2015

Epsilon Language

DRAFT 2015

THE EPSILON PROJECT

UNIVERSITY OF BATANGAS | College of Information and Communications Technology

The Epsilon Language

2015 Draft Manual

Researchers:

Rexwyn Nohay

Anjanette Briones

Trisya Jayne Ibon

University of Batangas

College of Information and Communications Technology

CONTENTS

1 INTRODUCTION

The Epsilon Project is a research project developed by a number of researchers in the University of Batangas. The goal of this project is to enable programmers to build software using a language that has most of the features of C, the object-oriented concept of C++, and the safety and ease of use of BASIC. This language is called Epsilon, an object-oriented language inspired by C, C++ and BASIC. The version of the language described in this manual, the Epsilon 2015, is experimental and limited and the writers of these language encourages other researchers to enhance its features and implementations.

The official repository of the project is managed using Git and is hosted in GitHub. The project site is https://github.com/rexwynnohay/Epsilon-Project/

1.1 LEXICAL CONVENTIONS

Epsilon programs are stored in files, which are called translation units. Epsilon doesn't support separate compilation, but it does support inclusion of library files by using preprocessors.

There are six classes of tokens; identifiers, keywords, constants, string literals and separators. Identifiers are user-defined tokens used for naming different kinds of entities in the program. Keywords are tokens that cannot be used as identifiers because they have semantic value of their own. Keywords and identifiers in Epsilon are *case-insensitive*. Identifiers (class names, function names, object names, variable names, data structure names, symbolic constants) should start with a letter followed by zero or more alphanumeric characters including underscores (_). The length of an identifiers can be up to 31 characters. This restriction is placed to enforced good programming practice (Long identifier names are hard to remember).

Constants are the literal values of a program such as integers and characters. String literals are collections of character constants terminated by null. Separators are ignored characters that separates other tokens. Whitespaces are considered separators of tokens while the newline character separates statements. All whitespaces (space, tab and newline) are all ignored by the compiler.

Comments in Epsilon are like C++ comments. Multi-line comments begins with `/*` and ends with `*/`. Single-line comments begins with `//` and ends in the end of the line.

[List keywords here]

1.2 MEANING OF IDENTIFIERS

As defined before, an identifier, also called name, may refer to many things: functions, variables, class, objects, data structures, symbolic constants, enumeration constants or aliases. A variable is a storage location that has two attributes: storage class and its type. The storage class determines the lifetime of the storage the associated with the identified variable while the type determines the meaning of the values found in the variable. A name also has a scope and a linkage to a storage location which other names may also refer.

Epsilon has two storage classes, automatic and static. Automatic variables are local to a block and are destroyed in block termination. Static variables on the other hand retains their value after block termination.

There are two kinds of data types in Epsilon; fundamental and derived. Fundamental types are discussed in Section 2.2. Derived types are constructed from fundamental types in the following ways:

- Arrays
- Functions
- Objects
- Pointers
- Records

1.3 SCOPE

Epsilon defines 2 levels of scope:

1. **Block scope** – objects declared in this level have scopes from the point of their creation until the end of the block they are created in. Example of this are variables declared inside a function or in the parameter list, inside control flow statements like `if..then..end` if.
2. **Program scope** – objects declared in this level have scopes from the point of their creation until the end of the program. These variables are often called global variables.

At any part of the program, an object can either be local or nonlocal. An active object is said to be local in a particular region if it is declared there while an active object is said to be nonlocal in a particular region if it is not declared there. In case a local and nonlocal variables with the same name is present in the same region, the compiler will hide the nonlocal variable.

2 DATA TYPES AND VARIABLES

Data types or types is a classification that determines an object's size in memory and the types of operations that can be used with it. A variable is a storage location associated with a symbolic name called *ID* which contains a value.

2.1 VARIABLE DECLARATION

Before a variable can be used, it must be declared first. Declaration is the process of announcing the existence of an object, therefore the compiler will start creating it by binding an ID to it and allocating a storage space for it. The variable's lifetime depends on the level of scope it has acquired.

