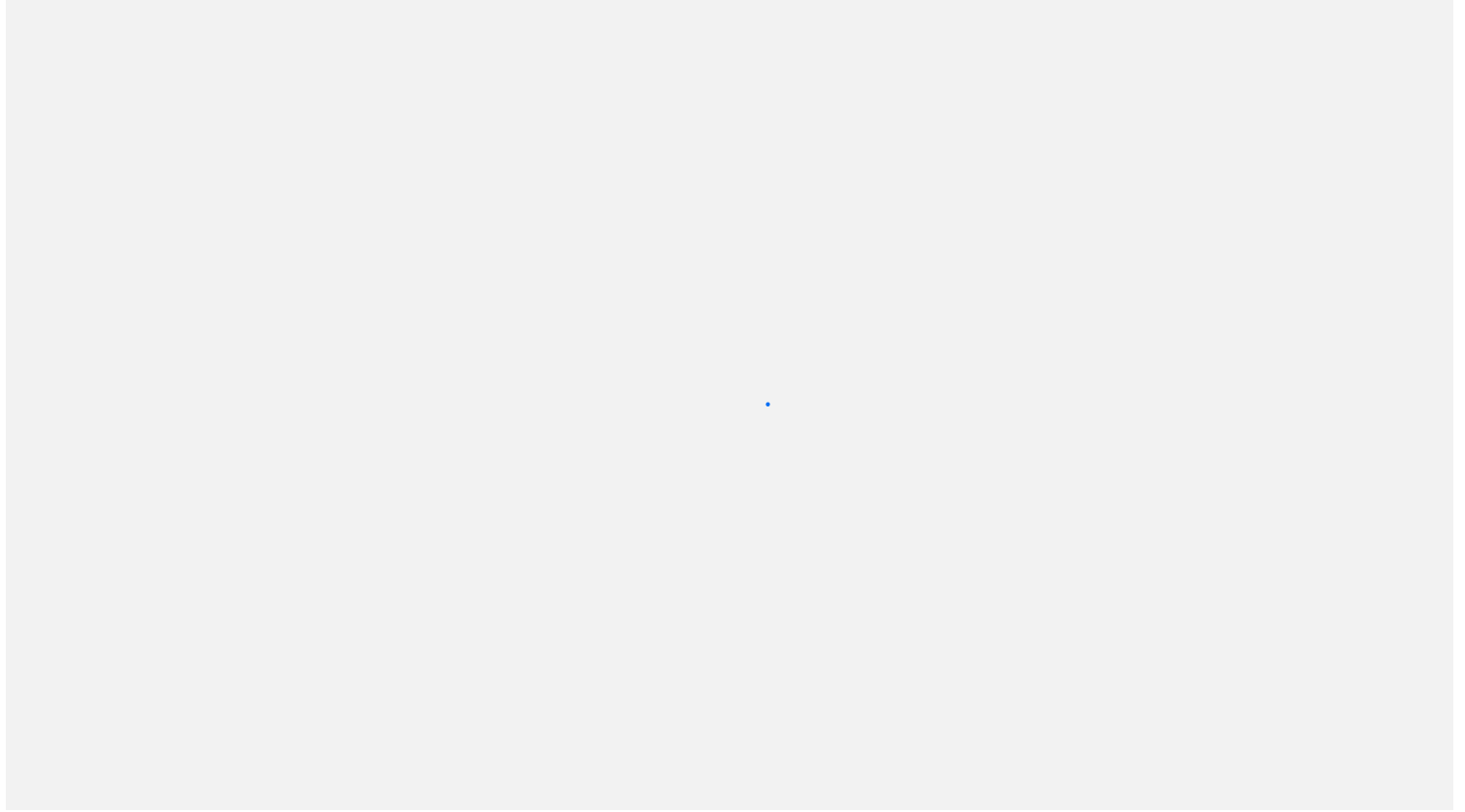


# Week 3 - Topic 4: Selection and Iteration

---

## Academic Integrity



---

# Flow Control

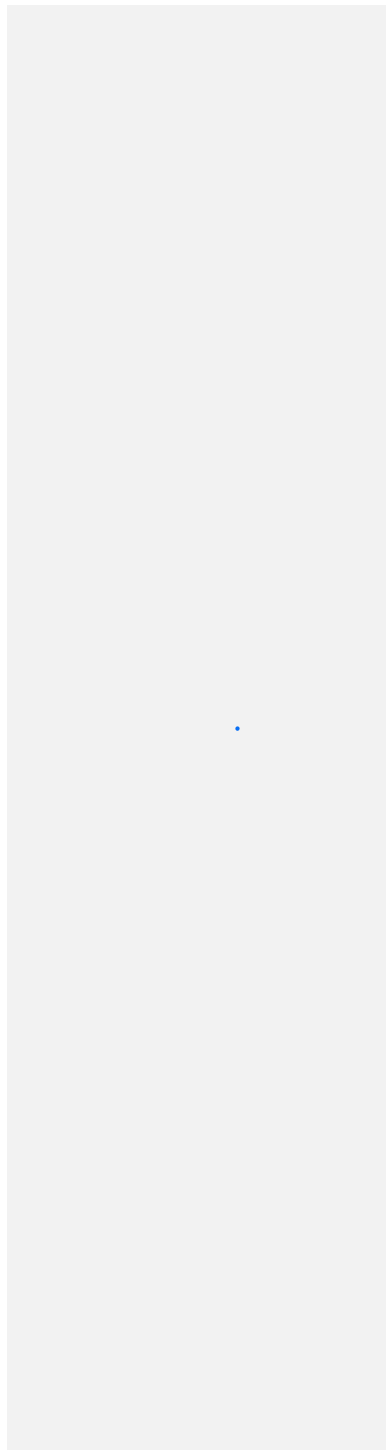
This lecture we look at flow control. This makes up two of the three fundamental principles/elements of structured programming (coding aspects):

- Sequence (we looked at in previous weeks)
- Selection
- Iteration/repetition/looping

---

## What we have done so far

So far our code has been based on sequences of statements:



