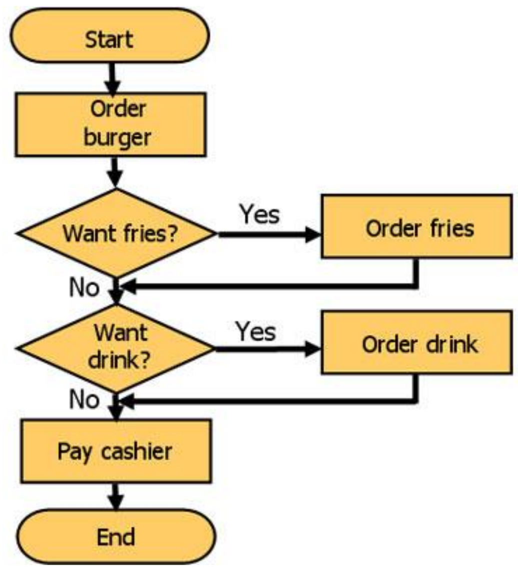
SIT105 - Critical Thinking and Problem Solving for IT
Class 10

Part 1 – Flowcharts
Part 2 – Assertions

A/Prof. Xiao Liu
xiao.liu@deakin.edu.au

Part 1 Content

1. Introduction
2. Symbols
    - Flow line
    - Terminal
    - Input and Output
    - Processing
    - Predefined processing
    - Decision



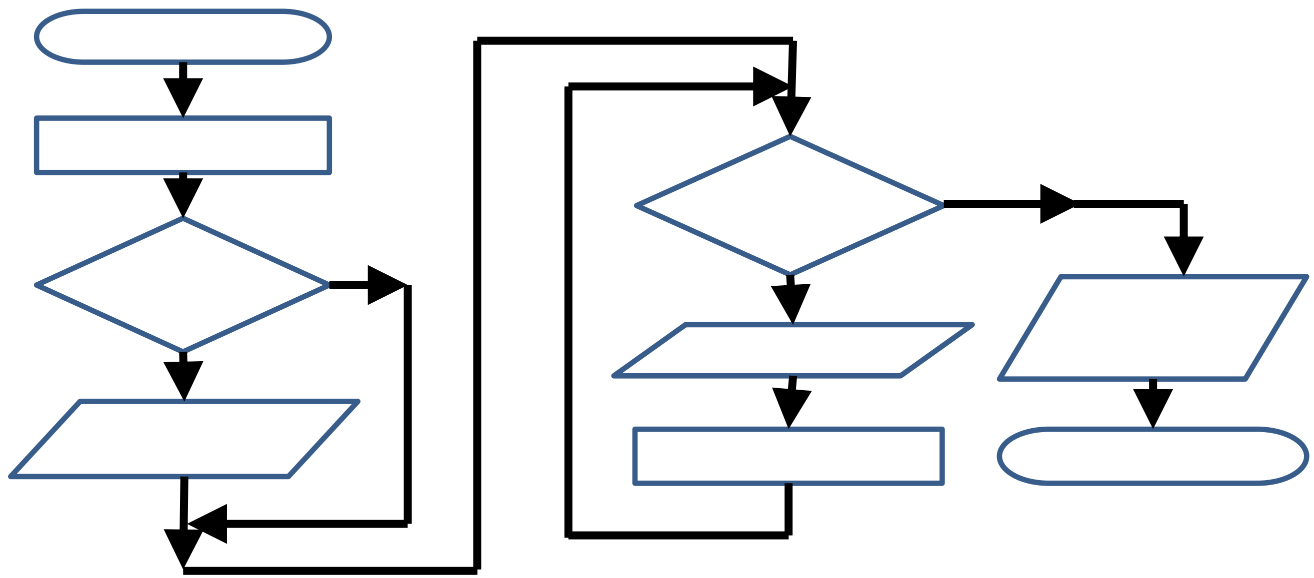
Introduction

Flow of control is used to determine what section of code is run in a program at a given time

