VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PRINCIPLES OF PROGRAMMING LANGUAGES - CO3005

D96 SPECIFICATION

Version 1.0

HO CHI MINH CITY, 01/2022

D96'S SPECIFICATION

Version 1.0

1 Introduction

D96, is a mini object-oriented programming language with type inference, which is designed primarily for students practising implementing a simple compiler for a simple object-oriented language.

Despite its simplicity, D96 includes most important features of an object-oriented language such as encapsulation, information hiding, class hierarchy, inheritance, and polymorphism.

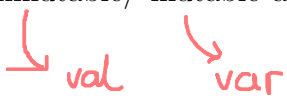
2 Program structure

As its simplicity, a D96 compiler does not support to compile many files so a D96 program is written just in one file only. A D96 program consists of many class declarations (subsection 2.1). The entry of a D96 program is a static special method, whose name is **main** without any parameter and whose return type is **main** in the class named **Program**. So, only such special method in the **Program** class is the entry of the program while the others will be considered as a common method.

2.1 Class declaration

Each class declaration starts with keyword **class** and then an identifier, which is the class name, and ends with a nullable list of members enclosed by a pair of curly parentheses. Between the class name and the list of members, there may be an optional colon (**:**) followed by an identifier which is the superclass name.

Every member in a class is named by the rule of identifier (subsection 3.3). It can be static if its name is a dollar identifier, or instance. A member of a class can be an attribute or a method. There are 2 kinds of attribute: mutable and immutable. An immutable/ mutable attribute is preceded by keyword **val**/ **var**.




```
class Shape {
    val $numOfShape: Int = 0;
    val immutableAttribute: Int = 0;
    var length, width: Int;

    $getNumOfShape() {
        return $numOfShape;
    }
```

```
}

class Rectangle: Shape {
    getArea() {
        return self.length * self.width;
    }
}

class Program {
    main() {
        Out.printInt(Shape::$numOfShape);
    }
}
```

2.2 Attribute declaration

An attribute starts with the keyword **val**/ **var** (for the mutability), followed by a non-nullable comma-separated list of attribute names. It continues with the compulsory part of type declaration with a colon (**:**) and a type name, respectively. Optionally, the next part of this declaration is a value initialization which is an equal (=) followed by a non-nullable comma-separated list of expressions (with the equivalent number of attributes). Finally, the end of this declaration is a semicolon (**;**).

For example:

```
val My1stCons, My2ndCons: Int = 1 + 5, 2;
var $x, $y : Int = 0, 0;
```

2.3 Method declaration

Each method declaration has the form:

```
<identifier> (<list of parameters>) <block statement>
```

The method has the name `<identifier>` and may be static if its name follows the dollar identifier. The `<list of parameters>` is a nullable semicolon-separated list of parameter declaration. Each parameter declaration has the form: `<identifier-list>: <type>`.

The `<block statement>` will be described in Section 6.10. In a class, the name of a method is unique that means there are not two methods with the same name allowed in a class. There are two special instance method in a class is **constructor** and **destructor** method:

- **Constructor** is a method whose name is `constructor` and it returns nothing in the body.

```
constructor (<list of parameters>) <block statement>
```

- **Destructor** is a method whose name is **destructor**. It has an empty list of parameters and also no return statement is in the body.

```
destructor () <block statement>
```

3 Lexical structure

3.1 Characters set

A D96 program is a sequence of characters from the ASCII characters set. Blank (' '), tab ('\t'), backspace (('\b')), form feed (i.e., the ASCII FF - '\f'), carriage return (i.e., the ASCII CR – '\r') and newline (i.e., the ASCII LF – '\n') are whitespace characters. The '\n' is used as newline character in D96.

This definition of lines can be used to determine the line numbers produced by a D96 compiler.

3.2 Program comment

In D96 programming language, there is only one type of comment: block comment. A block comment starts with `##` and ignores all characters (except EOF) until it reaches the nearest `##`. All characters in a block comment will be ignored.

For example:

```
## This is a
   multi-line
  comment.
##
```

3.3 Identifiers

Identifiers are used to name variables, constants, classes, methods and parameters. Identifiers begin with a letter (A-Z or a-z) or underscore (_), and may contain letters, underscores, and digits (0-9). D96 is case-sensitive, therefore the following identifiers are distinct: WriteLn, writeln, and WRITELN.

Dollar identifiers are special which are used for static members in a class. It begins with a dollar sign ($) and continues with a non-empty sequence of letters, underscores and digits.

