

# KOJO

## An Introduction

By Lalit and Vibha Pant

play hard, learn well...

[www.kogics.net](http://www.kogics.net)

## Front Matter

This is the first in a series of ebooks about using Kojo to play with Computer Programming, Art, and Maths. This book:

- Introduces the Kojo Environment.
- Makes you familiar with the different features of the Environment within the context of a guided activity.
- Takes you on a short adventure trail to get you started on the road to Kojo (and Programming/Maths/Art) mastery!

Future ebooks will cover the following topics:

- Learning to program.
- Fun Activities with Kojo.
- Playing with Geometry.
- Experimenting with Algebra.

This book is suitable for eight and ninth graders, but seventh and sixth graders should also be able to understand a lot of the material in the book.

Book version: Mar 16, 2010

Please download and install the latest version of Kojo for use with this book.

The Kojo download page is located at:

<http://www.kogics.net/sf:kojo-download>



*Copyright (C) 2010 Lalit Pant and Vibha Pant.*

*Distributed under the Creative Commons Attribution-ShareAlike License.*

*More information about Kojo is available at <http://www.kogics.net/sf:kojo>*

## Table of Contents

1 What is Kojo.....	5
1.1 Why work with Kojo?.....	5
1.2 Painting with Kojo.....	6
1.2.1 Rangoli.....	6
1.2.2 Fractal Tree.....	7
1.3 Animation with Kojo – Sine of an angle.....	8
1.4 Geometry with Kojo – Parallel Lines, and the angles of a triangle.....	9
2 The Kojo User Interface.....	10
2.1 Context Sensitive Actions.....	11
Exercises.....	11
2.2 Panning and Zooming.....	12
Exercises.....	13
2.3 Window Management.....	13
2.3.1 Maximizing Windows.....	14
Exercises.....	14
2.3.2 Minimizing Windows.....	14
Exercises.....	14
2.3.3 Moving and Docking Windows.....	15
3 A Guided tour of Kojo.....	16
3.1 The Programming Language.....	16
3.1.1 Kojo Commands.....	17
3.1.2 Flow control with repeat.....	17
3.2 Taking Kojo for a spin.....	19
3.2.1 Making a simple Shape.....	19
3.2.2 Accessing History.....	20
3.2.3 Playing with the Script Text.....	22
Format.....	22
Selecting Text.....	24
Exercises.....	24
Copy & Paste.....	24
Cut & Paste.....	25
Run.....	25
3.2.4 Recovering from Errors.....	26
3.2.5 Refining your programs.....	27

Exercises.....	29
3.2.6 Saving and Loading your work.....	29
Exercises.....	30
3.3 Checklist.....	30
4 A Kojo Adventure Trail.....	31
4.1 Activity 1 - A Triangle.....	32
Commands to be used.....	32
Hints.....	32
4.2 Activity 2 – A Face.....	33
Commands to be used.....	33
Hints.....	33
4.3 Activity 3 - Turning Squares.....	34
Commands to be used.....	34
Hints.....	34
4.4 Activity 4 - Triangles.....	35
Commands to be used.....	35
Hints.....	35
4.5 Activity 5 - Staircase.....	36
Commands to be used.....	36
Hints.....	36
4.6 Activity 6 – Five Squares.....	37
Commands to be used.....	37
Hints.....	37
4.7 Activity 7 – Five Squares revisited.....	38
4.8 Activity 8 - Squashed Five Squares.....	43
Hints.....	43
5 Give us feedback!.....	44

---

## 1 What is Kojo

Kojo is a fun and friendly graphical environment for playing with and understanding many different things. You can think of Kojo as:

- A Gym - where you can exercise your mind.
- A Studio – where you can create paintings.
- A Lab – where you can experiment with mathematical and scientific ideas.



When you do traditional Art – you paint on a Canvas using Oil or Water colors as a medium. With Kojo, you paint on your Computer Screen using Computer Programming as a medium.

You normally do Maths with Pen and Paper. With Kojo, you use Computer Programs to play and interact with Mathematical ideas.

---

### 1.1 Why work with Kojo?

- To practice and develop your logical thinking skills - by learning and doing computer programming.
- To practice and develop your creative thinking skills - by creating beautiful paintings using geometric shapes.
- To get better at Maths:
  - By developing the kind of thinking skills required by Maths (which are very similar to the thinking skills required by computer programming).
  - By using computer programs to create mathematical objects, thereby understanding them better.
  - By interacting and experimenting with Mathematical concepts inside a Virtual Lab.
- To gain a deeper understanding of computers, and to become proficient at using them.



### Why learn computer programming?

- It's fun!
- To exercise your mind!
- To develop your logical and creative thinking skills.
- To learn to use the computer as a virtual laboratory to play with ideas!
- To hone your reading and writing skills.
- The world around you is driven more and more by computer programs. It's good to be in on the secret. Computer programming has been called the *New Literacy* for the 21<sup>st</sup> century.
- It's a universal human activity, and gives you an opportunity to interact with your peers anywhere around the world.

## 1.2 Painting with Kojo

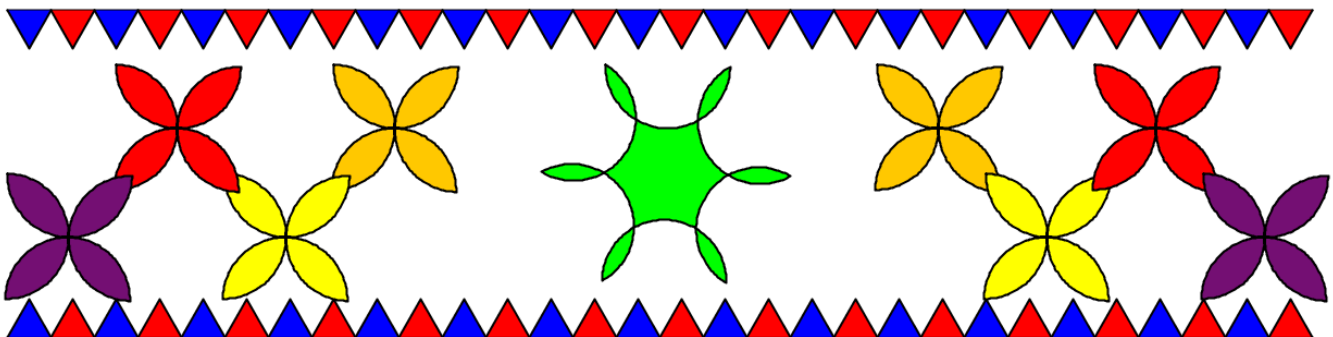
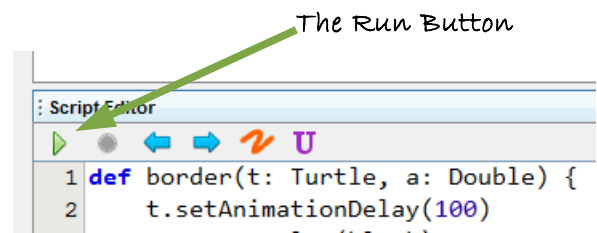
To give you a taste of the kinds of things that you can do with Kojo, let's look at some sample programs that come bundled with Kojo.

### 1.2.1 Rangoli

Click on *Samples* → *Turtle Art* → *Rangoli* within the Menubar at the top of the Kojo Window.

Then click the Run Button (shown in the figure to the right) within the Window called the Script Editor.

You should see turtles marching around, drawing a painting on the screen. After a little bit, you should see the following drawing on your screen:



Pretty, isn't it. This is an example of the kind of colorful paintings that you can make with Kojo.

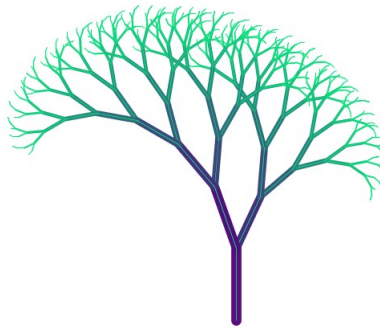
## 1.2.2 Fractal Tree

Did you know that you can use Mathematics to build very real looking pictures of natural objects like clouds, mountains, trees, flowers, etc. These realistic looking Mathematical objects are called Fractals. Here's one that you can look at within Kojo:

Click on *Samples* → *Advanced* → *Tree*

Then click the Run Button.

After a little bit, you should see the following drawing on your screen:



Looks pretty realistic, doesn't it?