2.2 PRIMITIVE TYPES AND VARIABLE DECLARATION

Epsilon has the following primitive or fundamental data types:

Primitive Type	Description
`boolean`	A single-byte representing `true` or `false`.
`char`	A single byte data type capable of holding one character
`int`	An integer with the natural size of integer in the local machine
`float`	Single-prescision floating point
`double`	Double-precision floating point

The syntax for declaring variables is:

```
[Modifiers] DataType variableName[ = InitialValue][Other Variables]
```

Variables are implicitly initialized to 0 in its declaration. Below are examples of variable declaration:

```
int x
char y, z
int a, b = 9, c   // b initialized with the value 9
```

2.3 POINTERS

A pointer is a variable whose value is a memory address. A pointer is always associated with a valid data type to determine the address range it can refer. The operator # is used to denote that a variable is a pointer. To declare a pointer:

```
int #ptr
```

The code above makes the pointer variable `ptr` points to a variable of type `int`. The operator `#` only applies to the variable it is attached to. This means, in the code

```
Int num, #numptr, num2
```

the only pointer variable is the `numptr`.

A pointer variable must always be initialized before it can be used. To initialize a pointer variable, assign it with a memory location either by memory allocation or using the address operator `@`, which returns the address of its operand.

```
Int num = 5, #numPtr, num2
numPtr = @num
```

To access variable that a pointer points, use the dereferencing operator `#`. It is a fatal logic error to use the dereferencing operator with an uninitialized pointer.

```
num2 = #numPtr     // num2 = 5
#numPtr = 10       // num = 10
```

2.3.1.1 Casting Pointers

A pointer variable, no matter what type of variable it refers, has the same size depending on the machine. This is because all pointer variables hold only one type of data, a memory address. The data type in the pointer declaration is required so the compiler can determine how many bytes a pointer refers. For casting pointers, the syntax used is:

```
(type #) pointerVarName
```

2.3.1.2 Generic Pointers

A generic pointer can represent any pointer type. Generic pointers are created by declaring a pointer variable that points to `void`. Example:

```
void #ptr
```

All pointer types can be assigned to a generic pointer and vice versa without the use of cast operation. A generic pointer cannot be dereferenced because there is no way the compiler can know the precise number of bytes the pointer refers.

2.3.1.3 Pointer to Pointers

To declare a pointer variable that points to another pointer variable, use the # operator multiple times.

2.4 COMPOSITE TYPES

Composite types are data structures derived from more than one data type. Epsilon has three composite types; arrays, records and objects.

2.4.1 Arrays

An array is a group of memory location all having the same name and data type. The syntax for declaring arrays is:

```
DataType arrayName[[Size]]
```

Example codes:

```
Int numbers[10]
Float grades[10], students[10]
```

The individual location in an array is called element. To access an element an array, use the following syntax:

The number between the square brackets is the index number of the element being accessed. If an array is declared to have 12 elements, the available indexes are 0 to 11. If the array is referenced using its ID only, the value of the first element is the one that will be returned. So,

```
int numbers[10]
numbers = 9 //The same as numbers[0] = 9
```

Index of array must be an integer number. Expressions that accesses an array element using index which is a negative integer or outside the range of possible elements is a logic error. The array name is a pointer to the first element of the array.

```
#numbers = 4      // the same as numbers[0] = 4
```

Programmers can also initialize an array in its declaration by using the initialization list. The syntax for this is:

```
DataType arrayName[[size]] = {Expression List}
```

Example:

```
Float numbers[10], grades[5] = {1.0, 1.25, 1.50, 1.75, 2.0}
```

When the size of an array is omitted in its declaration, the compiler will determine the size of the array base on the number of values in the initialization list. Omitting the array size without the initialization list is a compile-time error.

