OpenText™
1055 E. Colorado Boulevard
Suite 400
Pasadena, CA 91106-2375

Tel: (626) 229-9191

Email: EnCaseTraining@opentext.com
Web: www.opentext.com/encasetraining

opentext™ | Learning Services

DFIR445
ENSCRIPT™ FUNDAMENTALS

DFIR445-EnScript™ Fundamentals – Version 8.01pi (03.01.2019)

opentext™

Training overview

EnCase Learning Services

World-class training... flexible options

The best training available on critical, real-world issues

Corporations and government agencies use OpenText™ EnCase™ software to search, collect, preserve and analyze digital information for the purposes of computer forensics investigations, information assurance, electronic discovery collection, data loss prevention, compliance with mandated regulations and more. OpenText offers world-class training and a variety of professional training options to help you develop expertise in EnCase software and forensic security.

EnCase Annual Training Passport

Keep your staff up to date with the latest techniques and allow for improved planning

Organizations must ensure their investigative staff is properly trained to handle the continually evolving landscape of computer investigations. Budget burdens and scheduling conflicts may limit the amount of training your staff receives.

- The EnCase Annual Training Passport allows you to pay one, discounted flat rate for one or two years of training for your staff. If you wish to purchase either passport level within 30 days of taking a Training OnDemand, EnCase vClass or classroom course, that course fee can be applied toward purchase of the passport, resulting in significant cost savings.

EnCase Annual Training Passport fees per student

Program	Annual Passport	OnDemand Passport
One-Year Annual Training Passport	$6,495.00	$3,495.00
Two-Year Annual Training Passport	$11,995.00	$5,995.00

Annual Training Passport holders are entitled to unlimited EnCase courses at OpenText-owned classrooms, vClass or Training OnDemand.

OnDemand Passport holders can attend all of our EnCase Training OnDemand offerings.

Flex training package

Take advantage of maximum flexibility in scheduling and course selection

Organizations must constantly train investigative personnel to maintain the broad-based, changing skill set required for today's digital investigations. With increasing caseloads, personnel changes and unpredictable schedules, meeting this obligation can prove challenging.

OpenText has developed a solution that addresses these challenges at a practical price. Groups can purchase five or more classes at a reduced rate and use those training seats in the way that best suits their needs.

Program	Price USD (per seat)
Flex Pack (five-seat minimum)	$2,225.00

Features and benefits

- Structured management, budgeting and reduction of training expenses
- Qualify for CPE credits on all classroom and EnCase vClass courses
- Attendance at all courses, including EnCase Training OnDemand and EnCase vClass, qualifies for training hours earned toward OpenText EnCase certifications or renewals
- Train in one of our state-of-the-art facilities, at one of our Authorized Training Partners throughout the world or our expert EnCase instructors can come to you
- Customize a course to suit your organization's needs
- Enroll in one of our online courses with EnCase Training OnDemand
- Attend live, instructor-led virtual courses from your home or office with EnCase vClass
- Enhance professional standing by participating in one or all of our certification programs: EnCase Certified Examiner (EnCE), EnCase Certified eDiscovery Practitioner (EnCEP) or Certified Forensic Security Responder (CFSR)

For more information

Please contact EnCase Learning Services at **EnCaseTraining@opentext.com** or (626) 463 7966.

More than 80,000 students trained.

EnCaseCertification@opentext.com
opentext.com/encasetraining

opentext™

EnCase mobile training courses

Save on travel expenses and let OpenText bring the training to you. Our expert trainers will bring all the necessary equipment and materials to conduct the same high-quality instruction for your organization.

The pricing is the same as our regular instructor-led courses with the following additional charges:

Program	Price (USD)
Training Instructor Fee - 1 instructor/up to 12 students	$5,000.00
Training Instructor Fee - 2 instructors/13 to 24 students	$10,000.00
Mobile Lab Deployment	$1,000.00
Mobile Lab Deployment International	$1,500.00
**Custom training available*	Prices vary

EnCase Certified Computer Examiner (EnCE) Certification Bootcamp

The EnCE certification has become the gold standard for digital examiners. With this program, students can prepare for certification while learning how to maximize their use of EnCase software and solutions.

The bundle provides all required training and test preparation for EnCE certification. Students participating in this bootcamp can take advantage of three courses: Training OnDemand DF120-Foundations in Digital Forensics, Training OnDemand DF210 Building an Investigation and the DF310-EnCE Prep course in any delivery mode.

Program	Price (USD)
EnCE Certification Bootcamp	$5,085.00

Access to the EnCE written examination is offered to eligible students after the completion of the EnCE Prep class at no additional charge.

Training facilities

Los Angeles, CA (Pasadena, CA)
1055 East Colorado Boulevard
Suite 400
Pasadena, CA. 91106-2375

Washington, DC (Dulles, VA)
21000 Atlantic Boulevard
Suite 750
Dulles, VA. 20166

London, UK (Reading)
420 Thames Valley, Park Drive
Earley. Reading. RG6 1PT

For a complete listing of locations, including Authorized Training Partners around the world, please visit **www.opentext.com/encasetraining**.

opentext.com/contact

ENSCRIPT™ FUNDAMENTALS
CONTENTS

ENSCRIPT INTRODUCTION .... **1—3**
What Is EnScript? .... 1—3
Language Overview .... 1—4
Course Methodology and Prerequisites .... 1—4
Additional Sources of Information .... 1—5
EnScript Packaging .... 1—7
EnCase App Central .... 1—7

GETTING STARTED .... **2—11**
Introduction .... 2—11
Points to Note When Using and Programming EnScript Applications .... 2—11
Our First EnScript Program .... 2—12
Creating a New Script .... 2—12
Session Tabs .... 2—15
Showing Line Numbers .... 2—15
Basic EnScript Anatomy .... 2—16
MainClass .... 2—16
The Main() Function .... 2—17
The Scope Operator .... 2—19
Compiling and Running an EnScript Project .... 2—19
Compiling an EnScript Program .... 2—20
Running an EnScript Application .... 2—22
Script Comments .... 2—24
Clearing the Console Window .... 2—24
Programming Tips .... 2—25
Can I Compile and/or Run EnScript Applications without EnCase? .... 2—26
Will EnCase Include a Fully Featured Development Environment in Future Releases? .... 2—26

VARIABLES AND FUNCTIONS .... **3—29**
Introduction to Variables .... 3—29
Fundamental Variable Types .... 3—30
Numeric Variable Types .... 3—30
Conditional Statements .... 3—32
Functions .... 3—35
String Variables .... 3—38
Writing Formatted Output .... 3—40
Char Variable Types .... 3—41
Other Operators .... 3—41

LOOPS AND ADVANCED FUNCTIONS .... **4—45**
Introduction to Loops .... 4—45
Using the While Loop .... 4—45
Global vs. Local Variables .... 4—48
Passing Variables Into Functions By Reference .... 4—50
Using the Do/While Loop .... 4—53
Using the For Loop .... 4—55
Using the Foreach loop .... 4—56

CLASSES .... **5—63**
Introduction .... 5—63
Can I Do Without Classes? .... 5—63
Lesson Purpose .... 5—64
What Is a Class? .... 5—64

Creating a Class to Represent a Virtual Car .... 5—65
Adding Member Properties .... 5—66
Declaring, Instantiating, and Constructing Object Variables .... 5—66
The Constructor Function .... 5—68
Using the new Statement .... 5—69
Implicit new .... 5—69
Customizing the Constructor .... 5—71
Adding Additional Member Functions .... 5—73
Static Functions .... 5—75
Enumerated Types .... 5—76
Formatting Double Values .... 5—80
Includes .... 5—82

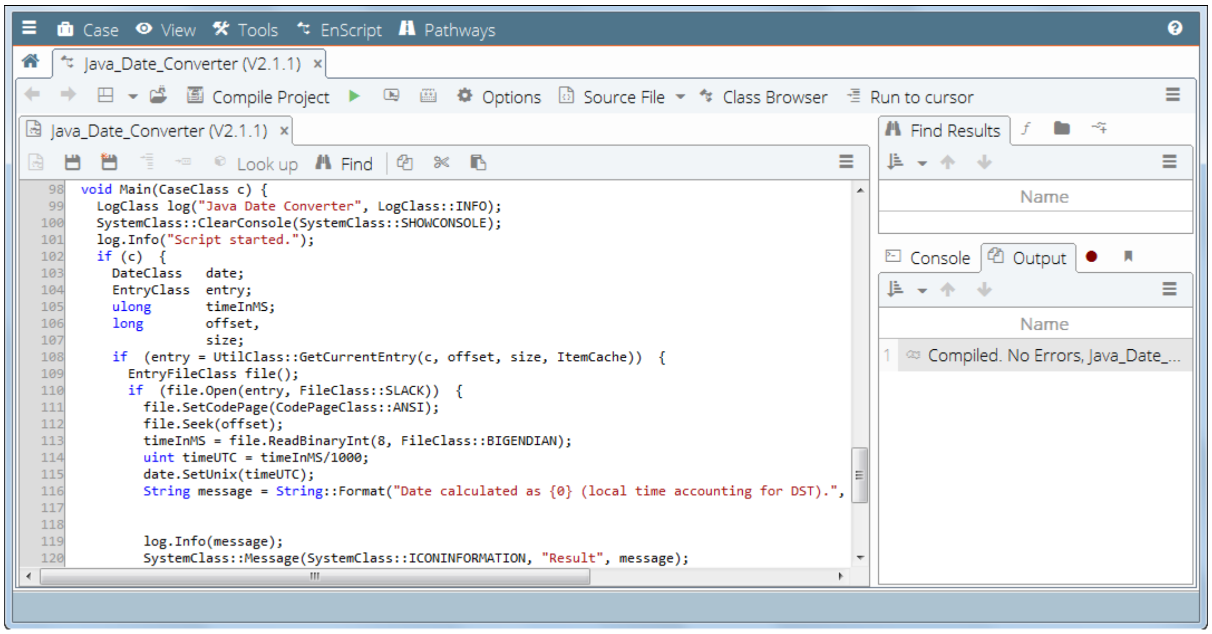
ENSCRIPT™ FUNDAMENTALS PRACTICAL EXERCISE # 1 85

ENSCRIPT™ FUNDAMENTALS PRACTICAL EXERCISE # 2 89

ENSCRIPT INTRODUCTION

WHAT IS ENSCRIPT?

EnScript is a language designed to allow a user with some knowledge of programming to fully tap into the data processing power of OpenText™ EnCase™ software (EnCase), to automate tasks, and even to create fully functional applications, which can be shared with other EnCase users.



```
void Main(CaseClass c) {
  LogClass log("Java Date Converter", LogClass::INFO);
  SystemClass::ClearConsole(SystemClass::SHOWCONSOLE);
  log.Info("Script started.");
  if (c)  {
    DateClass   date;
    EntryClass  entry;
    ulong       timeInMS;
    long        offset,
                size;
    if  (entry = UtilClass::GetCurrentEntry(c, offset, size, ItemCache))  {
      EntryFileClass file();
      if  (file.Open(entry, FileClass::SLACK))  {
        file.SetCodePage(CodePageClass::ANSI);
        file.Seek(offset);
        timeInMS = file.ReadBinaryInt(8, FileClass::BIGENDIAN);
        uint timeUTC = timeInMS/1000;
        date.SetUnix(timeUTC);
        String message = String::Format("Date calculated as {0} (local time accounting for DST).",


        log.Info(message);
        SystemClass::Message(SystemClass::ICONINFORMATION, "Result", message);
```

```
1  Compiled. No Errors, Java_Date_...
```

Figure 1-1 EnCase EnScript programming environment

Since its introduction in Version 2 of EnCase Forensic, the EnScript language has evolved into a powerful object-oriented language with inheritance, virtual functions, type reflection, and a threading model.

EnScript programming supports COM libraries from other applications, allowing you to automate document processing tasks and remote data retrieval through DCOM. It also enables you to integrate with .NET assemblies in the form of DLL files.

LANGUAGE OVERVIEW

The EnScript programming language has its roots in C/C++ but also has elements of Java and C#.

It is a case-sensitive language that ignores any whitespace that is not part of a quoted string.

EnScript source code is processed internally as Unicode but is be stored on disk as 8-bit text unless non-ASCII text is present.

A prior knowledge of C/C++ is not necessary to understand the EnScript programming language.

Our developers have gone to great lengths to provide the end user with a language that is both powerful and flexible without the overhead of having to use pointers and manage memory usage, which is necessary in order to program using those languages.

Taking this into account, putting yourself through the rigors of learning C/C++ pointer operations and memory management is unnecessary and likely to discourage you from learning EnScript programming techniques. It's much better to learn EnScript programming first and then progress to C/C++ if that's what you're interested in.

COURSE METHODOLOGY AND PREREQUISITES

The EnScript language is expansive and continues to undergo rapid development. It is not therefore possible to cover every aspect of the language in just four days.

That said, the course aims to give the student a good grounding in those areas of the language that are most likely to be of benefit during day-to-day forensic examinations.

Programming experience is not a prerequisite for attending the course in order to not discriminate against examiners who would like to learn how to harness the power of EnScript programming but have little or no programming experience.

Unfortunately, experience has shown that this can lead to quite a gap between those attendees who are experienced programmers and those who have little or no programming experience.

So as to try and bridge this gap, those sections of the student manual that document fundamental EnScript programming concepts (variables, operators, flow control, functions and basic class usage/construction) are available for anyone to download in a PDF document free-of-charge. The title of this document is "EnScript™ Fundamentals."

Inexperienced programmers are expected to review the content of the EnScript Fundamentals document in their own time to ascertain if the course is right for them. If they decide to attend the course then they should ensure that they have a good working knowledge of the programming concepts contained therein. Two practical exercises are included (together with suggested answers) to assist with this.

Please note the following:

- The subject matter contained in the EnScript Fundamentals document will ***not*** be covered in class in order to save time and allow for better coverage of those topics that are specific to EnCase and EnScript.
- It is not possible to provide tuition in advance of the course. If prospective students have any questions regarding the material contained in the document then they are advised to post them to the EnScript forum on the EnScript Support Portal as documented herein.

Additional Sources of Information

Notwithstanding the fact that this course will cover a large amount of EnScript functionality, it's already been stated that it's not possible to cover everything particularly as the EnScript language is in constant development.

The following sources of information are therefore offered in addition to the subject matter provided on this course.

- The EnScript Language Reference Guide. Originally authored by Guidance Software founder, Shawn McCreight, this concise and well-written document covers the core part of the EnScript language that is still in operation today. It details EnScript™ program structure, variable types, arrays, program control, functions, operators, classes, and object inheritance. Your instructor will make a copy of this document available to you in PDF format.

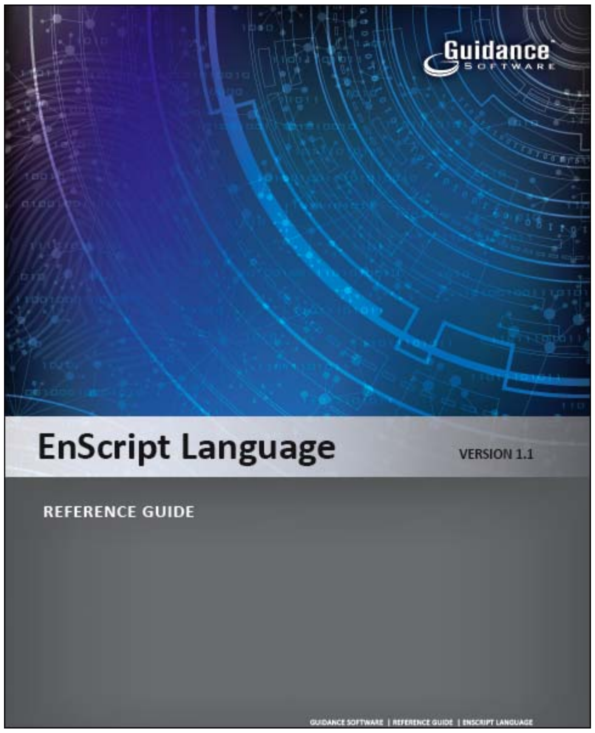
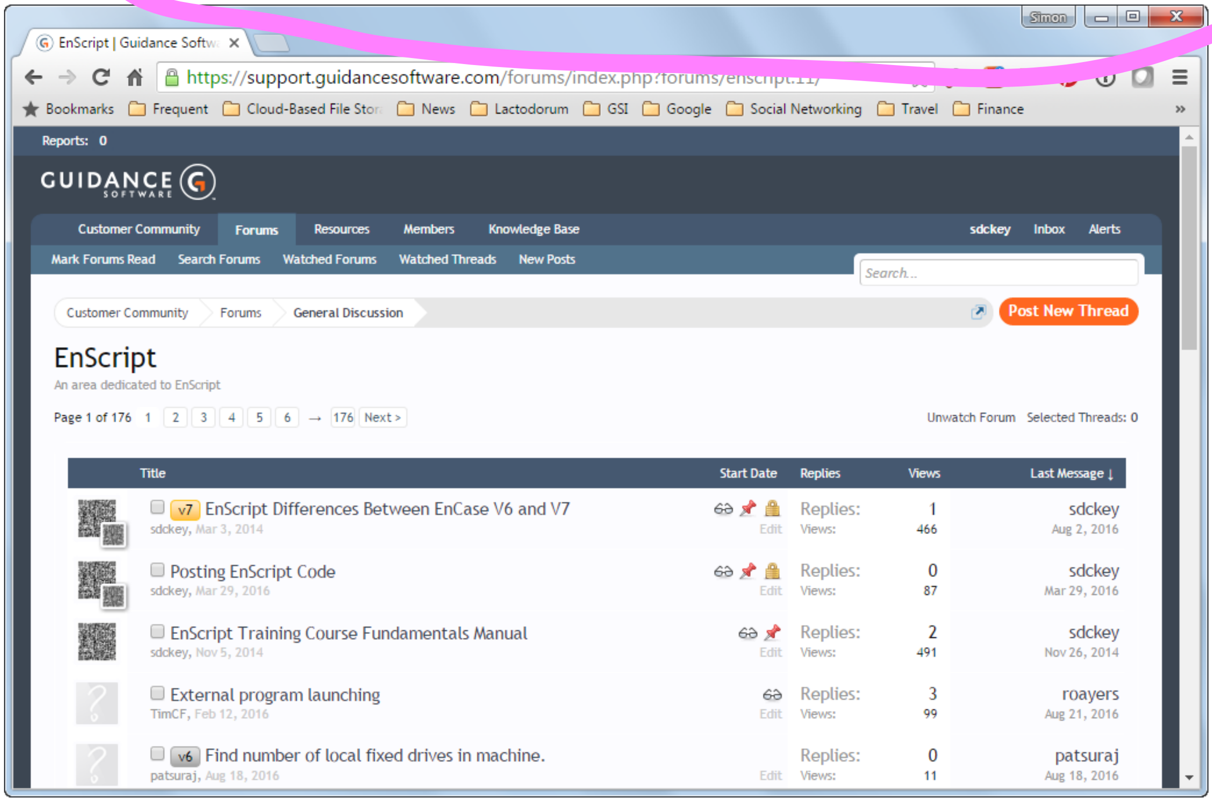


Figure 1-2 EnScript Language Reference Guide

- The EnScript Types tab in EnCase itself. This is tied to the EnScript compiler and outlines exactly what EnScript functionality is supported by the version of EnCase in question. This tab will be referred to many times during the remainder of the course.
- The EnScript help file, which contains a large number of valuable examples that illustrate how to use various aspects of the EnScript language. Please note that OpenText strives to release new and updated EnScript functionality as soon as it is available; there may therefore be a delay before this is reflected in the EnScript help file.

- The EnScript forum on the Support Portal. This is a valuable resource frequented by many experienced EnScript programmers, EnCase instructors, and Technical Services personnel; also by developers and internal EnScript programmers. It is one of the best ways to ask questions, obtain sample code, report bugs, and submit feature requests.



EnScript

An area dedicated to EnScript

Page 1 of 176 1 2 3 4 5 6 → 176 Next >

Unwatch Forum Selected Threads: 0

Title	Start Date	Replies	Views	Last Message ↓
v7 EnScript Differences Between EnCase V6 and V7 sdckey, Mar 3, 2014	Edit	Replies: Views:	1 466	sdckey Aug 2, 2016
Posting EnScript Code sdckey, Mar 29, 2016	Edit	Replies: Views:	0 87	sdckey Mar 29, 2016
EnScript Training Course Fundamentals Manual sdckey, Nov 5, 2014	Edit	Replies: Views:	2 491	sdckey Nov 26, 2014
External program launching TimCF, Feb 12, 2016	Edit	Replies: Views:	3 99	rroayers Aug 21, 2016
v6 Find number of local fixed drives in machine. patsuraj, Aug 18, 2016	Edit	Replies: Views:	0 11	patsuraj Aug 18, 2016

Figure 1-3 EnScript forum on the GSI Support Portal

- The Downloads section of the EnScript support portal. Sometimes a script may already exist that suits your purpose thus negating the need to develop one yourself. Not only that, but the code posted in the Downloads section may give you a pointer as to how to accomplish a particular task yourself. Please be aware that you need to search the Downloads section separately – it is not searched as part of searching the forums themselves.

EnScript Packaging

One of the best ways of learning how to program in the EnScript language is to examine someone else's code.

That said, many EnScript modules are made available in a packed, encrypted format with an *EnPack* file extension.

The reason for this is often misunderstood; many people believe that it's because the author of an EnScript module wishes to hide their code from prying eyes. This mistaken conclusion can be quite frustrating.

The truth of the matter is that many scripts are packed because they consist of more than one source-code file (we shall see this when we explore how to re-use code in more than one script later-on during the course) resource file, or DLL file. Distributing a script with multiple files is potentially problematic both for the script's author and end user. By packaging a script all of the necessary files are included by default; it's therefore easy to distribute the script, install, and run it.

There are of course many other reasons to package a script, but if you want to obtain the source code for one that's been packaged, it might be worth just dropping the script's author an email. He/she may be more than happy to provide you with a copy provided that it's being requested for the right reason(s).

EnCase App Central

EnScript programmers can now sign up as members of the EnCase Developer Network and market their EnScript applications via EnCase App Central, which is accessible via the following URL:

https://www2.guidancesoftware.com/appcentral/Pages/default.aspx

An EnScript developer can choose to charge for their applications or else make them available for free.

An individual signing up as an EnCase App Central EnScript developer will receive the following:

- EnCase Software Developer Kit, which includes EnCase Forensic.
- One-year EnCase Developer License (can only be used for EnScript development).
- Technical support.

The provision of an EnCase licence is particularly useful if the developer does not possess such a licence for his/her own exclusive use, which is often the case for many forensic examiners.

A developer's licence will be renewed on a yearly basis provided that he/she continues to publish EnScript applications on EnCase App Central on a regular basis.

NOTES

NOTES

NOTES

GETTING STARTED

INTRODUCTION

During this lesson we shall create our first EnScript™ program and explore the EnCase™ EnScript development environment during the process.

There is a lot of development functionality available to us in OpenText™ EnCase™ software (EnCase), but we shall concentrate on learning just enough to be productive without things getting overly complicated. As we progress through the course we shall introduce GUI functionality that will allow us to develop, debug, profile, and package the EnScript applications that we create.

Before we proceed to writing our first EnScript we need to lay some ground rules.

POINTS TO NOTE WHEN USING AND PROGRAMMING ENSCRIPT APPLICATIONS

The user should bear the following in mind when programming and/or using EnScript applications:

- Each EnScript application will consist of one or more source code files.
- Each EnScript application is compiled before it is executed.
- EnScript applications (like any other programming language) do not have access to unlimited memory.
- EnScript code will, in general terms, run more slowly than similar code running in a lower-level language such as C/C++.
- EnScript programming is case-sensitive and the EnScript compiler ignores whitespace unless it is part of a quoted string.
- The compiler will stop at the point it detects an error and cause the cursor to be placed at that location.
- The point at which the compiler detects an error may be many lines of code after the location of the error itself.
- EnScript code may be syntactically correct and yet encounter errors at run time. For instance, the user may try to write a file in a location to which he/she has read-only access.

- It's possible to write an EnScript program that is syntactically correct but that attempts the impossible. For instance, you could write code that only does a certain thing to a file if its file-extension is both *docx* and *xlsx*. This is a logical impossibility and so the task would never be executed.
- If you choose to write code using an external editor don't use tab characters – they will be ignored when the code is brought back into EnCase. Entering a tab in the EnCase script-editor dialog actually causes two space characters to be inserted.

Our First EnScript Program

It's customary to start any programming course with a "Hello World" example so that's what we shall do.

Creating a New Script

Let's start EnCase and take the option to create a new script from the EnScript menu. Note that at the moment we don't need to start a new case.

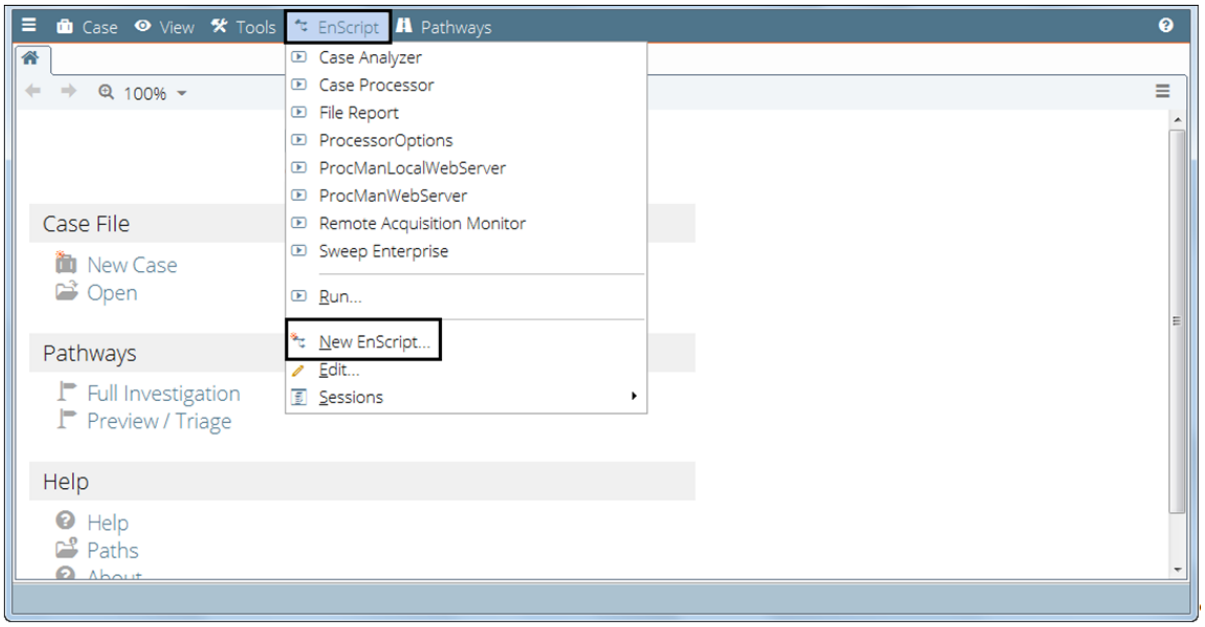
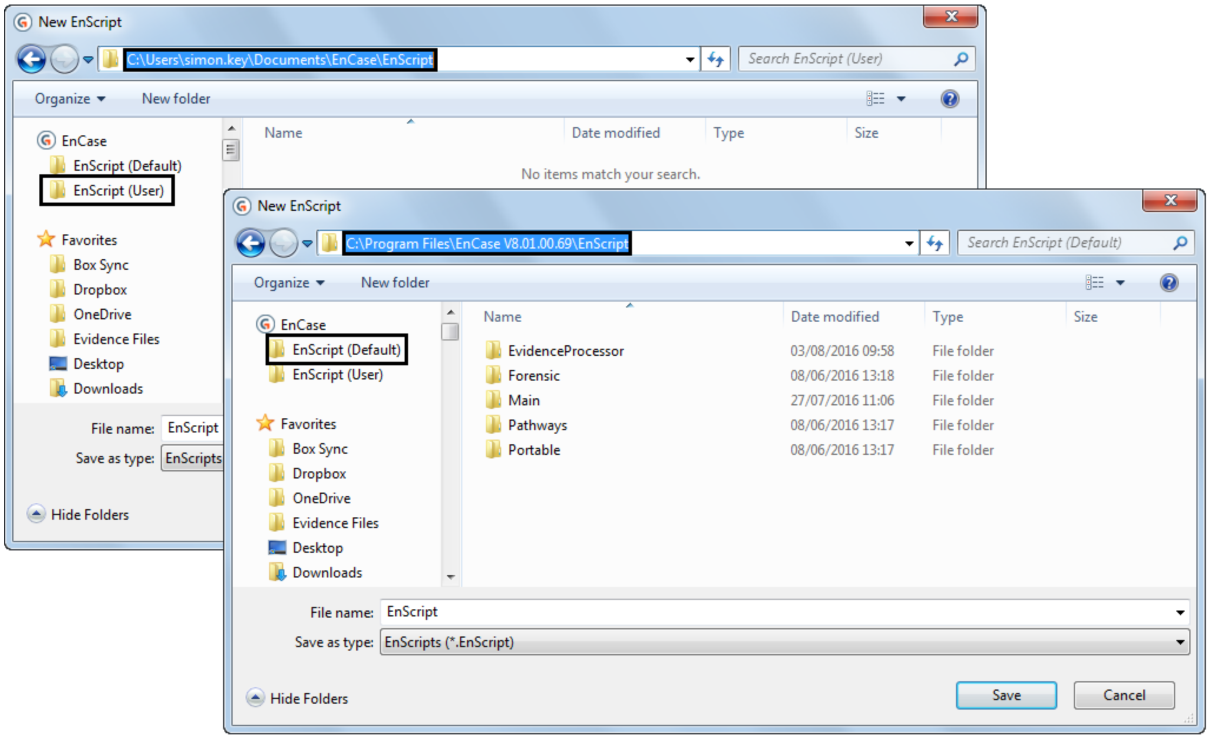


Figure 2-1 Creating a new EnScript application

The next dialog has two shortcuts that allow us to create EnScript source-code files within a subfolder of our own My Documents folder; alternatively we can use the EnScript folder that is a subfolder of the EnCase program folder.



