UniSharping: расширение библиотеки

# Введение

Система UniSharping состоит из движка и расширяемой части, которая является мостом между системными классами C# с их методами и конструкциями конечных языков. Вопросы такого соответствия можно условно распределить по таким уровням:

1. Строгое соответствие: в конечном языке есть метод с такими же параметрами и функциональностью;
2. Нестрогое соответствие: аналог есть, но либо параметры немного другие и не в том порядке, либо работает не совсем так же;
3. Экспресс-соответствие: вызов метода моделируется выражением на конечном языке;
4. Блок-соответствие: метод моделируется блоком операций конечного языка;
5. Слабое соответствие: логика настолько сложная, что она моделируется на уровне движка;
6. Принципиальное несоответствие: невозможно промоделировать средствами конечного языка;

Соответствия с 1 по 4 группу задаются на уровне текстовых файлов специального формата и сервисных классов конечного языка. Пятая группа оказалось относительно маленькой – с десяток случаев. Шестую группу мы не рассматриваем вообще – к ней относятся такие методы, которые ориентированы на специфические свойства Windows (например, системный реестр), разные графические возможности и т.п., что ограничивает кроссплатформенность продукта.

Здесь рассматриваются соответствия 1-4 групп.

Текстовые файлы должны располагаться в той же директории, где и исполняемые модули, в следующих папках:

* “C#” – папка с файлами соответствий классов, каждый класс (точнее, тип) в своём файле, формат описывается далее. Внутренняя структура подпапок может быть любой, она роли не играет, файлы собираются из всех папок и подпапок всех уровней;
* “Java” – сервисные классы для Java, при генерации они все попадают в пакет unisharp (или какой задан в xml-файле настроек проекта конвертации), так что результирующее множество пакетов является самодостаточным в рамках стандартного JDK 1.7 и выше;
* “Python” – сервисные классы Python;
* “PHP” – сервисные классы PHP;
* “JS” – сервисные классы JavaScript (Node.js);
* … и т.д. для других будущих языков

Отметим, что множество результирующих языков определяется на уровне движка и нее может быть расширено извне.

Сервисные классы моделируют те классы, которые отсутствуют в конечном языке или плохо соответствуют исходному классу C#. Например, для System.IO.Stream нет явного соответствия в Java, поскольку там есть отдельно InputStream и OutputStream. Поэтому в папке Java есть файл Stream.txt (расширение при генерации будет заменено на корректное) наследные от него MemoryStream, FileStream и пр., обрамляющие нативные java-методы.

Особо отметим статический класс Utils для всех языков – он как бы сборник для мелких сервисных методов, а также для блок-соответствий при настройке. То есть если не удаётся метод промоделировать выражением, то в Utils создаётся соответствующий метод, вызов которого и будет необходимым выражением.

# Формат настроечного файла

Настроечный файл имеет текстовой формат, расширение txt и имя описываемого типа (класса, интерфейса, структуры) без namespace. Например, DateTime.txt, Exception.txt, XmlWriter.txt.

Он состоит из блока описания класса и последующих блоков описаний членов.

Пробелы и переходы на новую строку роли не играют.

## Формат блока класса

<static> {class, interface, enum, struct, delegate}

Имя1 <Имя2 …> <женерик>

< : базовый класс 1 < базовый класс 1 … >>

<язык1 соответствие <язык2 соответствие…>>

В начале идёт ключевое слово, возможно предваряемое static, затем имя вместе с полным Namespace, причём имён может быть несколько через пробел (например, см. Assert или int), далее возможны generic-параметры в угловых скобках. Если есть базовый класс и интерфейсы, то они перечисляются после двоеточия.

Внимание! Здесь и далее везде, где идёт указание системного типа C#, не нужно задавать namespace – только имя. Namespace указывается только после ключевого слова.

После этого идут описания соответствий. Соответствия вообще могут отсутствовать или быть не для всех поддерживаемых языков. Порядок неважен.

Каждое соответствие начинается с ключевого слова конечного языка (java, python, php), после чего указывается класс конечного языка, причём если он не входит в стандартный пакет, то обязательно указывать полное имя с пакетом.

