

# EnScript<sup>®</sup> Tutorial

## *Getting Started with EnCase<sup>®</sup> Automation*

The purpose of this tutorial is to explain basic EnScript concepts. This tutorial is designed for EnCase users who are new to EnScripts and have little or no prior programming experience. Through this tutorial, I hope you learn the essential concepts needed to write some basic EnScripts and/or modify existing ones for a specific need.

The EnScript programming language is very C++ and Java-“ish”. If you have any experience with those two languages, then learning the EnScript language should be a snap. But, if you think C++ is almost as good as a B-, and Java is one of the major food groups, don't worry. We will break everything down into the fundamentals and it will all start making sense sooner than you think.

First, some disclaimers: I am not a programmer by profession. I learned the EnScript language out of necessity to automate processing of evidence. I have since written many EnScripts, some of which are now part of the public release version of EnCase and others are given to students during EnCase training, but I certainly do not consider myself an expert in writing EnScripts.

The information is provided as an instructional tool to introduce new users to some basics. There are no guarantees. So, don't just run some code (that you found here or elsewhere), and bet the farm on the results. Ultimately, the examiner must interpret and defend case findings.

This tutorial started as blog entries at [www.forensickB.com](http://www.forensickB.com). As I added new tutorials, readers had to travel back in time to track down previous entries. Danny Brown and the guys at CyberEvidence, Inc. copied down the blog entries and put them together into a single chronological document for easier reference. Danny called and asked if I was interested in collaborating on a tutorial that could be downloaded. This is the result. I would like to personally thank Danny for his hard work in reading through my tutorials and code examples and putting together this document.

Along the way, some new information was added. So, even if you have already read the blog entries, it might be worth another look.

With it's origins out of the free-flowing blog environment, I resisted the temptation to formalize the text. As such, the language and grammar may be a little casual.

If everything makes sense and all the examples work, you can thank me for contributing to the greater good. If something doesn't work — well, let's just say — it wasn't me.

After working through the tutorial, shoot me a line and let me know what you think – thumbs up, thumbs down, or suggestions for improvement.

Lance Mueller

# EnScript Tutorial - Part I

## A View From the Top

I have been teaching various EnCase training classes for almost 10 years and I know that the first exposure many users have to automation is when we discover the **EnScript** tab found in the lower right pane of EnCase and run some of the “canned” EnScripts (Initialize Case). But, alongside the **EnScript** tab, you will also find the **Conditions**, **Filters**, and **Queries** tabs.

Many EnCase users are confused over the differences between these features. Each serves a specific role in EnCase automation, and some of the functionality overlaps. So, let’s start by discussing these four topics so you understand their differences. This will help you decide which provides the best solution to meet your automation need. When you get the hang of it, you will be using all of them. After we look at the automation features, we will start adding some code to start making EnScripts perform some useful tasks.

## Conditions

A **Condition** is a special EnScript. It is the same language. Conditions **filter what you see**. The original concept was to filter files/folders based on some type of criteria; i.e. file extension, size, name, whatever. Conditions are the same as Filters (which we will discuss next), with one important exception, **you don’t need to know the EnScript programming language**. The **Conditions** tab allows you to use user-friendly criteria or selections to create a filter. By selecting certain criteria, such as name, contains, “mytext”, EnCase will automatically generate the necessary EnScript code to perform that filter.

## Filters

A **Filter** is another special EnScript. Like Conditions, filters serve a specific purpose, **to filter what you see**. The original concept was to filter files/folders based on some type of criteria; i.e. file extension, size, name, whatever. Only, with filters, **you** create and customize the code to meet specific needs.

EnCase actually does some background processing for you with a filter by automatically recursing all the evidence. Recursion means it looks at each entry in your evidence, evaluates it to see if it meets the condition(s) you specified and then asks, “Do you want to see this entry or do you want to hide it from view”? Recursion is an important concept in programming. Filters do it automatically, but we will be adding recursion to our EnScripts later on.

Since this tutorial focuses on getting started with EnScripts, I won’t get into the special considerations of Filters here. I moved the Filters info to the appendix of the tutorial.

## Queries

Prior versions of EnCase had limitations that allowed you to apply only one filter at a time. Let's say you have a filter that only shows you files that are larger than 10,000 bytes in size and another filter that only shows you files with the extension of JPG. What if you want to see only files that match both criteria? Solution - create a Query. A query is nothing more than two or more filters put together. By using a query you could take those filters and apply them simultaneously, the result would be only files whose size is greater than 10,000 bytes and that have a JPG extension would be displayed.

## EnScripts

An EnScript is the most powerful automation feature but it is also the most raw. "Raw" meaning that the EnCase software does very little for you automatically and the EnScript code you create is responsible for doing everything you want to do. The **EnScript** tab gives you access to the built-in EnScript editor and allows you to see the code for the EnScript, as long as it isn't compiled (EnPack format- more on that later).

An EnScript can do almost anything you want. It can access just about everything the user can access, or see, inside EnCase. It can also perform actions outside of EnCase, like creating folders and files on the local file system (*not the evidence - the evidence file can never be altered via EnCase*). In the Enterprise Edition, it can create directories and files on remote machines as well as delete them. It can also execute other win32 programs.

In the next part, I will begin to explain the EnScript programming language and how to perform simple actions.

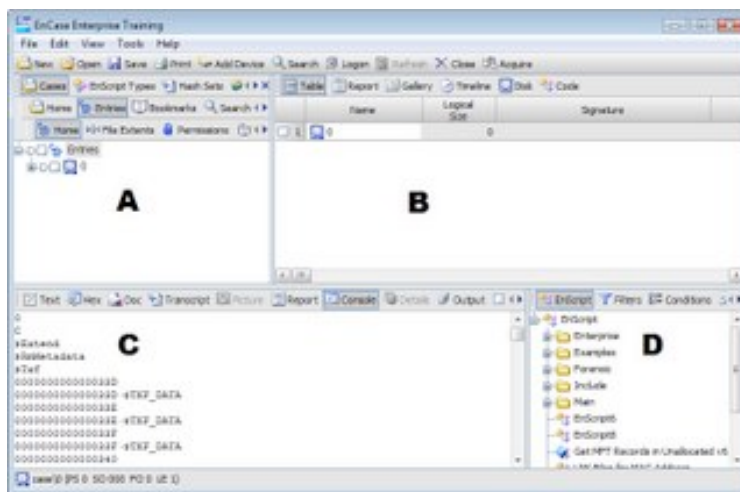
# EnScript Tutorial - Part II

## "Hello World"

In Part I, we reviewed the differences between Conditions, Filters, Queries, and EnScripts. In this part, we will begin to learn the EnScript programming language. Remember, when writing an EnScript, EnCase provides the basic structure. You are responsible for writing everything else. So, how do you make it do something?

First, let's review the various panes in EnCase and their respective names.

The EnCase program divided into four general panes or sub-windows.



(A) The upper left pane is the "tree pane".

(B) The upper right pane the table pane.

(C) The lower left pane is the view pane, and

(D) The lower right pane is the EnScripts/Filter pane.

The table pane generally shows you all the "objects" or files &

folders in a particular piece of evidence. The view pane displays the contents of a specific file or folder in various formats (text, hex, doc viewer).

## Getting Organized

Throughout the tutorial, we will create a number of EnScripts. Before we start cluttering up your **EnScript** tab, we should create a folder to keep them organized.

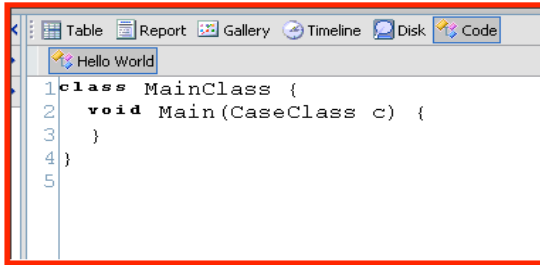
In the EnScripts/Filter pane (lower right), make sure the **EnScript** tab is active and then right-click on the root of the EnScript tree. Select "New Folder" and then name the folder "Lance EnScript Tutorial". And yes, you can name it something else, if you must. A new folder will be created with the name you provided.

## Writing Your First EnScript:

What programming tutorial would be complete without the traditional "Hello World" programming example?

In the EnScripts/Filter pane (lower right), make sure the EnScript tab is active and then right-click on the folder "Lance EnScript Tutorial". Select "New" and then name the EnScript "Hello World". A new EnScript will be created with the name you provided. The table pane (upper right) should have automatically made the "**Code**" tab active.

