Adventures with Kojo

Exploring Math, Art, and Programming

LEVEL I

By Lalit and Vibha Pant

Play . Learn

www.kogics.net

Front Matter

This is the second in a series of books about using Kojo to play with Computer Programming, Art, and Math. The focus of this book is:

- To make you familiar with the basic ideas of Computer Programming (and to relate these ideas to Algebra and Arithmetic, where appropriate).
- To get you to experiment with basic Geometry.
- To enable you to exercise your creative and logical thinking skills by making nice and colorful computer sketches - using just a few building blocks, and your knowledge of Programming and Geometry.

The book contains:

- Pre-written programs for you to type in, run, and study.
- Some of the theory behind these programs. You might find that you are not too interested in the theory to begin with. If that's the case, feel free to just lightly browse through it in your first reading of the book – to get a general idea of things. (But be aware that having a good understanding of the theory will make you a more effective programmer).
- Exercises for you to do - which test your understanding of the pre-written programs. Solutions to most exercises are provided at the end of the book; but you should look at them only after you have tried doing the exercise yourself.
- Mini-projects for you to try out.

As you go through the book, if you run into anything that you do not understand, or if you have any kind of feedback about anything that you see in the book, you are welcome to go over to the Forum for the book (at *http://www.kogics.net/forum*) to start a discussion. We'll be there to interact with you.

Book version: Feb 28, 2013

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

Table of Contents

Prerequisites

Before you start reading this book, please make sure that you have read (or at least browsed through) the previous book in this series: *Kojo, An Introduction* – which can be freely downloaded from:

http://www.kogics.net/kojo-ebooks

After going though the *Introduction* book, you should be at least somewhat familiar with the items specified in the Checklist at the end of that book.

Also, make sure that you have the latest version of Kojo. This can be obtained from the Kojo download page at:

http://www.kogics.net/kojo-download

Introduction

In this book, you are going to start learning how to write computer programs.

So what are (computer) programs?

The computer is a powerful device, waiting to do your bidding. You tell it what to do by writing programs, and then running them. A *program* is a series of instructions for the computer. When you ask a computer to run a program, the computer creates something called a *process* to run it. This *process* then runs and evolves according to the instructions specified in the *program*. To take an analogy - a *program* is similar to a cooking recipe, while a *process* is similar to the act of cooking based on the recipe. The computer (and more precisely, the CPU inside it) is the cook.

You write programs in a very precise language that the computer understands. Make one little mistake, and the computer will not do what you ask it to do (it will instead tell you that there's a problem). In other words, what you write in your programs is a *code* that the computer understands. Hence, the text of your programs is also called code (or source code).

A code is a system of letters and words with a well defined meaning. Some examples of code are:

Legal code – the law of a land.

Secret code – used by spies to exchange messages in secret; also used by us to shop online (to ensure security and privacy).

Source code – the text of computer programs.

This book consists of various activities that will guide you on your journey of learning how to write computer programs. Along the way, you'll practice and become proficient at:

- Systematic thinking and problem solving.
- Creative thinking.
- Basics of Geometry.

Ready? Let's go, then...

Exercises

1. What is a computer program?
2. What is a process? How is it different from a program?
3. What does it mean if we ask you to “write some code” to solve a problem?

Activity 1 – A sequence of commands

Type in the following code and run it:

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

You should see the figure above within the Turtle Canvas.

Let's try to understand what this program does...

As you know, a program contains a sequence of instructions. This program makes use of a special kind of an instruction called a *command*. You use or *call* 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).

In this program, two commands – `forward` and `right` – are called repeatedly to accomplish the desired goal. The `forward` command is also given an *input* (the number 100).

Here's a description of all the commands used in the program:

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

`forward(numberOfSteps)` – moves the turtle forward by the given number of steps (which is the input to the command).

Example - if you run the command:

```
forward(100)
```

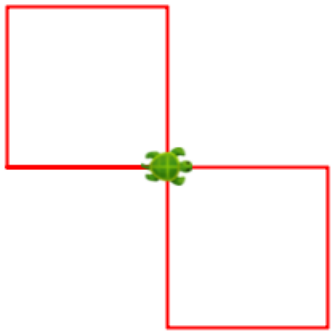
The turtle will move forward 100 steps and draw a line along the path that it travels.

`right()` - turns the turtle right through an angle of 90 degrees at its current position.

Try to understand how this program works before moving on.

Exercise

Write a program to make the following figure:



Note - within an exercise for an activity, you are expected to use only the commands that are present in the program right at the beginning of the activity.

Try to use the following features of Kojo (which are described in the *Kojo Introduction* book) while you write the program:

- Code Formatting (described on Pg 22 of that book).
- Copy and Paste (described on Pg 24).
- Code Completion (described on Pg 19).
- Incremental Running (and Undo, if required) (described on Pg 27).
- Error recovery (as required) (described on Pg 26).

Activity Checklist

It's good if you know the answers to the following questions before moving on:

1. What is a command?
2. In a program, what do commands do?
3. What do these commands do:
   a) `clear()`
   b) `forward(n)`
   c) `right()`

Activity 2 – Repeat after me

Type in the following code and run it:

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

This program creates the same output as the program in the previous activity. Is this program better than the previous one? Or was the earlier program better? Think about that before reading on...

This program is better because:

- It does the same thing as the earlier program in much fewer lines of code – by doing away with repetition.
- It is much clearer to read.

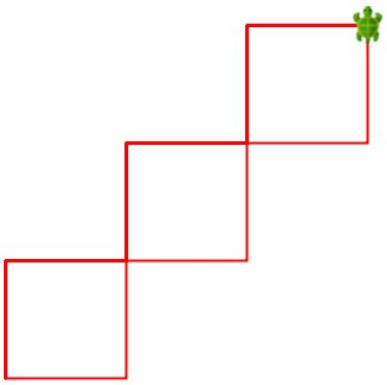
This program makes use of a new command - `repeat`. The `repeat` command allows you to run a sequence of other commands for a specified number of times. In the above program, the `forward` and `right` commands are repeated four times.

The flow of the process generated by this program is very different from the flow of the process generated by the program in the previous activity (which also makes a square). This process loops (goes back to an earlier point in the program and repeats certain instructions) as it runs the program, while the earlier process goes straight through, sequentially running all the instructions in the program.

Try to understand how this program works before moving on.

Exercise

Write a program to make the following figure:



Remember, you are allowed to use only the `forward` , `right` and `repeat` commands to make this figure.

Now, simplify your program by using the `left` command, which does the following:

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

Again, try to use the following features of Kojo while you write the program:

- Code Formatting.
- Copy and Paste.
- Code Completion.
- Incremental Running (and Undo, if required).
- Error recovery (as required).

We will not mention this again, but you are expected to keep using the above features of Kojo as you work with the Activities in the rest of this book. Using these features will help you in using Kojo more productively, and will give you useful practice in using the computer with proficiency.

Activity Checklist

It's good if you know the answers to the following questions before moving on:

1. What does the `repeat(n)` command do, and why do we use it?
2. What does the `left()` command do?

Activity 3 – Number Crunching

Type in the following and run it:

```
4 + 2
```

You will see something like the following show up in the Output Window:

```
res0: Int = 6
```

You asked Kojo to evaluate an Arithmetic *expression*, and Kojo responded with a result *value*. Kojo also gave this value a name (res0), in case you want to refer to it again (we'll do so shortly). It also told you the *type* of the result: **Int** – standing for integer.

There are three new words here that you're encountering for the first time – *expression, value,* and *type*.

What exactly is an *expression*? We'll, it's just another kind of an instruction in your program.

So now you know about two kinds of instructions in programs – commands and expressions.

What's the difference between them?

You call commands to have some effect. You evaluate expressions to compute results. These results are called *values*. For example:

- `forward(100)`, a command, moves the turtle forward by a hundred steps.
- `3 + 4`, an expression, computes a result value (seven).

The *type* of a value tells you what operators and commands it can work with. More on this later...

Now, try:

```
res0 * 3
```

Kojo will respond with:

```
res1: Int = 18
```

Can you guess what the * operator does when you call it?

Also, notice that Kojo remembered the named value (res0) from the previous step (you were able to use it), and it gave the result a new name (res1).

Now try:

```
res0 * 3.1
```

Kojo will respond with:

```
res2: Double = 18.6
```

Just like Int stands for integer, **Double** stands for *real*. A Double value represents a real number with a certain (double) degree of precision.

Do you remember your numbers? Here's a quick recap:

Natural numbers – 0, 1, 2, etc. Useful for counting

Integers – include the negative numbers.

Rational Numbers – etc. Useful for measuring.

Real Numbers – Pi, etc. Represent quantities that can not be measured exactly.

Now try the following (one line at a time), and note Kojo's response for each line:

```
10 - 5
4.0 / 3
(2.7 + 1.3) / 3
4 / 3
```

Does everything make sense?

Probably the last one came as a surprise. When both the numerator and denominator for a division operation are integers, Kojo does ***integer division***, and discards the portion of the answer after the decimal point.

Note – Kojo will print out answers for numerical calculations in the Output Window - only if the calculations are specified in a single line of code. If you run multiple lines of code, nothing will be shown in the Output Window. If you want to see a result in the Output Window (when running multiple lines of code), you will need to use the `print` command. That will be the subject of a later activity.

Exercise

Do the following calculations:

- 21233 plus 1104
- 981 minus 1052
- 214 times 389
- 10 divided by 3

Activity Checklist

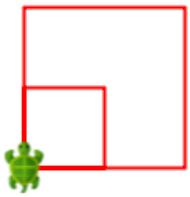
It's good if you know the answers to the following questions before moving on:

1. What are **expressions**?
2. What's a value?
3. What's the type of a value? Name two types?
4. What is **Integer Division?**

Activity 4 – What's in a name

Type in the following code and run it:

```
val size = 100
clear()
repeat (4) {
    forward(size)
    right()
}
repeat (4) {
    forward(size / 2)
    right()
}
```



You'll see a two square pattern show up within the Turtle Canvas.

What do you think you need to do to make a smaller pattern?

Just change the first line of the program to read: `val size = 50`, and then run it again.

How about making a larger pattern?

Change the first line of the program to read: `val size = 200`, and then run it again.

What you see here is the ability to refer to numbers by name, as opposed to their values.

Why would you want to do that?

For at least two reasons:

- **To make it easier to make changes in your program.** You saw this when you needed to make a smaller version of the pattern above. You just had to change one line at the beginning of your program to make the smaller pattern, instead of having to scan through your whole program and making multiple changes.
- **To make your program more understandable.** Looking at the program above, it becomes clear right away that the size of the square made by the program can be controlled.

You saw named values earlier in the Number Crunching Activity. The results of your calculations were given names (res0, res1 etc) by Kojo to make it easy for you to refer to them in subsequent calculations. Here, you see how you can define your own named values, via the use of the **val** *keyword*:

```
val size = 100
```

A *keyword* is a special word in your program that has a particular meaning for Kojo. So how is an instruction that contains a *keyword* different from a *command* or *expression*? Broadly speaking, *commands* and *expressions* let your programs do things and compute results; *keyword* instructions allow you to structure and organize the *commands* and *expressions* in your programs in better ways.

Within the Script Editor, any keywords that you type will look different from the other words in your program. By default, keywords are blue and bold. Take a look at the above program within your Script Editor. Does the val keyword look different? Instructions with keywords (let's call them **keyword instructions**) *are the third type of instruction that you have seen within programs* (the other two being commands and expressions).

In the program above, you see expressions being used in a couple of different ways:

- `val size = 100.`
  Here, the right hand side of the = sign is an expression (in this case, the value 100).
- `forward(size / 2)`
  Here, the input to the forward command is the result of evaluating the expression `size / 2`

This gives us useful insights into how expressions are used within programs. Imagine a program like this:

```
forward(100)
2 * 4 + 7
right()
```

Guess what happens to the value of the expression in the second line of the program?

It disappears into the ether and gets lost! The program cannot make any use of it.

So how can programs make use of the result values of expressions? A couple of ways are identified above, and are explained in more detail here:

- By capturing the result in a named value. e.g. `val size = 100`. This named value can then be used throughout the program.
- By passing the result as an input into a command. e.g. `forward(size / 2).` The command can then make use of the result of the expression.

