J--语言设计文档

J--语言是一款在风格上类似于C++和Java的面向对象的编程语言，但是对于两者语言的特性进行简化，较为轻便灵巧。可以作为上述两种语言的一种入门式语言。

## 词法语法定义

### Tokens

<type name> ::= <identifier>

<expression name> ::= <identifier>

<method name> ::= <identifier>

<identifier> ::= letter { letter | digit | \_ }\*

<literal> ::= <integer literal> | <floating-point literal> | <boolean literal> | <character literal> | <string literal> | <null literal>

<integer literal> ::= <decimal numeral>

<decimal numeral> ::= 0 | <non zero digit> <digits>?

<digits> ::= <digit> | <digits> <digit>

<digit> ::= 0 | <non zero digit>

<non zero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::= [A-Z] | [a-z]

<floating-point literal> ::= <digits> . <digits>

<signed integer> ::= <sign>? <digits>

<sign> ::= + | -

<boolean literal> ::= true | false

<character literal> ::= ' <single character> ' | ' <escape sequence> '

<single character> ::= <input character> except ' and \

<string literal> ::= " <string characters>?"

<string characters> ::= <string character> | <string characters> <string character>

<string character> ::= <input character> except " and \ | <escape character>

<null literal> ::= null

<keyword> ::= boolean | break | char | class | continue | do | double | else | extends | for | if | instanceof | int | new | private | protected | public | return | super | this | void | while | foreach

### Expressions

< expression> ::= <conditional expression> | <assignment>

<assignment> ::= <left hand side> = <expression>

<left hand side> ::= < e> | <field access> | <array access>

<conditional expression> ::= <conditional or expression>

<conditional or expression> ::= <conditional and expression> | <conditional or expression> || <conditional and expression>

<conditional and expression> ::= < equality expression > | <conditional and expression> && <equality expression>

<equality expression> ::= <relational expression> | <equality expression> == <relational expression> | <equality expression> != <relational expression>

<relational expression> ::= < additive expression> | <relational expression> < < additive expression> | <relational expression> > < additive expression> | <relational expression> <= < additive expression> | <relational expression> >= < additive expression>

<additive expression> ::= <multiplicative expression> | <additive expression> + <multiplicative expression> | <additive expression> - <multiplicative expression>

<multiplicative expression> ::= <unary expression> | <multiplicative expression> \* <unary expression> | <multiplicative expression> / <unary expression> | <multiplicative expression> % <unary expression>

<cast expression> ::= ( <primitive type> ) <unary expression> | ( <reference type> ) <unary expression not plus minus>

<unary expression> ::= + <unary expression> | - <unary expression> | <unary expression not plus minus>

<unary expression not plus minus> ::= <primary> | <expression name> | ! <unary expression> | <cast expression>

<method invocation> ::= <method name> ( <argument list>? ) | <primary> . <identifier> ( <argument list>? ) | super . <identifier> ( <argument list>? )

<field access> ::= <primary> . <identifier> | super . <identifier>

<primary> ::= <primary no new array> | <array creation expression>

<primary no new array> ::= <literal> | this | ( <expression> ) | <class instance creation expression> | <field access> | <method invocation> | <array access>

<class instance creation expression> ::= new <class type> ( <argument list>? )

<argument list> ::= <expression> | <argument list> , <expression>

<array creation expression> ::= new <primitive type> [ <expression> ]

<array access> ::= <expression name> [ <expression> ] | <primary no new array> [ <expression>]

### Blocks and Commands

<block> ::= { <block statement>\* }

<block statement> ::= <local variable declaration>| <statement>

<local variable declaration> ::= <type> <variable declarators>;

<statement> ::= <statement without trailing substatement> | <if statement> | <while statement> | <for statement> | <foreach statement>

<statement without trailing substatement> ::= <block> | <empty statement> | <expression statement> | <do statement> | <break statement> | <continue statement> | <return statement>

<empty statement> ::= ;

<expression statement> ::= <statement expression> ;

<statement expression> ::= <assignment> | <method invocation> | <class instance creation expression>

<if statement>::= if ( <expression> ) <statement> | if ( <expression> ) <statement> else <statement>

