back <- up -> next

UnrealScript Syntax

by: Robert J. Martin

This is intended as a quick reference for someone who is very familiar with Java/C++ style OO languages and is at least somewhat familiar with UnrealScript in general. In particular, it is directed at those using UScript for academic purposes. I don't go in to a great amount of detail for many things, especially for things found in the UScript reference. Please Contact me if you see any mistakes, need for extra clarification on anything, or have any contributions to make.

Language Overview

For the most part, UnrealScript is very close to Java in syntax and behavior. Objects and inheritence are treated in a very similar way, and source code is compiled into something resembling byte code. There is an environment analgous to the JRE, in which Objects are instantiated and data is bound dynamically and function calls operate by making references to the underlying native code where appropriate. The parser uses a two-pass strategy.

1. Like java, you will not have to use pointers directly or destroy objects. There is a service comparable to garbage collection that will do it for you.
2. There are NOT separate header/source files. Declarations and Definitions are done in xxx.uc source file.
3. Source files are compiled into a byte-code object file, xxx.u.
4. You can only reference classes defined in the same package or in one that was compiled before the class that is referencing it. Package compilation order is determined by its order in in the EditPackages section of the xxx.ini file that UCC.exe uses (UnrealTournament.ini by default).

Conventions (this document)-

- *italics* - used to represent keywords in UnrealScript
- `code` - used when I give examples in compilable, UScript

Basic Syntax

- General -
  - C/C++ style comments -
    - e.g.- //this will not be compiled... /*nor will this*/
  - code is NOT case sensitive

- Class Declaration -
  - *extends* and *expands* can be used interchangably for inheritance.
    - `class MyGame expands TournamentGameInfo;`
      or...
      `class MyGame extends TournamentGameInfo;`
      ...will both work
  - The class declaration must be the first line (except for comments).
  - *#exec* commands, if present, must follow immediately after class declaration

- Variables -
  - *var* -
    - It is the first word in class or state variable declaration
    - You CANNOT initialize variables when they are created.
      - Right: `var int myInt;`
      - Wrong: `var int myInt = 100;`
    - Var's are editable from UTEd when parentheses are used.
  - *local* -
    - Use *local* for declaring variable inside of a function. It will be recycled by the garbage collector when a function goes out of scope
    - example
      - ```
        function myFunction( ) {
        local int myInt;
        }
        ```

- Data Types -
  - *int* -
    - size: 32-bit
    - range: -2147483648 to 2147483647
    - literal = nnn
        ```

- *float* -
  - size: 32-bit
  - range: -8388608 to 8388608 (I think)
  - literal = nnn.nnn
  - floats literals from -1 to 1 exclusive MUST start with a 0
    - Right: `x = 0.314;`
    - Wrong `x = .314;`

- *bool* -
  - literal = true or false
  - does not support automatic string resolution
  - The convention is to start bool variables with "b". You will see this throught UScript code. Try to stick with this to maintain consistency .

- *byte* -
  - size: 8-bit (1 byte)
  - range: 0-255
  - literal = 0xnnn

- *enum* -
  - size: 8-bit
  - example:

    - enum EColor{
      CO_Red,
      CO_Green,
      CO_Blue
      };
      var EColor ShirtColor, HatColor;

      or...

      var enum EColor{
      CO_Red,
      CO_Green,
      CO_Blue,
      } ShirtColor, HatColor;

  - enums are stored internally as a byte. Don't use enums as parameters in functions. If an outside class attempts to call that function, you will get a compiler error even if both classes have the same definition. Use a byte instead:
    - Right: `function keyEvent(Byte key);`
    - Wrong: `function keyEvent(EInputKey key);`
  - useful functions
    - int enumCount(Enum) - returns the number of elements for this enumeration

- *string* -
  - literal = double quotes.
    - ```
      string s;
      s = "hello";
      ```
  - It is a primitive data type, NOT an object
  - many objects, int, float, Object, etc... support automatic string resolution. bool does not.
      ```

