PART Scripting Guide

Date: January 31, 2020

The scripting language used in PART has been modeled closely after the basic syntax of C# (based in turn on C/C++), though it has been adapted to fit PART better.

Contents

Data Types

There are four basic datatypes currently implemented in PART scripts. These are Integers, Doubles, Strings, and Booleans.

Integers, or `int` in declarations, are signed 32-bit integers. This means that they cannot hold any fractional values (values after the decimal place are automatically `floor`ed), can hold a value anywhere between −2,147,483,648 and 2,147,483,647.

Doubles, or `double` in declarations, are 64-bit floating point numbers. The computer tracks and thinks of these numbers effectively in scientific notation: reserving 11 of the bits to represent the exponent, 52 bits to represent the fractional component, and one to represent the sign. The technical limits of the numbers that doubles can represent are about +/- 1.7 * 10^308, where the smallest non-zero magnitude number that can be represented is about 5 * 10^-324.

Booleans, or `bool` in delcarations, simply represent a value of `true` or `false`. In principle they can be represented with only 1 bit, but because of how memory is utilized in different programming environments, they are frequently either 8 bits or 32 bits.

Strings, or `string` in declarations, represent text. In PART scripts, text is written surrounded by double quotation marks, like this: `"This is a string"`. There are a few ways to write special characters as well. The character `\` is known as an Escape character, because it lets you enter special characters (based on the one that follows it). If you need to include a double-quote character in the text, you do so by escaping it like this: `"Then he said, \"I don't believe it.\""` Similarly, new lines can be inserted with `\n\r`, and if you want a backslash in the text, then simply use `\\`.

Comments

Comments are helpful statements written in a block of code to help readers understand what you're doing. They are completely ignored by the program. Comments are either written with a double slash `//` or with the block style `/* Comment goes in between */`. When you use the double slash, the comment automatically ends at the end of the current line. The block style starts at the `/*` continues until it reaches the `*/` marker.

Semicolons

In PART Script, statements end with semicolons. This is how the program knows the statement has finished. With the semicolon omitted, you can continue a particularly long expression on the following line.

```
double exampleDuration = 10.0; //in milliseconds
int particularlyLongExample = (int)Math.Floor(3.14159 * 2.0 *
    0.001 * exampleDuration);
```

Declarations and Assignments

Broadly speaking, you need variables to do anything meaninful in a PART script. The simplest type of variable is a *local* one, which, as the name suggests, just lives in the script and ceases to exist whenever it is not running. Local variables can be declared and assigned like this:

```
int firstIntegerVariable;
int secondTestInteger = 10;
//Note: the value used in an assignment can be an expression...
int thirdInt = firstIntegerVariable * secondTestInteger;

double anExampleDouble = 20.0;
anExampleDouble = secondTestInteger;
/*Note: The above line is valid because Integers can be implicitly converted to
```

```
Doubles, as you're guaranteed not to lose any information.  The reverse, however
is not true, and requires a Cast*/

string anExampleString = "Here is a simple String";

bool anExampleBoolean = false;
```

If you don't assign it a value, as with `firstIntegerVariable`, then it defaults to 0.

Scope and Blocks

Scope is an important concept in Programming, and it's no different in PART Scripts. When a local variable is declared, it is only available to the current Block and every Block inside of it. Declaring a local variable at the top of a script object makes it available to the entire script, but to nothing outside.

Blocks are created with Curly Braces, `{` and `}`, and are frequently used in some of the structures we'll see soon.

```
int localInt;

{
    //This is perfectly valid, because localInt is available in this internal
block
    int smallerScopeInt = localInt;
}

//This line would be a syntax error because smallerScopeInt is not in this scope,
AND it no longer exists, besides
localInt = smallerScopeInt  //NO, BAD
```

`global` and `extern`

Local variables are great, but if you can't do anything with them and they only have the values you give them, then they'll hardly be helpful. That is where Global and Extern come into play. When a global variable is declared, it and its value are declared for the whole running battery. Other scripts can access globals by declaring them as well. Similarly, when a task makes use of a variable for one of its parameters, or when a task saves one of its outputs to a variable, these live in the same pool of memory as the globals. This is how scripts and tasks can talk to one another.

```
//Script 1
global int testInteger1;
global int testInteger2 = 1;
global int testInteger3 = 1;
```

