Week 2 - Topic 3: Functions and Structure Charts

About this Lesson

In this lesson we cover:

- Functions and procedures
- Signatures
- Libraries (gem files) – user defined, shared
- Variables and constants (local, global, other)
- Parameters and arguments
- Artifacts
- Structure charts, examples – parameter passing etc
- Resources
- Tasks for this week.

Procedures

A procedure is what you used last week - it groups together a block of code (in this case just one line) between the **`begin`** and the **`end`**. It also has a name - in this case **`main()`**, which is used to *call* the procedure.

Try and guess what the following procedure does before running it:

```ruby
def main()
    puts('This is a procedure called main!')
end

main()
```

Procedures and Side-effects

Procedures are called to produce *side effects*.

Procedures can call other procedures:

The output of the procedures below is a *side-effect*:

Run RUBY

```ruby
def subroutine()
    puts('This is a procedure called BY main!')
end

def main()
    puts('This is a procedure called main!')
    subroutine()
end

main()
```

Write a procedure

Change the code provided to add a procedure called `print_name()` that is called by main after `subroutine()` and outputs the name "Matt" to the terminal (hard code the name as a string literal).

For example with the name "Matt" the output should look as follows:

Passing Arguments

Procedures can take arguments. Arguments are data that is passed from one block of code to another. The argument is given a name (like a variable name) in the receiving procedure:

```ruby
def subroutine(arg)
    # 2. the argument 5 is received and given the variable name 'arg'
    puts('This is a procedure called BY main!')
    puts('It received the argument' + arg.to_s())
end

def main()
    puts('This is a procedure called main!')
    # 1. Pass the argument 5 to the procedure called 'subroutine'
    subroutine(5)
end

main()
```

Write a procedure that takes an argument

Write a procedure that takes an argument and prints it to the terminal.

Write the procedure in the code below.

Once you are happy with it, copy it to the screen on the right then click 'Mark" to see if it works correctly.

Run RUBY

```ruby
# put the procedure here:


def main()
    print_argument(12)
end

main()
```

Procedures can take multiple arguments

Run the code below then change it to add in the additional (String) argument "three" - print out the argument in the subroutine as follows:

"It received the argument three"

So the output should look as follows:

Arguments can be variables

Change the code so that the value "fred" is stored in a variable called `name` and then the variable `name` is used to pass the value "fred" to the called subroutine.

Procedures

Procedures are called for their side-effects

- True
- False

Functions

Functions return a value. Change the code below so that it returns 8 not 5.

Functions with arguments

Functions can take arguments - change the code provided so that argument passed is 10 not 7.

Functions

The following is true about functions:

- [ ] They can take parameters/arguments
- [ ] They can return values
- [ ] They can produce side-effects
- [ ] Any given function when called can return multiple values with different types

Signatures

Procedures and functions have a signature (a variation on this – which we see later - is called a prototype).

The signature identifies what formal parameters the function or procedure takes and what data type it returns (if any).

In the C programming language this would look as follows for a function:

```c
int read_integer_range(const char * prompt, int min, int max)
```

The Ruby equivalent is:

```ruby
def read_integer_in_range(prompt, min, max)
```

As Ruby is dynamically typed it does not require type information on parameters and return values (as C does). We return to this difference in later lectures. It also does not require prototypes.

Run RUBY

```ruby
def read_integer_in_range(prompt, min, max)
    puts(prompt + ' ' + min.to_s() + ' and ' + max.to_s())
    x = gets().to_i()
    while(x < min || x > max)
        puts(prompt + ' ' + min.to_s() + ' and ' + max.to_s())
        x = gets().to_i()
    end
    return x
end

read_integer_in_range("Enter a number between", 10, 20)
```

Signatures for Procedures

In C procedures and functions are distinguished by use of the void keyword:

In Ruby there is nothing explicit to distinguish functions from procedures. The use is determined by context (i.e whether it assigned to anything).

Thus in Ruby documenting code is important. There are tools to help with this.

Documenting Code

Use comments before functions and procedures to explain how to use your code, what arguments it takes and what it returns.

