XièXiè Programming Guide

Lukas Hermanns

March 2015

Contents

About the Author

My name is Lukas Hermanns (age-group 1990) and I started this project during my studies in 2014. By now I have over 12 years of experience in computer programming, started at the age of 12. I have been writing programs in Basic languages such as QBasic, PureBasic, and Blitz3D; in high level languages such as C, C++, C#, Objective-C, and Java; but also in scripting languages such as JavaScript and Python. I'm actually a preferred programmer in C++ (meanwhile C++11), low level stuff, and graphics programming with shading languages such as GLSL and HLSL.

However, the XièXiè programming language is intended to be simple and not tuned for performance. It was originally designed to be used for scripting in video games, but can also be used for general purposes.

Chapter 1

Introduction

The design of the XièXiè programming language is overall influenced by Java , C++ , and Python .

Chapter 2

Syntax

2.1 Basics

2.1.1 Commentaries

Commentaries are a fundamental part of programming languages and they are nearly identical to those in JAVA :

```
1 // Single-line comment
2
3 /* Single-line comment */
4
5 /*
6 Multi-line comment
7 */
```

Although they are very similar to the commantaries in JAVA , nested multi-line comments are allowed as well:

```
1 /*
2 Outer comment
3 /* Nested comment */
4 */
```

2.1.2 Identifiers

Identifiers (for variables, classes, etc.) must only contain alpha-numeric characters and the underscore. They must also begin with a letter or an underscore. But there is a small exception: they must not begin with `__xx__`, because all identifiers with this prefix are reserved for internal use of the compiler.

Valid identifiers are:

- `_name_`
- `FooBar`
- `number_of_wheels`
- `Customer01`
- ...

Invalid identifiers are:

- `naïve`
- `Foo.Bar`
- `number-of-wheels`
- `3over2`
- ...

2.1.3 Literals

These are the kinds of literals:

- Boolean Literal: `true`, `false`
- Integer Literal (Binary): e.g. `0b11001`, `0b0000`
- Integer Literal (Octal): e.g. `0o24`, `0o01234567`
- Integer Literal (Decimal): e.g. `3`, `12`, `999`, `1234567890`
- Integer Literal (Hexa-Decimal): e.g. `0xff`, `0x00`, `0xaB29`
- Float Literal: e.g. `0.0`, `3.5`, `12.482`
- String Literal: e.g. `"Foo Bar"`, `"Hello, World"`, `"\n\t"`, `"1st fragment" "2nd fragment`
- Verbatim String Literal: e.g. `@"\home\test"`, `@"a ""b"" c"`

2.2 Operators

The most operators as in JAVA or C++ are also available in XIÈXIÈ :

```
/* Arithmetic Operators */
a + b   // Addition
a - b   // Substration
a * b   // Multiplication
a / b   // Division
a % b   // Modulo
a << b  // Left Shift
a >> b  // Right Shift
-a      // Negate

/* Bitwise Operators */
a & b   // Bitwise AND
a | b   // Bitwise OR
a ^ b   // Bitwise XOR
~a      // Bitwise NOT

/* Boolean Operators */
not a   // Logic NOT
a and b // Logic AND
a or b  // Logic OR

/* Relation Operators */
a = b   // Equality
a != b  // Inequality
a < b   // Less
a <= b  // Less Or Equal
a > b   // Greater
a >= b  // Greater Or Equal
```

As the interested reader may have noticed, the *equality operator* is different to that in the most languages. This is because the *copy assignment* operators is `:=` and not `=`.

```
x := 5       // ok, set x to 5
a, b, c := 4 // ok, set a, b, and c to 4
a := b := 3  // error, b := 3 is not an expression
```

In the above example `a := b := 3` is invalid, because on the right hand side of the first `:=` must be an expression. But per definition in XIÈXIÈ `b := 3` is not an expression, but a statement! The variable list assignment (`a, b, c := ...`) is a comfort functionality, which is only supported for the copy assignment.

The modify-assign operators are available as well:

```
a += b
a -= b
a *= b
a /= b
a %= b
a <<= b
a >>= b
```

```
a &= b
a |= b
a ^= b
```

However, they are also not allowed inside another expression.

