# Contents

[Contents 1](#_Toc403935795)

[1 Classes and Objects 3](#_Toc403935796)

[1.1 Class Definition 3](#_Toc403935797)

[1.2 Object Declaration and Instantiation 4](#_Toc403935798)

[1.3 Constructors and Destructors 4](#_Toc403935799)

[2 Data Types and Variables 4](#_Toc403935800)

[2.1 Variable Declaration 5](#_Toc403935801)

[2.2 Primitive Types and Variable Declaration 5](#_Toc403935802)

[2.3 Pointers 5](#_Toc403935803)

[2.4 Composite Types 6](#_Toc403935804)

[2.4.1 Arrays 6](#_Toc403935805)

[2.4.2 Hash (Experimental) 8](#_Toc403935806)

[2.4.3 Enumeration 8](#_Toc403935807)

[2.4.4 Record 8](#_Toc403935808)

[2.5 Type Conversion 10](#_Toc403935809)

[2.6 Type Aliasing 10](#_Toc403935810)

[2.7 Scope 10](#_Toc403935811)

[3 Expressions, Operators and Statements 11](#_Toc403935812)

[3.1 Expressions 11](#_Toc403935813)

[3.1.1 Constant Expressions and Function Calls 11](#_Toc403935814)

[3.1.2 Arithmetic Expressions 11](#_Toc403935815)

[3.1.3 Relational and Logical Expressions 11](#_Toc403935816)

[3.2 Operator Precedence 12](#_Toc403935817)

[3.3 Statements 13](#_Toc403935818)

[3.3.1 Assignment Statements 13](#_Toc403935819)

[3.3.2 Function Call 13](#_Toc403935820)

[3.3.3 Control Flow Statements 13](#_Toc403935821)

[3.3.4 Macro Statements 13](#_Toc403935822)

[3.3.5 Variable Declarations 13](#_Toc403935823)

[3.3.6 Compound Statements 14](#_Toc403935824)

[4 Control Flow 14](#_Toc403935825)

[4.1 Conditional Expression 14](#_Toc403935826)

[4.2 Selection 14](#_Toc403935827)

[4.2.1 The Two-Way Selection Statement 14](#_Toc403935828)

[4.2.2 Multiple-Selection Statement 14](#_Toc403935829)

[4.3 Iteration 15](#_Toc403935830)

[4.3.1 The while Loop 15](#_Toc403935831)

[4.3.2 The do-while Loop 15](#_Toc403935832)

[4.3.3 The for Loop 15](#_Toc403935833)

[4.3.4 break and continue Statements 16](#_Toc403935834)

[4.4 Exception Handling 16](#_Toc403935835)

[5 Functions 16](#_Toc403935836)

[5.1 Function Definition 16](#_Toc403935837)

[5.2 Access Modifiers 16](#_Toc403935838)

# 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**.

## 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.

## 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)

## 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. This function should have a name similar to the class name and should not return any value. Constructors are usually used in initializing data of the class. Constructors must be defined as public functions. (See [Section 5.2 Access Modifiers](#_Access_Modifiers)) If the definition of class has no constructor, the compiler will provide a **default constructor**, which is a constructor with no parameter. Constructors are implicitly called in object creation.

# 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.

## 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 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.

Epsilon has 5 different classifications of data types: primitive, composite, pointers and abstract data types. Each of these classifications will be discussed in the subsequent sections.

## Primitive Types and Variable Declaration

Epsilon has the following primitive 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] variableName as DataType

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

x as int

y, z as char

a, b = 9, c as int // b initialized with the value 9

## 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:

#ptr as int

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

num, #numptr, num2 as int

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.

num = 5, #numPtr, num2 as int

numPtr = @num

To access variable that a pointer points, use the dereferencing operator #. An error of **UninitializedPointerException** will result if the dereferencing operator is used with uninitialized pointer.

num2 = #numPtr // num2 = 5

#numPtr = 10 // num = 10

#### 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

#### Generic Pointers

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

#ptr as void

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.

#### Pointer to Pointers

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

## Composite Types

**Composite types** are data structures derived from more than one data type. Epsilon has the following composite types:

* Array
* Hash
* Enumeration
* Record

### Arrays

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

arrayName[[Size]] as Type

Example codes:

numbers[10] as int

grades[10], students[10] as float

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 will return an **OutofBoundArrayException**. 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:

arrayName[[size]] = {Expression List}

Example:

numbers[10], grades[5] = {1.0, 1.25, 1.50, 1.75, 2.0} as float

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.