Numerical expressions in Kojo are related to the Mathematical expressions. Let's take a whirlwind tour of expressions in Math, and see how they relate to expressions in Kojo...

Arithmetic deals with numbers, and operators (e.g +, -, x, ÷) that work on numbers. Arithmetic expressions are made out of numbers and operators, and provide numeric values as results when they are evaluated.

Here are some examples of arithmetic expressions:

- 9 (evaluates to 9 itself)
- 3 * 4 (evaluates to 12, based on a call to the * operator)
- 2 * 3 + 5 * 8 (evaluates to 46, based on calls to the * and + operators)

The operators (if present) in an expression are used to carry out the calculations within the expression, and the result of these calculations becomes the result of the expression.

As we move from arithmetic to algebra, we encounter variables. Variables are letters, like x and y, that represent numbers.

In algebra, expressions still evaluate to numeric values. But now they contain variables. Some examples:

x

$3x^2+4x+7$

In general, expressions are built out of terms. Each term has factors, which might be constants or variables. For example:

- The expression $3x^2+4x+7$ has three terms: $3x^2$, $4x$, and 7
- The term $4x$ has two factors: 4 and x . Here 4 is a constant and *x* is a variable.

So what does all of this have to do with Kojo?

Arithmetic expressions in Math (remember, such expressions don't contain variables) are exactly the same as arithmetic expressions in Kojo. They both result in a number when they are evaluated. For example:

$2\times5+7\div14$ (Math)

`2*5 + 7/14` (Kojo)

Mathematical expressions that contain variables are very similar to numerical expressions in Kojo, with named values in Kojo playing the role of variables.

But the way in which these expressions are used in Math is different from the way they are used within Kojo.

In Math, *expressions involving variables are used*, along with the equality symbol, *to make statements*:

- Some of these statement are true for all numbers (e.g. a + b = b +a), and are called laws.
- Other statements are true for some numbers (e. g. $y=2\times x+5$), and are called equations. The example equation above specifies the relationship between the two variables x and y. If you know x, you can determine y, and vice-versa.

In Kojo, *expressions involving variables are used*, along with the equality symbol and the val keyword, *to attach names to values* (to create named values) - for example:

```
val y = 2*x + 5
```

Within Kojo, this is not a mathematical statement specifying a general relationship between the variables x and y. Instead, this is an instruction that does a very specific thing – it evaluates the expression on the right-hand-side, and *binds* it to the name on the left-hand-side, to create the named value `y`.

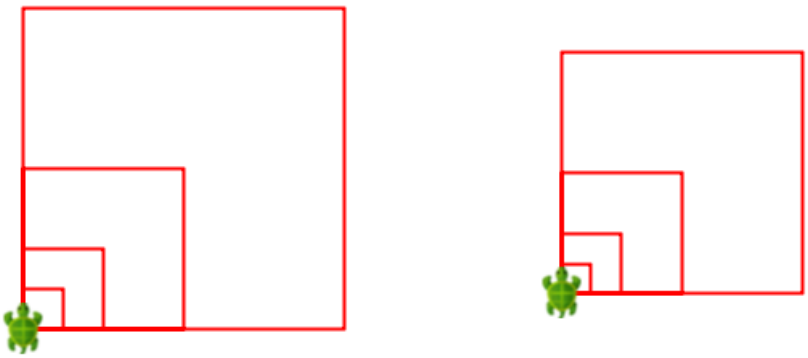
The expression on the right-hand-side above can only be evaluated if `x` is known. So, for example, this will work:

```
val x = 2
val y = 2*x + 5
```

Here, the first instruction creates the named value `x`, and the second instruction uses `x` to create the named value `y` (`y` becomes `2*2 + 5`).

Exercise

Write a program to make the following two figures (in two different runs). You are expected to change just one line of the program before running it a second time (to make the second figure).



Theory Exercises

1. Write down an expression that uses no operators.
2. Write down an expression that uses one operator.
3. Write down an expression that uses two operators.
4. Write down a Mathematical statement. Write down a similar Kojo instruction. How are they different?

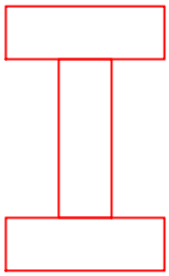
Activity Checklist

It's good if you know the answers to the following questions before moving on:

1. What's a **named value**? What are the benefits of using **named values?**
2. How can you create your own **named values**?
3. What are **keywords?**
4. What is an expression? What are operators?
5. When is the result of an expression not usable inside a program.
6. What are a couple of ways to capture the results of an expression and use them in a program.

Activity 5 - Practice Session I

1. Write a program to make the following figure:

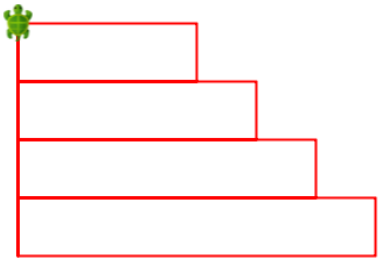


Hint:

All three rectangles are of the same size.

2. Make the following - in such a way that you need to change just one line of your program to change the size of the figure (the next time you run the program):

3. Draw the following, using two named values to specify the lengths and breadths of the rectangles:



Hint:

Only the length changes here

4. Now draw the following figure (again using two named values):

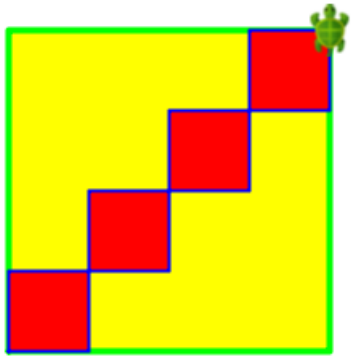
Hint:

Both length and breadth change in this case.

Activity 6 – A Touch of Style

Type in the following program and run it:

```
clear()
setAnimationDelay(100)

// make the big green/yellow square
setPenColor(green)
setPenThickness(4)
setFillColor(yellow)
repeat (4) {
    forward(200)
    right()
}
// make the four small blue/red squares
setPenColor(blue)
setPenThickness(2)
setFillColor(red)
repeat (4) {
    // make a single small square
    repeat (4) {
        forward(50)
        right()
    }
    // position the turtle for the next square
    penUp()
    forward(50)
    right()
    forward(50)
    left()
    penDown()
}
```



This program introduces a lot of new things. Let's start by looking at the new commands:

setAnimationDelay(delay) – Allows you to set the turtle's speed. The specified delay is the amount of time (in milliseconds) taken by the turtle to move through a distance of one hundred steps. The default delay is 1000 (1 second). A smaller delay makes the turtle move more quickly.

penDown() – Makes the turtle put its pen down - to draw lines as it moves. The pen is down by default.

penUp() – Makes the turtle not draw lines as it moves.

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:

SetPenThickness(1)

SetPenThickness(2) - ***Default***

SetPenThickness(4)

Color values – till now, you had seen only number values (1, 2, 3 etc). In this activity, you see values of another type – Color. Some examples of Color values are - red, blue, green etc.

Color is the third *type* you have seen. Can you remember the other two?

Let's dig a little deeper into *types*. As mentioned in Activity 3, the type of a value tells you what operators and commands it can work with.

Integers and Doubles (the types you saw earlier) work with the standard arithmetic operators (+, -, *, /).

Colors do not work with these operators. Try evaluating (within Kojo) the expression:

```
blue + green
```

Kojo will respond with a type error, telling you that this kind of addition is not possible.

Let's now see how types work with commands. We know that the forward command expects a number as an input (e.g. `forward(100)`). Try running the forward command with a color:

```
forward(blue)
```

Kojo will again respond with a type error.

These examples start to show us some of the benefits of types. Every command or operator within Kojo specifies the types that it works with. If you provide it the wrong type of input, Kojo will catch the error right away and let you know. This becomes very useful as you start to write larger programs.

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.

setFillColor(color) – specifies the background color of the figures drawn by the turtle. Example: setFillColor(blue) will fill the figure that you draw with blue color.

Next, this program has comments describing what it does. **Comments** start with `//` and continue to the end of the line. Comments are ignored by the computer. They are put there only for the human reader of a program.

Writing comments within your programs is a very good idea, because it makes it easy for you (and your friends) to take a look at the program and easily understand what each section of the program is doing.

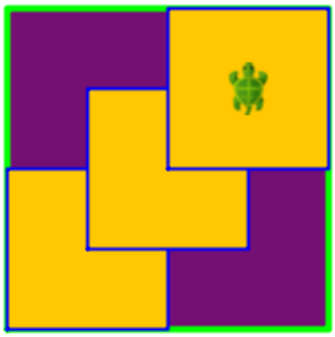
Finally, this program introduces a very important idea – when you are drawing a non-trivial sketch, your program should be structured in such a manner that it has sections that actually draw things on the screen **(Drawing code)**, and sections that just move the turtle and position it for subsequent drawing **(Positioning code)**. These sections should be clearly separated by comments. The fragment of code below (taken from the above program) shows this in action:

```
repeat (4) {
    // Drawing code: make a single small square
    repeat (4) {
        forward(50)
        right()
    }
    // Positioning code: position the turtle for the next square
    penUp()
    forward(50)
    right()
    forward(50)
    left()
    penDown()
}
```

In general, positioning code should start with a `penUp` command and end with a `penDown` command.

Exercise

Write a program to make the following figure:



Activity Checklist

It's good if you know the answers to the following questions before moving on:

1. What do these commands do?
   a) `setAnimationDelay(delay)`
   b) `penDown()`
   c) `penUp()`
   d) `setPenThickness(number)`
   e) `setPenColor(color)`
   f) `setFillColor(color)`
2. Why should you write **comments** in your programs? How do you write comments?
3. What is *drawing* code? What is *positioning* code? How should you structure your program when you're making a drawing?

Activity 7 – Commands of your own

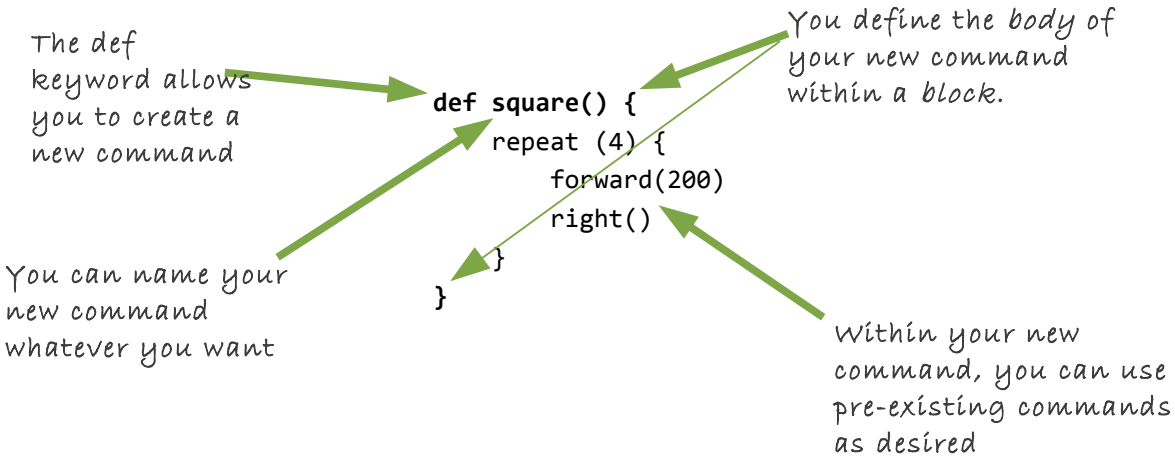
Type in the following code and run it:

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

clear()
square() // call the user defined command to make use of it
```

You see something new here – a user defined command: `square`.

The **def** *keyword* allows you to create new commands of your own. You can then call these commands just like you would call predefined Kojo commands.

The initial portion of the program above defines the square command. Here's a closer look at that fragment of code:

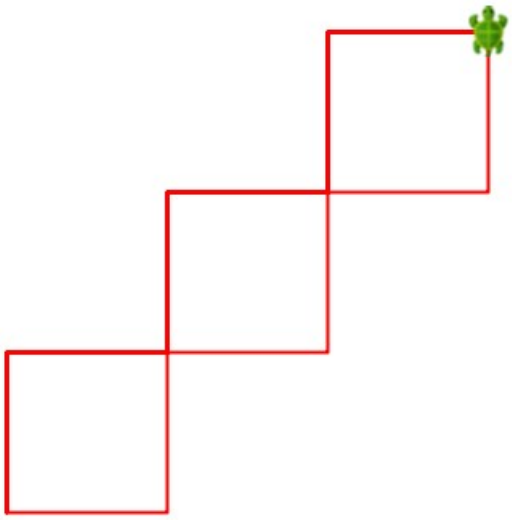


What's the benefit of creating your own commands?

These commands allow you to capture commonly used patterns of code, give them a name, and then reuse them. This reduces code duplication, and makes your programs easier to understand.

Exercise

Write a program (which uses the square command) to make the following figure:



Hint. You can refine your program from Activity 2 to come up with the desired program.

Activity Checklist

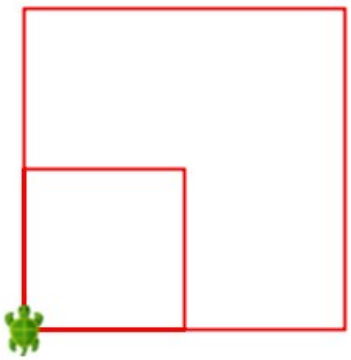
It's good if you know the answers to the following questions before moving on:

1. What are **user defined** commands?
2. How can you create a **user defined** command?
3. Why would you create a **user defined** command?

Activity 8 – Inputs to your own commands

Type in the following code and run it:

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

clear()
square(100)
square(200)
```



In the code above, you see another new feature of Kojo – inputs to your commands. When you define a command, you can specify that it needs an input:

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

In the bold portion of the code above, you are telling Kojo that the input to the square command is called ***side***, and that its type is ***Int*** (where, as you have seen earlier, Int stands for integer). Now, instead of always drawing squares of the same size, 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 (do you remember named values from Activity 4?).

Inputs also have ***types*** associated with them. The type of an input tells Kojo:

- the permissible values of the input.
- the operators and commands that it can work with.

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.

Once you have defined a command that requires inputs, you can *call* it with inputs of the appropriate type – to make it do its work:

```
square(100)
square(200)
```

Just like you can create user defined commands, you can also create user defined operators that can be used within expressions.

User defined operators are called functions if they are named using alphanumeric characters (like a, b, c, 1, 2, 3, etc) as opposed to operator characters (like *, +, <, etc). They are closely related to the mathematical idea of functions. The next book in this series will talk (much) more about this topic.

For now, just to give you a taste, here's how you can define a function that does something very similar to the + operator:

```
def sum(x: Int, y: Int) = {
    x + y
}
```

And here is how you call that function:

```
sum(2, 3)
```

Note that you create a user defined operator (or function) in a manner that is very similar to how you create a user defined command. The big difference is the use of the equals sign on the first line of the definition:

```
def sum(x: Int, y: Int) = {
```

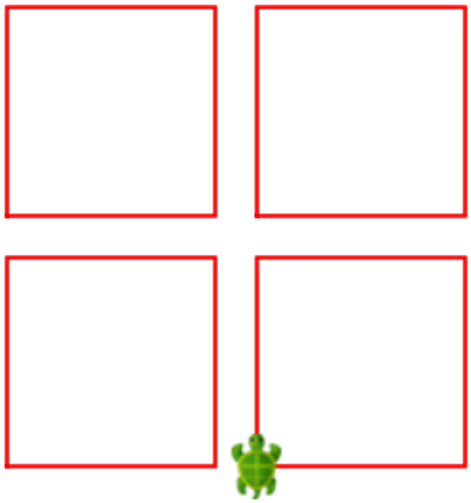
User defined commands do not have this = sign.

This is meant to signify that user defined operators are *equivalent* to the value that they calculate - and return to the caller - based on the inputs provided to them. In the above example, `sum(2, 3)` is equivalent to `5` – because it takes two inputs: `2` and `3`, and returns the value `5`.

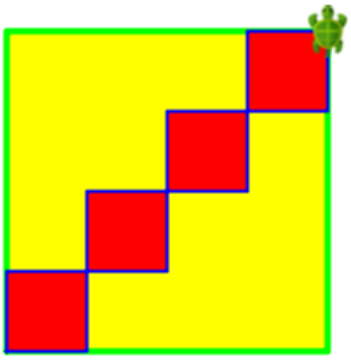
User defined commands, on the other hand, are not equivalent to anything – because they don't calculate and return any values; instead, they just take some action.

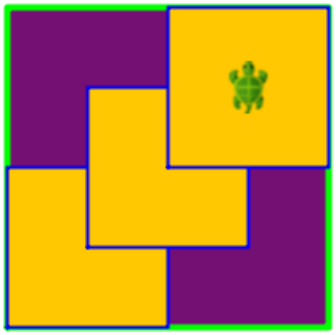
Exercises

1. Write a program to make the following figure. Use the square command (which takes an input value) to make the figure.



2. Remake the figures from Activity 6 using the square command (which takes an input value):





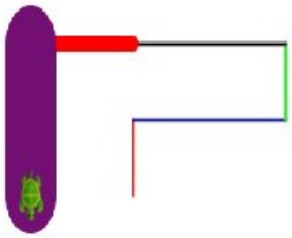
Activity Checklist

It's good if you know the answers to the following questions before moving on:

1. What are **inputs** to commands?
2. How do you create **inputs** for your own command?
3. What are the advantages of using **inputs** in your commands?

Activity 9 - Practice Session II

1. Try making the following figure:

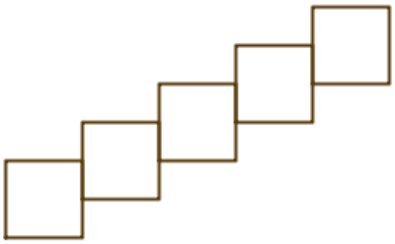


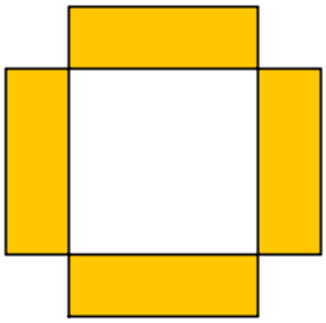
Hint:
make use of setPenColor() and setPenThickness()

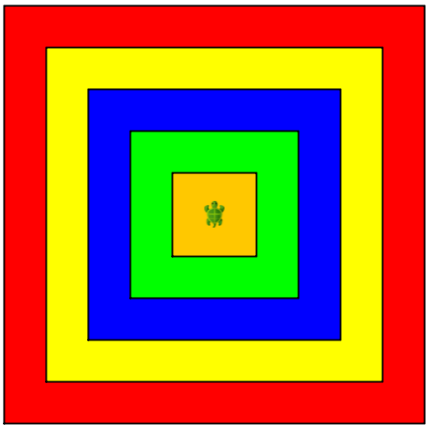
2. Make the following pattern:

Hint:
Start with the biggest square and then go down in size. If you start with a smaller square and then draw a bigger square over it, the smaller one will not be visible.

3. Create your own commands and then use them to make the following figures:

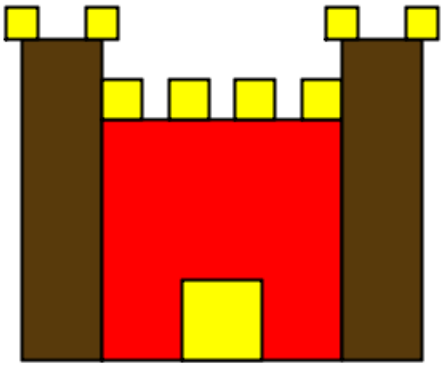






Activity 10 – Mini Project #1

Write a program to make the following figure:



Done?

Now compare your solution to the solution provided in the last section of the book. The provided solution demonstrates a lot of ideas and techniques for writing larger programs. Make sure you read the solution carefully and understand it fully.

Activity 11 – Break Free #1

For the next few days, just make your own creations with Kojo. Sketch out an idea on paper, and then implement it within Kojo. Just remember – the ideas that you try out should contain just straight lines – horizontal and vertical (that will work best with the commands that you have learnt till now).

And feel free to share your creations with us on the Kojo *Code Exchange*.

The *Code Exchange* is a website to which you can post your Kojo sketches and code - with one click of a button within Kojo.

This *Code Exchange* is a good place for Kojo users to showcase their work, look at and rate the work of others, provide comments and exchange ideas, and learn in a collaborative fashion.

Activity 12 – Geometry Interlude

Till now, you have played with making shapes with the turtle moving forward and turning right or left. The right and left turns were perpendicular (or 90 degree angle) turns.

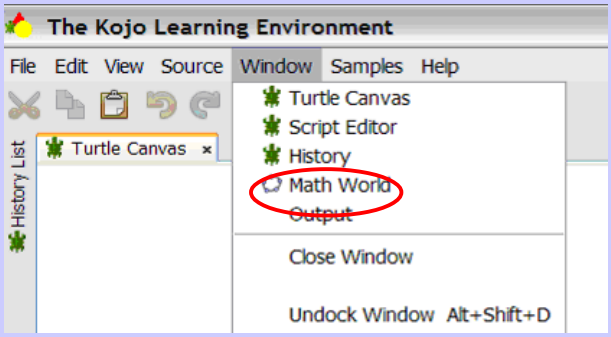
It's now time to get the turtle to make turns of angles other than 90 degrees. But before we do that, it'll be good for you to play with angles - to understand them better. Let's start this journey with a quick geometry primer.

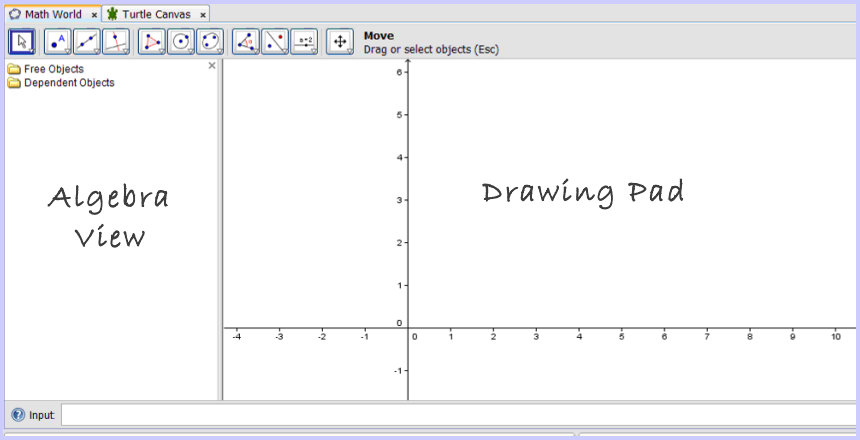
In order to play with and become familiar with different Geometrical concepts, you will use the *Math World* Window inside Kojo.

Ok, so how do you get to *Math World?*

To open up *Math World*, just click on the Windows Menu, and then select Math World.

This will open up the *Math World* Window for you.

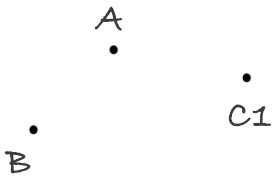




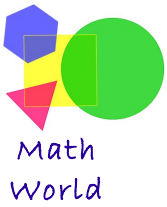
Points

Geometry, as you know, is the study of shapes (just like Arithmetic is the study of numbers). Shapes live within space. To start talking about shapes, the first thing we need to be able to do is to identify a *location* or *position* within space. A *point* allows us to do exactly this. We can think of a point as a dot on a piece of paper (at a particular location), or for that matter, on a computer screen (again at a particular location).

In order to identify a point, we usually give it a name using a letter and optionally a number. For example, the following points are identified as A, B and C_1

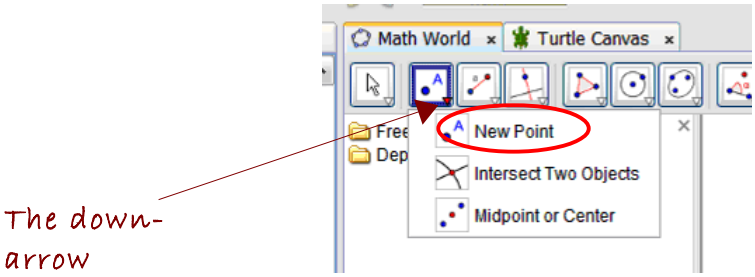


.