- concatenation -
  - *$* - used to concatenation string variables and/or literals together
    - ```
      string s, t;
      s = "Rocks";
      t = "UT" $s $"MyWorld"; // t = "UTRocksMyWorld"
      ```
- relevant functions -
  - int Len(string) - returns the length of the string
  - int inStr(string s, string t) - returns the offset of the first appearance of t in s, or -1 if it is not found
  - string Mid(string s, int i, optional int j) - returns the first j characters of the string starting at offset i
  - string Left(string, int i) - returns the i leftmost characters of the string
  - string Right(string, int i) - returns the i rightmost characters of the string
  - string Caps(string) - returns the string converted to uppercase

- *name* -
  - The *name* type is different from the *string* data type, but it can be cast as a string implicitly.
  - It is constant; it cannot be changed after an object has been created. You can change it at spawn time, or within UTEd.
  - An object's Name is guaranteed to be unique within a level. Default is the object's class concatenated with an enumerator
    - e.g.- `SniperRifle12`, `TMale2`, etc...
  - literal = single quotes
    - `if (other == 'Commander13') { ... }`
  - The name type can be used to represent class names as well in certain contexts. (see *Class*)

- *struct* -
  - UScript structs are similar to C style structs. They are used for variables, but not functions.
  - use -

    - struct Vector {
      var float X;
      var float Y;
      var float Z
      };

      var Vector destination;

  - predefined structs -
    - *Vector*: A unique 3D point or vector in space, with an X, Y, and Z component.
    - *Plane*: Defines a unique plane in 3D space. A plane is defined by its X, Y, and Z components (which are assumed to be normalized) plus its W component, which represents the distance of the plane from the origin, along the plane's normal (which is the shortest line from the plane to the origin).
    - *Rotator*: A rotation defining a unique orthogonal coordinate system. A rotation contains Pitch, Yaw, and Roll components.
    - *Color*: An RGB color value.
    - *Region*: Defines a unique convex region within a level.

- array -
  - Only one-dimensional arrays are valid
  - only static arrays are valid, you cannot define array size with a variable, only a literal.
    - Right: `var int myArray[10];`
      ```

        - Wrong: `var int myArray[x];`
    - relevant functions
        - int arrayCount(array) - returns the number of elements in the array

- *Object* -
    - constructors -
        - There are NO constructors. You can use the DefaultProperties section for default initialization, but there are no constructor functions for object creation.
    - instantiation (for non-Actor objects) -
        - ```
          Console C;
          C = new (none) class 'myNewConsole'; /* instantiate a new Console of type myNewConsole */
          ```
    - *None* - equivalent of NULL in Java
    - Use "." separation for object resolution
        - `myActor.MoveTo(myAmmo.location);`
    - typecasting - you can use typcasting as in Java, but it looks like a function call
        - ```
          local Actor A;
          A = enemy; //get this classes current enemy pawn
          A = PlayerPawn(A); // cast A as a playerPawn.
          ```
        - will return *None* if it cannot convert
    - *super*( ) -
        - used to explicitly use parent data or behavior
        - you can go multiple levels up the hierarchy
            - `super(ancestorClass).ancestorVersion( ); // calls the function of this class's ancestor of type "ancestorClass"`
            - `notice that there are no single quotes around ancestorClass in this context`
    - Objects v. Actors
        - instantiation (for Actors)
            - ```
              Actor A; // called from an actor
              A = spawn(class 'BoomStick', self, , location, rotation);
              ```
            - params
                1. Class - describes the class of the object you want to spawn (must be of type Actor)
                2. Actor - owner of the new Actor - optional
                3. Name - name of new Actor - optional
                4. Vector - location of new Actor - optional
                5. Rotator - rotation of new Actor - optional
        - Actors use *Self* instead of *This*;
        - Actors get their Tick(float) function called by the engine, Objects do not have this function.
        - Actors get their Pre/PostBeginPlay( ) as well as BeginPlay( ) functions called by the engine when the game starts up. This is another good place to put initialization and stuff. I don't know the difference between these functions.

