Objects

Explanations of the basic gameplay elements, Actors and Objects.

Unreal has a robust system for handling game objects. The base class for objects in Unreal is `UObject`. The `UCLASS` macro can be used to tag classes derived from `UObject` so that the **UObject** handling system is aware of them.

The UCLASS Macro

The **UCLASS** macro gives the `UObject` a reference to a `UCLASS` that describes its Unreal-based type. Each `UCLASS` maintains one Object called the **Class Default Object**(CDO). The CDO is essentially a default 'template' Object, generated by the class constructor and unmodified thereafter.

Both the UCLASS and the CDO can be retrieved for a given Object instance, although they should generally be considered read-only. The UCLASS for an Object instance can be accessed at any time using the `GetClass()` function.

A `UCLASS` contains a set of properties and functions that define the class. These are normal C++ functions and variables available to standard C++ code, but tagged with Unreal Engine-specific metadata that controls how they behave within the Object system. For more details about the tagging syntax, refer to the Programming Reference.

> A UObject class can include native-only properties that are not marked for reflection with UFUNCTION or UPROPERTY specifiers. However, only functions and properties that are marked with specifier macros will get listed within their corresponding UCLASS.

Properties And Function Types

UObjects can have member variables (known as properties) or functions of any type. However, for the Unreal Engine to recognize and manipulate those variables or functions, they must be marked with special macros and must conform to certain type standards. For details on those standards, refer to the Properties and UFunctions reference pages.

UObject Creation

UObjects do not support constructor arguments. All C++ UObjects are initialized on engine startup, and the engine calls their default constructor. If there is no default constructor, your UObject will not compile.

UObject constructors should be lightweight and only used to set up default values and subobjects, no other functionality should be called at construction time. For Actors and Actor Components, initialization functionality should be put into the `BeginPlay()` method instead.

UObjects should only be constructed using NewObject at runtime, or CreateDefaultSubobject for constructors.

Method	Description
`NewObject<class>`	Creates a new instance with optional parameters for all available creation options. Offers a wide range of flexibility, including simple use cases with an automatically generated name.
`CreateDefaultSubobject<class>`	Creates a component or subobject that provides a method for creating a child class and returning the parent class. When creating a default subobject, because they are constructed at engine startup the class constructor of a UObject should only work with local data or load static assets.

UObjects should never use the `new` operator. All UObjects are memory managed by Unreal Engine and garbage collected. When you manually manage your memory by using new or delete, you can cause corruption to your memory.

Functionality Provided by UObjects

It is not required or even appropriate to use this system in all cases, but there are many benefits to doing so, including:

- Garbage collection
- Reference updating

- Reflection
- Serialization
- Automatic updating of default property changes
- Automatic property initialization
- Automatic editor integration
- Type information available at runtime
- Network replication

> Most of these benefits apply to UStructs, which have the same reflection and serialization capabilities as UObjects. UStructs are considered value types and are not garbage collected. For more detail on each of these systems, refer to the Unreal Object Handling documentation.

The Unreal Header Tool

To harness the functionality provided by UObject-derived types, a preprocessing step needs to be run on the header files for these types to collate the information it needs. This preprocessing step is performed by the UnrealHeaderTool, or UHT for short. UObject-derived types have a certain structure that needs to be adhered to.

Header File Format

While a UObject's implementation in a source (.cpp) file is just like any other C++ class, its definition in a header (.h) file must adhere to a certain basic structure in order to work properly with Unreal Engine. Using the editor's **New C++ Class** command is the easiest way to set up a correctly-formatted header file. A basic header file for a UObject-derived class might look like this, assuming the UObject derivative is called UMyObject and the project in which it was created is called MyProject:

```
#pragma once

#include 'Object.h'
#include 'MyObject.generated.h'

UCLASS()
class MYPROJECT_API UMyObject : public UObject
{
GENERATED_BODY()

};
```

Copy full snippet

The Unreal-specific parts of this are as follows:

```
#include "MyObject.generated.h"
```

Copy full snippet

This line is expected to be the the last `#include` directive in the file. If this header file needs to know about other classes, they can be forward declared anywhere in the file, or included above `MyObject.generated.h`.

```
UCLASS()
```

Copy full snippet

The `UCLASS` macro makes `UMyObject` visible to Unreal Engine. The macro supports a variety of Class Specifiers that determine which features are turned on or off for the class.

```
class MYPROJECT_API UMyObject : public UObject
```

Copy full snippet

Specifying `MYPROJECT_API` is necessary if MyProject wishes to expose the UMyObject class to other modules. This is most useful for modules or plugins that will be included by game projects and which deliberately expose classes to provide portable, self-contained functionality across multiple projects.

```
GENERATED_BODY()
```

Copy full snippet

The `GENERATED_BODY` macro takes no arguments, but sets up the class to support the infrastructure required by the engine. It is required for all `UCLASS` and `USTRUCT`.

Unreal Header Tool supports a minimal set of C++. When wrapping parts of a UCLASS with custom `#ifdefs` macros, UHT will ignore macros that do not contain `WITH_EDITOR` or `WITHEDITORONLY_DATA` macros.

Updating Objects

Ticking refers to how Objects are updated in Unreal Engine. All Actors have the ability to be ticked each frame, providing you a way to perform any update calculations or actions that are necessary.

Actors and ActorComponents have their Tick functions called automatically when registered to do so, however, `UObjects` do not possess any built-in update ability. When it is necessary for your project, this can be added by inheriting from the `FTickableGameObject` class using the inherits class specifier. They can then implement the `Tick()` function, which will be called each frame by the engine.

Most in-game Objects will be Actors, which can tick at user-set minimum intervals rather than once per frame.

Destroying Objects

Object destruction is handled automatically by the garbage collection system when an Object is no longer referenced. This means that no `UPROPERTY` pointers, engine containers, `TStrongObjectPtr`, or class instances should have any strong references to them.

Note that Weak Pointers have no impact on whether an Object is garbage collected or not.

When the garbage collector runs, unreferenced Objects found are removed from memory. In addition, the function `MarkPendingKill()` can be called directly on an Object. This function sets all pointers to the Object to `NULL` and removes the Object from global searches. The Object is fully deleted on the next garbage collection pass.

Smart pointers are not intended to be used with UObjects.

- `Object->MarkPendingKill()` has been replaced with `Obj->MarkAsGarbage()`. This new function is now only for tracking stale objects, If `gc.PendingKillEnabled=true` then objects marked as `PendingKill` will be automatically nulled and destroyed by Garbage Collector.
- Strong references keep UObjects alive. If you don't want these references to keep the UObject alive, then those references should convert to using weak pointers, or be a normal pointer that is manually cleared by a programmer (if performance is critical.)

  You can replace Strong pointers with weak pointers and dereference them once during a gameplay operation as garbage collection only runs between frames.

- `IsValid()` is for checking if it's null or garbage, but most usages of IsValid can be replaced with proper programming conventions like clearing pointers to Actors when they have their `OnDestroy` event called.
- If `PendingKill()` is disabled, `MarkGarbage()` will flag to the owner of the object that it wants the object destroyed, but the object itself will not get garbage collected until all references to it are released.
- In the case of Actors, even if the Actor had `Destroy()` called on them, and they were removed from the level, then it will not be Garbage Collected until all references to it are released.
- The Main difference for licensees is that using the function `MarkPendingKill()` to force expensive objects to garbage collect will no longer work.

- Existing checks for nullptr should be replaced with `IsValid()` calls if you're not clearing them manually, since pointers will no longer get automatically cleared by the garbage collector through `MarkPendingKill()`.