JFlat Reference Manual

"Java, but worse"

Project Manager:	**David Watkins**	**djw2146**
System Architect:	**Emily Chen**	**ec2805**
Language Guru:	**Phillip Schiffrin**	**pjs2186**
Tester & Verifier:	**Khaled Atef**	**kaa2168**

Contents

1. Introduction

JFlat is a general purpose, class-based, object-oriented programming language. The principal is simplicity, pulling many themes of the language from Java. JFlat is a high level language utilizes LLVM IR to abstract away hardware implementation of code. Utilizing the LLVM as a backend allows for automatic garbage collection of variables as well.
JFlat is a strongly typed programming language, meaning that at compile time the language will be type-checked, thus preventing runtime errors of type.
This language reference manual is organized as follows:

- Chapter 2 Describes types, values, and variables, subdivided into primitive types and reference types
- Chapter 3 Describes the lexical structure of JFlat, based on Java. The language is written in the ASCII character set
- Chapter 4 Describes the expressions and operators that are available to be used in the language
- Chapter 5 Describes different statements and how to invoke them
- Chapter 6 Describes the structure of a program and how to determine scope
- Chapter 7 Describes classes, how they are defined, fields of classes or their variables, and their methods
- Chapter 8 Discusses the different library classes provided with the compiler and their definitions

The syntax of the language is meant to be reminescent of Java, thereby allowing ease of use for the programmer.

2. Types

Primitive Data Types

int

The integer type stores the given value in 32 bits. You should use integer types for storing whole number values (and the char data type for storing characters). The integer type can hold values ranging from -2,147,483,648 to 2,147,483,647.
Syntactically correct use of int:

```
<scope> int funcName( <formal-opts> ) { <body> }
<scope> <type> funcName( <formal-opts> ) { int i; <body> }
class className {
        <scope> int i;
        <cbody>
}
```

float

The float type stores the given value in 64 bits. You should use float types to store fractional number values or whole number values that do not fit into the range provided by the integer type. The float type can hold values ranging from 1e-37 to 1e37. Since all values are represented in binary, certain floating point values must be approximated.

Syntactically correct use of float:

```
<scope> float funcName( <formal-opts> ) { <body> }
<scope> <type> funcName( <formal-opts> ) { float i; <body> }
class className {
        <scope> float i;
        <cbody>
}
```

void

The void type is used to indicate an empty return value from a method call. As it is assumed that every method will return a value, and that value must have a type, the type of a return which has no value is null. A void type cannot be used for a variable type, only function return types. An example would look like: "public void inc(a) a = a + 1; "

Syntactically correct use of void:

```
<scope> void funcName( <formal-opts> ) { <body> }
```

char

A character constant is a single character enclosed with single quotation marks, such as 'p'. The size of the char data type is 8 bits or 1 byte. Some characters cannot be represented using only one character.

Syntactically correct use of char:

```
<scope> char[] funcName( <formal-opts> ) { <body> }
<scope> <type> funcName( <formal-opts> ) { char i; <body> }
class className {
        <scope> char i;
        <cbody>
}
```

bool

The bool type is a binary indicator which can be set to either True or False. A bool can take one of two values, 'true' or 'false'. A bool could also be null.

Syntactically correct use of bool:

```
<scope> bool funcName( <formal-opts> ) { <body> }
<scope> <type> funcName( <formal-opts> ) { bool i; <body> }
class className {
        <scope> bool i;
        <cbody>
}
bool i = true;
bool i = false;
```

Non-Primitive Data Types

Arrays

An array is a data structure which lets you store one or more elements consecutively in memory. Elements in an array are indexed beginning at position zero, not one.

You declare an array by specifying its elements, name, and the number of elements it can store. An example is:

```
int myArray[2];
```

You can also initialize the elements in an array when you declare it as:

```
int myArray[2] = {3, 4};
```

If you initialize the values of an array during declaration, you must initialize every element in the array. Thus the following code is invalid:

```
int myArray[2] = {3}; (* invalid code *)
```

The lexical convention for initializing an array is:

```
<type T> arrayName[dim1, dim2, ..., dimN];
<type T> arrayName[dim1, dim2, ..., dimN] = [[[], [],...;
<type T> arrayName[,]; (* Defines an array with 2 dimensions of type T *)
```