- *class* -
    - This is a class object. It is of *name* type and is a reference to a particular class type
    - all Objects (and Actors) have a class field which can be used to get its class for various uses.
    - relevant functions -
        - bool isA(name queryClass) - returns true if the possessing object is of the type
              ```

represented by the queryClass name.
- class limiter -
  - Use the classlimiter to make sure that an object is of a certain class.
  - This is like typecasting but for constraining classes DOWN the hierarchy tree
  - example -
    - other is of type actor, but we only want it to be non-None if a Pawn is assigned to it.
    - `var class<Pawn> C; // code from within in an Actor class`
      `C = other.class; //will return null if not of type pawn.`

- Variable Specifiers -
  - *private* - same as Java, but not used to the same extent. This specifier is used very little in practice
  - *const* - cannot be changed once initialized
  - *transient* - will not be saved to disk
  - *config* - configurable using xxx.ini and xxx.int files.
  - *native* - the variable will be stored using native C++ code rather than UScript code
  - editable - editable variables can be changed within UTEd. put parens after the word *var* to designate a variable as editable
    - `var( ) int MyInteger; // Declare an editable integer in the default category.`
    - `var(MyCategory) bool MyBool; // Declare an editable integer in "MyCategory".`

- Conversions -
  - UnrealScript supports many automatic conversions, e.g. - *rotators* to *vectors*
  - see UScript reference

- Operators - see UScript reference

- Flow Control -
  - *if* -
    - more like Java than C/C++.
    - you must use boolean statements in almost all contexts. There are exceptions but it is better to just stick to the rule to be safe.
      - Right - `if(Actor != none) { ... }`
      - Wrong - `if(Actor) { ... }`
    - Just like Java/C++, you don't have to use curly brackets if your consequent is one line.
      - `if(true) x = 10;`
      - `if(true) { x = 10; y = 12; }`
  - *switch* - almost exactly like C++/Java. I am not aware of any differences
  - Loops -
    - *for* -
      - just like *if*, you must have a true boolean value
      - you cannot declare a new variable in the condition section of the for loop
        - Right - `int i; for( i = 0; ....)`

        - Wrong - `for( int i = 0; ....)`

    - *while* -
        - just like *if*, you must have a true boolean value

    - *do-until* -
        - just like *if*, you must have a true boolean value
        - UScript's version of "do ... while", but the semantics are reversed
            - i = 0;
              j = 0;
              do
              {
              i = i + 1;
              j ++;
              } until(i == 4);
            - both i and j will be 4 when the loop terminates

    - *foreach (*iterator*)* -
        - these are special loops that can only be used from by subclasses of Actor.
        - this is a quick way to go through a list of all actors or actors of a certain type.
        - types - (Taken directly from UScript Reference)

            - AllActors (class BaseClass, out actor Actor, optional name MatchTag);
                - Iterates through all actors in the level. If you specify an optional MatchTag, only includes actors which have a "Tag" variable matching the tag you specified.
                - ```
                  Actor A;
                  foreach AllActors( class 'PlayerPawn', A) { //iterate through PlayerPawns
                  Log(A.location) // Log their location
                  }
                  ```
            - ChildActors(class BaseClass, out actor Actor);
                - Iterates through all actors owned by this actor. BasedActors(class BaseClass, out actor Actor); Iterates throgh all actors which are standing on this actor.
            - TouchingActors(class BaseClass, out actor Actor);
                - Iterates through all actors which are touching (interpenetrating) this actor.
            - TraceActors(class BaseClass, out actor Actor, out vector HitLoc, out vector HitNorm, vector End, optional vector Start, optional vector Extent);
                - Iterates through all actors which touch a line traced from the Start point to the End point, using a box of collision extent Extent. On each iteration, HitLoc is set to the hit location, and HitNorm is set to an outward-pointing hit normal.
            - RadiusActors(class BaseClass, out actor Actor, float Radius, optional vector Loc);
                - Iterates through all actors within a specified radius of the specified location (or if none is specified, this actor's location).
            - VisibleActors(class BaseClass, out actor Actor, optional float Radius, optional vector Loc);
                - Iterates through a list of all actors who are visible to the specified location (or if no location is specified, this actor's location).