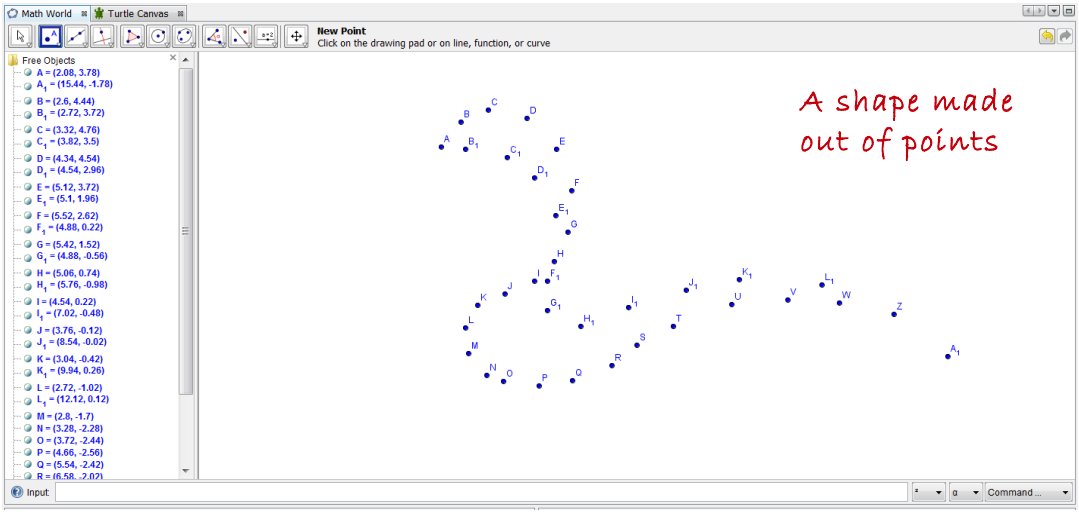
Let's play with points in *Math World*.

Open up *Math World* and click on the small down-arrow on the bottom-right of the "Point" button (as shown below). In the drop down menu select "New Point".



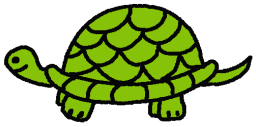
Now go and click inside the "Drawing pad" area. You will see that a point has been created at the place where you clicked, with a name (A, B, C etc) associated with it.

Click around inside the drawing pad and try to make a shape out of points (an example is shown below).



Here's one way that points relate to the Turtle Canvas:

The turtle is always at a particular location or position within the Turtle Canvas. This position is specified by a Point. You can find the turtle's current position by calling the **position()** function.



Lines

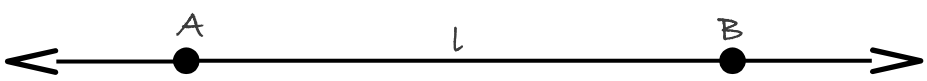
Line

In geometric terms, a line is a collection of points that extends forever in both directions.

The main characteristics of a line are:

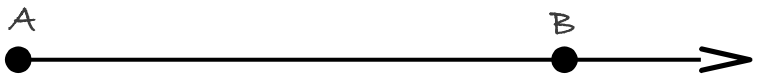
1. It is one dimensional, i.e., it has length, but no width and height.
2. It is straight.
3. It never ends.

In order to show that a line keeps going forever, we use arrows at both ends when we draw lines. A line is usually identified by a lowercase letter (like, "l" in the following example), or by marking two points on the line and then using them as the name with a line drawn over them. So the line below can be referred to as either "l" or as $\overleftrightarrow{AB}$.



Ray

A ***ray*** is a line that starts from one endpoint, and extends forever in the other direction. The following is a ray which starts from point A and passes through point B. It is denoted as $\overrightarrow{AB}$.



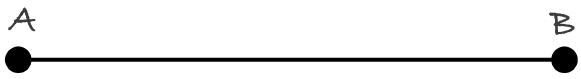
So, the main characteristics of a ray are:

1. It is one dimensional, i.e., it has length, but no width and height.
2. It is straight.
3. It has only one endpoint.
4. It extends forever in the other direction.

Line Segment

A line segment is a part of a line. Unlike a line, it does not extend forever, but has two endpoints. For example, in the previous figure of the line, the portion of the line between the points A and B is a line segment. A and B are the endpoints of the line segment.

A line segment can be drawn as a part of a line (as in the previous figure) or separately with two endpoints like in the following figure.



The name of a line segment is made up of its two endpoints with a line over them, and therefore the name of the above line segment is $\overline{AB}$.

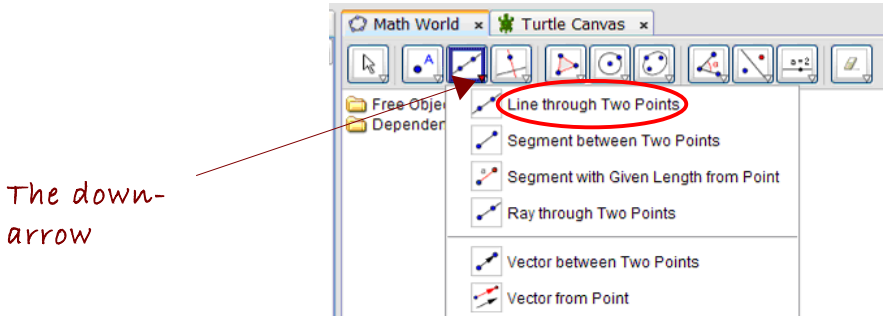
So, the main characteristics of a line segment are:

1. It is one dimensional, i.e., it has length, but no width and height.
2. It is straight.
3. It has two endpoints.
4. It can be a part of a line, or can be drawn separately.

Note – although a finite sized line between two points should strictly speaking be called a line segment, we sometimes just call it a line in this book - for ease of use.

Let's explore lines in *Math World*.

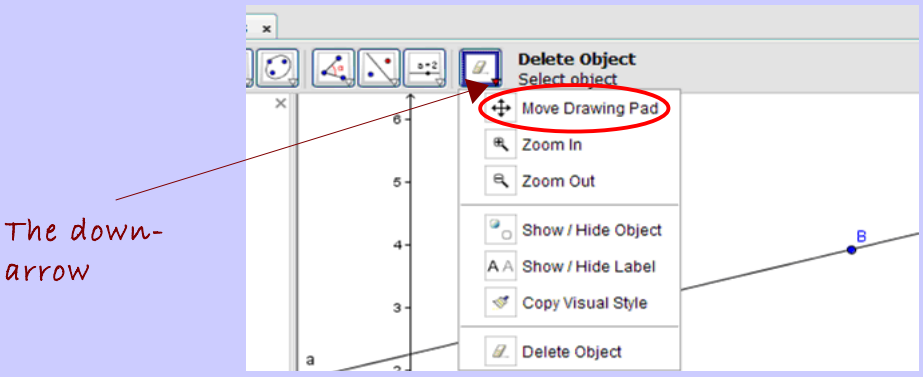
Open *Math World* and click on the down-button on the bottom-right of the "Line" button (the third one from the left). From the drop down menu select "Line Through Two Points". As the name suggests, using this tool you will be able to draw a line which passes through two points that you make on the drawing pad.



Now, go to the drawing area and click at the location where you want your first point. You will see a point appear where you clicked. Next, go and click where you want the second point to be. You will see a second point appear there, and a line will be created passing through the two points that you just made.

Make sure that the line extends forever in both directions by moving the Drawing Pad around and trying to get to the end of the line.

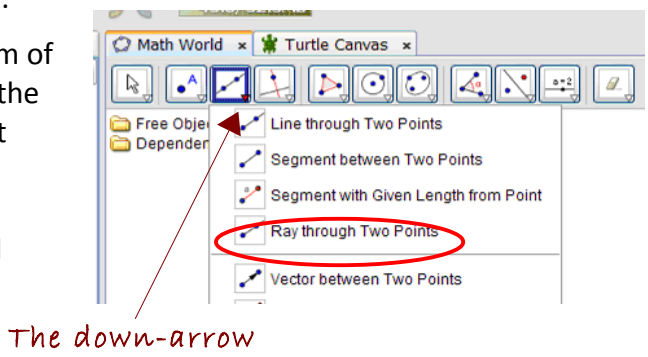
To move the Drawing Pad, select the dragging tool by clicking on the down-arrow on the last button on the right, and then selecting "Move Drawing Pad". This is shown in the figure below.



Now click on any point on the Drawing Pad and then drag the mouse around (holding down the left button). You will see that you can move the Drawing Pad around.

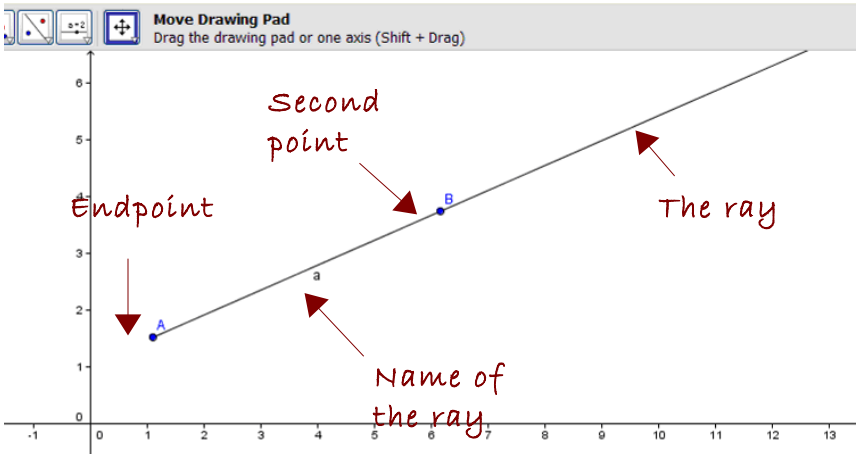
You can also make rays in *Math World*.

Click on the down-arrow on the bottom of the "Line" button (the third one from the left). From the drop down menu select "**Ray Through Two Points**". As the name suggests, this tool helps you to make rays starting from one point and extending forever through a second point.



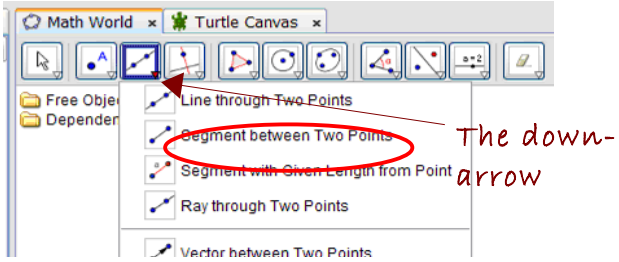
Go to the drawing area and click where you want your first point. You will see a point appear there. Next, go and click where you want the second point to be. You will see a second point appear there, and a ray will be created starting at the first point and passing through the second point.

You can make sure that this is in fact a ray by dragging the drawing board and trying to get to the end of the ray.



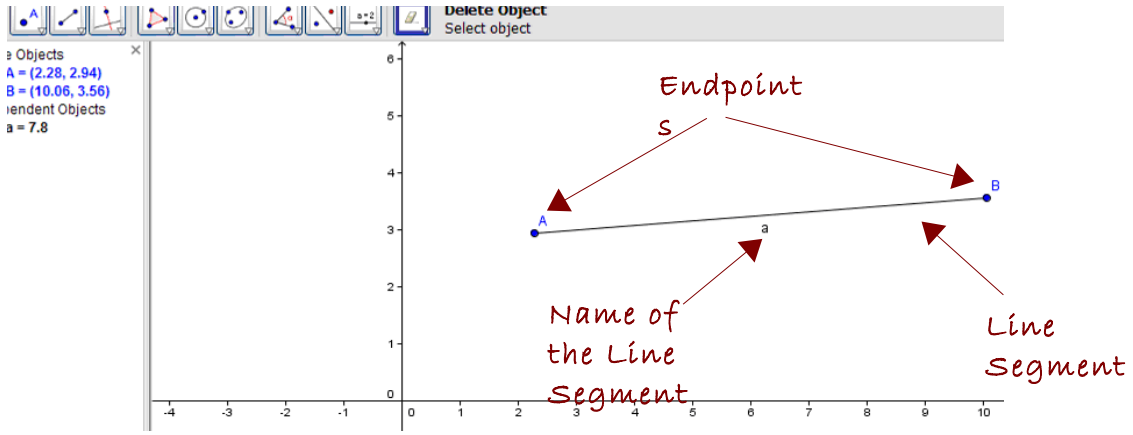
Next, let's explore line segments within *Math World*.

