mł. asp. Paweł Olber
biegły z zakresu badań informatycznych Laboratorium Kryminalistycznego Komendy Wojewódzkiej Policji w Olsztynie
pawel.olber@ol.policja.gov.pl

Wykorzystanie skryptów programu EnCase Forensic w kryminalistycznych badaniach cyfrowych nośników danych

Streszczenie

Celem niniejszej pracy jest przedstawienie użytkownikowi programu EnCase Forensic składni języka programowania EnScript, umożliwiającego tworzenie własnych rozwiązań programistycznych, a także przedstawienie możliwości jego wykorzystania w kryminalistycznych badaniach cyfrowych nośników danych. W celu zrozumienia omawianego zagadnienia opis elementów ww. języka programowania poparty został praktycznymi, autorskimi przykładami. Przykłady zamieszczone w niniejszej pracy pozwalają na zrozumienie kodu istniejących już skryptów, zawartych m.in. w tzw. dokumentacji EnCase API. Dokumentacja ta jest najlepszym źródłem pozwalającym na zdobycie szczegółowych informacji o klasach ww. języka programowania, ich składowych i funkcjach. Poza ww. dokumentem, w sieci Internet znajduje się wiele aplikacji tworzonych przez pasjonatów języka programowania EnScript, które ułatwiają pisanie własnych skryptów oraz stanowią cenne źródło informacji i inspiracji. Zaletą poznania ww. języka programowania jest możliwość tworzenia własnych rozwiązań programistycznych, pozwalających na automatyzację realizowanych zadań wykonywanych w ramach przeprowadzanych analiz śledczych, co jest bardzo istotnym elementem w pracy informatyka śledczego.

Słowa kluczowe EnCase Forensic, EnCase API, EnScript

Wprowadzenie i cel pracy

Niniejsza publikacja zawiera omówienie języka programowania EnScript zdefiniowanego w programie EnCase Forensic, który jest jednym z najbardziej cenionych na świecie programów z informatyki śledczej (*computer forensic*). Przeprowadzone badania dotyczące ww. języka programowania, oparte są w szczególności na pracy w programie EnCase Forensic, który ma wbudowane własne środowisko programistyczne oraz zawiera skrypty służące do automatyzacji czasochłonnych analiz śledczych. Środowisko programistyczne ww. programu jest bardzo proste i brakuje w nim wielu przydatnych funkcji służących do formatowania kodu źródłowego. Dlatego też przykładowe skrypty prezentujące składnię języka EnScript napisane zostaną przy użyciu aplikacji Microsoft Visual C++ Express skonfigurowanej w sposób umożliwiający współdziałanie z programem EnCase. Program Microsoft Visual C++ Express pozwala na zachowanie czytelnej i przejrzystej struktury tworzonych skryptów, co ułatwia ich analizę i zrozumienie oraz uniknięcie błędów programistycznych. Informacje dotyczące składni języka programowania EnScript zgromadzone zostaną przede wszystkim w wyniku analizy kodów źródłowych skryptów, zawartych w tzw. dokumentacji API programu EnCase Forensic. Ponadto przeprowadzona zostanie analiza dostępnych w sieci Internet skryptów, które są tworzone przez pasjonatów ww. języka programowania.

W pierwszej części artykułu przedstawione zostaną podstawowe informacje i pojęcia dotyczące języka programowania EnScript. Następnie przedstawiony zostanie prosty program, który będzie punktem wyjścia do tworzenia bardziej skomplikowanych aplikacji. W dalszej kolejności wyjaśnione zostaną elementy składni języka programowania EnScript, w szczególności podstawowe typy danych, zmienne złożone, instrukcje sterujące, operacje na zmiennych, a także zagadnienia dotyczące programowania obiektowego. Na końcu przedstawione zostaną autorskie rozwiązania programistyczne, powstałe na potrzeby realizowanych opinii kryminalistycznych.

Celem starań podjętych przez autora jest omówienie języka programowania EnScript zdefiniowanego w programie EnCase Forensic i przedstawienie jego

możliwości w kryminalistycznych badaniach cyfrowych nośników danych. Informacje zawarte w publikacji powinny być przydatne dla osób zajmującym się informatyką śledczą w tworzeniu autorskich narzędzi analitycznych służących do automatyzacji czasochłonnych czynności wykonywanych w trakcie przeprowadzanych analiz. Przedmiotowa publikacja wydaje się o tyle istotna, że w resorcie brakuje specjalistycznych szkoleń dotyczących omawianego zagadnienia.

EnScript – informacje podstawowe

EnScript jest obiektowym językiem programowania opartym na składni dwóch języków: C++ i Java. Język ten pozwala na automatyzację realizowanych zadań oraz uproszczenie wielu czasochłonnych czynności przeprowadzanych w trakcie wykonywanych analiz śledczych, takich jak np. wyszukiwanie i analizowanie konkretnych typów dokumentów, przetwarzanie oraz ekstrakcja danych zapisanych w różnego rodzaju plikach, tworzenie zbiorów zawierających funkcje skrótu itd. Korzystanie z tego języka programowania umożliwia również przetwarzanie informacji wyselekcjonowanych w wyniku przeprowadzonej analizy danych, które zostały dodane do zakładek (*bookmarks*) lub rekordów (*records*).

EnScript – składnia języka

Rozpoczęcie nauki tworzenia skryptów w języku EnScript wymaga zapoznania użytkownika z podstawowymi zagadnieniami oraz terminologią dotyczącą programowania. W artykule omówione zostaną również: struktura programu, przebieg przetwarzania składowych oraz funkcji programu. Poznanie ww. zagadnień jest bardzo istotne, ponieważ ułatwi użytkownikowi zrozumienie kodu źródłowego tworzonych przez niego aplikacji oraz pozwoli na skuteczną analizę istniejących już skryptów, m.in. domyślnie zdefiniowanych w programie EnCase oraz dostępnych w sieci Internet.

Białe znaki

Białe znaki, tj. spacje, tabulatory oraz znaczniki nowej linii, nie mają znaczenia w kodzie źródłowym i są ignorowane przez kompilator – program wykonujący kompilację, czyli tłumaczenie kodu źródłowego na język maszynowy.

Komentarze

Dowolny fragment kodu źródłowego programu może być objęty komentarzem. Komentarz jest całkowicie ignorowany przez kompilator, natomiast może być przydatny dla piszącego i czytającego kod. Komentarze stosowane są w celu opisu, wyjaśnienia pewnych fragmentów kodu programu, oddzielenia jednej części kodu od drugiej oraz oznaczania funkcji. W języku EnScript istnieją dwa dostępne sposoby komentowania kodu: liniowe oraz blokowe.

Przykład:

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

    // Przykład komentarza liniowego.

    /*
      Przykład komentarza blokowego,
      którym można objąć wiele linii.
    */

  }
}
```

Dyrektywa *include*

W języku EnScript zapis *include* to tzw. dyrektywa – informacja o dołączeniu do kompilowanego pliku źródłowego zawartości wskazanego pliku o rozszerzeniu *enscript*, określanego jako biblioteka. Biblioteki zawierają funkcje, z których można korzystać w pliku źródłowym. Nazwę pliku podaje się bez rozszerzenia.

Przykład:

```
  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"
```

Dyrektywa *typelib*

W języku EnScript zapis *typelib* umożliwia dołączenie do kompilowanego pliku źródłowego zawartości wskazanego pliku, tzw. biblioteki pochodzącej z zewnętrznej aplikacji, np. programów z pakietu Microsoft Office.

Przykład:

```
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);
    }
}
```

Konsola

Konsola jest w rzeczywistości plikiem, którego zawartość wyświetlana jest w oknie podglądu programu EnCase Forensic. Konsola dostępna jest po wciśnięciu zakładki „Console" (dot. V6) lub za pomocą menu: *View -> Console* (dot. V7) i umożliwia wyświetlanie wszelkiego rodzaju komunikatów oraz innych treści, np. informacji o przebiegu działania programu. W celu wyświetlenia komunikatu w konsoli należy posłużyć się poleceniem *WriteLine()*.

Pierwszy program

Tradycyjne książki o programowaniu zaczynają naukę programowania od przedstawienia aplikacji wyświetlającej w konsoli lub na ekranie monitora zdanie: „*Hello World*". Tradycja ta zostanie zachowana także i w niniejszej publikacji. Pierwszy program będzie punktem wyjścia do bardziej złożonych aplikacji, które pozwolą zrozumieć składnię języka EnScript.

Przykład:

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

    Console.WriteLine("Hello World");

  }
}
```

Wynik:

```
Hello World
```

Główna klasa programu

Program napisany w języku EnScript jest zbiorem zmiennych, definicji i wywołań funkcji. Każdy skrypt utworzony w programie EnCase ma klasę o nazwie *MainClass*, wewnątrz której znajduje się funkcja *Main*, przyjmująca jako parametr zmienną typu *CaseClass*, która jest odpowiednikiem sprawy utworzonej przez użytkownika programu. Wewnątrz klasy głównej *MainClass* definiowane są pozostałe klasy i funkcje. Po uruchomieniu skryptu program EnCase lokalizuje obiekt typu *MainClass* i wywołuje jego konstruktor. Zagadnienia dotyczące konstruktorów przedstawione zostaną w dalszej części publikacji. Następnie ww. obiekt wywołuje funkcję *Main*, która poprzedzona jest słowem kluczowym *void*. Oznacza to, że funkcja ta nie zwraca żadnych danych. Na koniec program EnCase uruchamia destruktor klasy *MainClass* i ostatecznie usuwa obiekt klasy głównej. W tym miejscu program kończy działanie.

Pojęcie zmiennej i podstawowe typy danych

Zmienna (*variable*) to miejsce w pamięci operacyjnej komputera przechowujące pojedynczą wartość określonego typu. Każda zmienna ma nazwę, dzięki której można się do niej odwoływać. Przed pierwszym użyciem zmienną należy zadeklarować, a więc poinformować kompilator, że pod taką nazwą kryje się zmienna danego typu. Składnia deklaracji zmiennej wygląda następująco:

```
typ nazwa_zmiennej = wartość_zmiennej;
```

Typ określa rodzaj informacji, jakie można przechowywać w zmiennej. Mogą to być np. liczby całkowite, rzeczywiste, tekst (czyli łańcuchy znaków, *strings*). Nazwa zmiennej powinna dobrze określać jej znaczenie i spełniać pewne kryteria. Oto najważniejsze z nich:

- nazwa zmiennej powinna być słowem lub skrótem, określającym jej przeznaczenie
- nazwa może się składać z małych i wielkich liter, a także cyfr i znaku podkreślenia. Należy pamiętać, że nazwa zmiennej nie może zaczynać się od cyfry. W odróżnieniu od innych języków programowania w nazwie zmiennej można używać znaków narodowych, np. liter *ć* czy *ź*.