- Functions -
                  ```

- definitions start with the word, *function*
- returntypes - return type is optional, void is implicitly assumed if there is not one specified.
- parameters -
  - *optional* - means this parameter does not need to be present when the function is called. To test whether it has been specified from within the function in question, just test to see if the variable has been set.
    - ```
      function bool bNoArgs(int j, string s, Actor A) {
      if( (j == 0) && (s == "") && (A == none) )
      return true;
      }
      }
      ```
  - *coerce* - can be used to force a parameter to be cast as a certain type when the function is called. The only context I see this used in is forcing parameters to be cast as strings.
    - function returnString(coerce string i, coerce string c, coerce string a) {
      return i $c $a;
      }
    - if this function is called in this way, with three non-"null" parameters:
      - int i;
        Class c;
        Actor a;
        ... // initialize i, a, and c
        Log(returnString(i, c, a));
    - ... then it will return the integer, class, and actor reference as a concatenated string
  - *out* - used for pass-by-reference.
- overriding & overloading -
  - overriding is possible, in fact extremely necessary.
  - overloading functions with different parameter lists or return type and the same name is impossible.
- Specifiers -
  - *static* -
    - so that the function can be called independent of an instance. You still need an object to be instantiated to call it though, I think using the Class object might work:
      - ```
        Class MyClass = MyObject.class;

        MyClass.myStaticFunction;
        ```
    - ... but I'm not sure. If someone knows better please contact me and I will update this section
  - *singular* - only one instance of this function can be called at once. Useful for avoiding infinite recursion for mutually dependent functions... see UScript Reference
  - *native* - can only be used inside of a class that is declared native. Every native function will require a native C++ definition.
  - *exec* - can be used to have this function callable from the console or using consoleCommand (string).

- *event* -

- Native functions that will be passed up from the engine.
- A prime example is keyEvents and mouseEvents from input, etc...

- States -
  - used to make FSM's within Objects. Mainly for AI purposes.
  - Makes procedurally based code rather than functionally based. Like Basic if you were ever unfortunate enough to use that thing.
  - CAN be used with non-Actor objects. I'm not sure why the UScript reference says "Actors". Though, in practice, it is almost always used within Actors. The exception I've seen is in UWindow stuff, which does not subclass from Actor.
  - data and behavior hiding -
    - each state can have different versions of the same data and functions. This essentially allows different logic for function calls depending upon the context (current state). e.g.- most pawn descendents have several different definitions for Bump(Actor other), so that the Actor possessing the function will behave differently.
  - *auto* - A state declared auto will be the default state of an Actor if another one is not prompted to take effect.
  - *label* -
    - *Begin* - default starting point
    - always use at least one label.
    - state
  - goto("label")
    - will go to the label within the state
  - gotoState('stateName', optional 'labelName')
    - goes to the state, with the option of going to a label other than begin
  - latent functions -
    - latent functions are native functions that take some amount of time to return. They can only be called from within state code. Examples are MoveToward(Actor), Sleep(float), and FinishAnim( ). beginState( ), endState( )
  - *ignores* -
    - can be used to tell the state to ignore events and function called by the engine. This is to avoid a lot of the confusion that arises from mixing inheritance and states. If you don't want some other class or logic calling the global version of Bump( ) that might take you out of your state, just tell your state to ignore the Bump function.
      - ```
        state myState {
        ignores Bump;
        }
        ```
  - BeginState/EndState functions:
    - you can use these to prepare or wrap-up things before and after a state is used, respectively
    - A state's BeginState( ) function is called before the state is entered, and EndState( ) is after the state is left (but before another state is entered), know matter what caused the state change.
  - *global* - see UScript reference

