Об enum-типах в Java

Денис Гладкий
denis.gladkiy@yahoo.com

Управление предоплаченных карт, ГК ЦФТ

12 августа 2013 г.

Цель данной статьи — познакомить читателя с различными вариантами использования enum-типов в Java. Полное формальное описание синтаксиса доступно в спецификации языка: `http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.9`. Статья предполагает, что читатель знаком с программированием на языке Java.

1 Простейшие enum-типы

Технически, enum (перечисление) — это ссылочный тип, единственные экземпляры которого (константы перечисления, enumeration constants) объявлены в нём, как статические поля. Можно попытаться дать и более общее определение, без привязки к языку: enum — это реализация шаблона проектирования «политон» (или же «мультитон», если на англоязычный манер), класса имеющего в каждый момент времени не более (или строго) X экземпляров. Пример объявления простейшего типа-перечисления:

```
enum DataTransfer
{
    HTTP_BODY_JSON,
    HTTP_BODY_PROTOBUF,
    URL_PARAMETERS,
}
```

Синтаксис разделителей тут допускает некоторые вольности: запятую после объявления последней константы можно не писать. В семантике означенной конструкции нет ничего военного. Она компилируется во что-то подобное:

```
final class DataTransfer extends Enum<DataTransfer>
{
    public static final DataTransfer HTTP_JSON_BODY
        = new DataTransfer();

    public static final DataTransfer URL_PARAMETERS
```

```
        = new DataTransfer();

    public static final DataTransfer HTTP_PROTOBUF_BODY
        = new DataTransfer();

    public static DataTransfer[] values()
    {
        //...
    }
}
```

Вот тут начинаются принципиальные отличия перечислений в Java от аналогичных конструкций других языков. В дополнение к константам компилятор генерирует объявление массива ссылок на них. Доступ к этому массиву осуществляется через метод values() (например, DataTransfer.values()). Порядок ссылок в этом массиве совпадает с порядком объявления соответствующих констант в самом перечислении. Далее, из generic класса Enum наследуются, в частности, методы ordinal() и name(). Первый возвращает индекс константы в вышеозначенном массиве, второй - имя константы в виде строки (java.lang.String). Полезность представленных особенностей перечислений поясняет пример с разбором аргументов командной строки (конечно, в боевой практике лучше использовать Apache CLI):

```
public final class Main
{
    private enum Argument
    {
        ARGUMENT_FILE,
        ARGUMENT_MODE,
    }

    public static void main(final String[] args)
    {
        //the code automatically handles new enum member addition
        if (args.length < Argument.values().length)
        {
            System.err.println("Illegal number of arguments.");

            System.out.print("Usage: app.exe ");

            //automatic tip generation from the constants list
            for (Argument argument : Argument.values())
            {
                System.out.print(argument.name() + ' ');
            }

            return;
        }
```

```
        final String file = args[Argument.ARGUMENT_FILE.ordinal()];
        final String mode = args[Argument.ARGUMENT_MODE.ordinal()];

        //...
    }
}
```

По переменным перечислимого типа можно делать switch:

```
final DataTransfer var = DataTransfer.HTTP_JSON_BODY;
switch(var)
{
    case DataTransfer.HTTP_JSON_BODY:
        //...
        break;
    case DataTransfer.URL_PARAMETERS:
        //...
        break;
    default:
        //...
}
```

Но, как известно, switch, зачастую, не самое хорошее решение. При добавлении новой константы в перечисление, придётся модифицировать и все соответствующие операторы switch. Поэтому стоит задуматься над рефакторингом, как минимум при появлении второго switch'а по одному и тому же перечислению.