A flowchart:

- Depicts an algorithm
- Consists of symbols and directed lines
- Represents the flow of control

For example:



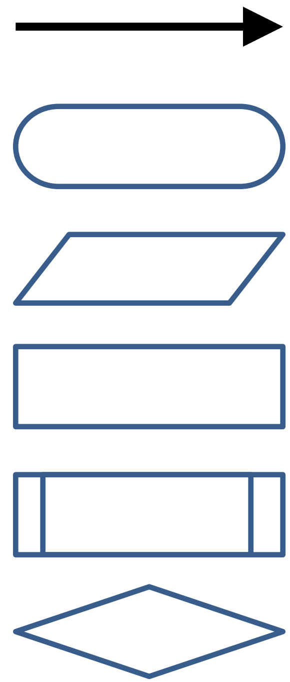
Top 4 Reasons for Using Flowcharts

1. Programming: complex program logic can be modelled effectively using a flowchart. Hence making it easier to understand by everyone in the team and clients.
2. Troubleshooting Guides: If we reach bugs or errors in our application, a good troubleshooting flowcharts can cut the problem solving time massively.
3. Training materials: these are often created using flowcharts because they're visually stimulating and easy to understand for all stakeholders involved.
4. Quality Management: your organisation may employ quality management systems - such as ISO 9000 for example. In such environments, flowcharts are not only useful but in certain situations they are actually required.

ISO International Organization for Standardization

Symbols: What do they mean?

- Flow line (joiner)
- Terminal
- Input and Output
- Processing
- Predefined processing
- Decision
- Etc.



We will discuss each of these in the next slides

NOTE – no explicit symbol for a loop. We just use flow line symbols.

Flow line Symbol

- This joins symbols together
- Usually has an arrowhead
- Indicate what to evaluate next
  - Shows the control flow

Flow of control is used to determine what section of code is run in a program at a given time

