

BFOIT ICSI

BFOIT - Introduction to Computer Programming

Background

- [Preface](#)
- [Instructor Notes](#)
- [Table of Contents](#)
- [What's Computer Programming?](#)

jLogo Programming

- [Commanding a Turtle](#)
- [Pseudocode](#)
- [Adding New Commands](#)
- [Iteration & Animation](#)
- [Hierarchical Structure](#)
- [Procedure Inputs](#)
- [Operators & Expressions](#)
- [Defining Operators](#)
- [Words & Sentences](#)
- [User Interface Events](#)
- [What If? \(Predicates\)](#)
- [Recursion](#)
- [Local Variables](#)
- [Global Variables](#)
- [Word/Sentence Iteration](#)
- [Mastermind Project](#)
- [Turtles As Actors](#)
- [Arrays](#)
- [File Input/Output](#)

Java

- [A Java Program](#)
- [What's a Class?](#)
- [Extending Existing Classes](#)
- [Types](#)
- [Turtle Graphics](#)
- [Control Flow](#)
- [User Interface Events](#)

Appendices

- [Jargon](#)
- [What Is TG?](#)
- [TG Directives](#)
- [jLogo Primitives](#)
- [TG Editor](#)
- [Java Tables](#)
- [Example Programs](#)
- [Installation Notes](#)

Updates

- [December 13, 2008](#)
- [January 6, 2012](#)
- [March 15, 2013](#)
- [January 20, 2014](#)
- [February 13, 2014](#)
- [July 29, 2014](#)

Lastly


```

to main
  hideturtle
  setpencolor blue
  waves -240 20
  waves -130 -15
  setpencolor forest
  fish 100 -100
  fish 50 -150
  ...
end

```

Defining the procedures **blue** and **forest** and then using them in appropriate places makes the program much easier to read. Trust me, this is good...

The symbolic constants for headings for the turtle can be used similarly. **north**, **east**, etc... procedures can be used as inputs to the **setheading** command. Get into the habit of starting all of your programs with *symbolic constants* - part of the vocabulary for the story you are about to write. **Yes, think of the process of writing a program as similar to writing anything that is descriptive.**

Creating Your Own Operators

The key to the above procedure definitions is the command, **output**. When a Logo interpreter performs an **output** command, it exits the procedure containing it, producing whatever **output**'s input is - *as the procedure's output*. This can be confusing at first. **output is not an operator**. Figure 9.1 shows what happens if you try to put it where an operator is required.



Figure 9.1

Ok, let's move on and write some more interesting procedures which are operators.

In the previous lesson there were examples of arithmetic expressions that computed and displayed the circumference and the area of a circle. Here is one of the procedures.

```

to printCircleCircum :radius
  println product 2 (product 3.14159 :radius)
end

```

It would be more useful to separate the **println** functionality from the computation piece. It's simple... Just replace **println** with **output**.

```

to circleCircumference :radius
  output product 2 (product 3.14159 :radius)
end

```

Notice that I changed the name of the procedure. You always want the name of the procedure to reflect what it does. This is how we achieve our abstraction.

This new procedure, now an *operator*, is simple to use with the **println** command, e.g., here is an example of its use in the CommandCenter.

```

? println circleCircumference 4
25.13272
?

```

Use the following TG applet to try it out for yourself. Type the **circleCircumference** definition into the Editor, then invoke it a few times in the CommandCenter to see what you get.

alt="Your browser understands the <APPLET> tag but isn't running the applet, for some reason." Your browser is completely ignoring the <APPLET> tag! TG Programming Environment Applet

[If this applet is broken and you are using Chrome, click here.](#)

Practice: A Couple More Operators For You To Write

You really don't learn about something deeply by reading about it or hearing someone lecture about it - you learn

by *trying to do it*. Here are a few exercises that you can play with.

1. [In the previous lesson you wrote printCircleArea](#). Write a procedure named **circleArea** with an input for the radius of the circle. It should produce an output that is the area of the circle. Test it with the inputs in Table 9.1 and make sure you get good answers.