Język EnScript obsługuje różne typy danych natywnie, co oznacza, że konwersje pomiędzy różnymi typami danych wykonywane są automatycznie oraz użycie danej określonego typu dopuszczalne jest w każdym kontekście.

Przykład:

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

    // deklaracja zmiennej "i"
    // przypisanie wartości 1 do zmiennej
    int i = 1;

    // wyświetlenie zawartości zmiennej "i"
    Console.WriteLine("i = " + i);

    // dodanie wartości 1 do uprzednio
    // zdefiniowanej zmiennej
    i = i + 1;

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

  }
}
```

Wynik:

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

Podstawowe typy danych

Typy danych określają sposób użycia pamięci w programie. Określając typ danych, przekazuje się kompilatorowi informację, jak utworzyć określony fragment pamięci, a także w jaki sposób wykonać na nim operacje. Dane różnych typów różnią się szybkością działania, rozmiarem zajmowanej pamięci i oczywiście funkcjonalnością, np. są dane, które wolno mnożyć przez siebie, ale i takie, dla których operacja mnożenia nie istnieje – choćby teksty.

char

Zmienna typu *char* przechowuje jeden znak. Do zmiennej tego typu można przypisać znaki UNICODE o kodach od 0 do 65535. Zmienne typu *char* mogą być przypisane do zmiennych typu całkowitego *int* i odwrotnie, zmienne typu całkowitego mogą być przypisane do zmiennych znakowych. W przypadku takiego zabiegu EnScript dokonuje konwersji znaków zgodnie z kodowaniem ASCII. W kodzie ASCII mała litera „a" ma przypisaną wartość 97. Gdy do zmiennej typu *char* zostaje przypisany znak, to w rzeczywistości jest on liczbą z zakresu od 0 do 255. Należy jednak zdawać sobie sprawę z różnicy pomiędzy wartością a znakiem, np.: 5 i „5" nie jest tym samym, ponieważ 5 jest wartością, natomiast „5" jest znakiem i ma wartość 53, tak jak w przypadku litery „a", która ma wartość 97.

Przykład:

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

    // Przypisanie liczby 82 do zmiennej int
    int i = 82;

    // Wyświetlenie zawartości zmiennej 'i'
    Console.WriteLine("i = " + i);

    // Przypisanie zmiennej 'i' do zmiennej
    // typu char
    char a = i;

    // Wyświetlenie zawartości zmiennej 'a'
    Console.WriteLine("a = " + a);

    // Przypisanie znaku do zmiennej typu char
    char b = 'v';

    // Wyświetlenie zawartości zmiennej 'b'
    Console.WriteLine("b = " + b);

    // Przypisanie zmiennej 'b' do zmiennej 'j'
    int j = b;

    // Wyświetlenie zawartości zmiennej 'j'
    Console.WriteLine("j = " + j);

  }
}
```

Wynik:

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

String

Zmienna typu *String* przechowuje ciągi znaków (tekst), na których można wykonywać wiele operacji. Ciągi znaków mogą być ze sobą łączone za pomocą operatora „+".

Przykład:

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

    // Usunięcie zawartości konsoli
    SystemClass::ClearConsole();
    // Przypisanie wartości do zmiennej s1
    String s1 = "Hello";
    // Przypisanie wartości do zmiennej s2
    String s2 = " World";

    // Zmienna s1 uzyskuje wartość "Hello World"
    s1 = s1 + s2;
    // Zmienna s1 uzyskuje wartość "Hello World."
    s1 = s1 + ".";
    // Przejście do nowego wiersza
    s1 = s1 + '\n';

    // Wyświetlenie wyników konsoli
    Console.Write(s1);
  }
}
```

Wynik:

```
Hello World.
```

Ciągi znaków mogą zawierać znaki specjalne. W celu wykorzystania znaków specjalnych w tekście należy posłużyć się znakiem ukośnika (*backslash*). Do grupy znaków specjalnych zaliczamy: tabulator, ukośnik, pojedynczy cudzysłów, znak końca linii.

Przykład:

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

    SystemClass::ClearConsole();

    String s1 = "Tabulator \t, ukośnik \\, ↵
                pojedynczy cudzysłów \",   ↵
                znak końca linii. \n";

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

Wynik:

```
Tabulator    , ukośnik \, pojedynczy cudzysłów
", znak końca linii
```

Istnieje również możliwość umieszczenia w tekście dodatkowych znaków specjalnych z tablicy UNICODE, np. znaku towarowego ®. Znak ten należy zapisać z ukośnikiem w postaci szesnastkowej[1].

1 Tablica UNICODE dostępna jest m.in. pod adresem: *http://unicode-table.com/en/.*

Przykład:

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

    SystemClass::ClearConsole();

    String s1 = "To jest znak towarowy: \x00AE";
    Console.Write(s1);

    String s2 = "Litera alfabetu rosyjskiego: ↵
                \x0414";
    Console.Write(s2);

  }
}
```

Wynik:

```
To jest znak towarowy: ®
Litera alfabetu rosyjskiego: Д
```

Ponieważ wszystkie znaki w cudzysłowie są znaczące, w tym również białe znaki, nie można wstawić nowego wiersza w środku łańcucha.

Przykład:

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

    SystemClass::ClearConsole();

    // Nieprawidłowy sposób

    s1 = "To jest bardzo długi tekst,
    którego nie można zapisywać w ten sposób.
    Próba uruchomienia skryptu zakończy się
    błędem";

  }
}
```

Aby połączyć bardzo długie ciągi znaków, wystarczy umieścić je w cudzysłowie, jeden pod drugim. Ciągi znaków zostaną połączone w czasie kompilacji programu. W przypadku bardzo długich ciągów znaków nie należy używać operatora łączenia „+", ponieważ spowoduje to niepotrzebne wydłużenie czasu kompilacji programu.

Przykład:

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

    SystemClass::ClearConsole();

    // Prawidłowe połączenie bardzo długich znaków
    s1 = "To jest bardzo długi tekst, "
    "który został zapisany prawidłowo. \n"
    "Uruchomienie skryptu zakończy się sukcesem.";

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

Wynik:

```
To jest bardzo długi tekst, który został
zapisany prawidłowo.
Uruchomienie skryptu zakończy się sukcesem.
```

Ciągi znaków mogą być traktowane jako tablica znaków i adresowane indywidualnie. **Uwaga:** Indeksowanie tablicy odbywa się zawsze od 0. Ostatni element ma indeks równy rozmiarowi tablicy – 1.

Przykład:

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

    SystemClass::ClearConsole();

    String s = "1234567890";
    Console.WriteLine("Pierwotny ciąg: " + s);

    // Wyświetlenie liczby 1
    Console.WriteLine("Pierwszy znak ciągu: " + ↵
                      s[0] = 'X';

    // Pusta linia
    Console.WriteLine(" ");

    // Wstawienie znaku 'X' na pierwszą pozycję
    s[0] = 'X';
    // s = X234567890

    Console.WriteLine("Nowy ciąg: " + s);
    Console.WriteLine("Pierwotny ciąg: " + s);

  }
}
```

Wynik:

```
Pierwotny ciąg: 1234567890
Pierwszy znak ciągu: 1

Nowy ciąg: X234567890
Pierwszy znak ciągu: X
```

Numeryczne typy danych

W języku EnScript dostępnych jest siedem typów numerycznych. Typy numeryczne mają stały rozmiar, niezależnie od procesora i platformy. Część typów numerycznych ma podobne nazwy, różnią się one jedynie pierwszą literą *u*. W języku EnScript jest to tak zwany modyfikator, przy użyciu którego można nieco wpływać na zakres danych, które można zapisywać w zmiennych. Modyfikator to nic innego jak określenie różnych odmian jednego typu. Modyfikator *u* określa, że dany typ reprezentuje jedynie dane dodatnie. Wszystkie numeryczne typy danych języka EnScript zostały przedstawione w postaci tabelarycznej i posortowane wg rozmiaru danych, jakie mogą przechowywać (tab. 1).

Tabela 1 Zestawienie numerycznych typów danych

Typ	Rozmiar (B)	Min. wartość	Max. wartość
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

Przykład:

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

    // Usunięcie zawartości konsoli
    SystemClass::ClearConsole();

    // Zmienna typu short
    short short_min_value = -32768;
    short short_max_value = 32767;

    Console.WriteLine("Min. wart. zmiennej typu ↵
        'short' wynosi: " + short_min_value);
    Console.WriteLine("Max. wart. zmiennej typu ↵
        'short' wynosi: " + short_max_value);
    Console.WriteLine("------------------------");

    // Zmienna typu ushort
    ushort ushort_min_value = 0;
    ushort ushort_max_value = 65535;

    Console.WriteLine("Min. wart. zmiennej typu ↵
        'ushort' wynosi: " + ushort_min_value);
    Console.WriteLine("Max. wart. zmiennej typu ↵
        'ushort' wynosi: " + ushort_max_value);
    Console.WriteLine("------------------------");

    // Zmienna typu int
    int int_min_value = -2147483648;
    int int_max_value = 2147483647;

    Console.WriteLine("Min. wart. zmiennej typu ↵
        'int' wynosi: " + int_min_value);
    Console.WriteLine("Max. wart. zmiennej typu ↵
        'int' wynosi: " + int_max_value);
    Console.WriteLine("------------------------");

    // Zmienna typu unit
    unit unit_min_value = 0;
    unit unit_max_value = 4294967295;

    Console.WriteLine("Min. wart. zmiennej typu ↵
        'unit' wynosi: " + unit_min_value);
    Console.WriteLine("Max. wart. zmiennej typu ↵
        'unit' wynosi: " + unit_max_value);
    Console.WriteLine("------------------------");

    Console.WriteLine(s1);

  }
}
```

Wynik:

```
Min. wart. zmiennej typu 'short' wynosi: -32768
Max. wart. zmiennej typu 'short' wynosi: 32767
---------------------------------------------
Min. wart. zmiennej typu 'ushort' wynosi: 0
Max. wart. zmiennej typu 'ushort' wynosi: 65535
---------------------------------------------
Min. wart. zmiennej typu 'int' wynosi:
-2147483648
Max. wart. zmiennej typu 'int' wynosi:
2147483647
---------------------------------------------
Min. wart. zmiennej typu 'uint' wynosi: 0
Max. wart. zmiennej typu 'uint' wynosi:
4294967295
---------------------------------------------
```

void

Ten specjalny, tzw. pusty, typ danych używany jest do wskazania, że funkcja nie zwraca żadnego wyniku. Przykład zastosowania ww. typu został również przedstawiony podczas omawiania funkcji.

Przykład:

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

    // Usunięcie zawartości konsoli
    SystemClass::ClearConsole();

    // Wywołanie funkcji NoData();
    NoData();

    // Wywołanie funkcji ReturnData()
    // i przypisanie zwróconej wartości do
    // zmiennej 'text'

    String test = ReturnData();
    Console.WriteLine(text);

  }

  // Funkcja typu void - nie zwraca żadnych
  // wartości
  void NoData(){
    Console.WriteLine("Funkcja NoReturn() nie ↵
        zwraca żadnej wartości.");
  }
```

```
  // Funkcja typu String - zwraca tekst
  String ReturnData(){
    String msg = "Funkcja ReturnData() zwraca
        wartość zmiennej typu String.";
    return msg;
  }

}
```

Wynik:

```
Funkcja NoReturn() nie zwraca żadnej wartości.
Funkcja ReturnData() zwraca wartość zmiennej
typu String.
```

variant

Zmienna typu *variant* przechowuje dane dowolnego typu. Zawiera również odniesienie do obiektu reprezentującego przypisaną wartość, który jest dostępny za pomocą funkcji *Type()*.

Przykład:

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

    // Usunięcie zawartości konsoli
    SystemClass::ClearConsole();

    // Deklaracja zmiennych typu variant
    variant v1 = "Witaj",
            v2 = 100,
            v3 = 2.5;

    Console.WriteLine("Zmienna typu "
              + v1.Type().Name() + " = " + v1);

    Console.WriteLine("Zmienna typu "
              + v2.Type().Name() + " = " + v2);

    Console.WriteLine("Zmienna typu "
              + v3.Type().Name() + " = " + v3);

  }
}
```

Wynik:

```
Zmienna typu String = Witaj
Zmienna typu int = 100
Zmienna typu double = 2,5
```

bool

Najprostszym, ale bardzo często używanym typem danych jest typ logiczny. Do jego określenia służy słowo kluczowe *bool* (skrót od Boolean — boole'owski, logiczny). Do zmiennej typu logicznego możemy przypisać wyłącznie wartość *true* (1) lub *false* (0). Zmienną tego typu najczęściej stosuje się do określenia, czy jest coś włączone czy wyłączone, prawdziwe lub fałszywe.

Przykład:

```
class MainClass {

  // Metoda, która zmienia wartość
  // zmiennej 'b' na false

  void Zmien(bool &b) {
    b = false;
  }

  void Main() {   // Klasa Main

    // Usunięcie zawartości konsoli
    SystemClass::ClearConsole();

    // Przypisanie wartości 'true' do zmiennej b
    bool b = true;

    Console.WriteLine("Wartość zmniennej
              logicznej b = " + b);

    Zmien(b);  // Zmiana wartości zmiennej 'b'

    Console.WriteLine("Wartość zmniennej b
              po zmianie: " + b);

    // Jeżeli warunek jest prawdziwy zostanie
    // wyświetlony komunikat
    if (b == 0) {

      Console.WriteLine("Warunek 'b = 0' jest
              prawdziwy");

    }
  }
}
```

Wynik:

```
Wartość zmiennej logicznej b = 1
Wartość zmiennej b po zmianie: 0
Warunek 'b = 0' jest prawdziwy
```

Zmienne złożone

Typ wyliczeniowy

Wyliczeniowy typ danych umożliwia powiązanie nazw z liczbami. Wyliczenie deklaruje się za pomocą słowa kluczowego *enum* (pochodzącego z języka C), które automatycznie numeruje listę stałych, nadając im wartości: 0, 1, 2 itd. Deklaracja typu wyliczeniowego przypomina deklarację struktury.

```
enum nazwa_typu { stała_1 [ = wartość], …}
```

Domyślna wartość pierwszej stałej w zbiorze typu wyliczeniowego wynosi 0. Wartość każdej następnej stałej przyjmuje wartość większą o 1 od wartości stałej poprzedzającej. Każda ze stałych może zostać zdefiniowana z określoną wartością. Jeżeli któraś z wartości nie zostanie podana, to zostanie ona wyliczona na podstawie wartości poprzedniej.

Przykład:

```
class MainClass {

  // Deklaracja typu wyliczeniowego
  enum Liczby {
    // Nazwy stałych
    Jeden,              // Wartość = 0
    Dwa,                // Wartość = 1
    Trzy = 20,          // Wartość = 20
    Cztery,             // Wartość = 21
    Pięć = Trzy + 10    // Wartość = 30
  }

  void Main(CaseClass c) {

    // Usunięcie zawartości konsoli
    SystemClass::ClearConsole();

    Console.WriteLine("Wartość = " ⏎
                  + Liczby::Jeden);
    Console.WriteLine("Wartość = " ⏎
                  + Liczby::Dwa);
    Console.WriteLine("Wartość = " ⏎
                  + Liczby::Trzy);
    Console.WriteLine("Wartość = " ⏎
                  + Liczby::Cztery);
    Console.WriteLine("Wartość = " ⏎
                  + Liczby::Pięć);

  }
}
```

Wynik:

```
Wartość = 0
Wartość = 1
Wartość = 20
Wartość = 21
Wartość = 30
```

Nazwę stałej typu wyliczeniowego można uzyskać poprzez jej wartość. W tym celu należy posłużyć się statyczną funkcją o nazwie *SourceText()*.

Przykład:

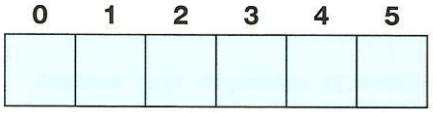
```
class MainClass {

  enum Liczby {
    Jeden,              // Wartość = 0
    Dwa,                // Wartość = 1
    Trzy = 20,          // Wartość = 20
    Cztery,             // Wartość = 21
    Pięć = Trzy + 10    // Wartość = 30
  }

  void Main(CaseClass c) {

    // Usunięcie zawartości konsoli
    SystemClass::ClearConsole();

    Console.WriteLine("Nazwa stałej: " ⏎
              + Liczby::SourceText(0));
    Console.WriteLine("Nazwa stałej: " ⏎
              + Liczby::SourceText(1));
    Console.WriteLine("Nazwa stałej: " ⏎
              + Liczby::SourceText(20));
    Console.WriteLine("Nazwa stałej: " ⏎
              + Liczby::SourceText(21));
    Console.WriteLine("Nazwa stałej: " ⏎
              + Liczby::SourceText(30));

  }
}
```

Wynik:

```
Nazwa stałej: Jeden
Nazwa stałej: Dwa
Nazwa stałej: Trzy
Nazwa stałej: Cztery
Nazwa stałej: Pięć
```

Tablice

Tablica jest strukturą złożoną z określonej liczby elementów tego samego typu. Wszystkie elementy tablicy umieszczane są w pamięci komputera razem w jednym miejscu i do każdego elementu można odwołać się za pomocą nazwy tablicy oraz indeksu określającego numer danego elementu.



Tablicę tworzy się za pomocą słowa kluczowego ***typedef***, po którym określa się rodzaj danych przechowywanych w tablicy. Po utworzeniu tablicy można zadeklarować oraz zainicjować zmienną typu tablicowego.

```
typedef typ_tablicy StringArray;
```

Przykład:

```
class MainClass {

  // Utworzenie dwóch tablic
  typedef String[] StringArray;
  typedef int[] IntArray;

  void Main(CaseClass c) {

    // Deklaracja zmniennej tablicowej 'nazwisko'
    StringArray nazwisko(3);
    // Przypisanie wartości
    nazwisko[0] = "Kowalski";
    nazwisko[1] = "Nowak";
    nazwisko[2] = "Mikulski";

    // Deklaracja zmniennej tablicowej 'liczba'
    IntArray liczba(3);
    // Przypisanie wartości
    liczba[0] = 1;
    liczba[1] = 2;
    liczba[2] = 3;
```

```
    // Wyświetlenie zawartości tablicy
    foreach (String s in nazwisko){
      Console.WriteLine(s);
    }

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

  }
}
```

Wynik:

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

Istnieje możliwość inicjalizacji tablicy w momencie jej deklaracji. Po nazwie tablicy należy w nawiasach klamrowych ({}) podać wartość kolejnych elementów tablicy. Na podstawie liczby wartości inicjalizujących kompilator automatycznie zdefiniuje rozmiar tablicy.

```
typ_tablicy nazwa_zmiennej {index[0], index[1], …};
```

Przykład:

```
class MainClass {

  // Utworzenie tablicy
  typedef String[] StringArray;

  void Main(CaseClass c) {

    // Deklaracja zmniennej tablicowej
    // Inicjalizacja zmniennej
    StringArray nazwisko {"Kowalski", "Nowak",
                          "Mikulski";}

    // Wyświetlenie zawartości tablicy
    foreach (String s in nazwisko) {
      Console.WriteLine(s);
    }

  }
}
```

Wynik:

```
Kowalski
Nowak
Mikulski
```

Tablice wielowymiarowe

W języku EnScript można utworzyć tablicę znacznie bardziej złożoną – na przykład tablicę dwuwymiarową, którą można graficznie zobrazować w następujący sposób:

	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

Tablica dwuwymiarowa zostanie utworzona za pomocą instrukcji sterującej *foreach*, która zostanie omówiona w dalszej części pracy. Rozmiar tablicy dwuwymiarowej określany jest przez liczbę wierszy i kolumn.

Przykład:

```
class MainClass {

  // Utworzenie tablicy
  typedef int[] IntArray;
  // Utworzenie zmiennej tablicowej 'multiArray'
  typedef IntArray[] multiArray;

  void Main(CaseClass c) {

    // Określenie rozmiaru tabeli
    multiArray m(2);

    // Utworzenie trzech tabel, kolejno w każdej
    // komórce tabeli 'multiArray'
    // Wynik: powstanie tablicy dwuwymiarowej
    foreach (IntArray a in m) {
      a = new IntArray(2);
    }

    // Inicjalizacja tabeli dwuwymiarowej
    m[0][0] = 1;
    m[0][1] = 2;
    m[1][0] = 3;
    m[1][1] = 4;

    int Index = 0,
        drugiIndex = 0;