3.4 Keywords

Keywords must begin with a lowercase letter (A-Z). The following keywords are allowed in D96:

`Break`	`Continue`	`If`	`Elseif`	`Else`
`Foreach`	`True`	`False`	`Array Int`	`Float`
`Boolean`	`String`			
`Null`				

3.5 Operators

The following is a list of valid operators:

`+`	`-`	`*`	`/`	`%`
`!`	`&&`	`||`	`==`	`=`
`!=`	`>`	`<=`	`>`	`>=`
`==.`	`+.`			

The meaning of those operators will be explained in the following sections.

3.6 Seperators

The following characters are the separators:

```
( ) [ ] . , ; { }
```

3.7 Literals

A literal is a source representation of a value of a integer, float, boolean, string, one of three types of array.

1. **Integer**: Integer can be specified in decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2) notation:
    - To use octal notation, precede the number with a 0 (zero).
    - To use hexadecimal notation precede the number with 0x or 0X.
    - To use binary notation precede the number with 0b or 0B.

Integer literals may contain underscores (_) between digits, for better readability of literals in decimal. These underscores are removed by D96's scanner.

For example:

```
1234    0123    0x1A        0b11111111      1_234_567
```

2. **Float**: A floating number consists of three components: integer, decimal and exponent part, respectively. Exponent part can be omitted if there exists both of remaining components. Otherwise, only one of them can be absent for this representation.

   - Integer part is just like an integer literal but used in only demical form.
   - Decimal part starts with an point (.) following by a representation like integer part or an empty.
   - Exponent part begins with a charater `e` or `E` and then an optinal sign (`-`, `+`). It finalized with an integer-part-like form.

   For instance:

   ```
   1.234   1.2e3   7E-10   1_234.567
   ```

3. **Boolean**: A **boolean** literal is either `True` or `False`, formed from ASCII letters.

4. **String**: A **string** literal includes zero or more characters enclosed by double quotes ("). Use escape sequences (listed below) to represent special characters within a string. Remember that the quotes are not part of the string. It is a compile-time error for a new line or EOF character to appear after the opening (") and before the closing matching (").

   All the supported escape sequences are as follows:

\b	backspace
\f	form feed
\r	carriage return
\n	newline
\t	horizontal tab
\'	single quote
\\	backslash

   Cần xem lại

   For a double quote (") inside a string, a single quote (') must be written before it: '" double quote

   For example:

   "This is a string containing tab \t"

   "He asked me: '"Where is John?'""

5. **Indexed Array**: An **indexed array** literal is a comma-separated list of literals (with an array) enclosed in '(' and ')' and started with keyword **Array**. The literal elements are in the same type.

   For example, `Array(1, 5, 7, 12)` or `Array("Kangxi", "Yongzheng", "Qianlong")`

6. **Multi-dimensional arrays** are such type of arrays which stores an another array at each index instead of single element. In other words, define multi-dimensional arrays as array of arrays.

   For example:

```
Array (
    Array("Volvo", "22", "18"),
    Array("Saab", "5", "2"),
    Array("Land Rover", "17", "15")
)
```

4 Type and Value

In D96, types limit the values that a variable can hold (e.g., an identifier x whose type is Int cannot hold value true...), the values that an expression can produce, and the operations supported on those values (e.g., we can not apply operation + in two boolean values...).

4.1 Primitive types

4.1.1 Boolean type

The keyword `Boolean` denotes a boolean type. Each value of type boolean can be either `True` or `False`.

`If` work with boolean expressions.

The operands of the following operators are in boolean type:

```
!       &&      ||      ==      !=
```

4.1.2 Integer type

The keyword `Int` is used to represent an integer type. A value of type integer may be positive or negative. Only these operators can act on number values:

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

4.1.3 Float type

The keyword `Float` is used to represent an float type. A value of type float may be positive or negative. Only these operators can act on number values:

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

4.1.4 String type

The operands of the following operators are in string type:

```
+.      ==.
```

4.2 Array type

An array type declaration is in the form of: `Array[<element_type>, <size>]`

trong đó:

- `<element_type>` is the element type of an array. It should be primitive type or, of course, array type.
- `<size>` is the length of array and it is required that there must be an integer literal. The lower bound is always 1.

  For example, `var a: Array[Int, 5];` indicates a five-element array: `a[1], a[2], a[3], a[4], a[5]`.

4.3 Class type

A class declaration defines a class type which is used as a new type in the program. `Null` is the value of an uninitialized variable in class type. An object of type `X` is created by expression `new X()`.

5 Expressions

Expressions are constructs which are made up of operators and operands. Expressions work with existing data and return new data.