Extern, on the other hand, works very similarly, but it will not create the global variable. It merely asserts that it must already exist.

```
//Script 2
global int testInteger1 = 2;
extern int testInteger2;
```

If you do not declare a variable as `global` or `extern` in a script, you cannot use it, even if it exists. This helps protect script writers against accidentally using a global when they didn't intend to, or confusing situations where it's unclear what variable or value is being used. Thus *Script 2* above would not be able to use `testInteger3` without declaring it with either `extern` or `global`.

Assignment in Global Declarations

It is very important to note that assignment in a global declaration (see `testInteger2` in *Script 1* above) only runs if the global was not already defined. This behavior may be somewhat confusing, but it makes a lot of code a lot easier to write.

```
//This line gives us access to the global testInteger1, and sets it equal to 0 if
it did not already exist
global int testInteger1 = 0;

//The following two lines gives us access to the global testInteger2, and then
always sets it to 0.
global int testInteger2;
// ---Snip ---
    testInteger2 = 0;
// ---Snip ---
```

The reason for this is that globals are generally used to make information more persistent, so without this behavior, code would have to be absolutely filled with tests of whether a global variable exists, prior to its declaration, in order to capture desired behaviors.

Operators and `if` Statements

Anyone who has done any programming is familiar with the concept of an If statement. If some condition is true, the following block of code runs. It is highly encouraged to always use Curly Braces for each block of an if statement, but you do not have to.

PART script offers all the comparison operators for values you might expect.

- Greater Than: `>`
- Less Than: `<`
- Greater Than Or Equal To: `>=`
- Less Than Or Equal To: `<=`
- Is Equal To: `==`
- Is Not Equal To: `!=`

Additionally, PART script offers the boolean operators you expect as well.

- And: `&&`
- Or: `||`
- Not: `!`
- Is Equal To: `==`
- Is Not Equal To: `!=`

```
int finalValue;
bool firstBool = false;
bool secondBool = (2 <= 3);

if (firstBool)
{
    //This code will not run
    finalValue = 1;
}
else if (3 == 5)
{
    //This code also will not run
    finalValue = 2;
}
else if (secondBool)
{
    //This code WILL run
    finalValue = 3;
}
else
{
    //This would run if all the above tests failed, but alas they did not
    finalValue = 4;
}

//The final value of finalValue is 3.
```

Math Operations

PART Script has many common mathematical operators, which work on Doubles and Integers.

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^`
- Modulo: `%`

Some further operations were included as static Math functions to round out the list. Functions are invoked by writing the function name, then surrounding the argument(s) with parantheses. As of yet, you cannot define your own functions.

- Natural Logarithm: `Math.Ln(x)`
- Base-10 Logarithm: `Math.Log10(x)`

- Round: `Math.Round(x)`
- Floor: `Math.Floor(x)`
- Ceiling: `Math.Ceiling(x)`
- IsNaN: `Math.IsNaN(x)`

Additionally, a few more functions simplify some statements.

- Max: `Math.Max(x,y)`
- Min: `Math.Min(x,y)`
- Clamp: `Math.Clamp(value,min,max)`

There are also a few inplace operators. These require a variable, but can make some common statements simpler.

- PlusEquals: `variable += x`
- MinusEquals: `variable -= x`
- TimesEquals: `variable *= x`
- DivideEquals: `variable /= x`
- PowerEquals: `variable ^= x`
- ModuloEquals: `variable %= x`
- PostIncrement: `variable++`
- PostDecrement: `variable--`
- PreIncrement: `++variable`
- PreDecrement: `--variable`
- AndEquals: `variable &= x`
- OrEquals: `variable |= x`

Writing `x++` is effectively the same as `x = x + 1`, or `x += 1`.

