java
<b>Описание</b>	Класс хранящий данные очереди в сериализованной форме в каталоге файловой системы.

```
package ru.yandex.lc.gdm;

import java.io.*;

//Объявляем параметр, требуя, чтобы он реализовывал интерфейс
//Serializable
//
//Вобщем-то мы можем требовать от параметра соответствия сразу нескольким
//интерфейсам. Например:
//<ElementType extends Serializable,Interface1,Interface2>, но для
//данной задачи нам это не надо.
```

## Использование параметризуемых классов в J2SE 5

```
public class FileQueue <ElementType extends Serializable>
    implements LiteQueue<ElementType>
{
    private File dir;

    private long head = 0;

    private long tail = -1;

    /**
     *
     *
     */
    public FileQueue(File dir)
    {
        if (!dir.isDirectory())
        {
            throw new Error(dir.getAbsolutePath() + " должен быть каталогом");
        }
        this.dir = dir;
    }

    public void push(ElementType element)
    {
        File tailFile = new File(dir.getAbsolutePath() +
            File.separator +
            ++tail);

        try
        {
            FileOutputStream fo = new FileOutputStream(tailFile);
            try
            {
                ObjectOutputStream ous = new ObjectOutputStream(fo);
                try
                {
                    ous.writeObject(element);
                }
                finally
                {
                    ous.close();
                }
            }
        }
    }
}
```

```
        }
        finally
        {
            fo.close();
        }
    }
    catch (Exception ex)
    {
        throw new Error(ex);
    }
}

public ElementType pop()
{
    if (head > tail)
    {
        return null;
    }
    try
    {
        File headFile = new File(dir.getAbsolutePath() +
            File.separator +
            head++);
        FileInputStream fi = new FileInputStream(headFile);
        Object obj;
        try
        {
            ObjectInputStream ois = new ObjectInputStream(fi);
            try
            {
                obj = ois.readObject();
            }
            finally
            {
                ois.close();
            }
        }
        finally
        {
            fi.close();
        }
    }
}
```



```
        }
        headFile.delete();
        return (ElementType) obj;
    }
    catch (Exception ex)
    {
        throw new Error(ex);
    }
}

public boolean isEmpty()
{
    return head > tail;
}
}
```

## 6. Тестируем результат

Теперь напишем класс тестирующий результат. Впрочем-то это для нас не так уж интересно. В целом и так понятно как классы будут работать и что должно получаться в результате, но без теста статья выглядит логически незавершенной.

В тестовом классе мы будем проверять прежде всего корректность работы наших классов, ну и заодно сравним скорость работы двух разных реализаций очереди.

Для измерения скорости работы участков кода мы воспользуемся методом `System.nanoTime()` появившемся в J2SE 5. Данный метод класса `System` возвращает нам количество наносекунд относительно некоторой точки в прошлом или будущем. В документации не говорится, какая именно дата является точкой отсчета. Вообще говоря, при каждом запуске Java машины эта дата может быть разной. Т.е. данный метод годится только для измерения временных интервалов в пределах времени работы одной Java машины.

При этом надо понимать, что по крайней мере для современных компьютеров, метод возвращает значение с определенной точностью, определяемой быстротой процессора, архитектурой компьютера и типом операционной системы. Т.е. вообще говоря для какого либо компьютера точность измерений проводимых с помощью `System.nanoTime()` может быть не более сотен микросекунд. А для некоего

гипотетического компьютера будущего метод может возвращать совершенно точное значение в наносекундах.

<b>Файл</b>	ru/yandex/lc/gdm/QueueTester.java
<b>Описание</b>	Тестирующий класс

```
package ru.yandex.lc.gdm;

import java.io.File;

public class QueueTester
{
    public static void main(String[] args)
    {
        testMemoryQueue();
        testFileQueue();
    }

    private static void testFileQueue()
    {
        System.out.println("--- testFileQueue ---");
        LiteQueue<String> q = new FileQueue<String>(new File("./queue"));
        testQueue(q);
    }

    private static void testMemoryQueue()
    {
        System.out.println("--- testMemoryQueue ---");
        LiteQueue<String> q = new MemoryQueue<String>();
        testQueue(q);
    }

    private static void testQueue(LiteQueue<String> mq)
    {
        int ati = 0;
        long start = System.nanoTime();
        for (int i = 0; i < 3; i++)
        {
            System.out.println("    Старт итерации");
        }
    }
}
```

## Использование параметризуемых классов в J2SE 5

```
        for (int j = 0; j < 4; j++)
        {
            mq.push(new String("i:" + ati++));
        }
        while (!mq.isEmpty())
        {
            System.out.println(mq.pop());
        }
    }
    System.out.println("Test length: " +
        (System.nanoTime() - start) * 0.000001 + " ms.");
}
}
```

А вот что получается в результате запуска теста

```
--- testMemoryQueue ---
    Старт итерации
i:0
i:1
i:2
i:3
    Старт итерации
i:4
i:5
i:6
i:7
    Старт итерации
i:8
i:9
i:10
i:11
Test length: 5.01628 ms.
--- testFileQueue ---
    Старт итерации
i:0
i:1
i:2
i:3
    Старт итерации
```

```
i:4  
i:5  
i:6  
i:7  
    Старт итерации  
i:8  
i:9  
i:10  
i:11  
Test length: 104.014896 ms.
```

## 7. Заключение

Итак как мы видим параметризуемые классы достаточно гибкий инструмент позволяющий разработчику отлавливать целый класс ошибок, связанных с приведением типов на стадии компиляции. Что в свою очередь однозначно позволяет повысить надежность Java программ.

Вывод прост. Надо использовать параметризуемые классы везде, где это уместно.

## 8. Список ссылок

1. [JSR 14: Add Generic Types To The Java™ Programming Language](#) Главная страница JSR-014
2. [Programming with the New Language Features in J2SE 5.0](#)
3. [Исходные коды примеров.](#)