Programming for Artists and Designers

Lecture 2: Introduction to Classes and Packages

Introduction
This lecture aims to introduce you to the basics of classes and packages used in unreal, and how they interact.

Links
This lecture owes its content to: http://wiki.beyondunreal.com/wiki/Package

Classes vs. Packages
It is important to note the differences between classes and packages.

Sometimes when referencing a class, the name of the file is used, and sometimes, it isn't, obscuring the issue.
A class representation, especially when called from another class, such as "Pawn.PlayerReplicationInfo" looks exactly like a variable reference, such as "PlayerReplicationInfo.PlayerName".

In addition, classes have an entirely different hierarchy than the files, and it's for the most part, completely unrelated. We find player pawn classes and mutators in all sorts of packages.

Hey, while we're at it, what exactly is a class file anyway, and where do I find it? Add that all together, and you have a potentially very confusing situation.

What Are Packages?
Many times you will see references to something called a package. Well, what exactly IS a package?
A **package** refers to ANY of the Unreal file formats - such as: .uc, .u, .ut2, .utx, .uax, .umx, .uz2, .uxx, .usx.
The term package can also apply to a combination of these file types.

Generally when coding for unreal we need only to worry about .uc, .u and .int files.
We also use the term package to refer to a collection of .uc files that represent a module or set of additions we have added to the game. The files that make up this module are part of the package. The compiled .u file is also known as the package.

Each of these extensions represents a file containing a different type of Unreal content. .u files contain unrealscript classes, .utx packages contain textures, .uax packages contain sounds, .ut2 packages contain maps, and so on.

The engine does not actually care about the extensions at all - the engine sees all packages equally, but it makes life easier for us when coding.
It would be perfectly valid to create a .utx file which contains nothing but classes (though it *would* be pretty tricky to coax the compiler into doing this). This is mentioned only for completeness, due it being an advanced topic.

At this point, it is best to think of each extension as representing a container file which holds a different type of content. Let's now begin to focus on the type of package we're interested in - a .u package!

Unreal .u files are essentially containers for the **class** scripts. It's much like a zip file, in fact, since you can put any type of class, regardless of what it does, into it. Our raw package, the one that contains the raw .uc files gets interpreted and the classes added to the .u file.

What Are Classes?

We briefly mentioned classes and Object Oriented Programming (OOP) last week. These concepts will be covered in more detail next week.

The whole idea behind UnrealScript is a programming concept called Object Oriented Programming. This means that we're going to create the entire program using individual little pieces, called objects, which define their own behaviours and parameters. Those objects interact with each other inside the realm of the Unreal engine, with scripting to tell them how to interact with each other. It's important to remember, here, that there is no external influence telling these objects how to behave (for the most part). They are fully self-contained objects which control their own behaviour and how they'll react when another object comes into their sphere of influence. That's an extremely basic overview of OOP.

A .u package file may contain many classes. A few examples: GameInfo, Mutator, UTServerAdmin, Inventory. Each of these classes tells the Unreal Engine about an object in the Unreal universe. In UnrealScript, each class that we create becomes an object in the Unreal engine. This means that for every "piece" of a mutator, mod, or whatever, we must write a separate class file for it. For instance, in the case of writing a new weapon, you must write one class (class = object) for the weapon, and another class for the ammo that goes into the weapon. You must write yet another class for the projectiles that the weapon fires, and yet another class for the shell casing (for example) that are produced when you fire a weapon. Another class must be written for the muzzle flash, yet another for the explosion your weapon causes when it hits something, etc. etc. So, you will write several classes for a simple weapon, each class describing a different object that the engine must know about in order for your weapon to be successfully loaded and used in the game. The number of "pieces", or classes you use is, for the most part, entirely up to you (although unreal script does force some coding standards), subject only to the laws of practicality.

For now, remember that .u files contain classes, which each define a single object in the Unreal engine. These classes tell the engine all about how that object interacts with the rest of the objects in the engine, and together, they create a mutator, mod, or what-have-you.

Components of .U Files

When you write a script, you use text, such as

```
class Mutator extends Info;

var PlayerPawn P;

function ModifyPlayer(Pawn Other);

defaultproperties
{
}
```

It would take an extremely long time, however, for the computer to parse these text statements during the game, so you must first compile the script. Compiling is the process of converting all of the text you've written into computer optimized code, commonly referred to as 'binary'. After the compilation process has completed, you should see a new .u package in your System directory, with the same name as the directory you created. If you look at compiled script (a .u package) in a text editor, it will look like gibberish:

```
×£<]$)\²=Pe6
õJ¯'ÁžP×²!À°
L¢bBƒ€mcìkÁ€
mcì[Á€mcìKÁ€
½Ë¹'Aümcî;Á¤
mcî3Á¥mcî3Á¦
mcî3Á§hcxBƒ€
mcî3Á¬$×£<]<
[×£$$)\£%<[=
```

Although you may be able to pick out certain bits of information, particular class headers:

```
// Based on Ammunition
class GetawayAmmo extends Ammunition;
```

Once the script has been compiled into a package this way, you can no longer modify the package. In order to make changes, you must delete the compiled package, make the changes to the original script text, and recompile it.

