Yet Again Another Still More Intro to Object-Oriented Classes

As I assume you're aware, a .jpg file includes not just the image, but also metadata called Exif[1] data. This includes obvious stuff such as the image size (both horizontal and vertical resolution), when the picture was taken (both before and after digitization, among others), and many, many other aspects of the data.

Now suppose you were writing an app that wanted access to this metadata. Say, a photos app that you wanted to be able to sort by data taken. Or group by season ("Summer photos 2017"). Or whatever.

Yeah you'd have to read the documentation and figure out exactly what a .jpg looks like internally, and then how to get at that data in C++, C#, whatever. But that's not the point; most people couldn't write a square root routine, but they can use the subroutine. Anyway, you could then do the following in C-ish pseudo-code:

```
struct DateTime WhenTaken;          // I forget what C's equivalent of DateTime is. Maybe just "date"?
char *Comment;

int ImgHandle = fopen("MyFile.jpg");    // See Note 1
void * pWhenTaken = GetJpgProperty(ImgHandle, "DateTimeOriginal");    // See Notes 2 and 3
if (pWhenTaken != nullptr) {    // Make sure the camera recorded this
   WhenTaken = (DateTime)*pWhenTaken;
}

// Now do the same thing to get field called "UserComment"
void * pUserComment = GetJpgProperty(ImgHandle, "UserComment");
if (pUserComment != nullptr) {
   Comment= (char *)pUserComment;    // Let's say it's "Uncle Fred being silly"
}

// To keep things simple, let's assume that we found both properties
printf("%s taken on %d/%d%d", Comment, WhenTaken.Year, WhenTaken.Month, WhenTaken.Day);
fclose(ImgHandle);          // I hope you didn't forget to do this
```

Note 1: A minor thing, but why should you have to bother knowing something about the file system routines (fopen, fclose, fread, file handles, etc) just to access your file's data? Consider that these days, a lot of programmers never read or write local disk files. They get all their data from a database, from web services, etc.

Note 2: A subroutine can't return different types unless it's got a *lot* of overhead under the covers (so to speak!). Which pretty well must, of necessity, proliferate throughout the entire language, slowing things down. Without this extensive (and expensive) runtime support, the only way you

[1] Exchangeable Image File

could access different data types (in the example above, some kind of data structure (or maybe a string) that holds the date and time information) is to return a pointer to the data, and you, the programmer, have to then cope correctly with whatever the pointer is pointing at.

Note 3: You can't expect a compiler, in general, to (a) know about .jpg's (and hey, who's to say that j-p-g doesn't stand for Junior Pyromaniacs Guild), (b) know that "DateTimeOriginal" is a property of a .jpg -- suppose you mistyped that name, and didn't (couldn't!) find the error until runtime!

A Short Digression on C#'s "Property" language feature

You may recall from older versions of Visual Basic that you could make a GUI widget visible or invisible with the .Visible property – e.g. MyTextBox.Visible = False. Or just check its visibility status via If MyTextBox.Visible = True Then …

Very easy to use, but it seems hardly likely that making a widget disappear involved nothing more than clearing a bit in memory. What actually happened is that when the TextBox widget was written, they had to make Visual Basic aware that they had a property called "Visible" that could be read or written by having VB invoke get_Visible / set_Visible[2] subroutines written by the TextBox author.

C# follows in Visual Basic's path in this case. If you wanted to define a Widget class as a type of Window, and with a Visible property, you could write:

```
class Widget : Window {   // " : Window" means that a Widget is a type of Window
// Add whatever fields/methods/whatever here that a Widget needs
public bool Visible {
        get {     // Add code here that figures out whether this Widget is visible or not
                  // and returns a bool
        |
        set { /* Add code here to make this Widget visible or not */ }
}
```

Another Digression – enums

Suppose you needed to pass the day of the week (0 = Sunday, 1 = Monday, etc.) to a method. For the sake of concreteness, let's say you wanted to pass in Sunday. You could pass in:

1. The string "Sunday".
    - This might work. But if you needed the numerical value at some point you'd have to convert the string to the number 0 at runtime.
    - And if you misspelled it as, say, Sunnday, the compiler couldn't help you out and at best you'd only find the error at runtime.
2. The explicit value 0.
    - Again, a typo of passing in 10 would lead to problems.
3. A const: const int Sunday = 0;
    - Much better, but still vulnerable to typos. Suppose you had another (admittedly unlikely) variable named Sonday with the value 112449 (e.g. a birthday).

[2] Or whatever the compiler wanted to call them.

- But you'd still have to define your method as taking an int as a parameter, and it would have to accept any integer you threw at it.

