EnScript Changes From EnCase V6 to EnCase V7

General Note

EnCase V6 keeps the majority of data in memory. This facilitates fast access to the evidence items in a case but is not conducive to handling large data-sets. In addition to that, records and entries had to be handled separately.

EnCase V7 behaves in a similar way to a database in that working through multiple evidence items is accomplished using an iterator. This makes for more stable processing and allows the EnScript programmer to handle both entries and records in a more streamlined way. It is possible, for instance, to iterate through all of the evidence items in a case (entries and e-mail attachments, for instance) quickly identifying those items that are pictures or documents.

It's very common for EnScript programmers to want to migrate their EnCase V6 workflows to EnCase V7. In doing so the EnScript programmer should consider the fact that the EnCase V7 evidence processor was designed so as to reduce the amount of additional steps that had to be undertaken (hash and signature analysis, thumbnail creation, Registry pre-processing, etc.) before a review of the evidence in the case could begin. Taking this into account, some tasks might be better performed using a custom evidence processor module, an example of which is to be found in the following folder –

    <EnCase Program Folder>\EnScript\EvidenceProcessor

The example is called **DemoModule.EnScript**. It can be activated by un-commenting the first line in **ModuleList.EnScript**, which is in the same folder.

Iterating EntryClass objects in a case

EntryClass now inherits from **ItemClass**. **ItemClass** represents other classes such as **BookmarkClass** and **RecordClass**.

There is a new method of iterating items in a case, and it utilizes **ItemIteratorClass**. This allows a script to iterate all **ItemClass** objects that are records or entries, or items in a result-set, in one pass.

V6

```
forall (EntryClass entry in c.EntryRoot()) {
  Console.WriteLine(entry.Name());
}
```

V7

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

ItemIteratorClass::NOPROXY is an enumerator specifying that EnCase should not return a proxy object, i.e. one that has dynamically calculated properties such as hash and file-signature. This will save some time during the iteration.

When using an iterator to iterate tagged items or those in a result-set, the list of tags or the result-set name is stored in the **ItemIteratorClass::Name()** property, which allows the iterator to be re-opened without having to re-prompt the user for the same information.

Iterating through items currently in view (using one of the **ItemIteratorClass::CURRENTVIEW** iteration modes) is generally much quicker because it processes those items that are already loaded into memory.

Objects returned via an iterator only have life during their own iteration; they cannot be stored in an array or linked-list like they could in EnCase V6. Attempting to do so will, at best, generate an internal error.

One way around this is to obtain and store an **ItemMonikerClass** object (a collection of three GUIDs) for each item and then use an instance of **ItemCacheClass** (see below) to resolve the each moniker back to an item. This is generally quite slow so it's usually better to process each item at the time it's returned by the iterator.

Accessing individual devices

EnCase v7 has added a layer between the DeviceClass object and the CaseClass object. The interface is EvidenceClass and it represents the properties of the evidence file that contains a device. The device is still obtainable.

V6

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

V7

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

Iterating EntryClass objects in a device

There is another method in ItemIteratorClass that accepts a DeviceClass as a parameter, and then will present each of the entries contained in the device in the same way as it does when passing the CaseClass object mention previously.

V6

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

V7

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

Currently Highlighted Entry

EnCase has allowed for a script to get an object representing the entry that the examiner currently has highlighted. In V7, the capability is expanded to getting the current **ItemClass** object, which again could be an **EntryClass**, **BookmarkClass**, **RecordClass**, etc.

