Paweł Olber
Expert in the field of forensic IT examination, Voivodeship Police Forensic Laboratory in Olsztyn
pawel.olber@ol.policja.gov.pl

The use of EnCase Forensic program scripts in forensic research of digital data carriers

Summary

The purpose of this work is to present users of EnCase Forensic programming language syntax, EnScript, to create their ownsoftware solutions, as well as to provide the possibility of its use in forensic research of digital storage media. In order to understand above described elements, description elements above, practical programming examples are given. Examples in this paper allow you to understand the existing code, scripts, contained, among others, in the EnCase API documentation. This documentation is the best source to get detailed information about the above classes of the above mentioned programming language, its components and functions. In addition to the aforementioned document, there are a number of applications created by EnScript programming language enthusiasts on Internet, which make it easy to write your own scripts, and represent a valuable source of information and inspiration. The advantage of knowing the above mentioned programming language is the ability to create ones own software solutions, allowing for the automation of tasks carried out within the context of investigative analysis, which is a very important element in the work of investigative science.

Keywords Encase Forensic, EnCase API, EnScript

Introduction and purpose of the article

This publication provides an overview of the EnScript programming language, defined in the *EnCase Forensic* program, which belongs to the one of the most respected computer forensic programs (*computer forensic*) in the world. The conducted examinations concerning programming languages mentioned above, are based in particular on work in the *EnCase Forensic* program, which has built-in own IT environment and includes some scripts enabling automation of time-consuming investigative analyses. The IT environment of the above-mentioned program is quite simple and lacks many useful functions for formatting the source code. Therefore, the sample scripts describing the syntax of the *EnScript* language will be written using *Microsoft Visual C++ Express* software that is configured in such a way as to be compatible with the *EnCase* program. Microsoft *Visual C ++ Express* allows maintaining clear and transparent structure of created scripts, which facilitates their analysis and understanding, and avoiding programming errors. Information on the syntax of the *EnScript* programming language will be collected primarily as a result of the analysis of the script source code contained in the so-called *API* documentation of the *EnCase Forensic* program. In addition, scripts available on the Internet will be examined, created by enthusiasts of the above mentioned programming language.

The the first part of this article the author will present basic information and concepts of the *Enscript* programming language. Then a simple program will be presented that will be the starting point for the creation of more complex applications. Subsequently, the syntax structure of the *Enscript* programming language will be clarified and, in particular, the basic data types, complex variables, control instructions, operations on variables, as well as issues related to object-oriented programming. At the end, the author's software solutions will be presented, formed for the purpose of ongoing forensic casework.

The aim of author's endeavours is to discuss the *EnScript* programming language, defined in the *EnCase Forensic* program and to show its potential in forensic examination of digital data carriers. The information contained in this publication should be useful to those in charge of computer forensics, in the creation of proprietary analytical tools used for automation of time-consuming tasks performed during the analyses. The present publication seems to be of value due to no specific training on the discussed issue available within the police.

Enscript – basic information

Enscript is an object-oriented programming language based on the syntax of two languages: *C ++* and *Java*. This language allows the automation of the tasks being performed as well as the simplification of many time-consuming activities carried out in the course of examinations, such as searching and analysing specific types of documents, processing and extraction of data stored in different types of files, creating files containing hash functions etc. Using the programming language, it is also possible to process the information selected by analysis of the data added to bookmarks or records.

Enscript – syntax

To begin learning how to create scripts in *EnScript* language, the user should be provided with some knowledge on basic issues and terminology regarding programming. In this article, the structure of a program, the course of processing its components and functions will be also discussed. The knowledge of these issues is quite important because it will help the user further to understand the source code of created applications and will allow for an effective analysis of existing scripts, among others, present in *EnCase* and available on the Internet.

Whitespace

Whitespace, i.e. spaces, tabs, and new line markers are not relevant in the source code and are ignored by the compiling program, which translates the source code into machine language.

Comments

Any portion of the software source code may be subject to commentary. The comment is completely ignored by the compiler, however it can be useful for writing and reading the code. Comments are used to describe, to clarify certain parts of the program code, to separate parts of the code and labelling functions. In the *EnScript* language there are two possible ways of commenting code: line and block.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // Example of line comment.
    /* Example of block comment,
      that can span multiple lines.
    */

  }
}
```

Directive *include*

In the language of *EnScript*, *include* is called a directive – information about joining contents of a specified file with the extension *enscript* to the compiled source file, referred to as a library. The libraries contain functions that can be used in the source file. The file name is given without an extension.

Example:

```
  This script is a modification of
  "Sweep Enterprise" of EnCase V4.
  Report all bugs and queries to
  technicalsupport@guidancesoftware.com
//////////////////////////////////////////
*/

include "GSI_Basic"
include "GSI_SweepCaseLib"
include "GSI_DefaultModules"
```

Directive *typelib*

In the *EnScript* language, *typelib* is used to connect, to the compiled source file, contents of a specified file, called the library, derived from an external application such as programs of the *Microsoft Office* packet.

Example:

```
typelib Excel "Excel.Sheet"
typelib Scripting "Scripting.FileSystemObject"
#ifdef Excel
class MainClass {
  void Main() {
    SystemClass::ClearConsole();
    CreateExcel();
  }
  void CreateExcel() {
    Excel::Application app;
    if (app.Create()) {
      app.SetVisible(true);
      Excel::Workbooks books  = app.Workbooks();
      Excel::Workbook book    = books.Add();
      Excel::Sheets sheets    = book.Worksheets();
      Excel::Wordsheet sheet  = Excel::Worksheet
                    ::TypeCast(sheets.Item(1));
      Excel::Range cells = sheet.Cells();
      AddTable(cells);
      AddPieChart(sheet);
  }
}
```

Console

The console is actually a file, the contents of which are displayed in the preview window of *EnCase Forensic*. The console is available by pressing the tab "Console" (V6) or via the menu: *View* -> Console (V7) and it allows viewing all sorts of messages and other content, e.g. information about the progress of the program. In order to display a message in the console, the command *WriteLine()* should be used.

The first program

Conventional books about programming start programming lessons with a presentation of an application that displays a sentence in a console or in a screen: "*Hello World*". This convention is also followed in this publication. The first program will be a starting point for more complex applications that allow to understand the syntax of the *EnScript language*.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    Console.WriteLine("Hello World");
  }

}
```

Result:

```
Hello World
```

The main class of the program

The program, written in *EnScript* is a set of variables, definitions and function calls. Each script created by *EnCase* has the class name *MainClass*, within which the *Main* function can be found, as a parameter adopting a variable of the *CaseClass* type, which is the equivalent of matter created by the user of the program. Inside the *MainClass*, other classes and functions are defined. When you run the script, the *EnCase* program localizes the *MainClass* object and calls its constructor. The issues considering the constructors will be presented later in the subsequent part of this publication. Then the aforementioned object calls the function *Main*, which is preceded by the keyword *void*. This means that the function does not return any data. Finally, the program *EnCase* runs the destructor of the *MainClass* and ultimately removes the object of the main class. At this point, the program terminates.

The concept of the variable, and the basic data types

Variable is the place in the operational memory of a computer which stores a single value of a specified type. Each variable has a name, enabling to make a reference. Before you start, the variable should be **declared**, in other words, the compiler should be informed that this name stands for a variable of a given data type. Variable declaration syntax is as follows:

```
type variable_name = variable_value;
```

The type specifies the sort of information that can be stored in a variable. These can be, for example, integers, real numbers, and chains of characters, called *strings*. The name of a variable should well state its importance and meet certain criteria. Here are some of them:

- variable name should be a word or an abbreviation that specifies its purpose
- name may consist of lower case and upper case letters as well as numbers and the underscore character. It should be noted that the variable name cannot begin with a digit. Unlike other programming languages, the name of the variable can contain national characters such as the letters *ć* or *ź*.

The *EnScript* language supports different data types natively, which means that conversions between different types of data are performed automatically and the use of a specific type is permitted in any context.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // declaration of variable "i"
    // assign value 1 to the variable
    int i = 1;

    // displaying the value of variable "i"
    Console.WriteLine("i = " + i);

    // adding value 1 to the
    // declared variable
    i = i + 1;

    Console.WriteLine("i = " + i);

  }
}
```

Result:

```
i = 1
i = 2
```

Basic data types

Data types specify the manner of using memory in the program. Specifying the data type, the information is passed to the compiler how to create a specific section of the memory and how to perform operations on it. Data of different types differ in the speed of action, the size of occupied memory, and of course, functionality; for example, there is data that can be multiplied by themselves as well as data that does not perform multiplication operation at all – even texts.

char

The variable of type *char* stores one character. UNICODE characters with codes from 0 to 65535 can be assigned to a variable of this type. Variables of *char* type can be assigned as integers (*int*) and vice versa, integer variables can be assigned to variable characters. In the event of such an operation, *EnScript* converts the characters in accordance with the ASCII encoding. In ASCII code lower case "a" is assigned to the value of 97. When a variable of the *char* type is assigned to a character, it is in fact a number from 0 to 255. However, one should be aware of the difference between a value, and a character, for example: 5 and "5" is not the same because 5 is a value, however "5" is a character and has the value of 53, as is in the case of the letter "a", which has the value of 97.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // assigning number 82 to the variable int
    int i = 82;

    // displaying the value of variable 'i'
    Console.WriteLine("i = " + i);

    // assigning variable 'i' to the char
    // type variable
    char a = i;

    // displaying the value of variable 'a'
    Console.WriteLine("a = " + a);

    // assigning a character to the char type
    // type variable
    char b = 'v';

    // displaying the value of variable 'b'
    Console.WriteLine("b = " + b);

    // assigning variable 'b' to the variable 'j'
    int j = b;

    // displaying the value of variable 'j'
    Console.WriteLine("j = " + j);
  }
}
```

Result:

```
i = 82
a = R
b = v
j = 118
```

String

The variable *String* stores character strings (text), which can perform a number of operations. Strings can be concatenated together using the "+" operator.

