Dice Reference Manual 

"Java, but worse"

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

Contents

1. Introduction

Dice is a general purpose, object-oriented programming language. The principal is simplicity, pulling many themes of the language from Java. Dice is a high level language that 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.
Dice 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 Dice, 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

There are two kinds of types in the Dice programming language: primitive types and reference types There are, correspondingly, two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive values and reference values.

```
Type:
    PrimitiveType
    ReferenceType
```

There is also a special null type, the type of the expression *null*, which has no name. Because the null type has no name, it is impossible to declare a variable of the null type. The null reference is the only possible value of an expression of null type. The null reference can always undergo a widening reference conversion to any reference type. In practice, the programmer can ignore the null type and just pretend that *null* is merely a special literal that can be of any reference type.

Primitive Types and Values

A primitive type is predefined by the Dice programming language and named by its reserved keyword.

```
PrimitiveType:
    NumericType
    bool
NumericType:
    IntegralType
    float
IntegralType: one of
    int char
```

int

A value of type *int* is stored as a 32-bit signed two's-complement integer. The *int* type can hold values ranging from -2,147,483,648 to 2,147,483,647, inclusive.

float

The float type stores the given value in 64 bits. 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.

char

The *char* data type is a 8-bit ASCII character. A *char* value maps to an integral ASCII code. The decimal values 0 through 31, and 127, represent non-printable control characters. All other characters can be printed by the computer, i.e. displayed on the screen or printed on printers, and are called printable characters.

The character 'A' has the code value of 65, 'B' has the value 66, and so on. The ASCII values of letters 'A' through 'Z' are in a contiguous increasing numeric sequence. The values of the lower case letters 'a' through 'z' are also in a contiguous increasing sequence starting at the code value 97. Similarly, the digit symbol characters '0' through '9' are also in an increasing contiguous sequence starting at the code value 48.

bool

A variable of type *bool* can take one of two values, *true* or $false$. A bool could also be *null*.

Non-Primitive Types

Non-primitive types include arrays and classes.

Arrays

An array stores one or more values of the same type contiguously in memory. The type of an array can be any primitive or an array type. This allows the creation of an n-dimensional array, the members of which can be accessed by first indexing to the desired element of the outermost array, which is of type *array*, and then accessing into the desired element of the immediately nested array, and continuing n-1 times.

Classes

Classes are user-defined types. See chapter 7 to learn about the 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 Dice source code. These elements are called tokens. There are six types of tokens: identifiers, keywords, literals, separators, and operators. 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 must be alphabetic.
An identifier cannot have the same spelling (character sequence) as a keyword, boolean or null literal, 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. Dice recognizes the following keywords:

if	else	for	while	
break	continue	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 is expressed in decimal (base 10). It 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.

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

Float Literals

A float literal has the following parts: an integer part, a decimal point (represented by an ASCII period character), and a fraction part. The integer 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.

```
FLOAT = "['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 single-quote character '.