V6

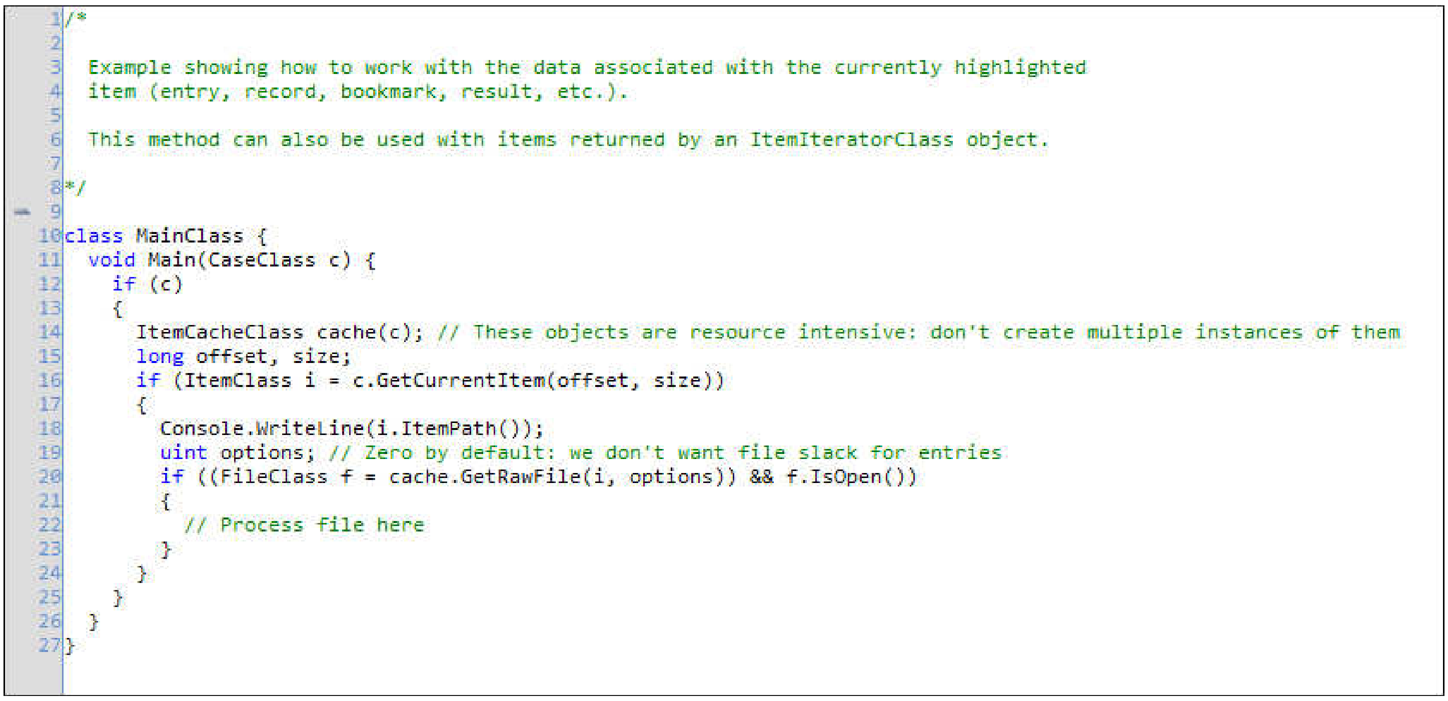
```
if (EntryClass entry = c.GetEntry(offset, size)) {
  Console.WriteLine(entry.Name());
}
```

V7

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

Working With File Data

A consistent way of working with the file-data associated with both records and entries is to use **ItemCacheClass** –



```
/*

  Example showing how to work with the data associated with the currently highlighted
  item (entry, record, bookmark, result, etc.).

  This method can also be used with items returned by an ItemIteratorClass object.

*/

class MainClass {
  void Main(CaseClass c) {
    if (c)
    {
      ItemCacheClass cache(c); // These objects are resource intensive: don't create multiple instances of them
      long offset, size;
      if (ItemClass i = c.GetCurrentItem(offset, size))
      {
        Console.WriteLine(i.ItemPath());
        uint options; // Zero by default: we don't want file slack for entries
        if ((FileClass f = cache.GetRawFile(i, options)) && f.IsOpen())
        {
          // Process file here
        }
      }
    }
  }
}
```

Bookmark Folders

Bookmark folders have changed very little. That said, **BookmarkClass** is now used to create bookmark folders instead of **BookmarkFolderClass**.

V6
```
BookmarkFolderClass folder(c.BookmarkRoot(), “folder name”);
```

V7
```
BookmarkClass folder
              (c.BookmarkRoot(), “folder name”, NodeClass::FOLDER);
```

A class called **BookmarkFolderClass** still exists but it is now used to create a graphical representation of an **ItemClass** folder and any sub-folders.

Bookmarks In General

Bookmarks are now separate objects that need to be declared instead of being created from a method called on the target bookmark folder.

Notes Bookmark

V6
```
folder.AddNote(“some text for a note”, …);
```

V7
```
BookmarkClass note(folder, “name of note”);
note.SetComment(“some text for a note”);
```

Notes can also be linked to ItemClass objects using the CopyItemData() method shown below.

Notable File Bookmark

V6
```
folder.AddBookmark(entry, 0, 0, …);
```

V7
```
BookmarkItemClass bmDecCls(folder, entry.Name());
bmDecCls.CopyItemData(entry);    // Will set the name of the bookmark
                                 // automatically. If you want a
                                 custom // name use the SetName()
                                 method after // using CopyItemData()
```

Highlighted Data Bookmark

V6
```
folder.AddBookmark(entry, 10, 50, …);
```

V7
```
BookmarkDecodeClass bmDecCls(folder, entry.Name());
bmDecCls.CopyItemData(entry);
bmDecCls.SetDataOffset(10);
bmDecCls.SetDataSize(50);
bmDecCls.SetCodingType(BookmarkDecodeClass::HEX);
```

BookmarkTextClass bookmarks are similar to **BookmarkDecodeClass** bookmarks but default to showing file-data as text.

