

Chapter 2: Class Structure and Basics

Table of Contents

- Introducing Unrealscript Classes..... 2
 - Anatomy of a Class 2
 - Header/Class Declaration..... 2
 - Variable Declarations 3
 - Functions 3
 - DefaultProperties 3
- Classes and Extensibility 4
 - Extended Classes and DefaultProperties 5
 - Extended Classes and Functions 6
- Declaring Classes as Variables 8
- Conclusion and Additional Resources..... 8

Introducing Unrealscript Classes

While we're assuming that you already understand some coding basics, those of you who are only acquainted with Actionscript might not understand the concept of Classes. Simply put, Classes in Unrealscript are objects; *any* objects, from concrete things like individual Kismet nodes to individual weapons and projectiles to big, overarching, abstract objects like gametypes, inventory systems, and HUDs. Any time you want to develop a new object that uses functionality not already in Unreal, you create a new Class for it. To put it simply, **every individual script** in your Unrealscript Library is its own Class.

Anatomy of a Class

When we start developing a new Class in Unrealscript, it has a few basic features.

```
class MegaObject extends Actor;

var float x;
var bool isGordonAlive

function DoStuff()
{
    if (isGordonAlive == true)
        {
            x++;
        }
    else
        {
            x-=0.5;
        }
}

defaultproperties
{
    x=1.0
    isGrdonAlive=true
}
```

This is just a sample class showcasing a few of the basic features of any Unrealscript class, composed of a header or Class declaration, several variable declarations, a function, and the defaultproperties entry. Let's go through them one by one.

Header/Class Declaration

```
class MegaObject extends Actor;
```

This is where we declare our Class, just like we'd declare any other variable. When declaring a Class, you must give it the same name as you give the Unrealscript (.uc) file that you're writing it to.

It's also customary **extend** your script off of another Class in order to take quick shortcuts and inherit functionality, which is a concept we'll be fleshing out in more detail in its own section a little further on in this chapter. For right now, though, know that you do this in the header. Other features may appear

in the header as well, depending on what type of object is being scripted. So, your header might look like this:

```
class FrameworkGame extends GameInfo
    config(game)
    native;
```

That's the header for the "FrameworkGame" script, one of UDK's many default scripts. Note that the position of the semicolon shifts to *after* the new declarations, meaning that the whole header gets one semicolon.

Variable Declarations

If you've coded before, you know this part:

```
var float x;
var bool isGordonAlive;
```

Variable declarations in Unrealscript happen the same place they do anywhere else--right at the top of your script, after the header but before you start coding functions. Their declarations are more or less identical to declarations in Javascript, with the word "var," followed by the variable type, followed by the name.

Functions

The body of any Unrealscript class is made up of many functions, like this one:

```
function DoStuff()
{
    if (isGordonAlive == true)
    {
        x++;
    }
else
    {
        x-=0.5;
    }
}
```

Functions are simply ways of compartmentalizing functionality to be called when needed. Every action that a Class can take is organized into a function, otherwise we get an error and the action will never run.

DefaultProperties

One thing you'll probably have noticed is that I didn't set any defaults for either "x" or "isGordonAlive" when I declared them, which would ordinarily be a no-no in any other scripting language.

In Unrealscript, however, we have the DefaultProperties section to do this for us, and it's the last section of Unrealscript Class.

```
defaultproperties
{
```

```
        x=1.0
        isGordonAlive=true
    }
```

Note that there are no semicolons used here.

It may seem unintuitive to put this information at the bottom of our Class, but this section offers us many advantages, not the least of which is organization. Without needing to know the script all that well, a designer can come in and tweak a variable quickly, knowing that they're all organized at the bottom of the page, re-compile the code, and then see how the script runs with the new numbers. It's also notable that it's called *default* properties, meaning that other code can easily re-define it for specific instances of objects.

Additionally, `DefaultProperties` doesn't *just* define the default values for the code that we write here. It also exposes the default values of variables and code that we *inherit* from other scripts that we extend. Before we get into that, though, we should explain what exactly it means to "extend" a script...