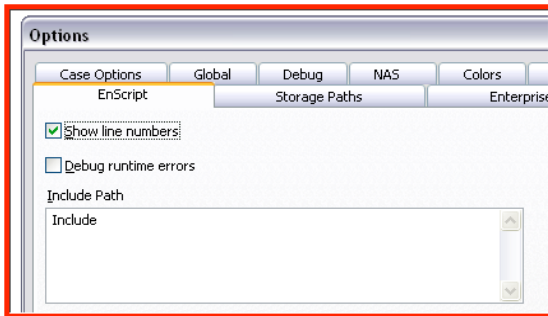
You should see the minimum EnScript code generated automatically by EnCase:



```
1 class MainClass {
2   void Main(CaseClass c) {
3   }
4 }
5
```

This is the minimum amount of code that must be present in order to be a valid EnScript. This EnScript will run, but will do absolutely nothing.

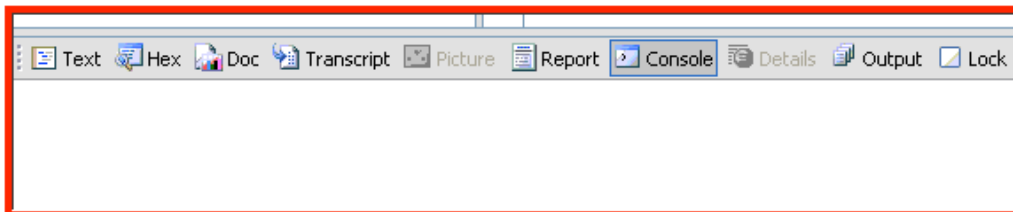
EnCase version 6.8 added several new EnScripting features. The first new feature, line numbering, is a very welcome addition. It is turned off by default, but if you go to "Tools" -> "Options" -> EnScript Tab. Checking the "Show line numbers" box will immediately enable line numbering in any/all EnScripts you may have open for editing.



Then, when you compile the EnScript, if there are any syntax errors, EnCase will generate an error in the "Output" tab

and show you the vertical line number and horizontal position of the code that generated the error. Very cool!....

In the view pane, there is a tab named "Console".



The **Console** tab is an output window for when you want your EnScript to write out information. You generally do not want to write important information, but instead use the **Console** as a kind of testing/debugging window to write out the status of your EnScript.

*OK Lance, could we make something happen already? Oh, Sure.*

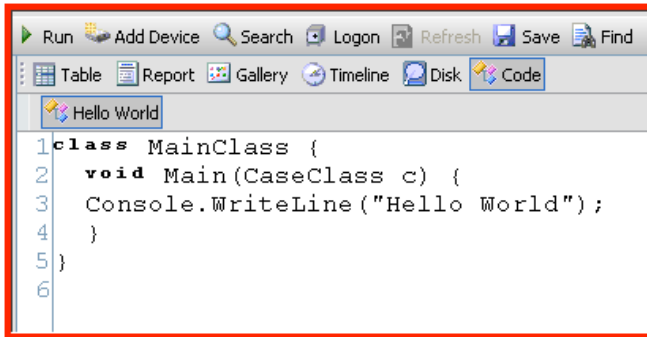
To print information to the Console, we use the "WriteLine" method.

To write "Hello World" to the Console, you would use this syntax:

```
Console.WriteLine("Hello World");
```

Notice that the text you want to appear in the Console tab "Hello World" is inside of "double quotes" and that the line ends with a semi-colon. You will generally end every line with a semicolon. There are exceptions, and we will discuss them later as they come up.

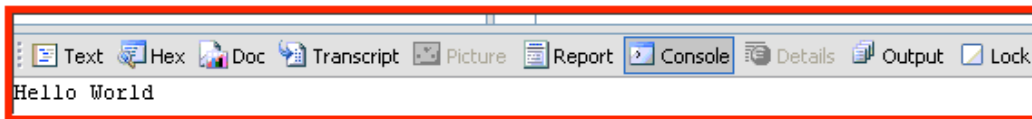
Add the line and your code should look something like this:



```
1 class MainClass {
2     void Main(CaseClass c) {
3         Console.WriteLine("Hello World");
4     }
5 }
6
```

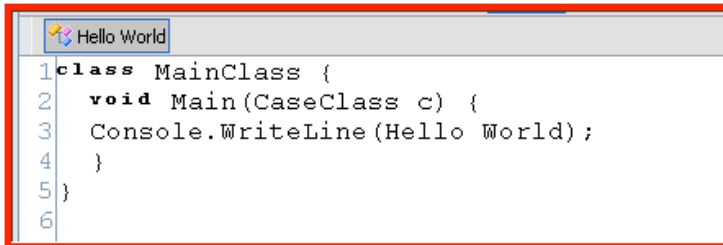
Now click on the **Run** button at the top of EnCase. When you hit the **Run** button, the EnScript is “compiled”, or converted to machine executable form. If everything checks out OK, the code runs, and away you go.

Switch to the **Console** tab. You should see “Hello World”.



```
Hello World
```

If so, great. If not, it's a good time to talk a little about syntax and debugging code. If the compile fails, you will see an error displayed in the **Output** tab. The message will vary, depending on the error. The code syntax is very specific and even a minor typo can prevent the script from running.

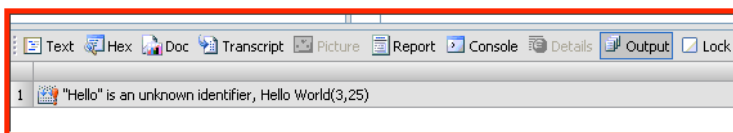


```
1 class MainClass {
2     void Main(CaseClass c) {
3         Console.WriteLine(Hello World);
4     }
5 }
6
```

For example, I have introduced a minor error in my code.

Check the code very carefully to be sure everything is correct (watch out for typos, capitalization, curly braces, parentheses, quote marks (“), semi-colons etc).

Here is the error displayed in the **Output** tab:

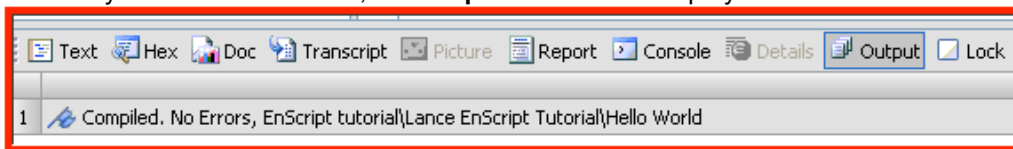


```
1 "Hello" is an unknown identifier, Hello World(3,25)
```

See the error? The “quote” marks are missing from “Hello World”.

Make a change and hit the **Compile** button. This will also identify any problems.

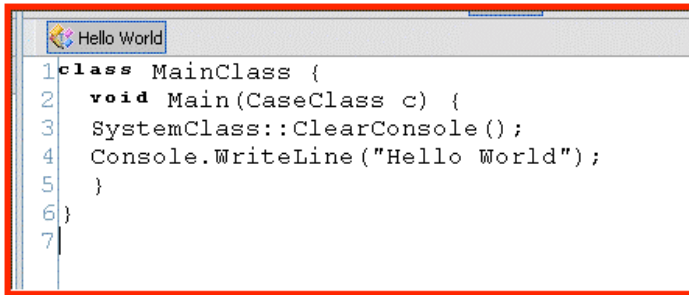
Once any errors are corrected, the **Output** window will display:



```
1 Compiled. No Errors, EnScript tutorial\Lance EnScript Tutorial\Hello World
```

Now hit the **Run** button and look at the **Console** tab. Hit the **Run** button several more times. What happens? Each time it runs, a new **"Hello World"** gets added to the list. So, why didn't it delete or write over the first **"Hello World"**? Because the EnScript does what it is told, and you didn't explicitly tell it to clear out any previous entries.

To programmatically clear the Console each time you run an EnScript, you could add:

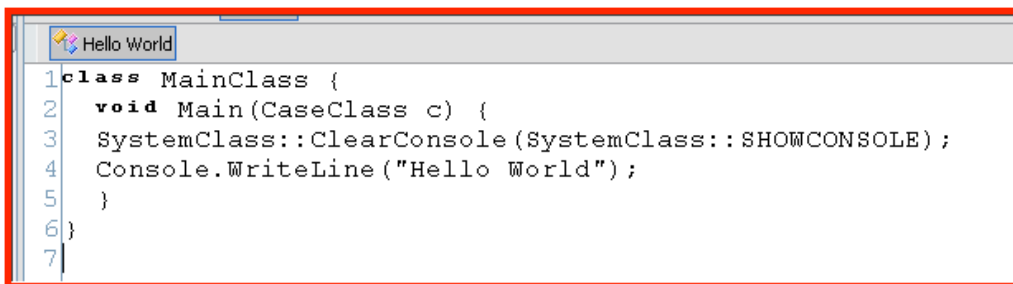


```
1 class MainClass {
2     void Main(CaseClass c) {
3         SystemClass::ClearConsole();
4         Console.WriteLine("Hello World");
5     }
6 }
7
```

Hit Run.