### 1.3 Animation with Kojo – Sine of an angle

If you're studying Trigonometry, you will have come across functions of angles (expressed in radians) - like *sine*, *cosine*, and *tan*. Ever wonder what a radian is? And how the sine of an angle is related to its value in radians?

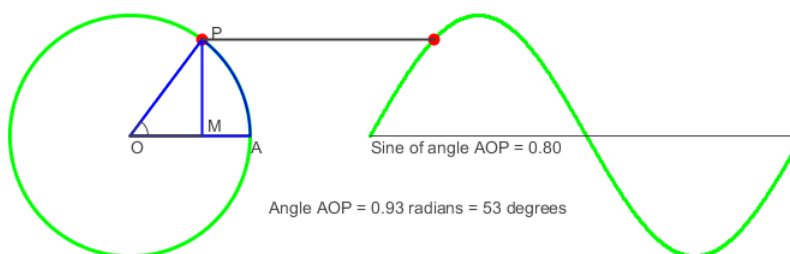
There's a sample program within Kojo that might give you some insights in this area...

Click on *Samples* → *Early Preview* → *Canvas* → *Sine of an angle*

Then click the Run Button.

An animation should show up:

This animation shows how the magnitude of an angle and its sine are defined, and how they relate to each other.  
 The angle of interest is AOP. Its magnitude is defined (in radians) as the ratio of lengths - AP/OP.  
 The sine of this angle is the ratio of lengths - MP/OP.  
 If you consider OP to be equal to one unit in length, the sine of AOP is equal to MP.  
 This magnitude is shown by the red dot on the curve to the right of the circle.



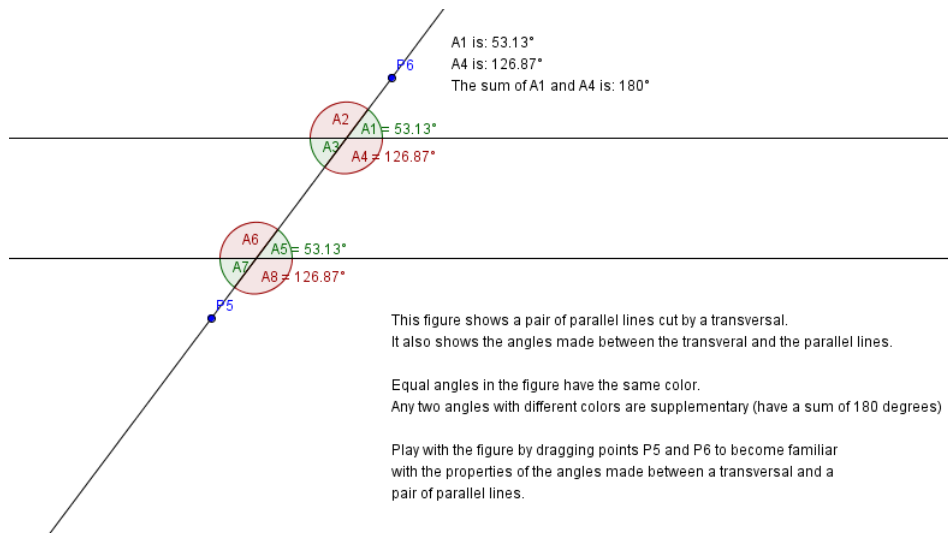
Did that help?

Kojo gives you tools to allow you to create animations for visualizing Mathematical concepts. Future ebooks will guide you on how to do this – to help you gain deeper insights into important mathematical ideas.

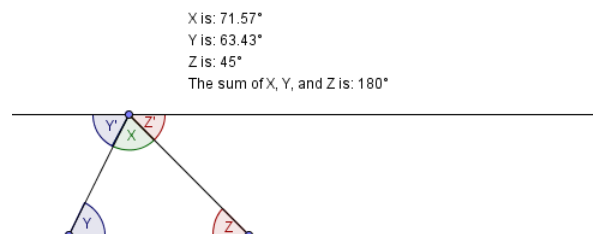
## 1.4 Geometry with Kojo – Parallel Lines, and the angles of a triangle

Ever wonder why the sum of the angles of a triangle is 180 degrees? Run a couple of Kojo sample programs to find out:

Samples → Early Preview → Math World → Parallel/Transversal



Samples → Early Preview → Math World → Angles of a Triangle



This figure shows a triangle inscribed between two parallel lines.

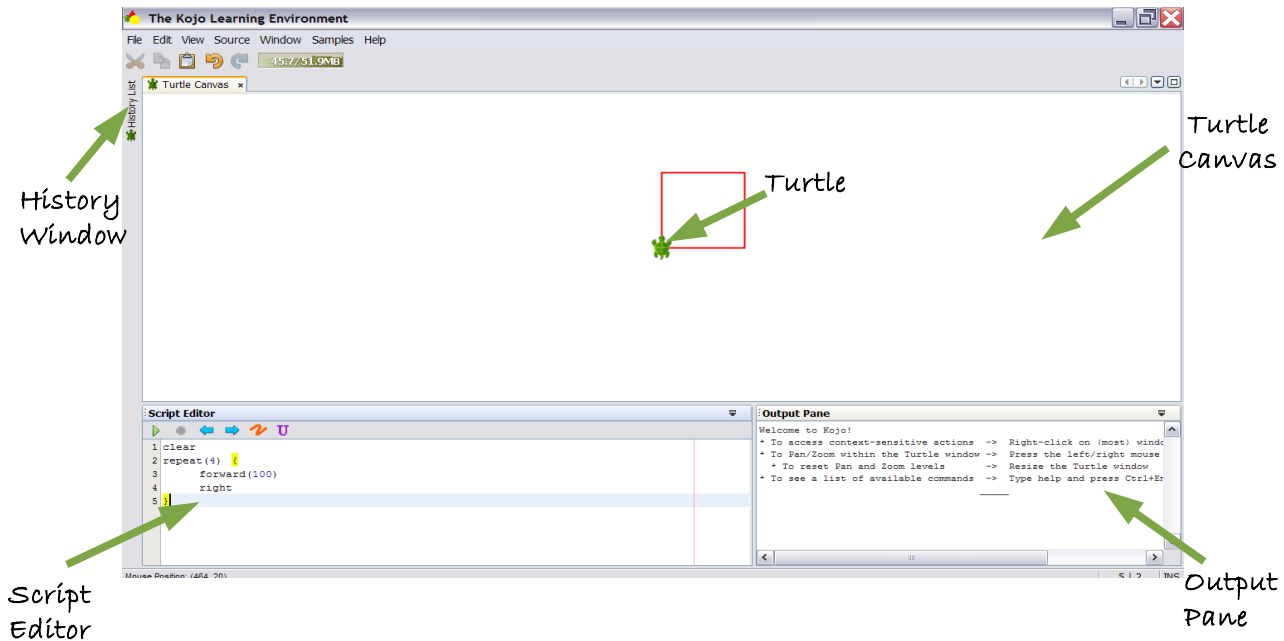
The sum of the angles X, Y', and Z' is 180 degrees (why?)  
Y = Y' (relates to a property of parallel lines cut by a transversal)  
Z = Z' (same as above)

Can you see why the sum of the angles of a triangle is 180 degrees?  
Play with the figure by moving the vertices of the triangle

Hopefully, that gave you some idea of the kinds of things that you can do with Kojo. That's just the tip of the iceberg, though. As you become proficient with Kojo and start working with it, you'll find that the only limit to what you can do with Kojo is your imagination! Seriously!

## 2 The Kojo User Interface

Let's start getting familiar with Kojo by looking at the different elements of the Kojo User Interface...



Within Kojo, you get a pet turtle to do your bidding - to make geometric shapes and paintings. You tell the turtle what to do, and the turtle does it for you. The main features of the Kojo User Interface are:

- **The Turtle** – this is the creature that follows your instructions to draw things on the screen.
- **The Script Editor** – this is where you write your programs (or scripts) that instruct the Turtle to do things on the screen.