Input	Area
2	12.56636
3	28.27431
5	78.53975
11	380.13239

Table 9.1

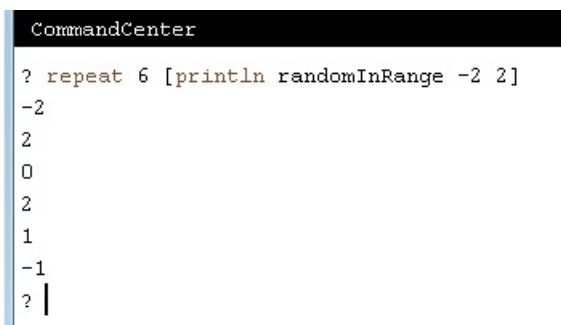
2. Part of the arithmetic expression you wrote to compute the area of a circle involved squaring a number. Squaring a number is a common thing to do. Write a procedure named **square** with an input **:number** that produces number-squared. There is also a constant value in the expression, **PI**, which should be defined as a *symbolic constant* - define the procedure **PI** which outputs 3.14159. Redefine your **circleArea** using these new procedures. Once again, test it...

[After you've completed writing these procedures, check what you came up with here.](#)

Project: Random In Range

OK, now for an operator that would have been nice to have in one of the programs we wrote in the last lesson, [RandomBoxes](#). To refresh your memory, we painted a bunch of solid, colored boxes at random locations on the graphics canvas. We needed a couple of random numbers in ranges other than the standard 0...x-1 provided by the **random** x operator. We needed random numbers in a range determined by the height and width of the graphics canvas. As an example, if the **canvaswidth** operator outputs 600, we needed random numbers in the range of -300 ... 300-boxWidth.

I want a new operator, let's call it **randomInRange**, which produces an output, a number that's in any range of numbers I provide as inputs. As an example, for the invocation "randomInRange -50 50" I want an output that is random and is greater-than or equal-to -50 AND is less-than or equal-to 50. Figure 9.2 demonstrates a working version producing random numbers in the range of -2 through 2.



```
CommandCenter
? repeat 6 [println randomInRange -2 2]
-2
2
0
2
1
-1
? |
```

Figure 9.2

Ok, either go off and write it or if you need a little help to get started, here's a skeleton.

```
to randomInRange :min :max
  ; output <random number gtr-or-eql :min and less-or-eql :max>
end
```

At this point, don't hesitate to play around in the TG applet to figure out what you need to do. Follow the steps we've been using to write all of our programs.

1. Understanding the Problem (figure out what you know, and what you don't know)
2. Devising a Plan (write a pseudo code description of what to do, draw a plumbing diagram or diagrams to visualize what you will do)
3. Carrying out the Plan - type in your source code and test it; does it work? If not, verify your code matches what you developed in step 2 and review what you came up with in steps 1 and 2.

Don't read further until you've at least made some attempt at writing the randomInRange procedure.

First hint...

If you think about it, your output needs to be at least what the input **:min** is given when **randomInRange** is

invoked. Since the smallest value returned by **random** is 0, you are going to need to add **:min** to the output from **random**. So, our pseudocode now looks like:

```
;output a random number that is
;greater than or equal to :min and
;less than or equal to :max
to randomInRange :min :max
  ; output sum :min random <magnitude of range of numbers>
end
```

Don't read further until you've made an attempt to complete this procedure and experimented with it

Second hint...

At this point, I suggest that you write an operator named **magnitude**. Here's the expanded pseudo code:

```
;output the absolute value of
;the difference of two inputs
to magnitude :min :max
  ; output <magnitude of range of numbers>
end

;output a random number that is
;greater than or equal to :min and
;less than or equal to :max
to randomInRange :min :max
  output sum :min (random magnitude :min :max)
end
```

OK... finish the procedure.

Use **repeat** to test your code. If it doesn't work, help is on the way - read the next section. If your code works, congratulations! Now read the next section because it's a really cool feature you are sure to use in the future.