4. An enum.

An enum is in many ways similar to const int Sunday = 0; const int Monday = 1; etc. You could write

```
enum DaysOfWeek {
        Sunday,         // Defaults to 0
        Monday,         // Defaults to 1 more than the previous, in this case 1
        // and so forth
}
```

But here's the important difference between this and the const approach: an enum is *a new data type*! So instead of defining

```
void foo(int DOW) { … },
```

you could write

```
void foo (DaysOfWeek DOW) { … },
```

and invoke it as foo(DaysOfWeek.Sunday);

So now you'd get a compile-time error if you wrote any of:

```
foo("Sunday");
foo("Sunnday");
foo(0);             // foo now takes a data type of DaysOfWeek, not an int
foo(10);
foo(Sunday);        // Assuming const int Sunday = 0;
```

enums are a small language feature, but they let you:

- Make the intent of the parameter as clear as possible to the reader
- Capture at compile-time several types of errors (especially typos)

C# Constructors (ctors)

Arguably the most common error in programming is that of undefined variables. So most object-oriented (OO) languages (C#, C++, Java, etc.) try to address this problem. In C#, if you define a method with the same name as the class it's in, it's considered a constructor. This method is run when you create a new instance of a class via the new keyword. To take a trivial example:

```
class foo {
        public int magic;
        public foo() {
                magic = 42;
        }
}
foo xxx = new foo();                    // Implicitly invokes the ctor
Console.WriteLine(xxx.magic);           // Writes 42
```

Back to Exif

So now let's imagine writing our own Exif class. What might it look like? Probably something like the following:

Note: The point here isn't to show how to extract the Exif data. There are any number of programs on the web that show how to do that. A quick search just now shows https://www.codeproject.com/Articles/27242/ExifTagCollection-An-EXIF-metadata-extraction-libr. But I haven't looked at the code and thus clearly can't vouch for it.

Note: for simplicity's sake I'm only going to talk about reading Exif data from the file using the BinaryReader class. You'd need to use a BinaryWriter to update the file. Again, to concentrate on the OO principles involved, I'm not going go into how you'd use these classes to get at the Exif data. And I'm going to ignore error handling, such as what to do if you've passed in the name of a file that doesn't exist.

```
class Exif {
        public string TheFile;      // "Remember where we parked!" – J. T. Kirk
        private BinaryReader Rdr;

        public Exif(string filename) {      // ctor
                TheFile = filename;
                // Open the Rdr based on the filename
        }

        public DateTime DateTimeOriginal {
                get {
                        DateTime dtOrig = /* code to get this field */;
                        DateTime dtOrig;
                }
        }

        public int ImageWidth {
                get {
                        int width = /* code to get the width */;
                        return width;
                }
                // If we were supporting writing Exif data, it would look like the following
                set { /* Code to update the file */ }
        }

        public int ImageHeight {
                get {
                        int height = /* code to get the height */;
                        return height;
                }
        }

        public string ImageDescription {
                get {
                        string ImgDesc = /* code to get this field */;
                        return ImgDesc;
                }
        }

        // Add as many other properties as you'd like
        // And, of course, any fields and auxiliary methods as needed, most (all?) of which would be private
}
```

So, finally, you could write:

```
Exif ex = new Exif("MyFile.jpg");
Console.WriteLine($"{ex.TheFile} is {ex.ImageWidth}x{ex.ImageHeight} pixels in size – {ex.ImageDescription}");
```

Note the Following

- Other than supplying the file name, the user of the Exif class doesn't need to know the internal format of a .jpg file, nor the file system calls to access the metadata.
    - Indeed, the filename you pass in may even be a URL and you'd have to use different routines to read the file from the Internet.
- The code inside the Exif class is presumably all private. In a multi-programmer environment, another developer can *use* the class, and can probably view the code in an editor, but can't call any of the non-public methods or refer to non-public fields[3]. Which is good; the current Exif spec is at version 2.3 and if version 2.4 is ever released and requires code changes, the other users of Exif internal methods might find their code broken with no indication that something's changed.
- Typos in the property names are now caught at compile-time, not run-time. If you were to type in ex.ImageDescriptoin, (note the mistake in the last 3 letters), your source code wouldn't compile. And that's less likely to happen these days; the Visual Studio editor works closely with the C# compiler even as you're editing your code, so the editor knows after you've typed ex. that the only valid thing you could type next would be (in this case) one of the public property names, and you could just select the one you want from a popup menu. (Microsoft calls this feature of Visual Studio, *Intellisense*.)
- Properties have real data types, not just the non-specific void * "type". The compiler knows that, for example, ImageHeight is an integer. More interestingly, DateTimeOriginal is an instance of the DateTime class[4], and when we try to display it as text via Console.WriteLine, its ToString() method will be called that will produce nice readable text from its internal fields of Year, Month, Day, etc.
- Yeah, I know, I never closed the file. There are a couple of easy ways to ensure that in C#, but going into them would have taken us farther afield than is desirable in this document.
- And finally, as we saw with the C-ish code at the beginning, everything could have been done in C. It's just that more modern languages have features such as classes that make programming easier (e.g. with properties), have more sophisticated data types (a class is a data type – note how DateTime has a method that can format itself), have fewer bugs (e.g. cutting down on undefined variables with constructors), and is safer (e.g. with explicit public vs. private settings on all fields, methods, etc.).