▶ Run

RUBY



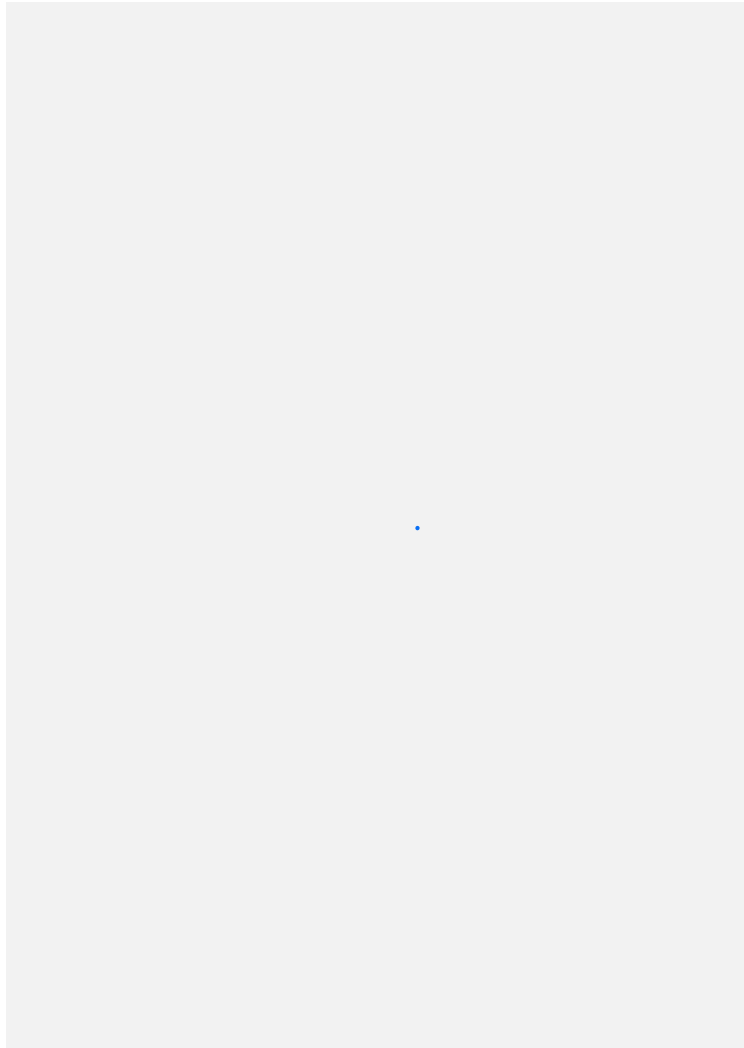
```
1 def main()  
2   puts("Please enter your name: ")  
3   name = gets.chomp()  
4   puts("#{name} is a silly name")  
5 end  
6  
7 main()
```



---

## Making code more dynamic

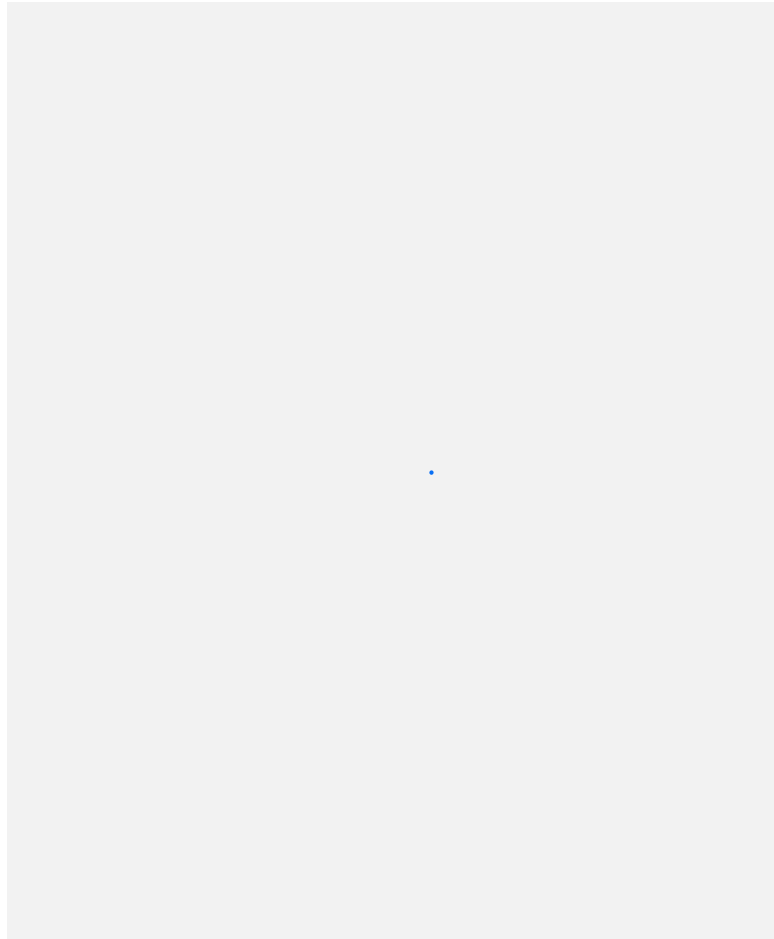
Now we are going to look at how to make our code more dynamic:



---

## Design for understanding

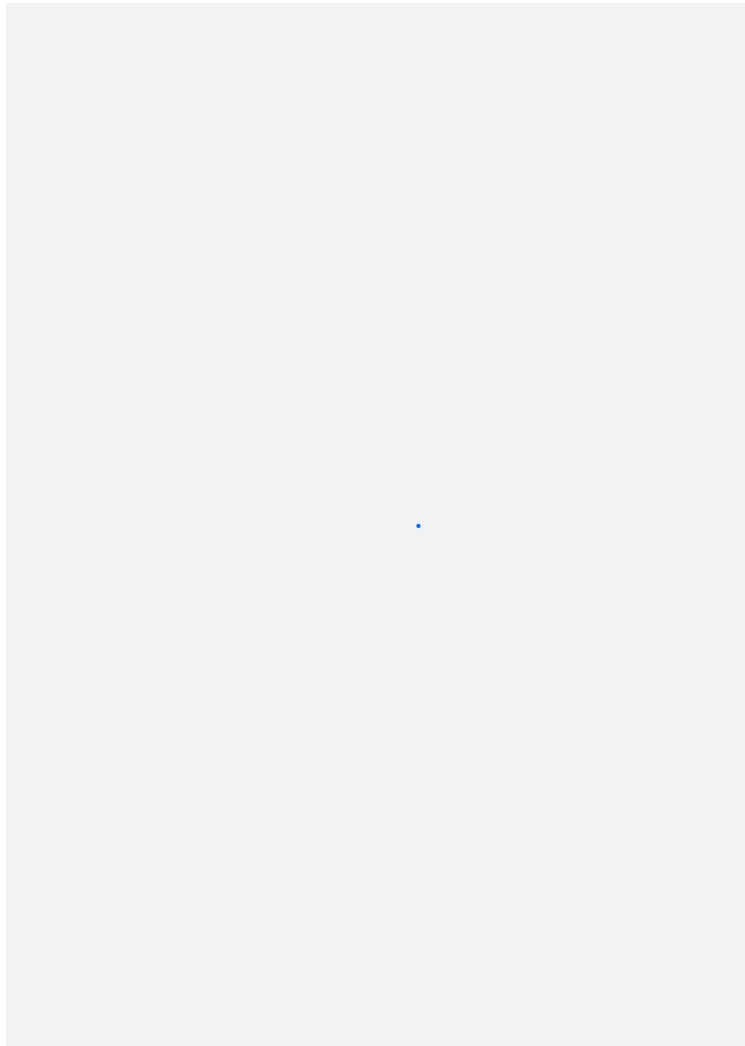
However, we need to carefully design our code so as to avoid creating 'spaghetti code':



---

## Single entry/exit

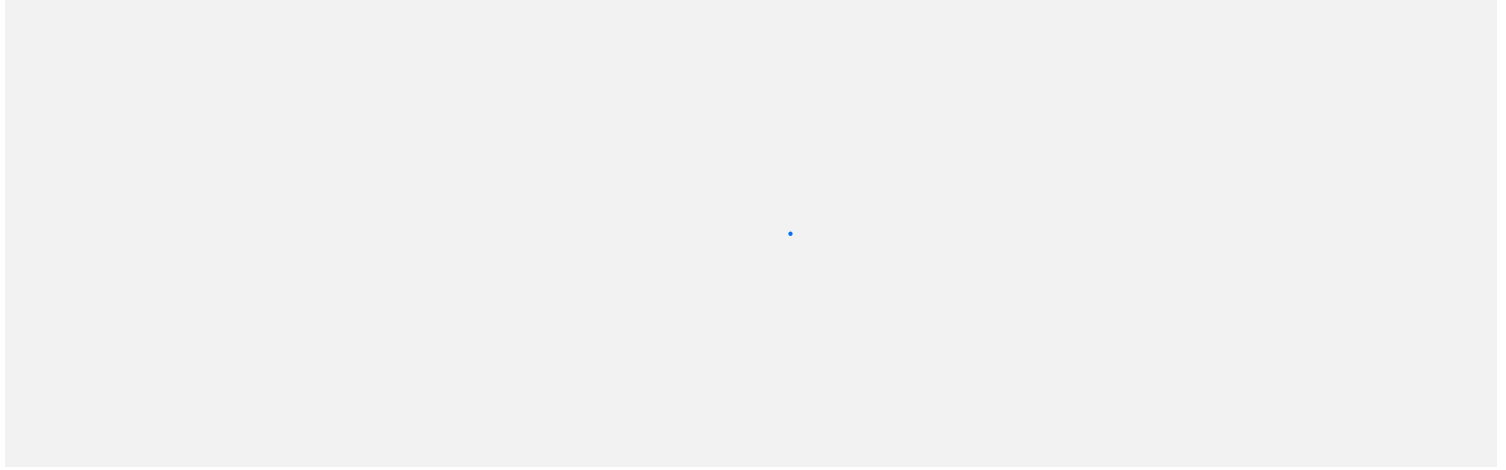
We try and use blocks of code with one entry and one exit point:



---

## Selection/Branching

Blocks can include decisions, or branches, where one of a variety of paths is taken: this is called selection :





---

## Iteration/Looping/Repetition

The other kind of block is a loop, where instructions are repeated a number of times: this is called repetition - in these cases the code keeps looping around until a test condition is false:



.

---

# Sequence, Selection, Repetition

Sequence, Selection, and Repetition: the key components of Structured Programming:

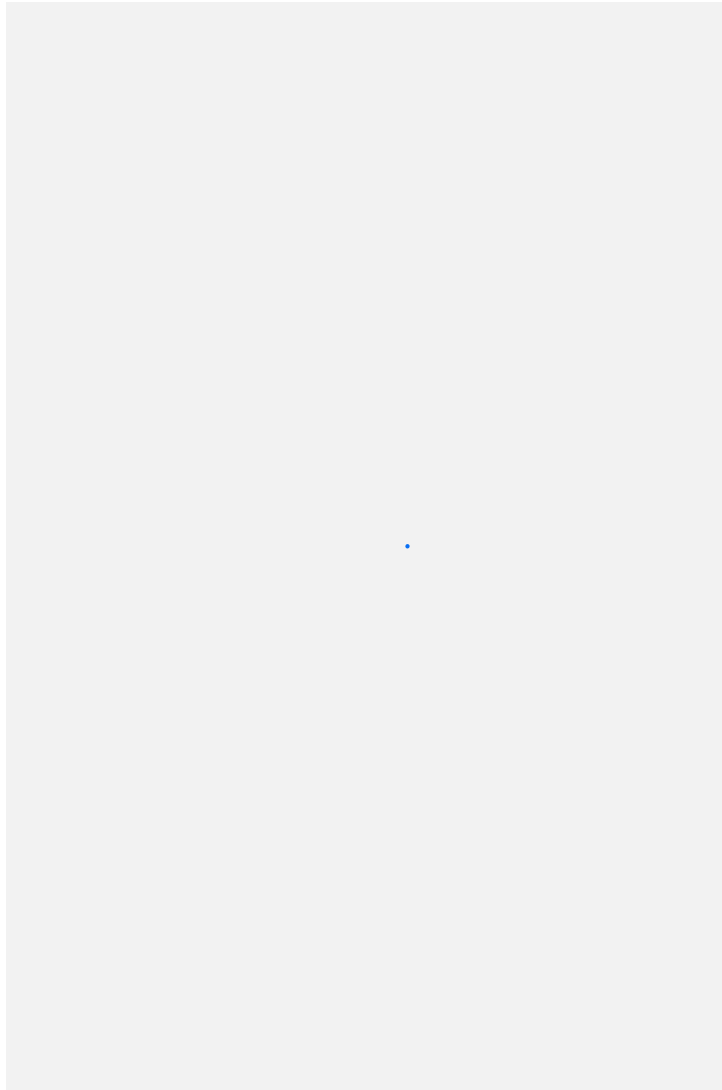


---

## Flow for a Simple Program

Below is a control flow diagram for the Silly Name program

This branching is represented using an **if** statement:



# Simple IF statements

Below is the simplest form of an *if* statement:

▶ Run

RUBY

```
1 count = 5
2
3 if (count == 5)
4   puts 'It matched!'
5 end
```

Notice the condition (double equals checks if two values are the same):

(count == 5)

and the consequent:

puts 'It matched!'

What happens in the above code if *count* is not equal to 5? (try it)

## Common IF statements

▶ Run

RUBY



```
1 # TRY CHANGING THE VALUE OF count BELOW then RUN:
2 count = 5
3
4 if (count == 5) # Condition
5   puts('It matched!') # Consequent
6 else
7   puts('No match!') # Alternative
8 end
```

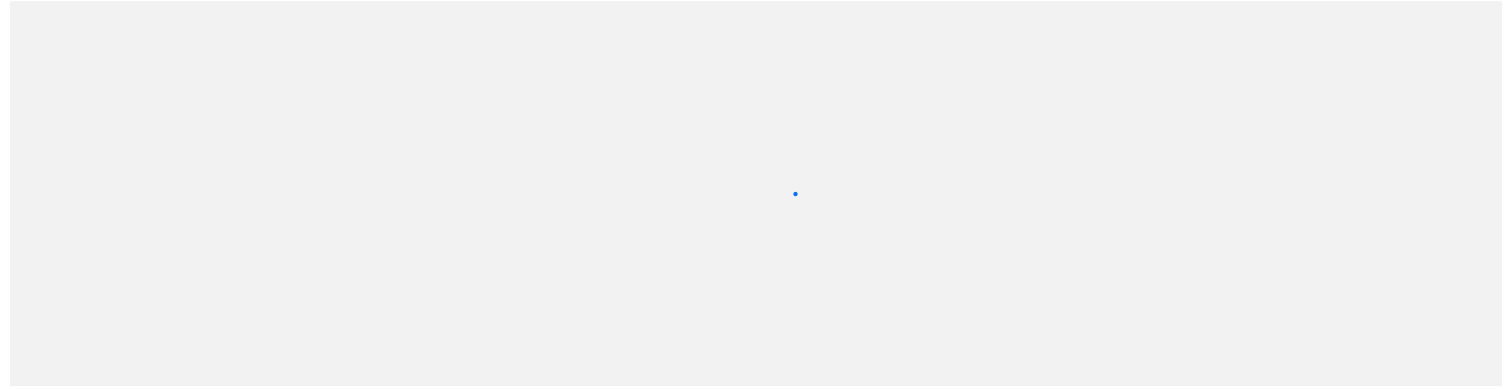


---

# Conditional Comparisons

Conditional comparisons evaluate to be either true or false (i.e a Boolean value).

Different languages have different operators:



For Ruby's operators see :

See: [https://www.tutorialspoint.com/ruby/ruby\\_operators.htm](https://www.tutorialspoint.com/ruby/ruby_operators.htm)

# Boolean conditions

Lets try some examples:

Single variable:

```
(value > 10) or (value < 0)
```

Multiple variables:

```
(age == 25) and (value != 5)
```

Try these out below:

▶ Run

RUBY

```
1 value = 10
2
3 if (value > 10) or (value < 0)
4   puts("The condition is true")
5 else
6   puts("The condition is false")
7 end
```

▶ Run

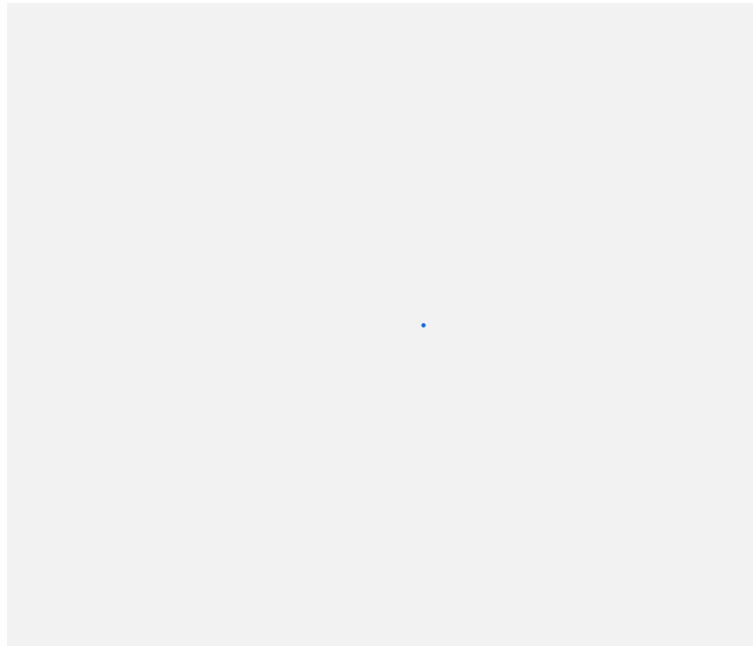
RUBY

```
1 age = 25
2 value = 5
3
4 if (age == 25) or (value != 5)
5   puts("The condition is true")
6 else
7   puts("The condition is false")
8 end
9
10
```



# Boolean Truth tables

The following tables show when a condition is true based on the values of two variables (**A** and **B**):



Source: [https://users.cs.jmu.edu/bernstdh/web/common/lectures/summary\\_logical-operators.php](https://users.cs.jmu.edu/bernstdh/web/common/lectures/summary_logical-operators.php)

▶ Run

RUBY

```
1 # TRY CHANGING THE VALUES OF a AND b BELOW:
2
3 a = true
4 b = false
5
6 if (a and b)
7   puts("AND: A and B are both True")
8 end
9
10 if (a or b)
11   puts("OR: Either one of both of A or B is true")
12 end
13
14 if (a ^ b) # ^ is the XOR operator in Ruby
```

---

## IF statements - variations

Remember the different types we saw:



.