Click on the down-arrow on the bottom of the "Line" button (the third one from the left). From the drop down menu select "Segment Between Two



Points". This tool will allow you to make a line segment on the drawing pad between any two points that you want.

Go to the drawing area and click where you want your first point. You will see a point appear there. Next, go and click where you want the second point to be. You will see a second point appear there, and a line segment will be created between the two (end)points.

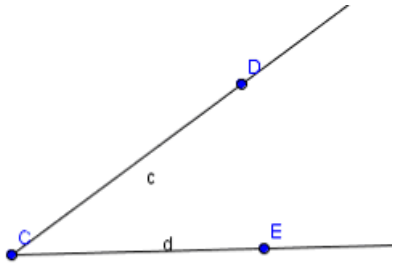


Play around by making line segments of different sizes and with different endpoints. Try to create an interesting shape out of them (like a house).

Angles

As soon as two rays or line segments intersect, they form an angle. Their angle of intersection tells us something about their relative direction. If the rays are almost parallel, their angle of intersection is small. If they are almost perpendicular, their angle of intersection is large.

If two rays meet at their endpoint, this point is called the **vertex** of the angle that is formed between the rays. The two rays are called the **sides** of the angle. In the figure to the right, C is the vertex of the angle, and the rays CD and CE are the two sides of the angle.



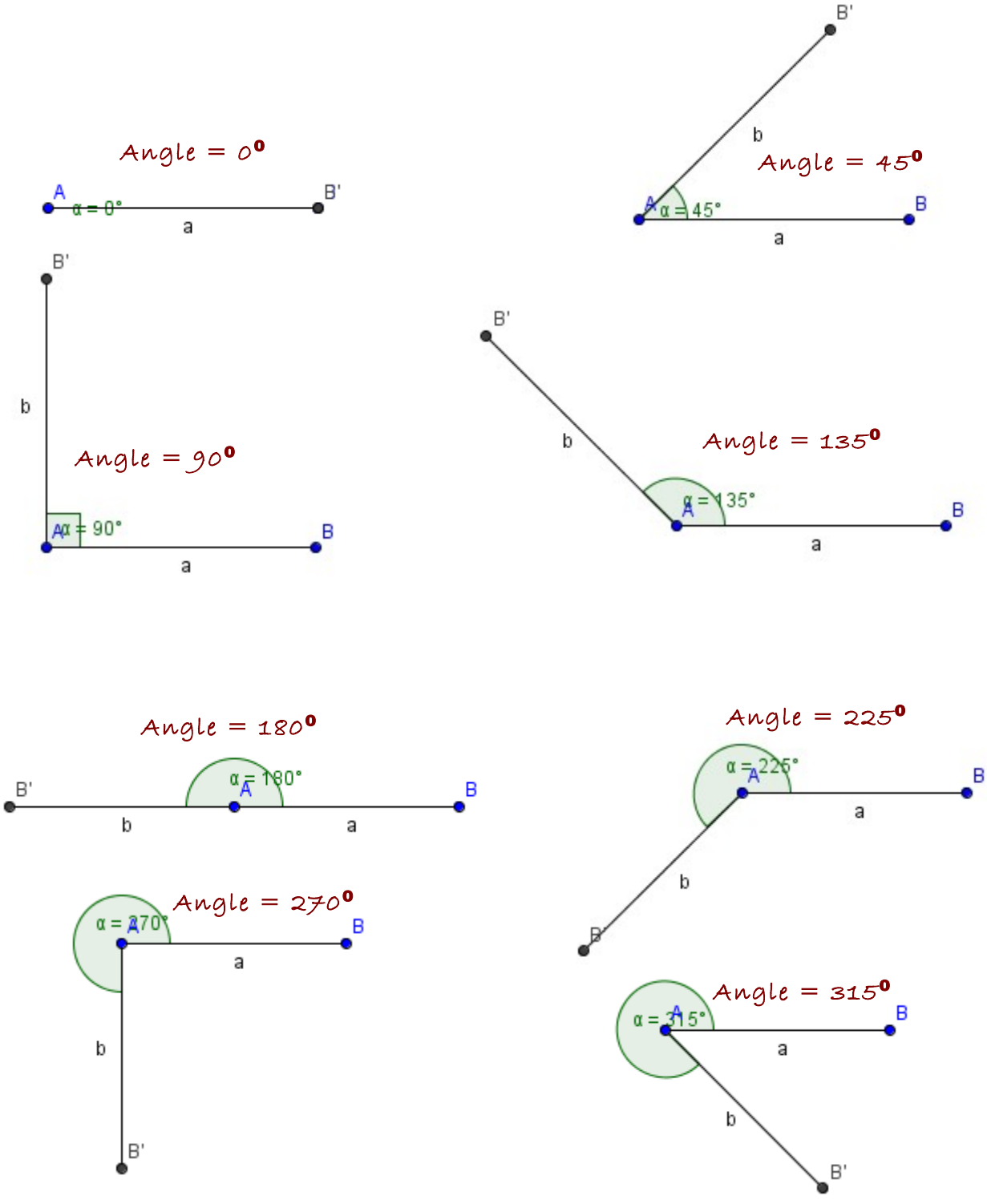
An angle is named using the name of a point on each of the rays and the vertex of the angle, and is denoted by the sign $\measuredangle$. For example, the angle above can be referred to as $\measuredangle$ DCE or $\measuredangle$ ECD.

Angles are measured in degrees (denoted as ^{o}) or radians (we'll focus on just degrees in this book). If you put two rays on top of each other, the angle between them is 0^o. As you move/turn the first ray away from the second, keeping both their endpoints at the same location, the size of the angle increases. If you turn the first ray all the way around till it is back pointing in the direction of the second, you will have covered an angle of 360^o. Thus, angles are also intimately tied to the idea of turning or rotating. A turn through a full circle corresponds to a turn of 360^o.

Why does it take 360^o to turn through a full circle?

There's no mathematical reason for this. This fact probably came about because ancient astronomers noticed that the stars travel in the sky along a circular path, and it takes them about 360 days to move through a full circle (because the Earth travels around the Sun and gets back to its starting point in a year, which is 365 days). The astronomers thus divided a full turn through a circle into 360 parts (called degrees) – to have a meaningful unit of measurement for circular motion.

The figures below shows a few different angle of different magnitudes:

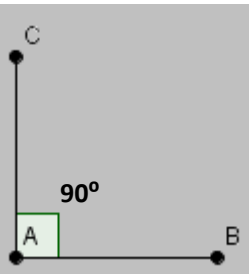


Notice how for each of the angles above, the size is shown within an arc (a partial circle). As the size of the angle increases, so does size of the arc. When the angle is 180^o the arc is a semicircle, and for an angle of 360^o the arc becomes a full circle.

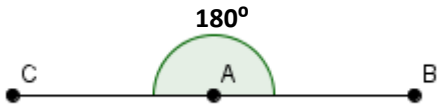
Types of angles

Angles are given different names depending on their size . The following are the main types of angles:

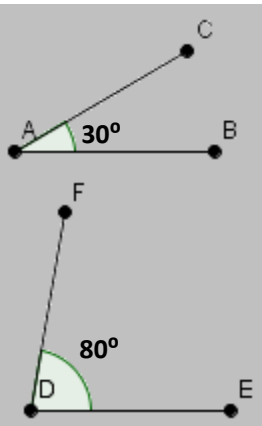
1. **Right Angle:** An angle which measures 90 degrees is called a **Right Angle**. The two lines that form a right angle are called **perpendicular lines**.



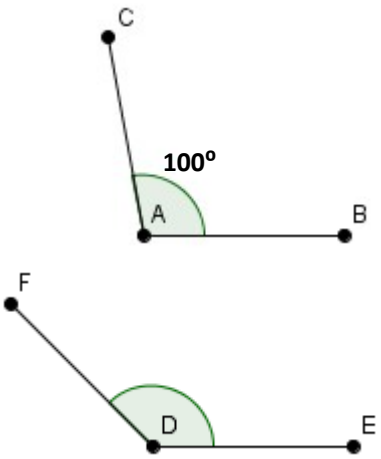
2. **Straight Angle:** An angle that measures 180^o is called a **Straight Angle**.



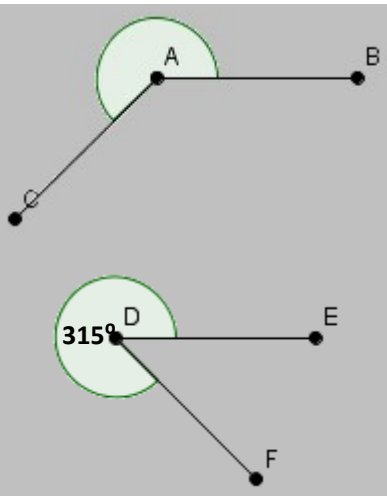
3. **Acute Angle:** Angles that measures between 0^o and 90^o are called **Acute Angles**. For example, a 30^o angle is an acute angle, and so is an 80^o angle.



4. **Obtuse Angles:** Angles that measure between 90° and 180° are called **Obtuse Angles**. For example, a 100° angle and a 135° angle are both Obtuse angles.



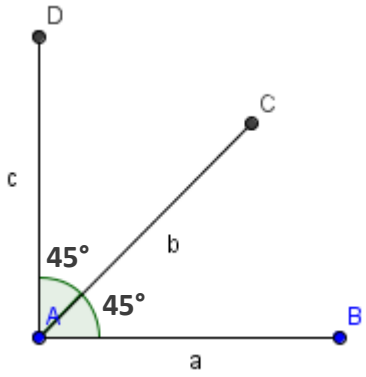
5. **Reflex Angles:** Angles which are between 180° and 360° are called **Reflex Angles**. For example, a 225° angle and a 315° angle are reflex angles.



Complimentary and Supplementary Angles

Two or more angles can be drawn in such a manner that they have a common side. For example, in the following figure:

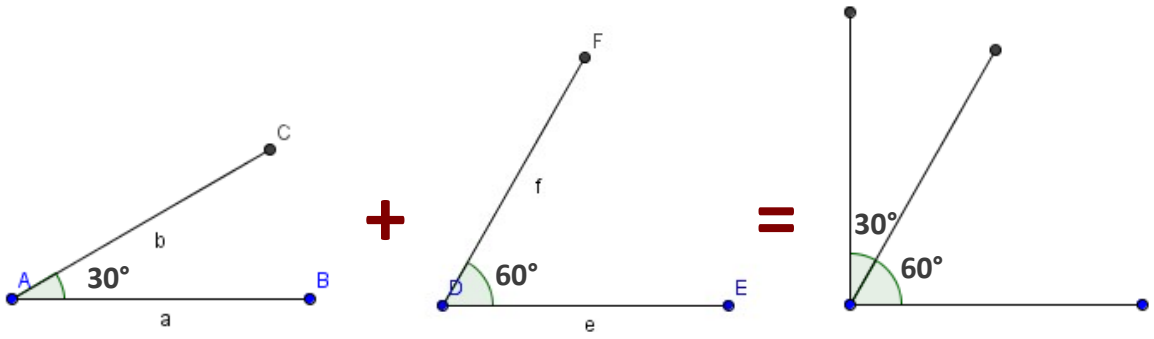
1. ∢ DAC and ∢ CAB share a common side AC (and they don't overlap).
2. ∢ DAB and ∢ CAB have the same side AB (and they overlap).



Complimentary Angles

When the sum of two angles is equal to 90°, they are called complimentary angles. The following two angles ∢ CAB and ∢ FDE are complimentary angles because 60°+ 30°= 90°.