2.4.1.1 Multi-dimensional Arrays

You can define a multi-dimensional array by adding subscript on its declaration. The syntax for this is:

```
DataType arrayName[[Size]][[Size]]…
```

To define a 3x4 array, the code will be

```
Int table[3][4]
```

To initialize multi-dimensional arrays, use nested initialization list. Example:

```
Int table[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}
Int matrix[2][2][2] = {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}
/*
table[0][0] = 1          matrix[0][0][0] = 1
table[0][1] = 2          matrix[0][0][1] = 2
table[0][2] = 3          matrix[0][1][0] = 3
table[0][3] = 4          matrix[0][1][1] = 4
table[1][0] = 5          matrix[1][0][0] = 5
table[1][1] = 6          matrix[1][0][1] = 6
table[1][2] = 7          matrix[1][1][0] = 7
table[1][3] = 8          matrix[1][1][1] = 8
table[2][0] = 9
table[2][1] = 10
table[2][2] = 11
table[2][3] = 12
*/
```

2.4.2 Record

A record is a collection of variables under one ID. These variables are called fields of the record and are accessed using the dot operator. To define a record type, use the syntax,

```
record recordName
      [Variables Declarations]
end record
```

Example:

```
record date
      int day
      int month
      int year
end record
```

To declare a variable of type `record`, use the syntax:

```
record recordName variableName
```

Example:

```
record date birthday
```

To access fields of a record variable, use the dot (.) operator. Example

```
birthday.year = 1995
```

2.4.2.1 Record Initialization

To initialize a record variable in its declaration, use the initialization list:

```
record recordName variableName = {[Expression List]}
```

Example:

```
record date birthday = {25, 2, 1995}
/*
birthday.day = 25
birthday.month = 2
birthday.yeat = 1995
*/
```

When the number of values in the initialization list is less than the number of fields in the record, the remaining fields will be initialized to zero. When the number of values in the initialization list is greater than the number of fields in the record, the compiler will issue an error.

2.4.2.2 Record Pointers

Like other variables, it is possible to declare a pointer to a record variable. For example:

```
record date birthday = {25, 2, 1995}
record date #bdayPtr = @birthday
```

2.4.2.3 Member-wise Assignment

When the assignment operator (=) is used for assigning a record variable to another record variable of the same type, the values of fields of the variable in the right of the assignment is copied individually to the same fields of the variable in the left of the assignment. This is called member-wise assignment.

2.4.2.4 Records and Functions

Records and record fields are passed to functions by value. To pass them by reference, pass their addresses instead. Arrays of records, like other arrays, are passed-by-reference.

2.5 Type Conversion

Some operators, depending on its operand, converts the value of an operand from one type to another. The compiler must handle this situation according to C++ standards. A programmer can force the conversion of type using the cast operator `(type)`.

2.6 Static Variables

Local variables normally dies in the event of function termination. The exception to this rule are the static local variables. These kind of variables still have block scope, but their values are retained and can still be retrieved in function calls. To declare a static local variable, use the syntax

```
static DataType variableName[ = InitialValue][Other Variables]
```

3 Expressions, Operators and Statements

3.1 Expressions

An **expression** is the combination of symbols (e.g. literals, operators, function calls), that when executed, yields a value. Epsilon expressions are *nestable*. The following are the types of Epsilon expressions:

- Arithmetic
- Logical
- Relational

- Function call
- Constant expression

3.1.1 Constant Expressions and Function Calls

A constant expression is simply an expression that involves only literal values, such as `5`, `17.59` and `'a'`. The literal value TRUE is a Boolean whose value is any non-zero integer while FALSE is equal to zero. Number constants can only be expressed as decimal numbers.

A function call is an expression that invokes a pre-defined function and yields a value based on what the function returns. The syntax for function call is:

```
functionName(argumentList)
```

3.1.2 Arithmetic Expressions

Arithmetic expression in Epsilon is done using the five arithmetic operators:

1. + Addition
2. - Subtraction
3. * Multiplication
4. / Division
5. % Modulo

Dividing both integer values yields another integer. For example, `9 / 2` yields `4`. The modulo operator yields the remainder of an expression after division.

3.1.3 Relational and Logical Expressions

Relational expression uses the four relational operators and two equality operators.

1. > Greater than
2. < Less than
3. >= Greater than or equal
4. <= Less than or equal
5. == Equal
6. != Not equal

When relational expressions are evaluated, they yield Boolean values, which are either the constant `TRUE` or `FALSE`.

Logical expressions also yields Boolean values, but they use the logical operators `OR`, `AND` and `NOT`.