    // Wyświetlenie zawartości tabeli
    foreach (IntArray a in m) {
      Index = 0;
      foreach (int i in a)
        Console.WriteLine("\tm[" + Index + "][" +
              + drugiIndex + "] = " + i);

        drugiIndex++;
    }
    Index++;
  }
}
```

Wynik:

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

Instrukcje sterujące

W każdym języku programowania istnieją instrukcje pozwalające na sterowanie wykonywaniem programu. Instrukcje wykorzystuje się przy podejmowaniu decyzji. Konieczność stosowania instrukcji wynika stąd, że kodowane algorytmy tylko w prostych przypadkach są czysto sekwencyjne, wykonywane linia po linii w kolejności ich występowania. Bardzo często kolejne kroki algorytmu są zależne od spełnienia pewnego warunku lub kilku warunków.

Instrukcja warunkowa *if*

Instrukcja *if* pozwala wykonać kod programu tylko wtedy, gdy spełniony jest określony warunek. Działanie ww. instrukcji sprowadza się więc do sprawdzenia warunku i, jeśli zostanie stwierdzona jego prawdziwość, wykonania wskazanego bloku kodu. Dzięki pewnej modyfikacji instrukcji *if* możliwe jest określenie poleceń, które zostaną wykonane, gdy warunek nie zostanie spełniony. Taką zmodyfikowaną instrukcję warunkową określa się jako *if... else*. Najprostsza forma instrukcji *if* wygląda następująco:

```
if (warunek) instrukcja_1 [else instrukcja_2]
```

Przykład:

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

    // Deklaracja zmiennej 'a'
    int s = 2;
    // Deklaracja zmiennej 's'
    String s = "Witaj";

    // Jeśli 'a' wykonaj instrukcję
    if (a)
      Console.WriteLine("Wartość zmiennej
            'a' równa się 2");

    // Jeśli 's' wykonaj instrukcję
    if (s)
      Console.WriteLine(s);

    if (a == 3)    // Jeśli 'a' = 3
      Console.WriteLine(s);
    else {         // w przeciwnym razie
      Console.WriteLine("Wartość zmiennej
            'a' jest różna od 3");
    }
  }
}
```

Wynik:

```
Wartość zmiennej 'a' równa się 2
Witaj
Wartość zmienna 'a' jest różna od 3
```

Instrukcja *while*

Instrukcja *while* jest blokiem instrukcji, które wykonywane są aż do momentu, gdy wyrażenie kontrolne (warunek) zwróci wartość *false*. Składnia ww. instrukcji jest następująca:

```
while (warunek) instrukcja
```

Przykład:

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

    int i = 0;       // Deklaracja zmiennej 'i'

    while (i < 10) {    // Sprawdzenie warunku

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

    }

  }
}
```

Wynik:

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

Instrukcja *do ... while*

Instrukcja *do... while* jest modyfikacją ww. pętli *while*. Różnica to moment sprawdzania warunku pętli – w pętli *while* warunek jest sprawdzany na początku, w pętli *do... while* – na końcu. Pętla *do... while* wykona zawsze co najmniej jeden przebieg. Składnia ww. instrukcji jest następująca:

```
do (instrukcja) while (warunek);
```

Przykład:

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

    int i = 10;      // Deklaracja zmiennej 'i'

    do {

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

    } while (i > 0);    // Sprawdzenie warunku

  }
}
```

Wynik:

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

Instrukcja *for*

Instrukcja *for* łączy w sobie trzy działania pętli: inicjalizację (poprzedzającą pierwszą iterację), sprawdzenie warunku oraz zmianę wartości. Pierwsza instrukcja to inicjalizacja zmiennej kontrolującej wykonywanie pętli. Druga instrukcja to sprawdzenie warunku. Trzecia instrukcja to akcja – zazwyczaj umieszcza się tu inkrementację (++) albo dekrementację (--) zmiennej kontrolującej pętle. Składnia ww. instrukcji jest następująca:

```
for (inicjalizacja; warunek; krok) instrukcja;
```

Przykład:

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

    for (int i = 0; i < 10; i++) {
      Console.Write(i + " ");     // Instrukcja
    }

  }
}
```

Wynik:

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

Instrukcja *foreach*

Instrukcja *foreach* pozwala na sekwencyjne przeglądanie różnych zbiorów danych, np.: tablic, list. Składnia ww. instrukcji jest następująca:

```
foreach (typ_elementu nazwa in kolekcja)
instrukcja
```

Przykład:

```
class MainClass {

  enum Tydzień {  // Wyliczeniowy typ danych
    Poniedziałek = 1;
    Wtorek,
    Środa,
    Czwartek,
    Piątek,
    Sobota,
    Niedziela
  }

  void Main(CaseClass c) {

    foreach (Tydzień d) {     // Pętla foreach
      Console.WriteLine(Tydzień::SourceText(d) ↵
                    + " = " + d);
    }

  }
}
```

Wynik:

```
Poniedziałek = 1
Wtorek = 2
Środa = 3
Czwartek = 4
Piątek = 5
Sobota = 6
Niedziela = 7
```

Instrukcja *forall*

Instrukcja *forall* pozwala na sekwencyjne przeglądanie różnych zbiorów danych, podobnie jak instrukcja *foreach*. Różnica pomiędzy ww. instrukcjami występuje podczas przetwarzania zbiorów danych w postaci struktury drzewa, które w języku programowania EnScript są obiektami *NodeClass*. Działanie instrukcji *forall* zostanie przedstawione w dalszej części pracy, podczas omawiania różnic w języku EnScript zdefiniowanego w wersji 6 i 7 programu EnCase. Składnia instrukcji *forall* jest następująca:

```
forall (typ_elementu nazwa in kolekcja)
instrukcja
```

Instrukcje: *break*, *continue*

Wewnątrz dowolnej konstrukcji tworzącej pętlę, można sterować jej przebiegiem za pomocą instrukcji *break* i *continue*. Instrukcja *break* powoduje opuszczenie pętli, bez wykonywania pozostałych zawartych w niej instrukcji. Natomiast instrukcja *continue* zatrzymuje wykonanie aktualnej iteracji, powodując powrót do początku pętli w celu rozpoczęcia nowej iteracji.

Przykład:

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

    for (int i = 1; i < 11; i++) {
      Console.WriteLine(i);

      if (i == 4) {
        Console.WriteLine("Zatrzymanie bieżącej ↵
                  iteracji i powrót do początku");
        continue;
      }

      if (i == 6) {
        Console.WriteLine("Pętla została ↵
              przerwana i dalsze cyfry: 7,8,9,10 ↵
              nie zostaną wyświetlone");
        break;
      }

    }

  }

}
```

Wynik:

```
1
2
3
4
Zatrzymanie bieżącej iteracji i powrót do
początku.
5
6
Pętla została przerwana i dalsze cyfry:
7,8,9,10 nie zostaną wyświetlone.
```

Instrukcja *switch*

Instrukcja *switch* wybiera jeden z fragmentów kodu na podstawie wartości wyrażenia całkowitego. Instrukcja oblicza wartość podanego wyrażenia i porównuje z podanymi wartościami. W instrukcji tej sprawdzana jest tylko relacja równości, nie ma możliwości wykorzystania innych relacji. Jeśli któraś ze sprawdzanych wartości jest równa wartości wyrażenia, to program przechodzi do podanych dalej instrukcji i wykonuje wszystko do końca bloku, tak długo, aż napotka instrukcję *break*. Jeśli żadna z wartości nie jest równa wartości wyrażenia, to wykonywane są instrukcje przy słowie kluczowym *default*. Składnia instrukcji *switch* jest następująca:

```
switch (wyrażenie)
{
  case wartość_całkowita_1: instrukcja; break;
case wartość_całkowita_2: instrukcja; break;
case wartość_całkowita_3: instrukcja; break;
case wartość_całkowita_4: instrukcja; break;
(...)
default: instrukcja;
}
```

Przykład:

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

    int liczba = 4;

    switch(liczba) {

      case 0: Console.Write("Zmienna 'liczba' ⏎
            ma wartość: 0"); break;
      case 1: Console.Write("Zmienna 'liczba' ⏎
            ma wartość: 1"); break;
      case 2: Console.Write("Zmienna 'liczba' ⏎
            ma wartość: 2"); break;
      case 3: Console.Write("Zmienna 'liczba' ⏎
            ma wartość: 3"); break;
      case 4: Console.Write("Zmienna 'liczba' ⏎
            ma wartość: 4"); break;
    }

  }
}
```

Wynik:

```
Zmienna 'liczba' ma wartość: 4
```

Operacje na zmiennych

Operacje na zmiennych dokonywane są przy użyciu operatorów. Operator działa na jednym lub kilku argumentach, a wynikiem jego działania jest zupełnie nowa wartość. Argumenty mają inną postać niż zwykłe wywołania funkcji, ale rezultat jest w obu przypadkach taki sam. Operatory w języku EnScript można podzielić na dwie grupy ze względu na liczbę argumentów, na których działają. Wyróżnia się więc operatory *unarne* – wymagające jednego argumentu – oraz *binarne* – wymagające dwóch argumentów.

Operatory *unarne*

Operatory *unarne* występujące w języku programowania EnScript, przedstawione zostaną w formie tabeli. Regułą jest, że operatory te znajdują się przed wyrażeniem. Wyjątek stanowią operatory „++" i „--", które również mogą występować po wyrażeniu.

Tabela 2 Zestawienie operatorów unarnych

Operator	Opis
New	Tworzenie nowego obiektu
typeof()	Określenie typu obiektu
!	Negacja logiczna – zwraca zanegowaną wartość wyrażenia
++	Inkrementacja – zwiększa wartość zmiennej
--	Dekrementacja – zmniejsza wartość zmiennej
-	Unarny minus – służy do zmiany znaku wyrażenia
~	Negacja bitowa – zwraca zmienną z zanegowanymi bitami

Operatory binarne

Operatory *binarne* występujące w języku programowania EnScript, również przedstawione zostaną w formie tabeli. Operatory binarne występują pomiędzy wyrażeniami. Operatory binarne można podzielić na operatory: arytmetyczne, bitowe, logiczne, relacji, przypisania oraz zasięgu.

Tabela 3 Zestawienie operatorów zasięgu

Operator	Opis
.	Dereferencja, czyli wyłuskanie, uzyskanie dostępu do pewnej wartości
::	Służy do uzyskania dostępu do funkcji statycznej

Tabela 4 Zestawienie operatorów arytmetycznych

Operator	Opis
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Operator reszty z dzielenia

Tabela 5 Zestawienie operatorów bitowych

Operator	Opis
<<	Przesunięcie bitowe w lewo
>>	Przesunięcie bitowe w prawo
&	Iloczyn bitowy
^	Różnica bitowa
|	Suma bitowa

Tabela 6 Zestawienie operatorów logicznych

Operator	Opis
&&	Iloczyn logiczny
II	Suma logiczna

Tabela 7 Zestawienie operatorów relacji

Operator	Opis
<	Oznacza: mniejsze od...
>	Oznacza: większe od...
<=	Oznacza: mniejsze lub równe z...
>=	Oznacza: większe lub równe z...
==	Oznacza: równe z...
!=	Oznacza: różne od...

Tabela 8 Zestawienie operatorów przypisania

Operator	Opis
=	Przypisuje zmiennej po lewej stronie wartość wyrażenia prawej strony
*=	Służą do skrócenia zapisu operacji, które można zapisać przy użyciu operatorów arytmetycznych oraz bitowych
/=	
%=	
+=	
-=	
<<=	
>>=	
&=	
^=	
|=	

Programowanie obiektowe

Każdy program (skrypt) programu EnCase składa się z jednej lub wielu klas. W dotychczasowych przykładach była to tylko jedna klasa o nazwie *MainClass*.

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

    Console.WriteLine("Hello World");

  }
}
```

