SWINBURNE UNIVERSITY OF TECHNOLOGY

Introduction to Programming

Topic 3 - Week 2

KNOWING

Introduction and Overview

About this lecture:

- 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:

```
1
2  def main
3    puts 'This is a procedure called main!'
4  end
5
6  main
```

- What does this procedure do?

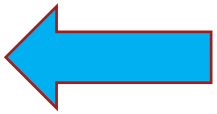
Procedures II

Procedures are called to produce **side effects.**

Procedures can call other procedures:

```ruby

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

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

main
```

```
MacBook-Pro:Code mmitchell$ ruby procedure.rb
This is a procedure called main!
This is a procedure called BY main!
MacBook-Pro:Code mmitchell$
```



This is a side effect.

Procedures III

Procedures can take arguments:

```
1
2  def subroutine arg
3    puts 'This is a procedure called BY main!'
4    puts 'It received the argument: ' + arg.to_s
5  end
6
7  def main
8    puts 'This is a procedure called main!'
9    subroutine 5
10 end
11
12 main
```

```
MacBook-Pro:Code mmitchell$ ruby procedure2.rb
This is a procedure called main!
This is a procedure called BY main!
It received the argument: 5
MacBook-Pro:Code mmitchell$
```

Procedures IV

Procedures can take **multiple** arguments:

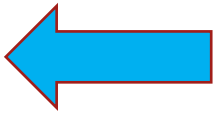
```ruby
def subroutine(arg_a, arg_b)
  puts 'This is a procedure called BY main!'
  puts 'It received the argument ' + arg_a.to_s
  puts 'It received the argument ' + arg_b.to_s
end

def main
  puts 'This is a procedure called main!'
  subroutine(5, 7)
end

main
```

```
MacBook-Pro:Code mmitchell$ ruby procedure3.rb
This is a procedure called main!
This is a procedure called BY main!
It received the argument 5
It received the argument 7
MacBook-Pro:Code mmitchell$
```

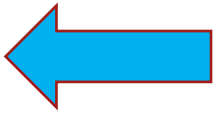
Functions

Functions **return** a value

```
2  def function
3    puts 'This is a function'
4    return 5
5  end
6
7  def main
8    value = function
9    puts 'The value returned is: ' + value.to_s
10 end
11
12 main
```



This is a side effect.



The value of whatever is here is returned

```
MacBook-Pro:Code mmitchell$ ruby function.rb
This is a function called BY main!
The value returned is: 5
MacBook-Pro:Code mmitchell$
```

This is a side effect.

Functions II

Functions can take arguments:

```ruby
def function arg
  puts 'This is a function that received arg: ' + arg.to_s
  return arg
end

def main
  value = function(7)
  puts 'The value returned is: ' + value.to_s
end

main
```

```
MacBook-Pro:Code mmitchell$ ruby function2.rb
This is a function that recieved arg: 7
The value returned is: 5
```

Functions III

In Ruby the **return** keyword is not required, the last statement is evaluated and that value is what is returned. Eg:

```ruby
def function arg
  puts 'This is a function that recieved arg: ' + arg.to_s
  5
end

def main
  puts 'The value returned is: ' + function(7).to_s
end

main
```

```
MacBook-Pro:Code mmitchell$ ruby function3.rb
This is a function that recieved arg: 7
The value returned is: 5
MacBook-Pro:Code mmitchell$
```

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:

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

- The Ruby equivalent is:

```
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**.

Signatures for Procedures

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

```
void print_student(student_record student)
```

- 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 lecture).
- EG:

```ruby
# This function takes an integer argument and returns an integer
# equal to the argument 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).

Ruby Documentation

Core API:

- 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

Effects of Local Scoping

```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
```

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* an *b* in *main* due to those being different variables with a different **local** context.

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

```
MacBook-Pro-6:Code mmitchell$ ruby scope1.rb
a = 2 b = 3
a = 2 b = 3
MacBook-Pro-6:Code mmitchell$ $
```

Effects of Global Scoping

```ruby
$a
$b

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

def change
  $a = 10
  $b = 5
end

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

main
```

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

```
MacBook-Pro-6:Code mmitchell$ ruby scope2.rb
a = 2 b = 3
a = 10 b = 5
MacBook-Pro-6:Code mmitchell$
```

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:

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

    def second_procedure
      puts "The shared variable has the value: #{@shared_variable}"
    end
end

def main
  demo = Demo.new().first_procedure(42)
end

main
```

@shared_variable is accessible in all methods.