3.1.4 The sizeof Operator

The sizeof operator returns the size in bytes of its operand. The operand can be a variable, object, record, or a data type (fundamental or derived). The syntax for using sizeof operator is

```
Sizeof(operand)
```

3.2 Operator Precedence

Epsilon operator follows these precedence rules:

Operators	Description	Associativity
()	Parentheses (function call operator)	Left to right
[]	Array subscript	
.	Member selection via object	
->	Member selection via pointer	
++	Unary post-increment	
--	Unary post-decrement	
++	Unary pre-increment	Right to left
--	Unary pre-decrement	
-	Unary minus	
NOT	Unary logical negation	
(*type*)	Unary cast	
#	Dereference	
@	Address	
sizeof	Determine size in bytes	
*	Multiplication	Left to right
/	Division	
%	Modulus	
+	Addition	Left to right
-	Subtraction	
<	Relational less than	Left to right
<=	Relational less than or equal to	
>	Relational greater than	
>=	Relational greater than or equal to	
==	Relational is equal to	
!=	Relational is not equal to	
AND	Logical AND	Left to right
OR	Logical OR	Left to right
=	Assignment	Right to left
+=	Addition assignment	
-=	Subtraction assignment	
*=	Multiplication assignment	
/=	Division assignment	
%=	Modulus assignment	
,	Comma	Left to right

3.3 STATEMENTS

A statement is the smallest single standalone element in a program. It consists of expressions or other statements. Expressions and statements differs from each other because the former yields a value or values after its computation, while the latter expresses actions that must be carried out.

The Epsilon language has the following types of statement:

- Assignment
- Function call
- Control flow
- Preprocessor statement
- Variable declaration
- Compound statements

Epsilon statements are separated by a newline (`CR+LF` or `\r\n`) character. Programmers can use the line continuation operator (_) to write statements in multiple lines. A newline character must immediately follow the line continuation operator (except for whitespaces) or a syntax error will be issued.

3.3.1 Assignment Statements

Assignment statements is used in copying a value from an expression, called the **rvalue**, to a variable, called the **lvalue**, by using the six assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`.

Below are the examples of assignment statements.

```
X = 5
Y += X      // Y = Y + X
Y -= X      // Y = Y - X
Y *= X      // Y = Y * X
Y /= X      // Y = Y / X
Y %= X      // Y = Y % X
```

3.3.2 Function Call

See Section 4.1.1 Constant Expressions and Function Call

3.3.3 Control Flow Statements

See Section 5 Control Flow

3.3.4 Preprocessor Statements

Preprocessor statements are statements that are processed first by the compiler before compiling programs. Preprocessor statements must always be placed before any other statements in the program. Epsilon has two macro statements, the `link` and `define`.

The `link` macro statements defines the library files that must be included to the source code before compilation. The syntax for this is

```
Link <filename>
Link "filename"
```

When the filename is enclosed with angle brackets, it means that the compiler will search the library file in the standard library location. If the compiler doesn't find the specified library file, it will search the current directory of the program. If it still cannot find the file, the compiler will issue an error. The same principle applies to filenames enclosed with double quotations, except that the compiler will search the current directory first then the standard library location.

The `define` statements defines symbolic constants. The syntax for this is

```
Define identifier replacement-text
```

Example

```
Define SIZE 12
```

Using this `define` statement, the compiler will search all subsequent identifiers that do not appear in string literals *after* the statement that matches "SIZE" and replace them with "12". Note that the matching process is *case-insensitive*. The define statement enables programmers to define constant expressions that doesn't require memory space.

3.3.5 Variable Declarations

See Section 3.2 Primitive Types and Variable Declarations

3.3.6 Compound Statements

A compound statement is also called a block statement because they are composed of multiple statements. Examples of compound statements are class definition, function definition and control flow statements.

4 CLASSES AND OBJECTS

A **class** is a collection of data, called **members**, and functions that manipulate these data, called **methods**, which serves as the template for creating **objects**.