Functions

Functions are a convenient way to collect some common, reused, or complicated operations in a reusable place. A function declaration always begins with a Type, is followed by a Name, then has its arguments listed, and is followed by the function body itself. A function that doesn't return a value uses the type `void`.

```
int sharedInt;

int Squared(int value)
{
    int valueSquared = value * value;
    return valueSquared;
}

void ModifyInt(int diff)
{
    sharedInt = sharedInt + diff;
}
```

PART Scripts support recursive functions as well, where a function can call itself.

```
//Get the index-th term of the Fibonacci sequence
int FibonacciNumber(int index)
{
    if (index < 2)
        return 1;

    return FibonacciNumber(index - 1) + FibonacciNumber(index - 2);
}
```

Returning Values

Some functions return a value. This is done with a `return` statement.

```
extern int trialCount;
extern int runCount;

bool GetValue()
{
    if (trialCount < 10)
    {
        return true;
    }

    return runCount++ < 3;
}
```

When any functions hits a `return` statement, it stops execution.

Loops

Both **For** and **While** loops see a lot of use in programming, and here they are implemented in the C# Style.

`while` Loops

While loops are the simpler case. They consist of a Condition, an a body. Let's begin with an example to dissect

```
int testInteger = 0;

while (testInteger < 20)
{
    testInteger = 2 * (testInteger + 1);
}
```

Every time the loop runs (including the first) it checks the condition (`testInteger < 20`, in this case). If this condition evaluates to `true`, then the loop runs again. If it evaluates to `false`, the loop ends.

The statement following the loop, a block in this case, is the loop's Body and that's what is executed on each loop.

When this loop terminates, testInteger will have a value of 30.

`for` Loops

Let's begin with an example to dissect

```
int testInteger = 0;

for (int i = 0; i < 100; i++)
{
    testInteger = i;
    if (i == 10)
    {
        break;
    }
}
```

The for loop contains 3 components separated by semicolons.

The first statement is the *Initialization* code. This runs once before the loop starts, and is frequently used to declare and initialize the iterating variable, `i` in this case. Any variable declared in this initialization statement is scoped to just the for loop, so once the loop ends the variable goes out of scope and is destroyed.

The second expression is the *Condition*. Every time the loop runs (including the first) it checks this condition. If this condition evaluates to `true`, then the loop runs again. If it evaluates to `false`, the loop ends.

The third statement is the Incrementer. It runs once after every successful execution of the loop. In simple loops it is frequently used to increment the iterating variable.

The statement following the loop, a block in this case, is the loop's Body and that's what is executed on each loop.

All together, the loop does the following: Initialization -> Condition -> Body -> Incrementer -> Condition -> Body -> Incrementer -> Condition...

Additionally, there are special `break` and `continue` keywords.

Break ends the loop immediately. In the above case, even though the Condition would run until `i` was 100, it will end after i is equal to 10 because of the `break`.

Continue ends the current pass of the loop, jumping straight to the Incrementer step and continuing onward.

`foreach` Loops

Let's begin with an example to dissect

```
List<int> testIntegers = new List<int>() {1, 1, 1, 1, 1, 10, 1};

int total = 0;
foreach (int testInt in testIntegers)
{
    if (testInt == 10)
    {
        continue;
    }
    total += testInt;
}

//total equals 6
```

The foreach loop contains a single statement. This statement begins with a declaration of a new variable to be used inside the loop, like `int testInt`, the keyword `in`, and then a container of some kind.

The statement following the loop, a block in this case, is the loop's Body and that's what is executed on each loop.

Additionally, there are special `break` and `continue` keywords.

Break ends the loop immediately.

Continue ends the current pass of the loop, jumping straight to the next value of the collection.

Data Containers

There are several data containers, each made to suit different needs

`List<T>`

A List is a flexible collection of data. You declare a list much like other data types, expect you specify what type of data it holds inside angle brackets, and construct one with `new`, like this:

```
List<int> anExampleIntList = new List<int>();
anExampleIntList.Add(1);
anExampleIntList.Add(5);
anExampleIntList.Add(7);

int testInteger = anExampleIntList[0];

int searchExample1 = anExampleIntList.IndexOf(6);
int searchExample2 = anExampleIntList.IndexOf(7);
anExampleIntList.Clear();
```

`List<T>` Constructors:

- `List<T>()` Constructs an empty list.