```
MacBook-Pro:Code mmitchell$ ruby class_example.rb
The shared variable has the value: 42
MacBook-Pro:Code mmitchell$
```

Artifacts I

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

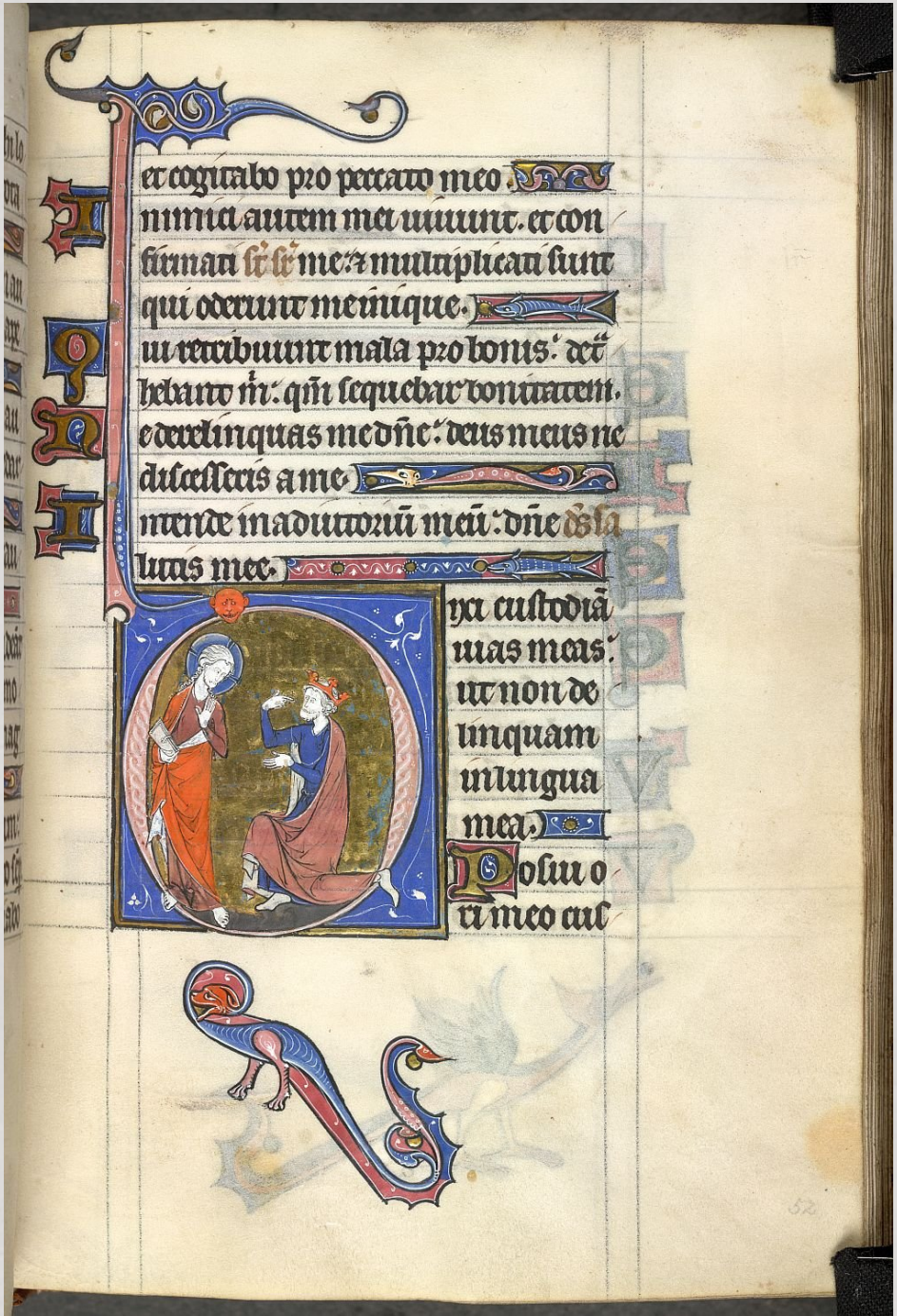
- Artifacts arise from people imposing their will to shape and design raw materials into a desired form.

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**.