4.1 CLASS DEFINITION

Defining a class has the following syntax:

```
Class ClassName
      [Data Creation]
      [Function Definitions]
End class
```

Below is an example of class definition.

```
class Main
      function int main()
            printf("Hello World!")
            return 0
      end function
end class
```

This is the definition of a class called `Main`, in which the function `main` is defined. The function `main` serves as the entry point of the program, which makes it as a requirement in every Epsilon program. There should only be one `main` function in the entire program.

```
class SwapInt
      int x, y, temp
      public function SwapInt(int x, int y)
            this->x = x
```

```
            this->y = y
      end function
      public function swap()
            temp = x
            x = y
            y = x
      end function
end class

class Main
      function main() as int
            int x = 7, y = 9
            SwapInt swapper = new SwapInt()
            swapper.swap(x, y)
            return 0
      end function
end class
```

Another example of class definition is presented above. The body of the class starts after the class header and ends before the `end class` statement.

4.2 Object Declaration and Instantiation

To declare an object, use the syntax

```
ClassName ObjectName
```

You can also declare multiple objects of the same class by separating names of the objects with comma (,).

```
ClassName ObjectName1, ObjectName2, ObjectName3
```

To instantiate an object, use the syntax

```
ClassName ObjectName = new ClassName(argumentList)
```

It is also possible to instantiate objects in the same line

```
ClassName Object1 = new ClassName(argumentList), Object2 = new
ClassName(argumentList)
```

Note that every object has its own copy of all data members of the class.

4.3 Constructors and Destructors

The arguments in the `argumentList` in object instantiation are passed to the constructor of the class. The class **constructor** is a special function that is called once the object is instantiated. Although a constructor is a function, the syntax for defining it is a bit different. The syntax for defining a constructor is

```
constructor ClassName([Parameters])
      [Statement List]
end constructor
```

Constructors are usually used in initializing data of the class. Constructors is treated as *public* functions that doesn't return anything, even `void`. (See Section 5.2 Access Modifiers) If the definition of class has no constructor, the compiler will provide a **default constructor**, which is a constructor with no parameter and statement. Constructors are implicitly called in object creation.

Another type of special member function is the **destructor**. Destructors are implicitly called when an object is destroyed. Destructors doesn't release the object's memory but rather it performs some termination housekeeping before the object's memory is reclaimed. To define a destructor, use the syntax,

```
destructor ClassName()
      [Statement List]
end destructor
```

If the definition of class has no destructor, the compiler will provide a **default destructor**, which is a destructor with no parameter and statement. Destructors are naturally a public function with no parameters. It is syntax error to define a destructor with parameters.

4.4 Objects and Functions

Just like record variables, class objects are passed to function by value.

4.5 Composition

Composition, or the *has-a relationship*, is the feature of object-oriented programming that allows classes to have objects of other classes as its members. These member objects are constructed in the order of their declaration in the class definition and not in the order of how they are initialized in the constructor.

4.6 Pointers to Objects and Object Size

Like any other variables, an object can also have a pointer variable that refers to it. For example,

```
Student stud = new Student()
Student #stud_ptr = @stud
Stud_ptr->x = 9   // the same as (*stud_ptr).x
```

Once a pointer to object is declared, the members of the object being referred can be accessed through the pointer using the arrow operator (`->`).

Logically, programmers may think that the size of an object constitutes the sizes of its data and function members. But physically, an object instance has a copy of only the data members. The compiler creates only one copy of the functions separate from all objects of the class. This copy is being shared by all instances.

4.7 Self-Referencing

Sometimes, the name of a local variable in a member function's parameters is the same with the data member of the class. In this case, the data member is hidden because the local variable will prevail in this scope. To access the data member that is hidden, use the `this` keyword. The `this` keyword is actually a pointer to the current object instance. Because it is a pointer, the arrow operator must be used with it.

4.8 Static Class Member

As mentioned before, objects normally have their own copy of data members. The exception to this rule are the static data members. These kind of data members are declared like static local variables and is shared among objects of that class. Static data members have class scope and can be declared as public or private.

Like data members, function members can also be declared as static. A static function member is a service of the class and is not attached to any object. They cannot accessed through *this* pointer because the latter points to an object and the former is not attached to any object. They also cannot access non-static data members because the latter belongs to a specific object. Violating these rules is a semantic error.