Try to avoid explaining how your code works (the reason for this is covered in a later lesson).
EG:

```ruby
# This function takes an Integer argument and returns an Integer of the
# value as the value received.

def function(arg)
    puts('This is a function that recieved arg: ' + arg.to_s)
    return(arg)
end
```

Now lets see the rdoc output for this (on my file system).

Documenting Code

To produce documentation run rdoc on the ruby file as follows:

rdoc function_doc.rb

then type:

cd doc

to change to the doc folder - then:

ls

to see the files and

less <filename>

to see the contents of any file.

Ruby Documentation

Core API:

https://ruby-doc.org/core-2.5.1/

Standard Library:
https://ruby-doc.org/stdlib-2.5.1/ - this is where the date library was that we used last week.

Operators:
https://www.tutorialspoint.com/ruby/ruby_operators.htm

Operator Precedence:
https://ruby-doc.org/core-2.5.1/doc/syntax/precedence_rdoc.html

Local Scoping

In *change(a, b)* the values of *a* and *b* are set to 10 and 5, but this change does not affect the values of *a* and *b* in *main* due to those being different variables with a different local context or **local** scope.

Effectively the *a* and *b* in *change()* are **a copy** of the *a* and *b* in *main.*

Run RUBY

```ruby
def change(a, b)
    a = 10
    b = 5
end

def main()
    a = 2
    b = 3
    puts('a = ' + a.to_s + ' b = ' + b.to_s())
    change(a, b)
    puts('a = ' + a.to_s + ' b = ' + b.to_s())
end

main()
```

Global Scoping

Here changing the values of $aandb$ in *change()* affect the output in main, as the same **global** variables are being used in both.

▶ Run RUBY

```ruby
# Global variables
$a
$b

# Effects of global scoping (Note : no parameters!)

def change()
    $a = 10
    $b = 5
end

def main()
    $a = 2
    $b = 3
```

Methods, attributes and classes

We are not teaching Object Oriented Programming (OOP), but you may come across some OOP terms during this course (in the resources, recommended texts, etc).

Here are some equivalencies:

- Methods: an OOP term that encompasses both functions and procedures.
- Attributes: includes variables, but also functions or methods that might be used to give values as though you were accessing a variable.
- Class/Object: In this unit we use classes as a way of grouping together either variables or methods.
- Classes/objects provide another scoping environment (actually more than one).

Some of the libraries/gems we use are written as OOP classes, but we do not need to know OOP beyond some minimal basics to use these.

Example of OOP variable scoping

You create an object using new(). Then variables declared with @ are accessible in all methods:

Run the code here to see how it works:

Run RUBY

```ruby
class Demo
    def first_procedure(arg)
        @shared_variable = arg
        second_procedure()
    end

    # note this procedure accesses shared_variable even though it is not
    # as a parameter/argument:
    def second_procedure()
        puts("The shared variable has the value: #{@shared_variable}")
    end
end

def main()
```

Variable Scoping

Which of the following are possible scopes within Ruby?

- ☐ Global (a variable or constant is accessible everywhere in the program)
- ☐ Local (a variable is only accessible in the block of code in which it is declared)
- ☐ Instance (a variable is accessible to all the procedures and functions - also called methods - with a class definition)

Artifacts I

Artifacts are anything man-made (i.e not occurring in nature by chance).
Eg (from the British Museum):

Artifacts arise from people imposing their will to shape and design raw materials into a desired form.

Artifacts II

- As a programmer you create artifacts.
- In doing so you are imposing your will to shape the raw material (in this case zeros and ones using core features of the language we are using) so as to implement your design (i.e your program).
- Just as skills (both technical and design) are required to make a useful and elegant Greek vase, so skills and design knowledge are required to make useful and elegant code.
- The sorts of artifacts we make as programmers include deciding what functions and procedures to create.
- We also need to determine what to call these artifacts. The names we give the things we create are called identifiers.

How to write your artifacts

- If you are going to spend a lot of time looking at code, it is better if it is nice code to look at!
- Thus programmers follow certain styles that make code easy to read.
- Distinguishes professional from amateur.

David, Psalm 39 (38), in Latin, Psalter, Paris ?, last quarter of 13th century

How to Write your Artifacts II

The inscription on the Praeneste Fibula. 7th Century BC.
The writing runs from right to left.
It says: “Manius made me for Numerius”
Source Wikipedia.