- DefaultProperties-
  - You can sort of think as this as a default constructor. When an Actor is created, values can be initialized here. This is also a way to set global variables that you may use for development. I owe a lot of sleepless nights to this section.
  - This must be the last section in the class file.
  - DO NOT end defaultproperty statements with a semicolon(;) and...
  - DO NOT put spaces in between the "=" and the variable and value
        ```

        - Right: `ThisValue=100`
        - Wrong: `ThisValue = 100;`
    - array elements are dereferenced using parentheses instead of square brackets
        - Right: `myArray(0)=5`
        - Wrong: `myArray[0]=5`

- Debugging

    - Log(string) -
        - This will write the string to whatever xxx.log file is defined in the games xxx.ini file.
    - BroadcastMessage(string) -
        - can only be called by classes inheriting from Actor. This will write a text message to the HUD so you can do runtime checking
    - ConsoleCommands -
        - see UScript reference(ConsoleCommands). You can declare a function to be of exec type and it can be called from the console if it is defined in certain classes: Console, HUD, etc.

Advanced Syntax-

- Dynamicism - This poorly labeled section is devoted to loading and manipulating things dynamically/by-name, which you cannot do with most languages.

    - Classes -
        - DynamicLoadObject(string className, name baseClass). used to get a Class object from a string.
          actually reads the file xxx.int package you provide to get the class object
            - ```
              String myClassName = "myPackage.myClass";
              Class myClassObject = DynamicLoadObject(myClassName, class 'Class');
              baseClass myObject = spawn (class <baseClass> myClassObject);
              ```
            - So, you need to have a string that contains the package and the class, and you can instatntiate an object by a dynamically determined class name... neat, huh? If you need to get the package by string, you will have to use the ConsoleCommand for listing

    - Data -
        - You can actually set and get variables using the string representation of a variable's name. in the statement, x = 20; the string is "x".
            - String getPropertyText(String propertyName) - returns the property as a string:
                - ```
                  int myVar = 300;
                  String sValue = getPropertyText( "myVar" ); // sValue is equal to "300"
                  ```
            - setPropertyText(String propertyName, String propertyValue) - sets propertyName to propertyValue uses built-in conversion rules (String to int, String to float, etc...)

    - Functions -
        - The key to calling functions by their string name lies in the Exec keyword. What you have to do is make whatever function you want to call an exec style function:
            - ```
              exec function myFunction(coerce string s) { }
              ```
              In this code, you have made 'myFunction' callable from the console, with whatever you write after any whitespace following the word 'myFunction' being pushed into the 's' variable

              so, from the console you would type :
              ```

> myFunction myArgument
... then 's' will be equal to "myArgument", and you can use that within myFunction

Now then, there is also a function within class Console called 'bool ConsoleCommand (coerce string s)'. to call your exec'd function, 'myFunction' from code, you type:

```
bool isFunctionThere; //optional
isFunctionThere = ConsoleCommand("myFunction myArgument");
```

- ... 'isFunctionThere' will be true if the function was there, false otherwise.
- NOTE: this technique will only take effect if you put your *exec* style function in an ACTIVE class that can use *exec* style functions. Examples include Console and HUD. Just because your class subclasses from one of these does not mean your function will be recognized. It has to be considered active by the engine (gameInfo).

- Operator Creation -
  - You can create your own operators as well. For clues, check out Object.uc. It isn't much more complicated than that.
  - example - I have an object I created that subclass directly from object for using XML. Let's say I want to override the concatenation operator so that it will make a new XML object when in the context of "XMLObject $string". The definition will start out something like this
  - `static final operator(40) XMLObject $ ( XMLObject A , string B ) { ... }`
  - This must be static and final. This means you can't make any references to instance variables or non-static functions within that class. I CAN, however make references to 'A' instance data.
  - The (40) is a priority number for deciding order of operations in mixed expressions.
  - In this example XMLObject is the return type and $ is the operator.
  - A is the left outfix parameter, B is the right.