У простейших перечислений есть ещё одно полезное применение. Например, рассмотрим следующий код:

```
//uiForm ---- some UI window abstraction
uiForm.show(true, false, true);
```

Тут явно зияет проблема читабельности кода: непонятно, что значат эти булевы значения и как они влияют на показ формы? В языке Objective-C и C# за версией 4.0 эта проблема решена тем, что при вызове метода указываются не только значения его аргументов, но и их имена. В Java же с помощью перечислений вышеозначенный вызов можно сделать более ясным следующим способом:

```
uiForm.show(MODAL, DO_NOT_SHOW_WINDOW_TITLE, FULLSCREEN);
```

Похожего синтаксиса можно добиться и другим способом, к примеру именованными константами типа int или boolean. Однако перечисления лучше, в виду создания нового типа, не совместимого по присваиванию с уже имеющимися. То есть в качестве аргумента типа int можно передать, как специальную именованную константу, так и какое угодно другое выражение, значение которого может и не встречаться среди ожидаемых функцией. Причём данную ситуацию компилятор не сможет обнаружить и об ошибке, в лучшем случае, мы узнаем только после запуска приложения. Так же использование перечисления более code com-

pletion friendly, чем те же int'ы: программисту, вызывающему функцию, сразу предложат то, что нужно, а не ворох «левых» идентификаторов.

Некоторые свойства перечисления, следующие из неизменяемости набора его единственных экземпляров — констант:

— вложенный класс-перечисление является static; компилятор, правда, не запрещает писать данный квалификатор при объявлении перечисления;

— перечисление нельзя объявить во вложенном нестатическом классе;

— перечисление нельзя объявлять анонимно (локально);

— класс-перечисление является final, то есть от него нельзя наследоваться; а вот тут компилятор ведёт себя неконсистентно относительно предыдущего пункта, выдавая ошибку на явное объявление final enum.

— нельзя напрямую (оператором new) создавать экземпляры перечисления, из-за чего нет смысла давать его конструктору квалификатор видимости отличный от private.

— нельзя пометить enum, как abstract-тип.

При использовании перечислений стоит помнить про одни грабли — использование значения, возвращаемого методами ordinal() и name(), для различных сериализационных задач. Более конкретно: **никогда не сохраняйте в данные, переживающие запуски вашей программы, то, что возвращают name() и ordinal()**. Что может произойти между запусками? Верно, может поменяться код. А с ним и порядок следования элементов перечисления (поменяли местами, добавили новый) или их имена (переработка, исправление дефектов). И что же с этим делать? Для ответа на этот вопрос перейдём к следующей части.

2 Перечисления могут иметь поля, методы и конструкторы

```
public enum HttpMethod
{
    GET(ArgumentsTransport.URL),
    PUT(ArgumentsTransport.JSON_BODY),
    POST(ArgumentsTransport.JSON_BODY),
    DELETE(ArgumentsTransport.JSON_BODY),
    ;

    private final ArgumentsTransport argumentsTransport;

    private HttpMethod(final ArgumentsTransport argumentsTransport)
    {
        this.argumentsTransport = argumentsTransport;
    }
```

```
    public final ArgumentsTransport getArgumentsTransport()
    {
        return argumentsTransport;
    }
}
```

Ответ на вопрос из конца предыдущего раздела, примерно, следующий: при необходимости сохранять какой-либо идентификатор константы перечисления, надо заводить в перечислении специальное поле (uid, к примеру), инициализируя его в конструкторе (проверку уникальности идентификаторов можно организовать в статическом конструкторе):

```
enum Bar
{
    _A(0),
    _B(1),
    ;

    static
    {
        //it is better to use TIntHashSet from GNU Trove
        final HashSet<Integer> used = new HashSet<Integer>();

        for (Bar constant : values())
        {
            if (used.contains(constant.getUid()))
            {
                throw new IllegalStateException();
            }

            used.add(constant.getUid());
        }
    }

    private final int uid;

    Bar(int uid)
    {
        this.uid = uid;
    }

    int getUid()
    {
        return uid;
    }
}
```