- `List<T>(IEnumerable<T> collection)` constructs a list populated with the elements in `collection`.
- `List<T>(int initialCapacity)` constructs a list with initial capacity of `initialCapacity`. Note, this does not prevent the list from going larger, just allocates additional space.

`List<T>` Properties:

- `int Count` returns the number of items in the list.

`List<T>` Indexer:

- `T [n]` accesses the `n`th item in the list, for reading or writing.

`List<T>` Methods:

- `void Add(x)` appends item `x` to the end of the list.
- `void Insert(n,x)` inserts `x` into the list such that it is the `n`th item.
- `void RemoveAt(n)` remove the `n`th item in the list.
- `bool Contains(x)` returns whether or not the list contains item `x`.
- `int IndexOf(x)` returns the first index where the item `x` appears, or `-1` if it does not.
- `void Clear()` empties out the list.

`Queue<T>`

A Queue is a lot like a list, but it lacks random access to the elements. Instead, you are expected just to `Enqueue` and `Dequeue` items. The first item `Enqueue`d is the the first one `Dequeue`d, in what's called FIFO, or "First In, First Out". In this way, it acts like a real-life Queue:

```
Queue<int> anExampleIntQueue = new Queue<int>();
anExampleIntQueue.Enqueue(1);
anExampleIntQueue.Enqueue(5);
anExampleIntQueue.Enqueue(7);

//testInteger will be set to 1
int testInteger = anExampleIntQueue.Dequeue();

anExampleIntQueue.Clear();
```

`Queue<T>` Constructors:

- `Queue<T>()` constructs an empty queue.
- `Queue<T>(IEnumerable<T> collection)` constructs a queue populated with the elements in `collection`.
- `Queue<T>(int initialCapacity)` constructs a queue with initial capacity of `initialCapacity`. Note, this does not prevent the queue from going larger, just allocates additional space.

`Queue<T>` Properties:

- `int Count` returns the number of items in the queue.

`Queue<T>` Methods:

- `void Enqueue(x)` appends item `x` to the end of the queue.
- `T Dequeue()` returns the item at the front of the queue, and removes it from the queue.
- `T Peek()` returns the item at the front of the queue, but leaves it in the queue.
- `bool Contains(x)` returns whether or not the queue contains item `x`.
- `void Clear()` empties out the queue.

`Stack<T>`

A Stack is like an inverted Queue. Instead, you `Push` and `Pop` items. The most recent item to be `Push`ed is the the next one `Pop`ed, in what's called LIFO, or "Last In, First Out". In this way, it acts like a real-life stack of cards:

```
Stack<int> anExampleIntStack = new Stack<int>();
anExampleIntStack.Push(1);
anExampleIntStack.Push(5);
anExampleIntStack.Push(7);

//testInteger will be set to 7
int testInteger = anExampleIntStack.Pop();

anExampleIntStack.Clear();
```

`Stack<T>` Constructors:

- `Stack<T>()` constructs an empty stack.
- `Stack<T>(IEnumerable<T> collection)` constructs a stack populated with the elements in `collection`.
- `Stack<T>(int initialCapacity)` constructs a stack with initial capacity of `initialCapacity`. Note, this does not prevent the stack from going larger, just allocates additional space.

`Stack<T>` Properties:

- `int Count` returns the number of items in the stack.

`Stack<T>` Methods:

- `void Push(x)` adds item `x` to the top of the stack.
- `T Pop()` returns the item at the top of the stack, and removes it from the stack.
- `T Peek()` returns the item at the top of the stack, but leaves it in the stack.
- `bool Contains(x)` returns whether or not the stack contains item `x`.
- `void Clear()` empties out the stack.

`RingBuffer<T>`

RingBuffers have a limited capacity. Adding items past this capacity bumps the oldest value out of the list.