Data Bookmarks

V6
```
folder.AddDatamark(dataRoot);
```

V7
```
BookmarkDataClass bmDatCls(folder, “name of data”);
bmDatCls.SetDataRoot(dataRoot);
```

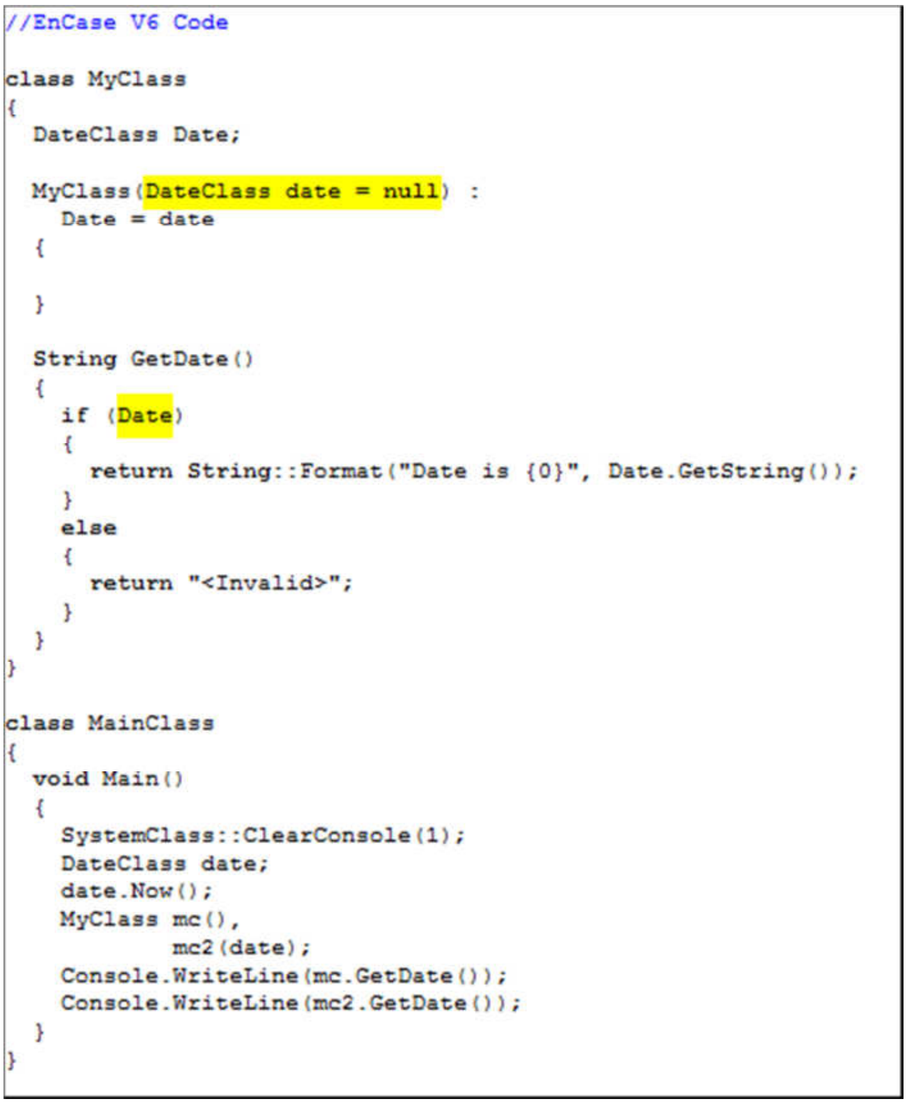
Note that custom data bookmarks (those that are represented by a custom class inheriting from **NodeClass** or **NameListClass**) invariably require the use of a custom **HandlerClass** object in order to display their data correctly.

Data bookmarks can also be linked to **ItemClass** objects using the **CopyItemData()** method mentioned above.

Constructing Class Objects That Have DateClass, GUIDClass or Other Similar Members

DateClass, **GUIDClass** and other similar class objects behave like fundamental types in EnScript.