In D96, there exist two types of operations: unary and binary. Unary operations work with one operand and binary operations work with two operands. The operands may be constants,

variables, data returned by another operator, or data returned by a function call. The operators can be grouped according to the types they operate on. There are five groups of operators: arithmetic, boolean, relational, index and key.

5.1 Arithmetic operators

Standard arithmetic operators are listed below.

Operator	Operation	Operand's Type
-	Number sign negation	Int/Float
+	Number Addition	Int/Float
-	Number Subtraction	Int/Float
*	Number Multiplication	Int/Float
%	Number Remainder	Int/Float

5.2 Boolean operators

Boolean operators include logical **NOT**, logical **AND** and logical **OR**. The operation of each is summarized below:

Operator	Operation	Operand's Type
!	Negation	Boolean
&&	Conjunction	Boolean
||	Disjunction	Boolean
==.	Compare two same strings	String

5.3 String operators

Standard string operators are listed below.

Operator	Operation	Operand's Type
+.	String concatenation	string

5.4 Relational operators

Relational operators perform arithmetic and literal comparisons. All relational operations result in a boolean type. Relational operators include:

Operator	Operation	Operand's Type
==	Equal	Int/Boolean
!=	Not equal	Int/Boolean
<	Less than	Int/Float
>	Greater than	Int/Float
<=	Less than or equal	Int/Float
>=	Greater than or equal	Int/Float

5.5 Index operators

An **index operator** is used to reference or extract selected elements of an array. It must take the following form:

```
element_expression -> expression index_operators
index_operators -> [ expression ]
                 | [ expression ] index_operators
```

5.6 Member access

1. An **instance attribute access** may be in the form:

   ```
   <expression>.<identifier>
   ```

   where `<expression>` is an expression that returns an object of a class and `<identifier>` is an attribute of the class.

2. A **static attribute access** may be in the form:

   ```
   <identifier>::<identifier>
   ```

   where the first `<identifier>` is a class name, and the second `<identifier>` is a static attribute of the class.

3. An **instance method invocation** may be in the form:

   ```
   <expression>.<identifier>(<list of expressions>)
   ```

   where `<expression>` is an expression that returns an object of a class and `<identifier>` is a method name. The type of the first `<expression>` must be a class type. The `<list of expressions>` is the comma-separated list of arguments, which are expressions. The type of the invocation is the return type of the invoked method.

4. A **static method invocation** may be in the form:

```
<identifier>::<identifier>(<list of expressions>)
```

where the first `<identifier>` is a class name and `<identifier>` is a static method name of the class. The others are the same as those in instance invocation.

5.7 Object creation

An object of a class type is only created by expression:

```
new <identifier>(<list of expressions>)
```

The `<identifier>` must be in a class type. The `<list of expressions>` is the comma-separated list of arguments. The list may be empty when the constructor of the class has no parameter.

5.8 Self

The keyword `self` expresses the current object of the enclosing class. The type of `self` is the class type of the enclosing class.

5.9 Operator precedence and associativity

The order of precedence for operators is listed from high to low:

Operator Type	Operator	Arity	Position	Association
Object creation	`new`	Unary	Prefix	Right
Member access	`.`, `::`	Binary	Infix	Left
Index operator	[,]	Unary	Postfix	Left
Sign	-	Unary	Prefix	Right
Logical	!	Unary	Prefix	Right
Multiplying	*, /, %	Binary	Infix	Left
Adding	+, -	Binary	Infix	Left
Logical	&&, ||	Binary	Infix	Left
Relational	==, !=, <, >, <=, >=	Binary	Infix	None
String	+., ==.	Binary	Infix	None

The expression in parentheses has highest precedence so the parentheses are used to change the precedence of operators.

5.10 Type coercions

In D96, mixed-mode expressions are permitted. Mixed-mode expressions are those whose operands have different types.

The operands of the following operators:

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

can have either type integer or float. If one operand is float, the compiler will implicitly convert the other to float. Therefore, if at least one of the operands of the above binary operators is of type float, then the operation is a floating-point operation. If both operands are of type integer, then the operation is an integral operation.

Assignment coercions occur when the type of a variable (the left side) differs from that of the expression assigned to it (the right side). The type of the right side will be converted to the type of the left side.

The following coercions are permitted:

- If the type of the variable is integer, the expression must be of the type integer.
- If the type of the variable is float, the expression must have either the type integer or float.
- If the type of the variable is boolean, the expression must be of the type boolean.

  Since an argument of a method call is an expression, type coercions also take place when arguments are passed to methods.

Note that, as other object-oriented languages, an expression in a subtype can be assigned to a variable in a superclass type without type coercion.