`RingBuffer<T>` Constructor:

- `RingBuffer<T>(int bufferSize)` constructs an empty RingBuffer with a capacity of `bufferSize`.

- `RingBuffer<T>(IEnumerable<T> values)` constructs a RingBuffer filled with the elements of `values`, with a capacity equal to the number of elements in `values`.
- `RingBuffer<T>(IEnumerable<T> values, int bufferSize)` constructs a RingBuffer filled with the elements of `values`, with a capacity of `bufferSize`.

`RingBuffer<T>` Properties:

- `int Count` returns the number of items currently in the ring buffer.
- `int Size` returns the total capacity of the ring buffer.
- `T Head` returns the item at the Head of the ring buffer, the one most recently added.
- `T Tail` returns the item at the Tail of the ring buffer, the oldest one added that still remains.

`RingBuffer<T>` Indexer:

- `T [n]` accesses the `n`th item in the ring buffer, for reading or writing.

`RingBuffer<T>` Methods:

- `bool Contains(x)` returns whether or not the ring buffer contains item `x`.
- `int CountElement(x)` returns the number of times `x` appears in the ring buffer.
- `int GetIndex(x)` returns the first index where the item `x` appears, or `-1` if it does not.
- `bool Remove(x)` removes item `x` from the ring buffer if it exists, and returns whether or not the operation removed an item.
- `void RemoveAt(n)` removes item at index `n` from the ring buffer.
- `void Resize(n)` resizes the ringbuffer to have a total capacity of `n`.
- `void Add(x)` adds item `x` to the head of the ring buffer.
- `void Push(x)` adds item `x` to the head of the ring buffer.
- `T Pop()` returns the item at the head of the ring buffer, and removes it from the ring buffer.
- `T PopBack()` returns the item at the tail of the ring buffer, and removes it from the ring buffer.
- `T PeekHead()` returns the item at the head of the ring buffer, but leaves it in the ring buffer.
- `T PeekTail()` returns the item at the tail of the ring buffer, but leaves it in the ring buffer.
- `void Clear()` empties out the ring buffer.

`DepletableList<T>`

DepletableLists are like Queues that get filled with elements, but non-destructively Dequeued via `PopNext()`. Using `PopNext()`, each element is returned in sequence, until `Reset()` is called (or the list is depleted *and* `AutoRefill` is set to `true`), where the DepletableList resets to its fully populated state.

`DepletableList<T>` Constructors:

- `DepletableList<T>()` constructs an empty depleteable bag and AutoRefill set to false.
- `DepletableList<T>(IEnumerable<T> values)` constructs a depleteable bag populated with the items in the list and with AutoRefill set to false.
- `DepletableList<T>(IEnumerable<T> values, bool autoRefill)` constructs a depleteable bag populated with the items in the list and with AutoRefill set the specified value.

`DepletableList<T>` Properties:

- `int Count` returns the number of active items currently in the container.

- `int TotalCount` returns the number of items (active and inactive) currently in the container.
- `bool AutoRefill` returns (and sets) whether or not the container marks all items as active when all items are depleted and a draw is attempted.

`DepletableList<T>` Methods:

- `bool Contains(x)` returns whether or not the container contains item `x`.
- `void Add(x)` adds item `x` to the end of the container, as active.
- `bool Remove(x)` removes item `x` from the container if it exists, and returns whether or not the operation removed an item.
- `bool DepleteValue(x)` depletes the first active instance of item `x` in the container if it exists, and returns whether or not the operation depleted an item.
- `bool DepleteAllValue(x)` depletes the all active instances of item `x` in the container if any exist, and returns whether or not the operation depleted any items.
- `bool RefreshValue(x)` sets as active the first depleted instance of item `x` in the container if it exists, and returns whether or not the operation set an item active.
- `bool RefreshAllValue(x)` sets as active all depleted instances of item `x` in the container if any exist, and returns whether or not the operation set any items active.
- `T PopNext()` returns the next active item in the container, and marks it depleted.
- `void Reset()` marks all items as active.
- `void Clear()` empties out the container.