Примеры:

struct System.DateTime

java java.time.LocalDateTime

python datetime.datetime

interface System.Collections.Generic.IList<T> System.Collections.IList : ICollection<T>

**arrayitemtype** T

java java.util.List

python list

Для классов, объекты которых участвуют в foreach, желательно указать тип элемента через arrrayitemtype[[1]](#footnote-1).

Здесь в примере объединены женерик и неженерик – с точки зрения UniSharp между ними нет разницы, и если есть generic, то аналогичный non-generic считается женериком с типом object.

class int System.Int System.Int32

java int **box** Integer

python int

Здесь для java после основного типа указывается после ключевого слова box его упаковочный класс, который используется, например, в женериках[[2]](#footnote-2).

static class System.Math

java Math

python math **import** math

Здесь для Питона после ключевого слова import следует библиотека, которую нужно добавить в начало файла через import, если используется класс math. Кстати, класс может вообще не указываться, и сразу следовать import – это распространяется на все используемые члены класса.

class System.Reflection.Assembly

java **ignored**

python **ignored**

Ключевое слово ignored говорит от том, что любые экземпляры этого класса следует игнорировать, и не выводить операции, в которых участвуют объекты этого типа (вызовы методов генерируются как описывается ниже).

class System.Attribute

java ignored ///**warning**: Inherited class ignored in Java

python ignored ///**warning**: Inherited class ignored in Python

В соответствия в конце можно добавлять комментарии через ///, причём если комментарий начинается с ключевого слова error или warning, то соответствующее сообщение будет выводиться движком UniSharping при формировании списка ошибок и предупреждений.

class System.IO.Stream

java **service** Stream

python io.IOBase import io

Для сервисных классов используется ключевое слово service. В примере для ссылки на класс Stream.txt из папки “Java”.

## Формат блока описания для метода

<**static**> **method** тип\_значения имя<generics>(<параметр1 <, параметр 2…>>)

<**throws** список исключений>

<язык1 соответствие <язык2 соответствие…>>

Здесь и далее типы значений (ТЗ) могут быть:

* void – ТЗ;
* имя системного типа C# (без указания namespace) – ТЗ;
* имя generic-параметра класса или текущего метода – ТЗ;
* если v – ТЗ, то v[ ] – тоже ТЗ (массив);
* если v – имя системного типа женерик, то v<ТЗ> - ТЗ.

Ссылки на имена системных типов можно только на такие, для которых есть настроечные файлы с определениями. Нельзя ссылаться на типы, которые

Для параметров можно указывать:

* ТЗ (кроме void);
* ? – означает один параметр любого типа;
* \* - означает любое число параметров любого типа, в том числе и нулевое;
* ref параметр – для параметров типа ref;
* out параметр – для параметров типа out;

Имён можно задавать несколько через пробел. Метод может иметь generic-параметры в угловых скобках.

Если аналог в Java формирует исключения, то обязательно необходимо перечислять аналоги C# для исключений Java в списке throws. Отметим, что имена должны быть именно C#, причём такие, чтобы для них имелся соответствующий настроечный файл. Если в C# не оказалось прямого аналога, то его всегда можно добавить – в реальном коде C# он никогда не встретится, а для моделирования списка throws пригодится (см, например, SAXException). Лишнего тоже добавлять не надо – список должен точно моделировать список java-метода.

Каждое соответствие начинается с ключевого слова конечного языка (java, python, php…), далее идёт определение. В общем случае определение – это выражение конечного языка, в котором используются следующие шаблонные конструкции, на место которых при генерации подставляются соответствующие значения:

* {0} – первый параметр метода при вызове;
* {1} – второй параметр и т.д.;
* {!} – левая часть для вызова нестатического метода (выражение);
* {\*} – текущий класс для вызова статического метода.

Например, метод Substring для string

method string Substring(int, int)

java {!}.substring({0}, {0} + {1})

python {!}[{0}:{0}+{1}]

В C# вторым параметром служит длина, а в java-варианте вторым параметром должна быть позиция сразу за концом извлекаемого фрагмента, что и отражено в шаблоне. На место {!} будет подставлено выражение левой части.

Допустима короткая форма записи, если параметры оригинала полностью совпадают с аналогом:

method string Replace(char, char)

java =replace

python =replace

В данном примере =replace эквивалентно {!}.replace({0}, {1}).