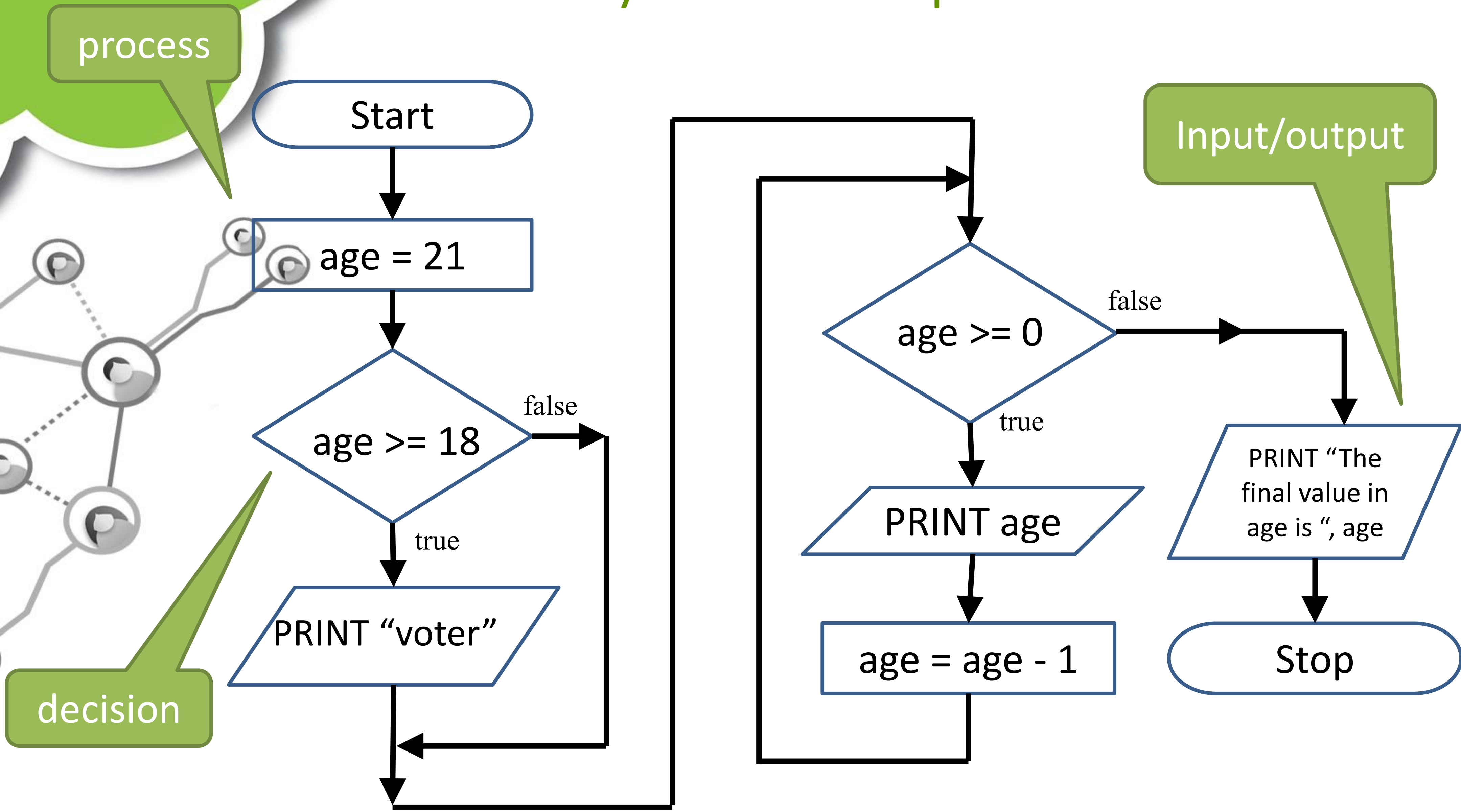
Flow line Symbol - Example 1 Algorithm

```
age = 21
IF age >= 18 THEN
      PRINT “voter”
ENDIF

DOWHILE age >= 0
      PRINT age
      age = age - 1
ENDDO
PRINT “The final value in age is ”, age
```

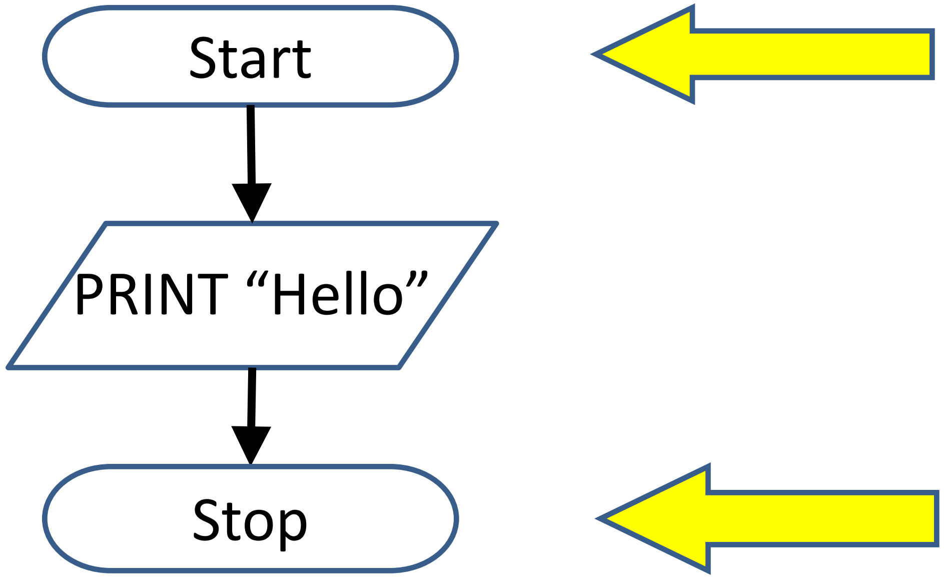
A flowchart depicting this algorithm is on the next page.