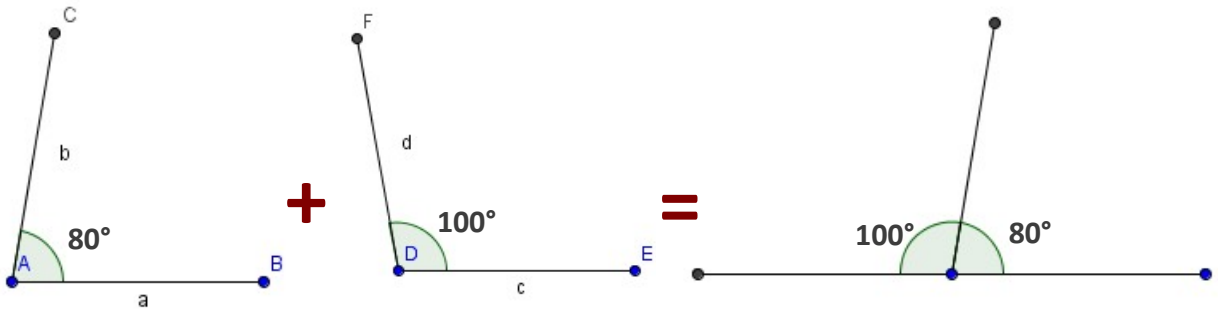
If the two complementary angles are put together, such that they have a common side and they don't overlap, we get a right angle.



Supplementary Angles

When the sum of two angles is equal to 180°, they are called **Supplementary Angles**. The following two angles ∢ CAB and ∢ FDE are supplementary angles because 80°+ 100°= 180°.

When two Supplementary Angles are put together, such that they have a common side and they don't overlap, they form a straight line.



Angles in Kojo

Within Kojo, you get to work with angles in the following two ways:

- Experimenting interactively with angles of different shapes and sizes within *Math World.*
- Making shapes and sketches that involve angles on the Turtle Canvas.

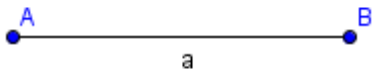
The rest of this Activity guides you in experimenting with angles within Math World.

Math World

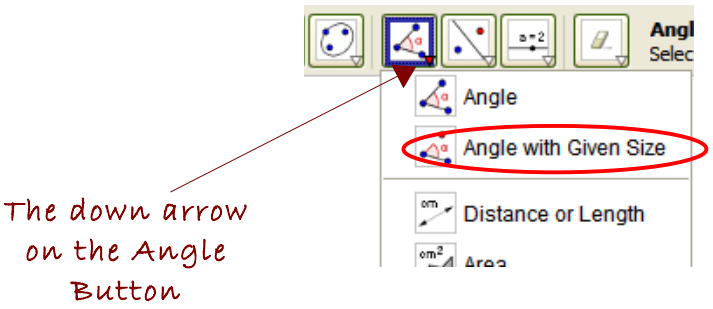
Experimenting with Angles in *Math World*

Follow these steps to make an angle:

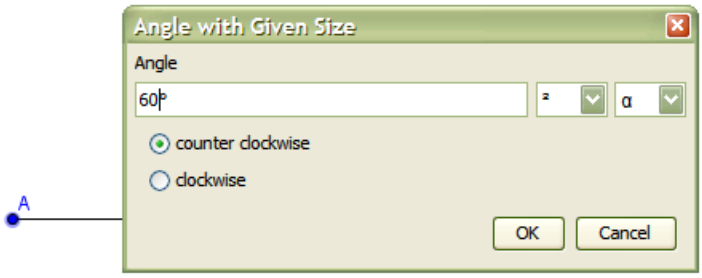
1. Open *Math World*, and draw a line segment between two points. This line segment will form one of the sides of the angle.



2. Click on the small down-arrow on the bottom-right of the "Angle" button (the fourth button from the right, as shown below) and select "Angle with Given Size" in the drop down menu.

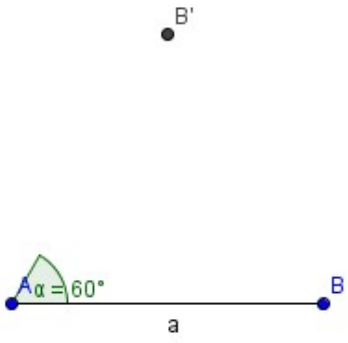


3. Click on that endpoint of the line segment that you want to be at the open end of the angle (point B).
4. Click on the endpoint that you want to be the vertex of the angle (point A).
5. A window will pop-up asking you for the size of the angle you want to make, and the direction of the angle. Enter 60°, and selected counter clockwise.

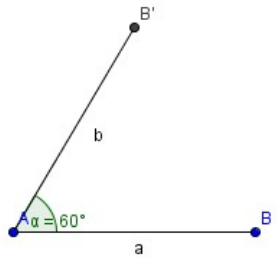


> Make sure you have a ° sign next to the size of the angle that you specify. If you forget to put in the ° sign, Kojo will assume that you are specifying the size of the angle in radians, and your angle will not come out like you expect.

6. Once you click OK after giving the size and direction of the angle, you will see a point that lies on an imaginary line at the desired angle from the given line segment.



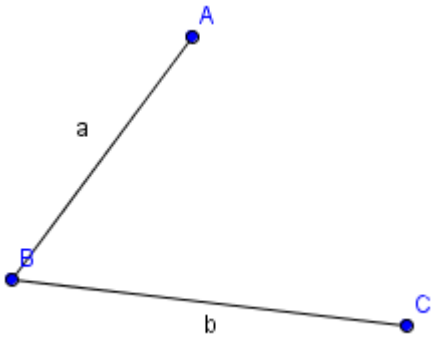
7. Draw a line segment between the vertex A and the new point B', and you will get a 60^{o} angle.



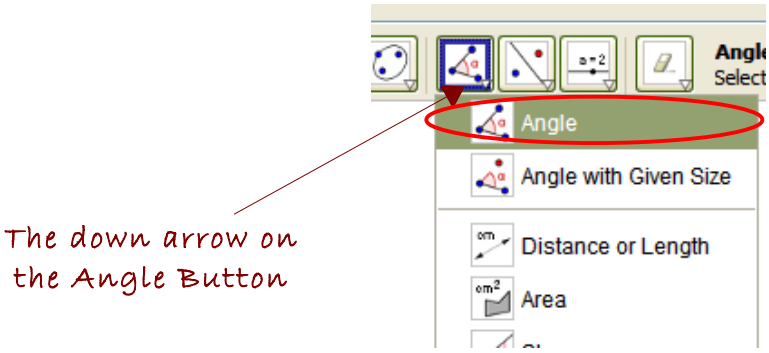
Measuring an angle in *Math World*

In *Math World*, you can also measure a given angle to find out its size. Let's take a look at the steps you need to follow in order to determine the size of an angle:

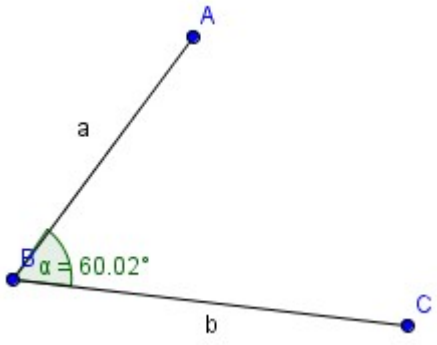
1. Make the following figure using the Line Segment tool:



2. Click on the small down-arrow on the bottom-right of the Angle Button and select "Angle" in the drop down menu.



3. Click on the three points A, B and C on the angle, and the measurement in degrees will be displayed within the angle.



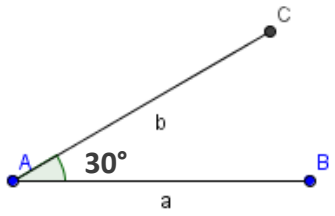
To measure an angle, you need to click on the first point, then the vertex, and finally on the third point. Math World will then show you the angle that is formed when you move counter-clockwise from the first point to the third (the reason for this is explained in the next section).

So, for a figure like the one above, if you want your measured angle to be less than 90° in size, you should measure CBA – by clicking first on C, then B and then A.

Also try reversing this order (click first on A, then B, and finally C) and then see what happens.

Negative Angles

By convention, the counter clockwise direction is considered to be the positive direction for angles, and the clockwise direction is considered to be the negative direction for angles. So, for example, in the following figure:



$\measuredangle$ BAC = 30°, while $\measuredangle$ CAB = - 30°. This is because, in $\measuredangle$ BAC, you have to move in a counter-clockwise direction to go from B to C (the first point to the third point), whereas in $\measuredangle$ CAB, you have to move in a clockwise direction to go form C to B (the first point to the third point).

If you measure $\measuredangle$ CAB within Math World, it will come out to be 330°. That is because Math World always measures angles in the counter-clockwise (positive) direction.

Something to think about: If turn by -30°, or if you turn by 330°, you end up pointing in the same direction. Why?

Activity Checklist

It's good if you know the answers to the following questions before moving on:

1. What's a line? A Ray? A Line Segment?
2. What's an angle? How are angles measured?
3. Why are there 360 degrees in a circle?
4. What's a right angle? A straight angle?
5. What's an acute angle? An obtuse angle? A reflex angle?
6. What are complementary and supplementary angles?
7. What are negative angles?

Activity 13 – Turning Practice

In earlier activities, you have come across a couple of commands - `right` and `left` - that make the turtle turn. Now that you know more about angles, we can be a little more precise about what these commands do. The right() command turns the turtle 90° to the right, and the left() command turns the turtle 90° to the left.

What if you want to make the turtle turn through an angle other an 90°. The following commands allow you to do that:

right(angle) - Turns the turtle to the right by the specified angle.

For example: right(45) will turn the turtle right by 45°

left(angle) - Turns the turtle to the left by the specified angle.

For example: left(30) will turn the turtle right by 30°

turn(angle) - Turn the turtle by the specified angle. Angles are positive for counter-clockwise turns, and negative for clockwise turns.

Examples:

turn(45) will turn the turtle left by 45 degrees

turn(-30) will turn the turtle right by 30 degrees

Some additional commands that help in working with angles are:

beamsOn() - Shows crossbeams centered on the turtle - to help with estimating angles as you work to draw a given shape.

beamsOff() - Hides the turtle crossbeams.

Crossbeams help in visually determining the angle that the turtle is pointing at, in relation to other lines in the figure.

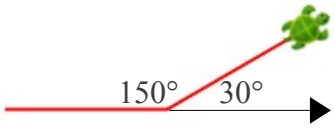
Now let us try using our newly gained knowledge to draw some angles. Type in the following code and run it:

```
clear()
right()
forward(100)
left(30)
forward(100)
```

The above program gets the turtle to make a 30° turn to the left, as shown in the figure above. Can you spot the 30° angle?

What's the angle made by the two red lines in the above figure? Think about that for a moment before reading on...

The figure below should make things clear:

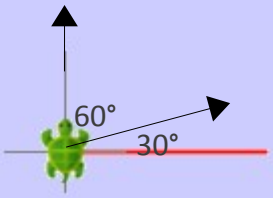


When you are making the turtle turn on the Turtle Canvas, your knowledge of Complementary and Supplementary angles should come in real handy. Switching on the crossbeams also helps while working with angles.

Let's look at a couple of special cases - which come up pretty often when you want to make triangles with angles of particular sizes. In these scenarios, you want to make an angle relative to a horizontal line (which we will call the baseline):

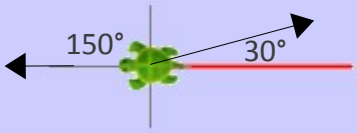
1. When the turtle is perpendicular to the baseline - In this case, if you want to make a 30° angle to the baseline, you need to make the turtle turn through 60° - as per our discussion of Complementary Angles.

   In this case the command will be right(60).



2. When the turtle is facing in the direction of the baseline - If you want to make a 30° angle to the baseline, you need to make the turtle turn through 150° - as per our discussion of Supplementary Angles.

   In this case the command will be right(150).