`DepletableBag<T>`

DepletableBags are like DepleteableLists, but are sampled randomly with calls to `PopNext()`. Using `PopNext()`, elements are returned in a random sequence, until `Reset()` is called (or the list is depleted *and* `AutoRefill` is set to `true`), where the DepletableBag resets to its fully populated state. This is ideal for counterbalancing, or *Sampling Without Replacement*.

`DepletableBag<T>` Constructors:

- `DepletableBag<T>()` constructs an empty depleteable bag with its own randomizer and AutoRefill set to false.
- `DepletableBag<T>(Random randomizer)` constructs an empty depleteable bag that uses the specified Random object to control randomization and AutoRefill set to false.
- `DepletableBag<T>(IEnumerable<T> values)` constructs a depleteable bag populated with the items in the list that uses its own randomizer and AutoRefill set to false.
- `DepletableBag<T>(IEnumerable<T> values, bool autoRefill)` constructs a depleteable bag populated with the items in the list that uses its own randomizer and AutoRefill set the specified value.
- `DepletableBag<T>(IEnumerable<T> values, bool autoRefill, Random randomizer)` constructs a depleteable bag populated with the items in the list that uses the specified Random object to control randomization and AutoRefill set the specified value.

`DepletableBag<T>` Properties:

- `int Count` returns the number of active items currently in the container.
- `int TotalCount` returns the number of items (active and inactive) currently in the container.
- `bool AutoRefill` returns (and sets) whether or not the container marks all items as active when all items are depleted and a draw is attempted.

`DepletableBag<T>` Methods:

- `bool Contains(x)` returns whether or not the container contains item `x`.
- `void Add(x)` adds item `x` to the end of the container, as active.
- `bool Remove(x)` removes item `x` from the container if it exists, and returns whether or not the operation removed an item.
- `bool DepleteValue(x)` depletes the first active instance of item `x` in the container if it exists, and returns whether or not the operation depleted an item.
- `bool DepleteAllValue(x)` depletes the all active instances of item `x` in the container if any exist, and returns whether or not the operation depleted any items.
- `bool RefreshValue(x)` sets as active the first depleted instance of item `x` in the container if it exists, and returns whether or not the operation set an item active.
- `bool RefreshAllValue(x)` sets as active all depleted instances of item `x` in the container if any exist, and returns whether or not the operation set any items active.
- `T PopNext()` returns a random active item in the container, and marks it depleted.
- `void Reset()` marks all items as active.
- `void Clear()` empties out the container.

`Dictionary<TKey,TValue>`

Dictionaries store values under keys of the specified type. They are also sometimes known as HashTables. You add elements with `Add`, but you need to specify both the value and the key. You access elements with the indexer using the appropriate key.

`Dictionary<TKey,TValue>` Constructors:

- `Dictionary<TKey,TValue>()` constructs an empty dictionary.

`Dictionary<TKey,TValue>` Properties:

- `int Count` returns the number of values currently in the dictionary.
- `IEnumerable<TKey> Keys` returns an enumeration of all of the Keys.
- `IEnumerable<TValue> Values` returns an enumeration of all of the Values, in the same order as the Keys.

`Dictionary<TKey,TValue>` Indexer:

- `TValue [x]` accesses the value stored under key `x`, for reading or writing.

`Dictionary<TKey,TValue>` Methods:

- `bool ContainsKey(x)` returns whether or not the container contains an item stored under key `x`.
- `bool ContainsValue(y)` returns whether or not the container contains an item `y`.
- `void Add(x,y)` adds item `y` to the container under key `x`.
- `bool Remove(x)` removes item stored under key `x` from the container if it exists, and returns whether or not the operation removed an item.
- `void Clear()` empties out the container.

`HashSet<T>`

HashSets are unordered collections of values. A value either is or is not in the set, but it is not built for accessing the values as much as it is for testing if a value exists in the set.

`HashSet<T>` Constructors:

- `HashSet<T>()` constructs an empty HashSet.
- `HashSet<T>(IEnumerable<T> values)` constructs a HashSet filled with the unique elements of `values`.