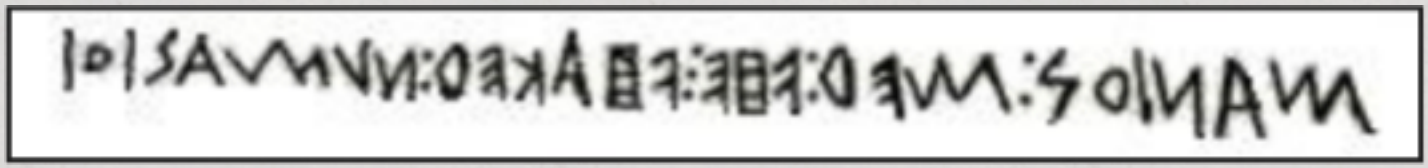
How to Write your Artifacts I

- 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

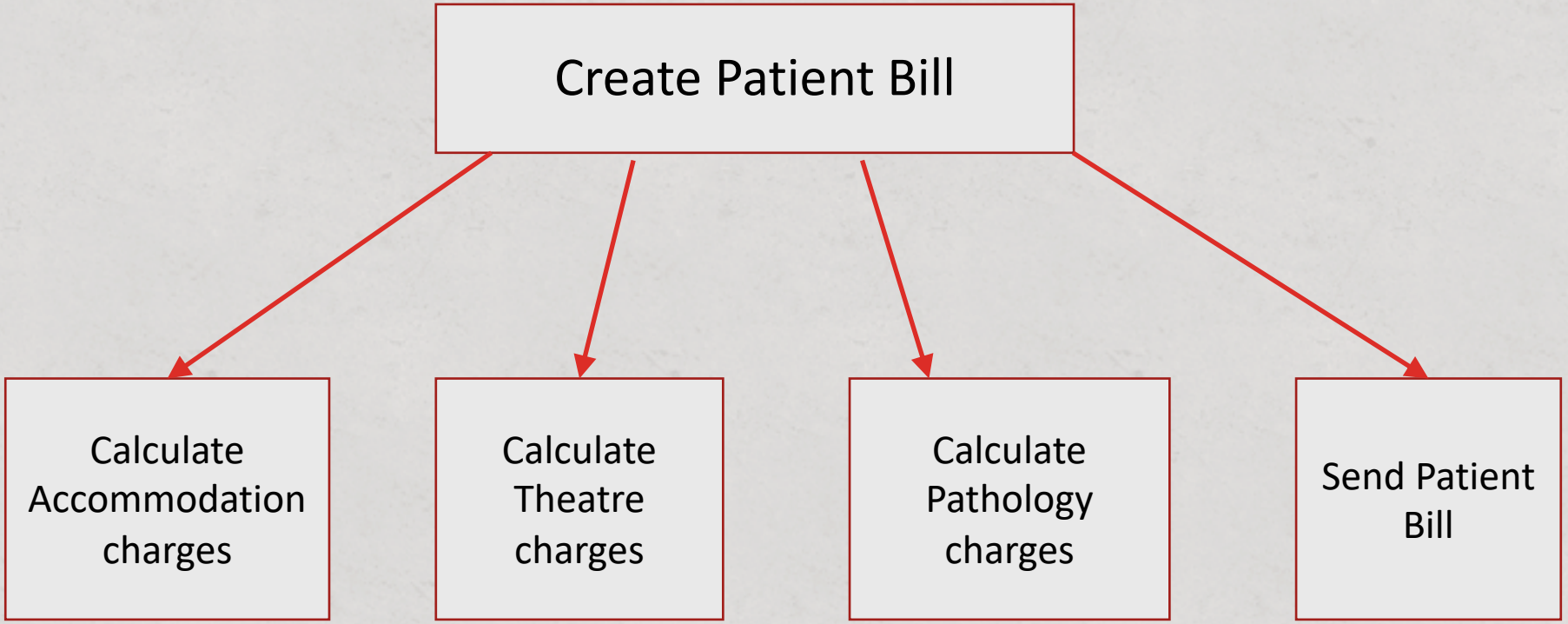
- **Procedures** and **functions** are both ways of grouping lines of code (called '**blocks**') and giving the group a name.
- Both functions and procedures can take **arguments**.