TRACE - A Debugging Tool

"Failure is an integral part of success," Mead says. "You learn from every one of your failures. I used to tell students, 'You've got to listen to the silicon. It's trying to tell you something.'"

If you build something or do something and it doesn't work out, he says you can curse and swear at it. Or, you can learn from it.

"The physical world is perfectly willing to share with you how it works. If you listen. But, if you have your mind made up, you can go for years and not hear it," Mead says.

from an article in Investors Business Daily
May 13, 2003

Time for you to learn more about [tracing](#) in TG. It can help you understand what's happening when your program is being executed and not doing what you expect. Remember, **trace** is a [directive](#) you enter into the CommandCenter. it's similar to a command, but can't be placed into the body of a procedure. It directs TG to print out very useful stuff as your program is executed.

So, once you have your **magnitude** and **randomInRange** procedures entered into the Editor, switch keyboard focus to the CommandCenter and type:

```
? trace magnitude
? println randomInRange -4 4
```

Since TG has been directed to **trace** the procedure **magnitude**, what you should see is something like:

```
? println randomInRange -4 4
Entering magnitude -4 4
Exiting magnitude, output: 8
-3
?
```

TG let you know that **magnitude** was invoked with the inputs: **-4** and **4**. It's body was executed and its output was **8**.

Well, with this information, you can see that there is a problem with the code I gave you. There's a [bug](#) in it. **randomInRange -4 4** will never output a **4** (the value provided for the **:max** input in our trial).

random 8 outputs a number in the range **0 - 7**. To get an output of **4** we need to give **random** an input equal to the output of **magnitude plus 1**.

Fix the bug (if you hadn't discovered it and fixed it on your own)...

Finally, modify your program from the last lesson that draws boxes at random locations so that it uses **randomInRange**. Also add symbolic constants where appropriate... Use them to make your program more *readable*.

You may want to use an operator that TG's interpreter understands that we have not talked about yet. You may not need it, but I used it. Table 9.1 gives you the scoop on **minus** which simply takes a number and negates it, just like multiplying the number by -1.