Switch over to the **Console** tab and you should see a single **"Hello World"**.

OK, now try this. Modify **ClearConsole** and add the following parameter:



```
1 class MainClass {
2     void Main(CaseClass c) {
3         SystemClass::ClearConsole(SystemClass::SHOWCONSOLE);
4         Console.WriteLine("Hello World");
5     }
6 }
7
```

Before you run it, make sure the **Console** tab is not active.

Hit **Compile**. If everything is OK, hit **Run**. In addition to clearing the **Console**, the **SystemClass::SHOWCONSOLE** parameter will switch to the **Console** tab. **Sweet**.

**And, And**, you can just put a "1" in the parens instead of **SystemClass::SHOWCONSOLE**. Does the same thing.

*WHAT? Lance, walk toward the light buddy. Could you slow down a little? I mean, how am I going to remember all this?*

OK, let's take a step back. Before we start throwing in a lot of code, now is a good time to talk about good programming practice. Indenting, formatting, and adding comments will help you (and others) document and understand what your EnScript is doing.

You can comment one line of code using two forward slashes “//”.

// After that, everything on the rest of the line will turn blue and be ignored at run time.

```
1 class MainClass {
2     void Main(CaseClass c) {           // Execution starts here
3         SystemClass::ClearConsole(1); // Clears the Console and (1) shows the Console
4         Console.WriteLine("Hello World"); // Writes out to the Console tab
5     }                                   // Execution stops here
6 }
7
```

To comment blocks of information, you can use the ANSI C commenting style of a “/\*” and then end your commenting block with the opposite “\*/”.

/\* Anything in-between these markers will turn blue and will be ignored at run time. The markers can be on the same line, or they can be 100 lines apart and everything in between will be ignored. \*/

```
1 class MainClass {
2
3     void Main(CaseClass c) {           // Execution starts here
4
5         SystemClass::ClearConsole(1); // Clears the Console and (1) shows the Console
6
7         Console.WriteLine("Hello World"); // Writes out to the Console tab
8
9     }                                   // Execution stops here
10
11 }
12
13 /* Anything in-between those markers will turn blue and will be ignored.
14 Those markers can be on the same line, or they can be 100 lines apart.
15 Everything in between will be ignored. */
16
```

### Here are some basic formatting rules when writing EnScripts:

- ▶ Make comments in your code (for future reference or explanation).
- ▶ White space is generally ignored (the exception is when inside double quotes). So, putting extra spaces between lines of code means nothing and it can help to logically separate various pieces of your code.
- ▶ Get in the habit of indenting your code inside functions, control structures or conditional statements. It makes your code easier to read and helps when debugging for errors
- ▶ **Control-Z** is the hot key for undo. So if you delete something or change a piece of code, but then want to undo your change, **Control-Z**
- ▶ **Control-F** will find specific text in your **EnScript**. When working with small EnScripts, finding text is not too difficult, but with larger ones, it helps to find variables, functions or specific text.

The formatting and comments help document what your EnScript is doing. Not a big deal on small EnScripts, but extremely helpful in documenting, understanding, and debugging larger, or more complex EnScripts.

So, there you are. Your first full blown, hand coded, debugged, fully functional EnScript. Impress your friends. Bask in the glow. In the next section, we go beyond “Hello World” and write EnScripts that do useful stuff.

# EnScript Tutorial III

## Beyond “Hello World”

Now let’s create an EnScript that does some useful work. First, we need to learn more about the EnScript environment. After that, we will create an EnScript to:

- list the name of the first root level item in the evidence
- list the name of the last root level item in the evidence
- Add a technique to list all root level items in the evidence
- Add a technique to list all items in the evidence
- Combine a text string with a variable to clarify the output

*The previous script did not require a case to be open because it was simply writing to the Console. The remainder of this tutorial will require you to have a case open and some evidence loaded.*

In the EnScripts/Filter pane (lower right), make sure the **EnScript** tab is active and then right-click on the folder “**Lance EnScript Tutorial**”. Select “New” and then name the EnScript “**List Evidence Items**”. A new EnScript is created with the name you provided. The table pane (upper right) should automatically make the “**Code**” tab active and you should see the minimum EnScript code generated automatically by EnCase.

```
1 class MainClass {
2     void Main(CaseClass c) {           // Execution starts here
3     }                                   // Execution stops here
4 }
```

## Structure

Every EnScript must have a **MainClass**. Lines 1 and 4 make up the MainClass. Take a look at the opening curly brace ( { ) and then the corresponding closing curly brace ( } ).

Think of curly braces as bookends. They mark the beginning and end of a block. For now, make a mental note that for every beginning, or open curly brace ( { ), there must be an ending, or closing, curly brace ( } ).

Inside the **MainClass**, every EnScript must also have a **Function** called “**Main**”.

The first line of the **Main** function is where this EnScript begins execution. Line 2 “void **Main(CaseClass c) {**” is where it will start. Generally, program execution stops with the corresponding closing curly brace “**}**” of the **Main** function (currently line 3).

When this EnScript begins, the EnCase program is going to hand your **Main** function one **variable**, named “**c**”. This **variable** is of the **CaseClass** type.

Excuse me, Lance. I'm havin a little difficulty getting my head around the whole Variable /Function/Parameter thing. Can you explain that? OK. But, only because you asked.

Step into the classroom.

*A Variable is an important concept in programming. If you survived Algebra, you may still have nightmares over the value of "x". We refer to "x" as a variable (its value varies). Another way to look at it is to think about cell "A10" in a spreadsheet. "A10" refers to a specific cell and it can contain text, numbers, dates, formulas, references etc. If you think of "x", "A10", and "c" as places to store values, then you begin to understand the concept of variables.*

*Functions perform useful operations. Spreadsheets use functions to make calculations easier. For example,*

A reference to "c", our **CaseClass** object, allows an EnScript to access items in the tabs (like entries, devices, bookmarks, E-mails, and History). This is your starting point.

This means that when your program begins, you will have a reference to the case that is open and active in EnCase at the time the EnScript executes. With the "c" variable (referencing the CaseClass type), you can obtain additional references or pointers to all the other information you may need in your EnScript.

There is a built-in method that is part of the **CaseClass** that gives you the top-level entry in the evidence using the method "**c.EntryRoot()**". This would get the first root entry in the evidence (think of how a hierarchical directory works with a top-level root directory). You could then print the name of that object out using the **Console.WriteLine()** function.

You could use the code:

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         SystemClass::ClearConsole(1); // Clear the Console
4         Console.WriteLine (c.EntryRoot().FirstChild().Name());
5     } // Execution stops here
6 }
```

Add the code and hit Run. So far, so good? But I know what you are thinking:

*"So Lance, what's up with all the Capitalization.And.Periods.?"*

That wasn't it? Well, we need to talk about it anyway.

Like any other written language, the EnScript language has strict rules for sentence construction, spelling, grammar, and punctuation. Normally, when entering syntax, the first letter of each word is capitalized. Also, in EnScripts, we separate each part with a period (.) usually called a "dot".

So, to print out the name of the first evidence item, we use:

```
Console dot WriteLine ( c dot EntryRoot() dot FirstChild() dot Name() );
```

After clearing the Console, the EnScript will:

- Go to the CaseClass (we call it "c")
- From there it can reference the **EntryRoot**
- From the **EntryRoot** we can get the **FirstChild**
- Print it's **Name** to the **Console** tab.

So, what if you want to print the name of the last physical or logical device? Don't look! Think about it, add a line of code and run it. Not enough coffee yet?

OK, here is what it looks like:

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         SystemClass::ClearConsole(); // Clear the Console
4         Console.WriteLine (c.EntryRoot().FirstChild().Name());
5         Console.WriteLine (c.EntryRoot().LastChild().Name());
6     } // Execution stops here
7 }
```

Enter and hit Run. This code would clear the **Console** and print the name of the **first** and **last** physical or logical device to the **Console** pane.

*What's that? How do we print the name of all root level physical or logical devices?*

Good question. Think about the possibilities. You may have 1 to "x" number of physical or logical devices (nice way to sneak in "x" don't you think)? If you have added at least one device, then you can see the FirstChild and LastChild, but how do you see the second, third, fourth, or..., or ninety-ninth? What we need is a method of accessing **each** root level entry (from 1 to x) so we can print out its name.