Klasy

Klasy są opisami obiektów, czyli bytów programistycznych, które mogą przechowywać dane oraz wykonywać polecone przez programistę zadania. Schematyczny szkielet klasy wygląda następująco:

```
class nazwa_klasy {
   //treść klasy
}
```

Obiekt, który został stworzony na podstawie danej klasy nazywany jest jej *instancją*. Obiekt tworzy się przy użyciu operatora *new*. Klasa jest jak gdyby matrycą służącą do tworzenia obiektów. Informuje ona wirtualną maszynę, jak należy utworzyć obiekt konkretnego typu. Każdy obiekt utworzony na podstawie klasy może mieć unikalne wartości składowych. Na przykład można użyć klasy o nazwie Pies, do stworzenia kilku różnych obiektów, z których każdy będzie np. innej rasy.

Przykład:

```
class MainClass {
  void Main(CaseClass c) {
    //Utworzenie nowych obiektów typu Pies
    Pies pies_1 = new Pies();
    Pies pies_2 = new Pies();
    Pies pies_3 = new Pies();
    // Składowe obiektów - informacje o obiektach
    pies_1.wielkość = 50;
    pies_1.rasa = "Husky";
    pies_1.nazwa = "Borys";

    pies_2.wielkość = 40;
    pies_2.rasa = "Bokser";
    pies_2.nazwa = "Kolec";

    pies_3.wielkość = 30;
    pies_3.rasa = "Beagle";
    pies_3.nazwa = "Rewin";
  }

  // Utworzenie klasy 'Pies'
  class Pies {
    int wielkość;
    String rasa;
    String nazwa;
  }
}
```

Powyższy przykład zawiera trzy zmienne referencyjne o nazwach: *pies_1, pies_2 i pies_3*. Zmienne referencyjne zawierają bity reprezentujące sposób dotarcia do danego obiektu. Można użyć operatora kropki (.) wraz ze zmienną referencyjną, aby uzyskać dostęp do składowych obiektu. Użycie operatora kropki wraz ze zmienną referencyjną można wyobrazić sobie jako naciskanie przycisku na pilocie sterującym konkretnym obiektem.

Funkcje

Klasy, oprócz pól przechowujących dane, zawierają także funkcje, które wykonują zapisane przez programistę operacje. Funkcje mogą modyfikować dane i zwracać różne wartości. Wywołanie funkcji polega na wpisaniu jej nazwy w programie. W momencie napotkania wywołania funkcji program przechodzi do wykonania kodu funkcji. Kiedy funkcja się kończy, program wraca do miejsca jej wywołania. Składnia deklaracji funkcji wygląda następująco:

```
typ nazwa (deklaracja argumentów)
{
instrukcje;
}
```

Przykład:

```
class MainClass {

  // Deklaracja pierwszej funkcji bezargumentowej
  void Wyświetl() {
    Console.WriteLine("Hello World");
  }

  // Deklaracja drugiej funkcji z argumentem
  void PobierzTekst(String text) {
    Console.WriteLine(text);
  }

  // Deklaracja trzeciej funkcji z 2 argumentami
  // zwracająca wartość całkowitą
  int Suma(int a, int b){
    int c = a + b;
    return c;
  }

  void Main(CaseClass c){ //Funkcja główna 'Main'
    // Wywołanie kolejno wszystkich funkcji
    Wyświetl();
    PobierzTekst("Tekst przekazywany do funkcji ↵
          jako argument");
    int d = Suma(3,7);
    Console.WriteLine(d);
  }
}
```

Wynik:

```
Hello World
Tekst przekazany do funkcji jako argument
10
```

Funkcje określają również czynności, jakie obiekt jest w stanie wykonać.

Przykład:

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

    //Utworzenie obieku
    Pies pies_1 =  new Pies();

    // Składowe obiektu - informacje
    pies_1.wielkość = 50;
    pies_1.rasa = "Husky";
    pies_1.nazwa = "Borys";

    pies_1.szczekaj();
  }

  // Utworzenie klasy 'Pies'
  class Pies {
    int wielkość;
    String rasa;
    String nazwa;

    // Funkcja - zachowanie obiektu
    void szczekaj() {
      Console.WriteLine("Hau! Hau!");
    }
  }
}
```

Wynik:

```
Hau! Hau!
```

Funkcje statyczne

Funkcje składowe klasy mogą być zadeklarowane jako statyczne. Funkcje te można wywołać nawet wtedy, gdy nie istnieje jeszcze żaden obiekt klasy. Do nazwy klasy statycznej odwołujemy się poprzez nazwę tej klasy oraz operatora zasięgu (::). Słowo kluczowe *static* pozwala wywoływać funkcję bez konieczności posługiwania się obiektem tej klasy. Funkcja statyczna umożliwia działanie, które nie zależy od składowych, a zatem nie wymaga użycia obiektu.

Przykład:

```
class MathClass {   // Klasa MathClass

  // Definicja funkcji statycznej
  static long Potęga(long x) {
    return x * x;
  }
}

class MainClass {
  void Main(CaseClass c) {

    // Wywołanie funkcji statycznej
    Console.WriteLine("5 do potęgi 2 = " ↵
              + MathClass::Potęga(5));

  }
}
```

Wynik:

```
5 do potęgi 2 = 25
```

Konstruktor

Konstruktor jest specjalną funkcją wykonywaną podczas tworzenia obiektu. Konstruktor zawiera kod, który zostaje wykonany w momencie użycia operatora *new*. Innymi słowy, konstruktor zawiera instrukcje wykonywane w momencie tworzenia nowego obiektu. Funkcja będąca konstruktorem nigdy nie zwraca żadnego wyniku i musi mieć nazwę zgodną z nazwą klasy.

```
class nazwa_klasy {
    nazwa_klasy(){
        // kod konstruktora
    }
}
```

Każda tworzona klasa ma konstruktor, nawet jeśli nie zostanie on napisany przez programistę. W takim przypadku konstruktor zostanie stworzony przez kompilator. Podstawową cechą konstruktora jest to, iż jest on wykonywany, zanim obiekt zostanie skojarzony z odwołaniem. W większości przypadków konstruktor używany jest do inicjalizacji stanu obiektu, czyli do określenia wartości jego składowych.

Przykład:

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

    //Utworzenie obieku
    Pies pies_1 =  new Pies();

    // Wyświetlenie cech obiektu
    Console.WriteLine(pies_1.wielkość);
    Console.WriteLine(pies_1.rasa);
    Console.WriteLine(pies_1.nazwa);
  }

  // Utworzenie klasy 'Pies'
  class Pies {
    int wielkość;
    String rasa;
    String nazwa;
    Pies() {   // Konstruktor

      // Określenie wartości obiektu
      wielkość = 50;
      rasa = "Husky";
      nazwa = "Borys";
    }
  }

    void szczekaj() {
      Console.WriteLine("Hau! Hau!");
    }
  }
}
```

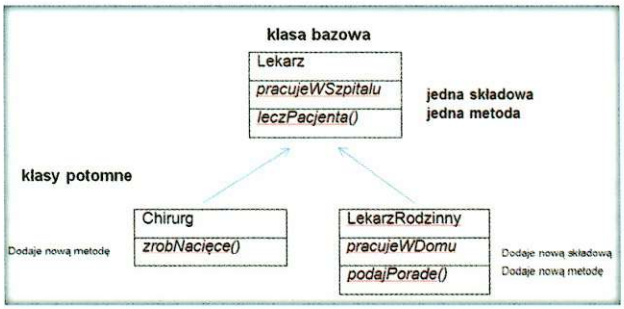
Wynik:

```
50
Husky
Borys
```

Dziedziczenie

Dziedziczenie to jeden z fundamentów programowania obiektowego. Umożliwia sprawne i łatwe wykorzystanie już raz napisanego kodu czy budowanie hierarchii klas przejmujących swoje właściwości. Dziedziczenie polega na budowaniu nowych klas na bazie już istniejących. Każda taka nowa klasa przejmuje zachowanie i właściwości klasy bazowej. Projektując klasy przy wykorzystaniu dziedziczenia, należy umieścić wspólny kod w klasie bazowej, a następnie poinformować klasy wyspecjalizowane, iż konkretna wspólna (bardziej abstrakcyjna) klasa, jest ich klasą bazową. W przypadku, gdy jedna klasa dziedziczy po drugiej, to klasa potomna dziedziczy po klasie bazowej. Potocznie mówi się, że klasa potomna rozszerza klasę bazową. Relacja dziedziczenia oznacza, że klasa potomna dziedziczy po klasie bazowej. W języku EnScript dziedziczenie wyrażane jest za pomocą dwukropka (:), a cała definicja schematycznie wygląda następująco:

```
class klasa_potomna:klasa bazowa {
  // wnętrze klasy
}
```



Ryc. 1. Przykład dziedziczenia.