Exercises

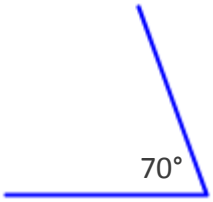
1. Draw the following angles using Math World:

   a) 50° d) 25° g) 150°

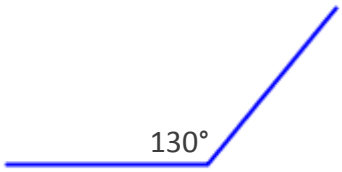
b) 70°
c) 145°
e) 189°
f) 275°
h) 345°
i) 320°
j) An Obtuse Angle
k) A Reflex Angle
l) An Acute Angle

2. Draw the following angles on the Turtle Canvas:

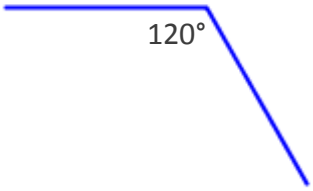
a)



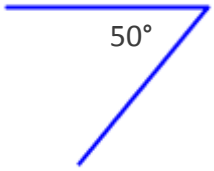
b)



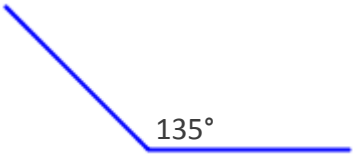
c)



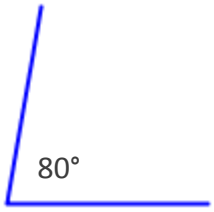
d)



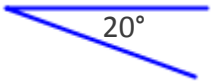
e)



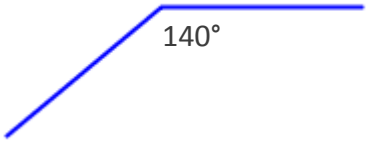
f)



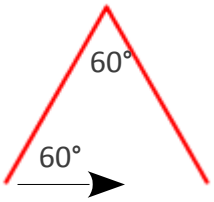
g)

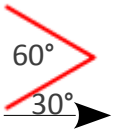


h)



3. Make the following figures (made out of the red lines) on the Turtle Canvas:

a) 

b) 

4. Draw the following letters on the Turtle Canvas:

   a) A

   b) N

5. Make the following (made out of the black lines) on the Turtle Canvas:

Activity Checklist

It's good if you know the answers to the following questions before moving on:

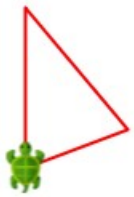
1. What do these commands do?
   - right(angle)
   - left(angle)
   - turn(angle)

   What happens if the angles supplied to these commands are negative?
2. How does your knowledge of supplementary and complementary angles help you in getting the turtle to make angles of the desired size?

Activity 14 – If only

Type in the following code and run it:

```
// makes an isoceles triangle, with equal sides of the
// specified length, and a top angle of the specified size
def isoceleseTriangle(side: Int, topAngle: Int) {
    if (topAngle >= 180) {
        print("Sorry, I can't make this triangle; topAngle should be < 180.")
    }
    else {
        forward(side)
        right(180 - topAngle)
        forward(side)
        home()
    }
}

clear()
setAnimationDelay(100)
isoceleseTriangle(100, 40)
```



There are quite a few new things here:

The **home()** command - moves the turtle to its original location on the Turtle canvas, and makes it point north.

The **if**-**else** *keyword instruction* – allows you to make a choice within your program and select certain portions of your program to run only when certain conditions are true.

This brings up the question – what's a condition?

For now, you can think of a condition as being the result of a comparison between numbers. The following table explains number comparisons in detail:

Operator	Operator name	Example	Example result
<	Is less than	4 < 5	true
		5 < 5	false
		5 < 4	false

Operator	Operator name	Example	Example result
<=	Is less than or equal to	4 <= 5 5 <= 5 5 <= 4	true true false
==	Is equal to	4 == 5 5 == 5	false true
>	Is greater than	4 > 5 5 > 5 5 > 4	false false true
>=	Is greater than or equal to	4 >= 5 5 >= 5 5 >= 4	false true true

You write the if-else instruction like this:

```
if (condition) {
    // do something
}
else {
  // do something else
}
```

The else part is optional, so you can also just write:

```
if (condition) {
    // do something
 }
```

Make sure that you fully understand the program for this activity before moving on...

Exercise

You'll notice that the isosceles triangles made by the above program lean to the left. Write a program that makes isosceles triangles pointing straight up, like in the following figure:

Activity Checklist

It's good if you know the answers to the following questions before moving on:

1. What is a condition?
2. Why do we need an **if-else** instruction?
3. What does the **home()** command do?

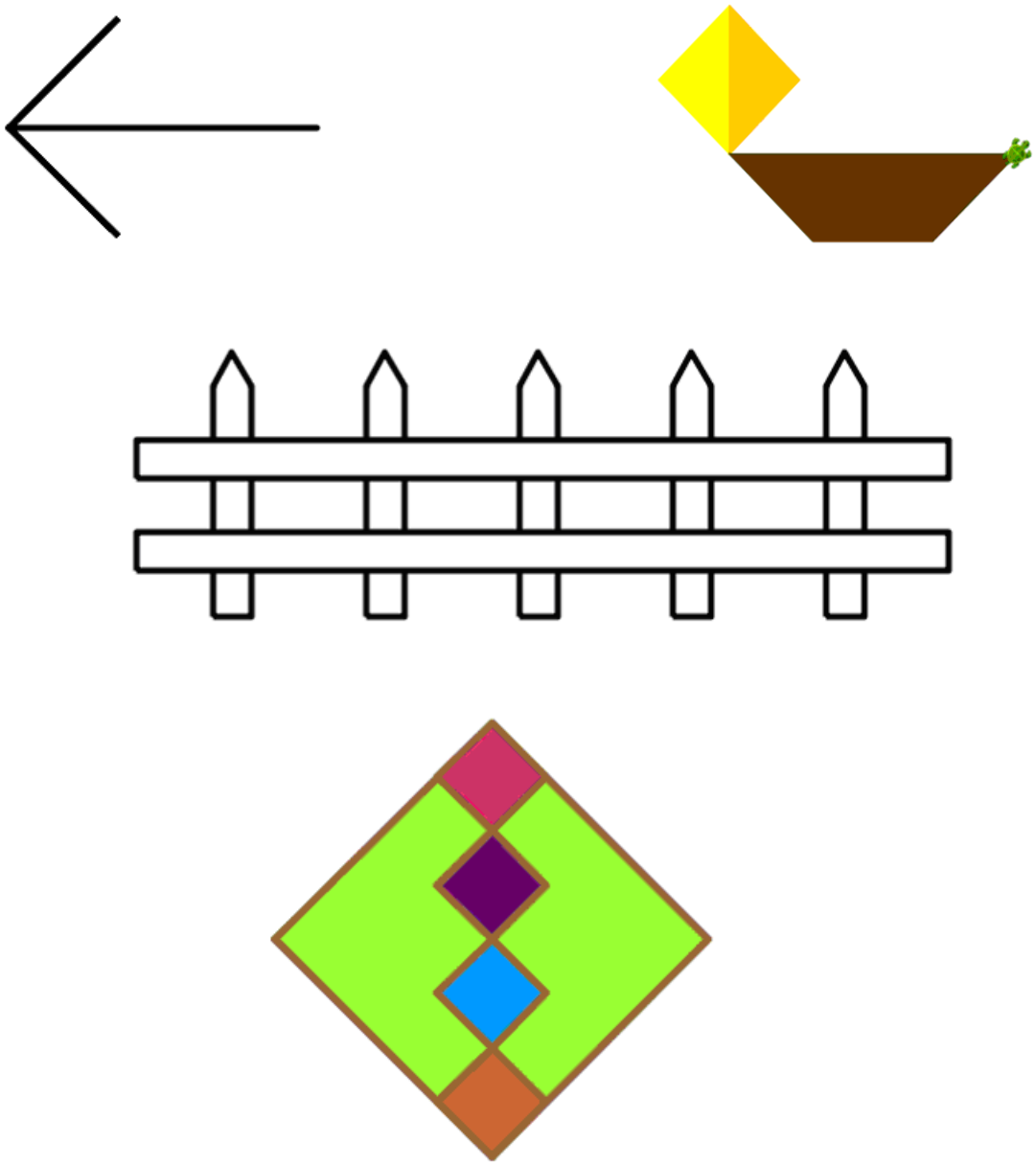
Interlude - Recap of Programming Concepts

By this point, you have learnt a lot of the core ideas in programming. Let's do a quick recap.

- A Program contains a series of instructions. These instructions can be either:
    - Commands – which take some action or have some effect.
    - Expressions – which compute a result.
    - Keywords instructions – to help structure the program.
- Expressions (mostly) make use of operators to compute results.
- Expressions are (mostly) used within commands and keyword instructions.
- Commands and operators are *called* with *inputs* of the appropriate type to get them to do their work.
- Operators *return results* to their caller. Commands don't return anything.
- The text of a program is called source code.
- A running program is called a process.
- You can use the repeat() command to remove repetition in a program. The repeat command also changes the flow of a process.
- Named values are an important part of a program. They make the program:
    - Easier to read
    - Configurable
- User defined commands allow you to capture commonly used patterns of code, give them names, and then reuse them. This reduces code duplication, and makes your programs easier to understand.
- You can make choices within your program with the help of the if-else instruction – to select certain portions of your program to run only when certain conditions are true.
- Comments help you to make your program easier to understand.
- When you are drawing a non-trivial sketch, your program should be structured in such a manner that it has sections that actually draw things on the screen **(drawing code)**, and sections that just move the turtle and position it for subsequent drawing **(positioning code)**. These sections should be clearly separated by comments. The positioning code should start with the `penUp()` command and end with the `penDown()` command.

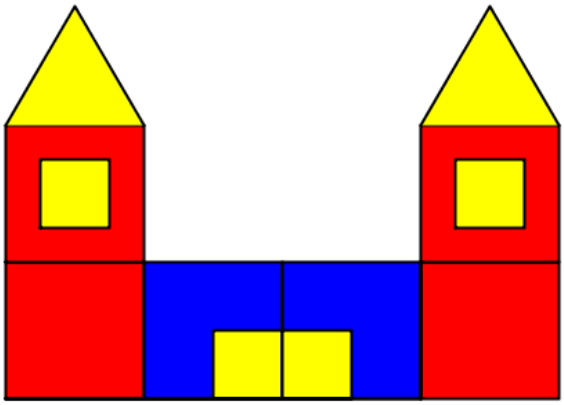
Activity 15 - Practice Session III

Write programs to make the following figures:



Activity 16 – Mini Project #2

Write a program to make the following castle:



Activity 17 – Circles and Curves

Till now, all your sketches have been made out of straight lines. It's now time for you to start making curved lines and circles.

Type in the following code and run it:

```
clear()
repeat(360) {
    forward(1)
    right(1)
}
```

Amazingly enough, the above code gets the Turtle to draw a circle (even though it seems to just be drawing straight lines via the forward command!).

Why do you think that happens?

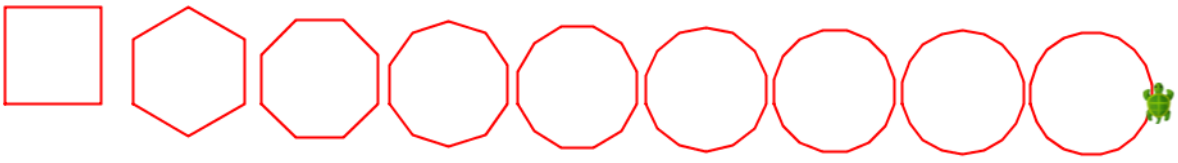
You'll learn more about this in the next book in the series. But the basic idea here is that a regular polygon, with a given perimeter, starts to look more and more like a circle - as the number of sides of the polygon increases.

A polygon is a closed figure made out of straight line segments. These line segments are called the sides of the polygon.

A regular polygon is a polygon with equal sides.

The perimeter of a polygon is the sum of the lengths of its sides.

