Interface classes are useful for ensuring that a set of (potentially) unrelated classes implement a common set of functions. This is very useful in cases where some game functionality may be shared by large, complex classes, that are otherwise dissimilar. For example, a game might have a system whereby entering a trigger volume can activate traps, alert enemies, or award points to the player. This might be implemented by a "ReactToTrigger" function on traps, enemies, and point-awarders. However, traps may be derived from `AActor`, enemies from a specialized `APawn` or `ACharacter` subclass, and point-awards from `UDataAsset`. All of these classes need shared functionality, but have no common ancestor other than `UObject`. In this case, an interface is recommended.

Interface Declaration

Declaring an interface class is similar to declaring a normal Unreal class, but with two main differences. First, an interface class uses the UINTERFACE macro instead of the UCLASS macro, and inherits from `UInterface` instead of `UObject` directly.

```
UINTERFACE([specifier, specifier, ...], [meta(key=value, key=value, ...)])
class UClassName : public UInterface
{
    GENERATED_BODY()
};
```

Second, the UINTERFACE class is not the actual interface. It is an empty class that exists only for visibility to Unreal Engine's reflection system. The actual interface that will be inherited by other classes must have the same class name, but with the initial "U" changed to an "I".

In your .h file (e.g. `ReactToTriggerInterface.h`):

ReactToTriggerInterface.h

```
#pragma once

#include "ReactToTriggerInterface.generated.h"

UINTERFACE(MinimalAPI, Blueprintable)
class UReactToTriggerInterface : public UInterface
{
    GENERATED_BODY()
};

class IReactToTriggerInterface
{
    GENERATED_BODY()

public:
    /** Add interface function declarations here */
};
```

The "U-prefixed" class needs no constructor or any other functions, while the "I-prefixed" class will contain all interface functions and is the one that will actually be inherited by your other classes.

The `Blueprintable` specifier is required if you want to allow Blueprints to implement this interface.

Interface Specifiers

Interface Specifier	Meaning
`BlueprintType`	Exposes this class as a type that can be used for variables in Blueprints.
`DependsOn=(ClassName1, ClassName2, ...)`	All classes listed will be compiled before this class. ClassName must specify a class in the same (or a previous) package. Multiple dependency classes can be specified using a single `DependsOn` line delimited by commas, or can be specified using a separate `DependsOn` line for each class. This is important when a class uses a struct or enum declared in another class as the compiler only knows what is in the classes it has already compiled.
`MinimalAPI`	Causes only the class's type information to be exported for use by other modules. The class can be cast to, but the functions of the class cannot be called (with the exception of inline methods). This improves compile times by not exporting everything for classes that do not need all of their functions accessible in other modules.

Implementing Your Interface in C++

To use your interface in a new class, simply inherit from your "I-prefixed" interface class (in addition to whatever `UObject`-based class you are using).

Trap.h

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ReactToTriggerInterface.h"
#include "Trap.generated.h"

UCLASS(Blueprintable, Category="MyGame")
class ATrap : public AActor, public IReactToTriggerInterface
{
    GENERATED_BODY()

public:
    /** Add interface function overrides here. */
}
```

Declaring Interface Functions

There are several methods you can use to declare functions in your interfaces, each of which is implementable or callable in different contexts. All of them must be declared in the "I-prefixed" class for your interface, and they must be `public` in order to be visible to outside classes.

C++ Only Interface Functions

You can declare a virtual C++ function in your interface's header file, with no `UFUNCTION` specifiers. These functions must be virtual so that you can override them in classes that implement your interface.

ReactToTrigger.h

```
public:
virtual bool ReactToTrigger();
```

You can then provide a default implementation either within the header itself or within the interface's `.cpp` file.

ReactToTrigger.cpp

```
bool IReactToTriggerInterface::ReactToTrigger()
{
    return false;
}
```

When you implement your interface in an Actor class, you can then create and implement an override specific to that class.

Trap.h

```
public:
virtual bool ReactToTrigger() override;
```

Trap.cpp

```
bool ATrap::ReactToTrigger()
{
    return false;
}
```

However, these C++ interface functions will not be visible to Blueprint.

Blueprint Callable Interface Functions

To make a Blueprint callable interface function, you must provide a `UFUNCTION` macro in the function's declaration with the `BlueprintCallable` specifier. You must also use either the `BlueprintImplementableEvent` or `BlueprintNativeEvent` specifiers, and the function must not be virtual.

ReactToTrigger.h

```
public:
/**A version of React To Trigger that can be implemented in Blueprint only. */
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent, Category=Trigger Reaction)
bool ReactToTrigger();
```

ReactToTrigger.h

```
public:
/**A version of React To Trigger that can be implemented in C++ or Blueprint. */
UFUNCTION(BlueprintCallable, BlueprintNativeEvent, Category=Trigger Reaction)
bool ReactToTrigger();
```

BlueprintCallable

Functions using the `BlueprintCallable` specifier can be called in C++ or Blueprint using a reference to an object that implements the interface.

BlueprintImplementableEvent

Functions using `BlueprintImplementableEvent` can not be overridden in C++, but can be overridden in any Blueprint class that implements or inherits your interface.

BlueprintNativeEvent

Functions using `BlueprintNativeEvent` can be implemented in C++ by overriding a function with the same name, but with the suffix `_Implementation` added to the end.

Trap.h

```
public:
bool ReactToTrigger_Implementation() override;
```

Trap.cpp

```
bool ATrap::ReactToTrigger_Implementation() const
{
    return false;
}
```

This specifier also allows implementations to be overridden in Blueprint.

Determining If a Class Implements Your Interface

For compatability with both C++ and Blueprint classes that implement your interface, use any of the following functions:

```
bool bIsImplemented = OriginalObject->GetClass()->ImplementsInterface(UReactToTriggerInterface::StaticClass());

bIsImplemented = OriginalObject->Implements<UReactToTriggerInterface>(); // bIsImplemented will be true if Origi

IReactToTriggerInterface* ReactingObjectA = Cast<IReactToTriggerInterface>(OriginalObject); // ReactingObject wi
```

NOTE

If the `StaticClass` function is not implemented in the "I-prefixed" class, attempting to use `Cast` on the "U-prefixed" class will fail, and your code will not compile.

Casting To Other Unreal Types

Unreal Engine's casting system supports casting from one interface to another, or from an interface to an Unreal type, where appropriate.

```
IReactToTriggerInterface* ReactingObject = Cast<IReactToTriggerInterface>(OriginalObject); // ReactingObject wil

ISomeOtherInterface* DifferentInterface = Cast<ISomeOtherInterface>(ReactingObject); // DifferentInterface will

AActor* Actor = Cast<AActor>(ReactingObject); // Actor will be non-null if ReactingObject is non-null and Origir
```

Blueprint Implementable Classes

If you want Blueprints to be able to implement this interface, you must use the `Blueprintable` metadata specifier. Every interface function that your Blueprint class is intended to override, must be a `BlueprintNativeEvent` or a `BlueprintImplementableEvent`. Functions marked as `BlueprintCallable` will still be able to be called, but not overridden. All other functions will be inaccessible from Blueprints.