Flow line Symbol - Example 1 Flowchart



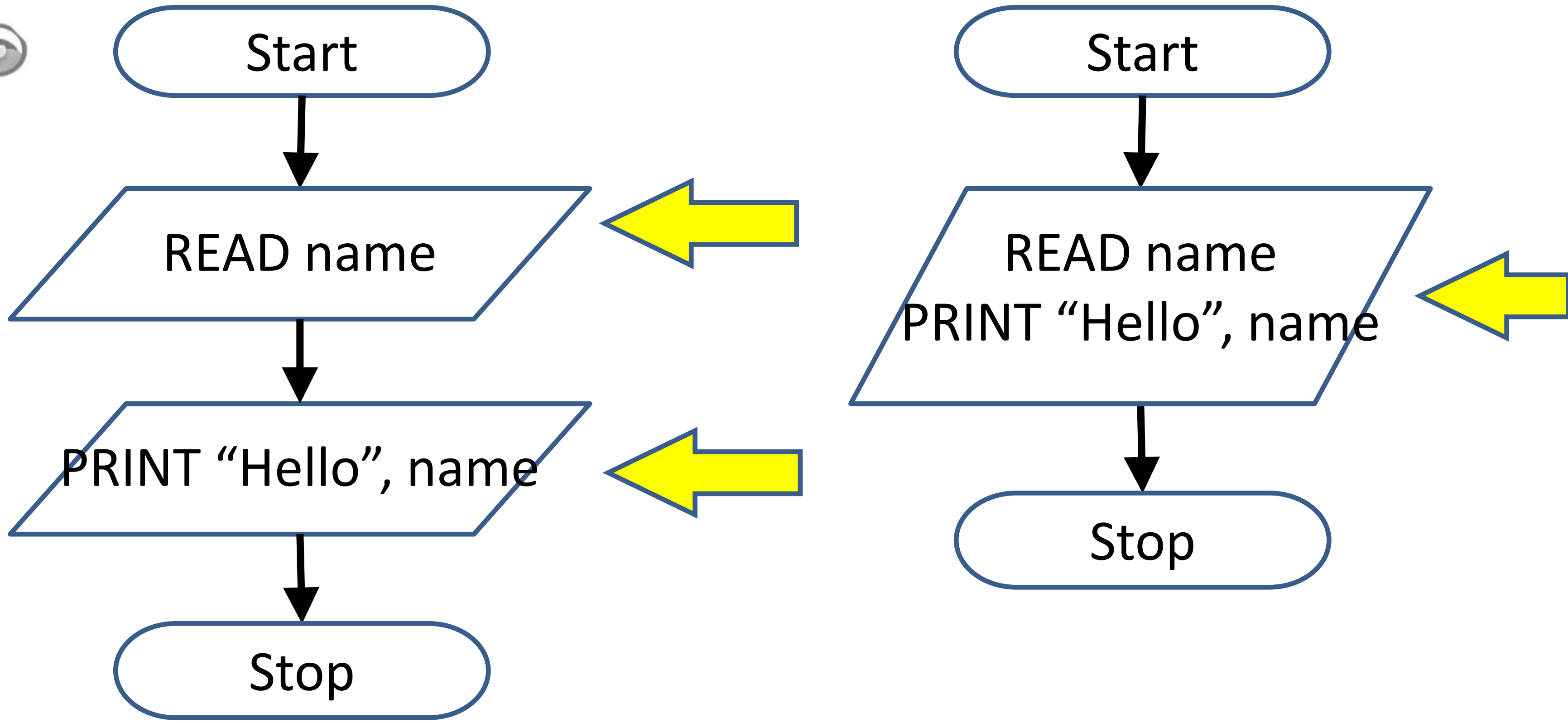
Terminal Symbol

- Depict the start