

# Week 1 - Topic 2: Writing a simple structured program

---

## Starting to learn to program

This lesson looks at the following:

- Programming principles – sequence, selection, iteration, modularisation.
- Keywords, variables, constants, literals, types.
- Input and Output – basic input/output, sources and destinations (eg: file, terminal, network).
- Formatted output. Formatted input.
- Desk checking, testing, selecting test data

---

# Sequence

Is Sequence important in the following:

- Get out of bed
- Have a shower
- Get dressed
- Have breakfast

☐ True

☐ False

---

# Selection

- A point of decision
- You may have seen this in flow charts

Eg:



We will look at selection in more detail in lesson 3.

---

## Iteration (repetition)

Iteration is to do something over and over again. E.g:

1. Lay first brick
2. Get next brick
3. Lay next brick

Repeat Steps 2 – 3 until wall is finished (or lunch time – whichever comes first)

We will look at how you do this in a program also in Lesson 3.

---

# Modularisation

A modular design in programming is one that has different separate (simpler) components, rather than one (usually more complex) integrated component.

Modular program design allows the re-use of components in new contexts.

Re-use has the following benefits:

1. Code is more reliable as re-used modules have generally already been well tested
2. Code is cheaper to produce as using existing modules/components saves you time of writing them yourself
3. Code is quicker to produce as you are combining existing components

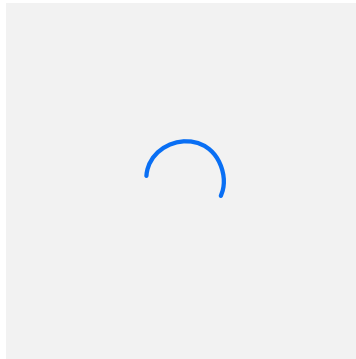
---

Which of the following is more modular:

Which of the following best demonstrates modular design principles:



NAD stereo system from Audio Trends in Ringwood – with separate components



Crosley Record player from ASOS. All-in-one.

---

# Writing Ruby Programs

We will look now at:



- Keywords
- Basic program structure
- Basic data types
- Variables
- Basic input and output

---

# Keywords

These are words which are reserved as having a special meaning in the language.

Eg: *def*, *end*.

 <b>Run</b>	RUBY	
<pre>1 2 def my_procedure 3   # some code 4 end 5</pre>		

They are usually highlighted in an IDE (Integrated Development Environment)

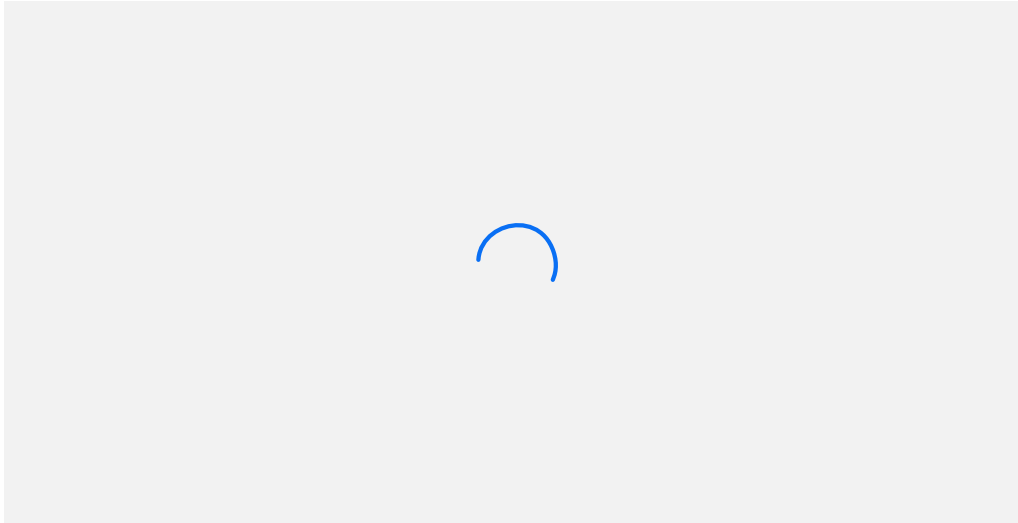
Visual Studio Code, Atom, Sublime Text, Notepad++ are all editors you can use to write Ruby code.

But we recommend using Visual Studio Code.



## Basic Program Structure

Below is a basic program:



▶ Run

RUBY



```
1 # Lines with a # are comments - i.e not code but annotations of code
2 # Below is the procedure called main.
3 # main includes/is a block of code.
4 def main()
5     # Below is the second line of code executed
6     puts("Hello World")
7     # Code in a block is indented
8 end
9
10 # This is the first line of code executed
11 # It calls the procedure main() above
12 main()
13
14
```

---

# Stepping through a Ruby program

In this case we will step through a Ruby program line-by-line.

Save the code here to your machine (with Ruby installed) save it as `first.rb`

In the Terminal or DOS prompt type:

```
ruby -r debug first.rb
```

Type 'list' - what is shown?

Type 'step' - what is displayed? What line of the program is being executed?

Type 'step' again - what line are we up to now?

Type 'step' once more - what has happened?

# Basic data types and the assignment operator

For now we will deal with three main **data types**:

- Strings – these are a 'string' of characters eg: "hello"
- Integers – these are whole numbers e.g: 10
- Real (or floating point) numbers e.g: 3.142
- Booleans – these are used to represent true or false (which are keywords)

We put a value in a variable by using the **assignment operator (=)** which looks like an 'equals' sign.

▶ Run

RUBY

```
1
2 a_string = "My Name"
3 an_int = 10
4 a_float = 24.56
5 a_boolean = true
6
7 puts("The Variables are: ")
8 puts(a_string)
9 puts(an_int)
10 puts(a_float)
11 puts(a_boolean)
```

Any of these values can be stored in the computer using **variables** - which are a named part of the computer's memory which can have its contents changed as the program executes.