You can access an element in an array by specifying the name of the array and the index of that element. You can use the accessed value for computation or you can change the value of the index as long as its type is the same type as the array. As an example, if you have an array of a = [1, 2, 3, 4], you can access the 3, which is the 3rd element of the array, with:

```
a[2]; (* 3 *)
a[2] = 5; (* change the value at the specified index to 5 *)
a[2]; (* 5 *)
```

The type of an array can be any primitive, including the array type. This means that you can declare an n-dimensional array, the members of which can be accessed by first indexing to the desired element of the first array, which is of type array, and then accessing into the desired element of the next array, and continuing n-1 times. For example, with a 2-dimensional array:

```
int a[2,3]; (* declaring a two element array,
                        each of which is a three element array of ints *)
```

JFlat will support arrays of arbitrary types as defined by the programmer. Accessing the length of an array may be done via:

```
int a[5];
int length = a.length;
```

Objects

See chapter 7 on classes to learn more about the syntax and usage of objects.

Casting

Casting is not supported in this language. There are interesting behaviors between ints and float defined in the section on operators that imitate casting, but there is no syntax to support casting between types directly.

3. Lexical Conventions

This chapter describes the lexical elements that make up JFlat source code. These elements are called tokens. There are five types of tokens: keywords, identifiers, constants, operators, and separators. White space, sometimes required to separate tokens, is also described in this chapter.

Identifiers

Identifiers are sequences of characters used for naming variables, functions and new data types. Valid identifier characters include ASCII letters, decimal digits, and the underscore character '_'. The first character of an identifier cannot be a digit.
An identifier cannot have the same spelling (character sequence) as a keyword, boolean literal, or the null literal, or a compile-time error occurs. Lowercase letters and uppercase letters are distinct, such that foo and Foo are two different identifiers.

```
ID = "['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z']|['0'-'9']|'\textunderscore')*"
```

Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. You cannot use them for any other purpose. JFlat recognizes the following keywords:

if	else	for	while	return
int	float	bool	char	void
null	true	false	class	constructor
public	private	extends	include	this

Literals

A literal is the source code representation of a value of a primitive type or the null type.

Integer Literals

An integer literal may be expressed in decimal (base 10). A positive integer is represented with either the single ASCII digit 0, representing the integer zero, or an ASCII digit from 1 to 9 optionally followed by one or more ASCII digits from 0 to 9. A negative integer is represented with the representation of a non-zero positive integer, prefixed with the negation operator, '-'.

```
INT = "['0'-'9']+ | '-' ['0'-'9']+"
```

Float Literals

A float literal has the following parts: a whole-number part, a decimal point (represented by an ASCII period character), and a fraction part. The whole number and fraction parts are defined by a single digit 0 or one

digit from 1-9 followed by more ASCII digits from 0 to 9. A negative float is represented with the standard float prepended with a negation operator, '-'.

```
FLOAT = "('-' ['0'-'9']+ | ['0'-'9']+) ['.'] ['0'-'9']+"
```

Boolean Literals

The boolean type has two values, represented by the boolean literals true and false, formed from ASCII letters.

```
BOOL = "true|false"
```

Character Literals

A character literal is always of type char, and is formed by an ascii character appearing between two single quotes. The following characters are represented with an escape sequence, which consists of a backslash and another character:

- '\\' - backslash
- '\"' - double-quote
- '\' - single-quote
- '\n' - newline
- '\r' - carriage return
- '\t' - tab character

It is a compile-time error for the character following the character literal to be other than a '.

```
CHAR = "\' ( ([' '-'!' '#'-'[' ']'-'~'] | '\\' [ '\\' '\"' 'n' 'r' 't' ]) )\' "
```

String Literals

A string literal is always of type char[] and is initialized with zero or more characters or escape sequences enclosed in double quotes. char[] x = "abcdef";

```
STRING = "\"( ([' '-'!' '#'-'[' ']'-'~'] | '\\' [ '\\' '\"' 'n' 'r' 't' ]) )*\""
```

Separators

A separator separates tokens. White space (see next section) is a separator, but it is not a token. The other separators are all single-character tokens themselves: ( ) [ ] ; , .

```
'('     { LPAREN }
')'     { RPAREN }
'{'     { LBRACE }
'}'     { RBRACE }
';'     { SEMI }
','     { COMMA }
'['     { LBRACKET }
']'     { RBRACKET }
'.'     { DOT }
```