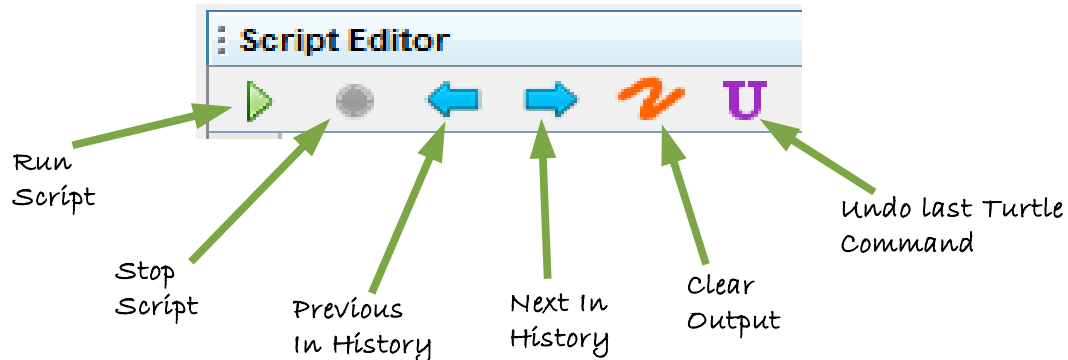


A **program** is a sequence of instructions for the computer. Within Kojo, your programs tell the turtle what to do.

A **script** is a small program.

- **The Turtle Canvas** – this is the area of the screen where your drawing shows up.
- **The Output Window** – this is where Kojo tells you things that you need to know, like the fact that there's a problem with the script that you want the Turtle to follow.
- **The History Window** – stores all the commands and scripts that you run, so that you can easily call them up again.

- **The Script Editor Toolbar** – has buttons that allow you to do core tasks within Kojo:



- The **Run Script** Button – runs the contents of the Script editor.
- The **Stop** Button – allows you to stop a script that is running. It also allows you to stop runaway scripts that are taking too long to finish.
- The **Previous in History** Button – calls up, within the Script Editor, the last command/script that you ran.
- The **Next in History** Button – along with the previous button, allows you to move back and forth within your command/script history.
- The **Clear Output** Button – Clears the output pane, making it easy for you to look at the output of scripts that you run after that.
- The **Undo** Button – allows you to undo the last command that you asked the turtle to carry out.

---

## 2.1 Context Sensitive Actions

Context Sensitive Actions are things that you can do with a particular Window. Right-clicking on a Window shows a Popup with the available actions for that Window.

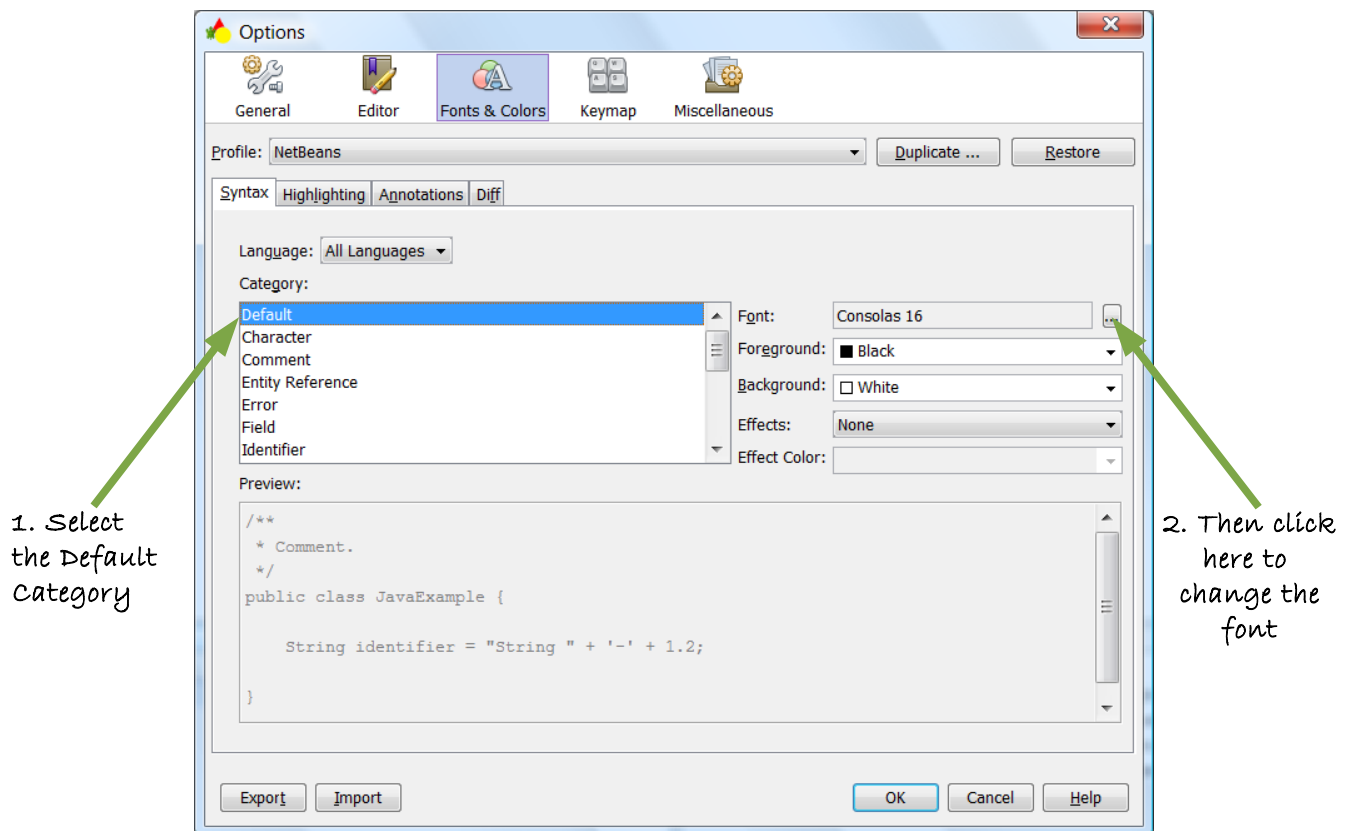
These actions are called Context Sensitive because they depend on the window that you are currently working with.

### Exercises

1. Right click on the Script Editor and make a note of the available Actions. Do the same for the Output Pane. Are the Actions for the two Windows different?

- Use the context actions to change the font (and the font size) for the Script Editor and the Output Pane. Select a Font that you like. Make sure it's nice and big in size.

To change the font for the Script Editor in the exercise above, you will need to click on the *Experimental Features* → *Advanced Options* menu item - to bring up the Dialog Box shown below. You then need to change the font for the Default Category (also shown below).



## 2.2 Panning and Zooming

Within the Turtle Window, you can:

- Pan (move the drawing around) by pressing the left Mouse button and dragging.
- Zoom (make the drawing bigger/smaller) by pressing the right Mouse button and dragging.

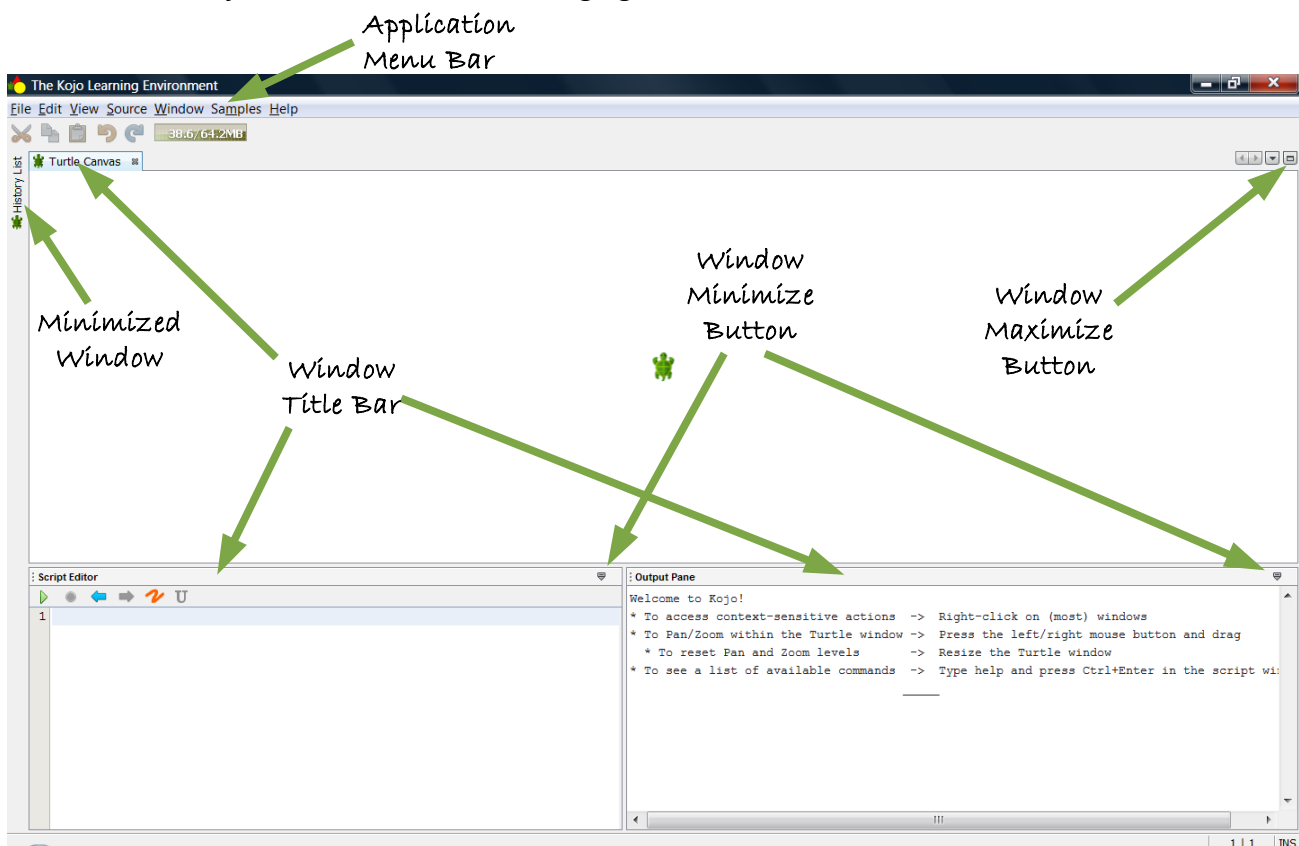
To reset the Pan and Zoom levels you need to re-size the Turtle Canvas (re-sizing a Window means making it bigger or smaller by dragging its border).