`HashSet<T>` Properties:

- `int Count` returns the number of values currently in the HashSet.

`HashSet<T>` Methods:

- `bool Add(x)` adds item `x` to the HashSet if it was not already in the collection, and returns whether the addition was successful.
- `bool Remove(x)` removes item `x` from the HashSet if it exists, and returns whether or not the operation removed an item.
- `bool Contains(x)` returns whether or not the HashSet contains item `x`.
- `void Clear()` empties out the HashSet.

Other Container Features

All of the containers are IEnumerable, so they can be used in the construction of one another. Additionally, they all support the Collection Initializer Syntax. Creating a collection with an initializer list calls the apporpriate `Add` method (`Add`, `Enqueue`, or `Push`) on each item in the list (see the following example).

```
int testInt = 5;
List<int> exampleListA = new List<int>() { 1, 2, 3, 4, testInt, 6 };
List<int> exampleListB = new List<int>(exampleListA);

//exampleListA and exampleListB contain numbers 1 through 6

exampleListA.Clear();

//Now exampleListA is empty and exampleListB contains numbers 1 through 6

Stack<int> exampleStackA = new Stack<int>(exampleListA);
Stack<int> exampleStackB = new Stack<int>(exampleListB);
Stack<int> exampleStackC = new Stack<int>() {1, 2, 3, 4, 5, 6};

//Now exampleStackA is empty and exampleStackB and exampleStackC contain numbers 1
through 6
```

Random

`Random` is a class that allows you to generate random numbers. If you specify a seed, then the sequence of numbers generated will be completely reproducible (by using the same seed in the future). If you do not specify a seed, then the sequence of numbers will be unique every time.

`Random` Constructors:

- `Random()` constructs a new instance with a random seed.
- `Random(int seed)` constructs a new instance with the specified seed.

`Random` Methods:

- `int Next()` returns a completely random integer between the max and min value for a signed 32-bit integer.
- `int Next(int maxValue)` returns a random integer between 0 (Inclusive) and `maxValue` (Exclusive).
- `int Next(int minValue, int maxValue)` returns a random integer between `minValue` (Inclusive) and `maxValue` (Exclusive).
- `double NextDouble()` returns a random double between 0.0 and 1.0.

Persistent User Data

There exist a few static `User` functions to allow you to store persistent data for a user. For the functions used to retrieve data, `key` refers to the string that the values are stored under, and the functions return a value of the named type.

- `int User.GetInt(key)`
- `double User.GetDouble(key)`
- `bool User.GetBool(key)`
- `string User.GetString(key)`
- `List<T> User.GetList<T>(key)`

Note: For all of the above, you can optionally specify a default value you would like the function to return if the value doesn't exist. In that case, you would call it like: `User.GetInt(key,defaultValue)`
Note: `User.GetList<T>(key)` requires that you specify the type of item stored in the list. So, if you were retrieving a `List<int>` from the user data, you would write something like:

```
List<int> trialList = User.GetList<int>("Last Session Trial List");
```

And the functions used to set the data are as follows

- `void User.SetInt(key,value)`
- `void User.SetDouble(key,value)`
- `void User.SetBool(key,value)`
- `void User.SetString(key,value)`
- `void User.SetList(key,value)`

And the following help with management and cleanup

- `bool User.HasData(key)`
- `void User.ClearData(key)`

Reports

Reports are meant to be a high-level summary of the performance of participants across different assessments. To add any value to the report for the current battery, use the `User.AddToReport(header,value)` function.

- `void User.AddToReport(header, value)` adds the string `value` to the report under string `header`

An example common use case follows, where the `STMThreshold` is converted to a string *implicitly* by concatenating it with the `" db"` string. To do so without applying any additional text, one could also just concatenate it with an empty string, like `STMThreshold + ""`. This is necessary because the arguments to the function are `string`s.

```
extern double STMThreshold;

void Run()
{
    User.AddToReport("STM", STMThreshold + " dB");
}
```