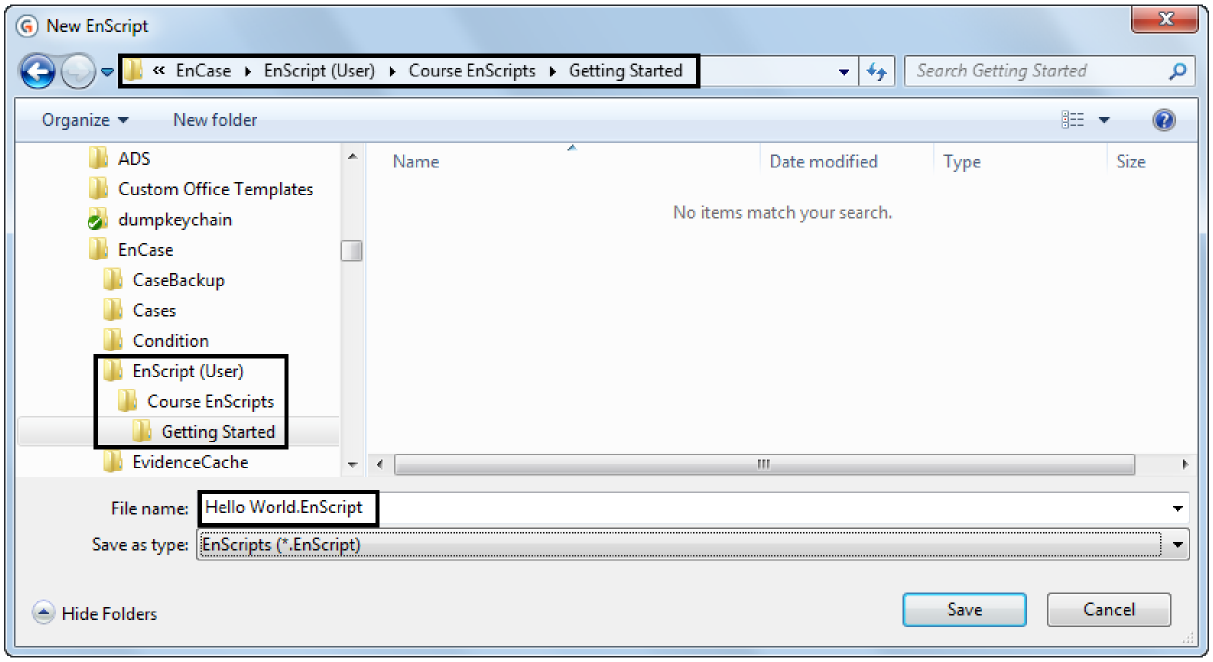
Name	Date modified	Type	Size
EvidenceProcessor	03/08/2016 09:58	File folder	
Forensic	08/06/2016 13:18	File folder	
Main	27/07/2016 11:06	File folder	
Pathways	08/06/2016 13:17	File folder	
Portable	08/06/2016 13:17	File folder	

File name: EnScript
Save as type: EnScripts (*.EnScript)

Figure 2-2 Choosing a location for a new EnScript program

Note that we're not obliged to save our script to one of these locations; we can save a script wherever we like.

Having created/selected the script folder for this lesson, go ahead and give your new script the name "Hello World.EnScript."



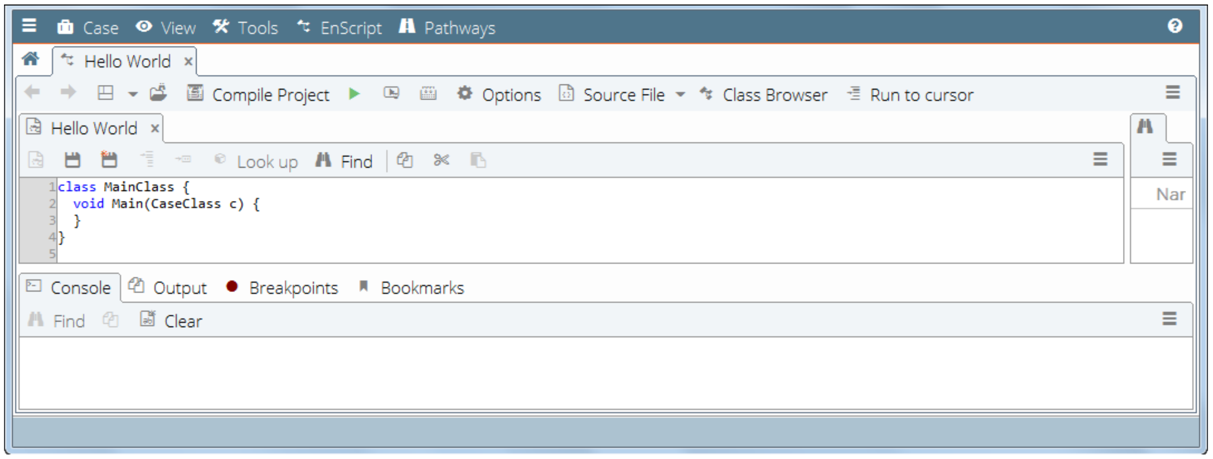
No items match your search.

File name: Hello World.EnScript

Save as type: EnScripts (*.EnScript)

Figure 2-3 Naming a new EnScript

Having named our new script, we will be presented with a new script with the barest amount of code that will run without error.



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

Figure 2-4 Viewing a newly created EnScript source-code file

The steps that we have taken thus far have led to the creation of an EnScript application with a single source-code file.

Session Tabs

Our new EnScript source-code file has been created within a new session tab.

Each session tab provides us with a customizable environment in which to develop a particular EnScript project/application. This environment makes it easy for us to work with a project that includes many different files. Previous versions of EnCase allowed us to have more than one script open at a time, but there was no way of grouping them together as a project.

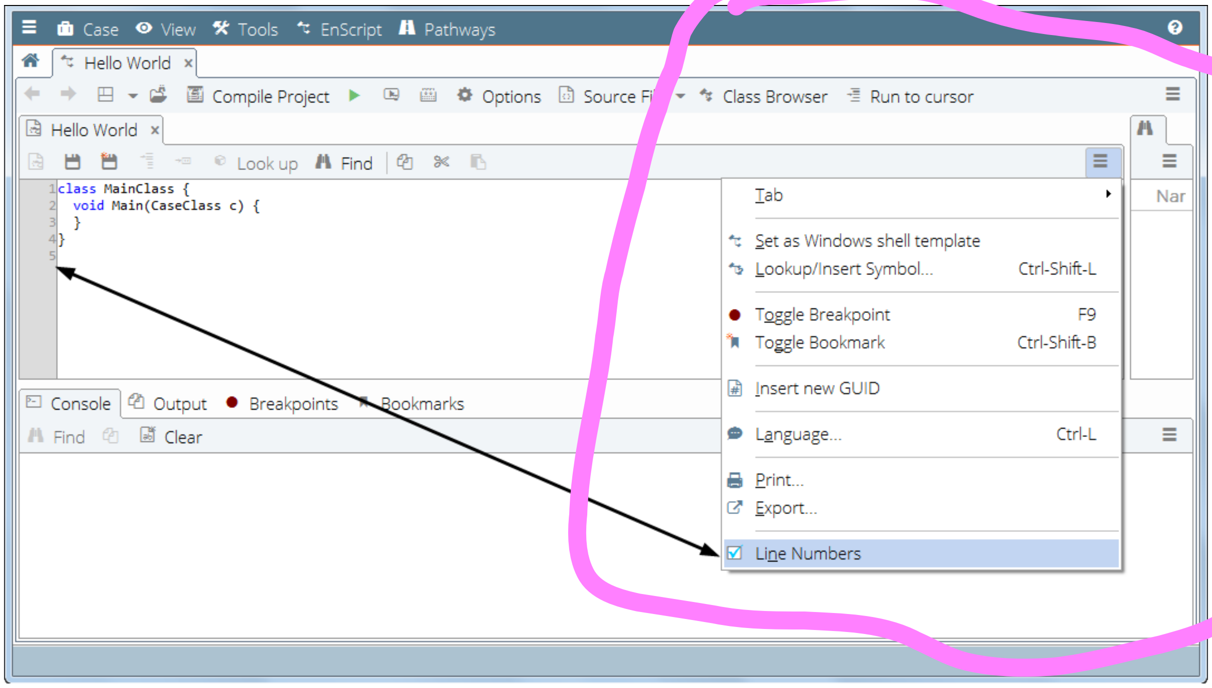
It's worth noting that the **EnScripts→Sessions→Undock** menu option allows us to move open session tabs into a separate window, which can be very useful when working with multiple monitors.

EnScript sessions offer many different debug options, but at this stage we only need to know the basics; we shall explore the other options later.

Showing Line Numbers

The first thing we want to do is make sure line numbering in enabled so we can refer to each line of code and make sense of any error messages that refer to line numbers.

To do that, we need to use an option on the script-editor pane's context menu.



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

Figure 2-5 Viewing the option to enable line numbers in the script editor

Before we go any further we need to take a brief look at the code that's already been written for us.

Basic EnScript Anatomy

For inexperienced programmers even these four lines of code can appear daunting.

That said, it's not as difficult as one might think.

What has to be remembered is that EnScript is an *object-orientated* language.

This means that when we're working with the EnScript language we're often working with objects, each one of which will have a defined set of properties as well as a number of methods that tell us how we can interact with a particular object. This is no different than the real world where we work with objects, their properties and methods every day.

A car, for instance, will have properties, such as owner, color, number of doors, top speed, fuel consumption figures, etc. The methods associated with a car will enable us to start the engine, stop the engine, accelerate, brake, refuel, etc.

When it comes to EnScript programming, all of the objects we work with, including executable EnScript applications, have to be defined by a template that the EnScript compiler can understand.

Coming back to the real-world example of a car, you might decide to use EnScript programming to create a racing game where you want to race one car against another.

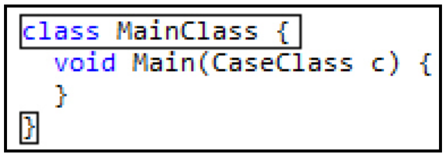
In order to make this work you'd need to create a template or *class* that would define the properties and methods for the cars in the game. For each car that you wanted to race, you'd give it a name (for identification purposes) and then tell the EnScript compiler to construct the car in memory and display it somewhere on the screen.

It's worth remembering that the properties of an object may themselves be other types of objects. For instance, a car will have an owner, which in itself will be an object that is a human being. Human beings have their own properties, such as name, weight, height, skin-color, age, etc. They also have methods such as run, jump, walk, sleep, etc.

So taking this into account, let's come back and examine our basic EnScript code.

MainClass

In order for an EnScript code to be recognised as an executable EnScript program it must contain a class definition for *MainClass*. This is identified by the first and last lines of the script.



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

Figure 2-6 Viewing the class definition that identifies an executable EnScript program

When the EnScript compiler is told to run a script it looks for a definition of a class (identified by the `class` keyword) with the name "MainClass." It then uses the code contained within the curly braces of that class definition to construct an instance of an executable EnScript program in memory and then run it.

THE MAIN() FUNCTION

In addition to an executable EnScript program having to have a MainClass definition, that definition must have a method or *function* called "Main."

A function is a block of code that performs a particular task. A function can be invoked again and again if needed. It can take one or more optional parameters as input; it can also produce or *return* a single value at the end of its processing.

Once the EnScript compiler has used the MainClass class definition to build an executable EnScript application in memory, it then looks for the Main function and invokes it. The Main function is therefore referred to as the "entry point" for the script. Its code is responsible for making the script do something, which may involve executing other functions and creating other class objects.

In the case of our basic script, lines two and three mark the definition of the Main function.

The second line consists of three elements:

- The *void* keyword. This tells the EnScript compiler that the Main function won't return a value.

  You could change this to specify that your Main function will return a value of a particular type but if you then fail to do so (by not using a *return* statement followed by a value of the type you specified), the EnScript compiler will refuse to construct the EnScript application in memory and run it; the compiler will generate an error message instead.

  Note that returning a value from the Main function of MainClass in an executable EnScript program is pointless because EnCase will discard it. EnScript filters on the other hand are expected to return a true or false (Boolean) value that will control whether an item to which a filter is applied will be shown (true) or hidden (false) from the EnCase GUI.

- The name of the function "Main." The function acting as the entry point for an executable EnScript program must have this name.

- A set of parentheses, which in this case contains a single parameter for our script to use if needed. That single parameter is a CaseClass object, which can be referred to in the Main function using the name or label of "c." As we shall see later, CaseClass is the name of a class definition that is used to refer to an EnCase case.

  There are a couple of things to note here.

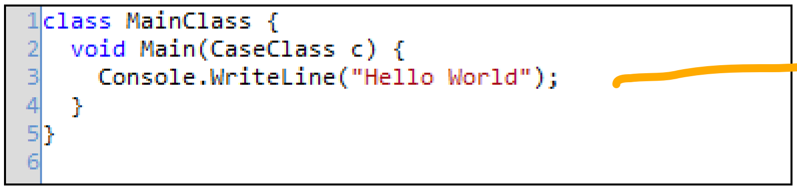
  - Firstly, there's no guarantee that the CaseClass value will be valid. If you remember, when we first loaded EnCase we did not create a case. It's our job as EnScript programmers to check that a case exists if that's what we intend to process. If it doesn't then we need to inform the user of that fact and then exit the script. We shall demonstrate how to do this later.
  - Secondly, we're not obliged to use the CaseClass value provided to us. We can ignore or remove it altogether, which would leave an empty set of parentheses.

- The opening curly brace, which marks the start of the code that would enable our function to do its job. We refer to this code as the "body" of the Main() function.

There is nothing between the opening and closing curly braces in our function definition, so our function won't actually do anything.

Therefore our next step is to add some code, which in this case will make the function write "Hello World" to the console window.

To do this we modify the code as follows. Don't forget that EnScript programming is case sensitive!



```
class MainClass {
  void Main(CaseClass c) {
    Console.WriteLine("Hello World");
  }
}

```

Body

Figure 2-7 Modifying the basic code

The new line of code highlighted in the previous screenshot has been inserted between the opening and closing braces that mark the block of code that forms the body of the Main function. It forms a statement, which must be terminated by a semicolon.

In this case we've only got one statement in the body of our function but we could add more, each one being terminated by semicolon.

Note that we indented our new statement to make it more readable. This is common practice for all statements contained within a block of code; it makes the code much easier to read.

So what does the new code do? Well let's break it down bit-by-bit.

- The word "Console" is the way we refer to the object that is the console window in EnCase. In the GUI, the console window is directly accessible from the bottom pane when working in a session tab but it is also accessible as a separate tab using the appropriate view option on the main EnCase menu.
- The period (.) character is how we link an object to one of its properties or methods. For a property, the period character can be thought as an apostrophe: if we saw code such as "Simon.HairColour" we could interpret it as the HairColour property belonging to the Simon object (whatever class of object that might be) or "Simon's HairColour."
- The word "WriteLine" represents a method (function) that can be used with the Console object or in fact any object that is a type of file or *FileClass* object that we can write to using EnScript programming. We will cover FileClass objects as part of a separate lesson.
- Immediately following the WriteLine function name, we see a set of parentheses containing a single function parameter, which in this case is the literal string "Hello World." A literal string is one that cannot be changed at run-time because it is hard-coded into the script. The WriteLine function takes this literal string and then writes it to the *FileClass* object to which it is being applied, in this case the Console object, which represents the Console window in the EnCase GUI.

The Scope Operator

Before we test our newly modified script, we ought to start introducing some of the nomenclature that we're likely to encounter when reading EnScript documentation.

When we refer to a function we normally do so by adding the opening and closing parentheses after its name as in "the WriteLine() function" or just "WriteLine()."

In addition to that, when we want to refer to the property or function of a class in general, without referring to an actual object of that class, we use two colons (::). Taken together, these colons represent the "scope" operator.

Taking this into account, if you were to read the text, "In order to write to the Console object in EnCase you'd need to use the FileClass::WriteLine() function," you'd know that you'd need to apply the WriteLine function that belongs to the FileClass class-template. This infers that the Console object is a FileClass object, which indeed it is.

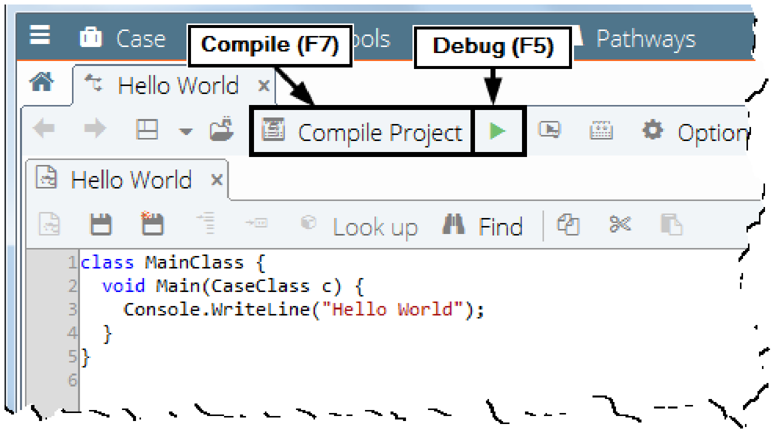
We could also refer to the Main function of the MainClass defined in our script as MainClass::Main().

As you'll see later, function and property names are not unique among different classes so if you want to refer to a function you may need to qualify its name by using the name of the class to which it belongs and the scope operator.

Don't worry if you're not getting the hang of this at the moment. We shall be coming back to it later.

Compiling and Running an EnScript Project

We have the choice of compiling or compiling and running our EnScript project using the appropriate toolbar and/or keyboard shortcut options.



```
class MainClass {
  void Main(CaseClass c) {
    Console.WriteLine("Hello World");
  }
}
```

Figure 2-8 GUI options for compiling and running an EnScript project

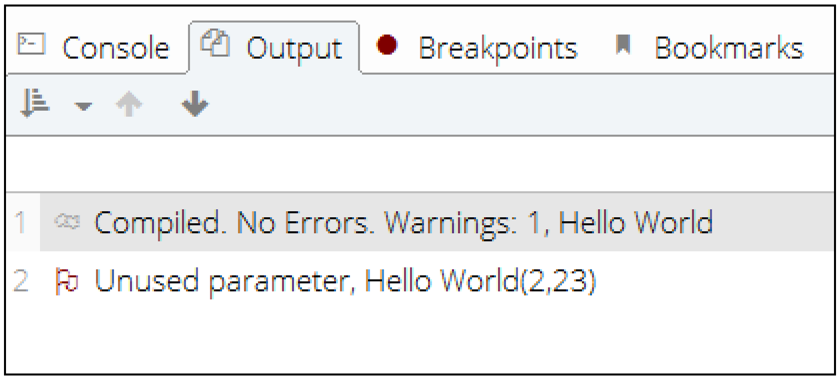
As we can see from the previous screenshot, we can compile an EnScript project using the F7 keyboard shortcut; we can run it using the F5 keyboard shortcut.

Compiling an EnScript Program

When you're writing a script, you should compile frequently, typically by using the F7 keyboard shortcut. There's nothing worse than having written thirty or forty lines of code only to find that it won't compile. You'd then have to fix the problem by finding the line(s) of code containing errors.

It worth noting that EnCase also provides the option to compile and run individual source-code files rather than an entire project. This can be useful if an EnScript application contains multiple source-code files – the examiner can check that one particular source-code file compiles even if the others haven't been completed yet.

When we compile our new Hello World EnScript project, we should see confirmation that it's compiled correctly in the Output window of the bottom pane.



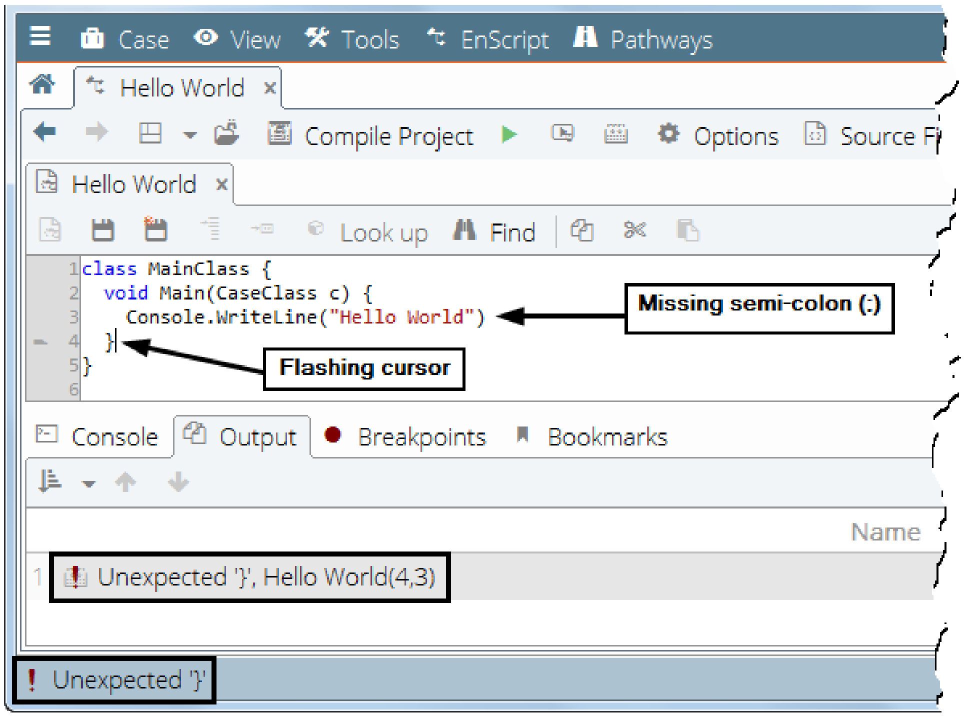
```
1  Compiled. No Errors. Warnings: 1, Hello World
2  Unused parameter, Hello World(2,23)
```

Figure 2-9 Viewing the compiler output in the Output window

Note that we get a warning telling us that we're not using the CaseClass value being provided to the MainClass::Main() function. We're already aware of this so we can safely ignore the warning.

Now let's remove the semicolon from the end of line 3 and try to compile again.

This time we get an error.



```
class MainClass {
  void Main(CaseClass c) {
    Console.WriteLine("Hello World")
  }
}
```

```
1 Unexpected '}', Hello World(4,3)
```

```
Unexpected '}'
```

Figure 2-10 Viewing compiler errors

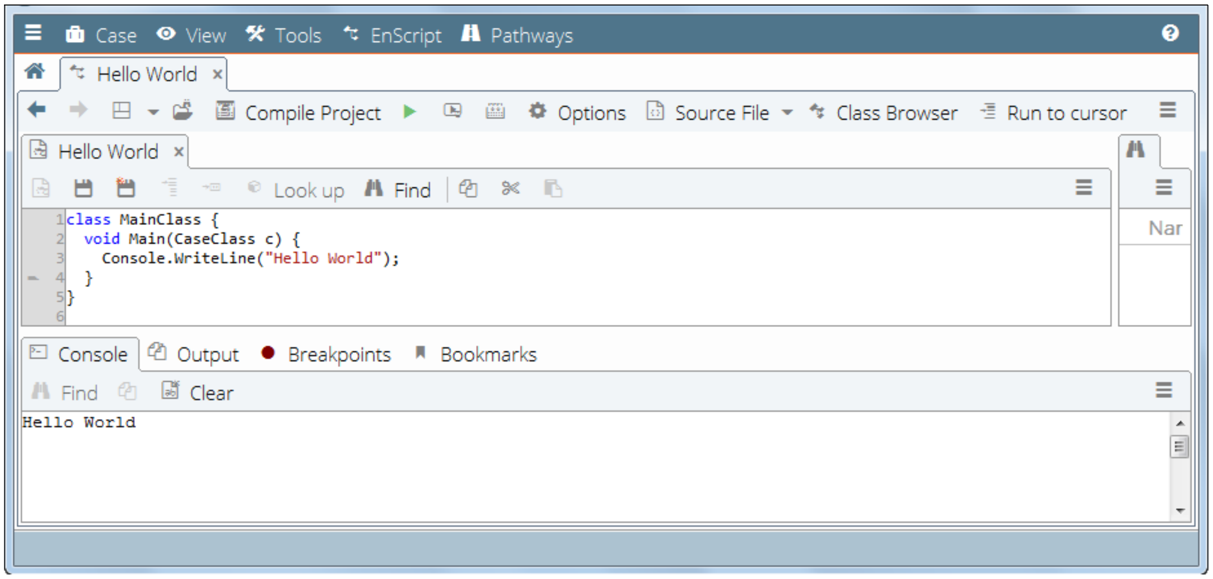
This is a compile-time error rather than a run-time error. Compile-time errors are usually easier to debug because they're caused by syntax errors, which can be identified just by re-examining the script code.

Run-time errors are generally harder to diagnose because they're usually linked to values stored in memory that only exist when the script is running. To fix this sort of problem, we'd need to use the debugger's advanced functionality, which we shall cover separately.

Anyway, in this case, we know what the problem is so let's fix it and run the script.

Running an EnScript Application

When we run the script using the toolbar option mentioned previously, or by pressing F5, we get the desired output in the console window.



```
class MainClass {
  void Main(CaseClass c) {
    Console.WriteLine("Hello World");
  }
}
```

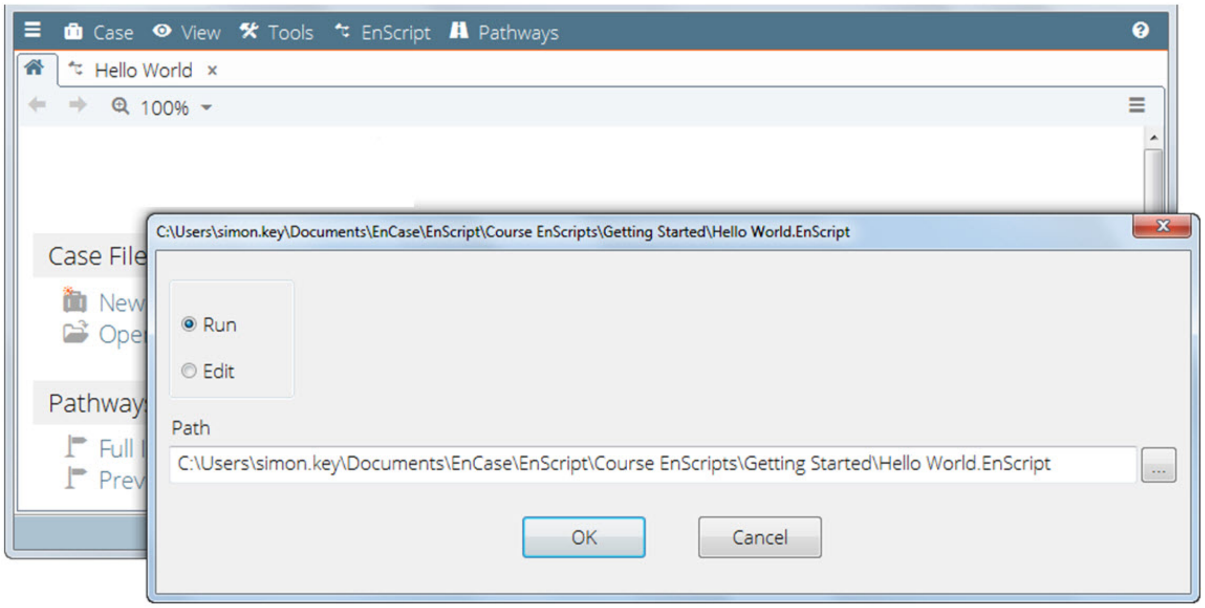
```
Hello World
```

Figure 2-11 Viewing the output of the script

It's worth pointing out that when we run a script from a session tab in this way, we're running it in debug mode.