Example:

```
class MainClass {
  void Main(CaseClass c) {
    // Cleaning the console display
    SystemClass::ClearConsole();
    // Assigning value to the variable s1
    String s1 = "Hello";
    // Assigning value to the variable s2
    String s2 = " World";
    // Variable s1 obtains value "Hello World"
    s1 = s1 + s2;
    // Variable s1 obtains value "Hello World."
    s1 = s1 + ".";
    // Starting a new line
    s1 = s1 + '\n';
    // Displaying the console output
    Console.Write(s1);
  }
}
```

Result:

```
Hello World.
```

Strings may contain special characters. In order to use special characters in the text, one should use a backslash (*backslash*). The group of special characters includes: tab, slash, single quotation mark, newline.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    SystemClass::ClearConsole();

    String s1 = "Tab \t, backslash \\, ↵
                 single quotation marks \",  ↵
                 newline \n";

    Console.Write(s1);

  }
}
```

Result:

```
Tab,    backslash \, single quotation marks ",
newline
```

It is also possible to include in text additional special characters from UNICODE, for example trademark symbol ®. This character should be written with a slash in hexadecimal form[1].

Example:

```
class MainClass {
  void Main(CaseClass c) {

    SystemClass::ClearConsole();

    String s1 = "This is the trademark symbol: ↵
                 \x00AE";
    Console.Write(s1);

    String s2 = "The Russian alphabet: \x0414";
    Console.Write(s2);

  }
}
```

Result:

```
This is the trademark symbol: ®
The Russian alphabet: Д
```

1 TA table of UNICODE is available at the following address: *http://unicode-table.com/en/*.

Because all characters in quotation marks are significant, including white space, you cannot insert a new row in the middle of a chain.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    SystemClass::ClearConsole();
    // Uncorrect connection
    s1 = "This is a very long text,
    that cannot be saved in this form.
    Running the script will fail";
  }
}
```

If you want to connect very long strings, just put them in quotation marks, one after another. Strings will be merged at compile time of the program. In the case of very long strings, do not use the operator connect "+", because this will cause unnecessary lengthening of the compile-time of the program.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    SystemClass::ClearConsole();
    // Correct connection of very long strings
    s1 = "This is a very long text "
    "that has been saved correctly. \n"
    "Running the script succeeds.";

    Console.WriteLine(s1);
  }
}
```

Result:

```
This is a very long text that has been saved
correctly.
Running the script succeeds.
```

Strings can be treated as an array of characters and addressed individually. **Note:** Indexing always takes place from 0. The last item has an index equal to the size of the table-1.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    SystemClass::ClearConsole();

    String s = "1234567890";
    Console.WriteLine("The original string: " +
                      s);
    // Displaying value 1
    Console.WriteLine(" The first character of
                      the string: " + s[0] = 'X';
    // Empty line
    Console.WriteLine(" ");
    // Inserting 'X' as the first character
    // of the string
    s[0] = 'X';      // s = X234567890

    Console.WriteLine("New string: " + s);
    Console.WriteLine("The first character of
                      the string: " + s);
  }
}
```

Result:

```
The original string: 1234567890
The first character of the string: 1

New string: X234567890
The first character of the string: X
```

Numeric data types

In the *EnScript* language there are seven numeric data types. Numeric types have a fixed size, regardless the process and platform. Part of the numeric types have similar names, they differ in only the first letter of *u*. In the *EnScript* language this is a so called modifier, and by using it one can somewhat influence the scope of the data that can be stored in variables. The modifier is nothing other than a way to identify different varieties of one type. The modifier *u* specifies that the type represents only positive data. All numeric data types of the *EnScript* language are presented in tabular form and sorted according to the size of the data that can be stored:

Table 1 Statement of the numeric data types

Type	Size (B)	Min. value	Max. value
Short	2	-32768	32767
Ushort	2	0	65535
Int	4	-2147483648	2147483647
Uint	4	0	4294967295
Long	8	-9223372032559808513	9223372032559808512
Ulong	8	0	18446744065119617025
Double	8	-1.79 * 1E+308	1.79 * 1E+308

Example:

```
class MainClass {
  void Main(CaseClass c) {
    // Cleaning the console display
    SystemClass::ClearConsole();

    // Short type variable
    short short_min_value = -32768;
    short short_max_value = 32767;
    Console.WriteLine("Min. value of a variable
        of type 'short': " + short_min_value);
    Console.WriteLine("Max. value of a variable
        of type 'short': " + short_max_value);
    Console.WriteLine("-----------------------");

    // Unshort type variable
    ushort ushort_min_value = 0;
    ushort ushort_max_value = 65535;
    Console.WriteLine("Min. value of a variable
        of type 'ushort': " + ushort_min_value);
    Console.WriteLine("Max. value of a variable
        of type 'ushort': " + ushort_max_value);
    Console.WriteLine("-----------------------");

    // Int type variable
    int int_min_value = -2147483648;
    int int_max_value = 2147483647;
    Console.WriteLine("Min. value of a variable
        of type 'int': " + int_min_value);
    Console.WriteLine("Max. value of a variable
        of type 'int': " + int_max_value);
    Console.WriteLine("-----------------------");

    // Unit type variable
    unit unit_min_value = 0;
    unit unit_max_value = 4294967295;
    Console.WriteLine("Min. value of a variable
        of type 'unit': " + unit_min_value);
    Console.WriteLine("Max. value of a variable
        of type 'unit': " + unit_max_value);
    Console.WriteLine("-----------------------");

    Console.WriteLine(s1);
  }
}
```

Wynik:

```
Min. value of a variable of type 'short':
-32768
Max. value of a variable of type 'short': 32767
-----------------------------------------------
Min. value of a variable of type 'ushort': 0
Max. value of a variable of type 'ushort':
65535
-----------------------------------------------
Min. value of a variable of type 'int':
-2147483648
Max. value of a variable of type 'int':
2147483647
-----------------------------------------------
Min. value of a variable of type 'uint': 0
Max. value of a variable of type 'uint':
4294967295
-----------------------------------------------
```

void

This special empty data type is used to indicate that the function does not return any result. Example of the use of the above mentioned type was also presented during the discussion of the function.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // Cleaning the console display
    SystemClass::ClearConsole();
    // Calling NoData() function;
    NoData();

    // Calling ReturnData() function
    // and assigning the returned value
    // to the variable 'text'
    String test = ReturnData();
    Console.WriteLine(text);
  }

  // Void type function - returns no values
  void NoData(){
    Console.WriteLine("The function NoReturn()
        does not return any value.");
  }

  // String type function - returns text
  String ReturnData(){
    String msg = "The function ReturnData()
        returns a value of a variable of the
        String type.";
    return msg;
  }

}
```

Result:

```
The function NoReturn() does not return any
value.
The function ReturnData() returns a value of
a variable of the String type.
```

variant

The variable of *variant* type stores data of any type. It also contains a reference to the object representing the assigned value, which is available using the function *Type()*.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // Cleaning the console display
    SystemClass::ClearConsole();

    // Declaration of variant type variables
    variant v1 = "Hello",
            v2 = 100,
            v3 = 2.5;
```

```
    Console.WriteLine("A variable of type " ↵
              + v1.Type().Name() + " = " + v1);

    Console.WriteLine("A variable of type " ↵
              + v2.Type().Name() + " = " + v2);

    Console.WriteLine("A variable of type " ↵
              + v3.Type().Name() + " = " + v3);

  }
}
```

Result:

```
A variable of type String = Hello
A variable of type int = 100
A variable of type double = 2.5
```

bool

The simplest, however quite commonly used data type is the logical. Its determination is the keyword *bool* (an acronym Boolean — logical). One assigns only the value of *true* (1) or *false* (0) o the logical variable type. A variable of this type is most often used to determine whether something is included or disabled, true or false.