В принципе определением может быть любое выражение конечного языка, сколь угодно сложное. Вот примеры нетривиального определения из BitConverter:

method byte[] GetBytes(int)

java java.nio.ByteBuffer.allocate(4).order(java.nio.ByteOrder.LITTLE\_ENDIAN).

putInt({0}).array()

python ({0}).to\_bytes(4, byteorder="little")

Если выражением не удаётся смоделировать, то задачу может решить метод в статическом классе Utils. То есть можно оформить код методом в Utils и указать её в определении. Например, для char:

method static bool IsWhiteSpace(char)

java Utils.isWhitespace({0})

python Utils.isWhitespace({0})

Вроде есть системные аналогичные функции, но они, как оказалось, работают не совсем тождественно C# (например, 0x1E в C# false, а в Java true). Поэтому написаны соответствующие функции в Utils для каждого языка:

static java.util.ArrayList<Integer> m\_Whitespaces =

new java.util.ArrayList<Integer>(java.util.Arrays.asList(

0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x20, 0x85, 0xA0, 0x1680, 0x2000,

0x2001, 0x2002, 0x2003, 0x2004, 0x2005, 0x2006, 0x2007, 0x2008,

0x2009, 0x200A, 0x2028, 0x2029, 0x202F, 0x205F, 0x3000 ));

public static boolean isWhitespace(char ch) {

if(m\_Whitespaces.contains((Integer)(int)ch)) return true;

return false;

}

wsChars = [0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x20, 0x85, 0xA0, 0x1680, 0x2000, 0x2001, 0x2002, 0x2003, 0x2004, 0x2005, 0x2006, 0x2007, 0x2008, 0x2009, 0x200A, 0x2028, 0x2029, 0x202F, 0x205F, 0x3000]

@staticmethod

def isWhitespace(ch):

co = ord(ch[0])

return co in Utils.wsChars

Надеюсь, идея понятна, примеры см. в реальных настройках.

Отметим ещё некоторые возможности. В шаблонных конструкциях можно указывать значения по умолчанию параметров после номера и знака равенства: {n=…}. Например, для Encoding:

method char[] GetChars(byte[], \*)

java Utils.decodeCharsetArr({!}, {0}, {1=0}, {2=-1})

python hardcode EncodingGetStringPython

В этом примере есть ещё ключевое слово hardcode, после которого следует имя реализованной в движке процедуры генерации этой функции для Python. Алгоритм не получилось смоделировать на конечном языке, и пришлось хардкодить. Такие случаи довольно редки, процесс хардкода пока не описывается (если потребуется, то его реализуют разработчики движка).

method void Write(string, \*)

java .format print

python .format print

Здесь у Console для форматированного вывода используется поддержанная на уровне движка процедура формирования результата, которая анализирует константную строку формата. В будущем планируется избавиться от этого ограничения, но пока оно есть.

method void WriteTo(Stream) throws IOException

java =writeTo

python shutil.copyfileobj({!}, {0}) **import** shutil

Здесь пример списка исключений, а также после ключевого слова import библиотека, используемая текущим выражением (если такая библиотека указана на уровне определения класса, то здесь указывать не нужно).

## Формат блока описания для конструктора

**.ctor** (<параметр1 <, параметр 2…>>)

<**throws** список исключений>

<язык1 соответствие <язык2 соответствие…>>

Здесь всё аналогично, как для методов, короткая форма (равенство) говорит о полном соответствии конструкторов и стандартном вызове.

Для DateTime:

.ctor(\*)

java java.time.LocalDateTime.of({0=1}, {1=1}, {2=1}, {3=0}, {4=0}, {5=0})

python {\*}({0=1}, {1=1}, {2=1}, {3=0}, {4=0}, {5=0})

Для StringBuilder:

class System.Text.StringBuilder

java StringBuilder

python io.StringIO import io

.ctor()

java =

python =

.ctor(?)

java =

python Utils.newStringIO({0})

## Формат блока описания для поля

<**static**> **field** тип\_значения имя

<язык1 соответствие <язык2 соответствие…>>

Соответствие задаётся также шаблоном, допустима короткая форма, когда после равенства задаётся имя аналогичного поля в классе-аналоге.

field double PI

java =PI

python =pi

field static DateTime MinValue

java java.time.LocalDateTime.MIN

python datetime.datetime.min

## Формат блока описания для элемента энумератора (enum)

**field** имя

<язык1 соответствие <язык2 соответствие…>>

Приведём пример полного описания для SeekOrigin:

enum System.IO.SeekOrigin

java int

python io import io

field Begin

java 0

python {\*}.SEEK\_SET

field Current

java 1

python {\*}.SEEK\_CUR

field End

java 2

python {\*}.SEEK\_END

Обратим внимание, что здесь в Java нет прямого аналога, и значения просто промоделированы целыми числами.