EnCase users that don't develop scripts will hardly, if at all, use the session tabs; they will use the option on the main EnScript menu. Alternatively they can run a script simply by dragging and dropping it onto the EnCase GUI.

It’s worth pointing out that using the drag-and-drop method is also one of the quickest ways to edit a script.



C:\Users\simon.key\Documents\EnCase\EnScript\Course EnScripts\Getting Started\Hello World.EnScript

Run [✓]

Edit

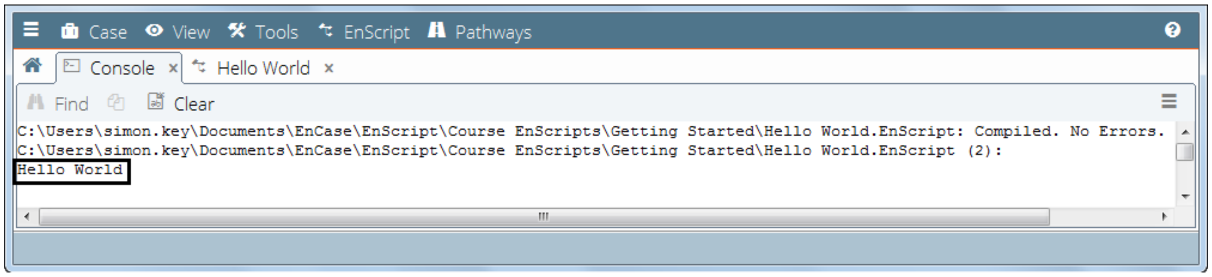
Path

C:\Users\simon.key\Documents\EnCase\EnScript\Course EnScripts\Getting Started\Hello World.EnScript

Figure 2-12 Running a script by dragging and dropping

Note that dragging and dropping a script is best done onto the EnCase title bar at the top of the program window. Dragging and dropping a script onto the EnScript editor tab will cause the file to be added to the current project as another source code file.

Regardless of which of the two aforementioned methods is used, users will need to switch to the Console tab manually in order to view any output made thereto; this isn’t done automatically.



```
C:\Users\simon.key\Documents\EnCase\EnScript\Course EnScripts\Getting Started\Hello World.EnScript: Compiled. No Errors.
C:\Users\simon.key\Documents\EnCase\EnScript\Course EnScripts\Getting Started\Hello World.EnScript (2):
Hello World
```

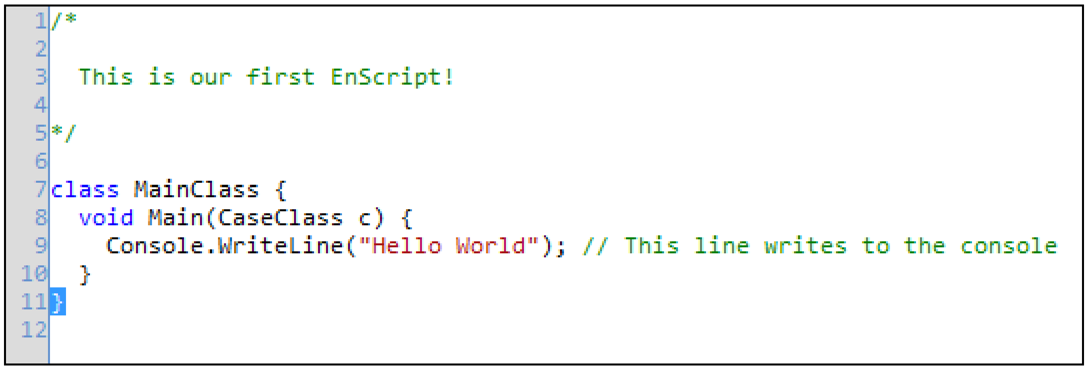
Figure 2-13 Viewing the main console tab

Script Comments

Comments can be added to a script in one of two ways.

Placing two forward slashes (//) on a line will cause the compiler to treat the remainder of the line as a comment.

Alternatively, the beginning and end of a comment (including a multi-line comment) can be marked as such using the /* and */ character sequences respectively.



```
/*

  This is our first EnScript!

*/

class MainClass {
  void Main(CaseClass c) {
    Console.WriteLine("Hello World"); // This line writes to the console
  }
}

```

Figure 2-14 Using comments

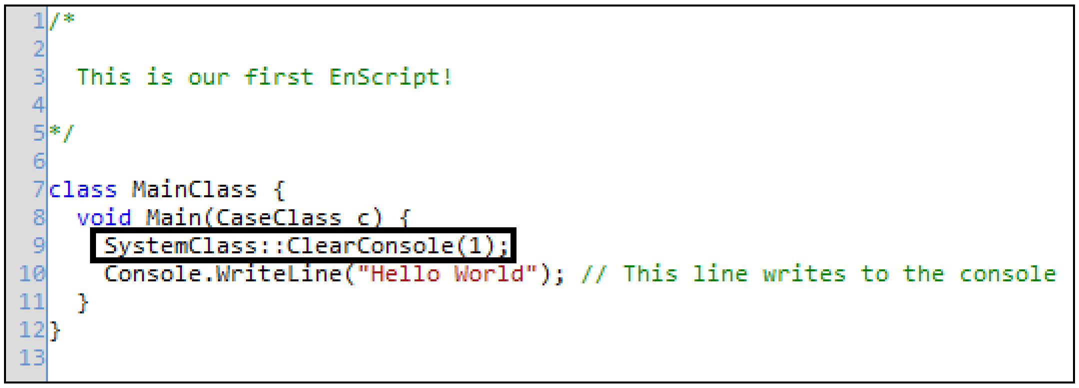
A good way of isolating code that you think is causing an error is to mark it as a comment; that way it won't get compiled.

It's also a good idea to mark what you think might be redundant code as a comment until you're absolutely sure you that don't need it. Many programmers have deleted code that they think they no longer need only to have to retype it all later!

Clearing the Console Window

You may have noticed that running our new script doesn't clear the console automatically. This makes it that little bit harder to check that our code is performing as it should.

Let's add a line of code that will cause our script to clear the console whenever we run it.



```
/*

  This is our first EnScript!

*/

class MainClass {
  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);
    Console.WriteLine("Hello World"); // This line writes to the console
  }
}

```

Figure 2-15 Writing code to clear the console

Running the script now causes the console to be cleared automatically, but how does our new line of code actually accomplish this?

Well, all we're doing is calling a function called "ClearConsole()" and giving it the integer value of 1, which tells EnCase that it should switch to the console window after the script has finished running.

It's very common to use numeric values to control the way in which functions behave because numbers are easy for a computer to interpret and don't take up as much memory to store as other values, such as strings.

If you're thinking that it will be difficult to remember which numbers do what when it comes to controlling a given function, then you're absolutely correct. Thankfully there is a way to give numeric values a label, which makes it much easier to remember what each value does. We will see how that works later on.

Having spent some time discussing the scope operator (::), you may have realized that the ClearConsole() function is defined as belonging to SystemClass.

SystemClass is a good example of a class that is used to group similar functions together. Unlike FileClass, SystemClass is not designed to represent objects of a particular type so we would never create a SystemClass object and then apply a method to it using the period (.) character.

SystemClass gets used a lot in EnScript programming. It contains many useful functions, such as displaying a message box, displaying a folder or file path dialog, opening files in Microsoft® Windows, and obtaining a script's path and command-line arguments.

Programming Tips

Many tips that apply to other programming languages apply to EnScript. This is especially so in a classroom environment where the code you write may need to be debugged by your instructor.

Taking this into account, try and stick to the following:

- Plan what you're going to do before you start writing any code. This will make large programming tasks much easier.
- Split your code up into manageable chunks using functions, which we shall explore a little later.
- Develop functions that handle intricate tasks separately. When they're written and functioning correctly, then you can add them to the rest of your code.
- If you're new to programming a particular language, write just one or two lines of code before compiling your script again.
- Always open and close parentheses (()), braces ({}), or square brackets ([]) together, then fill out their content afterwards. Failure to do this can lead to bugs that are extremely difficult to fix. This is especially so with braces; the compiler won't usually detect a missing closing brace until it reaches the end of the script!
- Take the time to format your code consistently and indent each block of code. This will make it much easier for someone else to review (and maybe fix) your code.

- Make use of whitespace so that you don't write long lines of code that have to be viewed by scrolling horizontally.
- Use copious script comments; you'll be surprised how difficult it is to review your code even a short time later.
- Use comments to isolate potential errors when debugging.
- Use comments to comment out redundant code until you're absolutely sure that you don't need it.
- Don't take ages staring at code that won't work or run properly – try and find a suitable person to review it. They'll often find the problem before you're even finished telling them what it is!
- Don't program when you're overtired.
- Use a version control system, such as TortoiseSVN [1] in order to track changes to your EnScript program and revert them if necessary. Your instructor may be able to demonstrate this for you.

CAN I COMPILE AND/OR RUN ENSCRIPT APPLICATIONS WITHOUT ENCASE?

No. If you don't have an EnCase licence and wish to develop EnScript applications, you may wish to sign up with the EnCase Developer Network as mentioned in a previous lesson.

WILL ENCASE INCLUDE A FULLY FEATURED DEVELOPMENT ENVIRONMENT IN FUTURE RELEASES?

A common request from EnScript developers is to have a fully featured development environment that supports auto-completion.

This is something that we're hoping to include, but we cannot guarantee if and when this will take place.

[1] http://tortoisesvn.tigris.org/

NOTES

NOTES

VARIABLES AND FUNCTIONS

INTRODUCTION TO VARIABLES

The whole point of any computer program is for it to receive input and provide output. In order to accomplish this, a computer program must be able to store the data received and generated at the time that it runs.

OpenText™ EnCase™ software (EnCase), for instance, has no idea as to the amount of data that it will need to process until you, the user, tells it what to do.

If you tell EnCase to create a new case and add some evidence to it, EnCase must be able to store that data in memory so that it can perform the tasks that you expect of it.

This concept of being able to store data at run time applies equally to any EnScript application you write - you must be able to store data so that you can work with it.

Taking this into account, the EnScript language (EnScript), like all other computer languages, enables you to store information, in memory, in the form of *variables*.

Each variable will have a name, type, and location in memory. Variable names cannot start with a number; they cannot be a reserved keyword, nor can they include characters used as operators.

EnScript is strongly-typed, which means that if we want to use a variable we have to declare both its name ***and*** type beforehand.

There is no support in EnScript for accessing variable memory locations directly. In other words, EnScript has no support for pointers of the type used by C and C++.

There are two main types of variable in EnScript: *fundamental types* and *object variables*.

Fundamental Variable Types

Fundamental variables are those variables for which there is native support in EnScript.

Fundamental variables in EnScript have the following characteristics:

- They're allocated memory automatically. To use one you just declare its name and type; you can then set its value.
- They have a default value. With the odd exception, languages such as C and C++ don't clear the memory allocated to variables when they're declared. This will often result in a variable containing garbage so it's necessary to initialize that variable's value before it can be safely read. This is unnecessary with EnScript and will actually slow your script down.
- Conversions between fundamental types are handled automatically. For instance, the FileClass::WriteLine() function, which takes a string parameter, will work fine if given a variable of another fundamental type, such as a numeric variable. This is because that variable will be converted automatically before the function has a chance to process it.

Numeric Variable Types

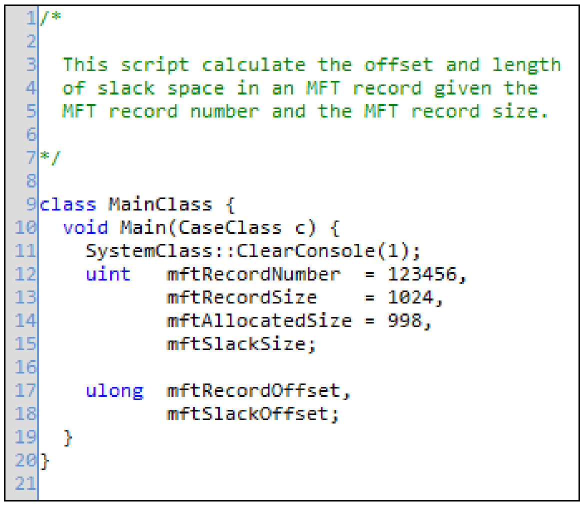
EnScript has support for the following fundamental, numeric variable types:

Type	Bytes	Minimum Value	Maximum Value
byte	1	0	255
short	2	-32,768	32,767
ushort	2	0	65535
int	4	-2,147,483,648	2,147,483,647
uint	4	0	4,294,967,295
long	8	-9,223,372,032,559,808,513	9,223,372,032,559,808,512
ulong	8	0	18,446,744,065,119,617,025
double	8	1.7E-308	1.7E+308

Figure 3-1 Numeric variable types supported in EnScript

The default value for each of these types is zero.

In order to explore how variables are declared and assigned, let's take a look at the following example.



```
 1 /*
 2
 3   This script calculate the offset and length
 4   of slack space in an MFT record given the
 5   MFT record number and the MFT record size.
 6
 7 */
 8
 9 class MainClass {
10   void Main(CaseClass c) {
11     SystemClass::ClearConsole(1);
12     uint   mftRecordNumber   = 123456,
13            mftRecordSize     = 1024,
14            mftAllocatedSize = 998,
15            mftSlackSize;
16
17     ulong  mftRecordOffset,
18            mftSlackOffset;
19   }
20 }
21
```

Figure 3-2 Declaring and assigning numeric variables

As we can see from the comment, this script is concerned with calculating the offset and length of slack space in an MFT record given the MFT record's size and length of allocated data; the offset of the slack space will be calculated from the start of the $MFT file.

Note that we will, wherever possible, try to use script examples that reflect code tasks that you may wish to perform when examining computer data in a forensic context.

In this case, the code we have written could well be used as part of a script that identifies resident data of a file that has since become non-resident; this might reveal a small amount of "deleted" data of evidential value.

Breaking the code down line-by-line:

- Line 11 clears the console as documented in a previous lesson.
- Lines 12 to 15 declare four uint variables for the purpose of storing the MFT record number, its total size, allocated size, and its slack size. The first three of these variables are allocated values, using the assignment operator (=), which you should note as being different from the operator used to check for equivalence (==). Using a comma avoids the need to declare each uint variable separately and saves time. The last of the four variables, mftSlackSize, will be used to store one of the three values that we will need to calculate in order for the script to perform its function.
- Lines 17 and 18 declare two ulong variables for the purpose of storing the offset of an MFT record and the offset of its slack space. These will be used to store the other two values that will be calculated by the script. Note that in this case we have used ulong variables so as to reflect the fact that the $MFT file might possibly be larger than could be represented by a uint value.
- All of the variables declare by the code are unsigned, i.e., they cannot take a negative value. This is absolutely fine because none of the values that we're working with will be negative; they will always have a value greater than or equal to zero.

It should be noted that three of the variables have been assigned constant values. In reality we might obtain these values in one of a number of ways:

- The mftRecordNumber value might be obtained by iterating through the entries chosen by the user in the EnCase GUI in which case our script would have to perform its calculations more than once.
- We should, in theory, calculate the value of mftRecordSize by reading the header of the associated NTFS volume. That said, we can, in normal circumstances, assume that it will be 1,024 bytes.
- When it comes to calculating the value of mftAllocatedSize, we would need to read this by reading the MFT record header, which would require us to read the associated $MFT as a file. Reading the data associated with an entry representing a file will be covered during a later lesson.

All of these points notwithstanding, the basic function of our EnScript program is to perform a numeric calculation; it really matters not where our numbers come from. So, taking this into account, we can proceed with writing the rest of our code.

As part of this process, we need to consider the possibility that the values with which our code will work will need some form of validation. For this we will need to use one or more *conditional statements*.

CONDITIONAL STATEMENTS

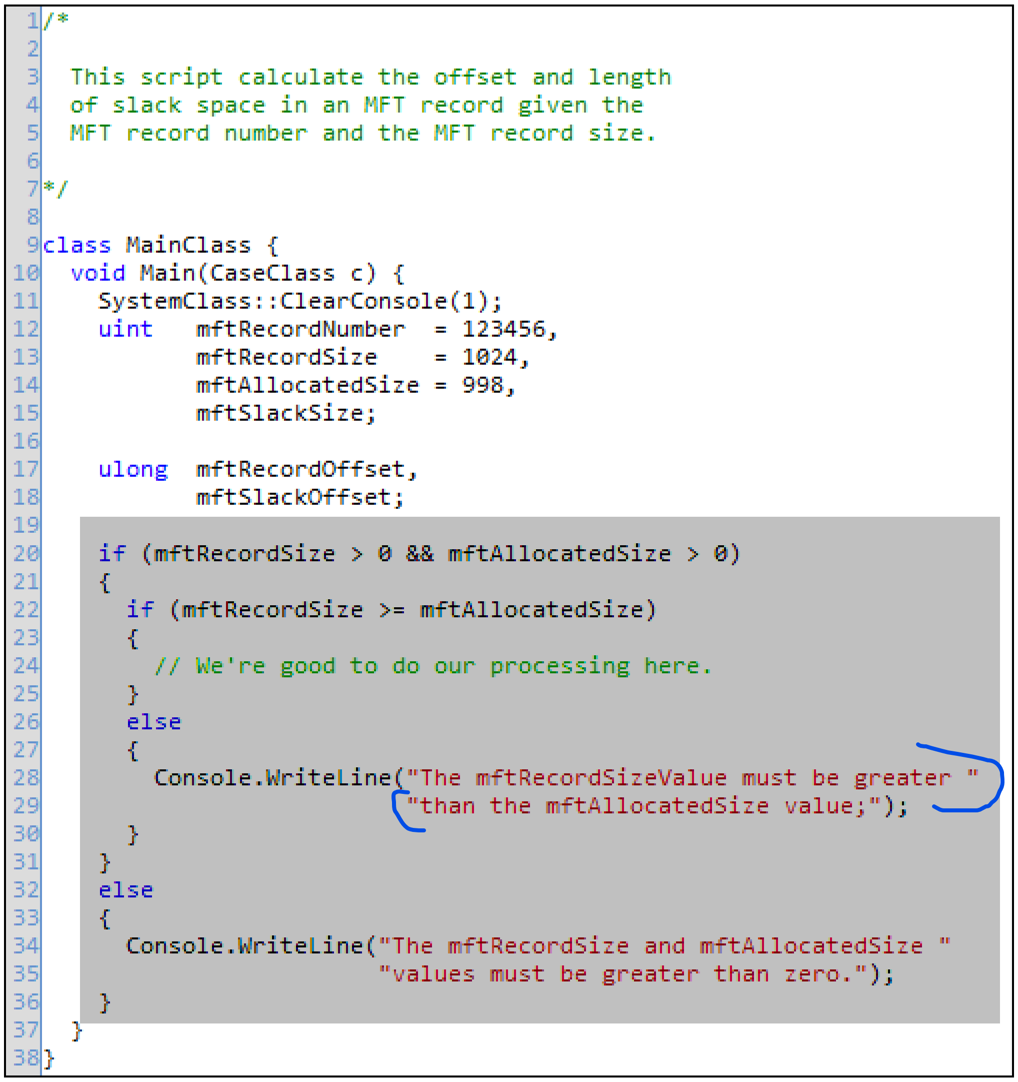
Our EnScript code will need to use values stored in the mftAllocatedSize and mftRecordSize variables in order to calculate length of MFT slack. This is a simple numeric calculation, which subtracts the former value from the latter.

That said, the values of these variables might, in certain circumstances, be incorrect. For instance, what if the values originated from the user and were invalid in some way? Or perhaps we had tried to read the values ourselves from the $MFT file but had done something wrong?

In such cases we can determine one or more conditions, which will control the course of action our EnScript will take.

So in our case, we might want to ensure that the value contained in the mftRecordSize variable is greater or equal to that contained in the mftAllocatedSize variable. We might also want to make sure that both values were non-zero.

To do this we can use two conditional *if-else* statements as shown in the following screenshot.



```
/*

  This script calculate the offset and length
  of slack space in an MFT record given the
  MFT record number and the MFT record size.

*/

class MainClass {
  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);
    uint    mftRecordNumber  = 123456,
            mftRecordSize    = 1024,
            mftAllocatedSize = 998,
            mftSlackSize;

    ulong   mftRecordOffset,
            mftSlackOffset;

    if (mftRecordSize > 0 && mftAllocatedSize > 0)
    {
      if (mftRecordSize >= mftAllocatedSize)
      {
        // We're good to do our processing here.
      }
      else
      {
        Console.WriteLine("The mftRecordSizeValue must be greater "
                          "than the mftAllocatedSize value;");
      }
    }
    else
    {
      Console.WriteLine("The mftRecordSize and mftAllocatedSize "
                        "values must be greater than zero.");
    }
  }
}
```

Figure 3-3 Using conditional if-else statements

Conditional if-else statements take the form:

if (*condition*)

Single statement or block

else if (*condition*)

Single statement or block

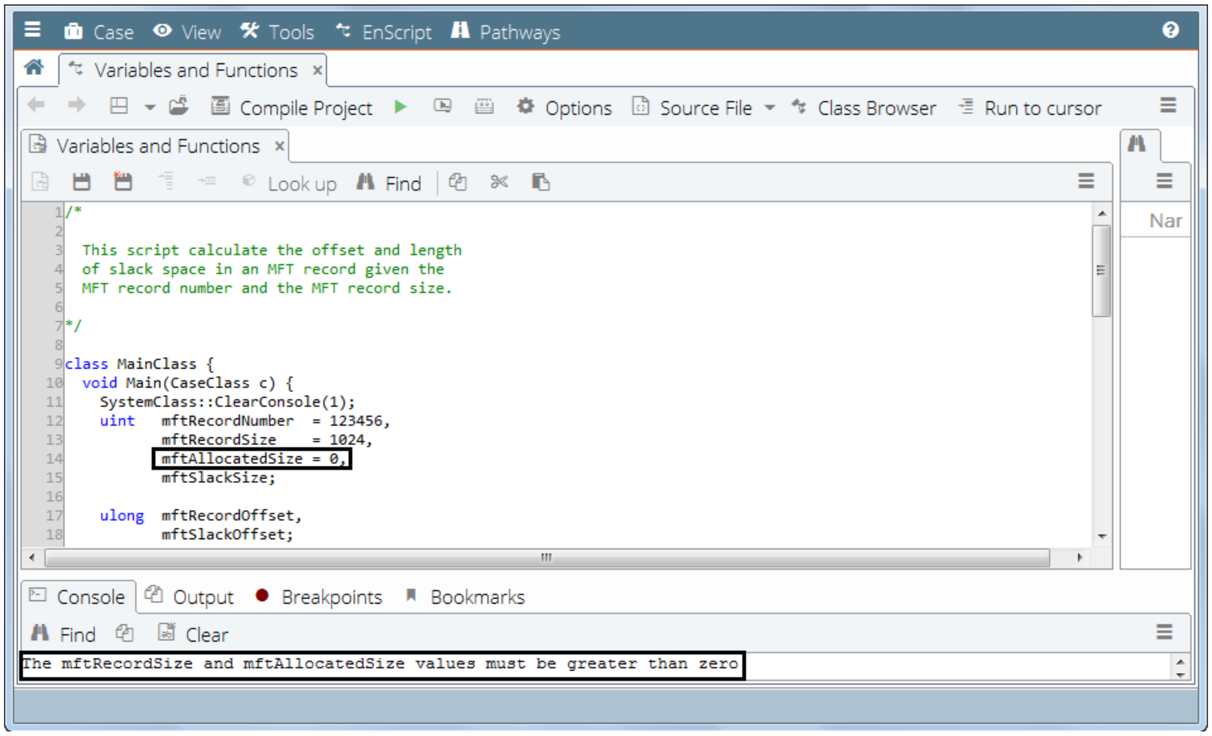
else

Single statement or block

We can break down our updated code as follows:

- Lines 20 to 36 mark the outermost *if-else* block, one that checks to see if the mftRecordSize and mftAllocatedSize variables are greater than zero. The operator that accomplishes this is the greater-than (>) operator. This is used twice, the output of both comparisons being joined through the use of the logical AND operator (&&). It's worth noting that:
  - The '&&' and '>' operators are processed according to the standard rules of operator precedence. These would take some time to cover in depth, something that we don't want to do here. If you need to ensure that operators are processed in a particular way, and you aren't sure of the rules of operator precedence, then you are advised to use parentheses.
  - The EnScript compiler combines the logical AND (&&) and OR (||) operators in a way that avoids unnecessary processing. For instance, if two values are combined using the logical AND operator (&&), the second value won't be evaluated if the first value evaluates as false. This is because the second value will make no difference to the overall calculation even if it is true.
- Lines 22 to 30 mark the innermost *if-else* block. It uses the greater-than-or-equals operator (>=) and will only be processed if the outer *if* block is evaluated as true.
- Lines 28 and 34 both use Console.WriteLine() to write an error (of sorts) to the console window. Note that both these lines demonstrate the correct way of splitting a constant string across multiple lines. Use of the string concatenation operator (+) is to be avoided; this will be discussed in more depth later.
- Line 24 will only be reached if both if blocks are evaluated as true. At that point we will have validated the mftRecordSize and mftAllocatedSize variables to our satisfaction.

We're now in a position to ensure that our code validates the two variables as we expect that it should. We can do this by manipulating the values of the variables, rerunning the script each time.



```
/*

 This script calculate the offset and length
 of slack space in an MFT record given the
 MFT record number and the MFT record size.

*/

class MainClass {
  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);
    uint   mftRecordNumber  = 123456,
           mftRecordSize    = 1024,
           mftAllocatedSize = 0,
           mftSlackSize;

    ulong  mftRecordOffset,
           mftSlackOffset;
```

```
The mftRecordSize and mftAllocatedSize values must be greater than zero
```

Figure 3-4 Validating the script

This works as expected, but the nested *if-else* blocks make our code a little harder to read (and edit). We could combine the two blocks into one, but that would lead to a lengthy *if* condition, which is something many programmers aren't keen-on.

A better option would be to encapsulate our validation routine into a *function*.

FUNCTIONS

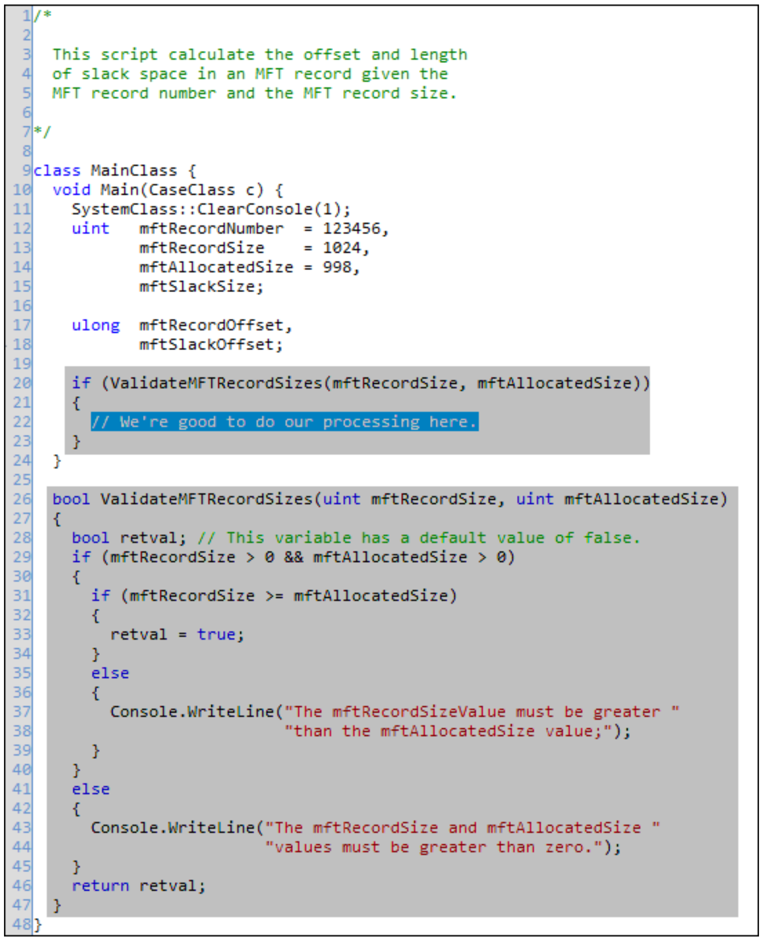
We have already encountered two internal EnScript functions:

- FileClass::WriteLine(). We have used this several times in order to write information to the console window.
- SystemClass::ClearConsole(). We have used this to clear the console window.