Example:

```
class MainClass {

  // Method that changes the value of
  // variable 'b' into false
  void Zmien(bool &b) {
    b = false;
  }

  void Main() {   // Main Class
    // Cleaning the console display
    SystemClass::ClearConsole();
    // Assigning value 'true' to the variable 'b'
    bool b = true;

    Console.WriteLine("Value of the logical ↵
              variable b = " + b);
    Zmien(b);  // Change of the value of the
              // variable 'b'
    Console.WriteLine("Value of the variable ↵
              (b) after the change: " + b);

    // If the condition if true, the message
    // will be displayed
    if (b == 0) {
      Console.WriteLine("Condition 'b = 0' is ↵
              true");
    }
  }
}
```

Result:

```
Value of the logical variable b = 1
Value of the variable (b) after the change: 0
Condition 'b = 0' is true
```

Complex variables

Enum variable type

The enum variable type allows the association of names with numbers. Enum is declared using the keyword *enum* (derived from C language), that automatically numbers the list of constants, giving them a value of: 0, 1, 2, and so on. The enum type declaration recalls the declaration structure.

```
enum type name {constant_1 [= value],...}
```

The default value for the first constant in the enum type is 0. The value of each next constant takes the value 1 greater than the preceding constant. Each constant can be defined with a specified value. If any of the values is not specified, then it will be calculated on the basis of the value of the previous one.

Example:

```
class MainClass {

  // Enum numbers
  enum Numbers {
    // Names of contents
    One,                  // Value = 0
    Two,                  // Value = 1
    Three = 20,           // Value = 20
    Four,                 // Value = 21
    Five = Three + 10     // Value = 30
  }

  void Main(CaseClass c) {
    // Cleaning the console display
    SystemClass::ClearConsole();

    Console.WriteLine("Value = " ↵
                + Numbers::One);
    Console.WriteLine("Value = " ↵
                + Numbers::Two);
    Console.WriteLine("Value = " ↵
                + Numbers::Three);
    Console.WriteLine("Value = " ↵
                + Numbers::Four);
    Console.WriteLine("Value = " ↵
                + Numbers::Five);

  }
}
```

Wynik:

```
Value = 0
Value = 1
Value = 20
Value = 21
Value = 30
```

The name of the enum constant can be obtained by its value. To do this, you must use a static function called *SourceText()*.

Example:

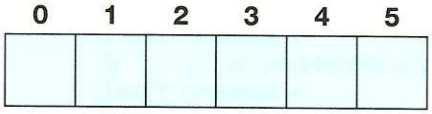
```
class MainClass {
  enum Numbers {
    One,                  // Value = 0
    Two,                  // Value = 1
    Three = 20,           // Value = 20
    Four,                 // Value = 21
    Five = Three + 10     // Value = 30
  }

  void Main(CaseClass c) {
    // Cleaning the console display
    SystemClass::ClearConsole();

    Console.WriteLine("Constant name: " ⏎
              + Numbers::SourceText(0));
    Console.WriteLine("Constant name: " ⏎
              + Numbers::SourceText(1));
    Console.WriteLine("Constant name: " ⏎
              + Numbers::SourceText(20));
    Console.WriteLine("Constant name: " ⏎
              + Numbers::SourceText(21));
    Console.WriteLine("Constant name: " ⏎
              + Numbers::SourceText(30));
  }
}
```

Result:

```
Constant name: One
Constant name: Two
Constant name: Three
Constant name: Four
Constant name: Five
```

Array

An array is a structure consisting of the specified number of elements of the same type. All the elements of an array are placed in your computer's memory together in one place and each element can reference the array name and index, specifying the item number.



An array is created by using the keyword *typedef*, after which the type of data stored in the array is specified. When you create an array, you can declare and initialize a variable of an array type.

```
typedef type_array StringArray;
```

Example:

```
class MainClass {

  // Creation of two arrays
  typedef String[] StringArray;
  typedef int[] IntArray;

void Main(CaseClass c) {

    // Declaration of array variable 'surname'
    StringArray surname(3);
    // Assigning the values
    surname[0] = "Kowalski";
    surname[1] = "Nowak";
    surname[2] = "Mikulski";

    // Declaration of array variable 'number'
    IntArray number(3);
    // Assigning the values
    number[0] = 1;
    number[1] = 2;
    number[2] = 3;
    // Displaying the value of the array
    foreach (String s in surname){
      Console.WriteLine(s);
    }

    Console.WriteLine("number[2] = " + number[2]);

  }
}
```

Result:

```
Kowalski
Nowak
Mikulski
number[2] = 3
```

You can initialize an array at the time of its declaration. After the name of the array there must be curly brackets ({}) indicating the value of the next array elements. On the basis of the number of initial values, the compiler automatically defines the size of the array.

```
Type_table variable_name {index[0], index[1]
...};
```

Example:

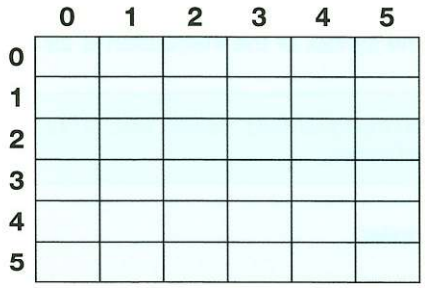
```
class MainClass {

  // Creation of array
  typedef String[] StringArray;

  void Main(CaseClass c) {

    // Declaration of array variable
    // Initialization of variable
    StringArray surname {"Kowalski", "Nowak", ⏎
                         "Mikulski";}

    // Displaying the value of the array
    foreach (String s in surname) {
      Console.WriteLine(s);
    }

  }
}
```

Result:

```
Kowalski
Nowak
Mikulski
```

Multidimensional arrays

In the *EnScript* language one can create an array of much more complexity, for example, a two-dimensional array, which can be graphically illustrated in the following ways:



A two-dimensional array is created using the control statement *foreach* that will be discussed later in this work. A two-dimensional array size is determined by the number of rows and columns.

Example:

```
class MainClass {

  // Creation of array
  typedef int[] IntArray;
  // Creation of array variable
  typedef IntArray[] multiArray;

  void Main(CaseClass c) {
    // Determination of array size
    multiArray m(2);
    // Creation of three arrays, subsequently in
    // every cell of the array 'multiArray'
    // Result: creation of two-dimensional array
    foreach (IntArray a in m) {
      a = new IntArray(2);
    }

  // Initialization of two-dimensional array
    m[0][0] = 1;
    m[0][1] = 2;
    m[1][0] = 3;
    m[1][1] = 4;

    int Index = 0, secondIndex = 0;
    // Displaying the value fo the array
    foreach (IntArray a in m) {
      Index = 0;
      foreach (int i in a)
        Console.WriteLine("\tm[" + Index + "][" ↵
              + secondIndex + "] = " + i);
        secondIndex++;
    }
    Index++;
  }
}
```

Result:

```
m[0][0] = 1
m[0][0] = 2
m[0][1] = 3
m[0][1] = 4
```

Processing instructions

In any language there are programming instructions to control the execution of the program. Instructions are used when making decisions. The need to use instructions results from the fact that encoded algorithms only in simple cases are purely sequential, executed line by line in the order of their occurrence. Quite often the next steps of the algorithm are dependent on the fulfilment of a certain condition or a few of them.

Conditional *if*

The statement *if* allows you to execute the program code only when a specific condition is fulfilled. The action of the above instruction boils down to verifying the condition and if found to be true, the indicated block of code is performed. Due to some modification of the instruction *if*, it is possible to specify commands that will be executed if the condition is not fulfilled. The modified conditional statement is defined as *if ... else*. The simplest form of instruction *if* looks like this:

```
if (condition) instruction_1
   [else instruction_2]
```

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // Declaration of variable 'a'
    int s = 2;

    // Declaration of variable 's'
    String s = "Welcome";

    // If 'a' execute instruction
    if (a)
      Console.WriteLine("The value of ↵
            the variable 'a' is equal to 2");

    // If 's' execute instruction
    if (s)
      Console.WriteLine(s);

    if (a == 3)    // If 'a' = 3
      Console.WriteLine(s);

    else {         // else
      Console.WriteLine("The value of ↵
            the variable 'a' is equal to 3");
    }

  }
}
```

Result:

```
The value of the variable 'a' is equal to 2
Welcome
The value of the variable 'a' is equal to 3
```

The statement *while*

The statement *while* is a block of instructions that are executeduntil the control expression (condition) will return a false value. The syntax of the instruction is as follows:

```
while (condition) statement
```

Example:

```
class MainClass {
  void Main(CaseClass c) {

    int i = 0;     // Declaration of variable 'i'
    while (i < 10) {    // Condition checking

      Console.Write(i + " ");     // Instruction
      i++;

    }
  }
}
```

Result:

```
0 1 2 3 4 5 6 7 8 9
```

The statement *do... while*

The statement *do... while* is a modification of the above mentioned loop *while*. The difference is the time checking of the condition of the loop – in the loop while the condition is checked at the beginning, in the loop *do... while* – at the end. Loop *do... while* performs at least one process. The syntax of the instruction is as follows:

```
do (instruction) while (condition);
```

Example:

```
class MainClass {
  void Main(CaseClass c) {

    int i = 10;    // Declaration of variable 'i'
    do {

      Console.Write(i + " ");     // Instruction
      i--;

    } while (i > 0);     // Condition checking
  }
}
```

Result:

```
10 9 8 7 6 5 4 3 2 1
```

The statement *for*

The statement *for* combines three activities: loop initialization (prior to the first iteration), check the condition and change the values. The first statement is the initialization of the variable controlling the loop. The second statement is a check of the condition. The third statement is an action; usually an increment (++) or decrement is placed here (--) of variable controlling loops. The syntax of the instruction is as follows:

```
for (initialization; condition; step)
    statement;
```

Example:

```
class MainClass {
  void Main(CaseClass c) {

    for (int i = 0; i < 10; i++) {

      Console.Write(i + " ");     // Instruction

    }

  }
}
```

Result:

```
0 1 2 3 4 5 6 7 8 9
```

The statement *foreach*

Statement *foreach* allows sequential viewing the data of different data sets, e.g. arrays, lists. The syntax of the instruction is as follows:

```
foreach (name of the type_element name in
collection) statement
```

Example:

```
class MainClass {

  enum Week {    // Enumerated data type

    Monday = 1;
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday

  }
```

```
  void Main(CaseClass c) {
    foreach (Week d) {      // Foreach loop
      Console.WriteLine(Week::SourceText(d) ⏎
                    + " = " + d);
    }
  }
}
```

Result:

```
Monday = 1
Tuesday = 2
Wednesday = 3
Thursday = 4
Friday = 5
Saturday = 6
Sunday = 7
```

The statement *forall*

The statement *forall* allows sequential viewing of different data sets, similar to *foreach*. The difference between the above instructions occurs during the processing of the data sets in the form of a tree structure, which in a programming language *EnScript* are objects of *NodeClass*. The instruction *forall* will be presented later in the work while discussing the differences in the *EnScript* language defined in versions 6 and 7 of *EnCase*. Syntax of the statement *forall* is as follows:

```
forall (name of the type_element name in
collection) statement
```

Instructions: *break*, *continue*

Inside any structure forming a loop, you can control its course by using *break* and *continue*. Using the statement break allows leaving the loop without executing the rest of the instructions. While the statement *continue* stops the execution of the current iteration, causing the return to the top of the loop to begin new iterations.

Example:

```
class MainClass {
  void Main(CaseClass c) {
    for (int i = 1; i < 11; i++) {
      Console.WriteLine(i);
      if (i == 4) {
        Console.WriteLine("Stops the current ⏎
        iteration and returns to the beginning.");
        continue;
      }
      if (i == 6) {
        Console.WriteLine("The loop was ⏎
              interrupted and further digits: ⏎
              7, 8, 9, 10 will not be displayed.");
        break;
      }
    }
  }
}
```

Result:

```
1
2
3
4
Stops the current iteration and returns to the
beginning.
5
6
The loop was interrupted and further digits: 7,
8, 9, 10 will not be displayed.
```

The statement *switch*

The statement *switch* selects one of the snippets of code, on the basis of the value of the total expression. This instruction computes the value of the specified expression and compares it with the specified values. This statement evaluates only the relationship of equality, it is not possible to use any other relationship. If any checked value is equal to the value of the expression, the program goes to the further instructions and executes all to the end of the block until it encounters a *break*. If no value is equal to the value of the expression, the instructions are executed by keyword *default*. Syntax of the statement *forall* is as follows:

```
switch (expression)
{
  case total_value_1: statement; break;
  case total_value_2: statement; break;
  case total_value_3: statement; break;
  case total_value_4: instruction; break;
  (...)
  default: statement;
}
```

Example:

```
class MainClass {
  void Main(CaseClass c) {

    int number = 4;

    switch(number) {

      case 0: Console.Write("The variable ⏎
          'number' has a value of: 0"); break;
      case 1: Console.Write("The variable ⏎
          'number' has a value of: 1"); break;
      case 2: Console.Write("The variable ⏎
          'number' has a value of: 2"); break;
      case 3: Console.Write("The variable ⏎
          'number' has a value of: 3"); break;
      case 4: Console.Write("The variable ⏎
          'number' has a value of: 4"); break;
    }

  }
}
```

Result:

```
The variable 'number' has a value of: 4
```

Operations on variables

Operations on variables are made using operators. An operator works on one or more of the arguments, and the result of its action is a completely new value. The arguments have a different character than the usual function call, but in both cases the result is the same. Operators in the *EnScript* language can be divided into two groups, based upon the number of arguments on which they operate. We distinguish between *unary*-with one argument and *binary*-with two arguments.

Unary operators

Unary operators in the *EnScript* programming language, will be presented in the form of a table. The rule is that these operators are located before the expression. The exception is the operators "+" and "--", which may also occur after the expression.

Table 2 Summary of unary operators

Operator	Description
New	Creating a new object.
typeof ()	Specification of the type of object.
!	Negates the logic - returns the negated logical value of the expression.
++	Increment - increases the value of the variable.
--	Decrement - lowers the value of the variable.
-	Unary minus, changes the sign of the expression.
~	Negates the bit - returns the variable with negated bits.

Binary operators

Binary operators in the *EnScript* programming language will be presented in the form of a table. Binary operators are placed between the expressions. Binary operators can be divided into operators: arithmetic, bitwise, logical relationships, assignments and range.

Table 3 Range operators

Operator	Description
.	Dereferencing, or shelling, to gain access to a certain value.
::	Is used to access static functions.

Table 4 Summary of arithmetic operators

Operator	Description
+	Add.
-	Subtraction.
*	Multiply.
/	Division.
%	The operator of the remainder.

Table 5 Summary of bitwise operat

Operator	Description
<<	Bit offset to the left.
>>	Bit offset to the right.
&	Bit product.
^	Bit difference.
|	Bit sum.

Table 6 Summary of logical operators

Operator	Description
&&	Logical product.
II	The logical sum.

Table 7 Summary of relational operators

Operator	Description
<	Means: less than ...
>	Means: more than ...
<=	Means: less than or equal to z ...
>=	Means: greater than or equal to z ...
==	Means: equal to z ...
!=	Means: different from ...

Table 8 Summary of assignment operators

Operator	Description
=	Assigns the variable on the left side the value of the expression on right side.
*=	Are used to reduce the write operation, you can save by using arithmetic and bit operators.
/=	
%=	
+=	
-=	
<<=	
>>=	
&=	
^=	
|=	

Object-oriented programming

Each program (script) of *EnCase* consists of one or more classes. In the previous examples, there was only one class called *MainClass*.

```
class MainClass {
  void Main(CaseClass c) {

    Console.WriteLine("Hello World");

  }
}
```

Classes

Classes are descriptions of objects, or programming unites, that can store data and perform tasks as instructed by the programmer. Schematic diagram of a skeleton class is as follows::

```
class class_name {
  //class content.
}
```

An object that was created on the basis of a class is called an *instance*. The object is created by using the operator *new*. The class is as a matrix used to create the object. It instructs the virtual machine how to create an object of a particular type. Each object created on the basis of the class can have unique component values. For example, one can use a class called *dog*, to create several different objects, each of which will be a different breed.

Example:

```
class MainClass {
  void Main(CaseClass c) {
    //Creation of new Dog type objects
    Dog dog_1 =  new Dog();
    Dog dog_2 =  new Dog();
    Dog dog_3 =  new Dog();
    // Components of objects
    // - information on objects
    dog_1.size = 50;
    dog_1.breed = "Husky";
    dog_1.name = "Borys";

    dog_2.size = 40;
    dog_2.breed = "Bokser";
    dog_2.name = "Kolec";

    dog_3.size = 30;
    dog_3.breed = "Beagle";
    dog_3.name = "Rewin";
  }

  // Creation of 'Dog' class
  class Dog {
    int size;
    String breed;
    String name;
  }
}
```

The above example contains three reference variables with the names: *dog_1*, *dog_2* and *dog_3*. Reference variables contain bits representing the way of arriving at a particular object. One can use the dot operator (.), along with a reference variable to access components of the object. Using the dot operator with a variable reference, one can imagine as pressing a button on the remote control steering a specific object.

Functions

Classes in addition to storing data fields also contain functions that perform operations written by the programmer. Functions can modify data and return different values. A function is called by entering its name in the program. At the moment the function call is encountered, the program executes the code of the function. When the function ends, the program returns to the place it was called. The syntax of the function declaration looks as follows:

```
type name (declaration of arguments)
{
instructions;
}
```

Example:

```
class MainClass {

  // Declaration of first function without
  // any argument
  void Display() {
    Console.WriteLine("Hello World");
  }
  // Declaration of second function with
  // an argument
  void DownloadText(String text) {
    Console.WriteLine(text);
  }
  // Declaration of third function
  // with 2 arguments returning
  int Sum(int a, int b){
    int c = a + b;
    return c;
  }

  void Main(CaseClass c){ //Function 'Main'
    // Calling all functions subsequently
    Display();
    DownloadText("The text passed to the ↵
          function as an argument");
    int d = Sum(3,7);
    Console.WriteLine(d);
  }
}
```

Result:

```
Hello World
The text passed to the function as an argument
10
```

Functions also determine the actions that the object is able to perform.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // Creation of an object
    Dog dog_1 =  new Dog();
    // Components of objects - information
    dog_1.size = 50;
    dog_1.breed = "Husky";
    dog_1.name = "Borys";

    dog_1.bark();
  }

  // Creation of 'Dog' class
  class Dog {
    int size;
    String breed;
    String name;

    // Function - behaviour of object
    void bark() {
      Console.WriteLine("Woof! Woof!");
    }
  }
}
```

Result:

```
Woof! Woof!
```

Static functions

Class member functions can be declared as static. These functions can be called even if no object class exists. We refer to the name of the static class through the name of the class and the operator scope (::). The keyword *static* allows calling a function without having to use an object of that class. Static functions allow operations that do not depend on components, and therefore do not require an object.