Przykład:

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

    // Utworzenie obiektu lekarza rodzinnego
    // Atrybuty
    LekarzRodzinny lekarz = new LekarzRodzinny();
    lekarz.pracujeWDomu = true;
    lekarz.pracujeWSzpitalu = true;
    Console.WriteLine("Lekarz rodzinny: ");
    Console.WriteLine("Pracuje w domu: " +
                      lekarz.pracujeWDomu);
    Console.WriteLine("Pracuje w szpitalu: " +
                      lekarz.pracujeWSzpitalu);
```

```
    Console.WriteLine("Porada lekarza: ");
    lekarz.podajPorade();

  }

  // Klasa bazowa 'Lekarz'
  class Lekarz {

    bool pracujeWDomu;
    void leczPacjenta() {
      Console.WriteLine("Leczy pacjenta");
    }

  }

  // Klasy podstawowe: LekarzRodzinny, Chirurg,
  // które dziedziczą po klasie Lekarz
  class LekarzRodzinny:Lekarz {

    bool pracujeWDomu;
    void podajPorade() {
      Console.WriteLine("Pacjent musi zażywać ↵
                         witaminę C");
    }

  }

  class Chirurg:Lekarz {
    void zróbNacięcie() {
      Console.WriteLine("Wykonuję nacięcie");
    }
  }

}
```

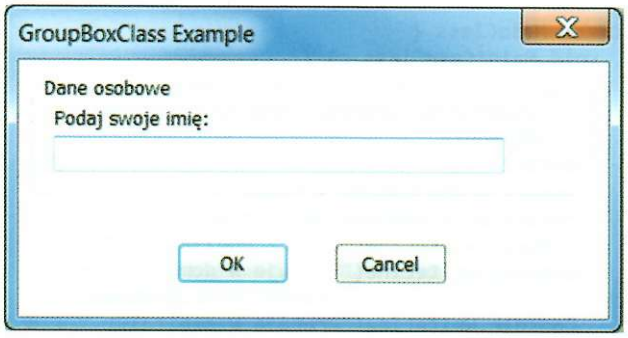
Wynik:

```
Lekarz rodzinny:
Pracuje w domu: 1
Pracuje w szpitalu: 1
Porada lekarza: Pacjent musi zażywać witaminę C
```

Graficzny interfejs użytkownika (GUI)

Wykorzystując obiekty, można tworzyć zaawansowane aplikacje działające w oparciu o graficzny interfejs użytkownika. Tworząc graficzny interfejs użytkownika, należy pamiętać, że składa się on właśnie z obiektów: przycisków, etykiet, pól tekstowych itd., które mają własne składowe oraz funkcje. W niniejszym artykule przedstawione zostaną jedynie dwa przykłady graficznego interfejsu użytkownika.

Przykład:

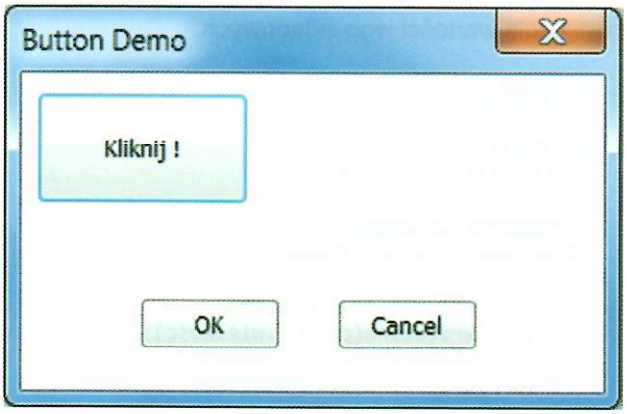


Ryc. 2. Przykładowe okno dialogowe (obiekt).

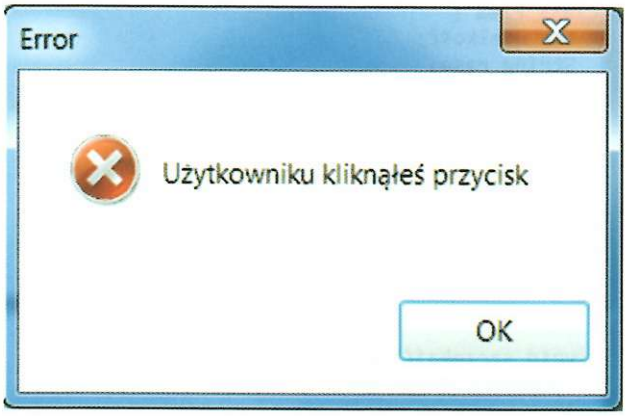
Kod źródłowy wyświetlający ww. okno:

```
class InputDialogClass: DialogClass {
  GroupBoxClass Group;
  StringEditClass StringEdit;
  InputDialogClass(DialogClass parent,String &s):
    DialogClass(parent, "GroupBoxClass Example"),
    Group(this, "Dane osobowe", START, START, ↵
          250, 40, 0),
    StringEdit(this, "Podaj swoje imię:", 15, ↵
          15, 200, 12, 0, s, 255, 0)
      {
      }
}

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

Graficzny interfejs użytkownika związany jest z przechwytywaniem zdarzeń generowanych przez użytkownika. W programie EnCase Forensic v.6 za przechwytywanie zdarzeń, np. kliknięcie przez użytkownika przycisku, odpowiedzialna jest klasa *EventClass*, zaś w wersji 7 ww. programu klasa *HandlerClass*.



Ryc. 3. Przykładowe okno dialogowe z przyciskiem 'Kliknij'.



Ryc. 4. Okno z komunikatem wyświetlone po kliknięciu przycisku 'Kliknij'.

Kod źródłowy wyświetlający ww. okna i przechwytujący zdarzenia:

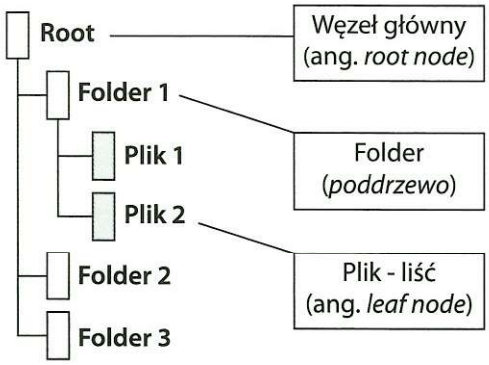
```
class ButtonDiagClass: DialogClass {

  ButtonClass Button;
  ButtonDialClass(DialogClass parent = null):
    DialogClass(parent, "Button Demo"),
    Button(this, "Kliknij !", START, START,
           75, 30, 0)   // Przycisk
    {
    }
  virtual void ChildEvent(const EventClass
                          &event) {
    // Obsługuje zdarzenie kliknięcia
    DialogClass::ChildEvent(event);
    if (Button.Matches(event)) {
      // Wyświetla komunikat
      ErrorMessage("Użytkowniku kliknąłeś
                    przycisk");
    }
  }
}

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

Listy połączone, struktura drzewa

W celu zrozumienia sposobu wyświetlania oraz przetwarzania danych w programie EnCase pomocne jest zrozumienie struktury drzewa oraz struktury list połączonych.

Lista połączona jest to struktura danych składająca się z elementów przystosowanych do łączenia się ze sobą w miarę potrzeb. Elementy listy nazywane są węzłami (*node*). Listy połączone występują w trzech odmianach: lista połączona pojedynczo, lista połączona podwójnie oraz drzewo.

Drzewo jest najbardziej złożoną odmianą listy łączonej oraz podstawową strukturą danych wykorzystywaną w programie EnCase służącą do prezentowania danych w oknie struktury danych (A). Drzewo to skończony zbiór węzłów (*nodes*) oraz liści (*leaf nodes*) z wyróżnionym węzłem głównym (wierzchołkiem) zwanym korzeniem (*root node*) i pozostałymi węzłami (wierzchołkami) podzielonymi na podzbiory, będącymi poddrzewami głównego drzewa. Każde poddrzewo ma swój korzeń, który z kolei ma swoje poddrzewa itd., a zatem węzeł jest korzeniem pewnego poddrzewa. Liść jest to węzeł (element) drzewa, który nie ma potomków, np. plik lub pusty folder. Często liśćmi są węzły najbardziej oddalone od korzenia.



Ryc. 5. Struktura danych w postaci drzewa.

Za przedstawienie danych w programie EnCase w postaci drzewa odpowiada klasa *NodeClass*. Większość klas programu EnCase dziedziczy składowe i funkcje ww. klasy. Wybrane funkcje klasy *NodeClass*, tj.: *Parent()*, *FirstChild()*, *LastChild()*, *Next()*, umożliwiają przetwarzanie danych w postaci struktury drzewa.

Przykład

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

EnCase Forensic v.6 i 7 – podstawowe różnice w języku EnScript

Język programowania EnScript jest językiem niekompatybilnym, co oznacza, że skrypty napisane w programie EnCase Forensic v.6 nie będą działały w najnowszej wersji programu. Jedynym wyjściem, jakie pozostaje użytkownikowi programu EnCase v.7, jest przepisanie skryptów z wersji 6 w taki sposób, aby działały one w najnowszej wersji programu. Przed przystąpieniem do aktualizacji skryptów warto poznać podstawowe różnice występujące w języku EnScript zdefiniowanym w obydwu wersjach programu EnCase Forensic.

Iteracja danych

Podstawowa różnica w języku EnScript zdefiniowanym w ww. wersjach programu EnCase Forensic wynika ze sposobu przetwarzania danych widniejących w programie w postaci struktury drzewa. W programie EnCase v.6 w celu przetworzenia danych należy użyć jednej z dwóch instrukcji sterujących: *foreach* lub *forall*. Instrukcja sterująca *foreach*, pozwala na uzyskanie dostępu do wszystkich wierzchołków węzła głównego, natomiast instrukcja *forall* pozwala na uzyskanie dostępu do wszystkich elementów struktury drzewa.

V6

```
class MainClass {
  void Main(CaseClass c) {
    forall (EntryClass entry in c.EntryRoot()) {
      Console.WriteLine(entry.Name());
    }
  }
}
```

W przypadku programu EnCase Forensic v.7 w celu przetworzenia lub wyświetlenia danych w postaci struktury drzewa należy posłużyć się obiektem typu *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());
      }

  }

}
```

Urządzenie – uzyskanie dostępu

Kolejną różnicą jest sposób uzyskania dostępu do urządzenia dodanego do sprawy. W programie EnCase Forensic v.6 bezpośredni dostęp do urządzenia można uzyskać za pomocą obiektu typu *DeviceClass*.