Sounds simple enough. We have been working with the CaseClass right? With a reference to CaseClass we have access to its objects, functions, methods and properties. But, after rummaging around in the CaseClass toolkit, there isn't a tool designed for handing every root level entry. *So, what do we do?*

Uh, Oh. I feel a teachable moment coming on.

Let's switch gears for a moment. Remember the purpose for a **Filter**? That's right - to show only entries that meet our criteria.

Here is the starting code for a Filter:

```
1 class MainClass {
2     bool Main(EntryClass entry) {
3         return true;
4     }
5 }
```

Why **Main(EntryClass entry)** instead of **Main(CaseClass c)**? Because the job of a Filter is to show (or hide) specific **entries**, so, we need access to the **EntryClass**.

Bonus question: What is "**entry**"? It is the variable name we assigned to the **EntryClass** object.

CaseClass, EntryClass, what's the difference? Think of classes as toolkits designed for specific purposes. The CaseClass toolkit gives us a way to access and work with general case related objects, while the EntryClass toolkit gives us some additional tools to work with entries. When we reference a particular class, we can "inherit" (or gain access to) its tools (objects, functions, methods, properties etc).

Got it now? OK. Back to the Main show. We want to print the name of each root level physical or logical device.

Here is what it looks like:

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         SystemClass::ClearConsole(1); // Clear the Console
4         Foreach ( EntryClass entry in c.EntryRoot() ) {
5             Console.WriteLine (entry.Name()); //Writes to the Console tab
6         }
7     } // Execution stops here
}
```

```
8 }
```

### Key Points

1. Since we will be accessing **entries**, we create a **variable** named **"entry"** of the **EntryClass** type. This variable will hold a pointer to the current root entry (**EntryRoot**).
2. The **foreach** function allows us to recurse through **each** physical or logical device in **EntryRoot**.
3. Watch out for the parentheses, curly braces, and semicolons.

This code clears the Console (line 3), then receives the top-level object in the case and assigns it to the variable named **"entry"**. It then executes all the code after the **foreach** statement until the closing curly brace (line 6), then goes back to the **foreach** statement and gets the next root level object in the case, assigns it to the **entry** variable and runs the code in between the curly braces again, until all the root level objects have been processed. If you look in the **Console** tab, you should see a complete listing of all the root level names from all of the loaded evidence.

To make the output a little clearer, add the following to WriteLine:

```
1 class MainClass {
2     void Main(CaseClass c) {           // Execution starts here
3         SystemClass::ClearConsole(1); // Clear the Console
4         foreach ( EntryClass entry in c.EntryRoot() ) {
5             Console.WriteLine ("Root Level Device: " + entry.Name());
6         }
7     }                                   // Execution stops here
8 }
```

This technique uses the "+" operator to concatenate the text string **"Root Level Device: "** with the name of the current **entry** to produce a clearer output.

Let's take it a step further. What if we want to print out the name of **all** objects, folders, and files in the evidence? There is another internal function that makes this *\*very\** simple. If you are familiar with Perl, you should be familiar with **"forall"** function that goes through a list or array, one object at a time. In EnCase it works the same way. Given the top-level object, **forall** will iterate through every object in the evidence.

```
1 class MainClass {
2     void Main(CaseClass c) {           // Execution starts here
3         SystemClass::ClearConsole(1); // Clear the Console
4         Forall ( EntryClass entry in c.EntryRoot() ) {
5             Console.WriteLine (entry.Name());
6         }
7     }
8 }
```

```
7     } // Execution stops here
8 }
```

This code clears the Console (line 3), then receives the top-level object in the case and assigns it to the variable named "entry". It then executes all the code after the **forall** statement until the closing curly brace (line 6), then goes back to the **forall** statement and gets the next object in the case and runs the code in between the curly braces again, until all the objects have been processed.

If you look in the **Console** tab, you should see a complete listing of all the file/folder names from all of the loaded evidence. Oh, yeah, if you have large (or a lot of) evidence files loaded, this EnScript may produce a really, really, long list.

Run it anytime someone gets near, so everyone in the office will know what a study code banger you are. Sit back, and bask in the glow. And, wait for the Rookie to ask, *"Excuse me Officer Studly, but, like, could you use an EnScript to locate a particular file? Rookies!*

Send the new guy for donuts and while the Rookie is gone, turn the page and learn how to combine the techniques we have already learned and add conditional statements to identify and process file/folders based on specific criteria.

# EnScript Tutorial Part IV

## Conditional Statements

In Tutorial III, we finished by building a basic EnScript that recursed the evidence and printed the name of evidence entry objects to the Console. While there are instances where you want to output everything, in most cases, you really want to locate specific items and process only those items.

In this tutorial, we will:

- ▶ Create conditional control statements that make decisions based on certain criteria.
- ▶ Do some other stuff

In Tutorial III, we finished with the following code:

```
1 class MainClass {
2     void Main(CaseClass c) {
3         SystemClass::ClearConsole(1);
4         forall ( EntryClass entry in c.EntryRoot() ) {
5             Console.WriteLine (entry.Name());
6         }
7     }
8 }
```

This prints out the names of **all** objects.

## Conditional Control Structures

The next step is to use conditional control structures to select files based on certain criteria.

- Scenario: We want to recurse through the evidence and find a file named “**system**”. When we find it, we want print out information about the item to the **Console**.

The code we created earlier printed the name of all entries to the console. Now, we want to be much more selective about which item(s) are printed. We will use the same code to recurse through all the entries, but we will add code to evaluate and identify entries of interest.

As a question, it might be phrased like this: *Is the **name** of the current entry equal to “system”?* We might expect the answer to be either “**Yes**” or “**No**”. Most of the time the answer will be “**No**”. But, whenever the answer is “**Yes**”, **then** we have found an item of interest and will probably do something with it.

To achieve this programmatically, we will use the “**if**” conditional statement.

Here is how we accomplish the task using the “**if**” structure:

```
if (entry.Name () == “system”)
```

The **if** statement will evaluate a condition and return a Boolean “**true**” or “**false**”.

Essentially, it means that **if** the name of the current **entry** is equal to “**system**” (returns “**true**”), we have found the **entry** we are searching for and will **then** take some action on that **entry** (print it’s name).

However, **if** the name of the current **entry** is not equal to “**system**” (returns “**false**”), we do not have the **entry** we are searching for and will **not** take any action on that **entry**. In this example, we simply move on to the next entry and try again.

It is important to note that this type of evaluation is **case-sensitive** and the text strings must be an **exact match**. **if** (entry.Name()=="system") would evaluate to **false** for “**System**” or “**system.dat**” and the conditional code would not execute.

There are two ways to code control statements.

The first way to handle “**if**” statements assumes you want to execute only one line of code (the **very next line**) if the conditional statement evaluates **true**. Example:

```
1 class MainClass {
2     void Main(CaseClass c) {
3         SystemClass::ClearConsole(1);
4         forall (EntryClass entry in c.EntryRoot()) {
5             if(entry.Name()=="system") // if true then execute next line
6                 Console.WriteLine (entry.Name()); //Writes to the Console
7         }
8     }
9 }
```

**if** the name of the current **entry** is equal to “**system**” (returns “**true**”), **then** we do have the **entry** we are searching for and the next statement, **Console.WriteLine**, will be executed.

```
1 class MainClass {
2     void Main(CaseClass c) {
3         SystemClass::ClearConsole(1);
4         forall (EntryClass entry in c.EntryRoot()) {
5             if(entry.Name()=="system") // if false then skip next line
6                 Console.WriteLine (entry.Name()); //if false, skip this line
7         }
8     }
9 }
```

**if** the name of the current **entry** is not equal to “**system**” (returns “**false**”), **then** we do not have the **entry** we are searching for will not take any action on that **entry**. The next statement, **Console.WriteLine**, will be **skipped** and the EnScript will continue on line 7 (and jump back to the **forall** loop and get the next **entry** for processing.)

The second way of handling **if** conditional statements is to use an opening curly brace **{** and then a closing curly brace **}**, putting any code you want to execute if the evaluation is **true**, inside the braces. This form is **necessary** if you are executing more than one line of code.

In this example, if the **entry.name** is equal to “**system**” then we execute everything between the open curly brace **{** and the ending curly brace **}**.

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         SystemClass::ClearConsole(1); // Clear the Console
4         forall (EntryClass entry in c.EntryRoot()) {
5             if(entry.Name()=="system") { // if true then execute block
6                 Console.WriteLine (entry.Name()); //Writes to Console
7                 Console.WriteLine (entry.LogicalSize()); //Writes to Console
8             }
9         }
10    } // Execution stops here
11 }
```

If it is “**true**”, **then** any code in the curly braces will execute.