Example:

```
class MathClass {   // class MathClass
  // Definition of static function
  static long Power(long x) {
    return x * x;
  }
}

class MainClass {
  void Main(CaseClass c) {
    // Calling static function
    Console.WriteLine("5 to the power of 2 = " ↵
              + MathClass::Power(5));
  }
}
```

Wynik:

```
5 to the power of 2 = 25
```

Constructor

A constructor is a special function that is performed when you create the object. A constructor contains code that is executed when you use the *new* operator. In other words, the constructor contains instructions executed when creating a new object. The function being a constructor never returns any result and must have a name in accordance with the name of the class.

```
class class_name {
  class_name(){
    // constructor code
  }
}
```

Each created class has a constructor, even if it is not written by the programmer. In this case, the constructor is created by the compiler. The basic feature of the constructor is that it is executed before the object is associated with the reference. In most cases, a constructor is used to initialize the state of the object, in other words, to determine the value of its components.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // Creation of an object
    Dog dog_1 = new Dog();
    // Display the features of objects
    Console.WriteLine(dog_1.size);
    Console.WriteLine(dog_1.breed);
    Console.WriteLine(dog_1.name);
  }

  // Creation of 'Dog' class
  class Dog {
    int size;
    String breed;
    String name;
    Dog() {  // Constructor

      // Determination of the value of object
      size = 50;
      breed = "Husky";
      name = "Borys";
    }
  }

    void bark() {
      Console.WriteLine("Woof! Woof!");
    }
  }
}
```

Result:

```
50
Husky
Borys
```

Inheritance

Inheritance is one of the foundations of object-oriented programming. It enables efficient and easy to use code, written once as well as the building a hierarchy classes adopting their properties. Inheritance involves the building of new classes based on existing ones. Every such new class adopts the behaviour and properties of the base class. When designing a class using inheritance, one should put the common code in the base class, and then inform the specialized class that the specific shared (more abstract) class is their base class. When one class inherits from another, a child class inherits from the base class. Colloquially it is said that the child class extends the base class. An inheritance relationship means that the child class inherits from the base class. In the *EnScript* language, inheritance is expressed using a colon (:), and the whole schematic definition is as follows:

```
class child_class: base class {
  // inner classes
}
```

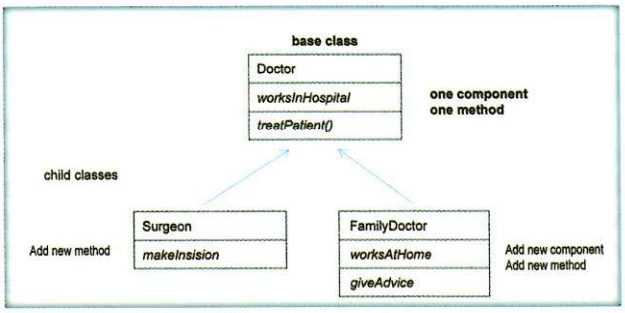


Fig. 1. Example of inheritance.

Example:

```
class MainClass {
  void Main(CaseClass c) {

    // Creation of an object of family doctor
    // Attributes
    FamilyDoctor doctor = new FamilyDoctor();
    doctor.worksAtHome = true;
    doctor.worksInHospital = true;
    Console.WriteLine("Family doctor: ");
    Console.WriteLine("Works at home: " +
                      doctor.worksAtHome);
    Console.WriteLine("Works in hospital: " +
                      doctor.worksAtHospital);
    Console.WriteLine("The doctor's advice: ");
    doctor.giveAdvice();
  }

  // Base class 'Doctor'
  class Doctor {
    bool worksAtHome;
    void treatPatient() {
      Console.WriteLine("Treats a patient");
    }
  }

  // Base classes: FamilyDoctor, Surgeon
  // which inherit from Doctor class
  class FamilyDoctor:Doctor {
    bool worksAtHome;
    void giveAdvice() {
      Console.WriteLine("The patient must
                         take vitamin C");
    }
  }

  class Surgeon:Doctor {
    void makeIncision() {
      Console.WriteLine("Makes an incision");
    }
  }
}
```

Result:

```
Family doctor:
Works at home: 1
Works in hospital: 1
The doctor's advice: The patient must take
vitamin C
```

The graphical user interface (GUI)

By using objects one can create advanced applications, acting on the basis of the graphical user interface. While creating a graphical user interface, remember that it consists precisely of objects: buttons, labels, text boxes, etc. that have their own components and functions. In this article only two examples of a graphical user interface will be presented.

Example:

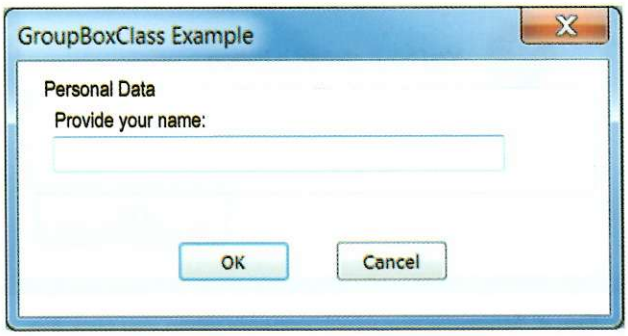


Fig. 2. Sample dialogue box (object).

The source code that displays the above window:

```
class InputDialogClass: DialogClass {
  GroupBoxClass Group;
  StringEditClass StringEdit;
  InputDialogClass(DialogClass parent,String &s):
    DialogClass(parent, "GroupBoxClass Example"),
    Group(this, "Personal Data", START, START,
          250, 40, 0),
    StringEdit(this, "Provide your name:", 15,
          15, 200, 12, 0, s, 255, 0)
      {
      }
}
```

```
class MainClass {
  void Main() {
    String s;
    InputDialogClass diag(null, s);
    diag.Execute();
  }
}
```

The graphical user interface captures events generated by the user. In the *EnCase Forensic* v. 6 program, for capturing, for example, a user clicking a button, the *EventClass* is responsible, and in version 7, the *HandlerClass of the above program* is used.

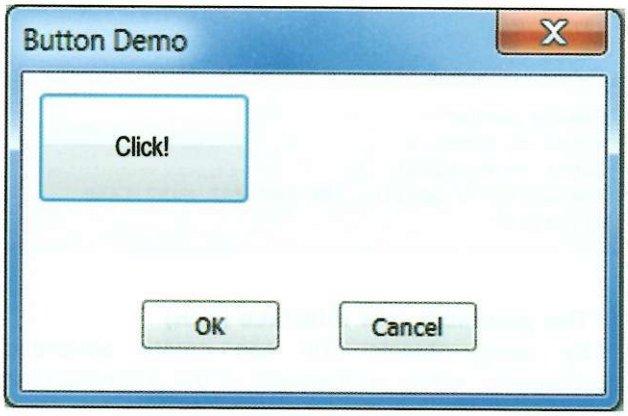


Fig. 3. Sample dialogue window with a button 'Click'.

Fig. 4. Message box which appears when you click the button 'Click'.

The source code for displaying the above window and capturing events:

```
class ButtonDiagClass: DialogClass {

  ButtonClass Button;
  ButtonDialClass(DialogClass parent = null):
    DialogClass(parent, "Button Demo"),
    Button(this, "Click!", START, START,
           75, 30, 0)   // Button
    {

    }

  virtual void ChildEvent(const EventClass
                          &event) {
    // Supports clicking event
    DialogClass::ChildEvent(event);
    if (Button.Matches(event)) {
      // Displays a message
      ErrorMessage("You have clicked a button");
    }
  }
}

class MainClass {
  void Main() {
    ButtonDiagClass bd();
    bd.Execute();
  }
}
```

Linked lists, tree structure

In order to understand how to display and process data in EnCase, it is helpful to understand the structure of a tree and the structure of a linked list.

A linked list is a data structure consisting of elements, designed to connect with each other as needed. The list items are called nodes. Linked lists are available in three versions: singly linked list, doubly linked list and tree.

The tree is the most complex variation of the combined list and the basic data structure used in EnCase, used to present data in a data structure window (A). The tree is a collection of nodes and leaf nodes, with a highlighted principal node (vertex) called a root node and the rest of the nodes (vertices) divided into subsets, which are the main sub-trees of the tree. Each sub-tree has its own root, whichin turn has its own sub-trees, etc., and therefore this node is the root of a sub-tree. A leaf is a node (element) of tree that does not have children, for example, a file or an empty folder. Often the leaves are the nodes farthest from the root.

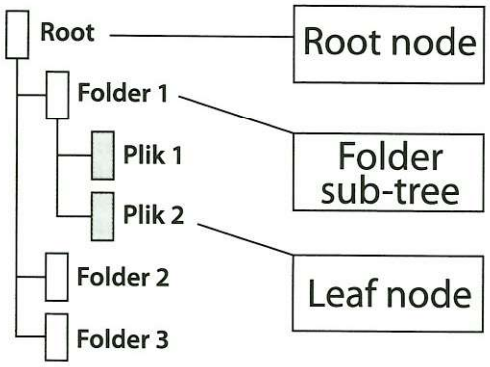


Fig. 5. Tree data structure.

The *NodeClass* is responsible for presenting data in *EnCase* in tree form. Most classes of *EnCase* inherit the components and functions of the above mentioned

class. Selected *NodeClass functions*, i.e.: *Parent(), FirstChild(), LastChild(), Next()*, allow the processing of data in the form of a tree structure.

Example