and end of a module's algorithm



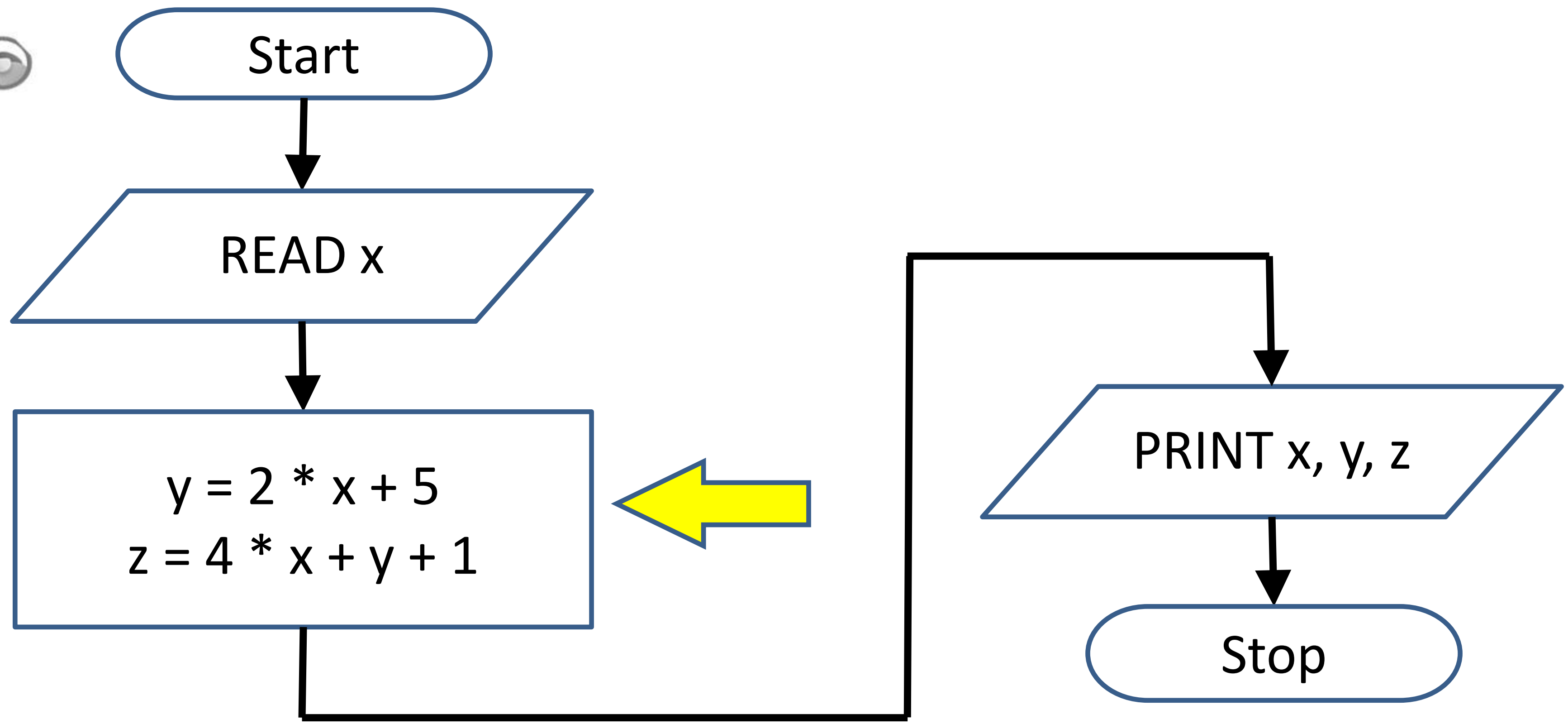
Input and Output Symbols

- Depict data input and output