Static data and function members can be accessed like ordinary members. They can also be accessed through the scope operator (`::`) using the syntax

```
className::staticMember
```

4.9 Member-wise Assignment

The assignment operator can be used to assign an object to another object of the same type. This assignment is performed through member-wise assignment, a process where the data member of the object in the right of the assignment operator is assigned individually to the same data members of the object in the left of the assignment operator.

4.10 Inheritance

Inheritance, or the *is-a relationship*, is another important feature of OOP that allows a class to absorb or inherit the all the capabilities of an existing class and modify or enhance them. The existing class is called the superclass (or *parent class* or *base class*) while the new class is called the subclass (or *child class* or *derived class*). To define heritance, use the following syntax,

```
class SubClassName extends SuperClassName
      [Statement List]
end class
```

Example:

```
class Person
      String name
      Int age

      Public function getName() as String
            Return name
      End function
      Public function setName(String name)
            this->name = name
      end function
      Public function getAge() as int
            Return name
      End function
      Public function setAge(int age)
            this->age = age
      end function
end class

class Student extends Person
      String course
      Public function getCourse() as String
            Return course
      End function
      Public function setCourse(String course)
            this->course = course
      end function
end class

class Main
      Student stud = new Student()
      Stud.setName("Rexwyn")
      Stud.setAge(19)
      Stud.setCourse("BSCS")
End class
```

In the example above, the data and function members of `Person` is inherited by `Student` and can be used by `Student` freely as if those members where defined inside it. It is also possible to override the members of the superclass by redefining them. Note that class inheritance includes the constructor and destructor of the superclass.

4.10.1 Class Hierarchy

Inheritance relationships between classes forms the class hierarchy. It is possible to define a subclass that extends another subclass of a superclass. For example, a class called `Laptop` may extend the class `Computer`, which extends the class `ElectronicDevice`. The class `Computer` is the direct base class of `Laptop` while `ElectronicDevice` is the indirect base class of `Laptop`. Note that Epsilon doesn't support multiple inheritance.

4.10.2 Overriding Member Functions

It is also possible to override the members of the superclass by redefining them. Note that this feature still conforms to the rule function signatures. This means defining a function in the subclass with the same name with a function in the superclass but with different signatures will not override the latter. Constructors and destructors can also be overridden by redefining them.

```
class Student extends Person
      String course
      Public function getCourse() as String
            Return course
      End function
      Public function setCourse(String course)
            this->course = course
      end function
      Public function setName(String name)
            this.name = "My name is " + name
      end function
end class
```

In the example above, the `setName()` member function of `Person` is overridden by the `Student` class.

4.10.3 Accessing Overridden Functions

Once a function in the superclass is overridden, the subclass can still access this function inside the subclass definition using the `parent` keyword and the arrow operator (`->`). The keyword `parent` is a pointer to the direct superclass of the subclass.

5 CONTROL FLOW

5.1 CONDITIONAL EXPRESSION

A conditional expression is what controls the execution of control flow statements. In general, all expressions are conditional expression. All non-zero values are equivalent to TRUE, while a zero value is equal to FALSE.

5.2 SELECTION

The selection statement provides means of choosing between two or more execution paths.

5.2.1 The Two-Way Selection Statement

The two-way selection statement has the syntax:

```
if Conditional-Expression then
      [Statement List]
end if
```

This statement is usually called if-then statement. Another variation, called the if-then-else statement, has the syntax

```
if Conditional-Expression then
      [Statement List]
else
      [Statement List]
end if
```

The nested-if has the following syntax:

```
if Conditional-Expression then
      [Statement List]
else if
      [Statement List]
…
[else
      [Statement List]]
end if
```

5.2.2 Multiple-Selection Statement

The multiple-selection statement, usually called the switch statement, has the following syntax:

```
Switch Identifier
      Case Expression:
```

```
            [Statement List]
      Break
      …
      [default:
[Statement List]]
End switch
```

It is possible to define multiple cases to simulate OR operation.

```
Switch Identifier
      Case Expression:
      Case Expression:
      …
            [Statement List]
      Break
      …
      [default:
[Statement List]]
End switch
```