Functions allow us to segregate code such that:

- The code can be called multiple times and from different locations in our EnScript program.
- The code is easier to edit. If we have copies of the same code in different locations and we want to fix/update all of the occurrences of that code then that will involve more work.
- The code is more portable. As we shall see later, we can move it into a class, which we can store in an EnScript library file. That file can then be used by more than one EnScript program.

Taking this into account, we can now rewrite our EnScript program as shown in the subsequent figure.



```
/*

  This script calculate the offset and length
  of slack space in an MFT record given the
  MFT record number and the MFT record size.

*/

class MainClass {
  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);
    uint   mftRecordNumber   = 123456,
           mftRecordSize     = 1024,
           mftAllocatedSize = 998,
           mftSlackSize;

    ulong  mftRecordOffset,
           mftSlackOffset;

    if (ValidateMFTRecordSizes(mftRecordSize, mftAllocatedSize))
    {
      // We're good to do our processing here.
    }
  }

  bool ValidateMFTRecordSizes(uint mftRecordSize, uint mftAllocatedSize)
  {
    bool retval; // This variable has a default value of false.
    if (mftRecordSize > 0 && mftAllocatedSize > 0)
    {
      if (mftRecordSize >= mftAllocatedSize)
      {
        retval = true;
      }
      else
      {
        Console.WriteLine("The mftRecordSizeValue must be greater "
                          "than the mftAllocatedSize value;");
      }
    }
    else
    {
      Console.WriteLine("The mftRecordSize and mftAllocatedSize "
                        "values must be greater than zero.");
    }
    return retval;
  }
}
```

Figure 3-5 Including a validation function

Our EnScript program now has a new validation function: ValidateMFTRecordSizes().

We can interpret our updated code as follows:

- Lines 26 to 47 represent the new function. It returns a *Boolean* value on line 46, indicating if the validation was successful or not. Rather than using multiple return statements, line 28 of the function declares a single Boolean value called "retval," which has an initial value of false. The retval value will only set as true if line 33 is reached; this requires both *if* statements to be evaluated as true.

  If you start your function with two lines that declare and then return a variable of the type specified by the function definition, you will satisfy the compiler and can write the remaining function code, testing it as you go.

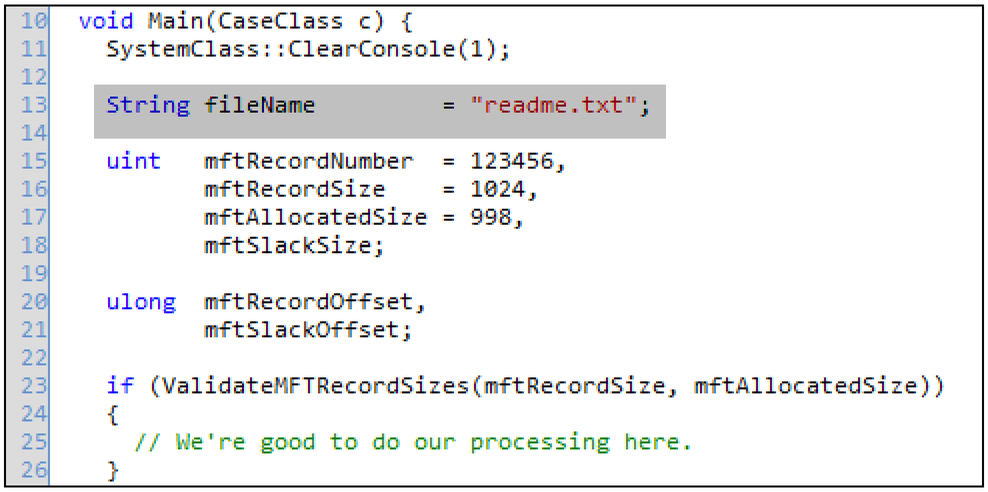
- The function declaration on line 26 declares two uint function parameters: mftRecordSize and mftAllocatedSize.

  As we shall see later, these parameters are separate and distinct from the variables of the same name that declared in in the Main() function; the parameters do not exist (or have *scope*) outside of the ValidateMFTRecordSizes() function.

  What we are actually doing is passing-in a *copy* of each variable to the function so that it can work with its value. We use the same variable names because it makes it easier to understand what the function is doing. Regardless, when the function finishes its work, the copy of each variable will be destroyed.

Our EnScript program is now much tidier and we can pretty much ignore our validation function unless we need to update it later.

We're now in a position to proceed with our calculation, but before we do that let's add a string variable representing the (fictitious) name of the file to which MFT record number 123456 relates. We will allocate that variable the value "readme.txt."



```
void Main(CaseClass c) {
  SystemClass::ClearConsole(1);

  String fileName           = "readme.txt";

  uint    mftRecordNumber   = 123456,
          mftRecordSize     = 1024,
          mftAllocatedSize = 998,
          mftSlackSize;

  ulong   mftRecordOffset,
          mftSlackOffset;

  if (ValidateMFTRecordSizes(mftRecordSize, mftAllocatedSize))
  {
    // We're good to do our processing here.
  }
```

Figure 3-6 Adding a string variable

STRING VARIABLES

A string variable can be empty; it can also contain one or more characters, which are stored internally by EnCase as Unicode.

A string will contain two bytes per character plus a terminating null character, which will consist of two null (zero) bytes.

An empty string will contain nothing but the null character.

Strings are very easy to work with in EnCase, but you should not lose sight of the fact that there is quite a lot of work going on behind the scenes to make that possible. Also, strings take up more memory than you might think and can be slow to manipulate.

For instance, take the strings “Guidance” and “Software” and “Inc.”

These strings contain eight, eight, and four characters, respectively; that is a total of twenty characters.

A string *n*-characters in length will occupy *(n * 2) + 2* bytes in memory. The strings mentioned previously therefore occupy eighteen, eighteen, and ten bytes, respectively.

It is possible to add, or concatenate, strings together using the addition operator (+).

This should however be avoided because each concatenation operation is handled separately. Take the following EnScript statement:

```
String val = “Guidance” + “Software” + “Inc.”;
```

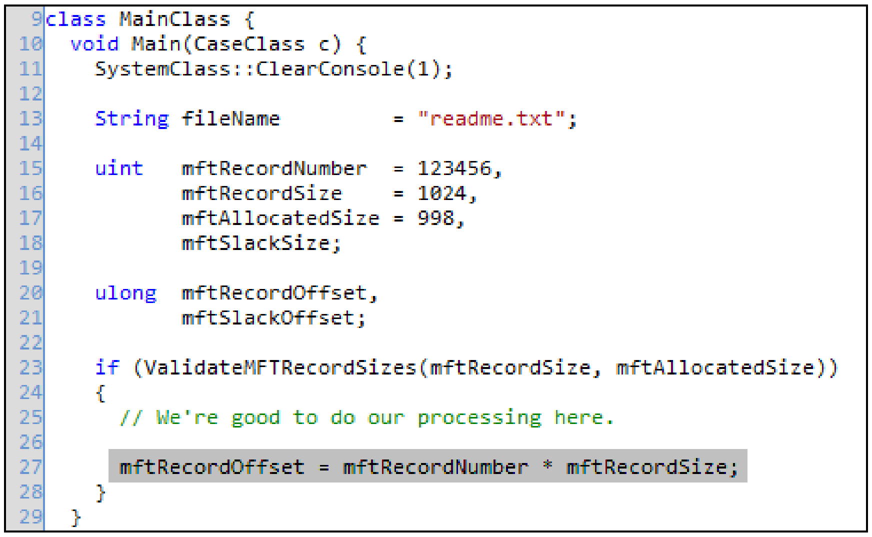
EnCase would first concatenate the strings “Guidance” and “Software.” To do so it would use thirty-four bytes of memory, which is the space needed for a total of sixteen Unicode characters plus a terminating null character.

Having performed the first concatenation, EnCase would then have to repeat the process in order to concatenate the string “Inc.” onto the string “GuidanceSoftware.” That would result in a final memory allocation of forty-two bytes. The memory containing the intermediary result would no longer be needed and would have to be de-allocated.

So, taking this into account, it can be seen that concatenating three or more strings will result in the use of intermediate memory, which can slow-down a script considerably.

Because of this, EnScript provides a number of methods for concatenating strings more efficiently. We will explore the first of the methods when we come to writing the result of our EnScript program to the console window.

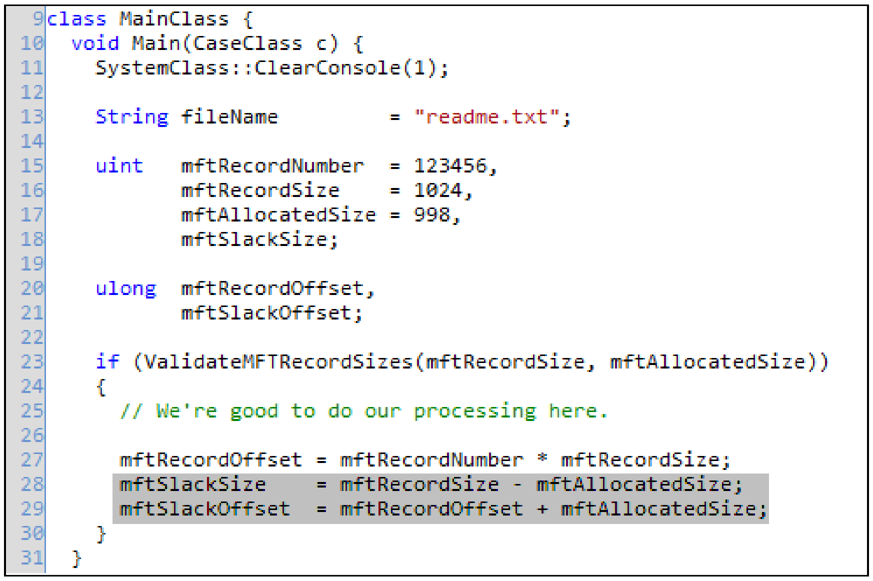
This first step to calculating our final result is to calculate the offset to the MFT record in question, so let's update our code as the following screenshot displays.



```
class MainClass {
  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);

    String fileName         = "readme.txt";

    uint   mftRecordNumber  = 123456,
           mftRecordSize    = 1024,
           mftAllocatedSize = 998,
           mftSlackSize;

    ulong  mftRecordOffset,
           mftSlackOffset;

    if (ValidateMFTRecordSizes(mftRecordSize, mftAllocatedSize))
    {
      // We're good to do our processing here.

      mftRecordOffset = mftRecordNumber * mftRecordSize;
    }
  }
```

Figure 3-7 Calculating the MFT record offset

We can now calculate the size and offset of slack space for the MFT record in question. Remember that the offset is being calculated from the start of the $MFT file, not from the start of the record.



```
class MainClass {
  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);

    String fileName         = "readme.txt";

    uint   mftRecordNumber  = 123456,
           mftRecordSize    = 1024,
           mftAllocatedSize = 998,
           mftSlackSize;

    ulong  mftRecordOffset,
           mftSlackOffset;

    if (ValidateMFTRecordSizes(mftRecordSize, mftAllocatedSize))
    {
      // We're good to do our processing here.

      mftRecordOffset = mftRecordNumber * mftRecordSize;
      mftSlackSize    = mftRecordSize - mftAllocatedSize;
      mftSlackOffset  = mftRecordOffset + mftAllocatedSize;
    }
  }
```

Figure 3-8 Calculating the offset and size of MFT slack

Having calculated the numeric values that we're interested in, we now need to write them to the console in some form of comprehensible format, using a method that avoids the string concatenation problem mentioned earlier.

Writing Formatted Output

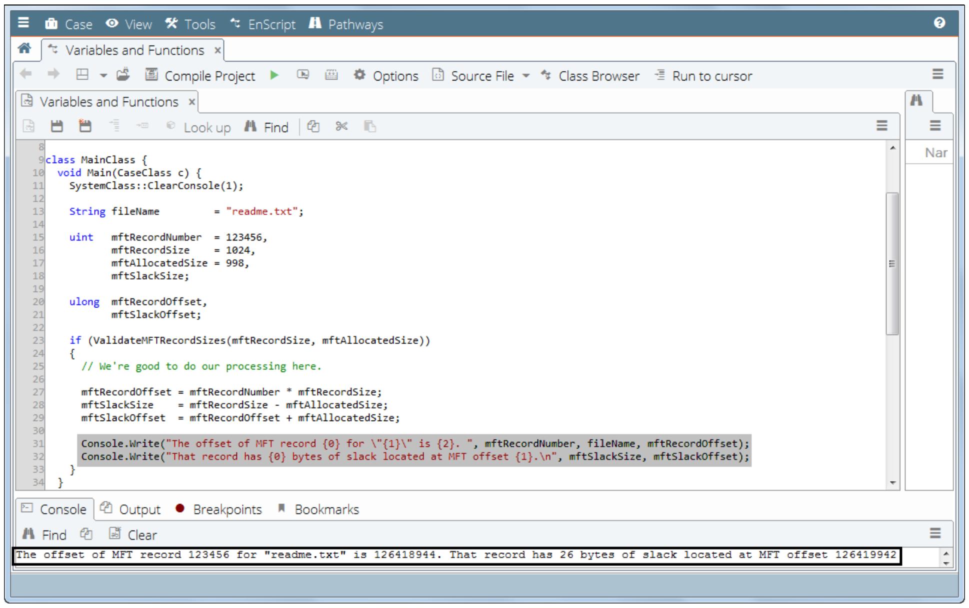
Writing formatted output is best accomplished using the FileClass::Write() or FileClass::WriteLine() methods.

The FileClass::Write() function is very similar to the FileClass::WriteLine() function that we've used to write information to the Console object in EnCase – remember that object is a type of file or FileClass object.

The only difference between the two methods is that FileClass::Write() does not append a newline character to whatever FileClass object is being written to.

Different versions of the FileClass::Write() and FileClass::WriteLine() methods allow us to write a formatted string containing placeholders for up to another three string values. The three placeholders are identified in the formatting string as {0}, {1} and {2}. The values to be written into those placeholders are passed in as additional parameters to the Write() and WriteLine() functions. The order in which they're written is derived from the order in which they're passed-in as parameters.

So we can now update our code as shown in Figure 3-9.



```
 8
 9class MainClass {
10  void Main(CaseClass c) {
11    SystemClass::ClearConsole(1);
12
13    String fileName         = "readme.txt";
14
15    uint   mftRecordNumber  = 123456,
16           mftRecordSize    = 1024,
17           mftAllocatedSize = 998,
18           mftSlackSize;
19
20    ulong  mftRecordOffset,
21           mftSlackOffset;
22
23    if (ValidateMFTRecordSizes(mftRecordSize, mftAllocatedSize))
24    {
25      // We're good to do our processing here.
26
27      mftRecordOffset = mftRecordNumber * mftRecordSize;
28      mftSlackSize    = mftRecordSize - mftAllocatedSize;
29      mftSlackOffset  = mftRecordOffset + mftAllocatedSize;
30
31      Console.Write("The offset of MFT record {0} for \"{1}\" is {2}. ", mftRecordNumber, fileName, mftRecordOffset);
32      Console.Write("That record has {0} bytes of slack located at MFT offset {1}.\n", mftSlackSize, mftSlackOffset);
33    }
34  }
```

```
The offset of MFT record 123456 for "readme.txt" is 126418944. That record has 26 bytes of slack located at MFT offset 126419942
```

Figure 3-9 Calculating the offset and size of MFT slack

Lines 31 and 32 both use Console.Write() with the formatted string method. This avoids string concatenation and is much easier to read in the long term.

Note that the formatting string used on line 31 surrounds the file-name in double quotes. These have to be escaped, using the backslash character (\) because otherwise the EnScript compiler would read them as string delimiters.

Note also that line 32 uses the special '\n' character to write a new line after the output has been written. We could have avoided this by using Console.WriteLine() on line 32 rather than Console.Write();

CHAR VARIABLE TYPES

The char variable type represents a single character stored in Unicode using two bytes.

We won’t cover it in any more depth at this point other than to say that the difference between single and double quotes in EnScript is that single quotes are used with char variables; double quotes are used with string variables.

OTHER OPERATORS

During the course of this lesson we have used a small number of operators including ‘=’, ‘>’, ‘+’, ‘-’, ‘>=’, ‘&&’ and ‘*’. We also mentioned the logical OR operator (||).

There are a number of other operators that can be used but it’s not necessary to document them all here. We will have the opportunity to demonstrate the majority of commonly used operators during the remainder of the course.

NOTES

NOTES

NOTES

Lesson 4

LOOPS AND ADVANCED FUNCTIONS

INTRODUCTION TO LOOPS

A good majority of EnScript programs (EnScript) involve repetitive processing, i.e., they perform one or more tasks multiple times.

For instance, an EnScript may read the records from a SQLite database file, writing each record into a tab-delimited spreadsheet file.

Alternatively an EnScript might iterate through the tagged items in the case, bookmarking and extracting each one to a file on the local system.

Any form of repetitive processing will require the use of one or more programming *loops*.

Loops are quite difficult to demonstrate in a real-life sense at an introductory-level because their use is typically linked with processing case data in some way. As we haven't yet covered the way in which case data is accessed, this is a little tricky to do in some depth.

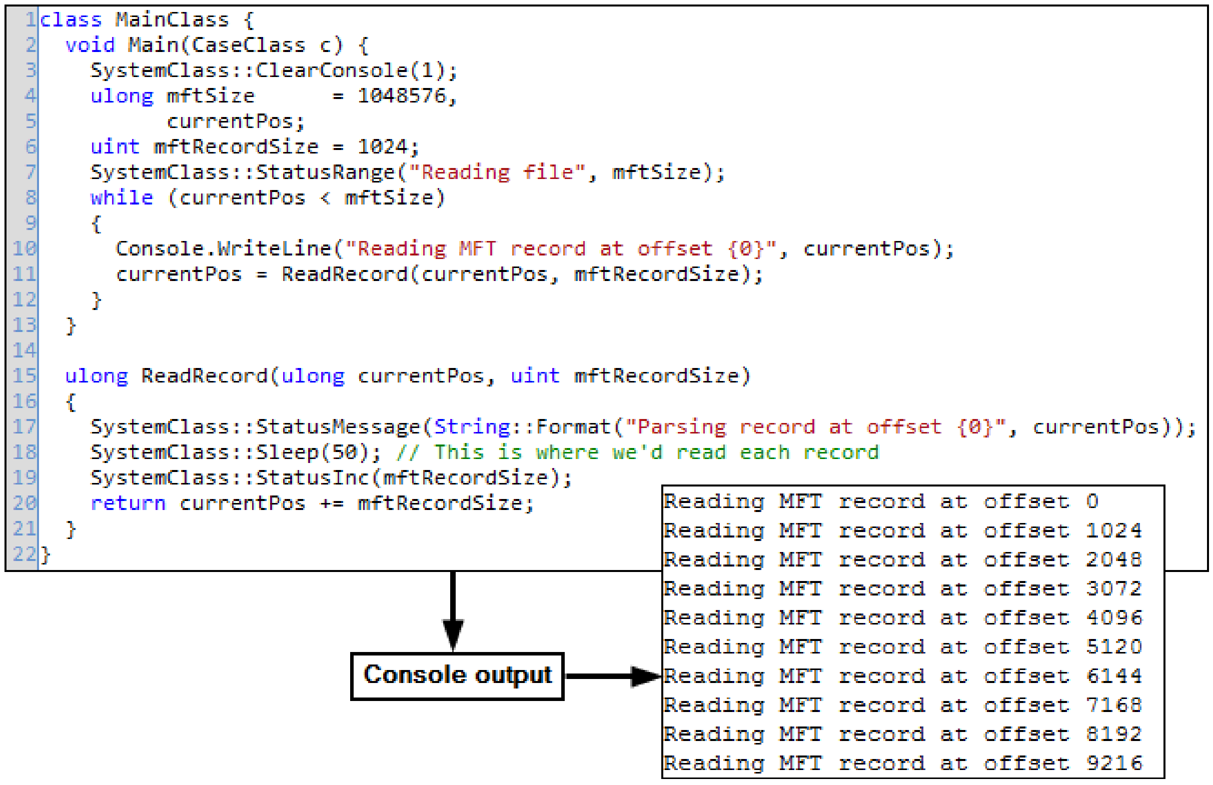
That said, we can demonstrate the basic principles behind the different types of loops and explain when they might be used.

USING THE WHILE LOOP

The while loop is used to perform a task one or more times until the condition controlling the loop is evaluated as false.

A very good use for the while loop is to read records from a file until the end of the file is reached.

We haven't covered reading file data as yet but we can simulate it, using the following displayed EnScript.



```
 1 class MainClass {
 2   void Main(CaseClass c) {
 3     SystemClass::ClearConsole(1);
 4     ulong mftSize       = 1048576,
 5           currentPos;
 6     uint mftRecordSize = 1024;
 7     SystemClass::StatusRange("Reading file", mftSize);
 8     while (currentPos < mftSize)
 9     {
10       Console.WriteLine("Reading MFT record at offset {0}", currentPos);
11       currentPos = ReadRecord(currentPos, mftRecordSize);
12     }
13   }
14
15   ulong ReadRecord(ulong currentPos, uint mftRecordSize)
16   {
17     SystemClass::StatusMessage(String::Format("Parsing record at offset {0}", currentPos));
18     SystemClass::Sleep(50); // This is where we'd read each record
19     SystemClass::StatusInc(mftRecordSize);
20     return currentPos += mftRecordSize;
21   }
22 }
```

Console output

```
Reading MFT record at offset 0
Reading MFT record at offset 1024
Reading MFT record at offset 2048
Reading MFT record at offset 3072
Reading MFT record at offset 4096
Reading MFT record at offset 5120
Reading MFT record at offset 6144
Reading MFT record at offset 7168
Reading MFT record at offset 8192
Reading MFT record at offset 9216
```

Figure 4-1 Using the while loop

The functionality of the previously shown script can be explained as follows:

- The script simulates reading records from a $MFT file 1,048,576 bytes (1MB) in length. The length of the file is stored in the variable mftSize declared on line 4.
- Each record is defined as being 1,024 bytes in length. This is stored in the mftRecordSize variable declared on line 6.
- The script will keep reading records until it reaches the end of the file. The current position within the $MFT file is stored in a variable called currentPos declared on line 5.
- Line 7 sets the OpenText™ EnCase™ software (EnCase) status bar so as to represent a maximum value, equating to the size of the $MFT file being parsed. This is accomplished, using a SystemClass function called "StatusRange()," which takes two parameters: the maximum value that can be represented by the status bar and the initial status bar text.
- Line 8 marks the start of the while loop, which has a condition, specifying that the following block will only be executed if the current position within the $MFT file is less than its length. This prevents us from trying to read past the end of the file.
- Line 10 writes the current position in the $MFT file to the console window.

- Line 11 calls a function called "ReadRecord()," which simulates reading a record from the $MFT file. It accepts two parameters: the current position within the file and the size of the MFT records that the file contains.
- Each time the ReadRecord() function is called, it updates the EnCase status bar text by calling another SystemClass function called "StatusMessage()," which accepts a string.
- Because we want the offset of the current record to be displayed in the status bar in a comprehensive format, we use a String function called "Format()." It returns a formatted string in the same way that FileClass::Write() and FileClass::WriteLine() write a formatted string to a file. This is another way of avoiding time-consuming string concatenation.
- Line 18 puts the script into an efficient wait state for 50 milliseconds, using another SystemClass function called "Sleep()." This simulates the time that we might spend reading the record starting at the current position from the file. Were we not to include this line (or exclude it by way of a comment) then the script would run too fast for us to see what it's doing.
- Line 19 increments the status bar by the length of the MFT record that has just been read; this is accomplished by the SystemClass::StatusInc() function.
- Line 20 returns the updated file position, i.e,. the previous file position plus the length of the MFT record that has just been read.
- Line 11 stores the value returned by the ReadRecord() function in the currentPos variable. This effectively updates that variable so that it stores the new file position rather than the old file position.
- If the value of the updated currentPos variable is greater or equal to the size of the $MFT file, then the while loop's condition evaluates as false and the loop terminates.

Having reviewed the operation of this script, it wouldn't be unusual for you to ask the following questions:

1. The size of an MFT record is unlikely to vary – do I really need to keep passing it into the ReadRecord() function?
2. Is there a way that the ReadRecord() function can update the currentPos variable directly without having to return an updated value that then has to be used to set the currentPos variable manually?

One possible answer to the first question requires us to understand the concept of *local* versus *global* variables.

Global vs. Local Variables

As already mentioned during a previous lesson, every variable will occupy memory and have a label enabling us to refer to it.

In order to optimize the use of memory, and to minimize the number of variables in active use, variables in EnScript have a life or *scope*.

By limiting this scope we are able to use variables only where we need them. Once each variable has gone out of scope, EnCase will dispose of the variable and return its memory to the system so that it can be reused.

Not only does this method of variable management save memory, it also prevents every variable from being present at every point within our EnScript program. Were this not to be the case, we might have tens or even hundreds of active variables, each one of which would need a unique name.

A variable's scope is determined by the following rules:

- A variable has scope only within the block that's it's declared.
- A variable defined as a function parameter will only have scope within the block of code associated with that function.

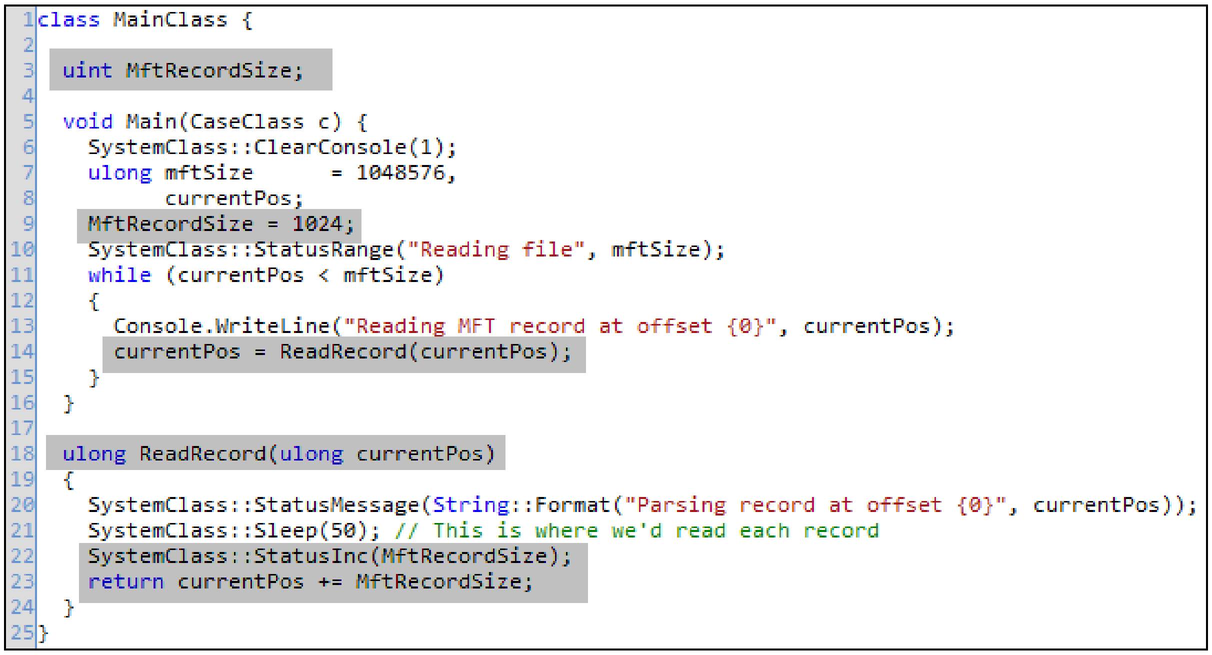
Taking this into account, the mftRecordSize variable declared at line 6 is separate and distinct from the mftRecordSize variable declared as a function parameter at line 15; we will demonstrate this later in the lesson.

If we want to make a variable containing the MFT record size to be accessible to both the Main() and ReadRecord() functions, then we need to declare that variable within a block that encapsulates those functions, i.e., the block that contains the definition for the whole of MainClass and the functions that it contains.