Command	Inputs	Description
MINUS	number	Outputs the negative of number

Table 9.1

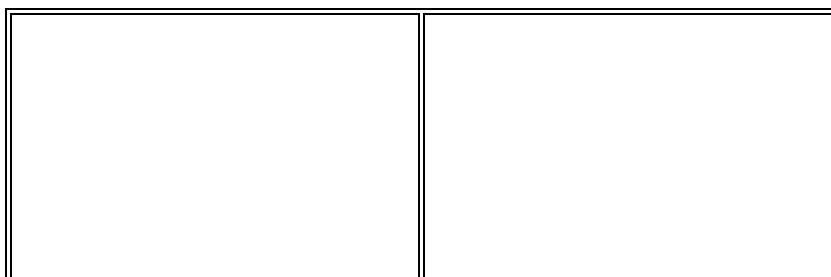
Project: A Grid Toolkit

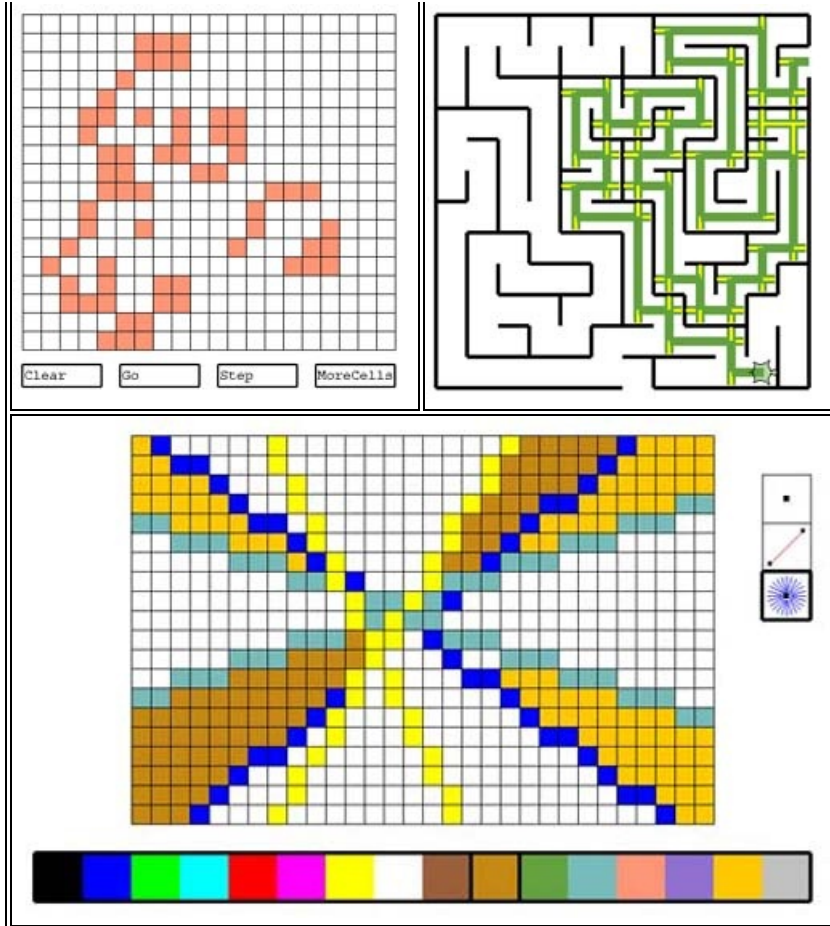
The reason that I have had you drawing axes and grids in many of the previous lessons is that they are very common in programs. Here are some windows which I captured; they are, top-to-bottom, left-to-right:

- [Programming a Robot in Machine Language](#)
- [Mastermind Game](#)
- [Sudoku Puzzle Assistant](#)

The screenshot displays two windows from a 'Grid Toolkit' application. The upper window features a coordinate grid with axes labeled from 00 to 37. A small robot icon is positioned in the center of the grid. Below the grid are controls for 'RUN' and 'STEP', and a progress indicator showing 'PC' and 'RA' with corresponding bar graphs. The lower window is divided into two sections. The left section contains a color palette with a vertical bar of colors (red, yellow, green, blue, purple) and a 4x4 grid of colored squares. Below this are buttons for 'Clear Color', 'Guess', and 'New Game', along with a 'You Win' message and an 'Allow Dups' checkbox. The right section displays a 9x9 Sudoku puzzle grid with some numbers filled in and a green '6' highlighted in the center. Below the grid are buttons for 'Guess', 'Hint', and 'Help'.

- [Game of Life](#)
- [Turtle Solving Maze](#)
- [Paint Program \(Big Pixels\)](#)





Each of these programs contains at least one graphical object that is grid-like. I am going to take our use of *abstraction* to another level. We are going to write a bunch of procedures that provide stuff we will need in many programs. We are going to get started creating a *GridToolkit*.

A Grid Toolkit Contract

The first thing I'm going to do is write the rules for using the GridToolkit. These rules are going to be its *contract*. We will follow the rules when we write the procedures in the toolkit. And... we will follow the rules when we use the procedures in the toolkit in our programs. I talked a bit about contracts, as they applied to a single procedure, [when I introduced defining you own procedures](#). Our first GridToolkit rules will be

1. Definition: our *grid* is a bunch of squares, all of equal size, arranged in *rows*, each with an equal number of *columns*
2. Definition: the squares making up our grid will be called *cells*
3. All procedures in the toolkit will have names starting with "grid"
4. A cell can be identified in one of two ways:
 - by its row and column numbers. Rows are ordered from top to bottom, with the top row numbered zero. Columns are ordered left to right with the leftmost column numbered zero.