```
class NodeClass {
  NodeClass(parent = null); // constructor
  NodeClass FirstChild();
  NodeClass LastChild();
  NodeClass Next();
  NodeClass Parent();
}
```

EnCase Forensic v. 6i 7 - fundamental differences in the EnScript language

The *EnScript* programming language is not compatible, which means that scripts written in *EnCase Forensic v. 6* will not work with the latest version of the program. The only solution for *EnCase v. 7* users is rewriting scripts of version 6 in such a way that they operate in the latest version of the program. Before attempting to update the scripts you might want to know the basic differences that occur in the *EnScript* language as defined in both versions of *EnCase Forensic*.

Iteration of data

The primary difference in *EnScript*, as defined in these versions of *EnCase Forensic*, is the method of processing data viewed in the program in the tree structure. In *EnCase v. 6*, in order to process the data, one of two control statements is used: *foreach* or *forall*. The control statement foreach allows you to gain access to all vertices of the root node, and the statement *forall* allows you to gain access to all the elements of a tree structure.

V6

```
class MainClass {
  void Main(CaseClass c) {
    forall (EntryClass entry in c.EntryRoot()) {

      Console.WriteLine(entry.Name());

    }
  }
}
```

In the case of *EnCase Forensic v. 7*, in order to process or display data in a tree structure, use the object type *ItemIteratorClass*..

V7

```
class MainClass {
  void Main(CaseClass c) {

    ItemIteratorClass iter
      (c, ItemIteratorClass::NORECURSE |
          ItemIteratorClass::NOPROXY,
          ItemIteratorClass::ALL);
    while (EntryClass entry = iter.GetNextEntry())
      {
          Console.WriteLine(entry.Name());
      }

  }

}
```

Device-gaining access

Another difference is the way to gain access to the device added to the case. In *EnCase Forensic v.6*, direct access to the device can be made using an object of *DeviceClass*.

V6

```
class MainClass {
  void Main(CaseClass c) {

    foreach (DeviceClass dev in c.DeviceRoot())
    {
      Console.WriteLine(dev.Name());
    }

  }
}
```

In *EnCase Forensic v.7*, the object *EvidenceClass* is contained between objects of *DeviceClass* and *CaseClass*. Examples:

V7

```
class MainClass {
  void Main(CaseClass c) {

    foreach (EvidenceClass ev in c.EvidenceRoot())
    {
      EvidenceOpenClass evOpen();
        if (DeviceClass dev = e.GetDevice
                              (c, evOpen)) {
          Console.WriteLine(dev.Name());
        }
    }

  }

}
```

Iterating through the contents of the device

V6

```
class MainClass {
  void Main(CaseClass c) {

    foreach (DeviceClass dev in c.DeviceRoot()) {
      foreach (EntryClass entry in
               dev.GetRootEntry()) {
        Console.WriteLine(dev.Name());
      }
    }

  }
}
```

V7

```
class MainClass {
  void Main(CaseClass c) {

    foreach (EvidenceClass ev in c.EvidenceRoot())
    {
      EvidenceOpenClass evOpen();
        if (DeviceClass dev = e.GetDevice
                              (c, evOpen)) {
        Console.WriteLine(dev.Name());
        ItemIteratorClass iter(dev);
        while (EntryClass entry =
               iter.GetNextEntry()) {
          Console.WriteLine(entry.Name());
        }
      }
    }

  }
}
```

Gaining access to a file

EnCase scripts allow access to a file selected by the user. In *EnCase v. 6* this functionality was limited solely to items presented by objects of type *EntryClass*, visible in the window *Entries*.

V6

```
class MainClass {
  void Main(CaseClass c) {

  long offset = 0;
  long size = 20;

  if (EntryClass entry = c.GetEntry(offset,size){
      Console.WriteLine(entry.Name());
    }

  }
}
```

In version 7, functionality was largely expanded and it is possible to gain access to the selected object type *EntryClass, BookmarkClass, RecordClass* and so on.

V7

```
class MainClass {
  void Main(CaseClass c) {

    long offset = 0;
    long size = 20;
    if (EntryClass entry = c.GetEntry::TypeCast
         (c.GetCurrentItem(offset,size))) {
      Console.WriteLine(entry.Name());
    }
  }

}
```

Bookmarks - creating folders

The manner of creating folders in *EnCase* was slightly changed. In version 6, one should use the object class *BookmarkFolderClass*, and in version 7 – the object class *BookmarkClass*.

V6

```
class MainClass {
  void Main(CaseClass c) {

  BookmarkFolderClass folder(c.BookmarkRoot(),
                             "Test");

  }
}
```

V7

```
class MainClass {
  void Main(CaseClass c) {

  BookmarkClass folder(c.BookmarkRoot(),
                 "Test", NodeClass::FOLDER);

  }
}
```

Bookmarks-creating notes

V6

```
class MainClass {
  void Main(CaseClass c) {

  BookmarkFolderClass folder(c.BookmarkRoot(),
                             "Test");
  folder.AddNote("Note", 0, 16,
                 BookmarkClass::SHOWREPORT);

  }
}
```

V7

```
class MainClass {
  void Main(CaseClass c) {

  BookmarkClass booknote = c.BookmarkRoot();
  BookmarkClass note(booknote, "Note");
  note.SetComment("Content of note");