Declaring a variable at this level results in it having scope that is global within the parent class. This makes the variable a *global variable*.

A variable whose scope is limited to a function, or a block within a function, is referred to as a *local variable*.

So, putting this into practice…



```
 1 class MainClass {
 2
 3   uint MftRecordSize;
 4
 5   void Main(CaseClass c) {
 6     SystemClass::ClearConsole(1);
 7     ulong mftSize      = 1048576,
 8           currentPos;
 9     MftRecordSize = 1024;
10     SystemClass::StatusRange("Reading file", mftSize);
11     while (currentPos < mftSize)
12     {
13       Console.WriteLine("Reading MFT record at offset {0}", currentPos);
14       currentPos = ReadRecord(currentPos);
15     }
16   }
17
18   ulong ReadRecord(ulong currentPos)
19   {
20     SystemClass::StatusMessage(String::Format("Parsing record at offset {0}", currentPos));
21     SystemClass::Sleep(50); // This is where we'd read each record
22     SystemClass::StatusInc(MftRecordSize);
23     return currentPos += MftRecordSize;
24   }
25 }
```

Figure 4-2 Using global variables

We can explain our updated code as follows:

- Line 3 declares a global variable called "MftRecordSize." Note that many programmers distinguish global variables, using a name starting with an uppercase character; this is what has been done here.
- It isn't possible to declare and set the value of global variables in the same way as local variables so, taking this into account, the value of the MFTRecordSize global variable has been set on line 9. We will expand on the use of global variables when we document the creation and use of classes in more depth, also when we explain how to create custom dialog boxes.
- The declaration of the ReadRecord() function has been updated on line 18 so as to reflect the fact that the global MFTRecordSize variable is now available to both functions; it does not therefore need to be passed into the ReadRecord() function as a separate parameter. Line 14 has been modified likewise.

The script now functions exactly as before but is a little more concise.

We can now turn our attention to having the ReadRecord() function update the value of the Main() function's currentPos variable directly.

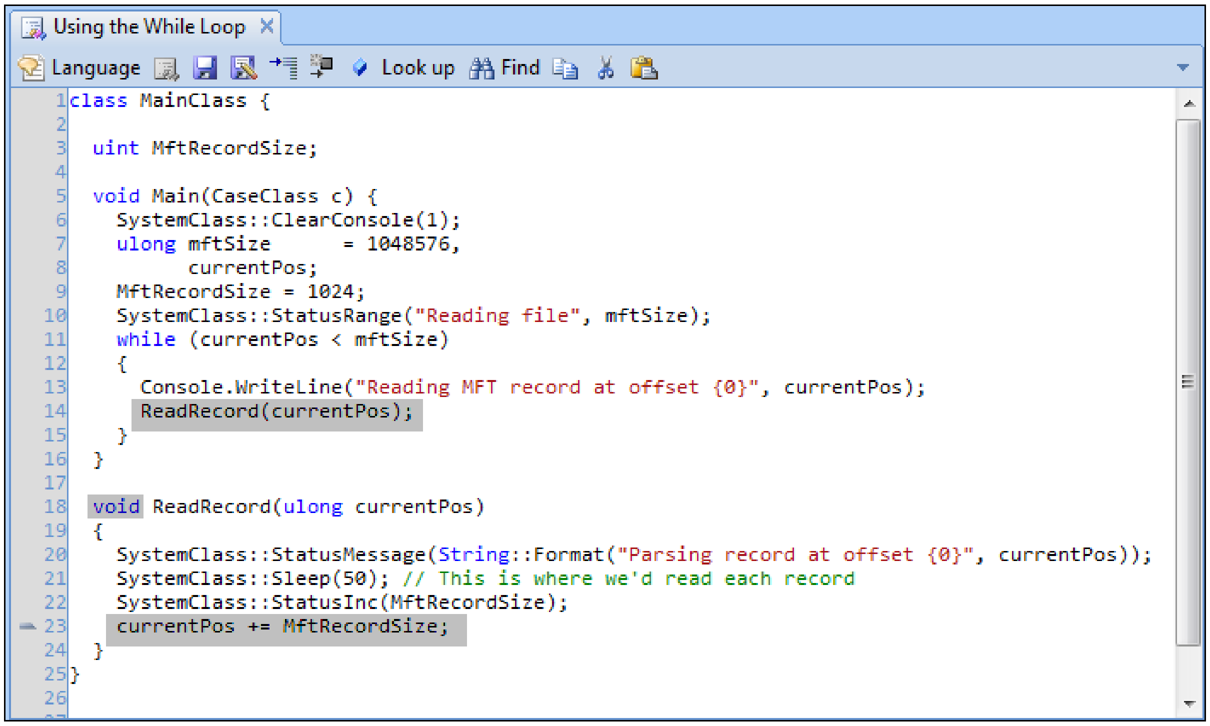
We could, in theory, use a global variable to accomplish that, but there is another method, one that sometimes has to be used with internal EnScript functions. These don't have access to our EnScript program's global variables – they can only access variables passed in as parameters.

The method that we are talking about here is passing variable into a function *by reference*.

Passing Variables Into Functions By Reference

As things stand at the moment, our ReadRecord() function is accessing a copy of the currentPos variable declared by the Main() function, not the variable itself. We chose to use the same name for the copy of the variable in the ReadRecord() function so as to make things a bit more understandable.

To test this, let's rewrite our script as though the currentPos parameter in the ReadRecord() function is the same as the currentPos variable in the Main() function.



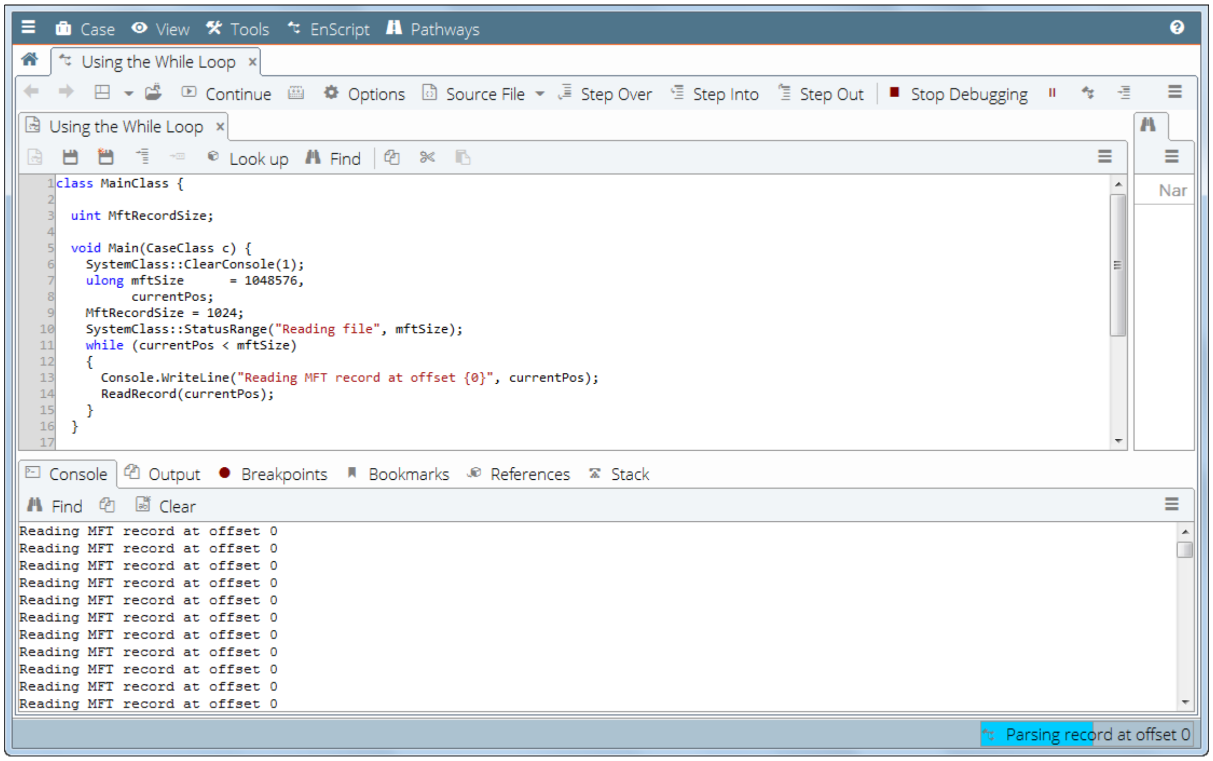
```
 1 class MainClass {
 2
 3   uint MftRecordSize;
 4
 5   void Main(CaseClass c) {
 6     SystemClass::ClearConsole(1);
 7     ulong mftSize      = 1048576,
 8           currentPos;
 9     MftRecordSize = 1024;
10     SystemClass::StatusRange("Reading file", mftSize);
11     while (currentPos < mftSize)
12     {
13       Console.WriteLine("Reading MFT record at offset {0}", currentPos);
14       ReadRecord(currentPos);
15     }
16   }
17
18   void ReadRecord(ulong currentPos)
19   {
20     SystemClass::StatusMessage(String::Format("Parsing record at offset {0}", currentPos));
21     SystemClass::Sleep(50); // This is where we'd read each record
22     SystemClass::StatusInc(MftRecordSize);
23     currentPos += MftRecordSize;
24   }
25 }
26
```

Figure 4-3 Viewing the EnScript program after a further modification

In the preceding example:

- The definition of the ReadRecord() function on line 18 now specifies its return value as being void, i.e., it doesn't return a function.
- Because of this, there is no longer a return statement in the ReadRecord() function. All that happens is that the currentPos variable gets incremented by the length of the MFT record just read (line 23).
- Line 14 has been updated to reflect the fact that the ReadRecord() function doesn't return any value.

Having made the changes, if the currentPos variable being accessed in the ReadRecord() function ***was*** the same as that referred to in the Main() function, the script would run just as before. The trouble is that it doesn't: the currentPos variable in the Main() function never gets updated; it always has a value of zero and the script never finishes. In order to stop the script we have to double click the status bar or click the Stop Debugging button.



```
class MainClass {

  uint MftRecordSize;

  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);
    ulong mftSize      = 1048576,
          currentPos;
    MftRecordSize = 1024;
    SystemClass::StatusRange("Reading file", mftSize);
    while (currentPos < mftSize)
    {
      Console.WriteLine("Reading MFT record at offset {0}", currentPos);
      ReadRecord(currentPos);
    }
  }

```

```
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
Reading MFT record at offset 0
```

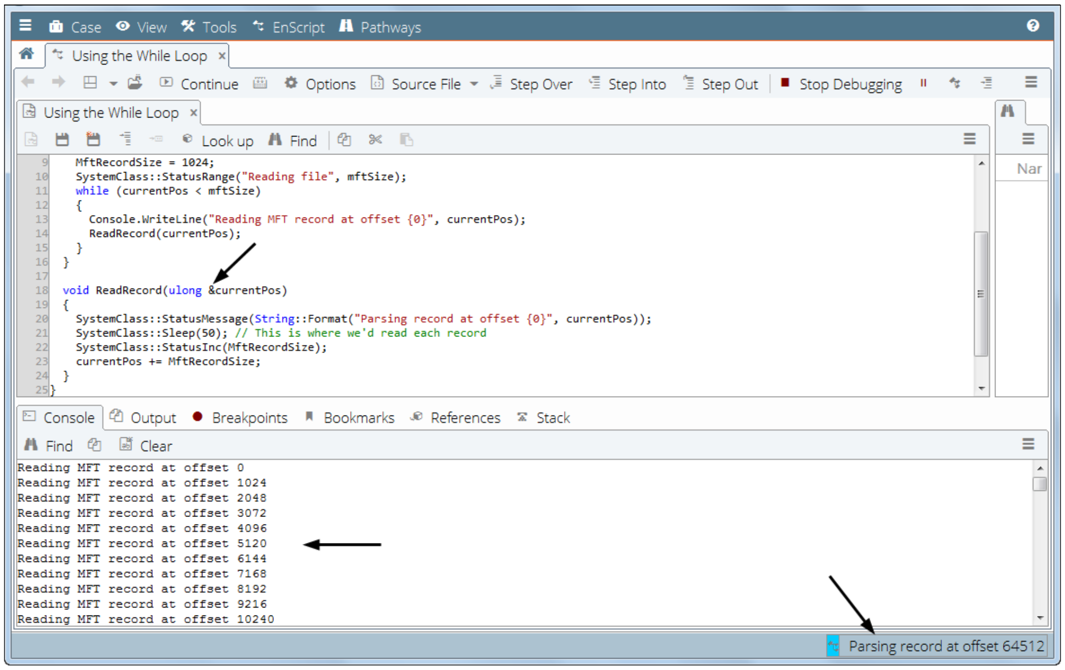
Figure 4-4 Viewing the updated EnScript, which never finishes processing

Our modified code demonstrates the fact that fundamental variables passed into functions are passed in *by value* rather than *by reference*.

In order to have the ReadRecord() function update the *actual* currentPos value, we need to rewrite the function definition so as to specify that that value will be passed in by reference.

To do this we simply prefix the variable name with an ampersand (&). That will allow the function to have access to the original variable so that it can be changed.

Having done that the script performs as before.



```
  MftRecordSize = 1024;
  SystemClass::StatusRange("Reading file", mftSize);
  while (currentPos < mftSize)
  {
    Console.WriteLine("Reading MFT record at offset {0}", currentPos);
    ReadRecord(currentPos);
  }
}

void ReadRecord(ulong &currentPos)
{
  SystemClass::StatusMessage(String::Format("Parsing record at offset {0}", currentPos));
  SystemClass::Sleep(50); // This is where we'd read each record
  SystemClass::StatusInc(MftRecordSize);
  currentPos += MftRecordSize;
}
}
```

```
Reading MFT record at offset 0
Reading MFT record at offset 1024
Reading MFT record at offset 2048
Reading MFT record at offset 3072
Reading MFT record at offset 4096
Reading MFT record at offset 5120
Reading MFT record at offset 6144
Reading MFT record at offset 7168
Reading MFT record at offset 8192
Reading MFT record at offset 9216
Reading MFT record at offset 10240
```

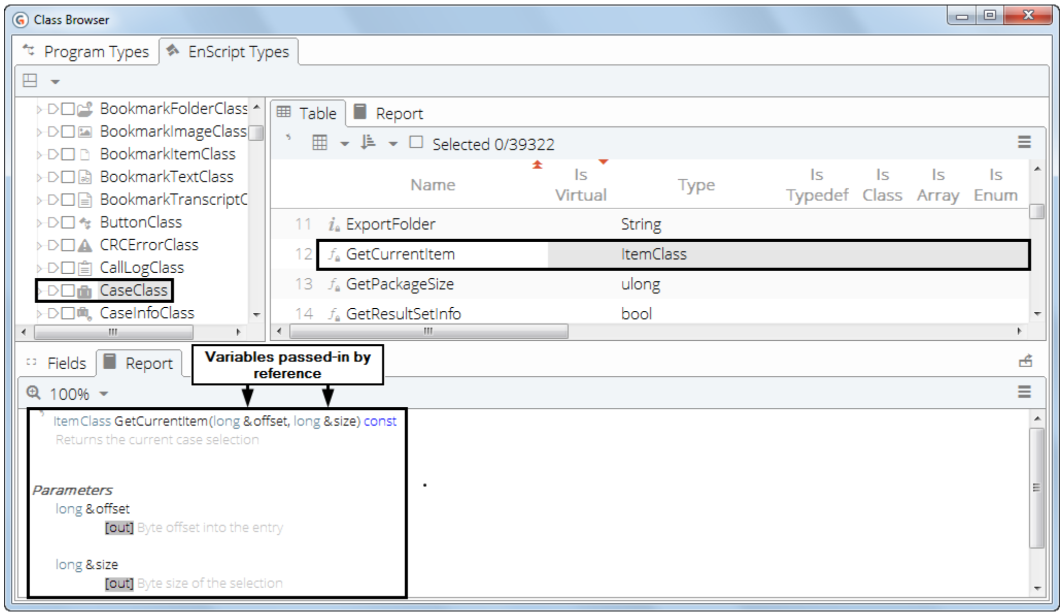
Parsing record at offset 64512

Figure 4-5 Running the EnScript program once more

Returning More Than One Value from a Function

Functions can only return one value and so passing variables into a function by reference provides an alternative way of having a function "return" more than one value.

A very good example of this is the function CaseClass::GetCurrentItem(). The following screenshot shows how that function is defined in the EnScript Class Browser.



	Name	Is Virtual	Type	Is Typedef	Is Class	Is Array	Is Enum
11	ExportFolder		String				
12	GetCurrentItem		ItemClass				
13	GetPackageSize		ulong				
14	GetResultSetInfo		bool				

ItemClass GetCurrentItem(long &offset, long &size) const
Returns the current case selection

Parameters
long &offset
[out] Byte offset into the entry

long &size
[out] Byte size of the selection

Figure 4-6 Viewing the CaseClass::GetCurrentItem() function definition

This function, when applied to a CaseClass object, returns the item highlighted by the user in the EnCase GUI. It is also able to "return" the offset and size of the highlighted data through the use of two variables passed in by reference.

In this case the use of variables for the purpose of output is not only indicated by the presence of the ampersand sign, it is also indicated by the presence of the [out] marker. When reviewing the documentation for other internal EnScript functions, you may well come across parameters marked with both [out] *and* [in].

We will use the CaseClass::GetCurrentItem() function later in the course in order to demonstrate how to access case data.

Passing Object Variables by Reference

An object variable is an instance of a class.

We haven't covered classes and object variables in very much depth, but having covered the concept of passing fundamental variables into a function by reference, it's not unusual for students to wonder whether object variables should be passed into functions by reference so as to access the actual variable rather than a copy of it.

The basic answer to this question is that this isn't usually necessary.

This is because the EnScript compiler makes object variables accessible through the use of behind-the-scenes pointers.

When an object variable is passed into a function as a parameter, the function actually receives a copy of that object's pointer. Because the copy of the pointer contains the same memory address as the original, the function still has access to the original object and can change it.

A good example of when you ***would*** need to pass an object variable into a function by reference is where that object will be allocated memory and constructed within the function.

Let's now turn our attention to the do/while loop.

USING THE DO/WHILE LOOP

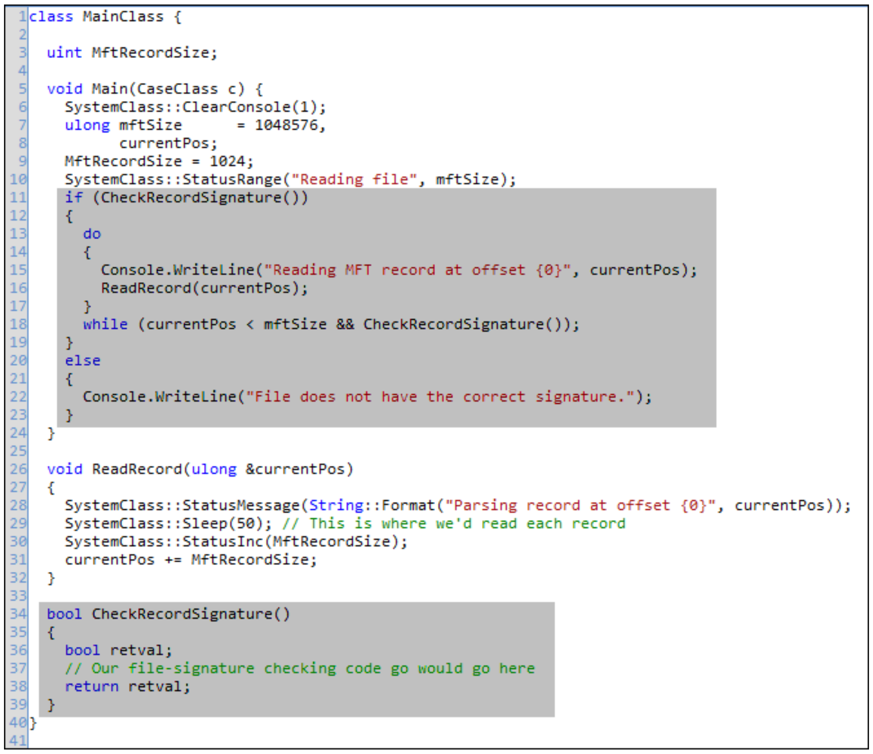
The do/while loop takes the form:

```
do
{
      // Code here
}
while (<condition>)
```

This type of loop is often used when reading records from a file. Take our previous MFT record EnScript as an example.

It would be reasonable to check that the file we are about to read is indeed a valid $MFT file. One way of doing that would be to look for the FILE[0*] GREP signature, belonging to the first record, the one that relates to the $MFT file itself.

So taking this account, we might update our script.



```
class MainClass {

  uint MftRecordSize;

  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);
    ulong mftSize      = 1048576,
          currentPos;
    MftRecordSize = 1024;
    SystemClass::StatusRange("Reading file", mftSize);
    if (CheckRecordSignature())
    {
      do
      {
        Console.WriteLine("Reading MFT record at offset {0}", currentPos);
        ReadRecord(currentPos);
      }
      while (currentPos < mftSize && CheckRecordSignature());
    }
    else
    {
      Console.WriteLine("File does not have the correct signature.");
    }
  }

  void ReadRecord(ulong &currentPos)
  {
    SystemClass::StatusMessage(String::Format("Parsing record at offset {0}", currentPos));
    SystemClass::Sleep(50); // This is where we'd read each record
    SystemClass::StatusInc(MftRecordSize);
    currentPos += MftRecordSize;
  }

  bool CheckRecordSignature()
  {
    bool retval;
    // Our file-signature checking code go would go here
    return retval;
  }
}
```

Figure 4-7 Updating the script to use a do-while loop in conjunction with an additional if-else statement

We've now introduced a new function called CheckRecordSignature(), which returns a Boolean value. This function would check that the signature of an MFT record about to be parsed is valid.

Note that, in reality, we would probably give this function the file being parsed as a parameter. That said, working with files isn't something that we've covered yet.

The updated script works as follows:

- The CheckRecordSignature() function is first called to check the signature of the record at the start of the $MFT file.
- The return value of the function is evaluated using an if statement. If the value returned is false then an appropriate error message is written to the console.
- If the CheckRecordSignature() returns true then the do/while block is entered and the first record is parsed.
- After the first record has been read, the loop's condition is evaluated for the first time. This causes the position of the file and the signature of the next record to be checked. If the condition returns true, the loop will be repeated for a second time and will continue to repeat until the end of the file is reached or an invalid MFT record signature is detected.

It's important to note that this version of the $MFT file-parsing script won't run because the CheckRecordSignature() return-value is always false.

Other Uses for the Do/While Loop

In addition to this type of file-parsing, the do-while loop is suited to a number of internal EnScript classes, including one that will iterate through records in a SQLite database and another one that will iterate through files and folders on the local file-system. The basic principle is to"

1. Check if we can get the first file/record matching our criteria. If not report an error.
2. If we can get the first file/record, process it within the body of a do-while loop.
3. Keep looping while the do-while loop's condition is satisfied. That condition will typically call another function, one that will return a true value if it can get another file/record matching the criteria specified.

We can now take a look at the for loop.

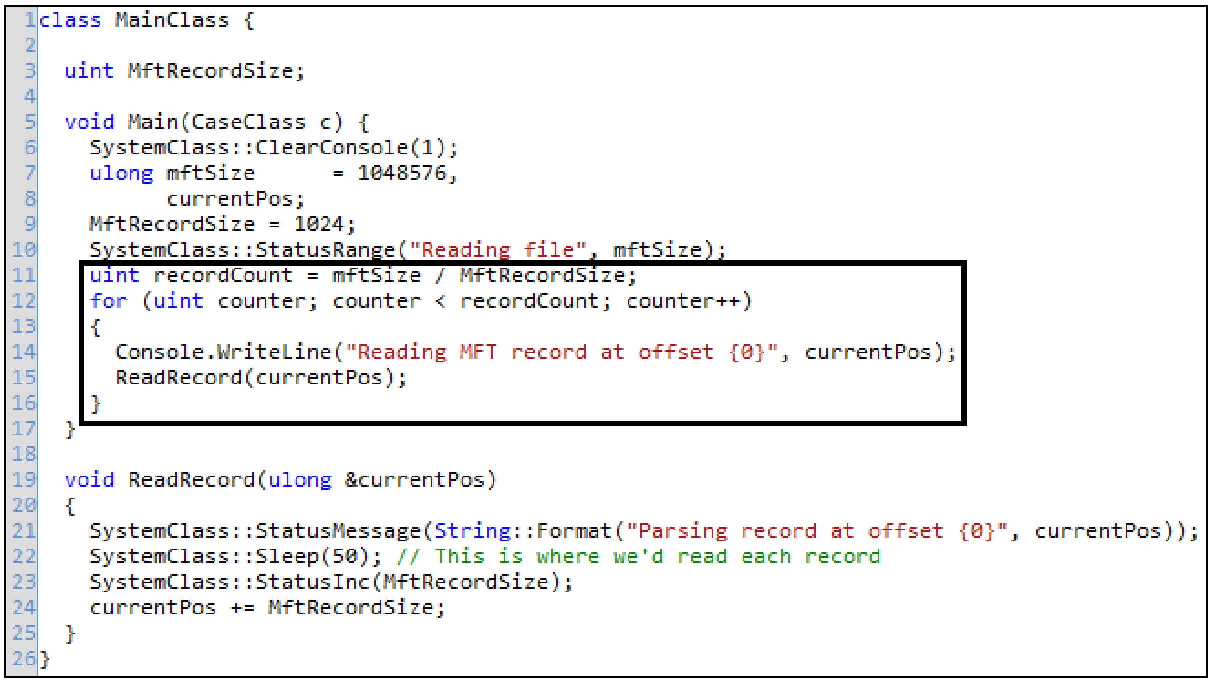
USING THE FOR LOOP

The for loop tends to be more common that either the while or do/while loops. It typically takes the following form:

```
for(<initialize variable>; <condition>; <increment variable>)
{


}
```

It's generally advisable to use for loops in place of while loops because the latter are a common cause of scripts failing to terminate. This isn't because while loops don't work properly, it's because the way in which they operate often results in the programmer forgetting to implement the code needed for a while loop's condition to be evaluated as negative.

So, taking this into account, we can modify our while example to use a for loop instead.



```
class MainClass {

  uint MftRecordSize;

  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);
    ulong mftSize      = 1048576,
          currentPos;
    MftRecordSize = 1024;
    SystemClass::StatusRange("Reading file", mftSize);
    uint recordCount = mftSize / MftRecordSize;
    for (uint counter; counter < recordCount; counter++)
    {
      Console.WriteLine("Reading MFT record at offset {0}", currentPos);
      ReadRecord(currentPos);
    }
  }

  void ReadRecord(ulong &currentPos)
  {
    SystemClass::StatusMessage(String::Format("Parsing record at offset {0}", currentPos));
    SystemClass::Sleep(50); // This is where we'd read each record
    SystemClass::StatusInc(MftRecordSize);
    currentPos += MftRecordSize;
  }
}
```

Figure 4-8 Replacing a while loop with a for loop

The updated script no longer uses the file position in order to control the loop that processes each record.

What it does instead is to calculate the number of MFT records that will be contained in the $MFT file given its size. It uses this value in conjunction with a for loop and a local uint variable called "counter."

Repeated iteration of the loop will only take while the value of the counter variable is less than the number of records in the file. The counter variable is incremented after each iteration of the loop.

Note that the number of records in the file is a 1-based count whereas the counter variable is 0-based. This is the reason for using the less-than operator (<) in the loop's condition.

The nice thing about for loops is that a variable declared within the loop declaration only has scope with the loop. This makes it easy to re-use variable names.

The last loop that we will look at during this lesson is the foreach loop.

USING THE FOREACH LOOP

The foreach loop is one of three loops that are used to iterate through the items contained in a *collection*.

A collection is the collective term for the items stored in an object that is a *container*. There are two types of container used in EnScript arrays and linked lists.

Arrays

An array is a collection of objects that are stored contiguously in memory.

Iterating through an array is quite fast because of the fact that the objects in an array are stored together, one after another.

That said, adding or removing objects from an array is slower because the array's contents must be maintained in the form of a contiguous list. If an item is to be added then this will usually entail making a copy of the array with sufficient additional space to add the new object afterwards; the original array will then be deleted.

Notwithstanding the benefit of speed, arrays don't tend to be used so much in EnScript because linked lists are a lot more flexible and can be used by the programmer in order to create their own custom lists.

Linked-lists

A linked-list is a collection of objects that do not have to be stored together in memory. The objects form a list because each one has two *sibling* links: one pointing to the memory location of the previous object in the list, and one pointing to the memory location of the next object.