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

- by its index. the index (number) of the top-left cell is zero and then indices increase left-to-right and then top-to-bottom.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

The Grid Toolkit Source Code

Given these rules, here are the procedures that make up our first pass at a *GridToolkit*.

```
;Symbolic constants for inputs to setpencolor
```

```

to black
  output 0
end
to white
  output 7
end

;color for a cell's background
;SYMBOLIC CONSTANT
to gridCellColor
  output white
end

;color of grid's frame
;SYMBOLIC CONSTANT
to gridFrameColor
  output black
end

;size of the sides of a grid cell
;SYMBOLIC CONSTANT
to gridCellSize
  output 40
end

;one-half the size of a side of a grid cell
to gridHalfCellSize
  output quotient gridCellSize 2
end

;number of columns in the grid
;SYMBOLIC CONSTANT
to gridNumCol
  output 8
end

;number of rows in the grid
;SYMBOLIC CONSTANT
to gridNumRow
  output 5
end

;number of cells in the grid
to gridNumCells
  output product gridNumCol gridNumRow
end

;height of grid in turtle steps
to gridHeight
  output product gridNumRow gridCellSize
end

;width of grid in turtle steps
to gridWidth
  output product gridNumCol gridCellSize
end

;X coordinate for left side of the grid
;SYMBOLIC CONSTANT
to gridLeftX
  output -160
end

;Y coordinate for bottom of the grid
;SYMBOLIC CONSTANT
to gridBottomY
  output -100
end

;Y coordinate for top of the grid
;SYMBOLIC CONSTANT
to gridTopY
  output sum gridBottomY gridHeight
end

;move turtle to top-left corner of grid
to gridGotoTopLeft
  penup
  setxy gridLeftX gridTopY
end

;move turtle to bottom-left corner of a cell
;specified by the index :idx
to gridGotoCell :idx
  gridGotoTopLeft
  setheading 180
  forward product gridCellSize (int quotient :idx gridNumCol)
  forward gridCellSize
  setheading 90
  forward product gridCellSize (remainder :idx gridNumCol)
end

```



```

;move turtle to the center of the cell
;specified by the index :idx
to gridGotoCellCtr :idx
  gridGotoCell :idx
  setheading 90 forward gridHafCellSiz
  left 90 forward gridHafCellSiz
end

;draw/redraw the cell specified by the index :idx
;the :color input specifies the cell's new background color
to gridCellFill :idx :color
  gridGotoCellCtr :idx
  setpencolor gridCellSize setpencolor :color
  setheading 0 pendown
  forward gridHafCellSiz back gridCellSize forward gridHafCellSiz
  penup forward gridHafCellSiz left 90 forward gridHafCellSiz
  setpencolor 1 setpencolor gridFrameColor
  setheading 90 pendown
  repeat 4 [forward gridCellSize right 90]
end

;draw the grid
to gridPaint
  gridGotoTopLeft
  setheading 180 forward quotient gridHeight 2 setheading 90
  setpencolor gridHeight setpencolor gridCellColor
  pendown forward gridWidth
  gridGotoTopLeft
  setpencolor 1 setpencolor gridFrameColor
  setheading 180 pendown
  repeat gridNumRow [fd gridCellSize lt 90 fd gridWidth bk gridWidth rt 90]
  gridGotoTopLeft
  setheading 90 pendown
  repeat gridNumCol [fd gridCellSize rt 90 fd gridHeight bk gridHeight lt 90]
end

```

Checkout the Grid Toolkit

If you have the TG programming environment on your computer, copy & paste the GridToolkit source code into TG's Editor so that you can play with it. If you want to use [the TG applet on this web page](#) to play, use the "**loadcode GridToolkit_9.jlogo**" directive in the CommandCenter to get the source code in the Editor.

It has initial default values in place in all of the procedures acting as [symbolic constants](#). These will be changed as necessary in programs that you write that includes the *GridToolkit*. But, with the default values in place, you can display a grid by invoking a single procedure: **gridPaint**. Then you can paint individual cells by invoking **gridCellFill** with two inputs, the cell's index and the color number it is to be filled with. Add the following code after the GridToolkit stuff in the Editor or type it into the CommandCenter.

```

gridPaint
gridCellFill 0 1
gridCellFill (difference gridNumCells 1) 4

```

Then try filling random cells with random colors; here's an instruction that does this.

```

repeat gridNumCells [gridCellFill random gridNumCells random 32]

```

Play with the code... change some symbolic constants like the size of the cells, the number of rows and or columns...