  }
}
```

EnScript API

EnCase is distributed along with a rich documentation in the form of an *HTML*file known as *API* (*Application Programming Interface*). This documentation is the best source to get detailed information about the classes, their components and functions. API documentation also contains sample programs (scripts) that are very helpful when creating your own applications. The contents of the EnCase Forensic API can viewed by using the following menu: *Help -> EnScript Help*.

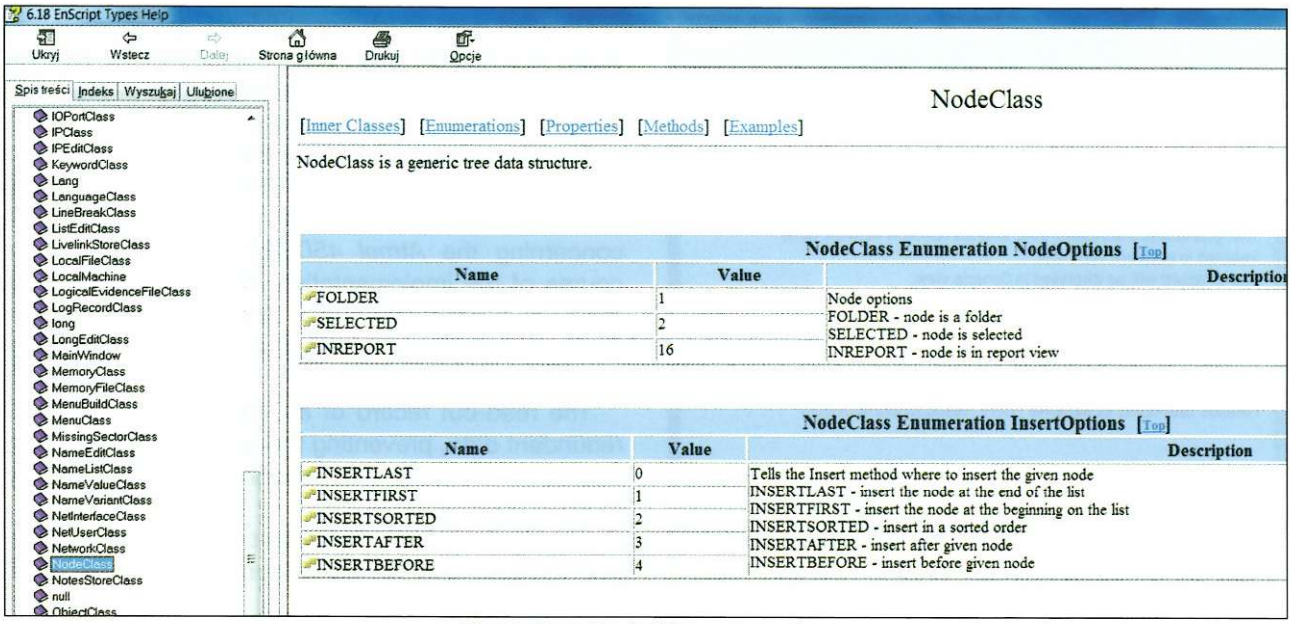


Fig. 6. Enscript API Documentation.

Examples of copyright software solutions

ProgramMAC Address from LNK

The script *MAC Address from LNK* is intended to read the MAC addresses of network adapters, saved in files with the extension *LNK*, otherwise known as short-cuts, created on your hard disk with the file *NTFS* system. Each time a short-cut is created by a user to a file or program on the disk using the *NTFS* file system, this results in saving the MAC address of the network adapter being installed on your computer in the created file. The script takes into account only the files with the extension *LNK*, which were not copied or transferred from other data carrier or partition.

The script turned out to be crucial in the case aiming at, among others, reading the MAC address of the network adapter. The evidence in the above matter consisted of a notebook and a hard drive sent loose. As a result of action of the above mentioned evidential script of hard drive (sent loose) 2 MAC addresses of network adapters were read out.

The *MAC Address from LNK* script is based on data contained in the document *"The Meaning of Link files in Forensic Examination"*[2] by Harry Parsonage. The document contains a detailed description of the analysis of the contents of *LNK* files in a hexadecimal editor, used in the process of creating the script.

When running the script in *EnCase Forensic v. 6*, a dialogue box displays to specify the path and file name in which the results are saved in the script.

In the upper-left corner of the window the button "*Help*" appears. When you click the above button, a window displays containing a brief description of the script.

Unique MAC addresses and the ID of the hard disk read from LNK files, are displayed in the console window of *EnCase Forensic v. 6*.

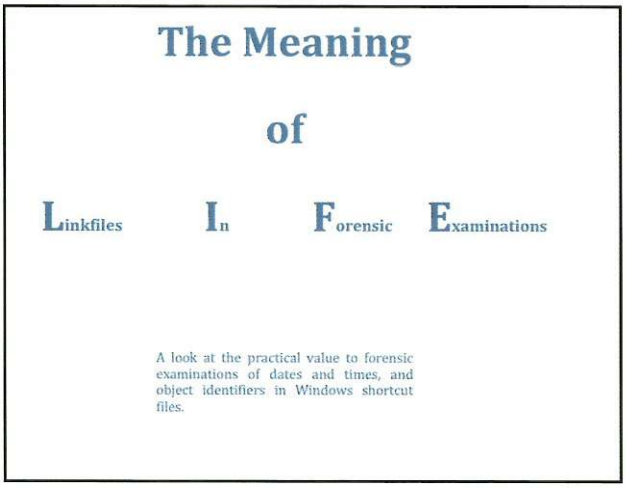


Fig. 7. Document "The Meaning of Linkfiles in Forensic Examinations".

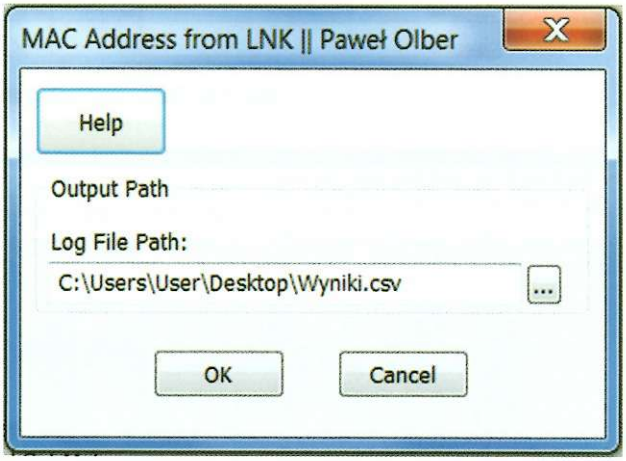


Fig. 8. MAC Address script dialogue window from LNK.

2 http://computerforensics.parsonage.co.uk/downloads/TheMeaningofLIFE.pdf

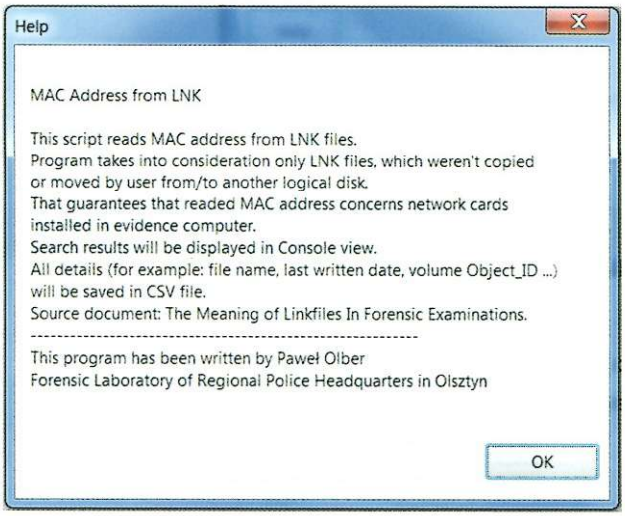


Fig. 9. Help window in script MAC Address from LNK.

Detailed data which are obtained, are saved in the script as a text file *CSV*. The resulting text file with the extension CSV contains, inter alia, a list of files with recovered MAC addresses along with detailed data.

The Redundant Remover Script

The *Redundant Remover* Script was made in the course of forensic research within the forensic evidence concerning the *Atmel 45DB321D* skimmer. In the course of the implementation of the aforementioned research, the contents of the device were read, which had been saved in the form of a binary file containing a sound signal record

The read-out record of an audio signal contained redundant data, preventing the correct analysis of the amplitude of the signal and reading the copied identity data of credit cards.

In order to remove redundant data from the sound file, the contents of the evidential binary file were

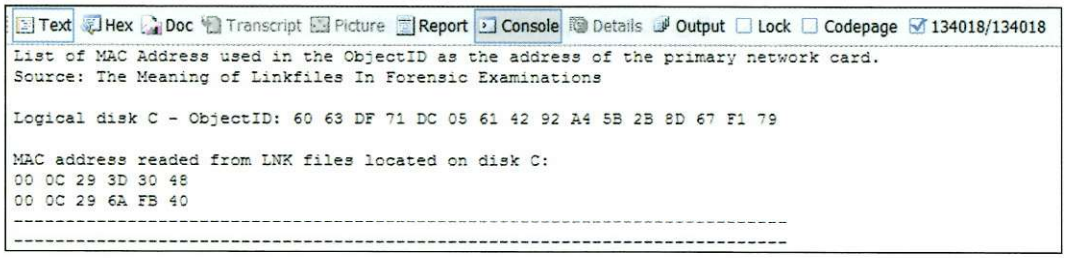


Fig. 10. Console window-information of the script the MAC Address from LNK.

Name	File Ext	Mac Address	Volume Object_ID	Is Deleted
1956 Mercedes-Benz 300SL Gullwing Coupe_3.JPG.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
1956 Mercedes-Benz 300SL Gullwing Coupe_4.LNK	LNK	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
1956 Mercedes-Benz 300SL Gullwing Coupe_4.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