<while statement> ::= while ( <expression> ) <statement>

<do statement> ::= do <statement> while ( <expression> ) ;

<for statement> ::= for ( <for init>? ; <expression>? ; <for update>? ) <statement>

<for init> ::= <statement expression list> | <local variable declaration>

<for update> ::= <statement expression list>

<foreach statement> ::= foreach(<local variable declaration > : <expression name>) <statement>

<statement expression list> ::= <statement expression> | <statement expression list> , <statement expression>

<break statement> ::= break ;

<continue statement> ::= continue ;

<return statement> ::= return <expression>? ;

### Types

<type> ::= <primitive type>[]? | <class type> []?

<primitive type> ::= int | double | char | boolean

<class type> ::= <type name>

<modifier> ::= public | protected | private

### Declarations

<type declarations> ::= (<type declaration>)\*

<type declaration> ::= <class declaration> | ;

<class declaration> ::= class <identifier> [extends <class type>] { (<class body declaration>)\* }

<class body declaration> ::= <field declaration> | <method declaration> | <constructor declaration>

<constructor declaration> ::= <constructor modifiers>? <constructor declarator> <constructor body>

<constructor modifiers> ::= < modifier> | <constructor modifiers> <constructor modifier>

<constructor declarator> ::= <simple type name> ( <formal parameter list>? )

<formal parameter list> ::= <formal parameter> | <formal parameter list> , <formal parameter>

<formal parameter> ::= <type> <variable declarator id>

<constructor body> ::= { <explicit constructor invocation>? <block statements>? }

<explicit constructor invocation>::= this ( <argument list>? ) | super ( <argument list>? );

<field declaration> ::= < modifier> <type><variable declarator id>[ =<expression>];

<variable declarator id> ::= <identifier> | <identifier> [ ]

<method declaration> ::= <method header> <block>

<method header> ::= < modifiers><result type> <method declarator>

<result type> ::= <type> | void

<method declarator> ::= <identifier> ( <formal parameter list>? )

<variable initializers> ::= <expression> | <variable initializers> , <expression>

### Programs

<compilation unit> ::= <type declarations>?

## 语言特性

### 类型系统

本语言属于强类型语言，在进行类型转换时要进行显示的类型转换。

* + 1. 基础类型: 在函数传参时传的是值

int ：32位有符号整数

double： 64位有符号浮点数

char ：8位有符号字符

boolean： 8位二值逻辑

* + 1. 引用类型：在函数传参的时传的是引用

数组，其中char[]当做字符串

自定义类

### 控制流

控制的逻辑包括有顺序逻辑

If-then, if-then-else 判断语句

while, do-while, for, foreach循环语句。

### 类

* + 1. 访问权限

类的成员可以进行public ,private和protected的声明类中的函数通过括号传递参数然后用return返回返回值，用.操作符调用成员变量。

* + 1. 类的继承

类之间可以进行继承，只有public的声明可以被外部调用，private只有自己可以调用，protected的成员可以被继承类调用，类中所定义的函数可以被子类的函数继承

* + 1. this与super

类的定义中，this表示当前类，super表示父类

* + 1. 重载

函数或操作符根据参数列表的不同将执行出不同的效果

### 操作符与内建函数

* + 1. 操作符

支持常见计算表达式（例如算数运算+，-，\*，/，%，()，逻辑运算，优先级处理等）。

支持注释，以#开头

* + 1. 输入、输出

输入：input(Object o)，将输入的字符存入o中，此函数有不同类型的重载

输出：print(Object o)，将o转化为字符串后输出

* + 1. 数组的操作

数组有方法int length()返回数组的长度

## 运行时模型

### 主函数

程序文件的后缀名为jmm。程序在执行时，会寻找与文件名\*\*\*.jmm相同的类\*\*\*中的主函数public void main()，以此为程序的进入点。

### 报错机制

当输入的程序出现词法、语法、语义错误时，编译器会适当报错。

### 代码优化

编译器将对代码进行适当优化。

### 其他问题

其他运行时问题基本上与java类似。

数组：计数器（考虑循环），堆，纯编译，不支持嵌套类