### Exercises

1. Pan the Turtle Canvas (by pressing the left Mouse button and dragging) to move the turtle around.
2. Zoom the Canvas (by pressing the right Mouse button and dragging) to make the Turtle bigger or smaller.

## 2.3 Window Management

Kojo has a very flexible Window System. The following figure highlights some of the important elements of the Kojo Users Interface for managing Windows.



The following are some of the things that you can do to control the Windows within the Kojo workspace:

### 2.3.1 Maximizing Windows

When you maximize a window, it grows in size to fill up all the screen space allocated to Kojo. You can maximize a Window by:

- Double clicking on its Title Bar.
- Or clicking on its Maximize Button, if it has one. Once a window is maximized, its Maximize Button changes to a Restore Button.

You can restore a maximized Window to its earlier size by:

- Double clicking on its Title Bar.
- Or clicking on its Restore Button.

#### Exercises

1. Maximize the Turtle Canvas, and then restore it to its original size.
2. Maximize the Script Editor, and then restore it to its original size.

### 2.3.2 Minimizing Windows

When you minimize a window, it shrinks and becomes a label at the side or bottom of the Kojo Workspace. You can minimize a Window by clicking on its minimize button, if it has one.

If you bring the Mouse pointer over the label of a minimized Window, the Window will slide out to show you its contents. If you place the Mouse pointer outside the Window, it will slide back in, and will again become a label at the side or bottom of the Kojo Workspace.

To restore a minimized Window, you need to right-click on its minimized label to bring up a Popup, and then un-check the *Minimize Window* item within the Popup by clicking on it.

#### Exercises

1. Minimize the Script Editor, Slide it in and out from the edge of the Kojo Workspace (by placing your Mouse pointer over the minimized label), and then restore it to its original size.
2. Slide the History Window in and out from the edge of the Kojo Workspace, restore it so that it's visible within the workspace, and then minimize it again.

### 2.3.3 Moving and Docking Windows

If you want to change the arrangement of the different Windows within the Kojo Workspace, you can click on a Window's Title Bar and drag it to the new location that you want to place it in. Before letting go with your Mouse, make sure that you see an orange outline for where Kojo will *dock* your window at the new location. If you don't see an orange outline, the Window will slide back to its original location when you release the Mouse button.

*Note:* If you mess up the Kojo Workspace while playing with the different Windows, you can click on the *Window* → *Reset Windows* menu item to restore the different Windows to their default settings.

## 3 A Guided tour of Kojo

In this section, you will get to work your way through the creation of a simple Geometric painting. Along the way you'll learn about:

- The programming language used within Kojo.
- Making step-by-step progress towards the creation of a program that generates the desired painting.
- Manipulating the text of your programs.
- Recovering from errors.
- Working with the History Window.
- Saving the program in a file after it is done.
- Loading the program from the saved file within a new Kojo Session.

---

### 3.1 The Programming Language

Kojo supports a very powerful language called Scala. Everything that you write within the Script window will be in the Scala language.



More information about Scala is available on the Scala website: <http://www.scala-lang.org>.

You might find it interesting to know that the Kojo Environment itself is written in the Scala programming language.

One of the great things about Scala is that it makes it very simple to do simple things (while still allowing much more complex things with a little more effort).

In this section, you will be looking at a very small and simple subset of Scala that will allow you to explore Kojo. This will include some Turtle commands and a flow control command.

### 3.1.1 Kojo Commands

You use commands within a program to:

- Get the program to take some action (like drawing something on the screen).
- Have some effect that modifies the future behavior of the program (like changing the colors used by the program to draw things on the screen).

Let's look at a few predefined Kojo commands:

***forward(numberOfSteps)*** – moves the turtle forward.

Example - if you run the command:

```
forward(100)
```

The turtle will move forward 100 steps and draw a line along the path that it travels.

***back(numberOfSteps)*** - moves the turtle back.

Example - if you run the command:

```
back(100)
```

The turtle will move back 100 steps and draw a line along the path that it travels.

***right()*** - turns the turtle right through 90 degrees at its current position.




***left()*** - turns the turtle left through 90 degrees at its current position.

***clear()*** - clears the canvas area and gets the turtle back to its original position.

***home()*** - moves the turtle to its original location, and makes it point north.

***setPenThickness(number)*** - specifies the thickness (as a number) of the pen that the turtle draws with.

The default thickness of the pen is 2. A bigger number means a thicker pen, as shown below:

	<i>SetPenThickness(1)</i>
	<i>SetPenThickness(2)</i> - <b>Default</b>
	<i>SetPenThickness(4)</i>

***setPenColor(color)*** - specifies the color of the pen that the turtle draws with.

Example: `setPenColor(blue)` will make the line drawn after the command blue in color.

### 3.1.2 Flow control with repeat

Used individually, the commands above allow you to instruct the Turtle to do a particular thing as described. You can also put many commands together in a sequence to instruct the turtle to do a series of things. For example, the following script gets the turtle to make a particular geometric shape (can you figure out what it is?):

```
forward(100)
right()
forward(100)
right()
forward(100)
right()
forward(100)
right()
```

How about if you want to do a certain number of things, and then go back and repeat them. For example, you can see that in the previous example, the forward and right commands are used over and over again. Surely there's a better way of doing this.

There is! The repeat command allows you to instruct the turtle to repeat a sequence of other commands for a specified number of times. Using repeat, the previous script becomes:

```
repeat (4) {
  forward(100)
  right()
}
```

Isn't it much clearer now what kind of shape the script creates?

How does Kojo know how many commands to repeat? You tell it by putting the commands within a *block* – which is enclosed within an opening and a closing brace. Here's another example of a block with multiple commands inside it:

```
{
  forward(100)
  left(45)
  back(100)
  right(45)
}
```

With a basic knowledge of the commands described in this section, you are now in a position to start exploring how to create a simple drawing with Kojo.

---

---

## 3.2 Taking Kojo for a spin

Let's get our hands dirty and try out a few things!

### 3.2.1 Making a simple Shape

Let's start with something that you have seen before:

```
repeat (4) {  
  forward(100)  
  right()  
}
```

Type this into the Script Editor. But wait, you don't have to type in all of the above. Kojo is there to help you with a feature called code-completion.



Code is the name given to the instructions in your program, because they represent a special *code* that the computer understands.

There are two different types of code:

- The readable text of your program is called *source code*. It is also sometimes just called *the source* or *the code*
- The actual binary instructions that are understood by the computer are called *machine code*.

Special programs called compilers or interpreters convert from source code to machine code. Kojo has an interpreter inside it that converts the *source code* that you write into *machine code* before running it.



Code completion (also known as IntelliSense) is a feature that enables Kojo to automatically complete a word (of source code) that you are typing based on the current context and the letters that you have already typed. Code completion will save you a lot of typing grunt-work as you write programs within Kojo.

Code completion will also provide you opportunities to explore new commands within Kojo.