- The elements of good style in coding make the code readable for others.
  Eg: just as indicating end of words and sentences in writing helps reading.

How to Write your Artifacts III

Naming is a critical element in writing code – the names you choose are important.

They give meaning to what you are expressing in your code and with your artifacts.

Historically meaningful names were given to specific artifacts made by people. Eg: Swords:

- ”Excalibur”, - famous inscription on both sides.
- Durandal – the Sword of Roland (means “Strong Flame”, “enduring”)
- Joyeuse – The sword of Charlemagne (means Joyous)

But perhaps more importantly are the names given to types of artifacts (the type/token distinction which is important in programming)

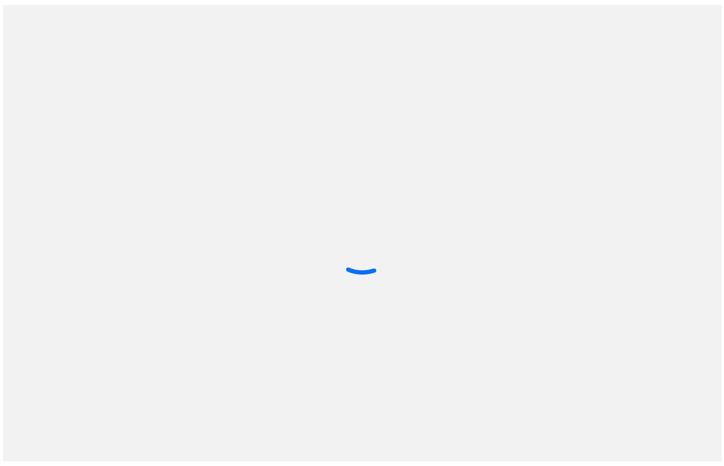
We will talk about appropriate naming as we move through the course.

Essentials of Procedures and Functions

- **Procedures** and **functions** are both ways of grouping lines of code (called ‘blocks’) and giving the group a name.
- Both functions and procedures can take **arguments**.
- Functions are different from procedures in that they have a **return value**.
- Instead of returning a value procedures produce **side-effects**.

Introduction to Program Design

We talked about breaking code into modules



One way to think about and represent modular code is using a structure chart.

Structure Charts

Adapted from: Robertson, L.A 2014, *Students Guide to Program Design*, Newnes. Section 7.2

Structure charts – Ruby

Below provides an outline of a Ruby program for this task.

But how is data passed around? We need to extend our structure chart notation.

Note: += adds to the total with the value returned from the function.

Structure charts - with data

Below we see arrows annotating the chart to show the movement of data.

Now we have some data being passed down and up.

These represent return values and parameter passing in the code.

Adapted from: Robertson, L.A 2014, Students Guide to Program Design, Newnes.
Section 7.2

Structure charts - Hierarchy

Structure charts are hierarchical with multiple layers possible:

Resources

Remember the links in each task to books and online resources.

- The Help Desk – you should be using this.
- Discussion forums.
- Think about study groups.
- Make sure you are going to tutorials.

Tasks for this week

You need to complete the following code:

```ruby
# Have a look at the following file to see what functions it contains.
# These are available to be used in this program because of the followin
# line - you DO NOT need to copy them into this file:
require './input_functions'

def read_patient_name()
    # write this function - use the function read_string(s)
    # from input_functions.rb to read in the name as follows:
    # name = read_string('Enter patient name: ')
    # make sure you 'return' the name you read to the calling module
end

def calculate_accommodation_charges()
    charge = read_float("Enter the accommodation charges: ")
```

Using the following (from input_functions.rb):

```ruby
# Display the prompt and return the read string
def read_string prompt
    puts prompt
    value = gets.chomp
end

# Display the prompt and return the read float
def read_float prompt
    value = read_string(prompt)
    value.chomp.to_f
end

# Display the prompt and return the read integer
def read_integer prompt
```

Tasks for this Week 2

This code demonstrates how to use the functions in the file `input_functions.rb`

Structure charts

Structure charts are a hierarchical method for designing and representing programs.

- ○ True
- ○ False