- Functions are different from procedures in that they have a **return value**.
- Instead of returning a value procedures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



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

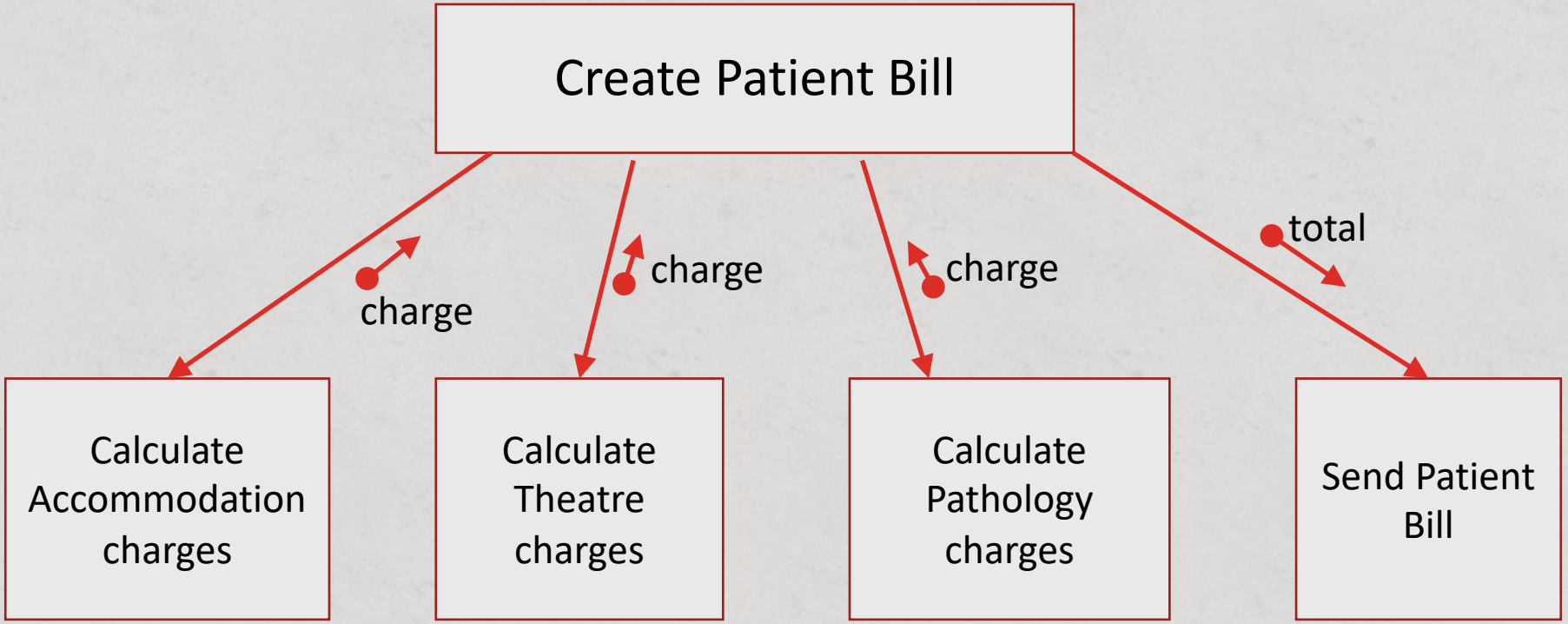
Structure charts – Ruby

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

But how is data passed around? We need to extend our notation and our code.