Operators

The following operators are reserved lexical elements in the language. See the expression and operators section for more detail on their defined behavior.

```
+   -   *   /   =
==  !=  <   <=  >
>=
```

White Space

White space refers to one or more of the following characters:

- the ASCII SP character, also known as "space"
- the ASCII HT character, also known as "horizontal tab"
- the ASCII FF character, also known as "form feed"
- LineTerminator

White space is ignored, except when it is used to separate tokens. Aside from its use in separating tokens, it is optional. Hence, the following two snippets of source code are equivalent.

```
public int foo()
{
        print( "hello, world\n" );
        return 0;
}

public int foo(){print("hello, world\n"); return 0;}
```

```
WHITESPACE = "[' ' '\t' '\r' '\n']"
```

Comments

All the text from the ASCII characters (* to the ASCII characters *) is ignored. Multiline comments can be distinguished from code by preceding each line of the comment with a * similar to the following:

```
(* This is a long comment
* that spans multiple lines because
* there is a lot to say. *)
```

```
COMMENT = "(\* [.]* \*)"
```

4. Expressions and Operators

Expressions

An expression is composed of one of the following:

- A literal value (See Literals section of chapter 2)
- A reference to the current object via *this* (See chapter 7)
- An ID of a variable
- An operand followed by an operator followed by an operand
- The initialization of an object (See chapter 7)
- The access of an array
- an expression between ()

An arithmetic expression consists of at least one operand and zero or more operators. Operands are typed objects such as constants, variables, and method calls that return values. Here are some examples of expressions:

```
42;        (* Expression evaluates to int 42 *)
1 + 1;     (* Expression evaluates to int 2 *)
3.0 - 2.0; (* Expression evaluates to float 1.0 *)
```

Parentheses group subexpressions:

```
( 2 * ( 2 + 2 ) - ( 3 - 2) );  (* Evaluates to 7 *)
```

Method Calls as Expressions

A call to any method which returns a value is an expression.

```
print("Hello");  (* Evaluates to null *)
```

Object Initialization as Expressions

A call to a constructor of an object will evaluate to an instance of that object.

```
String("Hello");  (* Evaluates to String *)
```

Array Access as Expressions

Creating an array evaluates to a type ¡type¿[] with dimensions of what is passed

```
int a[2] = [1,2];
a[2];  (* Evaluates to 1 *)
```

```
Class.methodReturnsInt() + 3;  (* Assuming method returns value 4, the expressions evaluates to 7 *)
```

Operators

An operator specifies an operation to be performed on its operands. Operators may have one or two operands depending on the operator.

Assignment Operators

Assignment operators store values in variables. JFlat provides several variations of assignment operators.

The standard assignment operator "=" simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand cannot be a literal or constant value. Null assignments are valid as well.

```
int x = 10;
float y = 4.0 + 2.0;
int z = (2 * (3 + Class.methodReturnsInt() ));
int x = null;   (* Valid *)
3 = 10; (* Invalid! *)
```

Arithmetic Operators

JFlat provides operators for standard arithmetic operations: addition, subtraction, multiplication, and division, along with negation.

```
(* Addition. *)
int x = 5 + 3;
float y = 57.53 + 10.90;

(* Subtraction. *)
x = 5 - 3;
y = 57.53 - 10.90;

(* Multiplication. *)
x = 5 * 3;
y = 57.53 * 10.90;

(* Division. *)
x = 5 / 3; (* Integer division of positive values truncates towards zero, so 5/3 is 1 *)
y = 57.53 / 10.90;

(* Negation. *)
int x = -5;
float y = -3.1415;
```

Type designation for mixed types (ints and floats) occurs from left to right.

```
1.0 + 3;   (* Expression evaluates to float 4.0 *);
1 + 3.0;   (* Expression evaluates to int 4 *);
```

Conditional Operators

You use the comparison operators to determine how two operands relate to each other: are they equal to each other, is one larger than the other, is one smaller than the other, and so on. When you use any of the comparison operators, the result is either "true" or "false". The not-equal-to operator "!=" tests its two operands for inequality.
The equal-to operator "==" tests its two operands for equality. The result is a "true" boolean if the operands are equal, and "false" if the operands are not equal.

```
int x = 5;
int y = 5;
bool z = (x == y);  (* z evaluates to "true" *)
x = x + 1;
z = (x != y);       (* z evaluates to "true" *)
```