## Формат блока описания для свойства (property)

Свойство описывается двумя независимыми блоками – для get и для set (если есть).

<**static**> **property { get/set }** тип\_значения имя

<**throws** список исключений>

<язык1 соответствие <язык2 соответствие…>>

Соответствие задаётся методом, причём для set метод считается с одним параметром.

ВНИМАНИЕ! Здесь короткая форма после знака равенства недопустима, так как это приводит к неоднозначности толкования.

Вот некоторые свойства DateTime:

property get int Second

java {!}.getSecond()

python {!}.second

property get int Millisecond

java ({!}.getNano() / 1000000)

python ({!}.microsecond / 1000)

property static get DateTime Now

java {\*}.now()

python {\*}.now()

property static get DateTime Today

java java.time.LocalDateTime.of(java.time.LocalDate.now(), java.time.LocalTime.of(0, 0))

python {\*}.today()

А вот для Stream:

property get long Position throws IOException

java {!}.getPosition()

python {!}.tell()

property set long Position throws IOException

java {!}.setPosition({0})

python {!}.seek({0}, io.SEEK\_SET)

Это для Encoding:

property static get Encoding UTF8

java java.nio.charset.Charset.forName("UTF-8")

python "UTF-8"

property static get Encoding BigEndianUnicode

java java.nio.charset.Charset.forName("UTF-16BE")

python "UTF-16BE" ///**error** this encoding not supported in python

Обратим внимание ещё на одну возможность при задании соответствий (для любых элементов, не только свойств). Если в комментарий начинается с error или warning, то при парсинге соответствующие строки попадают в ошибки и предупреждения при обнаружении этого элемента в коде.

## Формат блока описания для индексаторов (indexer)

Как и свойство, индексатор описывается двумя независимыми блоками – для get и для set (если есть).

<**static**> **indexer {get/set}** тип\_значения имя [параметр1 <,параметр2…>]

<**throws** список исключений>

<язык1 соответствие <язык2 соответствие…>>

Пример для StringBuilder:

indexer get char [int]

java {!}.charAt({0})

python Utils.getCharAtStringIO({!}, {0})

indexer set char [int]

java {!}.setCharAt({0}, {1})

python Utils.setCharAtStringIO({!}, {0}, {1})

Отметим, что для set последний параметр является значением, но он явно в заголовке блока описания не фигурирует.

Вот пример для IDictionary<K, V>:

indexer get V [K]

java {!}.get({0})

python {!}[{0}]

indexer set V [K]

java {!}.put({0}, {1})

python {!}[{0}] = {1}

# Сервисные (обёрточные) классы

Когда аналог класса полностью отсутствует в конечном языке, то можно этот аналог создать самим и поместить файл в соответствующую директорию Java, Python, PHP… Но всё равно в C# нужно создать настроечный файл, в котором сослаться на сервисный класс с помощью ключевого слова service.

Рассмотрим пример для XmlWriter. В Java есть аналог javax.xml.stream.XMLStreamWriter, но из-за некоторых нестыковок напрямую его использовать не получается, и приходится создавать обёртку над этим классом и размещать его в папке Java. Вот начальный фрагмент этого файла:

public class XmlWriterWrapper implements AutoCloseable {

public javax.xml.stream.**XMLStreamWriter** wr;

private java.io.FileOutputStream str;

private java.io.CharArrayWriter saw;

private StringBuilder sbres;

private FileStream fstr;

public String encoding = "utf-8";

public XmlWriterWrapper(String fname, XmlWriterSettings s) throws java.io.FileNotFoundException, javax.xml.stream.XMLStreamException {

str = new java.io.FileOutputStream(fname);

javax.xml.stream.XMLOutputFactory fact = javax.xml.stream.XMLOutputFactory.newInstance();

if(s != null) {

//if(s.getIndent())