```
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\n";
```

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

Separators

A separator separates tokens. White space 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

The characters (∗ introduce a comment, which terminates with the characters ∗). 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

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein.

Primary Expressions

Primary expressions involving . , subscripting, and function calls group left to right.

Identifier

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration.

Literal

Any of the literal types discussed in Chapter 3 is a primary expression, which evaluates to the type of the literal.

(expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

primary-expression [expression]

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. The primary expression has type *array* of . . . and the type of the result is . . . . The type of the subscript expression must be a type that is convertible to an integral type, or a compile-time error occurs.

primary-expression (expression-list-opt)

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The result of the function call is the function's return type. Recursive calls to any function are permissible.

primary-lvalue . member-of-structure

An lvalue expression followed by a dot followed by the name of a class member is a primary expression. The object referred to by the lvalue is assumed to be an instance of the class defining the class member. The given lvalue can be an instance of any user-defined class.

Unary operators

Expressions with unary operators group right-to-left.

- expression

The result is the negative of the expression, and has the same type. The type of the expression must be *char*, *int*, or $float$.

not expression

The result of the logical negation operator *not* is *true* if the value of the expression is $false$, $false$ if the value of the expression is *true*. The type of the result is *bool*. This operator is applicable only to operands that evaluate to *bool*.

Multiplicative operators

The multiplicative operators * and / group left-to-right.

expression * expression

The binary * operator indicates multiplication. Operands of *int*, $float$, and *char* types are allowed. If both operands are of type ..., the result is type .... If the operands are of two different types of the ones listed above, the result is the type of the left-most operand.

expression / expression

The binary / operator indicates division. The same type considerations as for multiplication apply.

Additive operators

The additive operators + and - group left-to-right.

expression + expression

The value of the result is the sum of the expressions. The same type considerations as for multiplication apply.

expression - expression

The value of the result is the difference of the expressions. The same type considerations as for multiplication apply.

Relational operators

The relational operators group left-to-right.

expression < expression

expression > expression

expression <= expression

expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield *true* is the specified relation is true and $false$ otherwise. The same type considerations as for multiplication apply.

Equality operators

expression == expression

expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence.

Logical operators

expression and expression

Both operands must evaluate to a value of type *bool*. The *and* operator returns *true* if both its operands evaluate to *true*, $false$ otherwise. The second expression is not evaluated if the first evaluates to $false$.

expression or expression

Both operands must evaluate to a value of type *bool*. The *or* operator returns *true* if either of its operands evaluate to *true*, and $false$ otherwise. The second operand is not evaluated if the value of the first operand evaluates to *true*.

Assignment operators

lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. Both operands must have the same type.

5. Statements

A statement forms a complete unit of execution.

Include Statement

If a .dice file contains a statement of the following form:

```
include(mylib)
```

then all classes defined in *mylib* are available to be used in definitions of classes in the .dice file in which the include statement appears.

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 = Bicycle();
```

Declaration Statements

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

```
float aValue;
```

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 Dice 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:

```
int loopCounter;
for (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;
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.

```
int i;
int j;
for (i=1; i<11; i++) {
        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;
int j;

i = 1;
while (i<1001) {
        for (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 i;
int counter;
counter = 1;
while (counter < 1001) {
        print("hello");
```

```
        for (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;
                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 *)
        }
}
```

Dice Functions

There are several reserved functions in Dice 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;
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;
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;
bool success;

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

bool fclose(int fd)

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

```
bool success;
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;
a = char[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 Dice 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.dice*. 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 *dicecmaster.dice*. If an included module defines a class that has the same name as one of the classes defined in the including module, then the compiler throws an error. The compiler does not resolve recursive includes; if $foo.dice$ includes *bar.dice* and *bar.dice* includes $foo.dice$, the compiler throws an error.
A program consists of zero or more include statements, followed by one or more class definitions. Each class defined in a module must have a distinct name. Only one class out of all classes may have a main method, defined with *public void main(char[][] args)* which designates the entry point for a program to begin executing code. All Dice files are expected to end with the file extension *.dice* and follow the following syntactic layout.

```
include(stdlib)
include(mylib)

class FOO {

        (* my code *)

}

class BAR {

        (* my code *)

        public 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;
                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;
                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 Dice 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 (see identifiers in chapter 2) and the class body. The class body, enclosed by a pair of curly braces, 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; there is no notion of a private class in Dice. 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 the following format:

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

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, enclosed between the ASCII characters '{' and '}', zero or more variable declarations followed by zero or more statements. 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 char[][]. 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. If two methods in the same class have the same signature, the compiler throws an error.

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 definition has the following format:

```
constructor (<comma-separated formal arguments>) {
        <list of variable declarations>
        <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 (char[][] args) {
                int x;
```

```
                Foo myFooObj;
                x = 5;
                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 members of its direct superclass and superclasses of that class. To define a class Y that inherits members of an existing class named "X" and all superclasses of X, use the keyword *extends* when defining Y.

```
class Y extends X {...}
```

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

Dice 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.