V6

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

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

  }
}
```

W programie EnCase Forensic v.7 pomiędzy obiektami typu *DeviceClass* i *CaseClass*, znajduje się obiekt *EvidenceClass*. Przykłady:

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());
        }
    }

  }

}
```

Iteracja zawartości urządzenia

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());
        }
      }
    }

  }
}
```

Uzyskanie dostępu do pliku

Skrypty programu EnCase pozwalają na uzyskanie dostępu do pliku, który został zaznaczony przez użytkownika. W programie EnCase v.6 funkcjonalność ta była ograniczona wyłącznie do elementów prezentowanych przez obiekty typu *EntryClass*, widocznych w oknie *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());
    }

  }
}
```

W wersji 7 funkcjonalność ta została znacznie rozszerzona i możliwe jest uzyskanie dostępu do zaznaczonego obiektu typu *EntryClass*, *BookmarkClass*, *RecordClass* itd.

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());
    }
  }
}
```

Zakładki – tworzenie folderów

Sposób tworzenia folderów w zakładkach programu EnCase został zmieniony nieznacznie. W wersji 6 należy posłużyć się obiektem klasy *BookmarkFolderClass*, zaś w wersji 7 programu obiektem klasy *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);

  }
}
```

Zakładki – tworzenie notatek

V6

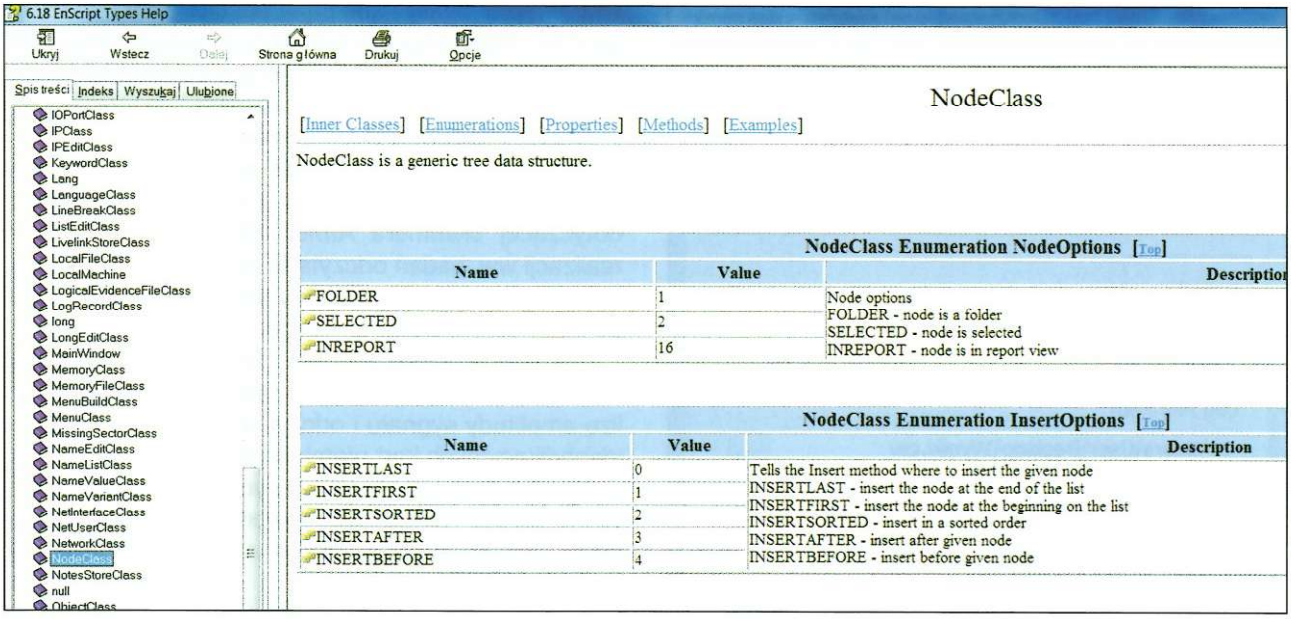
```
class MainClass {
  void Main(CaseClass c) {

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

  }
}
```

V7

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

  BookmarkClass booknote = c.BookmarkRoot();
  BookmarkClass note(booknote, "Notatka");
  note.SetComment("Zawartość notatki");