1957 Chevrolet Corvette Convertible _1.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
44.jpg.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
Adv Searching KWL.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
classic cars.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
classic cars.LNK	LNK	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
classic cars.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
Desktop.lnk	lnk	00 0C 29 6A FB 40	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
Desktop.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
Desktop.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No
Desktop.lnk	lnk	00 0C 29 3D 30 48	60 63 DF 71 DC 05 61 42 92 A4 5B 2B 8D 67 F1 79	No

Fig. 11. Data read using the MAC Address from LNK script.

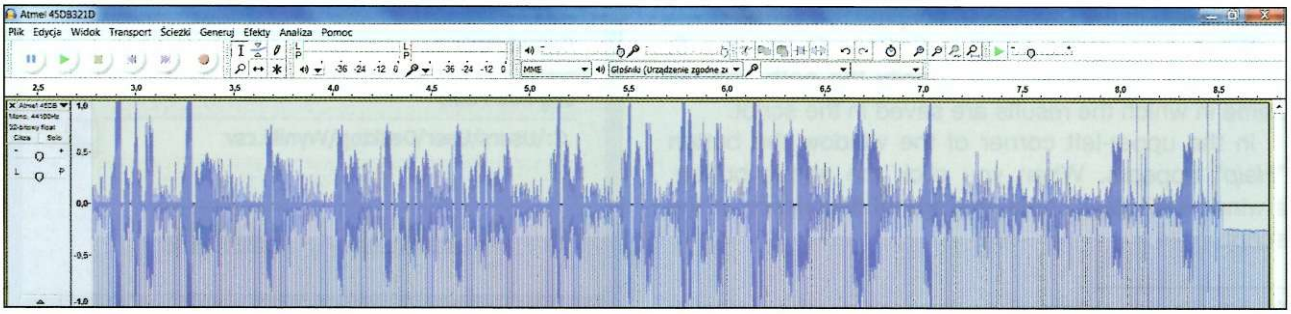


Fig. 12. Amplitude of the audio signal containing redundant data.

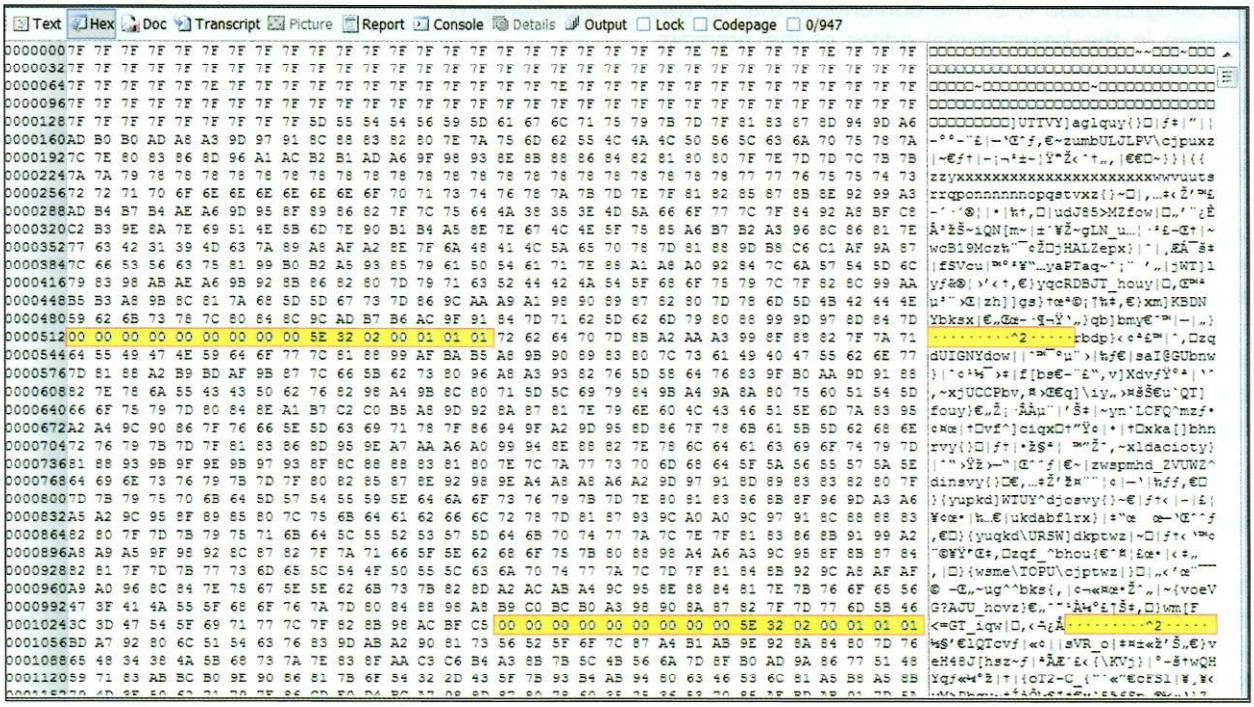


Fig. 13. Contents of the evidential binary file - in hexadecimal notation.

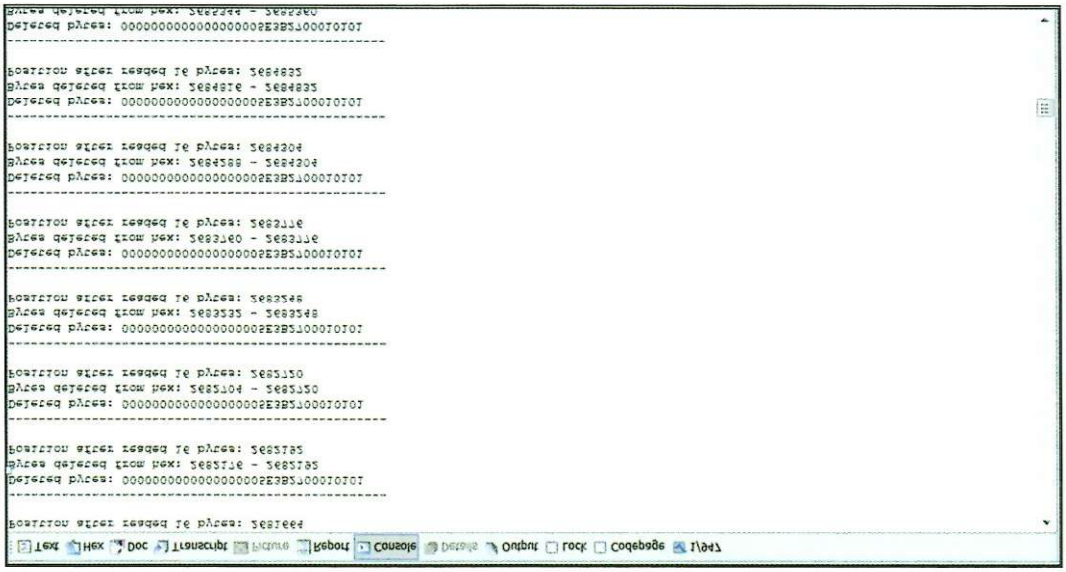


Fig. 14. Console window - information about deleted redundant data.

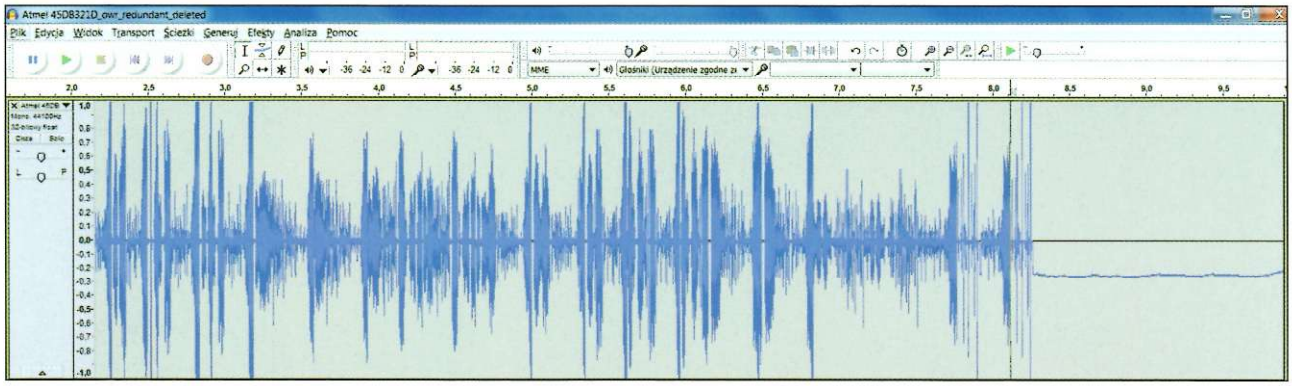


Fig. 15. Amplitude of the audio signal without redundant data.

read using *EnCase v. 6*. The contents of the file was analysed in the hexadecimal editor of the program. It was found that the redundant data were cyclically stored in the form of typical 16 bytes.

Using the script *Redundant Remover* the redundant data was removed from the evidential binary file.

The contents of the file is loaded again using *Audacity 2.0 program* and analysis was performed of the sound wave by assigning to each amplitude binary value which was then converted to decimal values.

As a result of activities carried out in the memory skimmer, intended for reading data from the magnetic strips of credit cards, records containing credit card numbers were detected.

Summary

The purpose of this publication was to provide an overview the programming language of *EnScript*, defined in *EnCase Forensic program* as well as presenting opportunities for its use in forensic research on digital data carriers. The information collected in the course of the study allowed presenting the syntax of the above mentioned programming language in a systematized way. In addition, these syntax elements were backed up with examples. In order to provide a practical way to use scripts in forensic studies of digital data carriers, two scripts were presented, created for the purpose of forensic implementation.

To sum up, it can be concluded that the research taken at work do not cover overall issues related to the EnScript programming language. In view of the difficult nature of the subject, it would be necessary to conduct further studies to create a few applications in EnScript, which can be used, inter alia, in forensic research on digital data carriers. The source code of the created application and detailed descriptions of their creation would be an excellent source of knowledge for those involved in information technology research.

Bibliography

Literary studies

Publisher, year and place of issue

1. Bruce E.: "Thinking in C++. Polish edition", Helion 2002.
2. Bruce E.: "Thinking in Java. Polish edition", Helion 2006.
3. Herbert S.: "Java. Programmer's reference", Helion 2012.
4. Stasiewicz A.: "C++ practical exercises", Helion 2011.

II Instructions:

1. EnCase Forensic Version 6.11 user's Guide
2. Enscript Programs Version 6.3 User Manual
3. EnCase Version 7.05 user's Guide
4. Encase EnScript Language Reference

III. Other sources

1. Enscript API
2. http://www.forensickb.com/2007/09/enscript-tutorial-part-i.html
3. http://codeslack.blogspot.com
4. http://www.geoffblack.com/forensics/
5. http://www.slideshare.net/markmorgan47/enscript-workshop
6. http://encase-forensic-blog.guidancesoftware.com/2014/04/enscript-changes-from-encase-version-6.html

Source

Figs. 1–14: author
Tables 1–8: author

Translation Ronald Scott Henderson