Стоит так же опасаться ещё одних граблей — циклической зависимости по инициализации данных между перечислениями. Например, вот так писать не надо:

```
enum Foo
{
    ELEMENT(Buz.ELEMENT.getReference()),
    ;

    private Object reference = null;

    Foo(final Object reference)
    {
        this.reference = reference;
    }

    Object getReference()
    {
        return reference;
    }
}

enum Buz
{
    ELEMENT(Foo.ELEMENT),
    ;

    private Object reference = null;

    Buz(final Object reference)
    {
        this.reference = reference;
    }

    Object getReference()
    {
        return reference;
    }
}

public static void main(String[] args)
{
    System.out.println(Foo.ELEMENT.getReference());
}
```

3 Перечисление может иметь абстрактные методы

Абстрактные методы объявляются, как явно, прямо в перечислении, так и неявно из реализуемого интерфейса или наследуемого класса. Синтаксис тут интуитивно понятный и лучше всего его рассмотреть на примере:

```
enum DataTransfer
{
    HTTP_JSON_BODY
    {
        @Override
        public void packData(final HttpRequest request)
        {
            //...
        }
    },

    URL_PARAMETERS
    {
        @Override
        public void packData(final HttpRequest request)
        {
            //...
        }
    },

    HTTP_PROTOBUF_BODY
    {
        @Override
        public void packData(final HttpRequest request)
        {
            //...
        }
    },

    ;

    public abstract void packData(final HttpRequest request);
}
```

Абстрактные методы перечислений, в частности, позволяют решить проблемы, возникающие при использовании switch-конструкций.

Напоследок читателю предлагается следующий код, по максимому использующий возможности перечислений:

```
enum Operation
{
```

```
PLUS("+")
{
    @Override double evaluate(final double x, final double y)
    {
        return x + y;
    }
},
MINUS("-")
{
    @Override double evaluate(final double x, final double y)
    {
        return x - y;
    }
},
TIMES("*")
{
    @Override double evaluate(final double x, final double y)
    {
        return x * y;
    }
},
DIVIDED_BY("/")
{
    private final static double EPSILON = 0.0000001;

    @Override double evaluate(final double x, final double y)
    {
        if (false == check(x, y))
        {
            throw new IllegalArgumentException();
        }
        return x / y;
    }

    @Override boolean check(final double x, final double y)
    {
        return EPSILON <= Math.abs(y);
    }
};

private final String symbol;

Operation(final String symbol)
{
    this.symbol = symbol;
}

abstract double evaluate(final double x, final double y);
```

```
boolean check(final double x, final double y)
{
    return true;
}

private String symbol()
{
    return symbol;
}

private enum Argument
{
    ARGUMENT_X,
    ARGUMENT_Y,
    ARGUMENT_OPERATION
}

public static void main(final String args[])
{
    try
    {
        if (args.length < Argument.values().length)
        {
            System.err.println("Illegal number of arguments.");
            return;
        }

        final HashMap<String, Operation> operations
            = new HashMap<String, Operation>
                (Operation.values().length);

        for (Operation operation : Operation.values())
        {
            operations.put(operation.symbol(), operation);
        }

        final double x
            = Double.parseDouble(args[Argument.ARGUMENT_X.ordinal()]);

        final double y
            = Double.parseDouble(args[Argument.ARGUMENT_Y.ordinal()]);

        final String operationSymbol
            = args[Argument.ARGUMENT_OPERATION.ordinal()];

        final Operation operation = operations.get(operationSymbol);

        if (null == operation)
        {
```

```
                System.out.println(
                    "Unknown operation: " + operationSymbol);

                return;
            }

            if (operation.check(x, y))
            {
                System.out.println(x + " " + operationSymbol
                    + " " + y + " = " + operation.evaluate(x, y));
            }
            else
            {
                System.out.println("Invalid arguments.");
            }
        }
        catch (NumberFormatException e)
        {
            System.out.println("Invalid arguments: " + e.getMessage());
        }
    }
}
```

В заключение хотелось бы ещё раз отметить, что перечисления в Java не просто типизированная замена int'ам, а мощный инструмент абстракции, позволяющий реализовывать, как полиморфное поведение (перегрузка методов в константах), так и т.н. data driven системы с конструкциями, схожими с кодогенерацией (построение структур данных по массиву констант перечисления).