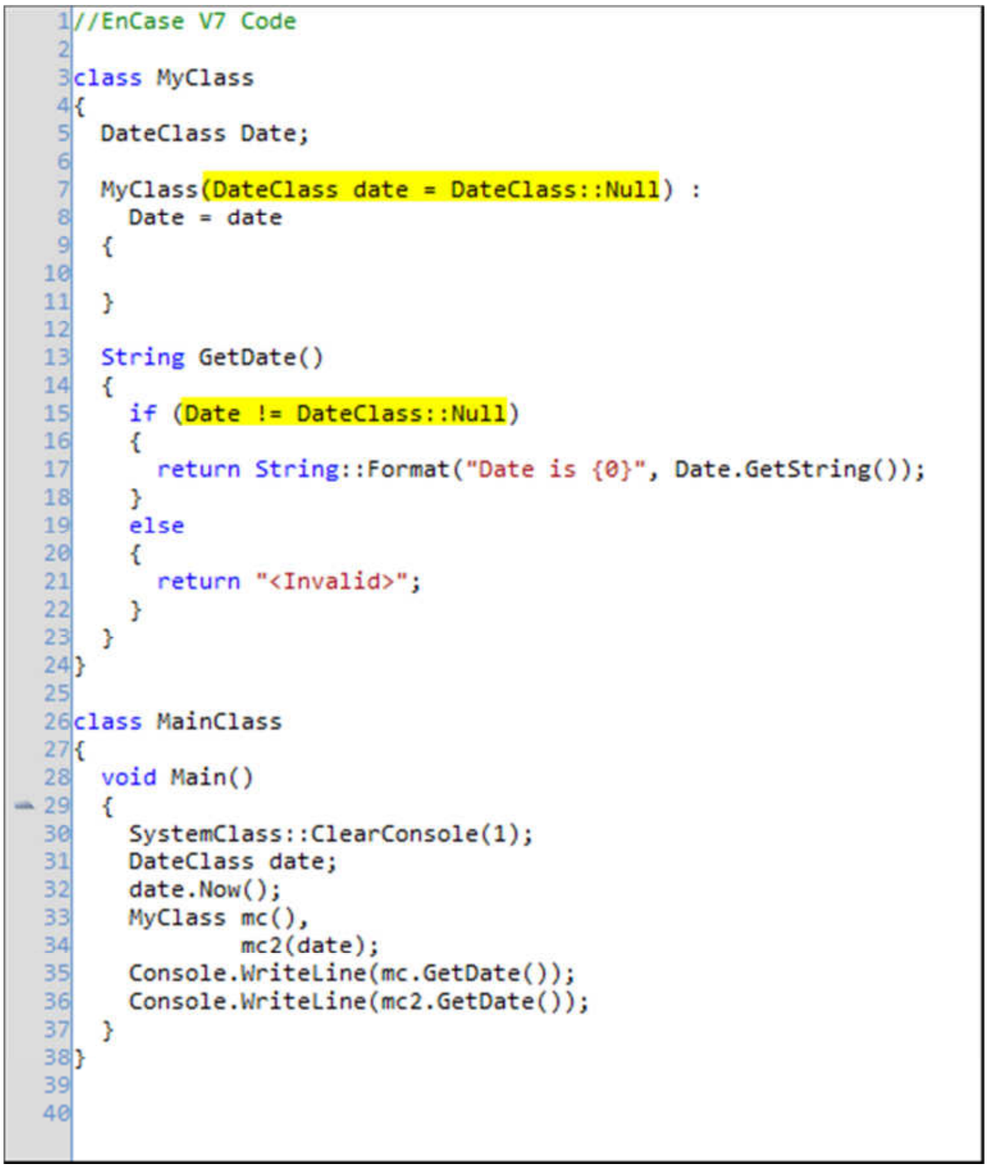
In EnCase V6 it was possible to write the following code. Note the highlighted sections –



```
//EnCase V6 Code

class MyClass
{
  DateClass Date;

  MyClass(DateClass date = null) :
    Date = date
  {

  }

  String GetDate()
  {
    if (Date)
    {
      return String::Format("Date is {0}", Date.GetString());
    }
    else
    {
      return "<Invalid>";
    }
  }
}

class MainClass
{
  void Main()
  {
    SystemClass::ClearConsole(1);
    DateClass date;
    date.Now();
    MyClass mc(),
            mc2(date);
    Console.WriteLine(mc.GetDate());
    Console.WriteLine(mc2.GetDate());
  }
}
```

The **DateClass** object being passed into the constructor can, in EnCase V6, have a default null value.

In EnCase V7 this is not permitted: DateClass::Null must be used instead. The following code demonstrates this. Again, note the highlighted sections –



```
//EnCase V7 Code

class MyClass
{
  DateClass Date;

  MyClass(DateClass date = DateClass::Null) :
    Date = date
  {

  }

  String GetDate()
  {
    if (Date != DateClass::Null)
    {
      return String::Format("Date is {0}", Date.GetString());
    }
    else
    {
      return "<Invalid>";
    }
  }
}

class MainClass
{
  void Main()
  {
    SystemClass::ClearConsole(1);
    DateClass date;
    date.Now();
    MyClass mc(),
            mc2(date);
    Console.WriteLine(mc.GetDate());
    Console.WriteLine(mc2.GetDate());
  }
}
```

It should be noted that the above code will run in both EnCase V6 and EnCase V7.

Originally written by James Habben <james.habben@encase.com>

Last updated 12th March 2014 by Simon Key <simon.key@encase.com>