However, **if** the name of the current **entry** is not equal to “**system**” (returns “**false**”), **then** we do not have the **entry** we are searching for will not take any action on that **entry**. The EnScript skips the *two* **Console.WriteLine** statements and continues.

Side bar, Your Honor? OK. I kinda snuck in another property of the current entry (**entry.LogicalSize**) to get you thinking.

Field Trip! In EnCase, go to the Help menu, click on EnScript Help, then click the Search tab and type in “EntryClass”. Click List Topics and double-click on EntryClass.

Scroll down on the screen to the right and take a look at all the Properties of the EntryClass.

Think about how you would use this new knowledge to modify your EnScript to write additional data to the Console. Now, back to the show.

So, what if you get everything right, no errors, and you still don’t see anything in the **Console** window? Well, what happens if your evidence file doesn’t have a file named “**system**”? What prints to the **Console**? **Nothing!**

To prove that the EnScript is running, add the following line:

```
1 class MainClass {
2 void Main(CaseClass c) { // Execution starts here
3   SystemClass::ClearConsole(1); // Clear the Console
4   forall (EntryClass entry in c.EntryRoot()) {
5     if(entry.Name()=="system")
6       Console.WriteLine (entry.Name()); //Writes to the Console tab
7     Console.WriteLine (entry.LogicalSize());
8   }
9 }
10 Console.WriteLine ("EnScript finished");
11 // Execution stops here
12 }
```

Run the EnScript, and, even if you **don't** have an **entry** named "**system**", at least you have peace of mind that the code ran and all is well in your little EnScript world.

The Rookie should be back from the donut shop about now and will surely be impressed with your EnScriptin prowess. You have now reasserted your Alpha status.

However, you can already see the question building, "*Officer Study, what if the file you are looking for is kinda like "System", or "system.dat", or stuff...?"*

*"Go get me some coffee while I whip out some code and show you."*

The next section expands on some of the built-in functions available to make the job easier.

# Tutorial V

## String Functions

Searching, comparing, and manipulating “strings” is an important part of creating useful EnScripts, and EnCase provides a number of built-in functions to help make matching text a little easier. In this tutorial, we will:

- Use the Contains() function to find partial matches
- Use the Compare() function to add options for case sensitivity

In tutorial IV we ended up with the following code:

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         SystemClass::ClearConsole(1); // Clear the Console
4         forall (EntryClass entry in c.EntryRoot()) {
5             if(entry.Name()=="system"){
6                 Console.WriteLine (entry.Name()); //Writes to the Console tab
7                 Console.WriteLine (entry.LogicalSize());
8             }
9         }
10        Console.WriteLine ("EnScript finished");
11    } // Execution stops here
12 }
```

Remember, this type of evaluation is **case-sensitive** and must be **exact**.

There are times we need a little more flexibility. We are going to use a couple of built-in functions to add flexibility to our searches.

Generally, you pass inputs or “parameters” to functions. The function processes the information and “returns” a value. If the concept of functions is still a little vague, now is a good time to reread the Cyber Chalkboard on page 10.

## Contains()

What if you don’t know the exact file name and want to identify any file name that contains “**sys**”. The **Contains()** function looks inside a test string for a fragment and then returns a Boolean **true** if that fragment is inside the initial string. For example:

```
if (entry.Name().Contains("sys")){
// code goes here
}
```

In this example, as the EnScript recurses each object in the evidence, it will evaluate the **Name** field to see if it “contains” the letters “**sys**”. If it does, then it evaluates **true** and the code inside the curly braces would execute.

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         SystemClass::ClearConsole(1); // Clear the Console
4         forall (EntryClass entry in c.EntryRoot()) {
5             if(entry.Name().Contains("sys")==true){
6                 Console.WriteLine (entry.Name()); //Writes to Console tab
7                 Console.WriteLine (entry.LogicalSize());
8             }
9         }
10        Console.WriteLine ("EnScript finished");
11    } // Execution stops here
12 }
```

The **Contains()** function is **not** case-sensitive and it does not matter where the string fragment is located inside the initial string. Any file name containing “sys” will result in the function returning **true**. The file name “**system**.dat” would evaluate **true**, as would the file named “mms**system**.dll” since they both have “**sys**” inside them somewhere.

Why is there an “==0” at the end of the statement? Well, look at the whole statement. The **if** conditional statement must evaluate to a Boolean true or false. The **Contains()** function will return a Boolean true for any file name containing “sys”. Essentially, the remainder of the statement asks: Is true equal to true? The answer would be true. Got that?

*Hey, Lance, the **if** statement evaluates to a Boolean true or false, right? Right.*

*So, if the **Contains()** function returns a Boolean true, is it really necessary to do the “==true” stuff? I mean, I mean, isn’t that redundant? Well, yes it is, yes it is. Nice catch.*

So, get rid of it. Like this:

```
if(entry.Name().Contains("sys")){
```

Run it.

*Hey, Lance, if you knew that all along, why did we have me go through all the “==true” stuff.*

Because, it sets up a teachable moment that makes the next function easier to understand.

*So, you set me up?*

Yep.

## Compare()

What if you know the name of the file you need, but you need some flexibility in evaluating based on case sensitivity? Then you can use the **Compare()** function.

**Compare()** takes two inputs,

1. The string you want to evaluate
2. A parameter to indicate whether you want to evaluate on **case sensitivity**. By default, the function assumes you do **not** want to test based on case sensitivity.

In this example, we test the entry.name to determine whether it is **equal to** "system".

For example:

```
if (entry.Name().Compare("system") == 0) {  
  // code goes here  
}
```

Enter the code and run it.

```
1 class MainClass {  
2   void Main(CaseClass c) { // Execution starts here  
3     SystemClass::ClearConsole(1); // Clear the Console  
4     forall (EntryClass entry in c.EntryRoot()) {  
5       if(entry.Name().Compare("system")==0) {  
6         Console.WriteLine (entry.Name()); //Writes to Console  
7         Console.WriteLine (entry.LogicalSize());  
8       }  
9     }  
10    Console.WriteLine ("EnScript finished");  
11  } // Execution stops here  
12 }
```

You should get only files named "system", but case doesn't matter because the Compare() function is not case sensitive (by default).

### Breaking it Down

**Compare()** doesn't return a Boolean. It returns a numeric value.

**Compare()** returns a value of zero (0) for any text which is lexically identical to the text you are looking for.

**Compare()** returns a non-zero value for any text which is lexically less than, or greater than, the text you are looking for.

This function will return a zero (0) for any entry named "system", including any of the permutations like "System", or "SYSTEM" (all 2<sup>6th</sup> power of them).

I know, it seems reversed since a zero traditionally means **false** in Boolean. Sorry, but that's the way it is. But, think about it. Now, for once, being a zero is a good thing. That should make budding EnScripters who spent most of the seventh grade stuffed in lockers feel better about themselves. Or not.

Why is there an “==0” at the end of the statement? Well, look at the whole statement. The **if** conditional statement must evaluate to a Boolean true or false, but the Compare() function only returns a numeric value. We have to test the return value of the Compare() function to see if it is equal to zero (0). Got it?

Think of it like this, *“If the return value of the Compare() function is equal to zero (0), then execute the following line(s)”*.

Now, this is one of those times when the more you explain something, the more confusing it becomes. So, here goes.

Now, let's finish this thing out. Modify your code to add the following:

```
if (entry.Name().Compare(“system”, false) ==0 ) {
```

Run the code. Did anything change? No. Why? Remember, the Compare() function has a parameter to indicate if it is case sensitive. What was the default? The default is **false** (not case sensitive). So, including it in the Compare() function doesn't change the result.

Change it to **true**. Run it again.

```
if (entry.Name().Compare(“system”, true) ==0 ) {
```

True makes the function case sensitive.

*Uh, hey, Lance. This code makes my head hurt. Doesn't it do the same thing as the code we used earlier: **if entry.Name() == “system”** ?*

Well, yes it does.

*Then, why use it.*

Compare(), and other string handling techniques can be combined to offer us a lot of additional options that we won't get into here. Trust me, it's a good thing.

*“What's a good thing – the fact that it offers additional options – or the fact that you won't get into them here?”*

Both. Let it go.

To learn more about internal functions such as **Contains()**, you can go to **“View” > “EnScript Types”** Click on the **“String”** class in the left column and switch the right pane to **“Table”**. The **Contains()** function is a member of the **StringClass**. Look at the other string functions and a brief syntactical explanation of their use.

When the Rookie gets back with your coffee, (s)he will no doubt have gained new respect for your laser-sharp code skills.

*OK, Lance, now we know how to find stuff. What else can we do?*

Good question. Next up, we will start creating Bookmarks.