Bottom Line

OK, so if you don't count whatever bit-fiddling you have to do to deconstruct a .jpg file to get at the properties, the Exif class is pretty simple, a constructor and some properties.

It really boils down to how you view the world. If all you want to do is to get the resolution of an image, that's one thing. But people don't usually think in terms of how-do-I-write-a-subroutine-to-do-*whatever*. They normally think in terms of *things* (objects!) that can do things (methods!) and that have certain properties (uh, properties!).

But if you start looking at the problems you want to solve in more human terms, you may find that programming becomes that much easier.

Here's a guideline – Write out a description of your problem in English. In general, the *nouns* become *classes*, the *verbs* become *methods*. Back to jpg's:

[3] And with appropriate file permissions set, can't modify the code.

[4] Strictly speaking, DateTime is a *struct*, not a *class*. But for this document, they're essentially the same.

A jpeg is a particular type of *Image*. You can *load*[5] it into memory and *save* it back. It has a number of properties called *Exif data*.

OK, One Final Example Involving JPEGs

So let's consider one final set of related examples. As we saw, an Exif class is pretty simple. So let's step up a level or two. Again we'll concentrate on the *structure* of our *things*, not the actual code to implement what we want to do. I'll just write an ellipsis (…) to represent whatever code is required.

Consider a .jpg file as a whole. Exif properties is just one aspect of it. There are other things we might want to do with a .jpg file. You can manipulate it by *changing its resolution*, by *reducing its color depth*, *rotating* it, produce special effects by *Solarizing* it, and so on.

```
public class Jpeg {
        // Fields
        public string       Filename;       // Filename should be here, not in the Exif class
        public Exif         Properties;

        // A simplification of the internal format of a .jpg file.
        // See https://en.wikipedia.org/wiki/JPEG#Syntax_and_structure and following
        private byte[]              StartOfImage;           // 0xFFD8
        private Frame               BaseLineDCT;            // Assume these classes are defined somewhere…
        private Frame               ProgressiveDCT;
        private Huffman[]           HuffmanTables;          // Compression data. See
                                                            // https://en.wikipedia.org/wiki/Huffman_coding
        // And so forth
        private byte[]      ImageContents;          // The bitmap itself. Perhaps in the original binary format inside the file,
                                                    // or uncompressed for easier manipulation, etc.

        // Methods
        // Note: For simplicity's sake, I've made all methods return void.
        public Jpeg(string filename) { … }          // ctor. Note that the file name could be something other
                                                    // than a .jpg file (e.g. .bmp, .png, etc) and this routine
                                                    // could convert it to a .jpg.

        public void Save() { … };                   // Perhaps with some properties changed
        public void Save(string newFilename) { … };         // Or you might want to save a copy, perhaps converting
                                                    // this to a new format (e.g. .png). In that case you'd probably want
                                                    // to have an overload of this method that took an enum of the
                                                    // filetype you wanted this converted to.

        // Methods to manipulate the image
        public void ChangeResolution(int newWidth, newHeight) { … }; // Your call – does this (a) throw away pixels,
                                                    // (b) duplicate pixels and make the picture blocky, (c) interpolate
                                                    // new pixels to blend with neighboring ones, (d) something else
        public void ConvertToGrayScale(){ … };
        public void ReduceColorDepth(){ … };        // e.g. from 32 bit color to 16 bit color
        public void Solarize(){ … };                // Make look psychedelic
        public void Rotate(double angle) { … };
        public void Resize(double percentage) { … };        // Make larger or smaller
        public void Recompress(CompressionFactor cf) { … };         // An enum
        public void Shrink(int maxSize) { … };      // Figure out how to do some combination of reducing the color
                                                    // depth, resizing it, etc, to create an output file of no more than
                                                    // maxSize bytes
```

[5] Remember, a constructor is a method.

```
        // And so on
}
```

So Exif becomes just another class that the main Jpeg class uses. You could then write:

```
Jpeg j = new Jpeg("MyFile.jpg");
j.Solarize();
j.Rotate(45);
j.Shrink(32 * 1024);
j.Save();
```

In fact, with a few changes to the methods, they could not just modify the in-memory version of the file[6], but could (at the cost of some additional RAM) return a new in-memory instance of a Jpeg image and you could write:

```
Jpeg j = new Jpeg("MyFile.jpg");
j.Solarize().Rotate(45).Shrink(32 * 1024).Save();
```

Spelling it out, it would be as if you wrote:

```
jTemp1 = j.Solarize();
jTemp2 = jTemp1.Rotate(45);
jTemp3 = jTemp2.Shrink(32 * 1024);
jTemp3.Save();
```

[6] In other words, not a new disk file.