5.11 Evaluation orders

D96 requires the left-hand operand of a binary operator must be evaluated first before any part of the right-hand operand is evaluated. Similarly, in a function call, the actual parameters must be evaluated from left to right.

Every operands of an operator must be evaluated before any part of the operation itself is performed. The two exceptions are the logical operators && and ||, which are still evaluated from left to right, but it is guaranteed that evaluation will stop as soon as the truth or falsehood is known. This is known as the **short-circuit evaluation**. We will discuss this later in detail (code generation step).

6 Statements

A statement indicates the action a program performs. There are many kinds of statements, as described as follows:

6.1 Variable and Constant Declaration Statement

Variable and constant declaration statement should be the same as the attribute in a class. Variable should start with the keyword `var` while constant should start with the keyword `val`. However, the static property of attribute cannot be applied to them so its name should not follow the dollar identifier rule.

6.2 Assignment Statement

An assignment statement assigns a value to a left hand side which can be a scalar variable, an index expression. An assignment takes the following form:

```
lhs = expression;
```

where the value returned by the `expression` is stored in the the left hand side `lhs`.

6.3 If statement

The **if statement** conditionally executes one of some lists of statements based on the value of some boolean expressions. The if statement has the following forms:

```
If (<expression>) <block statement>
Elseif (<expression>) <block statement>
Elseif (<expression>) <block statement>
Elseif (<expression>) <block statement>
...
Else <block statement>
```

where the first `<expression>` evaluates to a boolean value. If the `<expression>` results in true then the statement-list following the reserved word Then is executed.

If `<expression>` evaluates to false, the `<expression>` after `Elseif`, if any, will be calculated. The corresponding `<block statement>` is executed if the value of the expression is true.

If all previous expressions return false then the statement-list following `else`, if any, is executed. If no `else` clause exists and expression is false then the if statement is passed over.

The `<block statement>` includes zero or many statements.

There are zero or many `Elseif` parts while there is zero or one `Else` part.

6.4 For/In statement

In general, **For/In statement** allows repetitive execution of <**block statement**>. For statement executes a loop for a predetermined number of iterations. For statements take the following form:

```
foreach (<scalar variable> In <expr1> .. <expr2> [By <expr3>]?)
    <block statement>
```

First, `<expr1>` will be evaluated and assigned to <scalar variable>. Then D96 calculates `<expr2>`. The value of `<expr3>` is evaluated next if it exists, otherwise, it can be considers as 1. The type of three expression `<expr1>`, `<expr2>`, `<expr3>` must be in integer. <scalar variable> will be incremented by `<expr3>` if the value of `<expr1>` is less than or equal to the value of `<expr2>`, otherwise it is a a decrement repetition. The process continues until the <scalar variable> hits the value of <expression2>. If <scalar variable> is greater/ smaller than <expression2>, the <statement> will be skipped (i.e., the statement next to this for loop will be executed).

```
For (i In 1 .. 100 By 2) {
    Out.printInt(i);
}

For (x In 5 .. 2) {
    Out.printInt(arr[x]);
}
```

6.5 Break statement

Using the **break statement**, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. It must reside in a loop. Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A break statement has the following form:

```
break;
```

6.6 Continue statement

The **continue statement** causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. It must reside in a loop . Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A continue statement has the following form:

```
continue;
```

6.7 Return statement

A **return statement** aims at transferring control and data to the caller of the function that contains it. The return statement starts with keyword `return` which is optionally followed by an expression and ends with a semi-colon.

A return statement must appear within a function.

6.8 Method Invocation statement

A **method invocation statement** is an instance/static method invocation, that was described in subsection 5.6, with a semicolon at the end.

For example:

```
Shape::$getNumOfShape();
```

6.9 Block statement

A **block statement** begins by the left parenthesis `{` and ends up with the right parenthesis `}`. Between the two parentheses, there may be a nullable list of statements.

For example:

```
{
    var r, s: Int;
    r = 2.0;
    var a, b: Array[Int, 5];
    s = r * r * self.myPI;
    a[0] = s;
}
```

7 Scope

There are 4 levels of scope: global, class, method and block.

7.1 Global scope

All class names, static attributes and method names have global scope. A class name or a static attribute is visible everywhere and a method can be invoked everywhere, too.

7.2 Class scope

All instance attributes of a class have class scope, i.e., they are visible in the code of all methods of the class and its subclasses.

7.3 Method scope

All parameters/variables declared in the body block have the method scope. They are visible from the places where they are declared to the end of the enclosing method.

7.4 Block scope

All variables declared in a block have the block scope, i.e., they are visible from the place they are declared to the end of the block.

8 Change log