```
2  def calculate_accommodation_charges
5
6  def calculate_theatre_charges
9
10 def calculate_pathology_charges
13
14 def send_patient_bill
17
18 def create_patient_bill
19   calculate_accommodation_charges
20   calculate_theatre_charges
21   calculate_pathology_charges
22   send_patient_bill
23 end
24
25 def main
28
29 main
```

Structure charts - Introduction



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

Structure charts – Ruby

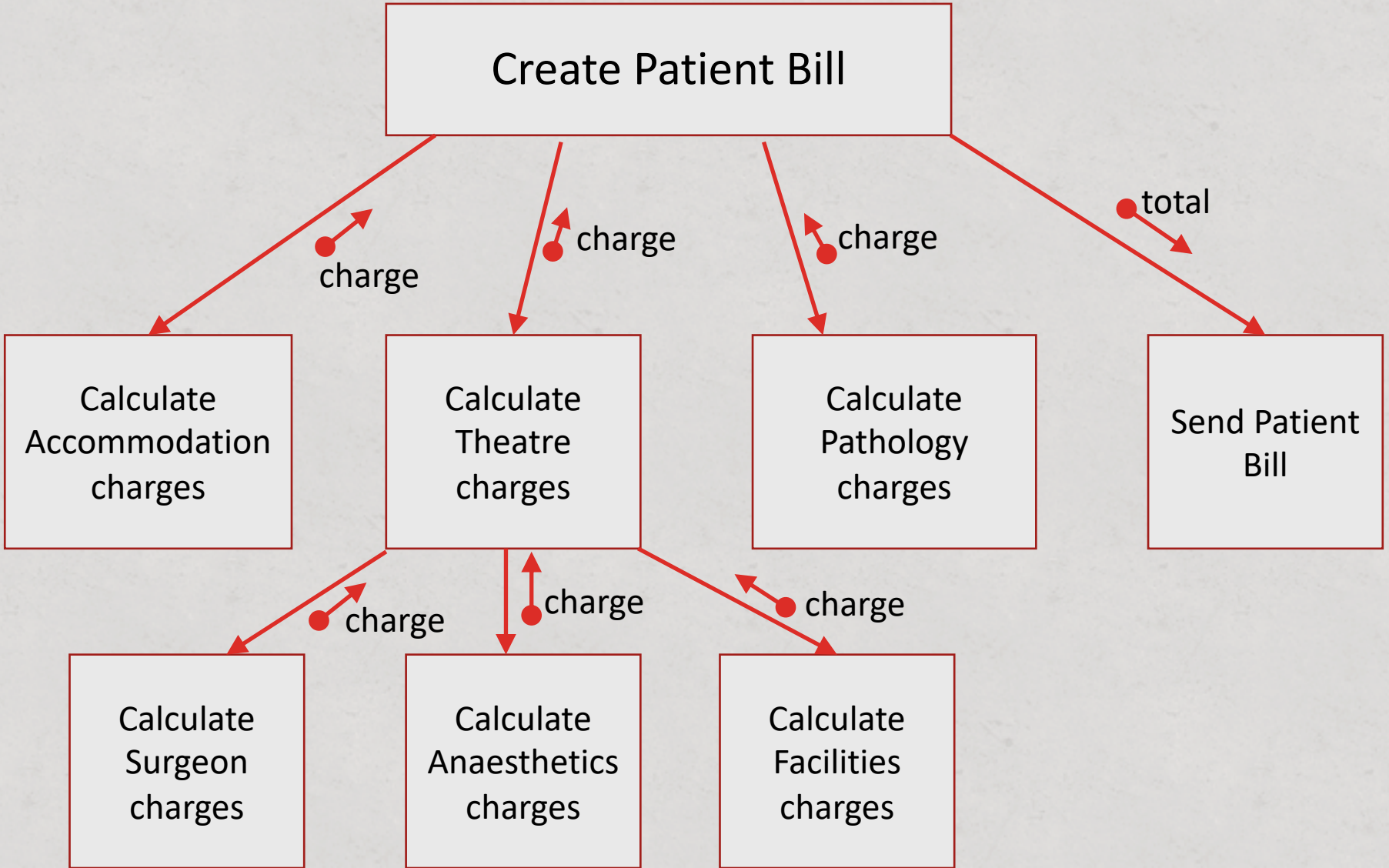
Now we have some data being passed down and up.

These represent return values and parameter passing in Ruby.

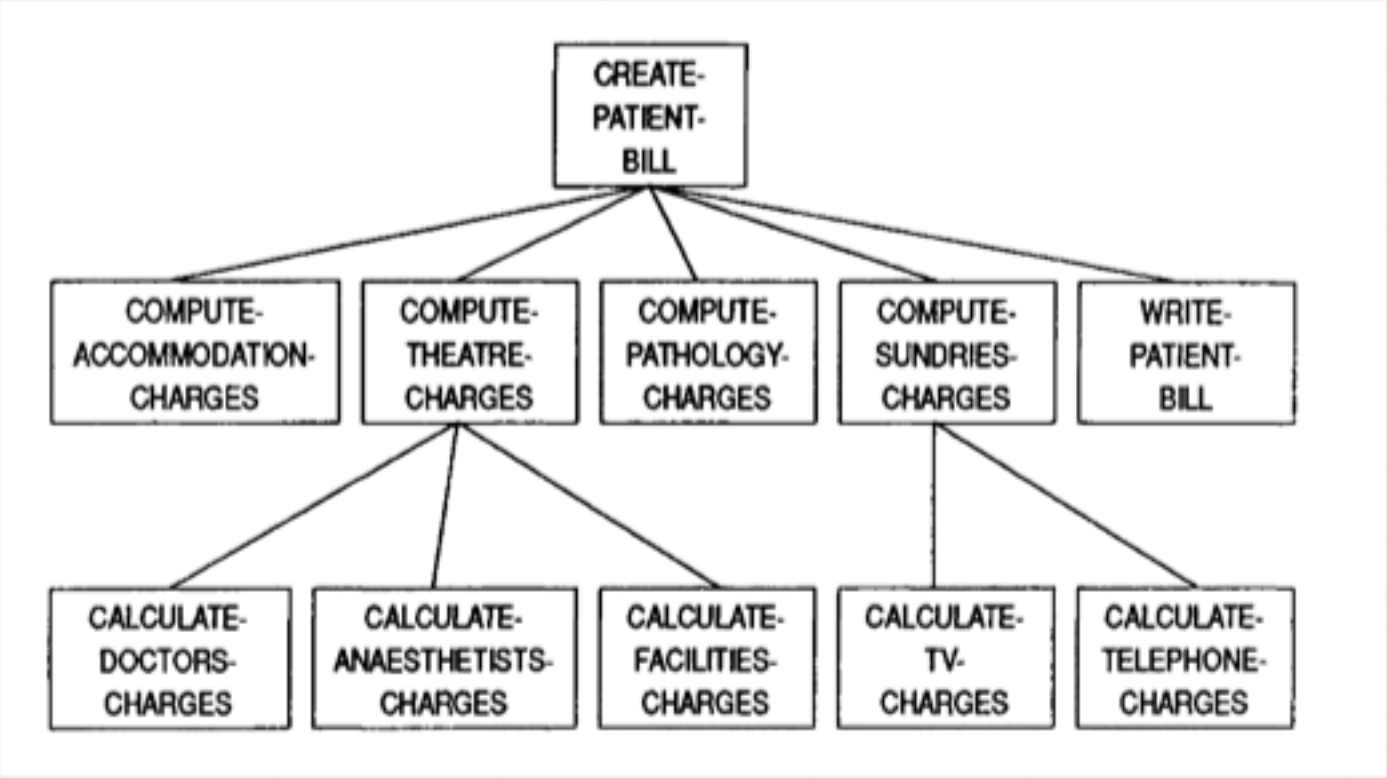
Note: += adds to the total with the value returned from the

```ruby
def calculate_accommodation_charges

def calculate_theatre_charges

def calculate_pathology_charges

def send_patient_bill(total)

def create_patient_bill
  total += calculate_accommodation_charges
  total += calculate_theatre_charges
  total += calculate_pathology_charges
  send_patient_bill(total)
end

def main

main
```

Structure charts –Adding Layers



Structure charts - Introduction



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 – 2.3P

You need to complete the following code:

```
16  if true
17    puts 'You are a Brexit supporter'
18  else
19    puts 'You are NOT a Brexit supporter'
20  end
```

Using the following (from *input_functions.rb*):

```
33 def read_boolean prompt
34   value = read_string(prompt)
35   case value
36   when 'y', 'yes', 'Yes', 'YES'
37     true
38   else
39     false
40   end
41 end
```

Interactive Ruby (online)

https://www.tutorialspoint.com/execute_ruby_online.php

You can also use irb (interactive Ruby) from the command line or terminal.

Next Week ..

We look more at conditional statements (if statements) and looping (iteration) in Ruby code.

Thank you.