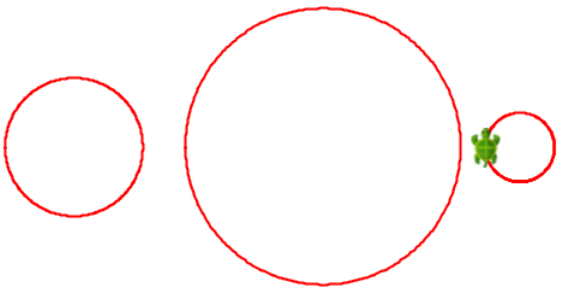
Take a look at the following figure to see this idea in action. The figure shows polygons (from left to right) with 4, 6, 8, 10, 12, 14, 16, 18, and 20 sides. The polygon on the extreme right, with 20 sides, looks almost like a circle, doesn't it?



In other words, we can use very small straight lines to approximate the appearance of curves and circles!

Next, let us look at code to make circles of different sizes:

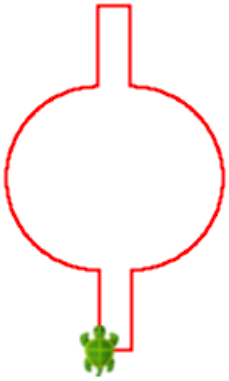
```
clear()
setAnimationDelay(100)

// normal sized circle
repeat(360) {
    forward(1)
    right(1)
}

penUp()
right()
forward(150)
left()
penDown()

// Bigger circle.
// Note that we take bigger steps forward to make a bigger circle
repeat(360) {
    forward(2)
    right(1)
}

penUp()
right()
forward(250)
left()
penDown()

// Smaller circle.
// Note that we take smaller steps forward to make a smaller circle
repeat(360) {
    forward(0.5)
    right(1)
}
```



Make sure you understand the code above before moving on...

Exercises

1. Write a program to make the following figure:



2. Now, make minimal modifications to your program to get it to make the following figure:

Can you do this by changing just two lines in your original program? By changing just *one* line?

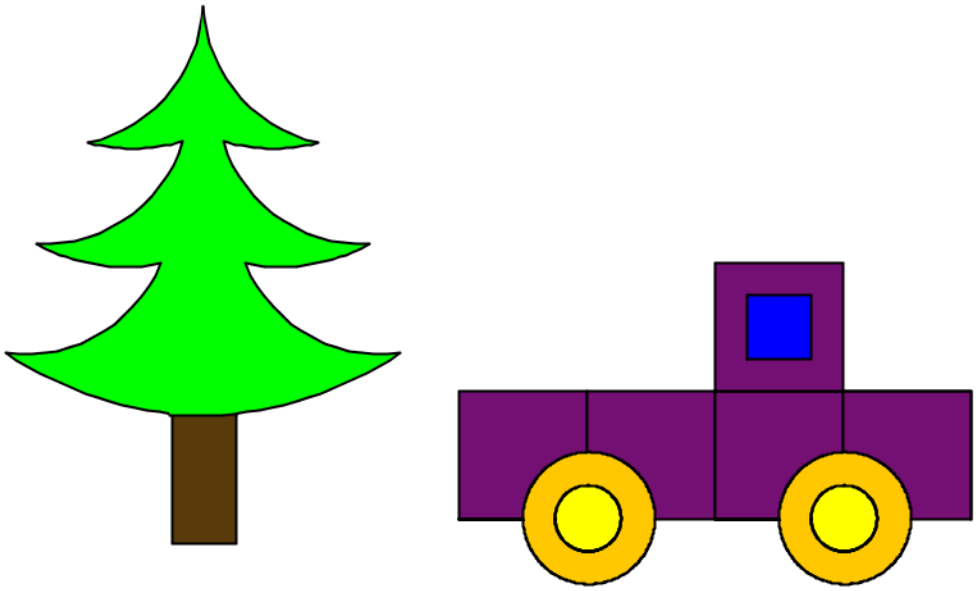
Activity Checklist

It's good if you know the answers to the following questions before moving on:

1. What's a regular polygon?
2. How can you make a circle from straight lines?

Activity 18 – Mini Project #3

Write a program to make the following figure:



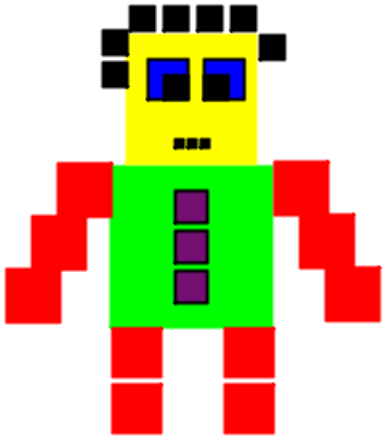
Hints:

- The tree is symmetrical about a vertical line drawn through its center.
- You can change the *curvature* of curves by changing the relative magnitudes of the inputs to the `forward` and `turn` commands (you saw this towards the end of Activity 17). In the two repeat loops below, the first makes a *flatter* curve, while the second one makes a *more curved* curve.

```
repeat(50) {
   forward(2)
   right(1)
}
repeat(50) {
   forward(0.5)
   right(1)
}
```

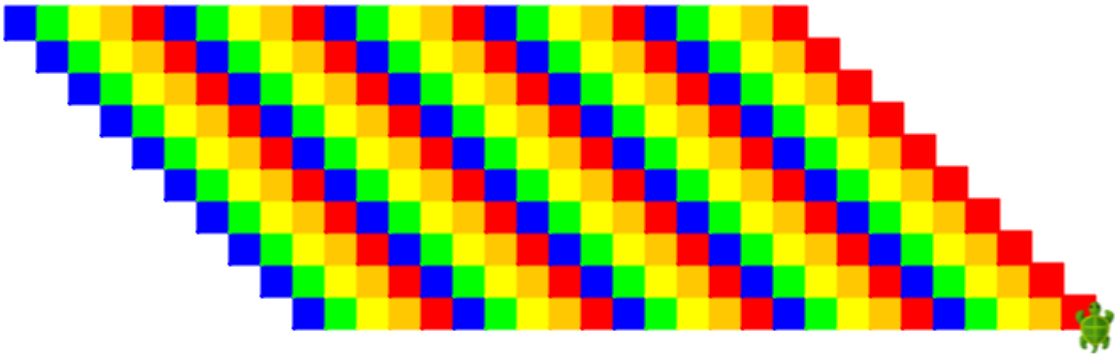
Activity 19 – Mini Project #4

Write a program to make the following figure:



Activity 20 – Mini Project #5

Write a program to make the following figure:



Hints:

- Identify patterns, and create user defined commands to make them
- Make use of repeat loops to:
  - Make your own commands.
  - Use these commands to make the figure.

Activity 21 – Break Free #2

This one is up to you. Use the skills that you have acquired while reading and working with this book – to come up with a creation of your own.

And then share your creation with us on the Kojo *Code Exchange*. We look forward to seeing you there...

Solutions to Exercises

Activity 1

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

right()

forward(200)
right()
forward(100)
right()
forward(100)
right()
forward(100)
right()
```

Activity 2

Without using the `left` command.

```
clear()
repeat (3) {
    repeat (4) {
        forward(100)
        right()
    }
    forward(100)
    right()
    forward(100)
    repeat (3) {
        right()
    }
}
```

Using the `left` command.

```
clear()
repeat (3) {
    repeat (4) {
        forward(100)
        right()
    }
    forward(100)
    right()
    forward(100)
    left()
}
```

Activity 4

```
val size = 200
clear()
repeat (4) {
    forward(size)
    right()
}
repeat (4) {
    forward(size / 2)
    right()
}
repeat (4) {
    forward(size / 4)
    right()
}
repeat (4) {
    forward(size / 8)
    right()
}
```

For the second run, the first line of the program becomes:

```
val size = 150
```

Activity 6

```
clear()
setAnimationDelay(100)

// make the big green/yellow square
setPenColor(green)
setPenThickness(4)
setFillColor(purple)
repeat (4) {
    forward(200)
    right()
}

setPenColor(blue)
setPenThickness(2)
setFillColor(orange)
// make the four small blue/red squares
repeat (3) {
    // make a single small square
    repeat (4) {
        forward(100)
        right()
    }
    // position the turtle for the next square
    penUp
    forward(50)
    right()
    forward(50)
    left()
    penDown
}
```

Activity 7

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

clear()
repeat (3) {
    square()
    forward(100)
    right()
    forward(100)
    left()
}
```

Activity 8

```
val size = 100
val space = 20

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

clear()
setAnimationDelay(100)

// first square
square(size)

// position turtle for next square
penUp
left()
forward(size + space)
right()
penDown

// second square
square(size)

// position turtle for next square
penUp
left()
left()
forward(size + space)
right()
right()
penDown

// third square
square(size)

// position turtle for next square
```

```
penUp
right()
forward(size + space)
left()
penDown

// fourth square
square(size)
```

Activity 10

```
// Notice how we create named values for all sizes of interest
val tWidth = 50 // tower width
val tHeight = 200 // tower height
val sz1 = 20 // size of small square
val sz2 = 30 // size of medium square
val dSz = 50 // size of door
val wSz = 150 // size of red wall

// make a rectangle with the given size and background color
def rect(l: Int, w: Int, s: Color) {
    setFillColor(s)
    repeat(2) {
        forward(l)
        right()
        forward(w)
        right()
    }
}

// make a square with the given size and background color
def square(n:Int, c:Color) {
    rect(n, n, c)
}

// make a tower
def tower() {
    // make the basic tower
    rect(tHeight, tWidth, brown)
    // go to position for first square on top
    penUp()
    forward(tHeight)
    left()
    forward(sz1/2)
    right()
    penDown()

    // first square
    square(sz1, yellow)
```

```
    // go to position for second square on top
    penUp()
    right()
    forward(tWidth)
    left()
    penDown()
    square(sz1, yellow)

    // go back to where we started making the rectangle
    // its important to follow this convention to allow the
    // tower command to be used reliably
    penUp()
    setFillColor(brown)
    left()
    forward(tWidth - sz1/2)
    left()
    forward(tHeight)
    right()
    right()
    penDown()
}

// start drawing
clear()
setAnimationDelay(100)
setPenColor(black)

// make left tower
tower()

// go to bottom left of red wall
penUp()
right()
forward(tWidth)
left()
penDown()

// make wall
square(wSz, red)

// go to position to make yellow squares on top of wall
penUp()
forward(wSz)
```

```
penDown()

// make first yellow square
square(sz2, yellow)

repeat (3) {
    // go to position for making next square
    penUp()
    right()
    forward(sz2)
    forward((wSz - 4*sz2)/3)
    left()
    penDown()

    // make next square
    square(sz2, yellow)
}

// go to position for making door
penUp()
right()
forward(sz2)
right()
forward(wSz)
right()
forward(wSz - (wSz-dSz)/2)
right()
penDown()

// make door
square(dSz, yellow)

// go to position for making second tower
penUp()
right()
forward(wSz - (wSz-dSz)/2)
left()
penDown()

// make second tower
tower()
invisible()
```

Activity 16

```
def square(n:Int , c:Color) {
    setFillColor(c)
    repeat(4){
        forward(n)
        right()
    }
}

def triangle(a:Int, c:Color) {
    setFillColor(c)
    left()
    repeat(3){
        right(120)
        forward(a)
    }
}

def tower() {
    // lower square
    square(100, red)

    forward(100)

    // upper square
    square(100, red)

    penUp()
    right()
    forward(25)
    left()
    forward(25)
    penDown()

    // window
    square(50,yellow)

    penUp()
    left()
    forward(25)
```

```
    right()
    forward(75)
    penDown()

    // top triangle
    triangle(100, yellow)
}

clear()
setAnimationDelay(100)
setPenColor(black)
//left tower
tower()

// adjust position to make first blue square
penUp()
back(100)
left()
forward(200)
left(180)
penDown()

// first blue square
square(100,blue)

// adjust position to make first doorway
penUp()
right()
forward(50)
left()
penDown()

// first doorway
square(50, yellow)

// adjust position to make second blue square
penUp()
right()
forward(50)
left()
penDown()

// second blue square
```

```
square(100, blue)

// second doorway
square(50, yellow)

// adjust position to make second tower
penUp()
right()
forward(100)
left()
penDown()

// second tower
tower()
invisible()
```

Activity 17

```
clear()

repeat (2) {
    forward(50)
    left()
    repeat (180) {
        forward(1)
        right(1)
    }
    left()
    forward(50)
    right()
    forward(20)
    right()
}
```