Classes and Extensibility

Classes in UnrealScript are heavily based on the concept of extensibility. Notice in the header of the sample class above that we had this little bit of code:

```
class MegaObject extends Actor;
```

The "extends" bit refers to what type of Class our new Class is extending itself off of; in this case, we're extending off of one of the most basic Classes, "Actor," which describes objects in UDK in their simplest form--and therefore we inherit all of the features and variables of the "Actor" Class without having to re-invent the wheel.

If we wanted to we could replace it with "Vehicle," in which case our class would inherit all the functions and data that generally define vehicles in the Unreal engine, saving us a lot of time and effort that we'd ordinarily spend on details like defining seats in the vehicle as well as most of the vehicle physics in the game. It isn't quite the same as using an `#include` declaration in C++ in that you can only extend one class at a time, meaning that you can't have an object that's both a vehicle *and* a weapon. Therefore, it takes a good deal of care and foresight to develop a hierarchy of UnrealScript classes, and you need to be certain of what types of objects you're developing and how they pass their functionality down to one another.

The existing scripts both in UDK's engine scripts and the default Unreal Tournament code all use this concept to great advantage, with scripts for individual vehicles and weapons taking up very little space--hundreds of lines of code as opposed to thousands--and being very easy to navigate, with very little need to re-code basic functions from the ground up.

One of the other practical upshots of this model is that there's never more code running at any given time than has to exist. One of the disadvantages of this model, however, is that often times there are many, many layers of UnrealScript extensions to one fairly basic-seeming object, making it difficult to tell where the functionality we want to edit actually is.

To give an example, let's look at the Unreal Tournament Link Gun. The script that defines the Link Gun is called `UTWeap_LinkGun.uc`. So, we trace its extension to the very lowest level of the code we can...

- `UTWeap_LinkGun` extends `UTBeamWeapon`
- `UTBeamWeapon` extends `UTWeapon`
- `UTWeapon` extends `UDKWeapon`
- `UDKWeapon` extends `Weapon`
- `Weapon` extends `Inventory`
- `Inventory` extends `Actor`
- `Actor` extends `Object`
- `Object` doesn't extend anything

Yikes! That's a lot of extending, and a lot of seemingly redundant Classes layered on top of one another just to get to the Link Gun. Fortunately a lot of it is simply taking care of heavy lifting for us and we can just ignore much of it, but if we want to script new weapons--ones that *aren't* Unreal Tournament-based--then where do we start? Do we start at `Weapon`, `UDKWeapon`, or is it actually safe to start at `UTWeapon` and would that not in fact save us a lot of time? This is the kind of thing that intimidates the heck out of modders and game designers trying to learn this language as it requires a lot of time and research to understand--but again, there are many advantages, and as we go into further chapters we'll be fleshing out a road map of where to start with any given asset or object.

Extended Classes and DefaultProperties

As we explained earlier, the `DefaultProperties` value of a Class doesn't *just* define the variables that you declare; it also *re*-defines variables that your Class inherits from extending other classes. Looking back at the Link Gun again and its `DefaultProperties` value, we can see things like...

```
WeaponEquipSnd=SoundCue'A_Weapon_Link.Cue.A_Weapon_Link_RaiseCue'
```

But if we were to search the script, we'd find no reference to any variable called "WeaponEquipSnd" other than this line in `DefaultProperties`. If we go down the classes that it extends as far as `UTWeapon`, though, we'll see the variable declared for us clear as day at the start of the script:

```
/** Sound to play when the weapon is Equipped */  
var(Sounds) SoundCue WeaponEquipSnd;
```

In other words, `WeaponEquipSnd`--among many, many other variables--is a general variable that's held in common among *all* UT Weapons; *all* of them have an equipping sound to play, but we're able to re-define it specific to the Link Gun in its own script.

Extended Classes and Functions

This ability to re-define lower-level code for specific instances of extended Classes isn't limited to just re-defining variables, however. We can also re-define functions.