Let's get Kojo to do some of your typing work for you. Punch in the word *rep* and then press Control+Space (Control and Space keys together) to activate code completion.

Kojo will infer that you are trying to write a repeat statement, and will input the following into the Script Editor:

```
repeat (n) {  
  
}
```

Kojo does not know how many times you want to repeat something, so it just puts an *n* within the parenthesis (where *n* stands for a number that you need to provide). Very conveniently, the *n* is highlighted, and the cursor is positioned right over it, so that you can just type in the repeat count that you want. If you type in 4, *n* will be replaced by 4.

Now, just hit *Enter*. Kojo will automatically position your cursor at the right location on the next line where you can start entering the first command within your repeat block.

Next, type in *forw*. The script editor should now contain:

```
repeat (4) {  
    forw  
}
```

Hit Control+Space to have Kojo fill in the forward command for you.

Proceed in this way till you have the script for making a square typed in. Now click on the *Run Script* button. The turtle will start moving and make a nice square for you.

Congratulations! You just wrote your first Kojo program.

### 3.2.2 Accessing History

The History Feature gives you a convenient way to access the commands/scripts that have recently been run by you (Kojo keeps track of approximately the last 1000 commands/scripts run by you).

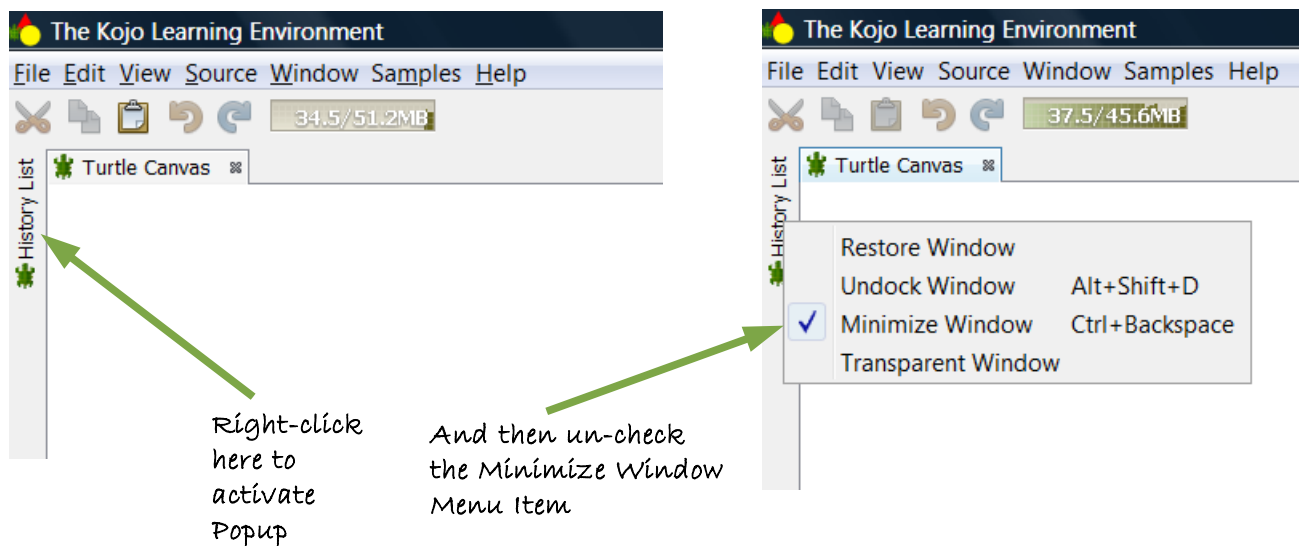
You can access and switch between recently run commands in three ways:

- Using the History Window
- Using the History buttons in the Script Editor toolbar

- Using the Keyboard:
  - Control+Up-Arrow loads the previous command/script in the script editor
  - Control+Down-Arrow loads the next command/script in the script editor

Let's see how you can use the History Window to access history.

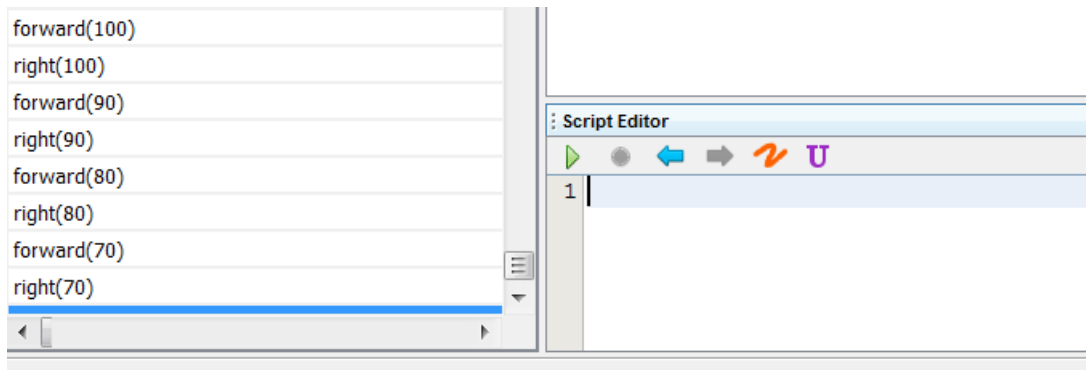
First, Right-click on the **History List** Label to activate a Popup, and then un-check the *Minimize Window* Item within the Popup - to show the History window (which is minimized by default):



Then, run the following commands (to create some items in your history):

- forward(100)
- right(100)
- forward(90)
- right(90)
- forward(80)
- right(80)
- forward(70)
- right(70)

The commands that you just ran should be visible in the history window, as shown below:



Click on the right(90) command; notice that it appears in the Script Editor.

Click on the forward(70) command. See it show up in the Script Editor.

Try this with a few other commands in your history. Notice that you can run the commands that show up in the Script Editor after clicking the history window - just like you would run something that you typed in yourself.

Now, try accessing the commands in your history using the history buttons in the Script Editor toolbar.

Next, try accessing the commands in your history using the Control+Up-Arrow and Control+Down-Arrow keys.

### 3.2.3 Playing with the Script Text

As you work with Kojo, you will be spending a lot of time within the Script Editor. It is useful to know about some of the powerful capabilities of this window...

#### *Format*

When you write a computer program, you have two different audiences for the program:

- The computer, which will run your program
- Other programmers, who will read your program, understand it, learn from it, and possibly extend it.

When you format your code, you make it easier for other programmers to understand the structure of your code. Take a look at the following two snippets of code, one badly formatted, and the other nicely formatted:

```
repeat(4){ forward(100)
right()}
```

```
repeat (4) {
  forward(100)
  right()
}
```

Which one do you think is easier to understand?

Kojo makes it very easy to format the code that you have written:

- Press Ctrl+Shift+F
- **Or** Use the Source → Format menu item

The one thing that you need to do to get Kojo to help you with code formatting - is to get your new-lines right. For example, let's take the badly formatted code that we looked at earlier:

```
repeat(4){ forward(100)
right()}
```

To make code nicely formatable, you should follow the rule that an opening brace should never have any text after it on the same line, and a closing brace should never have any text before or after it on the same line. Applying this rule to the code above, we get:

```
repeat(4) {
forward(100)
right()
}
```

Now, if we press Ctrl+Shift+F, Kojo will format the above code for us and make it look like this:

```
repeat(4) {
  forward(100)
  right()
}
```

That is exactly what we wanted!

## Selecting Text

You can select a fragment of text by using either the Mouse or the Keyboard:

- Mouse - left-click and then drag your Mouse pointer over the text fragment that you want to select.
- Keyboard - go to the beginning of the text fragment that you want to select, press the Shift key, and then move the cursor by using the right-arrow key till you get to the end of the text fragment.

### Exercises

1. Using the History Window, pull up the script for making a square. Try to select the text within the repeat block by using first the Mouse, and then the Keyboard.

Why would you want to know how to select text? Because once you have a selected fragment of text, you can do things with it:

## Copy & Paste

Let's say you have some code to move the turtle:

```
forward(100)
right()
```

You now want to add some additional commands, say `forward(100)` and `right(45)`, below the existing commands. This is very easy to do with Copy & Paste:

- Select the two lines that contain the existing commands.
- Copy them into the clipboard by pressing Ctrl+C or using the copy command from the Edit menu.
- Move the cursor to the line where you want to add the two new commands.
- Paste the lines from the clipboard into the location specified by the cursor by pressing Ctrl+V or using the paste command from the Edit menu.

Your code should now look like this:

```
forward(100)
right()
forward(100)
right()
```

You can now go into the fourth line and add `45` within the `right()` command to get the code that you want.