In addition to having sibling links, an object will also have a *parent* link and a *first-child* link. This allows a linked list to form a hierarchical structure similar to the structure of files and folders stored within a volume.

Note that any or all of the aforementioned links can be empty (null). For instance, the parent and sibling links of a list's root object will be null; it will only have a first-child link (assuming that there are other objects in the list).

It's very easy to add and remove items to/from a linked-list. The only requirement is that there's sufficient memory available; there's no necessity to copy the existing items to a new memory location before new items can be added.

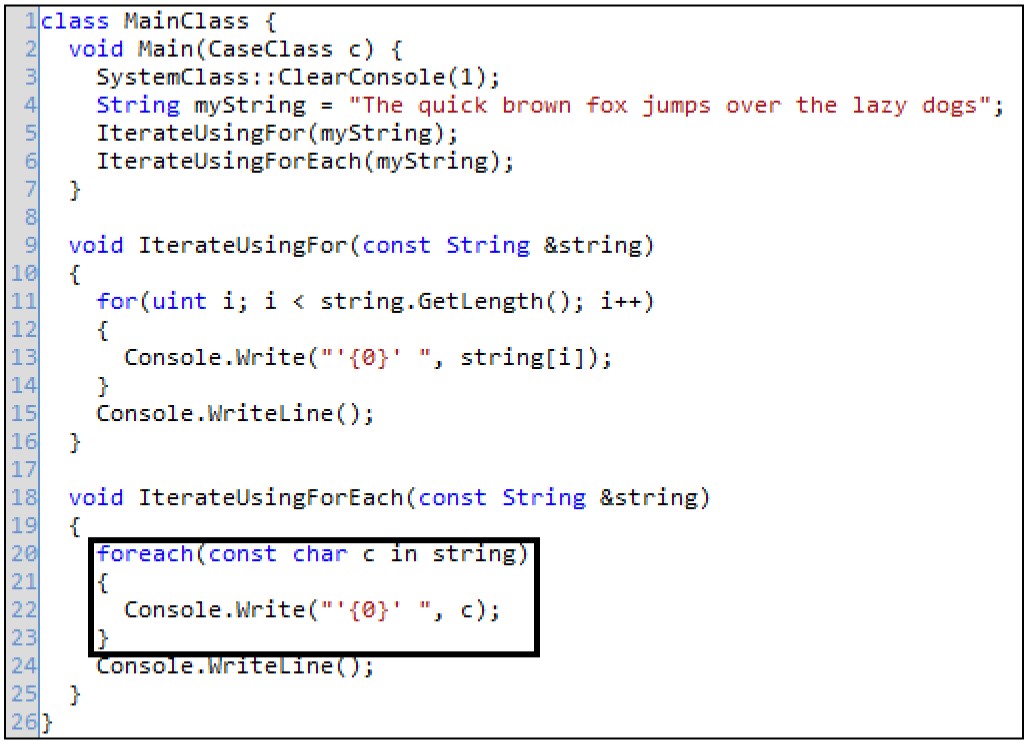
That said, iterating or finding items in a linked-list is slower because the links have to be followed in order to traverse from one object to another.

Linked-lists are everywhere in EnCase and EnScript. Examples of linked-lists are entries representing files and folders, records representing email and Internet history items, and bookmarks.

Demonstrating Use of a Foreach Loop

A foreach loop can be used to iterate through the items in an array or the immediate child items of an object in a linked list.

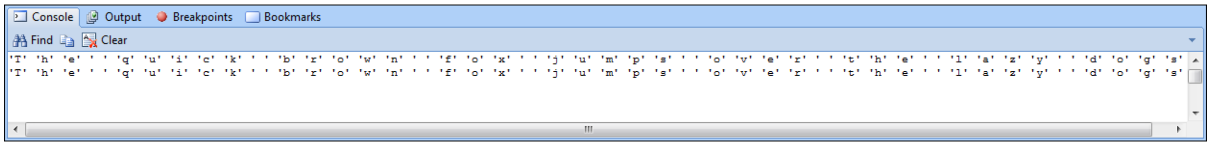
The easiest way to demonstrate the use of a foreach loop at this point in the course is to use a string, which is an array of characters.



```
class MainClass {
  void Main(CaseClass c) {
    SystemClass::ClearConsole(1);
    String myString = "The quick brown fox jumps over the lazy dogs";
    IterateUsingFor(myString);
    IterateUsingForEach(myString);
  }

  void IterateUsingFor(const String &string)
  {
    for(uint i; i < string.GetLength(); i++)
    {
      Console.Write("'{0}' ", string[i]);
    }
    Console.WriteLine();
  }

  void IterateUsingForEach(const String &string)
  {
    foreach(const char c in string)
    {
      Console.Write("'{0}' ", c);
    }
    Console.WriteLine();
  }
}
```

Figure 4-9 Using a foreach loop to iterate through a string as an array of characters

The preceding script iterates a string called "mystring" that uses two functions, one of which uses a for loop; the other uses a foreach loop. Both produce the same result.



```
'T' 'h' 'e' ' ' 'q' 'u' 'i' 'c' 'k' ' ' 'b' 'r' 'o' 'w' 'n' ' ' 'f' 'o' 'x' ' ' 'j' 'u' 'm' 'p' 's' ' ' 'o' 'v' 'e' 'r' ' ' 't' 'h' 'e' ' ' 'l' 'a' 'z' 'y' ' ' 'd' 'o' 'g' 's'
'T' 'h' 'e' ' ' 'q' 'u' 'i' 'c' 'k' ' ' 'b' 'r' 'o' 'w' 'n' ' ' 'f' 'o' 'x' ' ' 'j' 'u' 'm' 'p' 's' ' ' 'o' 'v' 'e' 'r' ' ' 't' 'h' 'e' ' ' 'l' 'a' 'z' 'y' ' ' 'd' 'o' 'g' 's'
```

Figure 4-10 Viewing the results

The following notes are provided with regard to this EnScript program's operation:

- Both of the iteration functions are defined as accepting a string passed by reference (lines 9 and 18). The const keyword is used so as to prevent inadvertent changing of the string within the function body.
- The IterateUsingFor() function obtains each letter from the string using a for loop and the array indexing method that is common to many programming languages. Note that the first character in a string will have an array index of 0. The String::GetLength() function is used to obtain the length of the string so we can declare the for loop accordingly.
- The IterateUsingForEach() function should be fairly self-explanatory. Note that each character retrieved from the String must be declared as constant because the string being iterated is itself a constant.

The other loops that can be used with collections are the forall loop and the forroot loop. These differ from the foreach loop because they are designed to iterate through all of the levels of a linked list, not just the first-child level.

Linked lists in EnScript are extremely versatile and will be covered in more depth later in the course.

NOTES

NOTES

NOTES

CLASSES

INTRODUCTION

The ability to use classes in the EnScript and other programming languages is what gives us the power to create programs in an object-orientated way.

The strange thing about the usage of classes is that it makes the programmer's life far easier and yet is often perceived as being complicated. Many new and novice programmers hear the word "class" and immediately switch off thinking that the understanding of classes is beyond them.

It's true that using classes requires some additional understanding, but once you've grasped their purpose, how they work, and have put them to a practical use, you'll wonder how you ever coped without them.

CAN I DO WITHOUT CLASSES?

You may go a long time without having to write a class but you will still have to use them on a constant basis.

If you think about it, we've already encountered MainClass, which is a class that represents an executable EnScript program.

Because every variable and function that you will use and/or create in an EnScript module exists as part of a class, you're never going to be in a situation where you won't use classes in some way.

When it comes to the need to ***write*** a class, you need to understand that an awful lot of the data that you will want to process in EnScript programming is generated ***using*** classes.

Some of the best examples of this type of data are file records. These might be FAT directory entries, MFT records, records that track files downloaded from the likes of Limewire, eMule, BitTorrent, and other P2P networks; in fact, pretty much any type of file record you can think of will have been generated using code that sees these records as objects.

Taking this into account, if you ever need to read a file record that was created as a class-object, then there's a good chance that you'll have to treat it as such in order to read it efficiently using your own EnScript code.

LESSON PURPOSE

You no doubt want to learn the EnScript language in order to enhance your ability to perform forensic computer examinations and are anxious to start working with some actual case data.

Whilst this is our ultimate goal, you will struggle to accomplish any useful task in EnScript programming without using at least one of its internal classes. You therefore need to understand how classes are structured so that you can understand the documentation that is available and use the classes for your own purpose.

We have, up until this point, tried to use real-life examples whenever possible.

That said, experience has shown that the best way of explaining classes is to use real-world objects that we're all familiar with. That way we can concentrate on what classes do and how they work rather than bogging ourselves down trying to explain them in the context of a forensic investigation, which is quite difficult to do without accessing actual case-data.

It's for this reason that the first EnScript class that we shall create is one to represent a common or garden-variety car.

So, with this example in mind, let's consider what a class actually ***is***.

WHAT IS A CLASS?

In basic terms, a class is a grouping of global variables (also referred to as "properties") and functions (also referred to as "methods").

The properties and methods of a class are referred to as its "members."

Classes tend to be used in one of two ways.

- As a template to construct objects or *instances* of that class. This approach allows the programmer to create built-in class functions that can be applied to an object as a whole rather than trying to deal with its properties as separate variables. This will enable the programmer to create and process many instances of the class without having to use complicated and lengthy looping procedures.
- As a container to house utility functions that are linked in some way so as to make them reusable and portable.

Using a Class as a Template

We shall be creating a class that will represent the template of a virtual car. That template will define properties, such as the car's owner, color, number of doors, whether or not the car has air conditioning, and what its fuel-economy figures are.

We can then use this template to build actual instances of a car. Each car will share the same properties but the values of those properties will be specific to the car in question.

In addition to defining properties, our class will also include the definition of one or more methods, which will define what we can do with a car. For instance, we might create a function that will give us a textual summary of a car's properties and calculate its average fuel economy.

In certain circumstances we might want to define a method that can be used without actually constructing a car. For instance, a function that calculates overall fuel economy might take three miles-per-gallon figures (MPG) and calculate the average of them.

This type of function might come in use somewhere else, maybe in a program that implements a "What-if?" scenario, one designed to answer questions like, "If I had a car with these MPG figures what would be their average?".

In a program like that we'd simply want to feed the three figures into a function and be handed back the result – we wouldn't want to have to construct an entire car just for that purpose. We will demonstrate how to create and include such a function as part of this lesson.

Using a Class as a Container to House Utility Functions

Sometimes we may want to build up a library of utility functions that can be used again and again.

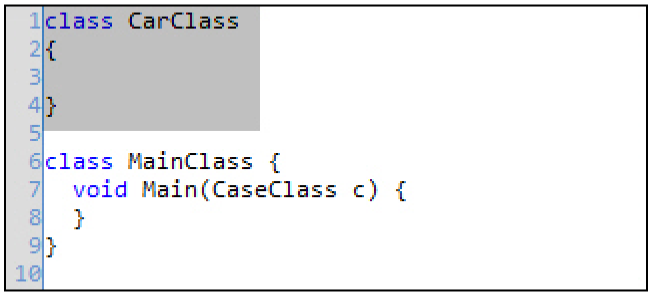
A good example of this might be a library of functions that perform mathematical calculations.

Because variables and functions cannot exist outside a class in EnScript, we would need to declare a class in which to store them.

That class could then be moved into a separate file and included in any EnScript program that needed it.

CREATING A CLASS TO REPRESENT A VIRTUAL CAR

The basic definition of a class consists of the `class` keyword, a name (typically ending with the word "class") and curly braces that mark the beginning and end of the class definition.



```
class CarClass
{

}

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

Figure 5-1 Creating a basic class definition

Note that while it doesn't matter in what order you declare the functions within a class, it does matter in what order you declare classes.

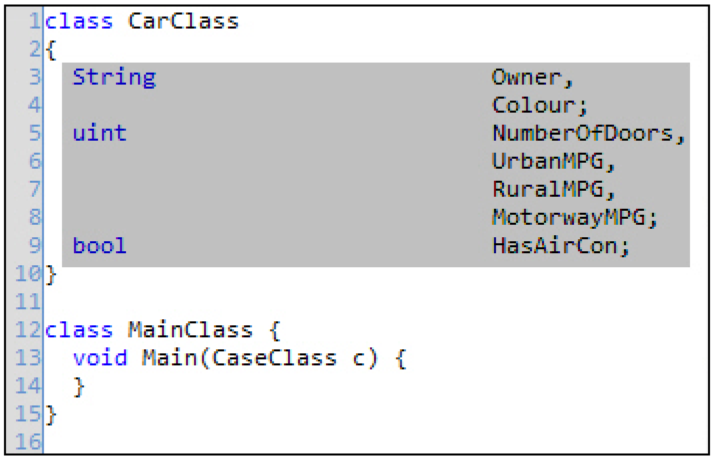
As a rule of thumb you must declare a class before it can be used by another class.

That said, one way around this is to give the compiler notice that the definition of a class exists later in the script by means of a *forward declaration*.

A forward declaration is made by adding a line of code consisting of the `class` keyword, the name of the class and a semicolon. When the compiler encounters this line of code it will jump forward in the script, find the full class definition, and process it.

Adding Member Properties

Our next step is to add some member properties to the car. These are declared in the same way that we declared MainClass global variables in a previous lesson.



```
class CarClass
{
  String                          Owner,
                                  Colour;
  uint                            NumberOfDoors,
                                  UrbanMPG,
                                  RuralMPG,
                                  MotorwayMPG;
  bool                            HasAirCon;
}

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

```

Figure 5-2 Adding member properties/global-variables

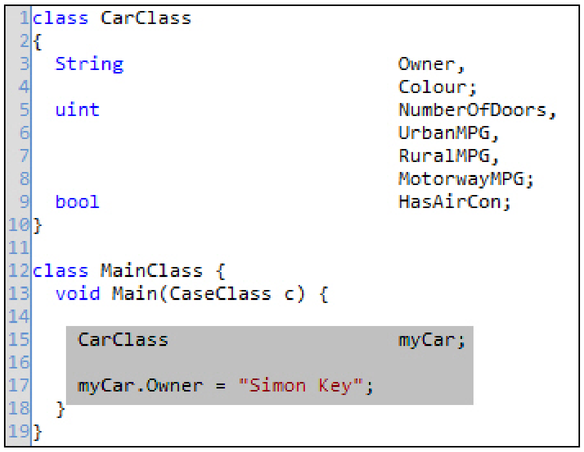
We can now declare our first CarClass object variable, but we have to do it in a slightly different way than the way in which we declare fundamental types.

Declaring, Instantiating, and Constructing Object Variables

When you declare a fundamental type, OpenText™ EnCase™ software (EnCase) allocates it memory (*instantiates* it) and assigns a default value.

Object variables are different because you have the power to decide how they should be constructed at the time they're instantiated in memory. For instance, if you have a class that represents a file record then you'll probably want EnCase to construct instances of that class by reading and interpreting data from the file in which the records you're trying to read are contained.

It is for this reason that declaring an object variable's type and name isn't enough to instantiate and construct the variable in memory. Let's demonstrate this.

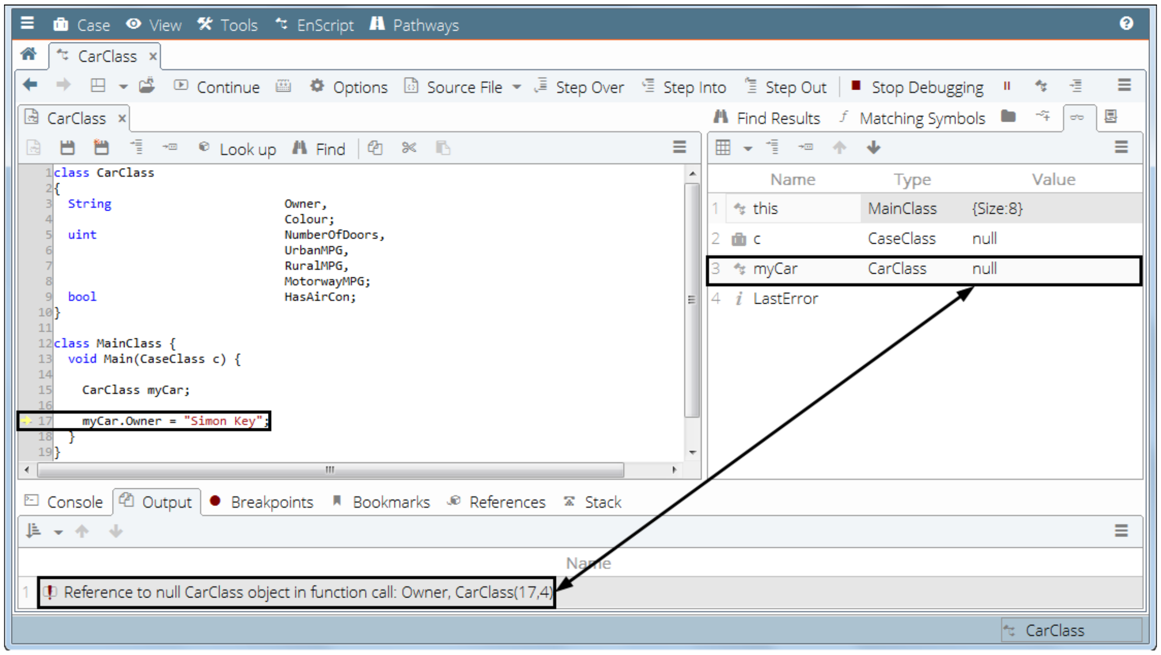


```
class CarClass
{
  String                          Owner,
                                  Colour;
  uint                            NumberOfDoors,
                                  UrbanMPG,
                                  RuralMPG,
                                  MotorwayMPG;
  bool                            HasAirCon;
}

class MainClass {
  void Main(CaseClass c) {

    CarClass                      myCar;

    myCar.Owner = "Simon Key";
  }
}
```

Figure 5-3 Attempting to set an object variables properties without instantiating the variable first

The previous script declares a CarClass object called "myCar" and tries to set its **Owner** property to **Simon Key**.

This script compiles without error but when we run it we get a runtime error and EnCase raises the debugger.



```
class CarClass
{
  String                          Owner,
                                  Colour;
  uint                            NumberOfDoors,
                                  UrbanMPG,
                                  RuralMPG,
                                  MotorwayMPG;
  bool                            HasAirCon;
}

class MainClass {
  void Main(CaseClass c) {

    CarClass myCar;

    myCar.Owner = "Simon Key";
  }
}
```

	Name	Type	Value
1	this	MainClass	{Size:8}
2	c	CaseClass	null
3	myCar	CarClass	null
4	LastError		

	Name
1	Reference to null CarClass object in function call: Owner, CarClass(17,4)

Figure 5-4 Runtime consequences of trying to access the properties of an unconstructed object variable

The EnCase output tab tells us that we're trying to refer to a null CarClass object, i.e. one that doesn't exist yet. The arrow in the gutter of the script editor window tells us where the problem occurs and the local window confirms that the value of myCar is null.

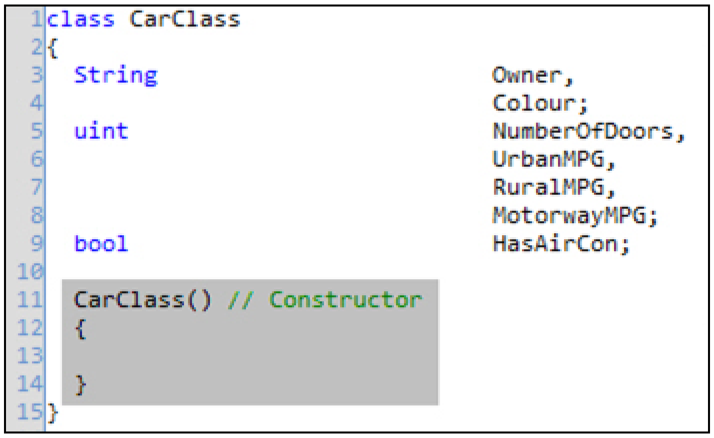
THE CONSTRUCTOR FUNCTION

Whenever we want to instantiate one of our own object variables we must allocate it to memory and call a special function called the "constructor."

The constructor has the same name as the class to which it relates; it also has the potential to accept one or more parameters.

If you haven't defined a constructor in your class then EnCase will declare a default constructor internally, one that doesn't take any parameters.

Let's stop the debugger using the **Stop Debugging** toolbar button and write a default constructor into our script so we can see what it looks like.



```
class CarClass
{
  String                          Owner,
                                  Colour;
  uint                            NumberOfDoors,
                                  UrbanMPG,
                                  RuralMPG,
                                  MotorwayMPG;
  bool                            HasAirCon;

  CarClass() // Constructor
  {

  }
}
```

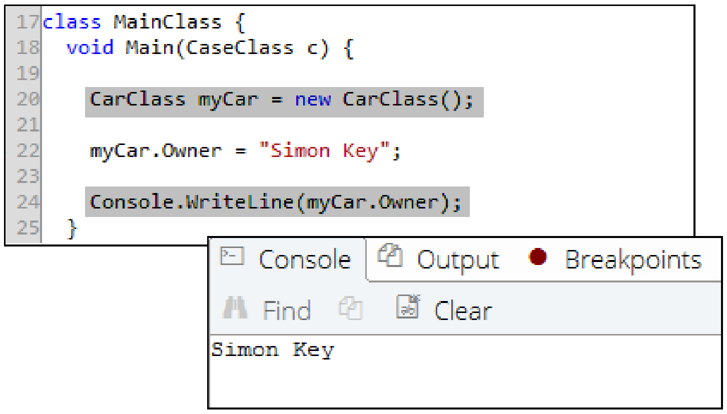
Figure 5-5 Writing a default constructor as part of a class definition

Now that this constructor is present it will take the place of the default constructor that EnCase would have created and used previously.

Our next step is to allocate memory for our CarClass object and call its constructor. We can do this in one of two ways

Using the new Statement

We can construct our CarClass object using the **new** statement as shown in the following screenshot.



```
17 class MainClass {
18   void Main(CaseClass c) {
19 
20     CarClass myCar = new CarClass();
21 
22     myCar.Owner = "Simon Key";
23 
24     Console.WriteLine(myCar.Owner);
25   }
```

```
Simon Key
```

Figure 5-6 Writing a default constructor as part of a class definition

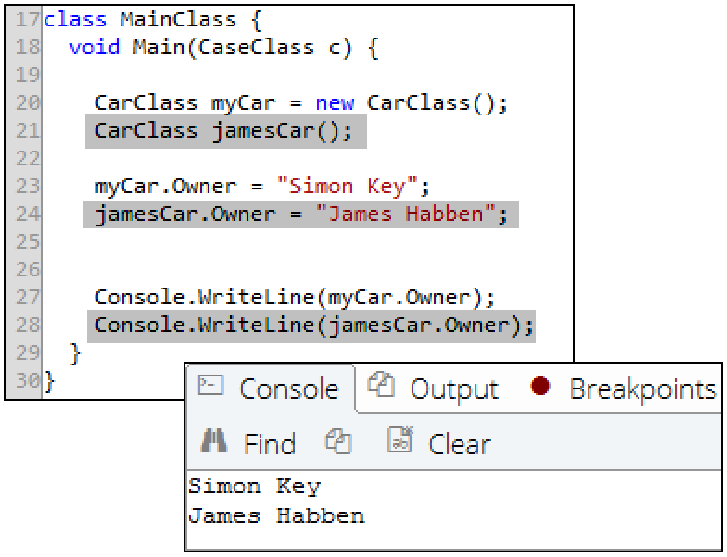
Line 20 is responsible for allocating memory for the CarClass object (using **new**) and calling its constructor.

At the same time as adding the constructor another statement was added to write the Owner property of the new myCar object variable to the console. As we can see, this worked just fine.

Let's have a look at a shortcut alternative to using the new keyword. This is called "implicit new."

Implicit new

In order to demonstrate implicit new let's create a new car called "jamesCar."



```
17 class MainClass {
18   void Main(CaseClass c) {
19 
20     CarClass myCar = new CarClass();
21     CarClass jamesCar();
22 
23     myCar.Owner = "Simon Key";
24     jamesCar.Owner = "James Habben";
25 
26 
27     Console.WriteLine(myCar.Owner);
28     Console.WriteLine(jamesCar.Owner);
29   }
30 }
```

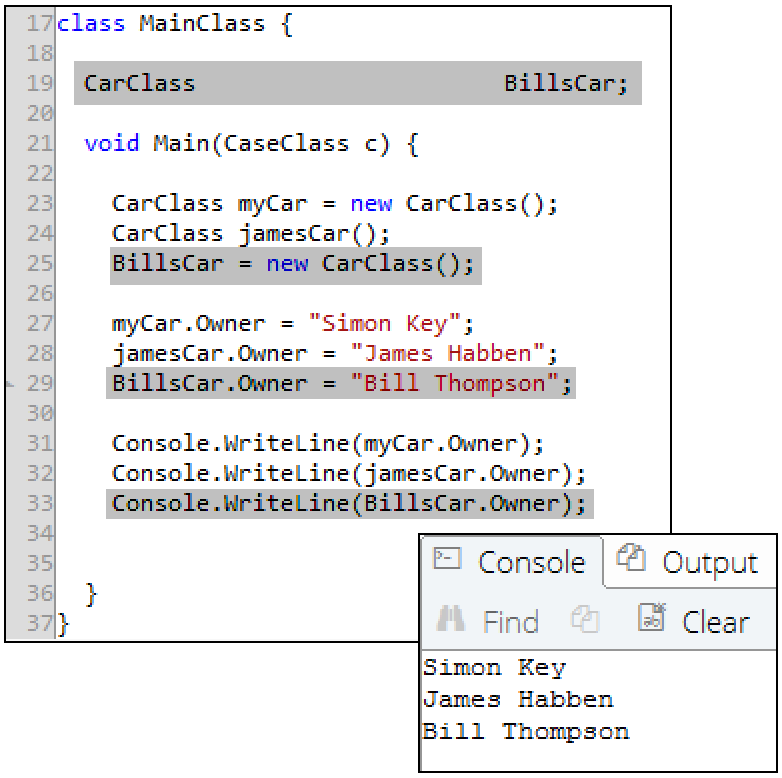
```
Simon Key
James Habben
```

Figure 5-7 Using implicit new

Line 21 uses implicit new as a shortcut to declaring, instantiating, and constructing the jamesCar CarClass variable.

While this is quick and convenient, it can't be used where an object variable has been declared as a global variable.

Let's demonstrate how we would declare, instantiate, and construct a global CarClass variable called "BillsCar."



```
17 class MainClass {
18
19   CarClass                        BillsCar;
20
21   void Main(CaseClass c) {
22
23     CarClass myCar = new CarClass();
24     CarClass jamesCar();
25     BillsCar = new CarClass();
26
27     myCar.Owner = "Simon Key";
28     jamesCar.Owner = "James Habben";
29     BillsCar.Owner = "Bill Thompson";
30
31     Console.WriteLine(myCar.Owner);
32     Console.WriteLine(jamesCar.Owner);
33     Console.WriteLine(BillsCar.Owner);
34
35
36   }
37 }
```

```
Simon Key
James Habben
Bill Thompson
```

Figure 5-8 Separating declaration from instantiation and construction

In this example, BillsCar is declared as a global MainClass variable on line 19.

It is then instantiated and constructed on line 25 before having its Owner property value set on line 29 and that value then being read-back on line 33.

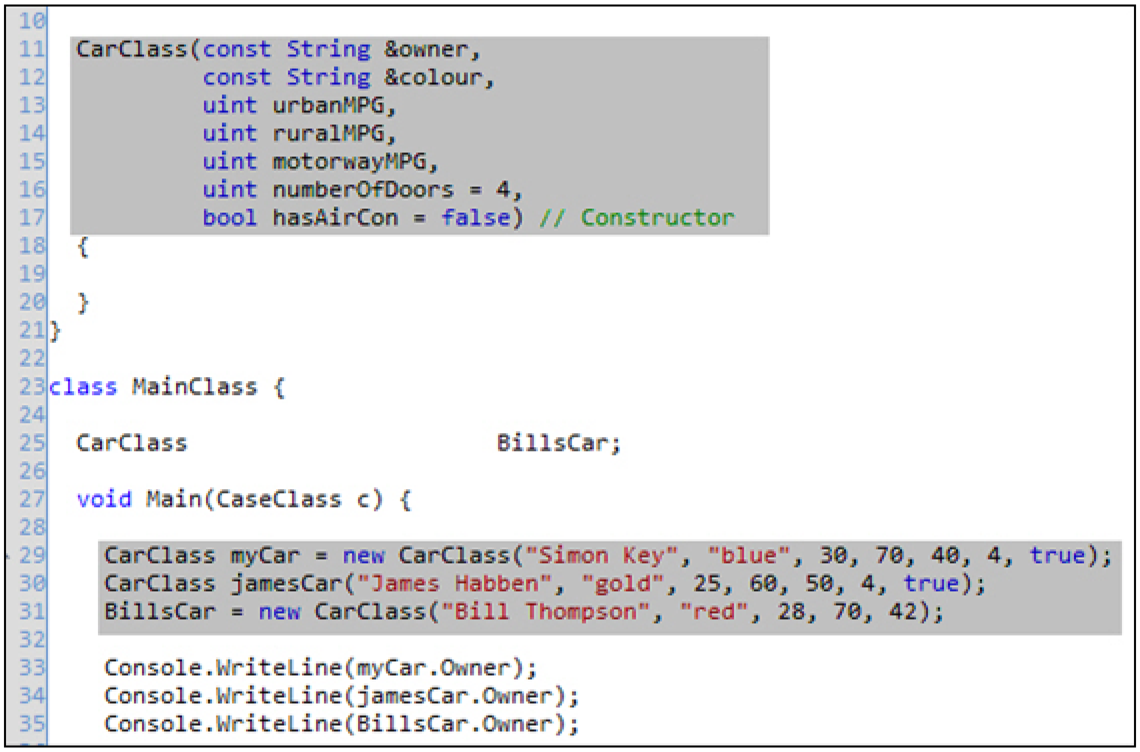
Using this method of instantiation and construction is necessary because, in general terms, global class variables cannot have their values set at the time they're declared.

While this method of declaring, instantiating and constructing object variables and then setting their properties afterwards works fine, it's usually easier to set those properties at the time the object-variable is constructed.

Customizing the Constructor

A constructor can be modified so as to accept parameters that can then be used to set the initial properties of a class object at the time it's instantiated.

Let's modify our script to demonstrate this.