//fact.setProperty("indent", "true");

if(s.getEncoding() != null)

encoding = s.getEncoding().name();

}

wr = fact.createXMLStreamWriter(str, encoding);

}

В Python аналога нет вообще – по крайней мере в системных библиотеках. Наверняка где-то есть, но мы ведь решили ничего стороннего не использовать, тем более простые аналоги. Поэтому такой аналог Xml.txt (имя файла в принципе может быть любым) был написан и размещён в папке Python, вот его начальный фрагмент:

import io

class XmlWriterSettings:

def \_\_init\_\_(self):

self.encoding = None

self.indent = False

self.indentChars = '\r\n'

class XmlWriter:

def \_\_init\_\_(self) -> None:

self.settings = None

self.\_\_m\_stream = None

self.\_\_m\_str\_build = None

self.\_\_m\_file\_name = None

self.\_\_m\_nodes = list()

self.\_\_m\_elem\_not\_ended = False

def \_\_enter\_\_(self): return self

def \_\_exit\_\_(self, typ, val, traceback): self.close()

@staticmethod

def create\_stream(output : io.IOBase, settings\_ : XmlWriterSettings=None) -> 'XmlWriter':

if (settings\_ is None):

settings\_ = XmlWriterSettings()

res = XmlWriter()

res.settings = settings\_

res.\_\_m\_stream = output

return res

Вот начальный фрагмент настроечного файла XmlWriter.txt из C#:

class System.Xml.XmlWriter

java service XmlWriterWrapper

python service XmlWriter

method static XmlWriter Create(string, \*) throws XmlException FileNotFoundException

java new {\*}({0}, {1=null})

python {\*}.create\_file({0}, {1=null})

method static XmlWriter Create(Stream, \*) throws XmlException FileNotFoundException

java new {\*}({0}, {1=null})

python {\*}.create\_stream({0}, {1=null})

method static XmlWriter Create(StringBuilder, \*) throws XmlException

java new {\*}({0}, {1=null})

python {\*}.create\_string({0}, {1=null})

method void Close()