  }
}
```

EnScript API

Program EnCase jest dystrybuowany wraz z bogatą dokumentacją w formie pliku *HTML* określaną jako API (Application Programming Interface). Dokumentacja ta jest najlepszym źródłem pozwalającym na zdobycie szczegółowych informacji o klasach, ich składowych i funkcjach. Dokumentacja API zawiera również przykładowe programy (skrypty), które są bardzo pomocne podczas tworzenia własnych aplikacji. Zawartość API programu EnCase Forensic można wyświetlić za pomocą menu programu: *Help -> EnScript Help*.



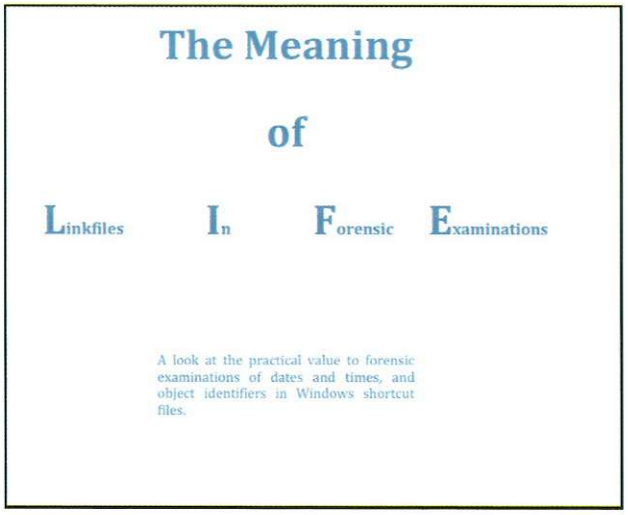
Ryc. 6. Dokumentacja Enscript API.

Przykłady autorskich rozwiązań programistycznych

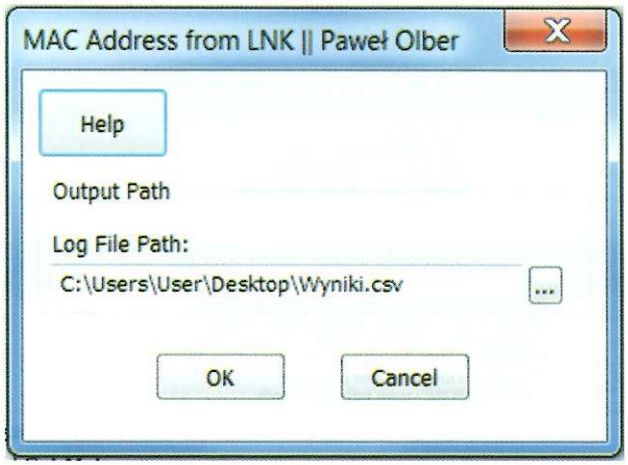
Program MAC Address from LNK

Skrypt MAC Address from LNK przeznaczony jest do odczytu adresów MAC kart sieciowych zapisanych w plikach o rozszerzeniu LNK, tzw. skrótach utworzonych na dysku twardym z systemem plików NTFS. Każdorazowe utworzenie przez użytkownika skrótu do pliku lub programu na dysku z systemem plików NTFS skutkuje zapisaniem w utworzonym pliku adresu MAC karty sieciowej zamontowanej w komputerze. Skrypt uwzględnia jedynie pliki z rozszerzeniem LNK, które nie zostały skopiowane lub przeniesione z innego nośnika danych lub partycji.

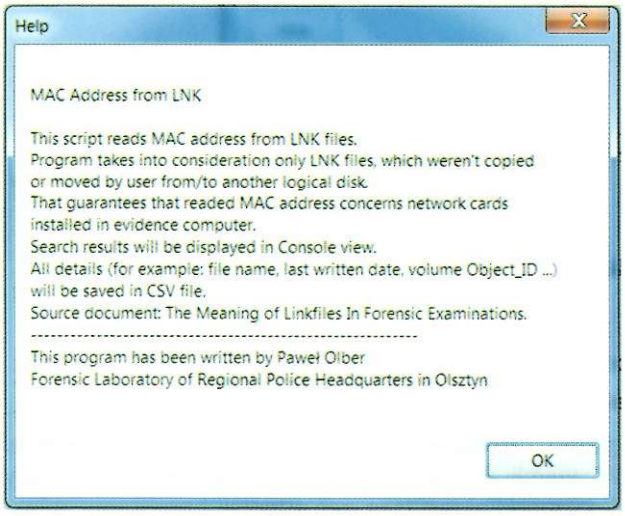
Przedmiotowy skrypt okazał się niezwykle istotny w sprawie, której celem było m.in. odczytanie adresu MAC karty sieciowej. Materiałem dowodowym w ww. sprawie był notebook oraz nadesłany luzem dysk twardy. W wyniku działania ww. skryptu z dowodowego dysku twardego (nadesłanego luzem) odczytano 2 adresy MAC kart sieciowych.



Ryc. 7. Dokument The Meaning of Linkfiles in Forensic Examinations.



Ryc. 8. Okno skryptu MAC Address from LNK.



Ryc. 9. Okno pomocy skryptu MAC Address from LNK.

Skrypt MAC Address from LNK powstał w oparciu o dane zawarte w dokumencie *The Meaning of Linkfiles in Forensic Examination*[2] autorstwa Harry'ego Parsonage. Dokument zawiera szczegółowy opis analizy zawartości plików LNK w edytorze szesnastkowym, z którego skorzystano w trakcie tworzenia skryptu.

Po uruchomieniu skryptu w programie EnCase Forensic v.6 wyświetla się okno dialogowe umożliwiające określenie ścieżki dostępu i nazwy pliku, w którym zapisane zostaną wyniki działania skryptu.

W lewym górnym rogu ww. okna znajduje się przycisk Help" Po kliknięciu ww. przycisku wyświetla się okno zawierające krótki opis działania skryptu.

Unikalne adresy MAC oraz identyfikator dysku twardego odczytane z plików LNK, wyświetlane są w oknie konsoli programu EnCase Forensic v.6.

Szczegółowe dane uzyskane w wyniku działania ww. skryptu zapisywane są w postaci pliku tekstowego CSV. Wynikowy plik tekstowy o rozszerzeniu CSV zawiera m.in. listę plików, z których odczytano adresy MAC wraz ze szczegółowymi danymi.

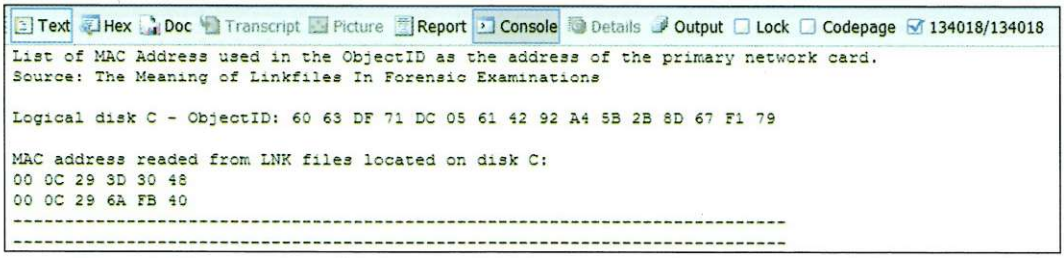
Skrypt *Redundant Remover*

Skrypt *Redundant Remover* powstał w trakcie wykonywania badań w ramach opinii kryminalistycznej dotyczącej skimmera *Atmel 45DB321D*. W trakcie realizacji ww. badań odczytano zawartość urządzenia, którą zapisano w postaci pliku binarnego zawierającego zapis sygnału dźwiękowego.

Odczytany zapis sygnału dźwiękowego zawierał nadmiarowe dane uniemożliwiające prawidłową analizę amplitudy sygnału i odczyt skopiowanych danych identyfikacyjnych kart płatniczych.

W celu usunięcia nadmiarowych danych z ww. pliku dźwiękowego zawartość dowodowego pliku binarnego odczytano za pomocą programu EnCase v.6.

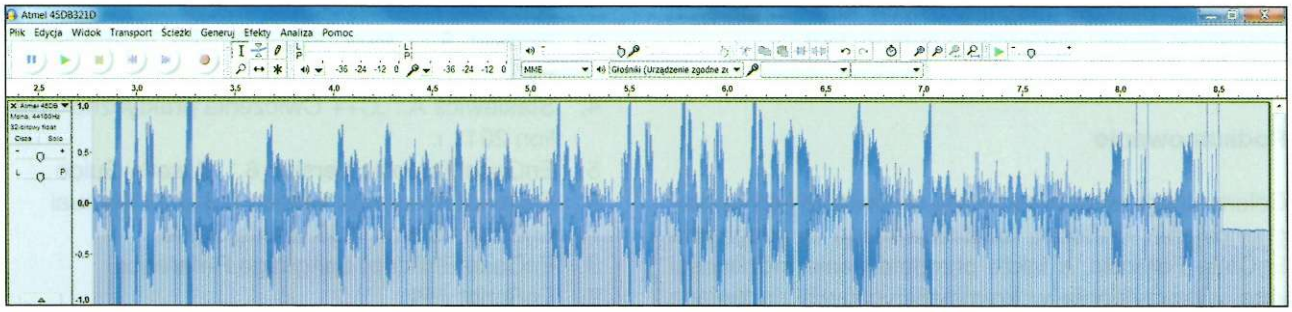
2 http://computerforensics.parsonage.co.uk/downloads/TheMeaningofLIFE.pdf



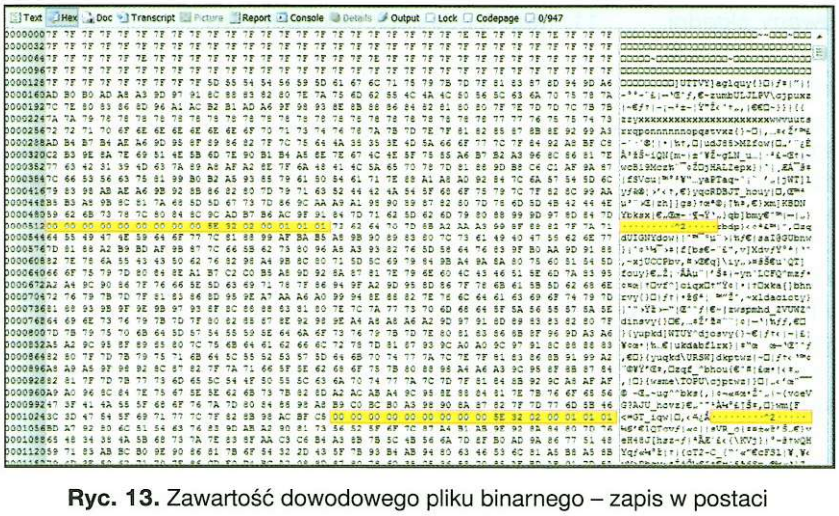
Ryc. 10. Okno konsoli – informacje dotyczące działania skryptu MAC Address from LNK.

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

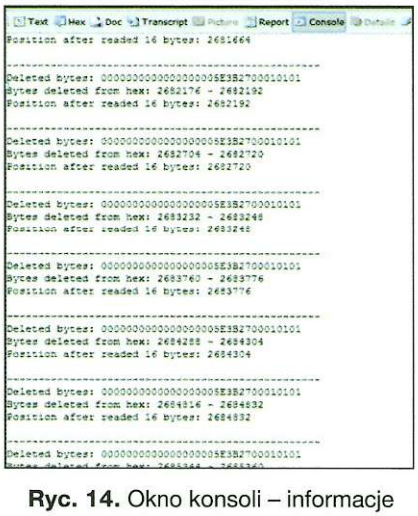
Ryc. 11. Dane odczytane za pomocą skryptu MAC Address from LNK.



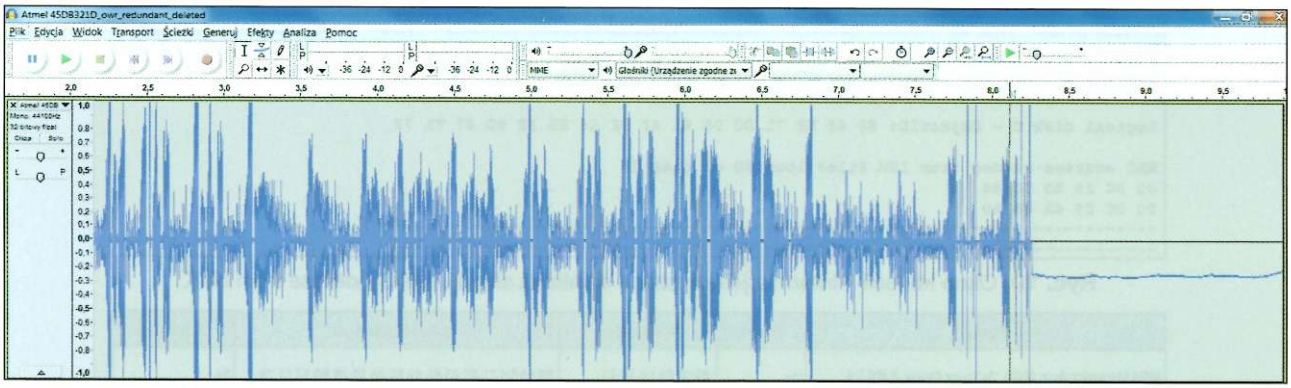
Ryc. 12. Amplituda sygnału dźwiękowego zawierająca nadmiarowe dane.



Ryc. 13. Zawartość dowodowego pliku binarnego – zapis w postaci szesnastkowej.



Ryc. 14. Okno konsoli – informacje o usuniętych nadmiarowych danych.



Ryc. 15. Amplituda sygnału dźwiękowego bez nadmiarowych danych.

Zawartość pliku poddano analizie w edytorze szesnastkowym ww. programu. Stwierdzono, że nadmiarowe dane zapisane są cyklicznie w postaci 16 charakterystycznych bajtów.

Przy użyciu skryptu *Redundant Remover* z dowodowego pliku binarnego usunięto nadmiarowe dane.

Zawartość pliku ponownie wczytano za pomocą programu Audacity 2.0 i przeprowadzono analizę fali dźwiękowej, przypisując poszczególnym amplitudom wartości binarne, które następnie przekonwertowano do wartości dziesiętnych.

W wyniku przeprowadzonych czynności w pamięci skimmera służącego do odczytu danych identyfikacyjnych z pasków magnetycznych kart płatniczych ujawniono zapisy zawierające numery kart.

Podsumowanie

Celem niniejszej publikacji było omówienie języka programowania EnScript, zdefiniowanego w programie EnCase Forensic, a także przedstawienie możliwości jego wykorzystania w kryminalistycznych badaniach cyfrowych nośników danych. Zgromadzone w toku przeprowadzonych badań informacje pozwoliły na przedstawienie w sposób usystematyzowany składni ww. języka programowania. Dodatkowo omawiane elementy składni języka poparte zostały autorskimi przykładami. W celu przedstawienia praktycznego sposobu wykorzystania skryptów w kryminalistycznych badaniach cyfrowych nośników danych zaprezentowano dwa skrypty, które powstały na potrzeby realizowanych opinii kryminalistycznych.

Podsumowując, można stwierdzić, że badania podjęte w pracy nie wyczerpują całokształtu problematyki związanej z programowaniem w języku EnScript. Z uwagi na trudny charakter podjętej problematyki, należałoby podjąć dalsze badania mające na celu stworzenie kilku aplikacji w języku EnScript, które można by wykorzystywać między innymi w kryminalistycznych badaniach cyfrowych nośników danych. Kody źródłowe powstałych aplikacji oraz szczegółowe opisy ich tworzenia, stanowiłyby doskonałe źródło wiedzy dla osób zajmujących się badaniami informatycznymi.

Bibliografia

1. Bruce E.: „Thinking in C++. Edycja polska", Helion 2002 r.
2. Bruce E.: „Thinking in Java. Edycja polska", Helion 2006 r.
3. Herbert S.: „Java. Kompendium programisty", Helion 2012 r.
4. Stasiewicz A.: „C++ Ćwiczenia praktyczne", Helion 2011 r.
5. EnCase Forensic Version 6.11 User's Guide
6. EnScript Programs Version 6.3 User Manual
7. EnCase Version 7.05 User's Guide
8. EnCase EnScript Language Reference
9. EnScript API
10. http://www.forensickb.com/2007/09/enscript-tutorial-part-i.html
11. http://codeslack.blogspot.com
12. http://www.geoffblack.com/forensics/
13. http://www.slideshare.net/markmorgan47/enscript-workshop
14. http://encase-forensic-blog.guidancesoftware.com/2014/04/enscript-changes-from-encase-version-6.html

Źródła rycin i tabel:

Ryciny 1–15: autor
Tabele 1–8: autor