Do you agree that using Copy & Paste is a powerful way to add new code to your program? Or do you think it's easier to type in all the things that you want in your program with the help of code-completion? In either case, it's useful to know both techniques.

### *Cut & Paste*

Let's now say that you have two lines of code within your program:

```
forward(100)
right()
```

You now want to reverse the order of these two commands. A very easy way to do this is to:

- Select the first line.
- Cut it (by pressing `Ctrl+X` or using the Cut command from the Edit Menu) to delete it from the program and put it into the Clipboard.
- Move the cursor to the line below the `right()` command.
- Paste the line from the clipboard into the location specified by the cursor.

Voila! You've done what you wanted without having to type everything in.

That was easy, wasn't it?

### *Run*

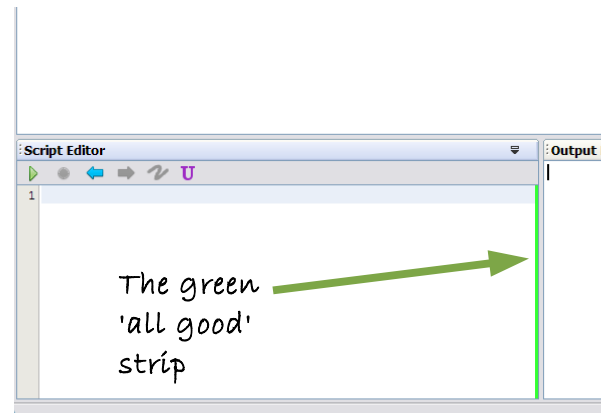
Once you select one or more lines of text, clicking on the Run Button will run just those lines of code, as opposed to running all the code within the Script Editor. This helps tremendously with incremental refinement of your program, as described in a future section.

### 3.2.4 Recovering from Errors

Once again, run the script to make a square:

```
repeat (4) {
  forward(100)
  right()
}
```

Notice that the Script Editor window displays a green strip on its right margin. This is meant to indicate that the script has no errors.



Some of the things in your script that will be considered errors are:

- References to non-existent commands .
- Misspelled commands.
- Misplaced or Mismatched braces - {}
- Misplaced or Mismatched parentheses - ()

Let's see what Kojo does if your script does have an error. Pull up the script that you just ran by clicking on the **Previous in History** Button. Now, miss-spell the forward command by adding an x at the end. The script editor should now contain:

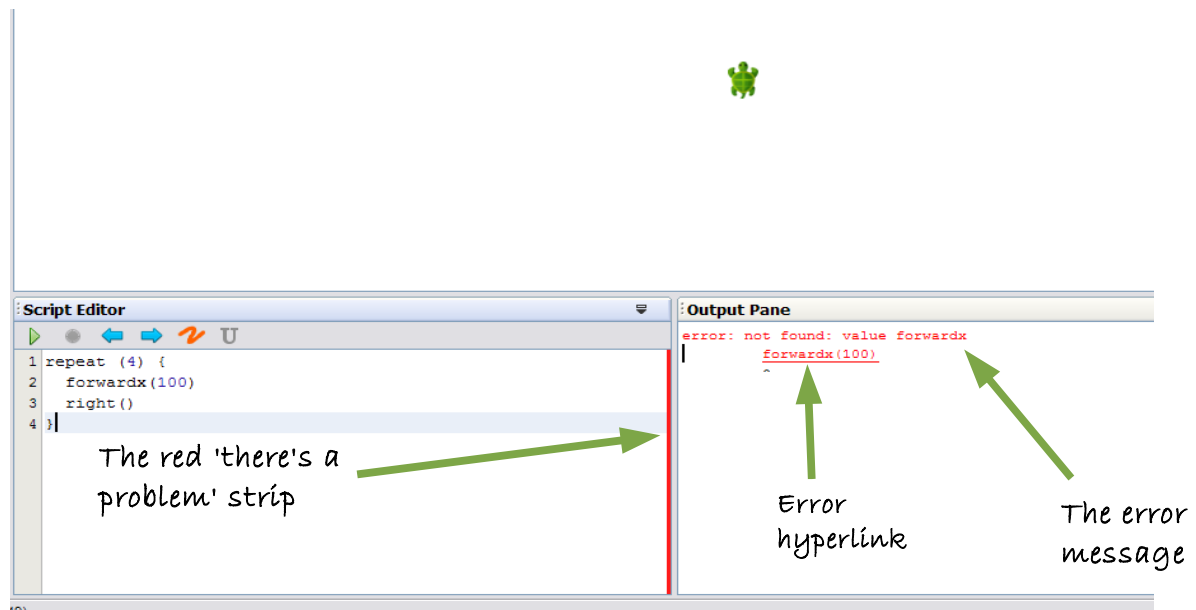
```
repeat (4) {
  forwardx(100)
  right()
}
```

Go ahead and run this script. Notice the following:

- The right margin of the script editor is red. This indicates that there is an error in the script.
- The output window has an error message describing exactly what the error is.
- A portion of the above message is underlined. If you move your Mouse over this, the Mouse cursor changes to a hand - indicating a hyper-link (a hyper-link is something that allows you to

jump from one location to another).

- The line below the hyper-link has a hat character (^) pointing at the precise location of the error.



Click on the hyper-link. The offending portion of text (which has caused the error) within the script editor will be highlighted, and your cursor will be positioned right at that location. You can now fix the error.

### 3.2.5 Refining your programs

All complex computer programs are built out of simpler programs. The process of building a complex program involves the following steps:

1. Build a simple program.
2. Make sure it works.
3. Add a layer of additional functionality.
4. Make sure it works.
5. Are you done? If so, great. If not, go back to step 3.

Kojo gives you a couple of features to help with this - **Incremental Running** and **Undo**

Let's say that you have a program that does what you want. You run it one more time to make sure that it does what you want. This gives you a nice painting within the Turtle Canvas.

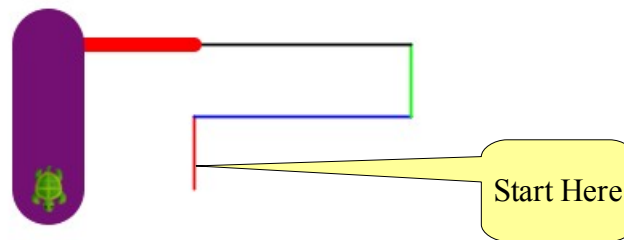
You like what you see, and you now want to build upon it to refine your painting. You add some more code to your script. At this point, you can press the Run Button to run your script and (re)make the entire painting, along with the little bit of new stuff that you added. Or, you can select just the one or

two new lines that you added, and then press the Run Button to run just those lines - to add a little bit of new stuff to your painting.

The second option sounds better, doesn't it. This is incremental development of your program.

Even better, if you don't like what the one or two new lines in your program do, you can hit the Undo button, and Kojo will take away the changes that you just made to your painting. You can now modify these lines, select them, and run them again to see if you like this new version better.

Let us use this technique to make the following figure:



A good place to start to draw this is the the vertical red line:

```
clear()
forward(50)
```



If you run this code, you will see the figure on the right.

```
Script Editor
1 clear
2 forward(50)
3 right
4 setPenColor(blue)
5 forward(100)
6
```

What do you think you need to do next? Look at the target figure above to determine that.

It's pretty clear that the turtle needs to turn right, set its pen color to blue, and move forward. Add the appropriate commands to your script, but don't run the whole thing all over again. Just select the new lines (as shown on the left).

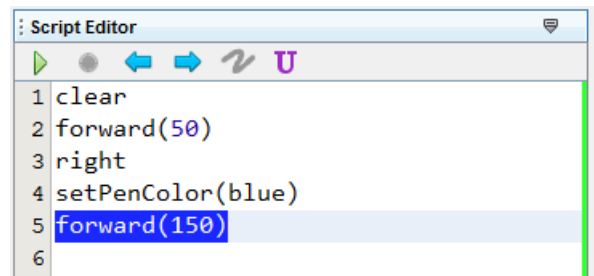
... and then click on the Run Button. Your drawing should now look like the figure on the right.



The blue line does not seem to be long enough. Where do you think the problem is?

It's the last `forward(100)` command, right. If only you could undo this and try something bigger than 100.

Well, you can. Just hit the Undo Button. This will undo the last command that the Turtle ran, which is `forward(100)` in this case. Now change the last line so that it says `forward(150)`. Select this line (as shown on the right)



```

Script Editor
1 clear
2 forward(50)
3 right
4 setPenColor(blue)
5 forward(150)
6

```

... and then click the Run Button to get something like the figure on the right. This looks much better!