Comparing float values for exact equality or inequality can produce unexpected results. This is due to the underlying implementation of LLVM where floating point is approximated and not precise.
Beyond equality and inequality, there are operators you can use to test if one value is less than, greater than, less-than-or-equal-to, or greater-than-or-equal-to another value. Null conditional comparisons are allowed as well.

```
int w = 5;
int x = 5;
int y = 6;
bool z = false;
String d = String("Hello");

z = (x < y);     (* z evaluates to "true" *)
z = (w <= x);    (* z evaluates to "true" *)
z = (w > x);     (* z evaluates to "false" *)
z = (w >= x);    (* z evaluates to "true" *)
z = d == null;         (* z evaluates to "false" *)
z = d != null;   (* z evaluates to "true" *)
```

The ==, !=, >=, >, <, <= operators are all defined to operate between any two values both either being of int or float. The The ==, != are also designated to compare any two values in JFlat, and if they are not both of type float, will only return true if the memory address is identical. The comparison of objects is recommended by defining an equals method within the class.
It is important to note that conditional expressions can be chained together using the *and*, *or*, *not* operators, like so:

```
int w = 5;
int x = 5;
int y = 6;
bool z = ((x == w) and (true)) or (y > x); (* z evaluates to "true" *)
```

Conditional expressions like this will terminate once a condition has been met, therefore the $(y > x)$ expression will not be evaluated because the first half, $((x == w) and (true))$, evaluated to true.

Array Operators

Creating an array evaluates to a type <type>[] with dimensions of what is passed

```
int a[2] = {1,2};
int b[1,2,1] = [[[1], [2]]];
a[2];  (* Evaluates to 1 *)
```

Operator Precedence

When an expression contains multiple operators, such as x + y * Class.methodReturnsValue(), the operators are grouped based on rules of precedence. For instance, the meaning of that expression is to call the method with no arguments, multiply the result by y, then add that result to x.
The following is a list of types of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right unless stated otherwise.

- Method calls, array subscripting, and membership access operator expressions.
- Unary operators, including logical negation and unary negative.
- When several unary operators are consecutive, the later ones are nested within the earlier ones: not-x means not(-x).
- Multiplication, division.
- Addition and subtraction expressions.
- Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to
- expressions.
- Equal-to and not-equal-to expressions.
- Logical AND expressions.
- Logical OR expressions.
- All assignment expressions, including compound assignment. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.

5. Statements

A statement forms a complete unit of execution.

Expression Statements

An expression statement consists of an expression followed by a semicolon. The execution of such a statement causes the associated expression to be evaluated. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;):

```
(* Assignment expressions *)
aValue = 8933.234;
(* Method invocations *)
Game.updateScore(Player1, 5);
(* Object creation expressions *)
Bicycle myBike = new Bicycle();
```

Declaration Statements

A declaration statement declares a variable by specifying its data type and name.

```
double aValue;
```

In addition to the data type and name, a declaration statement can initialize the variable with a value.

```
double aValue = 8933.234;
```

Control Flow Statements

The statements inside source files are generally executed from top to bottom, in the order that they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else), the looping statements (for, while), and the branching statements (break, continue, return) supported by the JFlat programming language.

if-then, if-then-else

The 'if-then' statement tells the program to execute a certain section of code only if a particular test evaluates to true. The conditional expression that is evaluated is enclosed in balanced parentheses. The section of code that is conditionally executed is specified as a sequence of statements enclosed in balanced braces. If the conditional expression evaluates to false, control jumps to the end of the if-then statement.

```
if (condition) {
        <stmt>
}
```

```
if (not condition) {
        <stmt>
} (* if statement is skipped *)
```

The 'if-then-else' statement provides an alternate path of execution when "if" clause evaluates to false. This alternate path of execution is denoted by a sequence of statements enclosed in balanced braces, in the same format as the path of execution to take if the conditional evaluates to true, prefixed by the keyword "else".

```
if (condition) {
        <stmt>
} else {
        <stmt2>
} (* <stmt2> executed when not condition *)
```

Looping: for, while

The 'for' statement allows the programmer the iterate over a range of values. The 'for' statement has the following format:

```
for (initialization; termination; update) { <stmt> }
```

- The 'initialization' expression initializes the loop counter. It is executed once at the beginning of the 'for' statement
- When the 'termination' expression evaluates to false, the loop terminates.
- The 'update' expression is invoked after each iteration and can either increment or decrement the value of the loop counter.