ClassName/PackageName Notation

Suppose you had written a class called 'MyWeaponClass', and named the folder for all your script 'MyWeaponMod'. Your compiled package name would

be 'MyWeaponMod.u'. There are two distinct hierarchies, or "trees", which can be used to refer to the MyWeaponClass class which now resides inside the MyWeaponMod.u package.
These two trees are commonly referred to the as the Package Tree, and the Class Tree. The Unreal engine uses a dotted notation system to access different branches of these trees, and the notation used for both systems is exactly the same, so this can be a little confusing at first.

Which tree you should use in order to reference a class depends on the situation.

The Package Tree looks very much like the directory structure of your Unreal directory. Within each folder, there are a number of .uc files. Each .uc file contains the script for one class. Each .u package contains one class for each .uc file that was in the respective folder. To reference a class using the package tree, the syntax is:

```
PackageName.ClassName
```

Note that the extension is omitted. Why? As I said above, the extensions are only to help humans keep everything straight - the engine saves and loads all packages exactly the same.

The Class Tree works quite differently. As you probably know by now, the Object class is the base class for all of the Unreal classes. If you take a look at Object.uc, inside the Core\Classes\ folder, you'll see that it is the only class that does not 'extend' or 'expand' another class. In the class tree, this is similar to the root directory of the game. All classes which directly subclass Object would be like the first level of folders in your Unreal installation, with the classes that subclass those classes being like the second level of subfolders, and so on. Referencing a class in the class tree is done directly - no package name is necessary.
In UT2003/4: MutInstaGib is the name of the class that defines the InstaGib mutator, and this class file is located in the "container" XGame.u **package**, which is in the System directory. The .u packages also contains many other classes which work together to define a number of behaviours for the game. In some cases, such as Engine.u, the classes contained by the package do not necessarily relate to each, while in other cases, such as IpDrv.u, all of the classes contained in the package are focused on a particular area of the game. To reference a class using the class tree, you do not specify the package at all:

```
class'ClassName'
```

As mentioned before the term Packages can also refer to the un-interpreted code. In this is case it refers to the base folder or directory, for instance XGame.u refers to the interpreted package, and the XGame folder is the un-interpreted package.

How Unreal Engine Maps Package Names To Class Names

Here we'll explain a little more in depth the role of the package as opposed to the role of the class to the Unreal engine.

The way it works is very similar to the way a webserver works. Let's say I'm hosting a website on my computer - www.scriptnewbie.com. On my computer, I have designated a particular folder, "C:\WebServer\", as the default directory that will contain all my webpages, such as home.htm, links.htm, etc...
Let's say I have some pictures I on my website, and these pictures are located on my hard drive in the

```
C:\Pictures
```

directory. However, when I set up my webserver, I set up a "virtual" directory that points to that C:\Pictures directory. The details of this are unimportant, but here's my point:
When you access my website, you will do so by typing

```
http://www.scriptnewbie.com
```

into your browser. You will then actually be accessing my "C:\WebServer" directory, but to you, it appears as though you are accessing the root directory of www.scriptnewbie.com . If you want to access my pictures, you would then type into your browser http://www.scriptnewbie.com/pics/ to access my C:\Pictures directory, but again, to you, it appears as though you're accessing the "pics" subdirectory of the website.
In similar fashion, the **packages** of the engine are like the actual directories on my hard drive: C:\WebServer, and C:\Pictures, whereas the **classes** are like the virtual folders you type into your browser. You could not type into your browser:

```
C:\www.scriptnewbie.com\Pics
```

and expect to arrive at my website nor would you type:

```
C:\Website\
```

on your computer and expect to arrive at that directory on my computer.
My content's physical location is C:\WebServer, but my WebServer software mediates between you (the user) and the physical location. Same thing goes for UT2K3. In both cases, content may be in any arbitrary physical location. It does not matter to the web visitor, because the server software handles the mapping.

When you are writing the script text, importing textures, referencing sounds, etc., you will be working with the file hierarchy in detail. It's important to know which classes reside in which .u files, which textures reside in which .utx file, which sounds reside in which .uax file, etc. This is like looking at my webserver from my computer. It's important that I know which actual directory

my pictures are located in, so that I can place the resources in the correct place, and configure my webserver to pull the resources from the correct location when you attempt to access the /pics/ directory.

In Unreal, as the script is being executed, it's unimportant (for the most part**) which particular .u file a class is in, because we're now working with a completely different hierarchy. We are now inside the "Engine" of Unreal, and will be accessing those textures, classes, sounds, etc. from the standpoint of an object (every aspect of Unreal is an object) inside the engine, much like the visitor to my website accesses my pictures using the "/pics/" directory. The actual .u file an object belongs to doesn't really matter inside the game, because I'm using the Unreal server to access the information I need. The actual folder that the content resides in on my computer doesn't really matter to my website visitor, because he is using my webserver to access that information.

** Protocol and intention is that no two objects can have the same name, such as Botpack.BlueFlag and MyMod.BlueFlag. However, there is nothing to prevent you from doing this, as it is syntactically correct. UnrealScript is designed to be much more pliant than other programming languages with regards to duplicate files and null references (this is intentional). As a result, you must be very careful if you choose to give a class the same name as a class in another package. It will be allowed by the compiler (unless it is in the same package) but you may get very unreliable results while playing.