### Exercises

Continue in this manner to make the complete figure, using the following guidelines:

- Keep adding new lines, selecting them, and then running them. As explained in the previous section, this is called Incremental Running.
- If you don't like something that you just did, use the Undo Button to undo your actions.
- Once in a while, run the whole script (by not selecting anything before pressing the Run Button), to make sure that your entire program does what you want it to do.
- You need to use the following commands to complete the figure:
  - `clear()`, `forward()`, `right()`, and `left()`.
  - `setPenColor(color)` and `setPenThickness(n)`.
  - `setAnimationDelay(100)` - to speed things up during incremental development.

### 3.2.6 Saving and Loading your work

You are now done working on your masterpiece for the day. You are confident that you will not lose any work that you have done - because Kojo saves every commands/scripts that is run, and makes it available within the History Window. But you might still want the facility of saving a particular script that you wrote into a file (to save you the trouble of hunting for it within the History window).

You can do this by right-clicking on the Script Editor, and then clicking on the *Save To* menu item in the Popup menu that comes up.

In a future session of Kojo, if you want to load this script into the Script Editor, you need to click on the *Load From* menu item within the Script Editor Popup menu.

### Exercises

- Save the script for the drawing that you just made in a file called hanging-turtle.kojo.
- Shut down and then restart Kojo.
- Load the hanging-turtle.kojo script into the Script Editor.
- Run it to make sure it still draws the figure from the previous section.

---

### 3.3 Checklist

Make sure that you know how to:

- Access context sensitive actions within the different Windows.
- Run your programs.
- Pan and Zoom the Turtle Canvas.
- Locate errors within your program.
- Locate the program you are looking for within the History List.
- Save your program to a file.
- Load your program from a file
- Format Text within the Script Editor
- Cut, Copy, and Paste Text within the Script Editor
- Run only a few lines from within the Script Editor by first selecting them
- Undo the last turtle command
- Work on your program in an incremental fashion using the previous two features

---

## 4 A Kojo Adventure Trail

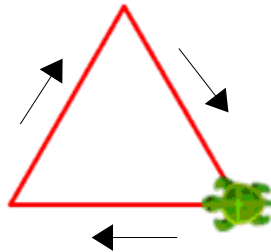
Now that you have some basic familiarity with the Kojo environment, and the kind of workflow that you should use to develop programs using Kojo, it's time to try out a few different activities to give you a flavor of the kinds of things that you can do with Kojo...

---

---

### 4.1 Activity 1 - A Triangle

Do you know what an equilateral triangle is? If not, go read up on equilateral triangles on Wikipedia. Now, try to make a triangle similar to the one shown below:



The maximum size of your program should be 6 lines!

#### Commands to be used

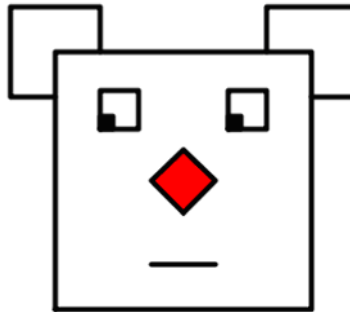
- `clear()` - this should be the first line of your program.
- `left()` - to turn the turtle left through 90 degrees.
- `right(angle)` - to turn the turtle right through a specified angle.
- `forward(n)` - to move the turtle forward.

#### Hints

- Start with your turtle turned to the left.
- Turn your turtle through exactly the same angle each time. The size of this angle should be equal to the size of the exterior angles of an equilateral triangle.
  - The sum of an interior angle and the corresponding exterior angle of a triangle is 180 degrees.
- Look at your code to see if you can use the *repeat* command.

## 4.2 Activity 2 – A Face

Try to make the following figure using the commands that you already know:



### Commands to be used

This figure is made up of squares of different shapes. In this activity, in addition to the regular basic commands (like, *forward()* and *right()/left()*), you will also get a chance to work with some commands that control the pen:

- `setPenColor(color)` ← 'Black' in this case
- `setPenThickness(number)` ← '4' for this figure
- `setFillColor(color)` ← 'red' for the nose, 'black' for the pupil, and 'none' otherwise
- `penUp()`
- `penDown()`

### Hints

- Start with the biggest square.
- Use `penUp()` to move the turtle to the position where you want the next square to start, and `penDown()` to start drawing again.
- Ears are not complete squares.

### 4.3 Activity 3 - Turning Squares

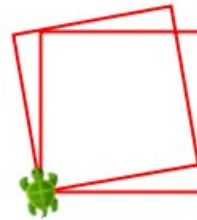
You already know how to make a square:

```
clear()
repeat (4) {
  forward(100)
  right()
}
```



Let's try to make a pattern out of squares:

```
clear()
repeat (4) {
  forward(100)
  right()
}
left(10)
repeat (4) {
  forward(100)
  right()
}
```



Are you starting to see the idea?

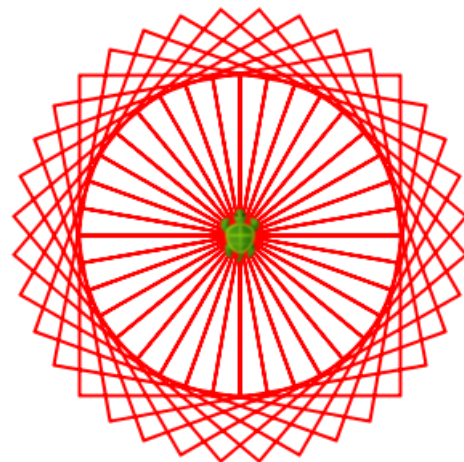
Try to make the pattern on the right.

#### *Commands to be used*

- `clear()`, `forward(n)`, `right()`, `repeat()`
- `setAnimationDelay(10)` - to speed things up

#### *Hints*

- Use a repeat block within a repeat block to make the figure

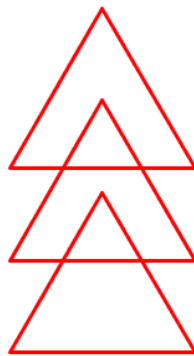


---

---

### 4.4 Activity 4 - Triangles

You made an equilateral triangle in the first activity. Below is a figure made out of multiple triangles. Try to make it.



#### Commands to be used

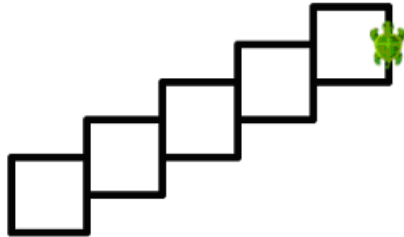
- `clear()`, `forward(n)`, `left()`, `right()`, `right(angle)`, `repeat()`
- `penUp()`, `penDown()`

#### Hints

- Use a repeat block to make a triangle.
- Use *penUp* to move the turtle to the position where you want the next triangle to start, and use *penDown* to start drawing again.
- Try to use repeat within repeat to get the desired figure.

### 4.5 Activity 5 - Staircase

Here's another figure made out of multiple squares. It looks like a staircase seen from the side, doesn't it? Try to make it.



#### Commands to be used

- `setPenColor(color)` ← 'Black' in this case
- `setPenThickness(number)` ← '4' for this figure
- `penUp()`
- `penDown()`
- `clear(), forward(n), right(), repeat()`

#### Hints

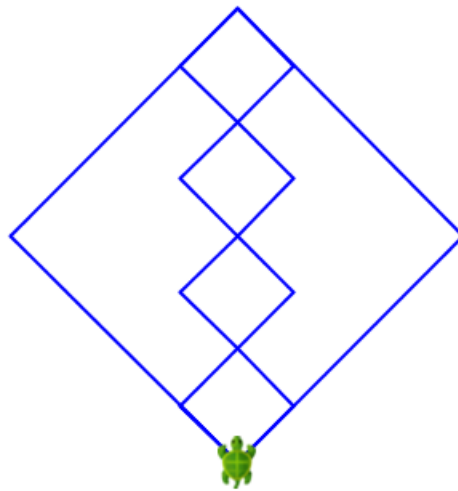
- Use a repeat block to make a square.
- Use `penUp` to move the turtle to the position where you want the next square to start.
- Try to use `repeat` within `repeat` to get the desired figure.