# IF-ELSE

Here are some IF-ELSE variations:

▶ Run

RUBY



```
1 count = 5
2
3 if (count == 5)
4   puts('It matched!')
5 else
6   puts('No match!')
7 end
8
9 count = 3
10
11 if (count < 3)
12   puts('Less than 3!')
13 elsif (count == 3)
14   puts('Equal to 3!')
```

---

## Write an IF statement

Write code that reads in a name, if the name is equal to "Jill" print "Awesome Name" otherwise print "That is a silly name".

# UNLESS Statements

A variation on IF (available in Ruby but not C):

▶ Run

RUBY



```
1 count = 5
2 unless (count == 5)
3   puts('Not 5!')
4 else
5   puts('It is equal to 5!')
6 end
7
8 puts('It is 5!') unless count != 5
9 puts('It is 5') if count == 5
10
```



## CASE statements

Another type of conditional test is the *case* statement - this checks multiple possible conditions (in C the equivalent is a *switch* statement) .

**Note:** there are some problems with the code below - we need to fix them!

▶ Run

RUBY



```
1 count = 10
2 case count
3 when 1..4
4   puts('Not 5!')
5 when 5
6   puts('It is 5!')
7 when 5..10
8   puts('It is between 5 and 10')
9 else
10  puts('It is less than zero or more than 10')
11 end
12
```



## Case Statements – For Types

We can also have conditions based not on the **value** of a variable, but on its **type**:

▶ Run

RUBY



```
1 something = 10
2 case something
3 when Numeric
4   puts("I'm a number.  My absolute value is #{something.abs}")
5 when Array
6   puts("I'm an array.  My length is #{something.length}")
7 when String
8   puts("I'm a string.  In lowercase I am: #{something.downcase}")
9 else
10  puts("I'm a #{something.class}")
11 end
12
```



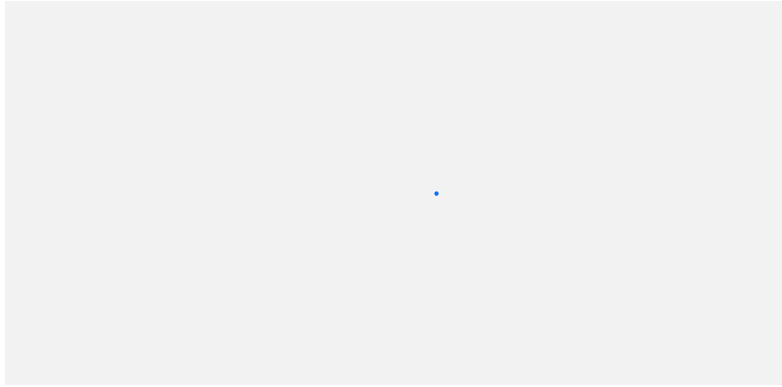
Acknowledgement: [http://rosettacode.org/wiki/Conditional\\_structures/Ruby](http://rosettacode.org/wiki/Conditional_structures/Ruby)

---

# Iteration (loops)

There are two basic types of loops:

- Pre-test
- Post-test



We will see various forms of both.

Loops often require **control** variables – we will see these as we go.



---

## When to use a post-test loop?

If you want the code in the body of the loop to execute at least once (i.e regardless of any conditional test), you use a post-test loop.

True

False

---

## Examples for when to use loop types

Which of the following situations are likely to require a post-test loop?

Reading from a file

Reading from a network connection

Reading a user's selection from a set of menu options

Incrementing a counter from zero to a number ranging from zero to 10 (as entered by a user)

# Pre-Test Loop - WHILE

Following is a while loop - which is a *pre-test* loop.

This is the most common loop structure.

Notice the use of *count* as a **control variable** to determine when to stop the loop:

 Run	RUBY	
<pre>1 count = 5 2 while (count &lt; 5) 3   puts('Count is ' + count.to_s()) 4   count = count + 1 5 end 6</pre> 		

---

## Pre-Test Loops – FOR

Notice the for loop auto-increments the counter

(i.e no need for `count = count + 1`):

Many languages (including C) have a type of FOR loop structure:

▶ Run

RUBY



```
1 count = 5
2 for count in 0..4
3   puts('Count is ' + count.to_s())
4 end
```



---

## Write a WHILE loop

Write a **while** loop that counts to 6 producing output as follows:

```
[user@sahara ~]$ ruby loop.rb
1
2
3
4
5
6
[user@sahara ~]$ █
```

## Post Test Loops – UNTIL

One option:

▶ Run

RUBY

```
1 count = 0
2
3 begin
4   puts('my line ' + count.to_s())
5   count = count + 1
6 end until (count == 6)
```

Or Equivalently:

▶ Run

RUBY

```
1 count = 0
2
3 begin
4   puts('my line ' + count.to_s())
5   count = count + 1
6 end while (count < 6)
7
```

Note: see: <http://rosettacode.org/wiki/Loops/Do-while#Ruby>

# BREAK statements

It is possible to use a *break* statement to 'jump' out of a loop or *if* statement:

Two variations are below - but these are not recommended in Ruby (often used in C):

▶ Run

RUBY

```
1 count = 0
2
3 loop do
4   puts('my line ' + count.to_s())
5   count += 1
6   break if (count == 2)
7 end
8
9 count = 0
10
11 loop do
12   puts('my line ' + count.to_s())
13   count += 1
14   break unless (count != 2)
```

# TIMES loop

We can execute a block of code a certain number of times:

▶ Run

RUBY



```
1 5.times do
2   puts('Enter a name: ')
3   name = gets.chomp
4   puts('Name is ' + name)
5 end
6
```





# REDO

Another option for post-test loops (in Ruby) is REDO:

▶ Run

RUBY

```
1 1.times do
2   puts('Enter a name: ')
3   info = gets.chomp()
4   unless (info == "exit")
5     puts('You entered: ' + info)
6     redo
7   end
8 end
9
10 puts("second loop")
11
12 1.times do
13   puts('Enter a name: ')
14   info = gets.chomp()
15   if (info == "exit")
16     puts("exit")
17   end
18 end
```

# Variable TIMES Loop

Fred Instead of hard-coding the number of times to loop, we can use a variable:

▶ Run

RUBY



```
1 count = 5
2 count.times do
3   puts(count.to_s() + ' Enter a name: ')
4   name = gets.chomp()
5   puts('Name is ' + name)
6 end
7
```



## Accessing the TIMES variable

If we want to know where we are up to in terms of how many times we have been through the loop already we can access the otherwise hidden control variable we give the variable whatever name we want, in this case *i*:

▶ Run

RUBY



```
1 count = 5
2 count.times do |i|
3   puts(i.to_s() + ' Enter a name: ')
4   name = gets.chomp()
5   puts('Name is ' + name)
6 end
```



---

## UPTO loop

The *upto* loop also auto-increments the counter  
(i.e no need for `count = count + 1`):

▶ Run

RUBY



```
1 1.upto(5) do
2   name = gets.chomp()
3   print('Name is ' + name)
4 end
```



# NESTED loops

You can have loops inside loops (called **nested** loops):

▶ Run

RUBY



```
1 1.upto(5) do |i|
2   puts(i.to_s() + ' Enter runner\'s names in order of completion: ')
3   name = gets.chomp()
4   print('Name is ' + name)
5   points = 0
6   i.upto(3) do |j|
7     points += j
8   end
9   puts('. You get ' + points.to_s() + ' points')
10 end
```

---

## Tasks for this week

- Silly name task
- Submenu task
- Shape Drawing Task

---

## Loops and Conditions in structure charts

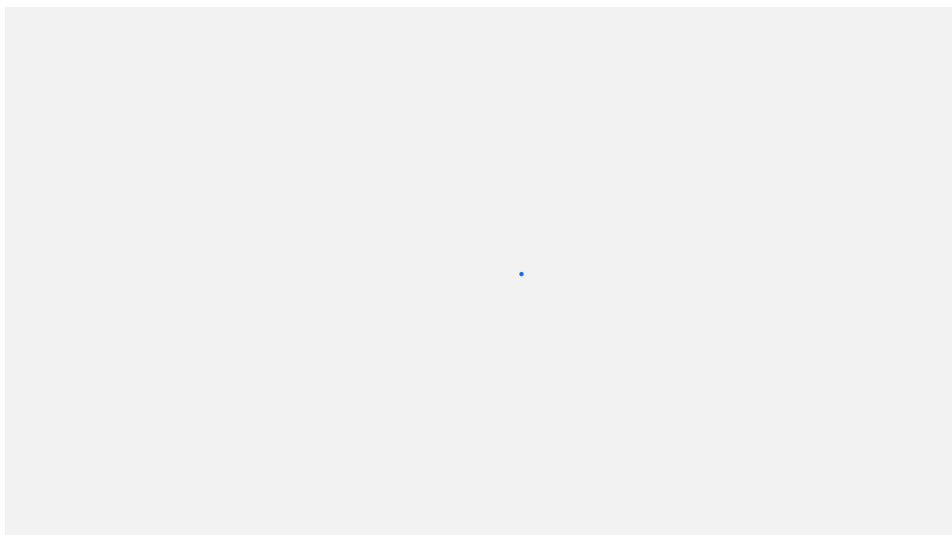
Representing Conditions and Loops in Structure Charts.

See the video for a structure chart for the Silly name Task:

<https://echo360.org.au/media/b54387d1-4df6-47a1-a55a-e52c74514a15/public>



Then the following shows the Simple Menu Program:



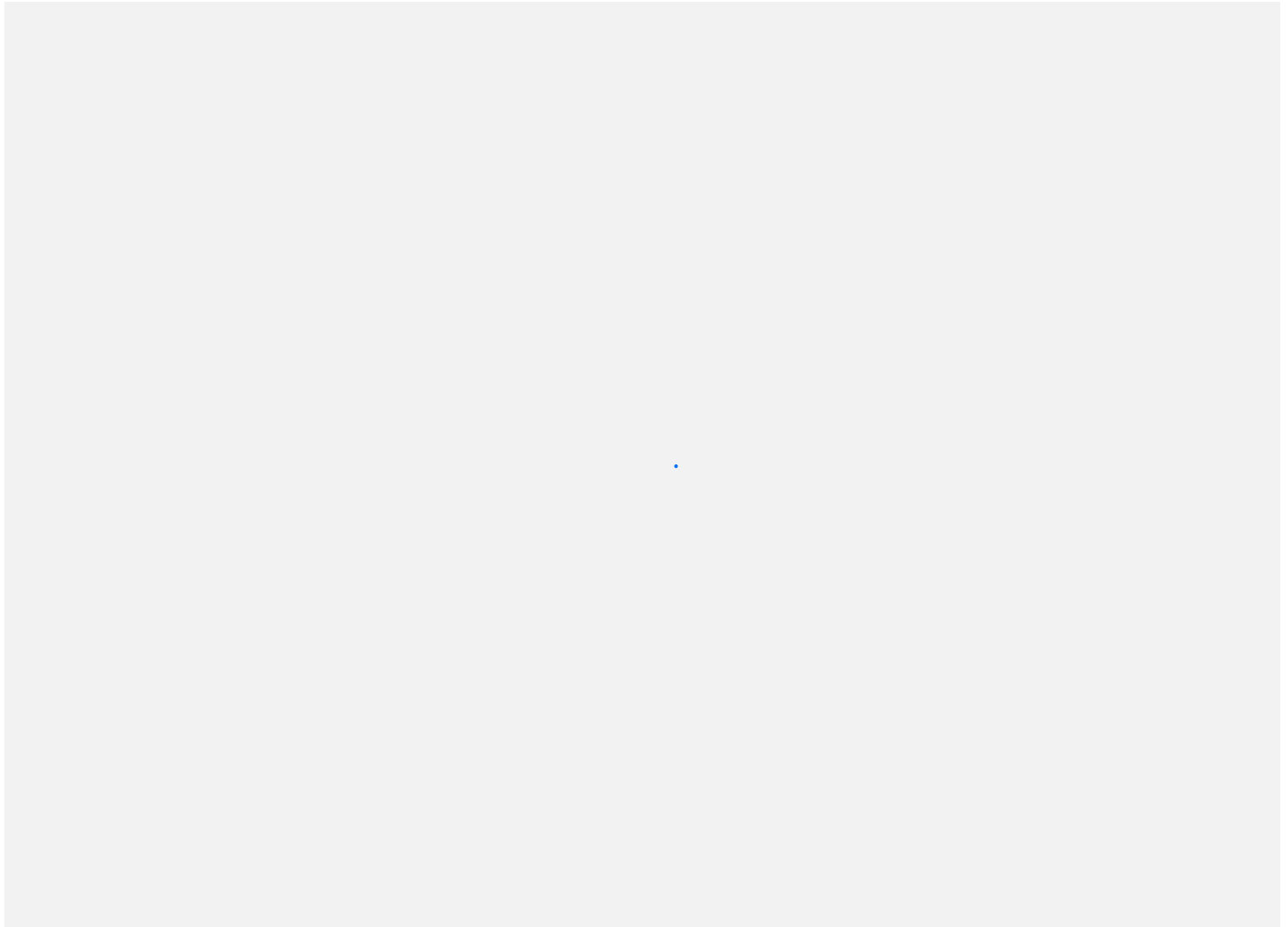




---

# Graphical Programming (shape drawing)

We use a co-ordinate system to place things on the screen:



---

# Graphical Programming: Rectangles

Drawing a rectangle:

```
Gosu.draw_rect(1, 1, 7, 3, Gosu::Color::BLUE, ZOrder::TOP, mode=:default)
```

A small blue dot is centered in the lower half of a large, light gray rectangular area. This dot represents the visual output of the Gosu code snippet above, which draws a rectangle at the top-left corner of the window.

# Shape Drawing using GOSU

**Follow the 3 steps below:**

RUBY

```
1 require 'rubygems'
2 require 'gosu'
3 require './circle'
4
5 # The screen has layers: Background, middle, top
6 module ZOrder
7   BACKGROUND, MIDDLE, TOP = *0..2
8 end
9
10 class DemoWindow < Gosu::Window
11   def initialize
12     super(640, 400, false)
13   end
14
```

Use the following site to select colours for the circle (which uses RGB values):

[https://www.rapidtables.com/web/color/RGB\\_Color.html](https://www.rapidtables.com/web/color/RGB_Color.html)

Or

Use the Gosu colour constants:

<https://www.rubydoc.info/github/gosu/gosu/master/Gosu/Color>

**1 Copy the code from the box above into your editor then save it as *gosu\_shapes.rb*.**



```
1 require "rubygems"
2 require "gosu"
3
4 class Circle
5   attr_reader :columns, :rows
6
7   def initialize(radius)
8     @columns = @rows = radius * 2
9
10    clear, solid = 0x00.chr, 0xff.chr
11
12    lower_half = (0...radius).map do |y|
13      x = Math.sqrt(radius ** 2 - y ** 2).round
14      right_half = "#{solid * x}#{clear * (radius - x)}"
```

**2 Copy the code above into your editor and save it as *circle.rb*.**

**3 Run *ruby gosu\_shapes.rb* It should look as follows: - then change it to make your own picture.**

Use this link to help you: <https://www.rubydoc.info/github/gosu/gosu/master/Gosu>

# Drawing Shapes using GOSU - Demonstration



**NOTE: To run Graphical programs in this tool (ED) - you need to use the Run button for the task (not the terminal).**

In this task you use Gosu to create a program that draws a picture.

GOSU is a development environment that makes it easy to create programs that use graphics, sounds, animations and other aspects relevant to creating small interactive games.

Follow these **3** steps:

**1.** Copy the code provided to your IDE (both `gosu_shapes.rb` and `circle.rb`) and use the demonstration shapes to create a picture of your own design. Your picture should include at least 3 different types of shapes (eg: a triangle, a rectangle and a circle)

Use the following site to select colours for the circle (which uses RGB values):

[https://www.rapidtables.com/web/color/RGB\\_Color.html](https://www.rapidtables.com/web/color/RGB_Color.html)

Or Use the Gosu colour constants:

<https://www.rubydoc.info/github/gosu/gosu/master/Gosu/Color>

eg: a Red circle with a radius of 50 pixels would be produced by the two statements:

```
img = Gosu::Image.new(Circle.new(50))
img.draw(200, 200, ZOrder::TOP, 0.5, 1.0, Gosu::Color::RED)
```

Or you could use the HEX values:

```
img.draw(300, 50, ZOrder::TOP, 1.0, 1.0, 0xff_ff0000)
```

