# Classes, Inheritance and the Game Info Class

## Classes and Our Code

All the code we do is stored as a class. The classes are stored in one file that contains the necessary code. One file is one class.

## What the Classes Do

Classes store code and can be roughly divided up into two types: classes that define game world, its behaviour and conditions and classes that define objects within the game world.
Examples of the first are mutators and game types, examples of the second are weapons and pick ups.

## Classes and Inheritance

**Definition:** Inheritance is a process or technique by which one class, the derived class, inherits the members and methods of another class, the base class. Additional members and methods are usually added to the derived class to make it more specialized.

Inheritance further models real world concepts by providing a natural classification of objects and for allowing the commonality of objects to be taken advantage of.

Inheritance is a relationship between class where one parent is the ancestor (or parent/whatever) of another class. It provides programming by extension rather than reinvention. For instance in UT2K3 we don't need to have a Player class for each game type, rather we have a base class Player which is then extended by the different player classes for the different game types.
You may also see inheritance referred to in is-a or is-a-kind-of terms, this is just another way of describing the concepts.
These inherited classes when further inherited form what is called a hierarchy.

A good example is that of the way taxonomists often divide up life.
Life is divided into kingdom, phyla, class, order, family, genus, and species.
You can imagine this like:

Animal → Mammal → Feline → Lion

Where the concept Lion would inherit (gets the features from) from Feline, Feline from Mammal and so on.
These concepts may contain information like:
Mammal – breathes air, moves
Feline – Extend Claws, be evil
The ideas of the Feline extending claws etc would be added to the idea in mammal. So a Feline would breathe air, move, extend claws, be evil.

These classes further down the line that inherit, are also said to be derived. This is a way of expressing an is-a relationship.
You derive the new class Feline from the class Mammal, therefore a class that adds new functionality to an existing class derives from the original class. The original class is generally called the base class.
Often a base class has more than one derived class, for instance we could add tigers:

Animal → Mammal → Feline → Lion
→ Tiger

In this case both lions and tigers are derived from Felines.

**Unreal and Inheritance**
Unreal 2k3 makes use of inheritance – all of the class, bar object, inherit from another class.

It is important to understand the high-level relationships of objects within Unreal. The architecture of Unreal is a major departure from that of most other games: Unreal is purely object-oriented (much like COM/ActiveX), in that it has a well-defined object model with support for high-level object oriented concepts such as the object graph, serialization, object lifetime, and polymorphism.

Historically, most games have been designed monolithically, with their major functionality hardcoded and unexpandable at the object level, though many games, such as Doom and Quake, have proven to be very expandable at the content level. There is a major benefit to Unreal's form of object-orientation: major new functionality and object types can be added to Unreal at runtime, and this expansion can take the form of subclassing, rather than (for example) by modifying a bunch of existing code. This form of extensibility is extremely powerful, as it encourages the Unreal community to create Unreal enhancements that all interoperate.

Object is the parent class of all objects in Unreal. All of the functions in the Object class are accessible everywhere, because everything derives from Object. Object is an abstract base class, in that it doesn't do anything useful. All functionality is provided by subclasses, such as Texture (a texture map), TextBuffer (a chunk of text), and Class (which describes the class of other objects).

Actor (expands Object) is the parent class of all standalone game objects in Unreal. The Actor class contains all of the functionality needed for an actor to move around, interact with other actors, affect the environment, and do other useful game-related things.

Pawn (expands Actor) is the parent class of all creatures and players in Unreal which are capable of high-level AI and player controls.

Class (expands Object) is a special kind of object which describes a class of object. This may seem confusing at first: a class is an object, and a class describes certain objects. But, the concept is sound, and there are many cases where you will deal with Class objects. For example, when you spawn a new actor in UnrealScript, you can specify the new actor's class with a Class object.

With UnrealScript, you can write code for any Object class, but 99% of the time, you will be writing code for a class derived from Actor. Most of the useful UnrealScript functionality is game-related and deals with actors.

**The class declaration**

Each script corresponds to exactly one class, and the script begins by declaring the class, the class's parent, and any additional information that is relevent to the class. The simplest form is:

class MyClass extends MyParentClass;

Here I am declaring a new class named "MyClass", which inherits the functionality of "MyParentClass". Additionally, the class resides in the package named "MyPackage".

Each class inherets all of the variables, functions, and states from its parent class. It can then add new variable declarations, add new functions (or override the existing functions), add new states (or add functionality to the existing states).

The typical approach to class design in UnrealScript is to make a new class (for example a Minotaur monster) which expands an existing class that has most of the functionality you need (for example the Pawn class, the base class of all monsters). With this approach, you never need to reinvent the wheel – you can simply add the new functionality you want to customize, while keeping all of the existing functionality you don't need to customize. This approach is especially powerful for implementing AI in Unreal, where the built-in AI system provides a tremendous amount of base functionality which you can use as building blocks for your custom creatures.

The class declaration can take several optional specifiers that affect the class:

native: Says "this class uses behind-the-scenes C++ support". Unreal expects native classes to contain a C++ implementation in the DLL corresponding to the class's package. For example, if your package is named "Robots", Unreal looks

in the "Robots.dll" for the C++ implementation of the native class, which is generated by the C++ IMPLEMENT_CLASS macro.

Abstract: Declares the class as an "abstract base class". This prevents the user from adding actors of this class to the world in UnrealEd, because the class isn't meaningful on its own. For example, the "Pawn base class is abstract, while the "Brute" subclass is not abstract – you can place a Brute in the world, but you can't place a Pawn in the world.

guid(a,b,c,d): Associates a globally unique identifier (a 128-bit number) with the class. This Guid is currently unused, but will be relevent when native COM support is later added to Unreal.

transient: Says "objects belonging to this class should never be saved on disk". Only useful in conjunction with certain kinds of native classes which are non-persistent by nature, such as players or windows.

config(section_name): If there are any configurable variables in the class (declared with "config" or "globalconfig"), causes those variables to be stored in a particular configuration file:
config(system): Uses the system configuration file, Unreal.ini for Unreal.
config(user): Uses the user configuration file, currently User.ini.
config(whatever): Uses the specified configuration file, for example "whatever.ini".