```
10
11  CarClass(const String &owner,
12           const String &colour,
13           uint urbanMPG,
14           uint ruralMPG,
15           uint motorwayMPG,
16           uint numberOfDoors = 4,
17           bool hasAirCon = false) // Constructor
18  {
19
20  }
21}
22
23class MainClass {
24
25  CarClass                      BillsCar;
26
27  void Main(CaseClass c) {
28
29    CarClass myCar = new CarClass("Simon Key", "blue", 30, 70, 40, 4, true);
30    CarClass jamesCar("James Habben", "gold", 25, 60, 50, 4, true);
31    BillsCar = new CarClass("Bill Thompson", "red", 28, 70, 42);
32
33    Console.WriteLine(myCar.Owner);
34    Console.WriteLine(jamesCar.Owner);
35    Console.WriteLine(BillsCar.Owner);
```

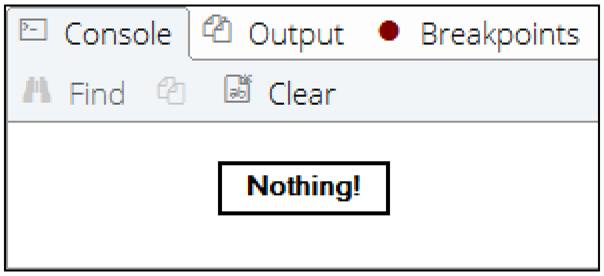
Figure 5-9 Providing an (incomplete) custom constructor

We've now modified our CarClass constructor so as to accept parameters that will be used to set the properties of each CarClass object. The names declared in the constructor are almost the same as the class properties that they'll set the values of but their first character is lower case.

Note how the equivalence sign (=) has been used to set default values for the numberOfDoors and hasAirCon constructor parameters (lines 16 and 17). These default values will be used if none are passed into the constructor (as at line 31). Note that parameters with default values must be placed last in a function's parameter list.

In addition to modifying the CarClass constructor we've removed the statements in MainClass::Main() that set the Owner property of myCar, jamesCar, and BillsCar and instead passed the desired Owner values into each car's constructor together with the other values that we want to set (lines 29, 30 and 31).

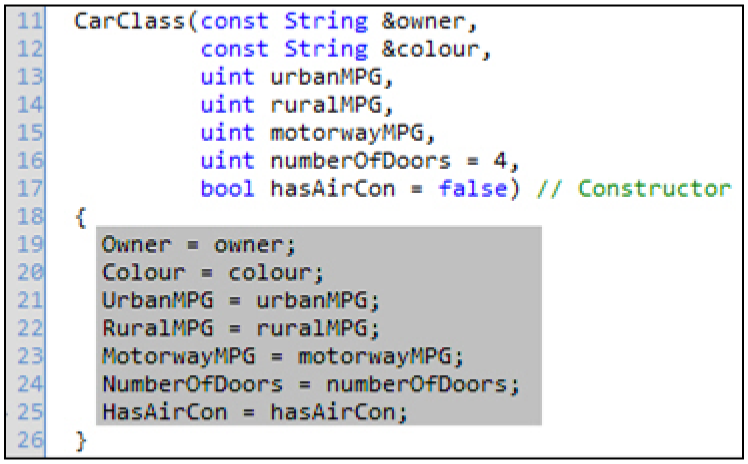
As things stand at the moment, the parameters passed into the constructor aren't being used for anything. When we run the script we get no output because of this.



Nothing!

Figure 5-10 Empty script output

To fix this issue we need use the parameters passed into the constructor to set the global variables (properties) of the class. This is why we use similar names for the constructor parameters – to remind us as to which property each relates to.

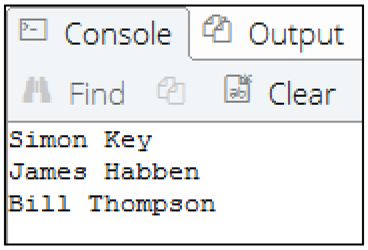


```
11  CarClass(const String &owner,
12           const String &colour,
13           uint urbanMPG,
14           uint ruralMPG,
15           uint motorwayMPG,
16           uint numberOfDoors = 4,
17           bool hasAirCon = false) // Constructor
18  {
19    Owner = owner;
20    Colour = colour;
21    UrbanMPG = urbanMPG;
22    RuralMPG = ruralMPG;
23    MotorwayMPG = motorwayMPG;
24    NumberOfDoors = numberOfDoors;
25    HasAirCon = hasAirCon;
26  }
```

Figure 5-11 Using the constructor parameters to set the properties of the class

Lines 19 through 25 assign the values of the incoming constructor parameters to the global variables (properties) of the class.

This brings us back on track and causes the script to output the Owner properties as before.



```
Simon Key
James Habben
Bill Thompson
```

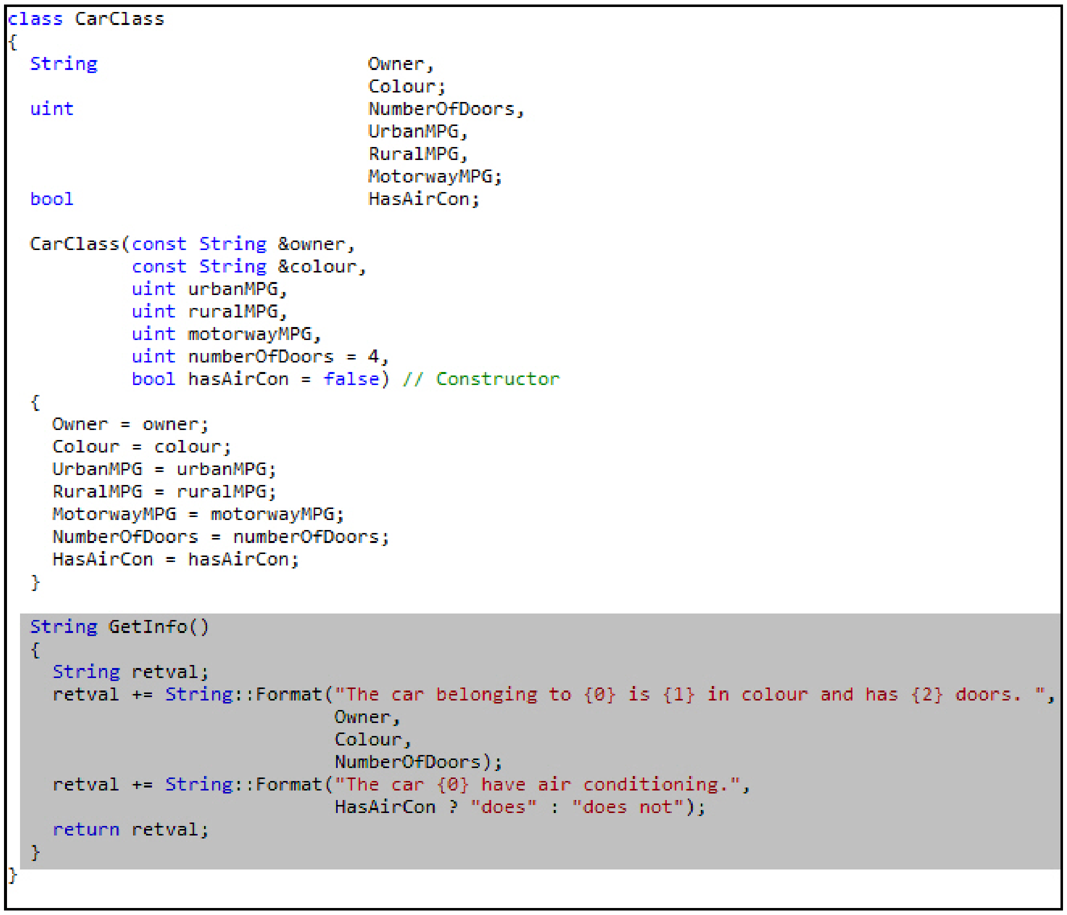
Figure 5-12 Viewing the correct output

Our next step is to add another CarClass method that will provide us more information about a CarClass object than just its owner.

ADDING ADDITIONAL MEMBER FUNCTIONS

The beauty of using classes is that each object variable contains its own properties, which are global variables and can be accessed by any method defined in the object's class.

Let's use this ability to create a CarClass method that we can use to give us a summary about each car and its properties.



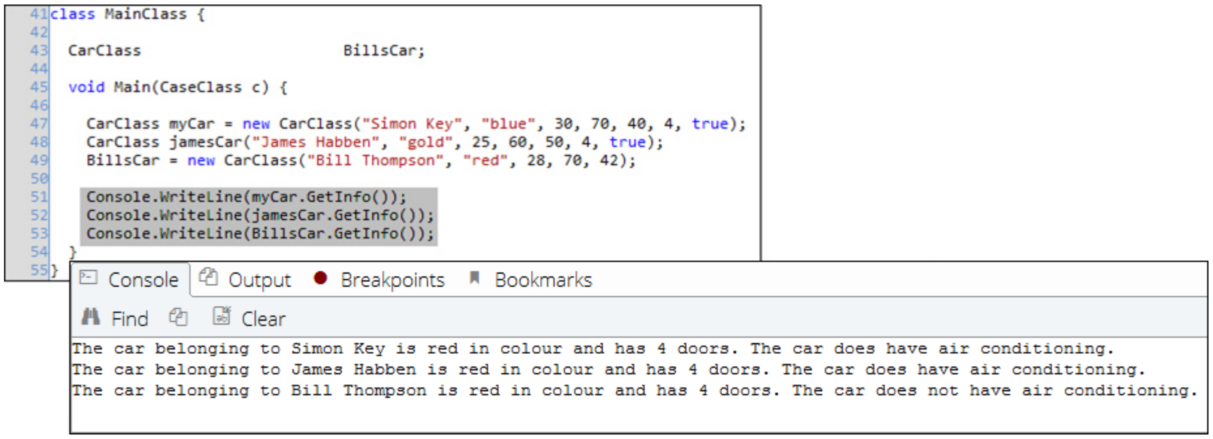
```
class CarClass
{
  String                        Owner,
                                Colour;
  uint                          NumberOfDoors,
                                UrbanMPG,
                                RuralMPG,
                                MotorwayMPG;
  bool                          HasAirCon;

  CarClass(const String &owner,
           const String &colour,
           uint urbanMPG,
           uint ruralMPG,
           uint motorwayMPG,
           uint numberOfDoors = 4,
           bool hasAirCon = false) // Constructor
  {
    Owner = owner;
    Colour = colour;
    UrbanMPG = urbanMPG;
    RuralMPG = ruralMPG;
    MotorwayMPG = motorwayMPG;
    NumberOfDoors = numberOfDoors;
    HasAirCon = hasAirCon;
  }

  String GetInfo()
  {
    String retval;
    retval += String::Format("The car belonging to {0} is {1} in colour and has {2} doors. ",
                             Owner,
                             Colour,
                             NumberOfDoors);
    retval += String::Format("The car {0} have air conditioning.",
                             HasAirCon ? "does" : "does not");
    return retval;
  }
}
```

Figure 5-13 The CarClass::GetInfo()

The new CarClass::GetInfo() method that we see in the previous screenshot returns a formatted string containing information about the properties of each CarClass object to which it is applied. The CarClass::GetInfo() method has access to these properties as a matter of course because they're global variables within CarClass.

Note that the method that we're using to create the formatted string still uses string concatenation even though it uses the String::Format() method a couple of times. This is not ideal but the better option is to use a file in memory, which we haven't covered yet.

Now, rather than writing the owner of each car to the console we can adjust the MainClass::Main() function to apply the CarClass::GetInfo() method and write its output to the console instead.



```
class MainClass {

  CarClass                       BillsCar;

  void Main(CaseClass c) {

    CarClass myCar = new CarClass("Simon Key", "blue", 30, 70, 40, 4, true);
    CarClass jamesCar("James Habben", "gold", 25, 60, 50, 4, true);
    BillsCar = new CarClass("Bill Thompson", "red", 28, 70, 42);

    Console.WriteLine(myCar.GetInfo());
    Console.WriteLine(jamesCar.GetInfo());
    Console.WriteLine(BillsCar.GetInfo());
  }
}
```

```
The car belonging to Simon Key is red in colour and has 4 doors. The car does have air conditioning.
The car belonging to James Habben is red in colour and has 4 doors. The car does have air conditioning.
The car belonging to Bill Thompson is red in colour and has 4 doors. The car does not have air conditioning.
```

Figure 5-14 Viewing the modified output

This hopefully goes some way to demonstrate the power and convenience of using classes. Our code allows us to create and process multiple CarClass objects very easily. The code is uncluttered and we can add, modify, and delete functionality quite easily.

Our next step is to update CarClass so as to calculate the average fuel consumption for each CarClass object.

Notwithstanding that we could create a member function that accesses the MPG figures as global variables, we want the function to be usable without having to create a CarClass object.

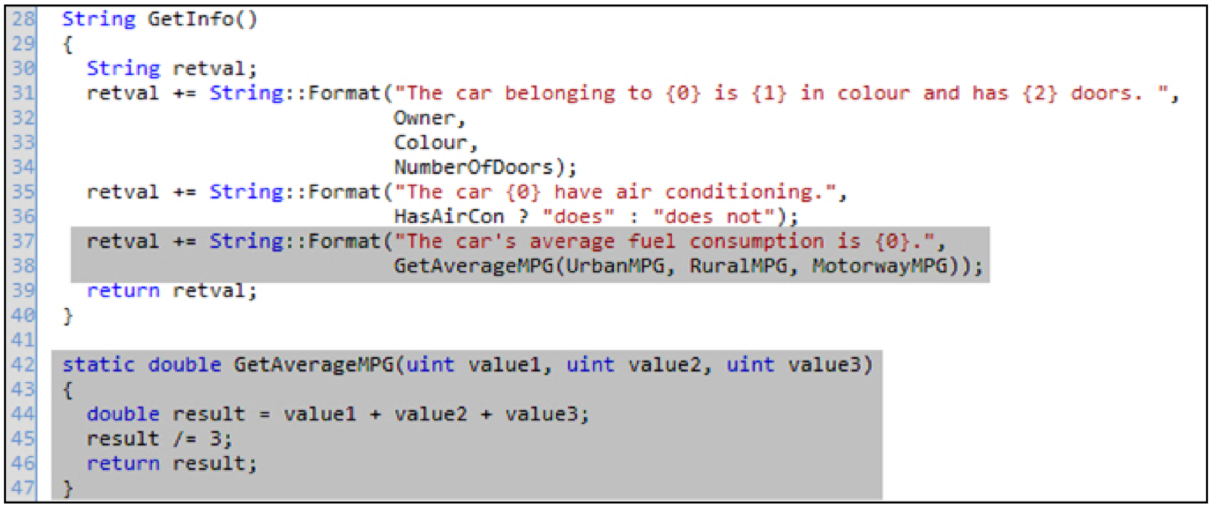
In order to do this we need to declare a *static* function.

STATIC FUNCTIONS

A function that is declared as static within a class can be used without being applied to an instance of that class. In order for this to work, the function cannot, for obvious reasons, use any use of the global variables (properties) of the class in which it is defined.

Taking this into account, the information needed by a static function to work must be passed to it in the form of one or more function parameters.

The following screenshot shows a new static function within CarClass that will accomplish what we need. The CarClass::GetInfo() function has also been updated so as to make use of it.



```
String GetInfo()
{
  String retval;
  retval += String::Format("The car belonging to {0} is {1} in colour and has {2} doors. ",
                           Owner,
                           Colour,
                           NumberOfDoors);
  retval += String::Format("The car {0} have air conditioning.",
                           HasAirCon ? "does" : "does not");
  retval += String::Format("The car's average fuel consumption is {0}.",
                           GetAverageMPG(UrbanMPG, RuralMPG, MotorwayMPG));
  return retval;
}

static double GetAverageMPG(uint value1, uint value2, uint value3)
{
  double result = value1 + value2 + value3;
  result /= 3;
  return result;
}
```

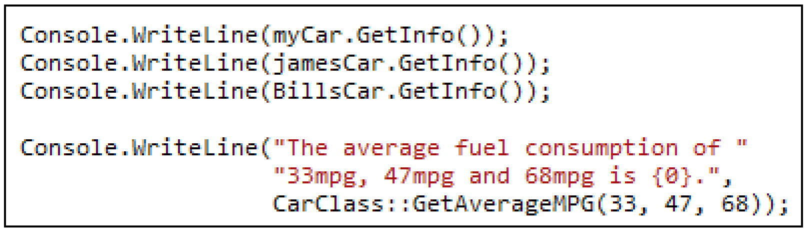
Figure 5-15 A static function

The CarClass::GetAverageMPG() static function declares itself as returning a double value, which is calculated by dividing the total of the three uint parameter values by 3.

As previously mentioned, the function is not allowed to access any CarClass properties and so the CarClass::GetInfo() function, which does have access to those properties, has to pass them into the CarClass::GetAverageMPG() function as parameters.

Note that the CarClass::GetInfo() function doesn't have to refer to the GetAverageMPG() static function as CarClass::GetAverageMPG(). This is because they're both members of the same class.

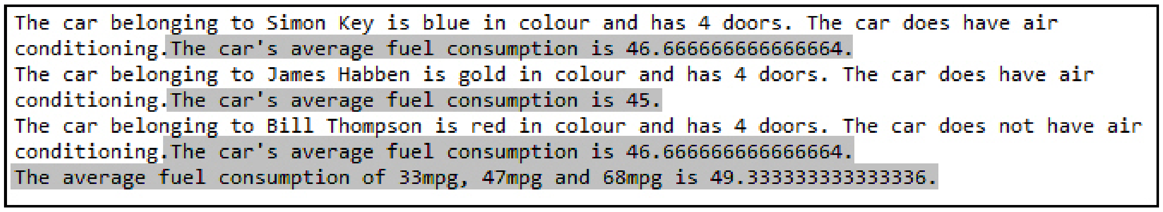
The way in which the CarClass::GetAverageMPG() function has been defined as static means that it's not only usable by the CarClass::GetInfo() function, it can be called directly from other functions using the scope operator as shown in the following screenshot.



```
Console.WriteLine(myCar.GetInfo());
Console.WriteLine(jamesCar.GetInfo());
Console.WriteLine(BillsCar.GetInfo());

Console.WriteLine("The average fuel consumption of "
                  "33mpg, 47mpg and 68mpg is {0}.",
                  CarClass::GetAverageMPG(33, 47, 68));
```

Figure 5-16 Calling the CarClass::GetAverageMPG() static function directly

The output of the script is as expected.



```
The car belonging to Simon Key is blue in colour and has 4 doors. The car does have air
conditioning.The car's average fuel consumption is 46.666666666666664.
The car belonging to James Habben is gold in colour and has 4 doors. The car does have air
conditioning.The car's average fuel consumption is 45.
The car belonging to Bill Thompson is red in colour and has 4 doors. The car does not have air
conditioning.The car's average fuel consumption is 46.666666666666664.
The average fuel consumption of 33mpg, 47mpg and 68mpg is 49.333333333333336.
```

Figure 5-17 Viewing the computed average MPG values

The one thing that could do with some extra work is the decimal precision with which the average MPG figures are displayed. We'll come back to that later once we understand the concept of *enumerated types* and how they can be incorporated into classes.

ENUMERATED TYPES

An enumerated type is a variable that take its value from one of a number of predefined integer values or *elements*. Each element is a constant (cannot change) and has a label.

Regardless of how many characters a label has, the value of an enumerated type will only ever occupy 4-bytes of memory.

Taking this into account, using an enumerated type to store the color of a car will take only 4-bytes and allow us to store up to 65,536 predefined colors.

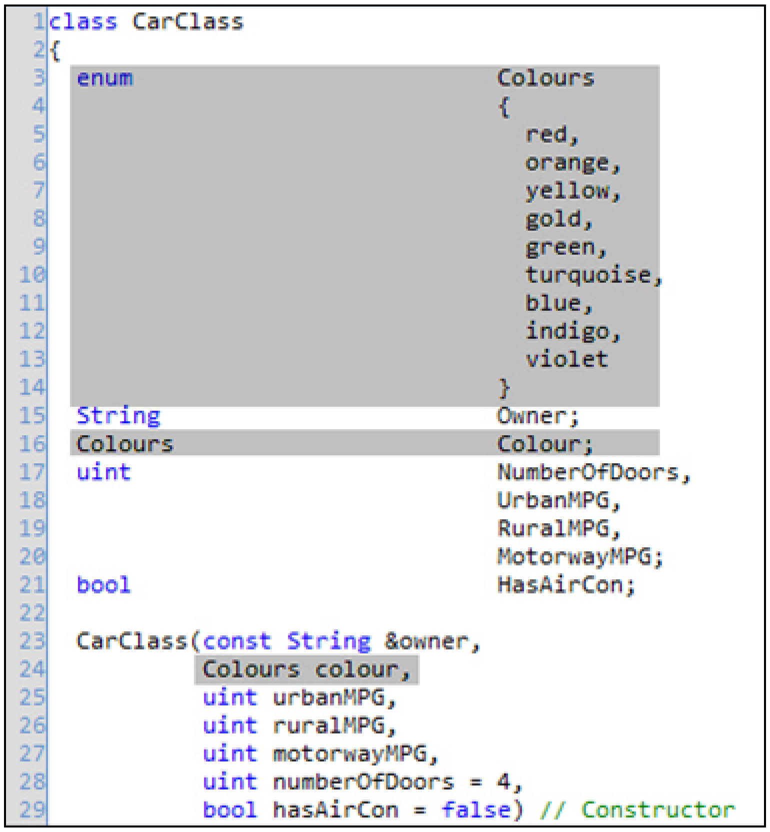
This is quite an efficiency saving, taking into account that a color, such as turquoise, would occupy 20 bytes if stored as a Unicode string.

Let's amend our script to make use of this functionality.

First we need to take care of the following:

- Declare the enumerated type, which we shall call "Colours"
- Identify that the CarClass::Colour property will be a Colours enumerated type
- Modify the CarClass constructor so that the color parameter is defined as a Colours enumerated type rather than a string

Once we've done that, the definition of our CarClass appears as shown in the following screenshot.



```
class CarClass
{
  enum                                                       Colours
                                                             {
                                                               red,
                                                               orange,
                                                               yellow,
                                                               gold,
                                                               green,
                                                               turquoise,
                                                               blue,
                                                               indigo,
                                                               violet
                                                             }
  String                                                     Owner;
  Colours                                                    Colour;
  uint                                                       NumberOfDoors,
                                                             UrbanMPG,
                                                             RuralMPG,
                                                             MotorwayMPG;
  bool                                                       HasAirCon;

  CarClass(const String &owner,
           Colours colour,
           uint urbanMPG,
           uint ruralMPG,
           uint motorwayMPG,
           uint numberOfDoors = 4,
           bool hasAirCon = false) // Constructor
```

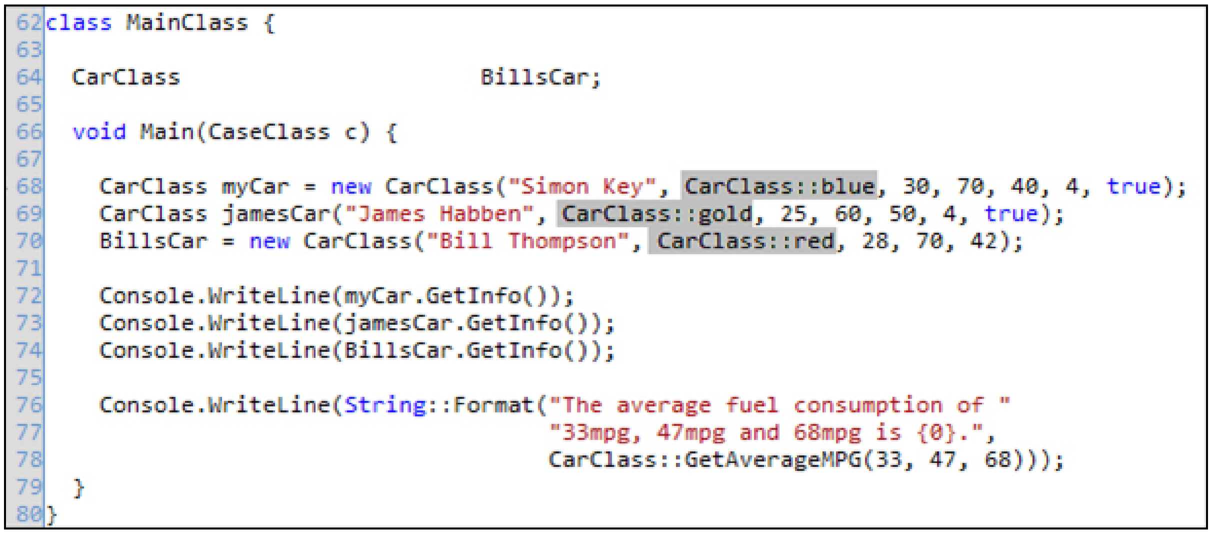
Figure 5-18 Modifying CarClass to support an enumerated Colour property

We should point out that the elements in the Colour enumerated type are defined as lower case to make our life easier but it's more usual to find them in uppercase.

Note that each element will be given an underlying integer value starting with 0 and incremented by one in each case. We could allocate our own integer values but there's usually no need to do so unless two or more of the values will be combined at a bit level, which doesn't apply here. We shall, however, see an example of that shortly.

Were we to try and compile the script now, we'd receive an error because the color parameters we used to construct the CarClass object variables in the MainClass::Main function were strings. We must change each of those parameters to be a CarClass::Colours enumerated type.

The following screenshot shows the result of these modifications.

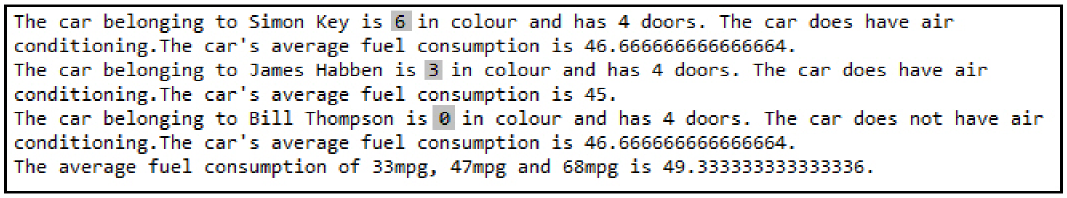


```
class MainClass {

  CarClass                      BillsCar;

  void Main(CaseClass c) {

    CarClass myCar = new CarClass("Simon Key", CarClass::blue, 30, 70, 40, 4, true);
    CarClass jamesCar("James Habben", CarClass::gold, 25, 60, 50, 4, true);
    BillsCar = new CarClass("Bill Thompson", CarClass::red, 28, 70, 42);

    Console.WriteLine(myCar.GetInfo());
    Console.WriteLine(jamesCar.GetInfo());
    Console.WriteLine(BillsCar.GetInfo());

    Console.WriteLine(String::Format("The average fuel consumption of "
                                     "33mpg, 47mpg and 68mpg is {0}.",
                                     CarClass::GetAverageMPG(33, 47, 68)));
  }
}
```

Figure 5-19 Modifying CarClass to support an enumerated Colour property

Note that it's not necessary in this case to qualify the scope of the color passed into each car's constructor. EnCase will automatically look for the definition of an enumerated value in the same scope as the function it is being passed into, in this case CarClass.

The script now compiles just fine but the console output looks a little strange.



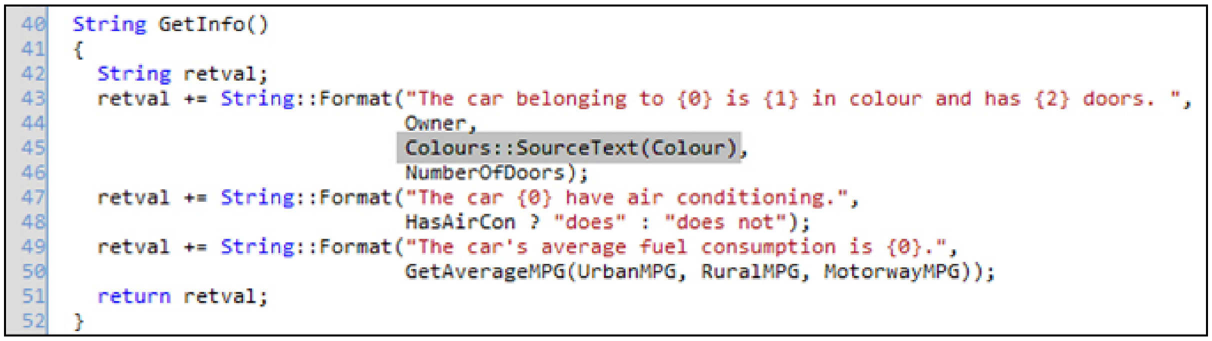
```
The car belonging to Simon Key is 6 in colour and has 4 doors. The car does have air
conditioning.The car's average fuel consumption is 46.666666666666664.
The car belonging to James Habben is 3 in colour and has 4 doors. The car does have air
conditioning.The car's average fuel consumption is 45.
The car belonging to Bill Thompson is 0 in colour and has 4 doors. The car does not have air
conditioning.The car's average fuel consumption is 46.666666666666664.
The average fuel consumption of 33mpg, 47mpg and 68mpg is 49.333333333333336.
```

Figure 5-20 Viewing the modified script's output

The problem we're seeing here is that although enumerations are fundamental types and can be converted to a string automatically, the conversion produces the enumerated type's integer value, not the associated label.

Thankfully, every enumerated type has an implied static function that will allow us take an enumerated type and display the label associated with its underlying value.

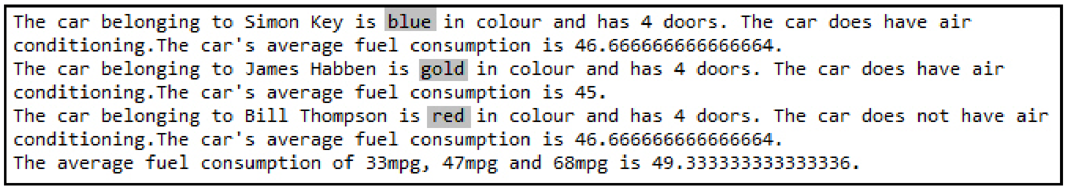
Taking this into account we can modify the script to appear as shown in the following screenshot.