#### Multi-dimensional Arrays

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

arrayName[[Size]][[Size]]… as Type

To define a 3x4 array, the code will be

table[3][4] as int

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

table[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}} as 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

\*/

### Hash (Experimental)

Hash is like an array. It is also a collection of memory location with the same name, but unlike an array, it is dynamic and the location of its elements are not necessarily contagiously stored in the memory. It is dynamic because the size of its elements can change in runtime. Another big difference it has from an array is that its elements are not accessed using index numbers, but rather, they are accessed using **keys** which are strings of characters that identifies each element. A **pair** is the combination of a key and the value that it identifies. Every key in a hash must be unique.

### Enumeration

**Enumeration** is a set of integer constants represented by IDs. The values in the enumeration starts at 0, unless specified otherwise, and are incremented by 1.

enum cardsuit

Clubs

Diamonds

Hearts

Spades

end enum

### 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:

variableName as record recordName

Example:

birthday as record date

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

birthday.year = 1995

#### Record Initialization

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

variableName = {[Expression List]} as record recordName

Example:

birthday = {25, 2, 1995} as record date

/\*

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.

#### Record Pointers

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

birthday = {25, 2, 1995} as record date

#bdayPtr = @birthday as record date

## Type Conversion

There are two ways data types in Epsilon is converted; implicitly and explicitly. When an operator has operands of different types, Epsilon performs an implicit conversion according to some rules. Automatic conversions are those that converts ‘narrower’ type to ‘wider’ type. This conversion doesn’t cause information loss. //NOT FINISH, SEE Appendix A Sec 6 of K&R Book

## Type Aliasing

Epsilon supports the mechanism of giving an alias to any type using the keyword alias. The syntax for type aliasing is:

alias type newTypeName

Example:

alias int Integer

alias record date Date

## 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, and inside the block…end block statements.
2. **File scope** – objects declared in this level have scopes from the point of their creation until the end of file they are created in.

At any part of the program, an object can either be local or nonlocal. An active object is said to be **local** in a defined region if it is declared there while. An active object is said to be **nonlocal** in a defined region if it is not declared there.

# Expressions, Operators and Statements

## 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

### 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.

Integer constants can also be expressed as octal or hexadecimal. Octal integers has a leading 0 while hexadecimals has a leading 0x or 0X.

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)

### 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.

### 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.

## 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 plus |  |
| - | Unary minus |  |
| NOT | Unary logical negation |  |
| ~ | Unary bitwise complement |  |
| ( *type* ) | Unary cast |  |
| # | Dereference |  |
| @ | Address |  |
| sizeof | Determine size in bytes |  |
| \* | Multiplication | Left to right |
| / | Division |  |
| % | Modulus |  |
| + | Addition | Left to right |
| - | Subtraction |  |
| << | Bitwise left shift | Left to right |
| >> | Bitwise right shift |  |
| < | 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 |  |
| & | Bitwise AND | Left to right |
| ^ | Bitwise exclusive OR | Left to right |
| | | Bitwise inclusive OR | Left to right |
| AND | Logical AND | Left to right |
| OR | Logical OR | Left to right |
| ?: | Ternary conditional | Right to left |
| = | Assignment | Right to left |
| += | Addition assignment |  |
| -= | Subtraction assignment |  |
| \*= | Multiplication assignment |  |
| /= | Division assignment |  |
| %= | Modulus assignment |  |
| &= | Bitwise AND assignment |  |
| ^= | Bitwise exclusive OR assignment |  |
| |= | Bitwise inclusive OR assignment |  |
| <<= | Bitwise left shift assignment |  |
| >>= | Bitwise right shift |  |
| , | Comma | Left to right |

## 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
* Macro statement
* Variable declaration
* Compound statements

All Epsilon statement must be terminated by a newline (CR+LF or \r\n) character.

### 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

### Function Call

See [Section 4.1.1 Constant Expressions and Function Call](#_Constant_Expressions_and)

### Control Flow Statements

See [Section 5 Control Flow](#_Control_Flow)

### Macro Statements

**Macro statements** are statements that are processed first by the compiler before compiling programs. Macro statements must always be placed before any other statements in the program. Epsilon has two macro statements, the \_link and \_define.

### Variable Declarations

See [Section 3.2 Primitive Types and Variable Declarations](#_Primitive_Types_and)

### Compound Statements

# Control Flow

## 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.

## Selection

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

### 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

### 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

## Iteration

### The while Loop

The while loop has the following syntax:

while Conditional-Expression

[Statement List]

end while

### The do-while Loop

The do-while loop has the following syntax:

Do

[Statement List]

while Conditional-Expression

### The for Loop

The for loop has the following syntax:

For (Variable Declaration | Variable Identifier) = Value to Value [reverse]

[Statement List]

Next

### 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.

## Exception Handling

# 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. All functions in Epsilon must be declared as methods of a class, which means it is not allowed to define a function outside a class.

## 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(x as int, y as int) as int

return x + y

end function

## 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.