The following example uses a 'for' statement to print the numbers from 1 to 10:

```
for (int loopCounter=1; loopCounter<11; loopCounter++) {
        print(loopCounter);
}
```

The 'while' statement executes a user-defined block of statements as long as a particular conditional expression evaluates to true. The syntax of a 'while' statement is:

```
while (expression) {
        <stmt>
}
```

The following example uses a 'while' statement to print the numbers from 1 to 10:

```
int loopCounter = 1;
while (loopCounter < 11) {
        print(loopCounter);
        loopCounter = loopCounter + 1;
}
```

Branching: break, continue, return

If a 'break' statement is included within either a 'for' or 'while' statement, then it terminates execution of the innermost looping statement it is nested within. All break statements have the same syntax:

```
break;
```

In the following example, the 'break' statement terminates execution of the inner 'while' statement and does not prevent the 'for' statement from executing its block of statements for all iterations of i from 1 to 10. This results in the the values of j from 100 to 110 being printed, in each of the 10 iterations of the 'for' loop.

```
for (int i=1; i<11; i++) {
        int j = 100;
        while (j<120) {
                if (j>110) {
                        break;
                }
                print(j);
                j = j + 1;
        }
}
```

In the following example, the 'break' statement terminates execution of the inner 'for' statement and does not prevent the 'while' statement from executing its block of statements for all iterations of i from 1 to 1000. This results in the the values of j from 100 to 110 being printed, in each of the 1000 iterations of the 'while' loop.

```
int i = 1;
while (i<1001) {
        for (int j=100; j<120; j++) {
                if (j>110) {
                        break;
                }
        }
        i = i + 1;
}
```

The continue statement skips the current iteration of a 'for' or 'while' statement, causing the flow of execution to skip to the end of the innermost loop's body and evaluate the conditional expression that controls the loop. The following example uses a 'continue' statement within a 'for' loop to print only the odd integers between 1 and 10. The code prints "hello" 1000 times and on each of the 1000 'while' loop iterations, prints the odd integers.

```
int counter = 1;
while (counter < 1001) {
        print("hello");
        for (int i=1; i<11; i++) {
                if (i - 2*(i/2) == 0) {
                        continue;
                } else {
                        print(i);
                }
        }
```

```
        counter = counter + 1;
}
```

The 'return' statement exits from the current method, and control flow returns to where the method was invoked. To return a value, simply put the value (or an expression that calculates the value) after the return keyword:

```
return count + 4;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, either no return statement is needed or the following 'return' statement is used:

```
return;
```

Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, BlockDemo, illustrates the use of blocks:

```
class BlockDemo {
        public void main(char[,] args) {
                bool condition = true;
                if (condition) { (* begin block 1 *)
                        print("Condition is true.");
                } (* end block one *)
                else { (* begin block 2 *)
                        print("Condition is false.");
                } (* end block 2 *)
        }
}
```

JFlat Functions

There are several reserved functions in JFlat that cannot be overridden and follow a particular syntax and return type.

File I/O

Manipulating files is an important aspect of any programming languages. Open files are denoted by a particular $intfd$; that can be used to read or write from a file. A file must be closed by the end of a program or else undefined behavior may occur.

int fopen(char[] filename, bool isWriteEnabled)

Accepts a filename and a flag to determine whether the file will be written to. If the file exists, it will be opened in append mode, otherwise a new file will be created. If it is in read mode, it will return a file descriptor as normal, or if the file doesn't exist will return '-1'. Likewise for write enabled, if there is an error it will return -1.

```
int fd = fopen("hello.txt", false);
```

bool fwrite(int fd, char[] values, int num, int offset)

Accepts an array of values to be written to a file, the number of characters it should write, and the offset into the value array it should write from. If there is an error, returns false, otherwise returns true.

```
bool success = fwrite(fd, "This should work", 4, 1); (* Writes "his " to a file *)
```

bool fread(int fd, char[] storage, int num)

Accepts an array to store values from the file that are to be read, and will read in num bytes. Returns true on success and false on error.

```
char a[100];
bool success = fread(fd, a, 20);
```

bool fclose(int fd)

Closes a file. Returns true on success, false on error.