# Tutorial VI

## Bookmarking

Once you have used a control statement such as “if”, and you identify a file of interest, you probably want to do something with that file, like read text from it or bookmark it. When completed, this EnScript will:

- ▶ Create a Bookmark Folder
- ▶ Recurse all evidence and identify a specific file
- ▶ Bookmark the logical file contents.

### Scenario

Let’s assume we want to locate and Bookmark data in an evidence file that contains boot up information. One way to accomplish this is to:

1. Find the “boot.ini” file using techniques we have already used,
2. Bookmark the logical file data.

To bookmark an object, you should first create a **folder** to hold your bookmarks. You could just bookmark your file in the root of the bookmark tree, but that would be messy and disorganized.

Here is the code to get us started:

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         BookmarkFolderClass folder;
4         folder=new BookmarkFolderClass(c.BookmarkRoot(),"My Bookmark Folder");
5
6         SystemClass::ClearConsole(1); // Clear the Console
7         forall (EntryClass entry in c.EntryRoot()) {
8             if(entry.Name()=="boot.ini") {
9                 Console.WriteLine (entry.Name()); //Writes to the Console tab
10                Console.WriteLine (entry.LogicalSize());
11            }
12        }
13        Console.WriteLine ("EnScript finished");
14    } // Execution stops here
15 }
```

Line 3 creates a variable named “folder” of the **BookmarkFolderClass** type. Then, line 4 instantiates (creates) the object named “folder” in the root of the bookmark tree, naming it “**My Bookmark Folder**”. Get it entered and hit **Run**. Check for a Bookmark folder. “**My Bookmark Folder**” should be there, but it’s empty.

The next step is to **Bookmark** any files or folders that you want with the following code:

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         BookmarkFolderClass folder;
4         folder=new BookmarkFolderClass(c.BookmarkRoot(),"My Bookmark Folder");
5
6         SystemClass::ClearConsole(1); // Clear the Console
7         forall (EntryClass entry in c.EntryRoot()) {
8             if(entry.Name()=="boot.ini") {
9                 Console.WriteLine (entry.Name()); //Writes to the Console tab
10                Console.WriteLine (entry.LogicalSize());
11                folder.AddBookmark(entry,0,entry.LogicalSize(),entry.Name(),
12                BookmarkClass::SHOWREPORT, BookmarkClass::LOWASCII);
13            }
14        }
15        Console.WriteLine ("EnScript finished");
16    } // Execution stops here
17 }
```

*NOTE: Line 11 and 12 represent one statement. Long statements can be broken down into multiple lines. Notice the semicolon at the end of line 12.*

Now we need to break everything down.

```
folder.AddBookmark(entry,0,entry.LogicalSize(),entry.Name(),BookmarkClass::SHOWREPORT,BookmarkClass::LOWASCII);
1 2 3 4 5 6
```

The segment "folder.AddBookmark" says add a bookmark into the "folder" object.

The second segment are the inputs that the function expects to tell EnCase WHAT to bookmark, WHERE to bookmark, and how to name the bookmark, as well as how to display it in the bookmark tab.

1. The first input (entry) is the EntryClass object that you want to bookmark.
2. The second input is where you want to begin the bookmark, in this example, from the beginning of the file – offset (0).
3. The third is the ending bookmark offset. In this example, to the end of the logical file.
4. The fourth input is the Name of the bookmark as you want it to appear in the bookmark tab, in this example, the Name of the file/folder object
5. The fifth input is the options, such as show in the report, show as a picture, etc. In this example, it will be shown in the Report View of the bookmarks
6. The sixth input is the viewtype. This tells EnCase how to display the data you have bookmarked. Should EnCase show it in hex, ASCII, a timestamp or HTML page. In this example display low ASCII characters (<128).

After running this EnScript, you should have a Bookmark folder named **“My Bookmark Folder”** in the root of the bookmark tab. File(s) that matched the criterion are bookmarked in that folder.

### Sorta Related Stuff

We mentioned earlier that working with “strings” is a big part of the EnScripting world. The next lesson includes some String variables and string handling. Now is as good a time as any to get a little more acquainted. Modify your code to add the following:

```
1 class MainClass {
2     void Main(CaseClass c) { // Execution starts here
3         BookmarkFolderClass folder;
4         folder=new BookmarkFolderClass(c.BookmarkRoot(),"My Bookmark Folder");
5         SystemClass::ClearConsole(1); // Clear the Console
6         forall (EntryClass entry in c.EntryRoot()) {
7             if(entry.Name()=="boot.ini") {
8                 Console.WriteLine (entry.Name()); //Writes to the Console tab
9                 Console.WriteLine (entry.LogicalSize());
10                folder.AddBookmark(entry,0,entry.LogicalSize(),entry.Name(),
11                BookmarkClass::SHOWREPORT, BookmarkClass::LOWASCII);
12            }
13        }
14    }
15    String myMessage= "EnScript Finished";
16    Console.WriteLine(myMessage);
17 } // Execution stops here
18 }
```

What happened? We created a variable (of String type) to store information. We stored the string “EnScript Finished” in the variable. We passed the myMessage variable name to WriteLine.

*So, Lance, why is this important now?* Well, its not. But, it will be a couple lessons down the road. And, who knows, it might impress the Rookie.

*BTW: The next time you attend your local EnScript User Group gathering, start throwing around terms like Boolean, recurse, iteration, permutations, concatenate, string comparison, etc. and you are a shoe-in to win the **“Wanna See My EnScript”** pocket protector door prize. Good Luck!*

In Tutorial VII, we will learn how to open and read data from a file (file I/O-Input/Output), expand on bookmarking, and get a little loopy.

# EnScript Tutorial VII

## I/O (Input/Output) & Do loops

In the last tutorial, we learned how to recurse through all the evidence in EnCase, select a specific object (file) based on its name matching some specific criteria, and Bookmark the file contents.

In this tutorial, we will:

- ▶ Recurse through all the evidence to find a file named "boot.ini".
- ▶ Open "boot.ini" so we can read specific data from it.
- ▶ Use a do loop to read in one line at a time
- ▶ Locate and Bookmark a small portion of the file in the same manner as an examiner may sweep certain text in a file and bookmark only that data.

### Scenario

Let's assume we want to Bookmark the line of data from an evidence file that specifies the default boot location if the user does not specify otherwise at boot time. One way to accomplish this is to:

1. Find the "boot.ini" file using techniques we have already used,
2. Add some code to read each line of the "boot.ini" file,
3. Evaluate the line of text to see if it is the one we want,
4. Bookmark just the data of interest (like a sweeping Bookmark)

If we only want to read a specific line in the boot.ini file and we know where that line is, we could "**open**" the boot.ini file and move the pointer to where we want to read the data.

For example, if we knew that the data we wanted to read started at offset 100, we could use the following function to move the pointer: **file.Seek(100)**. This would move the pointer to offset 100 in the file and if we begin reading data, it would start at offset 100.

But, what if we don't know the offset or it changes dynamically based on the computer system configuration? We could read through the file and then make a decision based on what we read.

In this example, we are opening the "boot.ini" file, which is part of the boot process on a Windows system. The boot.ini file I am using in this example looks like the following:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows
XP Professional" /fastdetect /NoExecute=OptIn
```

We want to determine if the computer it is capable of multi-booting different operating systems. This is the line we are looking for:

```
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
```

The information on this line varies but, in this case, there is some static text “**default=**” in the line that can be used to help us identify the data we want.

Here is the starting code:

```
1 Class MainClass {
2     void Main(CaseClass c) {
3         SystemClass::ClearConsole();
4         forall (EntryClass entry in c.EntryRoot()){
5             if (entry.Name() == "boot.ini"){
6                 Console.WriteLine(entry.FullPath());
7                 EntryFileClass file;
8                 file = new EntryFileClass();
9                 file.Open(entry);
10                file.SetCodePage(CodePageClass::ANSI);
11                String text;
12                do {
13                    file.ReadString(text, -1, "\\x0d\\x0a");
14                    if (text.Contains("default=")){
15                        Console.WriteLine(text);
16                        break;
17                    }
18                }while (file.Peek() != FileClass::EOF);
19            }
20        }
21    }
22 }
```

Let's break it down, line by line.

1. CaseClass
2. Main function
3. Clears the Console
4. Causes the EnScript to recurse through each object and temporarily assigns each object to the variable named “entry”.
5. Compares the name of each object to see if the name matches exactly “boot.ini”.  
If true, perform the following steps

6. Writes the full path to the Console: "Case 1\C\boot.ini"
7. Creates a variable named "**file**" of the EntryFileClass class type to hold a pointer to the file so we can then take action and perform additional operations.  
To do this we can use the following line of code: EntryFileClass file;
8. This line initializes the variable so it can be used. The open and closed parenthesis immediately after the variable name initializes the variable and prepares it to be used. Instead of two lines, you may also see this version EntryFileClass file(); — Either way works, the second example just uses one line of code as opposed to two.
9. Once we have created the proper variable type and initialized it, we use the "Open()" function of the EntryFileClass to open the current entry which is "boot.ini" After the file is opened, we can move a pointer around and read data.

This function accepts two options:

The first is an EntryClass object that is a pointer to the file you want to open. The file.Open() function needs to know what EntryFileClass object you wish to open. In this example, since we are using a conditional if statement, the EnScript will open the "entry" object as long as its name property equals "boot.ini".

The second parameter specifies if you want to open the file with some options, such as include SLACK space, don't treat an erased file as consecutive clusters, or if you want to write to the file. *This brings up a good point that needs clarification. EnCase does not allow you to alter evidence in any way. You cannot write to a file in the evidence file.* The WRITE option is used if you want to open (create) a file on your forensic machine and then write data to it.

10. Once you open the file, you have to tell EnCase how you want to read data from that file, for example as UNICODE, ANSI, UTF7, etc. To do this, we use the following function, which is a member of the FileClass.  
file.SetCodePage(CodePageClass::ANSI);

This tells EnCase we want to read the data as simple ANSI text, one byte at a time, as opposed to UNICODE, which will read two bytes at a time.

If you look at the EnScript Type tab in EnCase and examine the EntryFileClass type, you will notice at the very top it states "inherits:FileClass". This means that this class inherits all the functions and properties that the FileClass has. The FileClass has a member function named SetCodePage. Since EntryFileClass objects inherit functions from the FileClass, you can call this function on an EntryFileClass object.

11. This line creates a variable named "text" to store a "String" of text. We will be reading from the "boot.ini" file, one line at a time, and we need a place to temporarily store the text.

12. This line starts a “do loop” that will allow us to loop through each line in the “boot.ini” file so we can analyze the text.
13. We already have the file open and ready to be read. We can begin reading data. We have two options here: read one byte at a time or read a string of data. In this case, we will read each line and then decide if it is the line we want. The following code can be used to read the line: `file.ReadString(text, -1, “\x0d\x0a”)`. This function reads a string of data until a carriage return (0x0d in hex) and line feed (0x0a in hex) are encountered and places the string into the “text” variable.

Note: When a file is opened for reading, a pointer is created and placed at the very beginning of the file at offset zero. As you read data, the pointer moves along through the file so that if you read data again, the pointer knows where to continue from and it won’t read the same data again, unless you explicitly instruct it to do so.

14. If the data in the “text” variable contains “default=” then this is the line we are looking for
15. Print the entire line we just found
16. Break out of the do loop early since we just found the line we were looking for
17. End curly brace for the **if** conditional statement. If the line we just read doesn’t contain “default=”, the code jumps back up to the do line and reads the next line.  
Note: When a file is opened for reading, a pointer is created and placed at the very beginning of the file at offset zero. As you read data, the pointer moves along through the file so that if you read data again, the pointer knows where to continue from and it won’t read the same data again, unless you explicitly instruct it to do so.
18. End of the do loop (*we only get here if we do not find the information we are searching for*).

Here is a summary of key concepts

1	<code>class MainClass {</code>	
2	<code>void Main(CaseClass c) {</code>	
3	<code>SystemClass::ClearConsole();</code>	
4	<code>forall (EntryClass entry in c.EntryRoot()){</code>	
5	<code>if (entry.Name() == "boot.ini"){</code>	
6	<code>Console.WriteLine(entry.FullPath());</code>	
7	<code>EntryFileClass file;</code>	Create EntryFileClass variable
8	<code>file = new EntryFileClass();</code>	Initialize the variable
9	<code>file.Open(entry);</code>	Open the file currently pointed to by the entry variable
10	<code>file.SetCodePage(CodePageClass::ANSI);</code>	Set the codepage to ANSI
11	<code>String text;</code>	Create a string variable to hold a line of text as we read the file, line by line
12	<code>do {</code>	Enter a DO loop and continue until the condition is met on the WHILE line below
13	<code>file.ReadString(text, -1, "\x0d\x0a");</code>	Read a line of text until a carriage return and line feed character is encountered
14	<code>if (text.Contains("default=")){</code>	If the line we just read contains "default=" then its the line we want
15	<code>Console.WriteLine(text);</code>	If the line contains the "default=" text then print the entire line to the Console
16	<code>break;</code>	Break out of the loop since we found what we wanted and there is no need to continue reading
17	<code>}</code>	
18	<code>} while (file.Peek() != FileClass::EOF);</code>	Exit the loop when we reach the end of the file (we only get here if the data we are looking for is never found).
19	<code>}</code>	
20	<code>}</code>	
21	<code>}</code>	
22	<code>}</code>	

If we run this code, we get the following output in the **Console**:

```
Case 1\C\boot.ini
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
```

Wow. EntryFileClass objects, opening files, reading string contents from files, string comparisons, Do loops, - a lot to cover.

Next up, learn how to expand on the techniques we learned in this in this lesson and create a "sweeping" bookmark. And some other stuff.

# Tutorial IX

## Bookmarking Revisited

In Tutorial VI, we located a particular file and bookmarked the entire file. In this example we only want to bookmark specific information from the “**boot.ini**” file, the row containing “default=”.

However, we cannot simply bookmark the string in the “**text**” variable. Why? Because the “**text**” variable is just a temporary container that we use while the EnScript runs. When the EnScript ends, the text variable is destroyed. What we really want to do is locate the **exact** file offsets in the **original** evidence and use the **Bookmark** function to mark just the **specific** string we need. Getting there will require a little EnScript arithmetics.

Here is the starting code that combines Bookmarking from Tutorial VI and file I/O from Tutorial VIII. Take a look to get an idea of what’s going on.

*(Don’t try to enter the code on line 18. We need to talk first)*

```
1  class MainClass {
2      void Main(CaseClass c) {
3          BookmarkFolderClass folder;
4          folder=new BookmarkFolderClass(c.BookmarkRoot(), ("Boot Environment"));
5          SystemClass::ClearConsole();
6          forall (EntryClass entry in c.EntryRoot()){
7              if (entry.Name() == "boot.ini"){
8                  Console.WriteLine(entry.FullPath());
9                  EntryFileClass file;
10                 file = new EntryFileClass();
11                 file.Open(entry);
12                 file.SetCodePage(CodePageClass::ANSI);
13                 String text;
14                 do {
15                     file.ReadString(text, -1, "\x0d\x0a");
16                     if (text.Contains("default=")){
17                         Console.WriteLine(text);
18                     folder.AddBookmark(entry,file.GetPos()-text.GetLength(), text.GetLength(),entry.Name(),BookmarkClass::SHOWREPORT,BookmarkClass::LOWASCII);
19                     break;
20                 }
21                 } while (file.Peek() != FileClass::EOF);
22             }
23         }
24     }
25 }
```

I know. Line 18 is kinda hard to read. But, it brings up a good learning point. EnScripts execute one statement (generally one line) at a time. Usually, this isn't a problem, and in the EnScript code tab, the statement can extend off the screen. But a **really** long statement may require you to scroll around a lot to see the code. So, what do we do?

Remember, statements generally end with a semi-colon (;). We use spaces, indents, and blank lines to make code easier to manage. We can also "split" long statements into multiple rows. So, look at this version:

```
1 class MainClass {
2     void Main(CaseClass c) {
3         BookmarkFolderClass folder;
4         folder = new BookmarkFolderClass(c.BookmarkRoot(), "Boot Environment");
5         SystemClass::ClearConsole();
6         forall (EntryClass entry in c.EntryRoot()){
7             if (entry.Name() == "boot.ini"){
8                 Console.WriteLine(entry.FullPath());
9                 EntryFileClass file;
10                file = new EntryFileClass();
11                file.Open(entry);
12                file.SetCodePage(CodePageClass::ANSI);
13                String text;
14                do {
15                    file.ReadString(text, -1, "\\x0d\\x0a");
16                    if (text.Contains("default=")){
17                        Console.WriteLine(text);
18                        folder.AddBookmark
19                            (entry,
20                            file.GetPos()- text.GetLength(),
21                            text.GetLength(),
22                            entry.Name(),
23                            BookmarkClass::SHOWREPORT,
24                            BookmarkClass::LOWASCII);
25                    break;
26                }
27            } while (file.Peek() != FileClass::EOF);
28        }
29    }
30 }
31 }
```

Better? Good. Now Rows 18-24 represent one statement (the semi-colon (;) is at the end of line 24). Now we will break everything down and see how it works.

Here is a color coded view:

```
folder.AddBookmark
(
  entry,
  file(GetPos() - text.GetLength(),
  text.GetLength(),
  entry.Name(),
  BookmarkClass::SHOWREPORT,
  BookmarkClass::LOWASCII);
```

Let's break it down

```
folder.AddBookmark(entry, file(GetPos()-text.GetLength(), text.GetLength(), entry, Name(), BookmarkClass::SHOWREPORT, BookmarkClass::LOWASCII);
1                2                3                4                5                6
```

To add a bookmark we use: **folder.AddBookmark.**

We are adding a Bookmark into the **“folder”** object.

The second segment has six (6) inputs that the function expects in order to tell EnCase WHAT to bookmark, WHERE to bookmark and how to name the bookmark, as well as how to display it in the bookmark tab. The inputs are:

1. The first input **“entry”** is the EntryClass object that you want to bookmark.
2. The second input is where you want to begin the Bookmark, in this example, from the beginning of the line **“file(GetPos()-text.GetLength())”**.
3. The third is the ending Bookmark offset. In this example, **“text.GetLength()”**  
*(We will come back and revisit 2 and 3 a little later. For now, just enter the code and think about why we use this approach).*
4. The fourth input is the **Name** of the bookmark as you want it to appear in the bookmark tab, in this example, the **Name** of the file/folder object
5. The fifth input is the options, such as **show in the report**, show as a picture, etc. In this example, it will be shown in the **report view** of the Bookmarks
6. The sixth input is the **viewtype**. This tells EnCase how to display the data you have bookmarked. Should EnCase show it in hex, ASCII, a timestamp or HTML page. In this example display **lowASCII** characters (<128).

### Steps 2 and 3

In Tutorial VI, we Bookmarked the entire logical (from offset zero (0) to `entry.LogicalSize()`). Now, we only want to Bookmark only the line from the “**boot.ini**” that contains the “**default=**” text string.

Here’s what’s happening. After we locate the “**boot.ini**” file, we create a variable “**text**” and read in the “**boot.ini**” file, one line at a time. We loop through each row and look until we find the static text “**default=**”. This was the line we want. We printed this line of text to the Console. Still with me? Good.

Now we need to **Bookmark** the data in the **original** evidence. The **Bookmark** function requires a number of parameters (what, starting point, ending point, name of bookmark, where to bookmark, how to display).

So, using information available to us, we have to specify exactly where (in the original evidence file) the bookmark will begin and end. How?

Look at this part of the function:

```
folder.AddBookmark  
(entry,  
file(GetPos()-text.GetLength()),  
text.GetLength())
```

The starting point of the bookmark is determined using `file(GetPos()-text.GetLength())`.

Think back. As we read through a file, a file pointer keeps up with our position. The file pointer is one byte past the end of the “**default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS**” So, we can use `file.GetPos()` to locate the pointer. How do we find the beginning of the line? Well, if `file.GetPos()` tells us the current location of the file pointer, can’t we simply subtract the length of the string `text.GetLength()` to determine the first character we want to bookmark? Well, that’s exactly what we did.

Now we need to find the end. If we know the beginning, can’t we just add the length of our entry string `text.GetLength()` ? Sounds good to me.

Run the EnScript. Switch to the Console tab. You should see something like this:

```
Case 1\0\D\boot.ini  
default=multi(0)disk(0)rdisk(0)partition(2)\WINDOWS
```

The first line is the result of finding the “boot.ini” file (**Console.WriteLine** statement on line 8). The second line is the string of text we found by testing to see **if** the text “default=” is contained in the string.

Now switch to the Bookmarks Tab. You should have a new Bookmark folder named “**Boot Environment**” in the root of the **Bookmarks** tab that contains a sweeping

bookmark of the line in the “boot.ini” file that contains the “default=” string. Switch the lower left pane to the **Text** tab.

It should look kinda like this:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP
Professional" /fastdetect /NoExecute=OptIn
```

Cool! Now you are really EnScriptin!

*What? What do you mean I have a problem? You say its a “little “off”! This is a free tutorial. Do I have to tell you everything? What’s next- free coffee and donuts?*

*Okay, I’ll help you out one last time. Then you are on your own.*

By now, you have probably made at least a few errors that kept your EnScript from running successfully. Once you tracked down the little bugger, everything ran as intended. I assume from the silence that you agree.

In this example, the EnScript ran completely, but didn’t do quite what we expected. **Why? Maybe I am working so hard on the tutorial that my hard drive experienced a thermal runaway event, over-revved the platter, and I ended up with a slung bit. Ever think of That? Could Happen!**

Then again, maybe not. Paraphrasing the famous movie line, “*What we have here is a failure to communicate — accurately*”. We have a logic error, meaning somewhere along the way, we made a miscalculation that threw us a bit off. Or, is that a byte off?

How come? Think back. As we read through a file, a file pointer moves (so that we can go back, pick up where we left off, and read the next string). When we read in the line – “default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS”, the file pointer (in the original evidence file “boot.ini”) is actually located one byte past the last character we read. So, the pointer is one character off from where we want to be.

*Well, you ask, “How do we fix it”? WE nothing! YOU fix it. Oh, sure, we offer free fill-ups on coffee, and, while I’m pouring, I guess I might as well tell you how to do this so you don’t get too much of your jelly donut on the keyboard.*

If we know the position of the pointer (in the original file) and subtract one (1) then we should be on the last character in the file we want to bookmark. Modify your bookmark function to this:

```
folder.AddBookmark
(
entry,
(file. GetPos() -1) - text.GetLength(),
text.GetLength(),
```

Notice the extra set of parentheses. They aren’t **necessary** but help clarify what is happening.

Then, by subtracting the **length** of the string in the “**text**” variable, we can accurately calculate the location of the first character we want to bookmark.

```
(entry,  
(file. GetPos()-1) - text.GetLength(),  
text.GetLength())
```

The next parameter tells the EnScript how long the bookmark sweep will be.

```
(entry,  
(file. GetPos()-1) - text.GetLength(),  
text.GetLength())
```

Then you need all that other stuff to finish out the bookmark function.

Man! Now I need a donut.

If you haven't already done so, add the changes and Run the code.

Unless something **else** goes wrong, you should get the corrected, and expected, results.

```
[boot loader]  
timeout=30  
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS  
operating systems]  
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP  
Professional" /fastdetect /NoExecute=OptIn
```

Nice work.

## Big Finish

Well kids, that concludes this episode of the tutorial.

Next up, we will learn to open a file and write out information, and, uh, whatever else comes to mind.

If you have an idea for a good topic to add, drop me a line.

Thanks for playing along.

# Tutorial Appendix

## Filters

A **Filter** is a special EnScript. It is the same language. Like Conditions, filters serve a specific purpose, **to filter what you see**. The original concept was to filter files/folders based on some type of criteria; i.e. file extension, size, name, whatever. Only, with filters, you can customize the code to meet your specific needs.

Think of a Filter as a “special” EnScript. EnCase actually does some background processing for you with a filter by automatically recursing all the evidence. Recursion means it looks at the first entry in your evidence, evaluates it to see if it meets the condition(s) you specified and then asks, “Do you want to see this entry or do you want to hide it from view”?

The code in a filter can do just about anything a raw EnScript can do, but it must answer one important question, “Do you want to see files/folders (called entries) that match your criteria”? The line **return true** is responsible for answering that question. If you do not answer that question in your code, **your filter will not run**.

Here is the minimum code required for a filter:

<pre>1 class MainClass { 2   bool Main(EntryClass entry) { 3     return true; 4   } 5 }</pre>	The parameter passed to the <b>Main</b> function is <b>EntryClass entry</b>
---	---

There are two major differences between this Filter code and the EnScript code presented above.

1. Remember, a filter’s job is to show or hide each **entry** (like files with .jpg extensions). So, we need to pass the (**EntryClass entry**) parameter.
2. The second difference between EnScripts and Filters is the one added line that states, “**return true**”.

EnCase actually does some background processing for you with a filter by automatically recursing all the evidence. Recursion means it looks at the first entry in your evidence, evaluates it to see if it meets the condition(s) you specified and then asks, “Do you want to see this entry or do you want to hide it from view”?

It reads the above code that says **return true**, meaning **show** me this entry. EnCase then gets the next entry and repeats the process, until it goes through every entry in your evidence. If you have 20,000 entries in a piece of evidence, then this code will execute 20,000 times.

If you changed the line to say, **return false**, EnCase will evaluate each entry to see if it meets the condition(s) you specified and **hide** the entry from view until you remove the filter.