Generator Breakdown

Still having problems understanding generators, huh?

Don't worry - its easy stuff.

The basic syntax is:

```
<expression> for <declarations> in
<iterator> <conditional expression>
```

In practice, it generally looks like this:

```
<local variable> = <expression> for
<declarations> in <iterator> <conditional
expression>
```

Let's start off easy:

After the generator expression is run, <local variable> will become an enumerable object that you can iterate through, like an array, list, or collection. The contents of this enumerable object will be a collection of <expression>s.

<expression>s are variables that are put into <local variable> if <conditional expression> returns true. This means means that by using a <conditional expression> in a generator, you can decide which objects should be added to the enumerable <local variable>. <expressions> can be manipulated further: for instance, this syntax is valid:

```
<local variable> =
MethodToMutilateExpression(<expression>) for
<declarations> in <iterator> <conditional
expression>
```

In this instance, the return value of MethodToMutilateExpression will be added to the enumerable <local variable>. This makes generators useful to filter the input that MethodToMutilateExpression() will be called with.

"for <declarations> in <iterator>" is your typical for loop: "for item in collection," in otherwords.

Here's a simple working example of how generators work.

```
#An array of integers
crazyArray = (0, 1, 2, 3, 4, 5)
#Shazbot. We only want integers that are
greater than 3!

#We want an enumerable object composed only
of integers > 3.
validIntegers = value for value in
crazyArray if value > 3

#Here's some special Boo magic: turn
'ValidIntegers' back into an array!
#We can do this because all objects within
validIntegers are really integers.
magic = array(int, validIntegers)
print magic #print the type of this object
print join(magic) #turn this array into a
readable string.
```

Here's the output:

```
System.Int32[]
4 5
```