We can also declare **constants**, whose value is set once and should never change after that. Constants are usually given upper-case names e.g see the constant **PI** below:

▶ Run

RUBY



```
1 # a constant:
2 PI = 3.142
3 print("The constant value PI is: ")
4 print(PI)
5 # print an empty line
6 puts()
7
8 # a variable:
9 a_variable = "first value"
10 print("The variable value of a_variable is: ")
11 print(a_variable)
12 # print an empty line
13 puts()
14
```

What happens if we remove the two lines with `puts` from the code above? Try it and see

## Basic input and output

We have seen some simple output in the examples already using `puts()` - which outputs *string* type variables to the terminal window (or variables converted to string type using `.to_s`).

But we also can read in *string* values from the terminal (i.e read in text that the user types in). One way to do this is to use a function called `gets()` (we will explain functions in a later lesson, but basically a function provides a value that can be assigned to variable).

The following code demonstrates the use of both `puts()` and `gets()` to print out and read in string values from the terminal:

▶ Run

RUBY

```
1 def main()
2   puts("Please enter your child's name: ")
3   name = gets
4   puts("the child's name is: " + name)
5
6   puts("Please enter your child's age: ")
7   age = gets()
8   puts("Your child's age is: " + age)
9 end
10
11 main()
```


What is the `+` sign doing in the code above? Hint: it is not addition.

## Converting between data types

If we read a number in as a *string* we cannot do any calculations with the number until we convert it to a number format in the computer. We can convert a *string* value (like the value stored in the variable **age**) to an *integer* using **.to\_i** (we might first need to remove any white space eg: spaces, tabs or newlines using **.chomp**).

We can also change an *integer* to a string using **.to\_s** as done in the last part of the last line in the code below:

eg:

<b>▶ Run</b>	RUBY	
<pre>1 def main() 2   puts("Please enter your child's age: ") 3   age = gets() 4   puts("Your child's age is: " + age) 5 6   # But age is stored as a string, if we want to calculate with it 7   # we need to convert it to an integer: 8 9   # first remove white space: 10  age = age.chomp() 11  # then convert to an integer: 12  age = age.to_i() 13 14  # Now we calculate with age:</pre>		

# Operators and Statements

We have seen two operators already - one is the **assignment** operator: `=` the other is the `+` operator to concatenate strings . Most languages also have the standard mathematical operators such as `+`, `*` (multiplication) and `/` (division).

## Ruby Operators

The first line of the code below takes a **hard-coded** value (10) and assigns it to the variable *sub\_total*. The next line contains a **statement** that **evaluates** the value of the variable *sub\_total* then multiplies that by the value of the constant *FRACTION* before **assigning** the result of the addition to the variable *total*.

 Run	RUBY	
<pre>1 FRACTION = 0.5 2 3 sub_total = 10 4 total = sub_total * FRACTION 5 puts(total) 6 7</pre>		

---

## Quiz Question

Here are some definitions of terms we have encountered so far:

1. a syntactic unit of an imperative programming language that expresses some action to be carried out.
2. a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce final result.
3. determines the format in which data is stored.
4. a word that is reserved by a program because the word has a special meaning.
5. Places a value in a variable or constant.
6. produces a value from an expression

**Re-order the following terms so as to best match the order of the statements above:**

type

statement

keyword

evaluation

operator

assignment operator



# Scope

Variables and constants exist within a **scope**.



The following code shows two types of scope - the constant `FRACTION` has **global** scope - it is accessible to all blocks of code (i.e procedures).

The variable `total` has only **local** scope - i.e it is only accessible in the block in which it was first used (or defined) in this case the block is the procedure `add_to_total`:

Try and use `puts` to output `total.to_s` in `main` and see what happens (uncomment that line of code then comment it again after running).

What happens if we use `puts` to output `FRACTION.to_s` in `main` (uncomment the other line of code)?

Why the difference?

 Run	RUBY 
<pre>1 FRACTION = 0.5 2 3 def add_two_numbers(a, b) 4   total = a + b 5   puts 'total is: ' + total.to_s 6 end 7 8 def main() 9   # puts total.to_s 10  # puts FRACTION.to_s 11  add_two_numbers(16, 5) 12 end 13 14 main()</pre>	

---

# Designing and Testing Programs

When we write a program we need to think about:

- What do we want the program to do?
- How can we check it is doing what we expect?

Ideally we will have some test data that we can use to check that the program is doing what we expect.

We will look at how to approach testing a program in the tutorial/lab tasks this week.

---

## Tasks for This Week

- Tutors will go over these in the lab classes.
- You should try and complete (or at least start) tasks marked with a T in the tutorial/lab classes. These are referred to as Tutorial Tasks.
- Tasks with a P, C, D or HD in their abbreviation are referred to as Level tasks.

---

# Refresher Quizz: Programming Principles

Before we go - do you recall which of the following are Structured Programming Principles?

☐ Selection

☐ Modularisation

☐ Iteration

☐ Keywords

☐ Sequence

☐ Definitions

---

# Data Types

An error occurred.

---

Try watching this video on [www.youtube.com](https://www.youtube.com), or enable JavaScript if it is disabled in your browser.

A short video explaining data types.

---

## What else to do this week?

- Visit the Programming Help Desk in Discord.

You can work there even if you are not seeking help (but maybe if it is crowded allow space for those who need help).

- Check the timetable for when COS10009/COS60006 staff are rostered on.
- Use the booking system at the front of the room to add your name to the list.

**Make sure you attempt this weeks tutorial/lab task and the first Pass task.**