```
String GetInfo()
{
  String retval;
  retval += String::Format("The car belonging to {0} is {1} in colour and has {2} doors. ",
                           Owner,
                           Colours::SourceText(Colour),
                           NumberOfDoors);
  retval += String::Format("The car {0} have air conditioning.",
                           HasAirCon ? "does" : "does not");
  retval += String::Format("The car's average fuel consumption is {0}.",
                           GetAverageMPG(UrbanMPG, RuralMPG, MotorwayMPG));
  return retval;
}
```

Figure 5-21 Modifying the script to shown the labels associated with an enumeration

When we run the script the output now displays each color correctly.



```
The car belonging to Simon Key is blue in colour and has 4 doors. The car does have air
conditioning.The car's average fuel consumption is 46.666666666666664.
The car belonging to James Habben is gold in colour and has 4 doors. The car does have air
conditioning.The car's average fuel consumption is 45.
The car belonging to Bill Thompson is red in colour and has 4 doors. The car does not have air
conditioning.The car's average fuel consumption is 46.666666666666664.
The average fuel consumption of 33mpg, 47mpg and 68mpg is 49.333333333333336.
```

Figure 5-22 Viewing the modified script's output

We've covered a substantial amount of information relating to classes although there's still quite a lot more to learn.

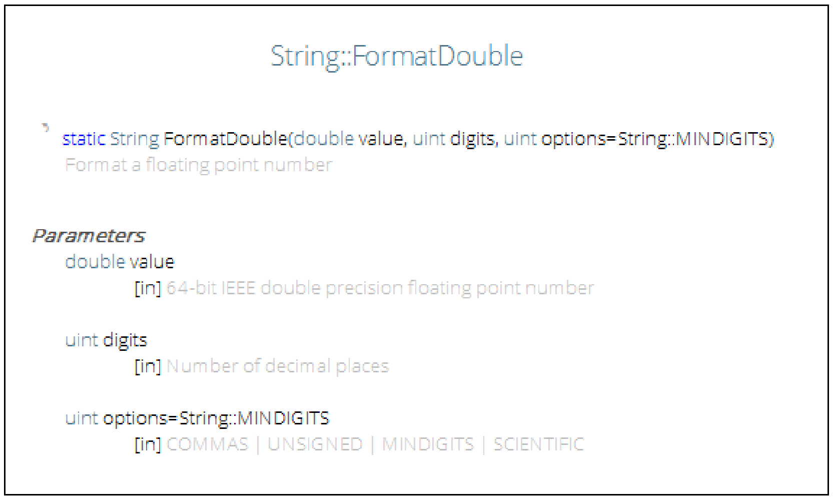
That said, we've learned enough to implement our own classes and better understand the EnScript documentation that is available.

Before we finish this lesson, let's return to the issue of precision associated with variables of the type "double." We will also take a look at EnScript library files or "includes".

Formatting Double Values

The static function that we created to calculate the average fuel consumption for a car works well but a precision of one decimal place would be better for our needs.

In order to accomplish this we can use the String::FormatDouble() function, which is defined as follows within the EnScript Class Browser.



String::FormatDouble

```
static String FormatDouble(double value, uint digits, uint options=String::MINDIGITS)
Format a floating point number
```

Parameters

double value
[in] 64-bit IEEE double precision floating point number

uint digits
[in] Number of decimal places

uint options=String::MINDIGITS
[in] COMMAS | UNSIGNED | MINDIGITS | SCIENTIFIC

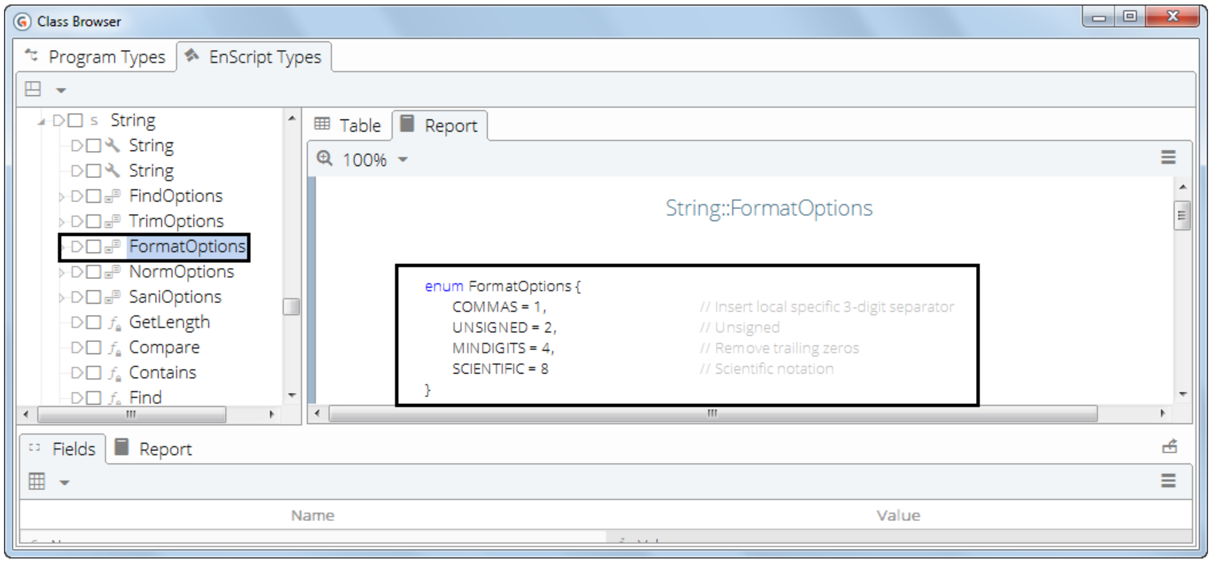
Figure 5-23 Viewing the definition of the String::FormatDouble() function

This function is defined as a static function so we know that we don't apply it to a string; we call it using the scope operator.

The function returns a string and takes three parameters.

- The first parameter specifies the double variable that we want to format
- The second parameter specifies the number of decimal places we want in the output string
- The third parameter is an unsigned integer, which needs to be set by combining one or more enumerated types at the bit level

The enumeration specified by the third parameter is defined by String::FormatOptions as shown in the following screenshot.



String::FormatOptions

```
enum FormatOptions {
    COMMAS = 1,           // Insert local specific 3-digit separator
    UNSIGNED = 2,         // Unsigned
    MINDIGITS = 4,        // Remove trailing zeros
    SCIENTIFIC = 8        // Scientific notation
}
```

Figure 5-24 Viewing the definition of the String::FormatOptions enumeration

The integer values of the elements defined in the String::FormatOptions enumeration (1, 2, 4, and 8) each represent a single bit, which allows two or more options to be combined using the bitwise OR operator (|). We don't need to specify any of the possible enumerated types because the only one we would want (String::MINDIGITS) is set as the default.

Now that we know how to use this function we can include it in our script.

```
static String GetAverageMPG(uint value1, uint value2, uint value3)
{
  double result = value1 + value2 + value3;
  result /= 3;
  return String::FormatDouble(result, 1);
}
```

Figure 5-25 Viewing the definition of the String::FormatOptions enumeration

Note how CarClass::GetAverageMPG() function has been modified to return a string. This makes sense seeing as the String::FormatDouble() function returns a string and we don't envisage doing anything with the calculated average other than writing it *as* a string.

When we run the script we find that the output has been adjusted accordingly.

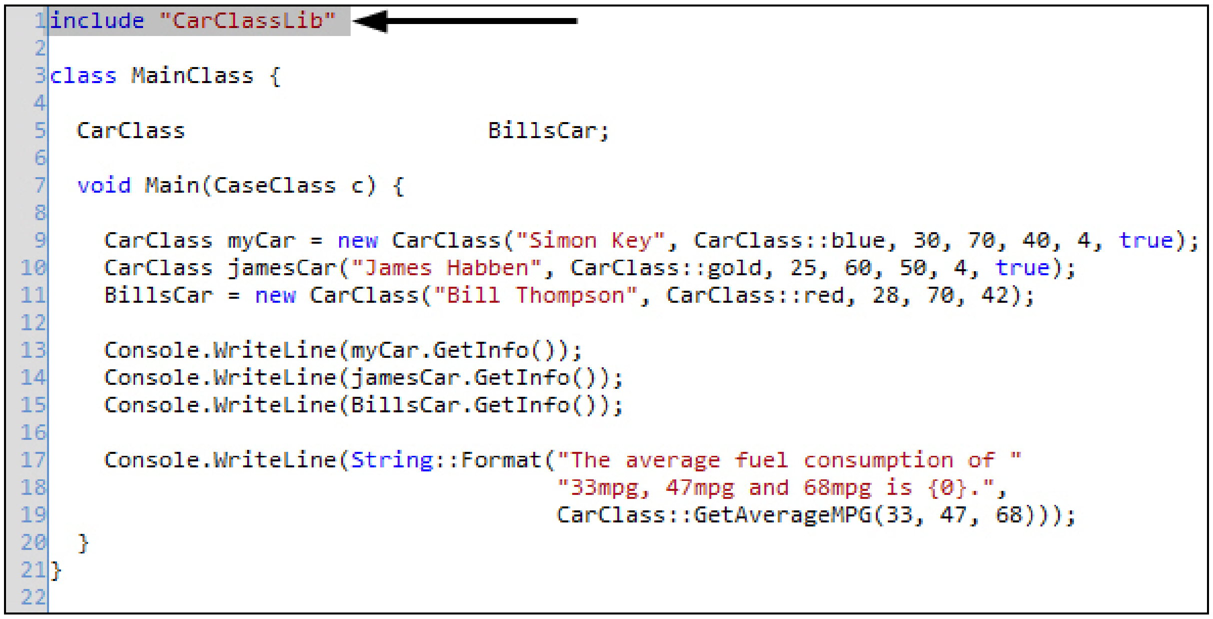
```
The car belonging to Simon Key is blue in colour and has 4 doors. The car does have air
conditioning.The car's average fuel consumption is 46.7.
The car belonging to James Habben is gold in colour and has 4 doors. The car does have air
conditioning.The car's average fuel consumption is 45.
The car belonging to Bill Thompson is red in colour and has 4 doors. The car does not have air
conditioning.The car's average fuel consumption is 46.7.
The average fuel consumption of 33mpg, 47mpg and 68mpg is 49.3.
```

Figure 5-26 Viewing the adjusted values in the output of the script

INCLUDES

Having finished developing CarClass we may decide to remove it from our executable EnScript file and write it into a separate EnScript *library* file, which should also have an EnScript extension.

This new file can be included in our executable EnScript program using an include statement, which is read by the compiler and actioned.



```
include "CarClassLib"

class MainClass {

  CarClass                        BillsCar;

  void Main(CaseClass c) {

    CarClass myCar = new CarClass("Simon Key", CarClass::blue, 30, 70, 40, 4, true);
    CarClass jamesCar("James Habben", CarClass::gold, 25, 60, 50, 4, true);
    BillsCar = new CarClass("Bill Thompson", CarClass::red, 28, 70, 42);

    Console.WriteLine(myCar.GetInfo());
    Console.WriteLine(jamesCar.GetInfo());
    Console.WriteLine(BillsCar.GetInfo());

    Console.WriteLine(String::Format("The average fuel consumption of "
                                     "33mpg, 47mpg and 68mpg is {0}.",
                                     CarClass::GetAverageMPG(33, 47, 68)));
  }
}
```

Figure 5-27 Using the include compiler directive

Unless the library file containing the class to be included is an *.EnPack file, it should only be referred to by name, not extension. The path to an included library file is relative to the path of the script. Additional paths (relative or absolute) can be set through the EnScript session options dialog.

NOTES

NOTES

ENSCRIPT™ FUNDAMENTALS PRACTICAL EXERCISE # 1

This practical exercise requires you to write an EnScript application (EnScript) that satisfies a number of requirements.

A screenshot of the required output, which should be written to the console window, is shown at the end of the practical exercise so that you can verify your results.

The requirements are:

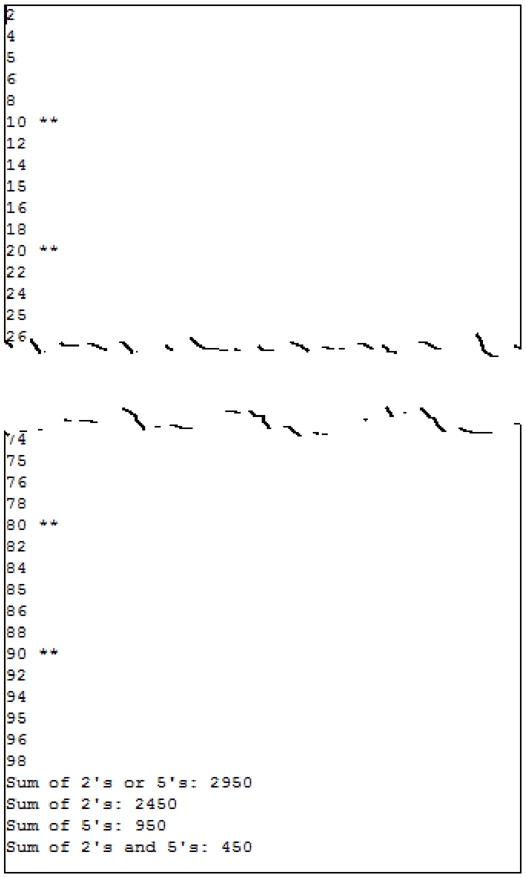
1. Your EnScript should display positive numbers with a value less than 100 that are multiples of 2 and 5. These numbers must be written in increasing numerical order.
2. Each number that is divisible by both 2 ***and*** 5 should be marked with a double asterisk.
3. Your EnScript should, having written the values required by in number 1, provide totals of those values that are:
   - Multiples of 2 or 5
   - Multiples of 2
   - Multiples of 5
   - Multiples of 2 and 5

You may need to use the modulus operator (%), which calculates the remainder left by dividing one integer by another. For instance:

- 13 % 5 results in an integer value of 3 (5 goes into 13 twice with a remainder of 3).

continued…

The output of the script should look similar to:



```
2
4
5
6
8
10 **
12
14
15
16
18
20 **
22
24
25
26
```

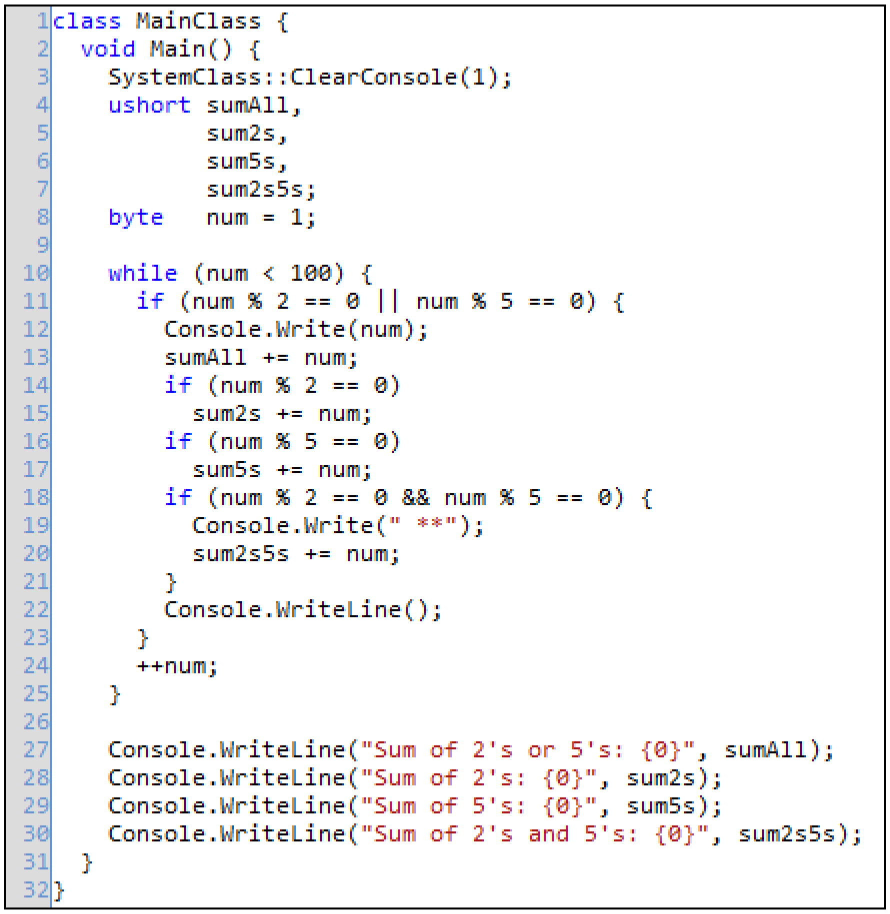
```
74
75
76
78
80 **
82
84
85
86
88
90 **
92
94
95
96
98
Sum of 2's or 5's: 2950
Sum of 2's: 2450
Sum of 5's: 950
Sum of 2's and 5's: 450
```

Sample script output – Practical Exercise #1

The EnScript code that produced this output is to be found over the page.

continued…

Example code:



```
class MainClass {
  void Main() {
    SystemClass::ClearConsole(1);
    ushort sumAll,
           sum2s,
           sum5s,
           sum2s5s;
    byte   num = 1;

    while (num < 100) {
      if (num % 2 == 0 || num % 5 == 0) {
        Console.Write(num);
        sumAll += num;
        if (num % 2 == 0)
          sum2s += num;
        if (num % 5 == 0)
          sum5s += num;
        if (num % 2 == 0 && num % 5 == 0) {
          Console.Write(" **");
          sum2s5s += num;
        }
        Console.WriteLine();
      }
      ++num;
    }

    Console.WriteLine("Sum of 2's or 5's: {0}", sumAll);
    Console.WriteLine("Sum of 2's: {0}", sum2s);
    Console.WriteLine("Sum of 5's: {0}", sum5s);
    Console.WriteLine("Sum of 2's and 5's: {0}", sum2s5s);
  }
}
```

Example code – Practical Exercise #1

This is just one example of how to produce the required output. If your code is different but obtains the same result then that's fine.

The second practical exercise follows.

-- end --

ENSCRIPT™ FUNDAMENTALS PRACTICAL EXERCISE # 2

This practical exercise requires you to write an EnScript application (EnScript) that satisfies a number of requirements.

A screenshot of the required output, which should be written to the console window, is shown at the end of the practical exercise so that you can verify your results.

The requirements are:

1. Add a string-variable to your EnScript that contains a passage of text of a reasonable length. You might, for example, choose a passage of text from a well-known book, *A Tale of Two Cities* by Charles Dickens for instance:

   It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way - in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

2. Add a function to your EnScript that will return the number of occurrences of a given character in a given string. Note that your function ***does not*** need to use an array or linked list in order to perform this task.
3. Use your new function to calculate the number of occurrences of each letter character and displaying the results in the console window. Your script should not write anything with regards to characters that don't exist; it should also ignore case sensitivity.

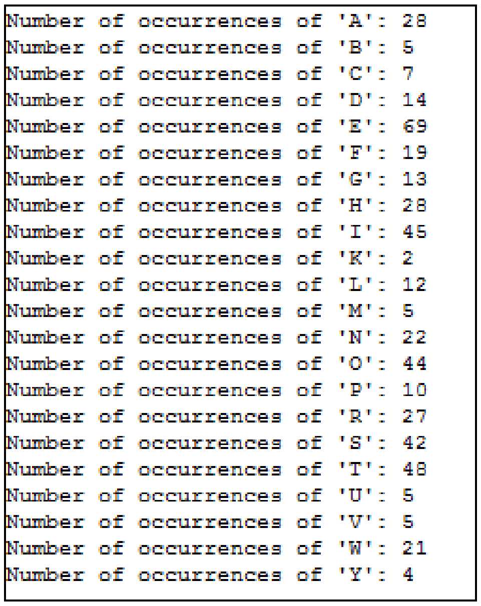
You ***may*** need to use one or more of the following functions:

- char::ToLower()
- char::ToUpper()
- String::ToLower()
- String::ToUpper()

Note that the two char functions are static whereas the two String functions are not. This will affect how you use them.

continued…

If you were to process the aforementioned text with your EnScript, its output should look similar to that displayed in the following image.



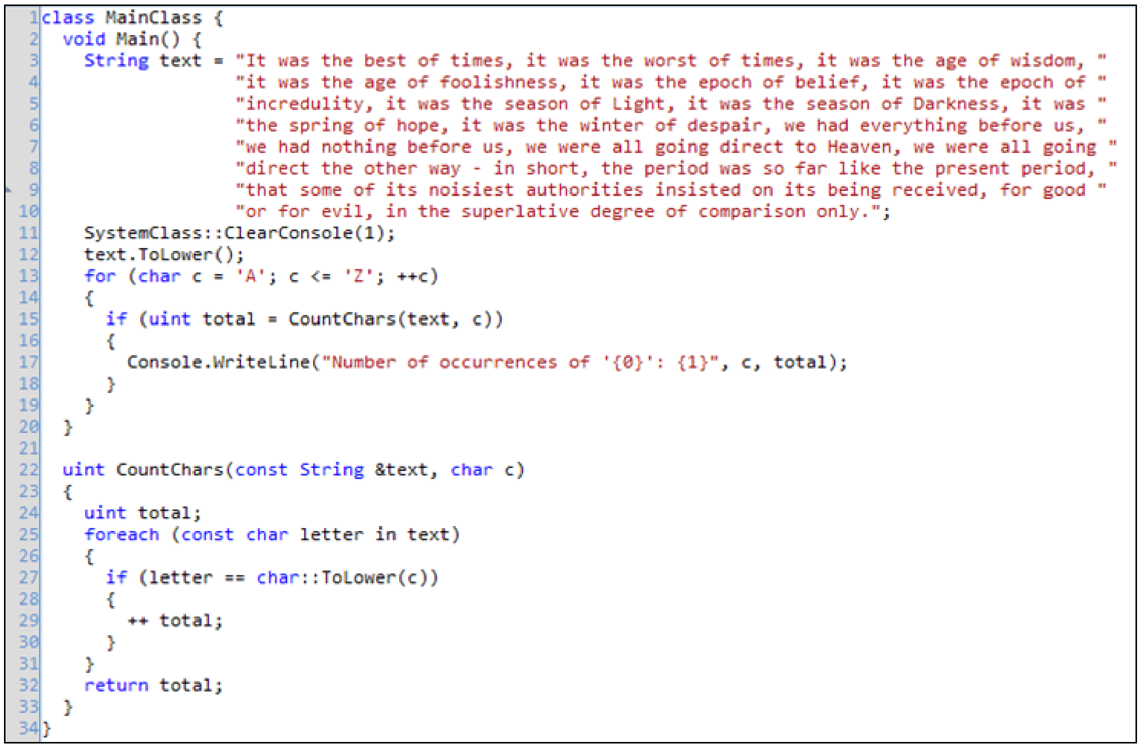
```
Number of occurrences of 'A': 28
Number of occurrences of 'B': 5
Number of occurrences of 'C': 7
Number of occurrences of 'D': 14
Number of occurrences of 'E': 69
Number of occurrences of 'F': 19
Number of occurrences of 'G': 13
Number of occurrences of 'H': 28
Number of occurrences of 'I': 45
Number of occurrences of 'K': 2
Number of occurrences of 'L': 12
Number of occurrences of 'M': 5
Number of occurrences of 'N': 22
Number of occurrences of 'O': 44
Number of occurrences of 'P': 10
Number of occurrences of 'R': 27
Number of occurrences of 'S': 42
Number of occurrences of 'T': 48
Number of occurrences of 'U': 5
Number of occurrences of 'V': 5
Number of occurrences of 'W': 21
Number of occurrences of 'Y': 4
```

Sample script output – Practical Example #2

The EnScript code that produced this output is to be found over the page.

continued…

Example code:



```
class MainClass {
  void Main() {
    String text = "It was the best of times, it was the worst of times, it was the age of wisdom, "
                  "it was the age of foolishness, it was the epoch of belief, it was the epoch of "
                  "incredulity, it was the season of Light, it was the season of Darkness, it was "
                  "the spring of hope, it was the winter of despair, we had everything before us, "
                  "we had nothing before us, we were all going direct to Heaven, we were all going "
                  "direct the other way - in short, the period was so far like the present period, "
                  "that some of its noisiest authorities insisted on its being received, for good "
                  "or for evil, in the superlative degree of comparison only.";
    SystemClass::ClearConsole(1);
    text.ToLower();
    for (char c = 'A'; c <= 'Z'; ++c)
    {
      if (uint total = CountChars(text, c))
      {
        Console.WriteLine("Number of occurrences of '{0}': {1}", c, total);
      }
    }
  }

  uint CountChars(const String &text, char c)
  {
    uint total;
    foreach (const char letter in text)
    {
      if (letter == char::ToLower(c))
      {
        ++ total;
      }
    }
    return total;
  }
}
```

Example code – Practical Example #2

It should be noted that, as already mentioned, the String::ToLower() function is not static; it must be applied to a string and will cause that string to be transformed.

The char::ToLower() function, on the other hand, is static. Rather than being applied to a char variable, it accepts a char variable and returns the lower case variant of it.

As mentioned previously, if your code is different but produces the same result, that's fine.

-- end --