---

---

### 4.6 Activity 6 – Five Squares

Getting bored with squares? Here's another one that uses squares! Please bear with us – this one is going to lead you a little deeper into things. Try to make this:



#### Commands to be used

- `clear()`, `forward(n)`, `right()`, `repeat()`

#### Hints

- Focus on the idea of creating five squares.
- Use a repeat block to create a square.
- Put a repeat block within another repeat block to create the four squares that are inscribed within the larger square.

### 4.7 Activity 7 – Five Squares revisited

Here's a quick quiz question for you – if you see your program doing the same thing over and over again, what can you do to make your program better?

If you've been paying attention, you know that you can use the repeat command to tackle such a situation.

For example, the following code:

```
forward(100)
right
forward(100)
right
```

can be rewritten as:

```
repeat (2) {
  forward(100)
  right
}
```



Why is the version of the code that used the repeat command better? Because it avoids code duplication. Any time you see the same code in different locations within your program, you should start thinking about how you can remove the duplication.

Duplication is bad mainly because:

- If there is something wrong with the duplicated code, it has to be fixed in all the different locations where the code exists. This is error prone and is needless extra work.
- If the duplicated code needs to be enhanced, it has to be enhanced in all the different locations where the code exists. This is also error prone and is needless extra work.

In the context of the Five-squares problem (the previous activity), you probably came up with a solution like the following:

```
clear()
setAnimationDelay(100)
setPenColor(blue)
left(45)
```

```
repeat (4) {
  forward(200)
  right()
}

repeat (4) {
  repeat (4) {
    forward(50)
    right()
  }

  penUp()
  forward(50)
  right()
  forward(50)
  left()
  penDown
}

penUp()
home()
```

This looks pretty good. But if you look closely, you will find that the code to make squares is duplicated:

```
clear()
setAnimationDelay(100)
setPenColor(blue)
left(45)
repeat (4) {
  forward(200)
  right()
}

repeat (4) {
  repeat (4) {
    forward(50)
    right()
  }

  penUp()
  forward(50)
  right()
  forward(50)
  left()
  penDown
}

penUp()
home()
```

The duplicated code is almost the same, but not quite the same. How can we capture the things that

are common across the two code fragments, and make our program better. In this case, we can't use another repeat command to avoid duplication because the duplicated code is not exactly the same (it's almost the same, but not exactly). In general, the repeat command cannot be used to avoid duplication if:

- The duplicated code is not exactly the same.
- Or - the different fragments of duplicated code are not adjacent to each other, like in the following code:

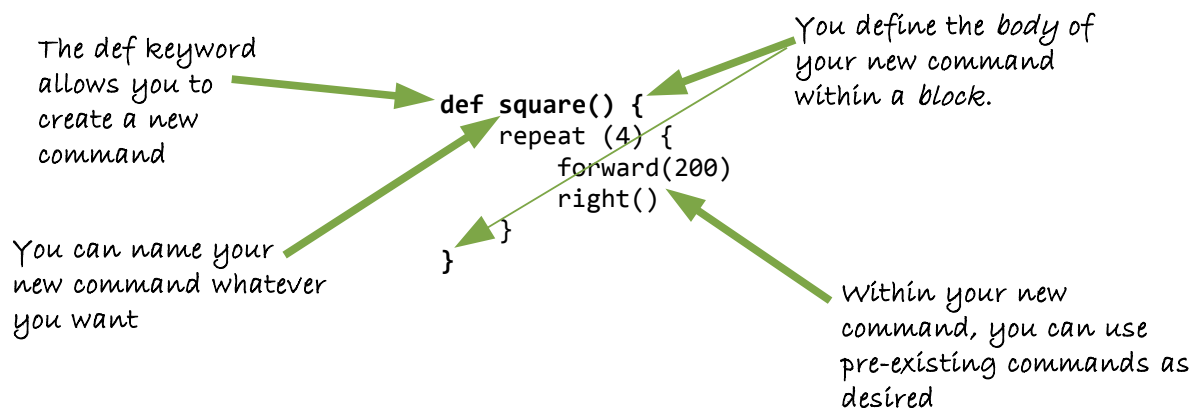
```
forward(100)
right()
back(50)
left()
forward(100)
right()
```

In situations where the repeat command cannot be used, Kojo has another powerful feature that can come to our aid – user defined commands.

Let's see how we can create a user defined command to make a square. We know that the following code makes a square:

```
repeat (4) {
  forward(200)
  right()
}
```

Using the code above, it is easy to create a user defined command to make a square:





A **keyword** is a special word in your program that is understood by Kojo. The **def** keyword allows you to create user defined commands within Kojo.

Keywords are different from commands. **Commands** let your programs do things. **Keywords** allow you to structure and organize the commands in your programs in better ways.

With the square command in place, you can just say `square` wherever you want to draw a square within your program.

Does this solve the problem that we're trying to solve? Let's try it out:

```
def square() {
  repeat (4) {
    forward(200)
    right()
  }
}

clear()
setAnimationDelay(100)
setPenColor(blue)
left(45)
square()

repeat (4) {
  square()
  penUp()
  forward(50)
  right()
  forward(50)
  left()
  penDown
}

penUp()
home()
```

Try running the code above (just copy it from above, paste it into the Script editor within Kojo, and then run it).

Does it do the right thing?

It doesn't. Can you see why?

The problem is that our user defined square command always makes squares with sides that are 200 steps long. In the five-squares problem, we want to create one square with sides that are 200 steps

long, but four squares with sides that are 50 steps long. That's why the above solution does not work. Kojo gives us another feature to tackle this problem – inputs to commands.

When you define a command, you can provide it an input:

```
def square(n: Int) {  
  repeat (4) {  
    forward(n)  
    right()  
  }  
}
```

In the bold portion of the code above, you are telling Kojo that the input to the square command is called *n*, and that its type is *Int* (Int stands for integer). Now, instead of always drawing squares with sides that are 200 steps long, the square command can draw squares of different sizes based on the input that is provided to it.



Inputs to commands are *named values* that can be used within the body of a command. Inputs have *types* associated with them. The type of an input tells Kojo:

- the permissible values of the input.
- the operations that are allowed with the input.

Telling Kojo the type of the input to your user defined command has a couple of advantages:

- It makes it easy for Kojo to identify problems with your usage of the input value, and to tell you if you make a mistake.
- It makes it easier for you (and your friends) to understand what the command does when you (or they) look at it later.

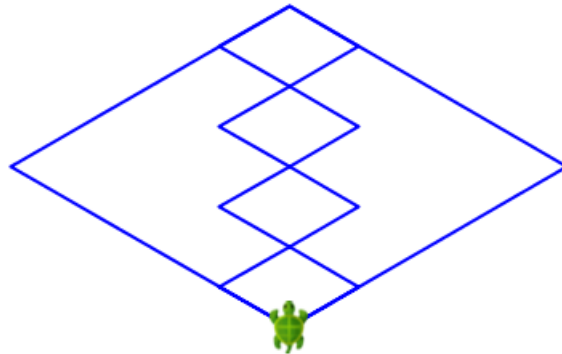
Use the information in the previous section to draw the five-squares figure using the user defined square command.

---

---

### 4.8 Activity 8 - Squashed Five Squares

Using the information and ideas provided in the previous activity, make the following figure:



#### Hints

- The basic building block of the above figure is a Rhombus, as opposed to a square.
- Do a Google search to determine the properties of a Rhombus. Use this information to come up with a command to draw a Rhombus. This command should take two inputs:

```
def rhombus(side: Int, angle: Int)
```

where *side* is the length of the sides of the rhombus, and *angle* is one of its internal angles.
- Combine the rhombus command that you have defined - with the repeat command - to create the above figure.

## 5 Give us feedback!

If you have liked what you have seen with Kojo so far, we want to hear from you. Your feedback will help us decide what to do next.

Some of the things that we want to know from you:

What topics are you interested in reading about (and playing with), using Kojo?

- Computer Programming in general (to practice logical thinking).
- Making great looking computer paintings (to unleash your creativity).
- Creating Animations (to unleash your creativity and visualize scientific concepts).
- Experimenting with Maths (in a virtual laboratory).
- Experimenting with Physics (in a virtual laboratory).

Would you be willing to pay (around Rs. 200 / \$5) for ebooks on the above subjects?

If you represent a school, would you be interested in starting (Co-Scholastic) activities based on Kojo in your school?

Let us know. Your feedback will encourage and guide us!

Write to us at [pant.lalit@gmail.com](mailto:pant.lalit@gmail.com) or [pant.vibha@gmail.com](mailto:pant.vibha@gmail.com)

Hoping to hear from you...