```
bool success = fclose(fd);
```

Reading and Writing from Console

Reading and writing to the console is defined by two simple to use functions that cannot be overriden.

void print(char[] string)

Accepts a char array and prints the string to the console.

```
print("hello world");
```

void print(int num)

Accepts an int and prints the int to the console.

```
print(1);
```

void input(char[] buf)

Accepts a buf that will hold read bytes from the console. Then it will write those bytes to the array passed. Terminates when a user enters a newline or an EOF.

```
char a[100];
input(a);
```

6. Program Structure and Scope

Program structure and scope define what variables are accessible and where. When inside a class, there are many different cases of scope, however those are better defined in chapter 7.

Program Structure

A JFlat program may exist either within one source file or spread among multiple files which can be linked at compile-time. An example of such a linked file is the standard library, or *stdlib.jf*. When an include statement is executed at compile time, it will load in the files mentioned at the includes and insert the code at that location as if it were part of the head source file. Therefore at compilation, one only needs to compile with *jfcmaster.jf*. A program consists of zero or more include statements, followed by one or more class definitions. Only one class out of all classes may have a main method, defined with $publicvoidmain(char[,]args)$ which designates the entry point for a program to begin executing code. All JFlat files are expected to end with the file extension *.jf* and follow the following syntactic layout.

```
include(stdlib)
include(mylib)

class FOO {

        (* my code *)

}

class FOO {

        (* my code *)

        public static void main(char[,] args)

}
```

Scope

Scope refers to which variables, methods, and classes are available at any given time in the program. All classes are available to all other classes regardless of their relative position in a program or library. Variable scope falls into two categories: fields (instance variables) which are defined at the top of a class, and local variables, which are defined within a method. Fields can be public or private. If a field is public then it is accessible whenever an instance of that class is instantiated. For instance, if I have a class X, then class Y can be defined as follows:

```
class Y {
```

```
        public int num;

        constructor() {

                X myObj = X();
                this.num = myObj.number;
        }
}

class X {

        public int number;

}
```

In this example, class Y has one field which is an int. In its constructor, an instance of class X is declared, and a public field within that object is used to set the value for the given int. If a field is declared private, however, it can only be accessed by the methods in the same class. For example, if there is a class Y with a private field, the following is valid:

```
class Y {

        private int num;

        constructor() {

                this.num = 5;
        }

        private int getNum() {

                return this.num;
        }
}
```

However, if I have a class X, that class cannot access the private field within Y. The following is invalid:

```
class X {

        public int number;

        constructor() {

                Y myObj = Y();
                (* This code is invalid since num is a private field within Y *)
                this.number = myObj.num;
        }
}
```

Methods are also declared as public or private, and their accessibility is the same as fields. They must have a scope defined on them.

Local variables are variables that are declared inside of a method. Local variables are only accessible within the same method in which they are declared, and they may have the same name as fields within the same class since fields in a class are only accessible by calling the *this* keyword.

7. Classes

Classes are the constructs whereby a programmer defines their own types. All state changes in a JFlat program must happen in the context of changes in state maintained by an object that is an instance of a user-defined class.

Class definition

A class definition starts with the keyword 'class' followed by the class name and the class body, enclosed by a pair of curly braces. The body of a class declares one or more of each of the following: fields, methods, and constructors.
The members of a class type are all of the following:

- Members inherited from its ancestors (its direct superclass and its ancestors)
- Members declared in the body of the class, with the exception of constructors

Access modifiers

Class member declarations must include access modifiers but the class declaration itself does not. Field and method declarations must include one of the access modifiers 'public' or 'private'. Fields and methods with the access modifier 'public' can be accessed by methods defined in any class. Fields and methods with the access modifier 'private' can be accessed by methods defined either in the same class or in successor classes (classes derived directly from that class and their successors).

Fields

The only fields that can be declared are instance variables, which are freshly incarnated for each instance of the class. Field declarations have one of the following formats:

```
<access modifier> <type> <VariableDeclaratorId>;
(* Example *) private int myInstanceVariable;

<access modifier> <type> <VariableDeclaratorId> = <VariableInitializer>;
(* Example *) private int myInstanceVariable = 5;
```

All instance variables must be declared before methods and constructors.

Methods

A method declares executable code that can be invoked, passing a fixed number of values as arguments. The only methods that can be declared are the 'main' method and instance methods. Instance methods are invoked with respect to some particular object that is an instance of a class type.

Method declarations constitute a method header followed by a method body. The method header has the following format:

```
<access modifier> <return type> <method name> <comma-separated list of parameters>
(* Example *) public double amountPaid(double wage, int duration)
```

The method body contains one or more statements enclosed between the ASCII characters '{' and '}'. If the type of the return value is not void, then the method body must include a return statement.

One and only one of the classes to be compiled must contain a definition for a method named "main" that executes when the program runs. The 'main' method is not callable as an instance method. The 'main' method must have a void return type and accept a single parameter of type String[]. Hence, its signature must be:

```
public void main (char[,] args)
```

If either zero or more than one class contains a definition for a method with the signature above, this results in a compile-time error.

Methods can be overloaded: If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not equivalent, then the method name is said to be overloaded. There can be multiple methods with the same name defined for a class, as long as each has a different number and/or type of parameters. The 'main' method can never be overloaded because it has one and only one accepted signature.

The name of a method may not be the same as the name of the class it is defined in, since the name of a class is reserved for constructors.

Constructors

Constructors are similar to methods but cannot be invoked as an instance method; they are used to initialize new class instances. A constructor has no return type and its formal parameters are identical in syntax and semantics to those of a method. A constructor is definition has the following format:

```
constructor <comma-separated list of parameters> {<list of statements>}
(* Example *) constructor (int a, char[] b) {...}
```

Unlike fields and methods, access to constructors is not governed by access modifiers. Constructors are accessible from any class.

Constructor declarations are never inherited and therefore are not subject to overriding.

If no constructors are defined, the compiler defines a default constructor. Like methods, they may be overloaded. It is a compile-time error to declare two constructors with equivalent signatures in a class.

When the programmer declares an instance of the class, either a user-defined constructor or the default constructor is automatically called.

```
class Foo {
        constructor (int x) {...}
        ...
}
class Bar {
        public void main (String[] args) {int x = 5; Foo myFooObj = Foo(x);}
}
```

Referencing instances

The keyword 'this' is used in the body of method and constructor declarations to reference the instance of the object that the method or constructor will bind to at runtime.

Inheritance

The members of a class include both declared and inherited members. A class inherits all methods of its direct superclass and superclasses of that class.

Overriding

Newly declared methods can override methods declared in any ancestor class. An instance method m1, declared in class C, overrides another instance method m2, declared in class A iff both of the following are true:

- C is a subclass of A
- The signature of m1 is identical to the signature of m2

8. Standard Library Classes

Accessing the Standard Library

To access the standard library, enter 'include(stdlib);' at the top of the source code. As noted earlier, including a file can only occur once, so do not include a class a second time.

String

JFlat provides certain standard library classes to assist the user with string manipulation and file I/O.

Fields

String has no public fields

Constructors

String(char[] a)

Accepts a char array, such as a string literal or a char array, and creates a String object

Methods

public bool contains(char[] chrs)

Returns true if and only if this string contains the specified sequence of char values.

public int indexOf(int ch)

Returns the index within this string of the first occurrence of the specified character.

public bool isEmpty()

Returns true if and only if length() is 0.

public int length()

Returns the length of the string.

public char[] toCharArray()

Returns the char array of this string.

File

The File class constructor takes one argument which is a char[] that points to a file on which the user wishes to operate. The constructor stores the given path in a field and then calls open() on the given path and, if successful, sets the objects file descriptor field to the return of open(). If open() fails, the program exits with error.

Fields

File has no public fields

Constructors

File(char[] path, bool isWriteEnabled)

Accepts a char array to open a file on, then creates a file object with the file descriptor. isWriteEnabled is a parameter that is used to determine whether the file can be written to or just read from.

Methods

public char[] read(int num)

Reads num bytes from the open file and returns the bytes in a char array.

public void close()

Closes the open file. On error, the program exits with error.

public void write(char[] arr)

Writes the contents of the char[] array to the file

References

[1] http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html *The GNU C Reference Manual.*. N.p., n.d. Web. 26 Oct. 2015.

[2] https://docs.oracle.com/javase/specs/jls/se8/html/index.html *The Java Language Specification.* . N.p., n.d. Web. 26 Oct. 2015.

[3] Edwards, Stephen. *"Programming Language and Translators."* Lecture.

[4] "Control Flow Statements." *The Java Tutorials Learning the Java Language Language Basics* N.p., n.d. Web. 26 Oct. 2015.