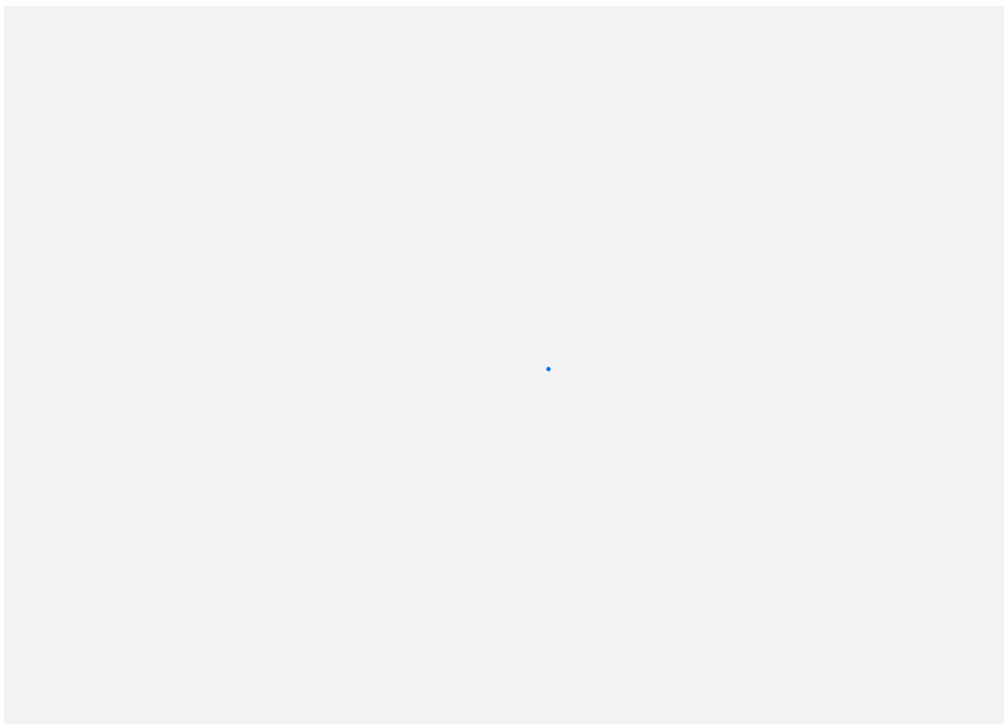
See here to work out HEX values:

<https://www.binaryhexconverter.com/decimal-to-hex-converter>

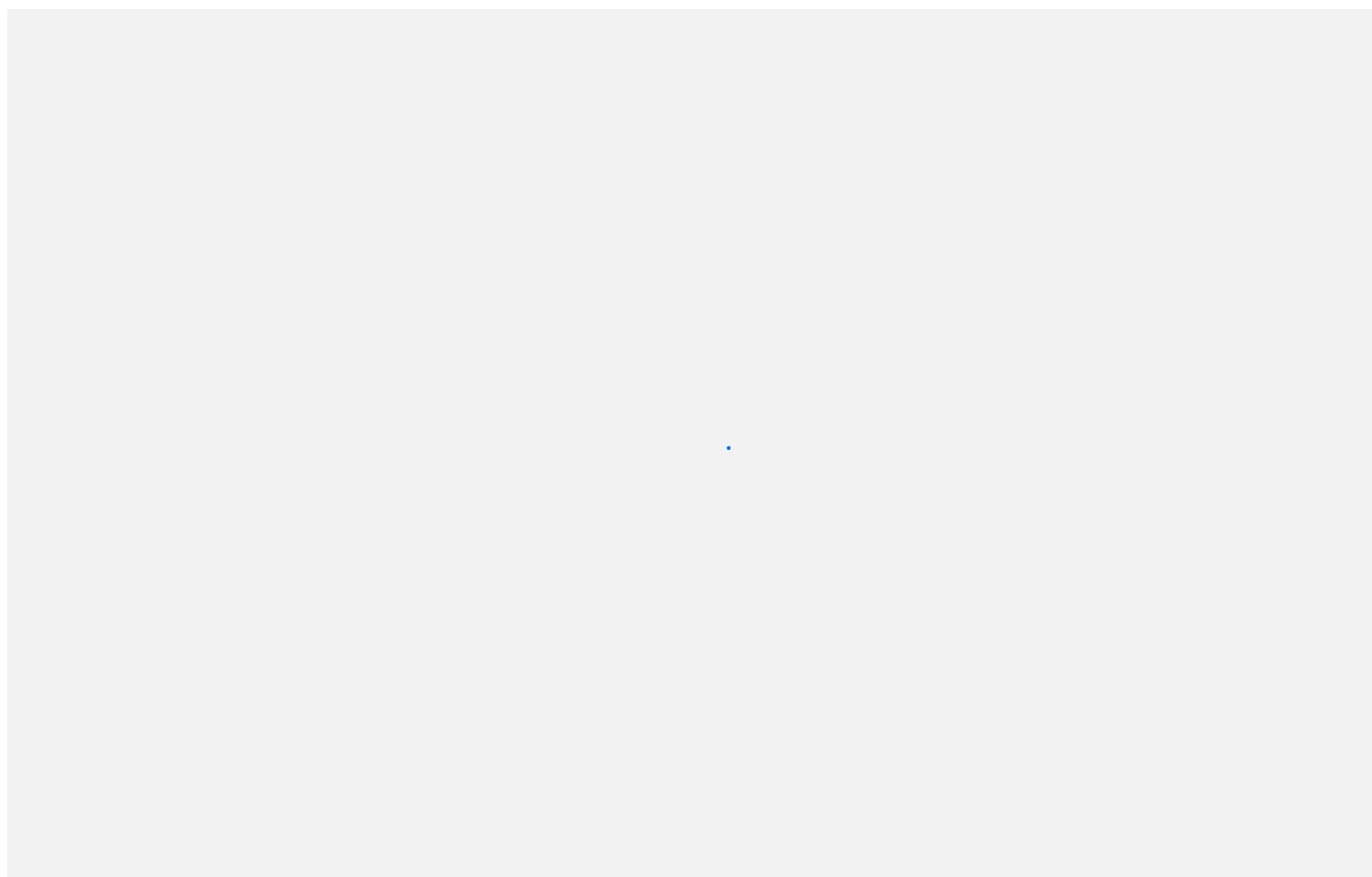
**2.** Run the code provided and study it to understand what is happening (you may need to run the command `gem install rubygems` to use the `circle.rb` code).

**3.** Using an IDE like Visual Studio Code, change the code to draw your own unique picture. You might want to draw it on paper first (perhaps graph paper or an electronic equivalent [like this](#)).

The co-ordinate system works as follows



Example:



Once your code is complete submit a copy and a screen shot to Doubtfire.

---

## Resources

Sobkowicz, M 2015 *Learn game programming with Ruby : bring your ideas to life with Gosu*, The Pragmatic Bookshelf (See chapter 7 for help the grid aspect of the Maze Task)

[Gosu Ruby Documentation](#)

[Gosu site](#)

[Gosu game video tutorial](#)