Why the Gift?

So, why did I give you all of the source code for the *GridToolkit* without making you work for it?

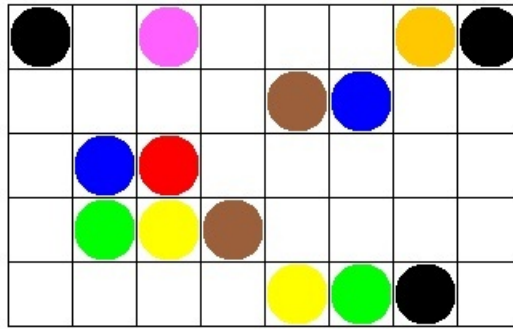
Answer: **you will learn a lot by reading someone else's source code.**

Long ago, when I was convinced that Logo was a much better language to use to introduce programming concepts than Java, the first thing I did was to read a lot of Logo source code. Fortunately for me, there is a lot of it available. Brian Harvey's *Computer Science, Logo Style* books, [available free on-line](#), are a great introduction. See [the ItP acknowledgements page](#) for additional sources of great Logo programs.

Now it's your turn to read some code. Read through the *GridToolkit* source code. Use it to complete the following project.

Draw Balls in the Cells

Write a procedure that draws a colored ball in the center of a specified cell. Then use it and procedures in the *GridToolkit* to write a program which draws random colored balls in random cells. Here is an example of what my program painted in the graphics canvas.

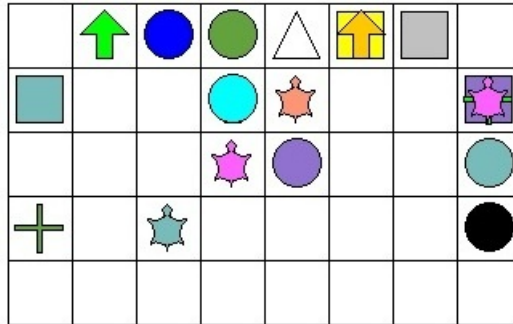


I used a **repeat** command to paint 20 randomly colored balls into 20 random cells.

```
repeat 20 [ gridGotoCellCtr random gridNumCells drawBall ballSize random 16 ]
```

Stamp Turtle Shapes in the Cells

Write a procedure that sets the shape of the turtle to a random one. Read about the **stamp** command using the **help** directive in the CommandCenter and then use it to paint shapes in random cells. Here is an example of what my program painted in the graphics canvas.



Two New Built-in Operators - Why?

In the *GridToolkit*, I used two built-in operators that you have not seen before. What are they? Why did I need to use them?

[After you've thought about it for a while and have some sort of answers, check what you came up with here.](#)

Why Count Starting With Zero?

In *GridToolkit*, rows and columns were numbered starting with zero instead of one. The index that is used to identify a particular cell also started with the first cell numbered zero. This convention has been in place for decades in the world of computer programming.

The reason is that this simplifies the code. Can you find the code in our GridWorld which is simpler than it would have to be if we started numbering the index at one instead of zero? How would the code need to be changed if the base was one instead of zero?

[After you've thought about it for a while and have some sort of an answer, check what you came up with here.](#)

Summary

We've made another quantum jump in how we can reduce the complexity of the programs we write. You now know how to define your own *operators*, procedures which output values. Symbolic constants, a very simple form of a user-defined operator, can make your programs much more readable. And, binding meaningful names to complex expressions, is just one more use of abstraction, an approach to writing programs that are much more easily understood. And, the side benefit is that a program that is easy to read, that's easily understood. is that it will probably do what you want, what you intended it to do.

Go to the [Table of Contents](#)

On to [Words & Sentences](#)

Feel free to e-mail comments/questions to bfoitGuy <at> gmail <dot> com



This work (BFOIT: Introduction to Computer Programming, by [Guy M. Haas](#)),
identified by [Berkeley Foundation for Opportunities in IT \(BFOIT\)](#),
is free of known copyright restrictions.