2.3 Type Denoters

2.3.1 Built-in Types

There are only the following three built-in data types:

```
bool  // Boolean type; can be 'true' or 'false'
int   // 32-bit signed integral type
float // 32-bit floating-point type
```

2.3.2 Objects

Types for class objects (more about classes in chapter 3) are written as follows:

```
// Empty string
String s

// List of strings
String s1 := "Hello, World", s2 := "Foo", s3 := "Bar"
```

2.3.3 Arrays

The only generic way for lists are the built-in arrays:

```
// Declare array objects with initializer lists
int[] intArray := { 1, 2, 3 }
float[][] floatArrayArray := { { 0.0, 1.5 }, { 3.5, 1.23 } }
String[] stringArray := { "a", "b", "c" }

// Access array elements
String s1 := stringArray[0]
String s2 := stringArray[intArray[0]]
float[] floatArray := floatArrayArray[1]
```

2.3.4 Automatic Type Deduction

Whereas automatic type deduction in C++11 is a very complex language feature, in XièXiè it can be summarized in this section. There are two keywords for automatic type deduction: `var` and `const`. As the name implies `var` denotes a variable type and `const` denotes a constant type. The latter type is the only way to define constants in XièXiè . Here are a few examples:

```
var i  := 1     // i is from type 'int'
var f  := 3.5   // f is from type 'float'
var s  := "."   // s is from type 'pointer of String'
var a  := { 5 } // a is from type 'array of int'
var aa := {     // aa is from type 'array of array of String'
  { "test" },
  { "a", "b" }
}

const ci := 5       // ci is a constnat int with value 5
const cj := ci*2    // cj is a constant int with value 10
const cf := 3.14    // cf is a constant float
const cb := ci > cj // cb is a constant bool with value 'false'
```

Chapter 3

Classes

A XièXiè program can only contain of class declarations. And classes can only be defined in the global scope. That means everything procedure must be defined inside a class and inner classes are currently not supported. A simple example program could look like this:

```
1 // XieXie Hello World Program
2 import System
3 class HelloWorld {
4     [[entry]]
5     static void main() {
6         System.out.writeLine("Hello, World!")
7     }
8 }
```

This merely writes the line "`Hello, World!`" to the standard output. Let's take a closer look at each line.

Line 2 imports the `System` class from the standard XièXiè library:

```
1 import System
```

This can also be omitted if the respective class file (here "System.xx") is added to the compilation process.

Line 3 declares the class `HelloWorld` which implicitly inherits from the base class `Object`, like it is done in Java :

```
1 class HelloWorld { /* ... */ }
```

To inherit from other classes, just write a colon and the identifier of the base class:

```
1 class SubClass : BaseClass { /* ... */ }
```

There is no multiple inheritance like in C++ or interfaces like in Java !

The next two lines declare the procedure `main`. In line 4, in *attribute* is defined for this procedure. This makes the procedure to the main **entry point**:

```
1 [[entry]]
2 static void main() { /* ... */ }
```

There are several attributes for class-, procedure-, or variable declarations. Currently only two attributes are supported:

```
1  // class A is marked as 'deprecated'
2  [[deprecated]]
3  class A {}
4
5  // class B is marked as 'deprecated' with a hint
6  [[deprecated("hint...")]]
7  class B {}
8
9  // procedure M1 is marked as the main entry point
10 [[entry]]
11 static void M1() {}
12
13 // procedure M2 with return type is marked as an
14 // alternative entry point named "entryAlt1"
15 [[entry("entryAlt1")]]
16 static int M2() { return 0 }
```

```
17
18 // procedure M3 with arguments 'args' is marked as
19 // an alternative entry point named "entryAlt2"
20 [[entry("entryAlt2")]]
21 static void M3(String[] args) {}
```

The last line of code prints the message to the standard output:

```
1 System.out.writeLine("Hello, World!")
```

`System` is a class from the standard XièXiè library, `out` is a *static* member from the type 'OutputStream', and `writeLine` is a function which takes a string as input.