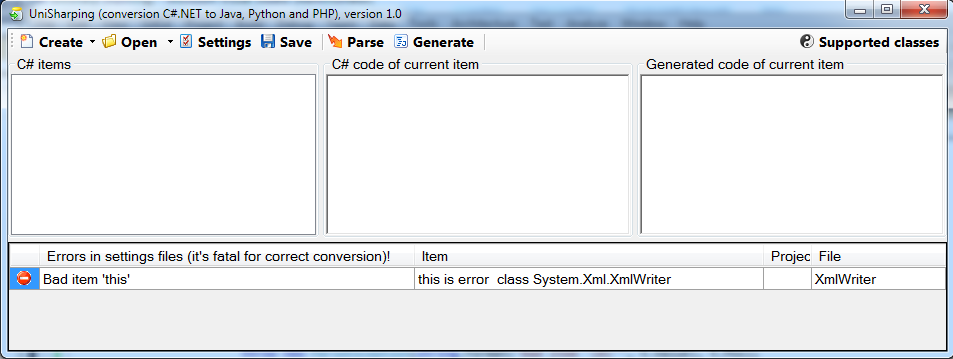
java =close

python =close

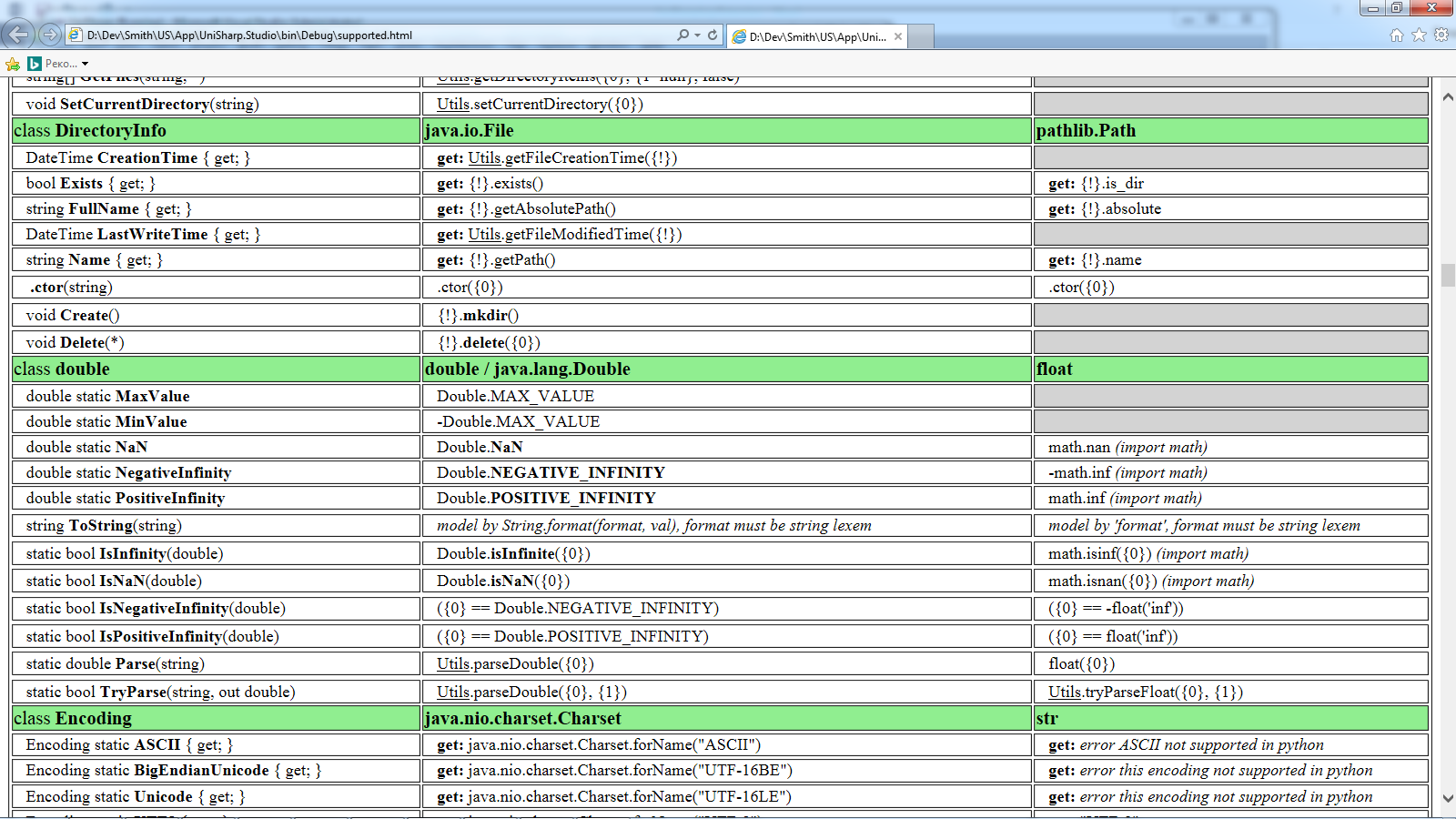
И здесь уже ссылки на методы идут на те, которые в сервисных классах. Сервисные классы попадают в результирующий пакет при генерации. Имя пакета unisharp, но можно его поменять на уровне xml-файла с конфигурацией (см. UniSharping.Overview.docx).

# Отладка настройки

Настроечные файлы считываются движком из той директории, где он сам находится. Если там обнаружены ошибки, то они при запуске UniSharping.Studio выводятся в нижнем окне ошибок с указанием текста сообщения, исходной строки с ошибкой и класса:



Возможно, в будущем разработаем более удобные средства отладки, но пока редактирование в текстовом редакторе и проверка путём запуска. Список поддержанных классов и членов можно получить по кнопке “Supported classes” на главной панели – будет сформирован отчёт в формате HTML и отображён в текущем браузере. В нём можно будет увидеть, как движок интерпретирует настроечные файлы:



Кстати, этот же список из web-демо можно получить по ссылке <http://unisharping.ru/DownloadFile.aspx?file=SupportedList>

1. Возможно, позже мы от этого откажемся и будем определять по базовым интерфейсам. [↑](#footnote-ref-1)
2. List<int> представляется ArrayList<Integer>, что является аналогом List<int?>, но в Java не поддержаны женериками примитивные типы. [↑](#footnote-ref-2)