For instance, and this may be getting a *little* bit ahead of ourselves, let's say we're writing a new Unreal Tournament gametype, extending UTGame--but our new gametype isn't going to add to players' scores based on killing. We want to use something else, like capturing objectives or making laps in a race. So, we take a look at the code for scoring kills, namely, the ScoreKill function:

```
function ScoreKill(Controller Killer, Controller Other)
{
    local PlayerReplicationInfo OtherPRI;
    local UTPawn KillerPawn;
    local UTGameReplicationInfo GRI;

    OtherPRI = Other.PlayerReplicationInfo;
    if ( OtherPRI != None )
    {
        OtherPRI.NumLives++;
        if ( (MaxLives > 0) && (OtherPRI.NumLives >=MaxLives) )
            OtherPRI.bOutOfLives = true;
    }

    Super.ScoreKill(Killer,Other);

    if ( (killer == None) || (Other == None) )
        return;

    // adjust bot skills to match player - only for DM, not team games
    GRI = UTGameReplicationInfo(GameReplicationInfo);
    if ( GRI.bStoryMode && !bTeamGame && (killer.IsA('PlayerController') ||
Other.IsA('PlayerController')) )
    {
        if ( killer.IsA('AIController') )
            AdjustSkill(AIController(killer), PlayerController(Other),
false);
        if ( Other.IsA('AIController') )
            AdjustSkill(AIController(Other), PlayerController(Killer),
true);
    }

    KillerPawn = UTPawn(Killer.Pawn);
    if ( (KillerPawn != None) && KillerPawn.bKillsAffectHead )
    {
        KillerPawn.SetBigHead();
    }
}
```

That's how ScoreKill is defined by default in Unreal Tournament's UTGame Class, which defines the base rules for any Unreal Tournament-based game. Wow, some complicated stuff, huh? But in *our* Class we can re-declare the function and re-write all of its innards like so:

```
class UTCustomGame extends UTGame
```

```

config(game);

function ScoreKill(Controller Killer, Controller Other)
{
    return;
}

DefaultProperties
{
    Acronym = "UTCGame"
    MapPrefixes[0] = "UTCGame"
}

```

Now in *our* gametype, ScoreKill does absolutely nothing. The simple "return" command we've inserted into its brackets completely re-writes all of the code from UTGame--but *only* for our gametype.

Being able to re-define functionality like this is very powerful and obviously has its uses, but suppose we want to just *add* to it? Wouldn't we have to copy all those lines of code? Well, we could, but there's a much more efficient way of doing it--namely, the "Super" command. "Super" is made specifically for inherited functions that you're trying to edit, passing down all the guts of the original function. So if we want ScoreKill and everything that goes with it...

```

function ScoreKill(Controller Killer, Controller Other)

{
    Super.ScoreKill(Killer,Other);
}

```

... we start by doing that instead. Thanks to the "Super" command, the function we've written takes in the same variables, then passes them back up to the *original* ScoreKill function as it was written in the script we're inheriting from. After that, we can add anything we want. For instance, we could re-write it so that the killer gets a message any time they score a kill.

```

function ScoreKill(Controller Killer, Controller Other)
{
    Super.ScoreKill(Killer,Other);
    Killer.Pawn.ClientMessage("This isn't gameplay, it's murder!");
}

```

These are just simple examples of ways that the concept of Class extensibility can be employed to your advantage when you start coding custom scripts. In the next chapter, we'll be putting a lot of these concepts to the test by actually developing a custom Unreal Tournament gametype, as well as discussing how classes communicate. The development of custom game rules will also offer us a good chance to get our feet on the ground and start figuring out where to begin when developing a new game.

Declaring Classes as Variables

As you may have noticed from my code above, variables can be defined not just as floats, integers, strings, and other common variable types, but also as the many different Classes that are defined in Unrealscript. Need a new variable to store a player's data? Simple.

```
var PlayerReplicationInfo PRI;
```

Just as you would declare any other variable, you can declare a new PlayerReplicationInfo, a new GameInfo, a new UTWeap_LinkGun, or whatever other Class you need in order to store data. This can be exceptionally handy for functions that commonly need to process custom variables that you might have created.

Conclusion and Additional Resources

While this is all just barely scratching the surface of how an Unrealscript Class works, we now have all the basics out of the way and have established a common vocabulary with which to start talking about programming. In our next chapter we will start doing exactly that and develop our own custom gametype, which will also give us a good starting point for talking about how Unrealscript Classes communicate with one another.

If you want to know more about the features and commands that Unrealscript has built in, the [UDN Unrealscript Reference](#) page has a full list with descriptions of every command and an even more complete breakdown of the Class structure.