5.3 ITERATION

5.3.1 The while Loop

The while loop has the following syntax:

```
while Conditional-Expression
      [Statement List]
end while
```

5.3.2 The do-while Loop

The do-while loop has the following syntax:

```
Do
      [Statement List]
while Conditional-Expression
```

5.3.3 The for Loop

The for loop has the following syntax:

```
For (Variable Declaration | Variable Identifier) = Value to Value [reverse]
      [Statement List]
Next
```

5.3.4 break and continue Statements

The break statement, when encountered inside the switch and iteration statements, stops the flow of execution in that statement and jumps to the next statement immediately after it. The continue statement, when encountered inside iteration statements, jumps the execution to the next iteration.

5.4 EXCEPTION HANDLING

An exception is an error in a program. Exception handling is a mechanism for resolving exceptions. This allows programmers to create application that are robust and fault-tolerant. Epsilon provides the try-catch statements for exception handling. The syntax for this is

```
Try
      [Try Statements]
[Catch [ExceptionType [ParameterName]]
      [Catch Statements]
Catch […]]
End Try
```

The try statements is the code section that may *throw* an exception. When this section really did throw an exception, the remaining statements in the `try` statement will be skipped and, according to the type of exception thrown, then appropriate catch handler will execute the its catch statements. A `catch` handler is chosen if its exception-parameter type is identical or a superclass of the type of the exception thrown. The exception-parameter can have an optional name to allow interaction with it. Each `catch` handler can only have one exception-parameter. When multiple catch handler have the same type of exception parameter, the first will be the one to execute.

Note that the `try` statements and `catch` statements are considered block of statements and variables declare within it dies in its termination.

Unhandled exception terminates the program. Exception is either thrown implicitly by the compiler or by the user using the throw statement.

```
Throw [Exception]
```

A very basic example of exception handling is

```
Try
      Throw 12
Catch int e
      printf("The program threw an exception of value %d\n, e)
end try
```

Generally, a result of an expression can be an operand of a `throw` statement. If the operand is an object of a class, it is called **object exception**. A `throw` statement must only have one operand or none at all. `Throw` statement executed outside a `try-catch` block terminates the program immediately.

It is also possible to define a `try-catch` block inside the `try` block or the `catch` block. In cases like this, the internal `try-catch` block may forward the exception to the external `try-catch` block.

When an exception is thrown but not caught in a particular scope, a process called **stack unwinding** will take place. In this process, the function that cannot handle the exception properly will terminate, destroying all local variables inside it and returns the control to the statement that called that function. If that statement is inside a `try-catch` block, an attempt to handle the exception is made. If the statement is not inside a `try-catch` block or there is no appropriate handler is specified, stack unwinding occurs again.

6 FUNCTIONS

Functions, also called methods or subprograms, are collection of valid statements that accepts some data as argument, called **parameters**, and return a value after its execution. If a function is declared outside a class, it will have a file scope.

6.1 FUNCTION DEFINITION

The syntax for defining a function is:

```
[Modifiers] function functionName(parameterList) as returnType
      [Statements]
end function
```

Below is an example of a function that accepts two integers and returns their sum.

```
public function add(int x, int y) as int
      return x + y
end function
```

Functions are identified using their signature. A **signature** of a function consists of its name and the data types of its parameters. Example, the function

```
function add(int x, int y) as int
      return x + y
end function
```

has a signature of

```
add int int
```

Function signature must specify the name and exact type and order of the parameters. It is possible to declare two functions with the same name as long as they have different signatures. This feature is called function overloading. Take note that a class constructor can also be overloaded. To simplify language implementation, Epsilon doesn't support argument coercion, which is the automatic type conversion of argument to match the data type of the parameters during function call.

6.2 Access Modifiers

Epsilon allows functions to be of two types based on how they can be accessed, public or private. Public functions can be accessed in any part of the program in which the class it is defined is active. Public functions are declared with the access modifier `public`. Private functions can only be accessed within the class they are defined and are declared with the access modifier `private`. The absence of access modifier in function definition makes the function public by default.