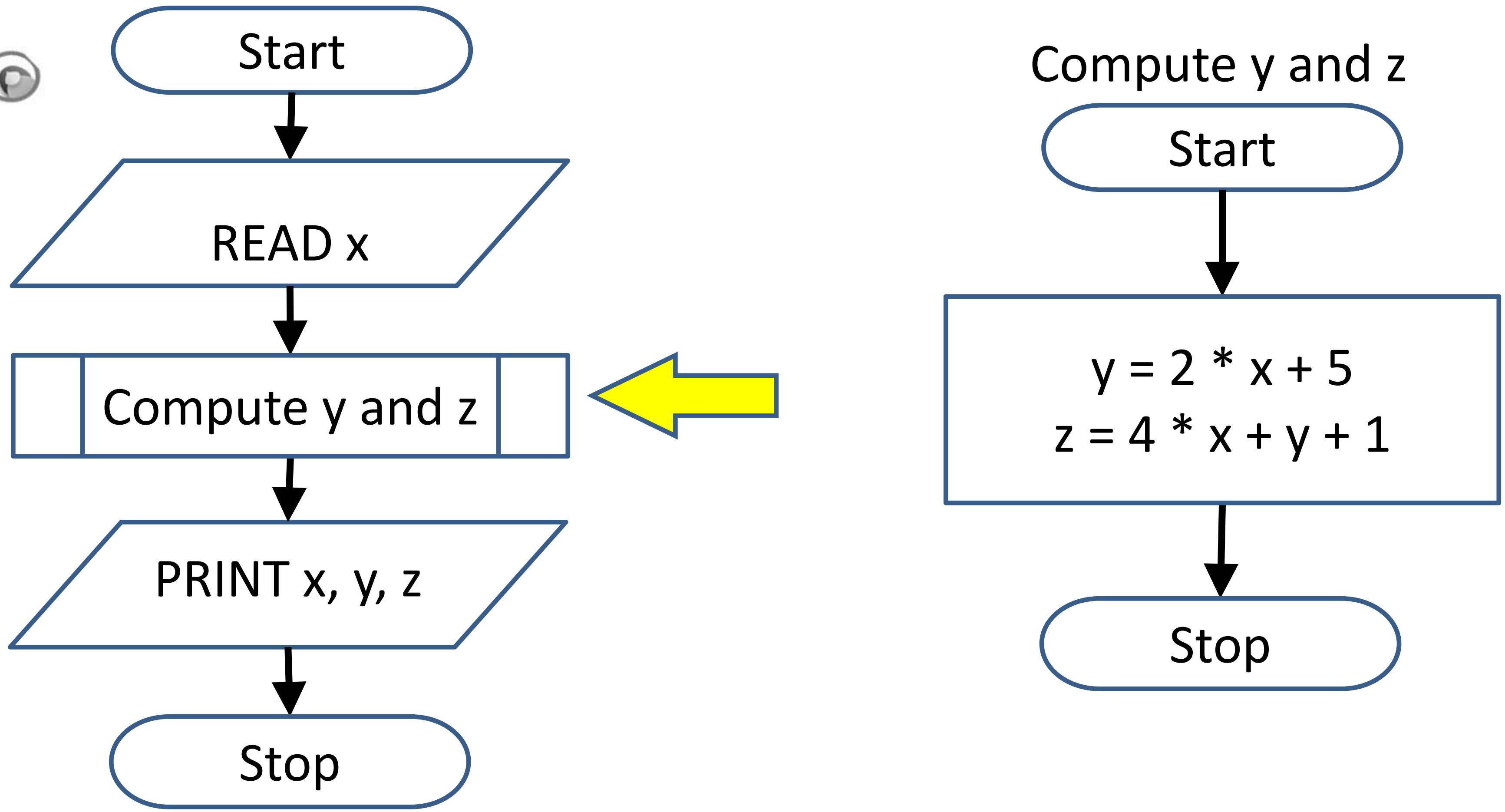
Processing Symbol

- Instruction statements such as assignments



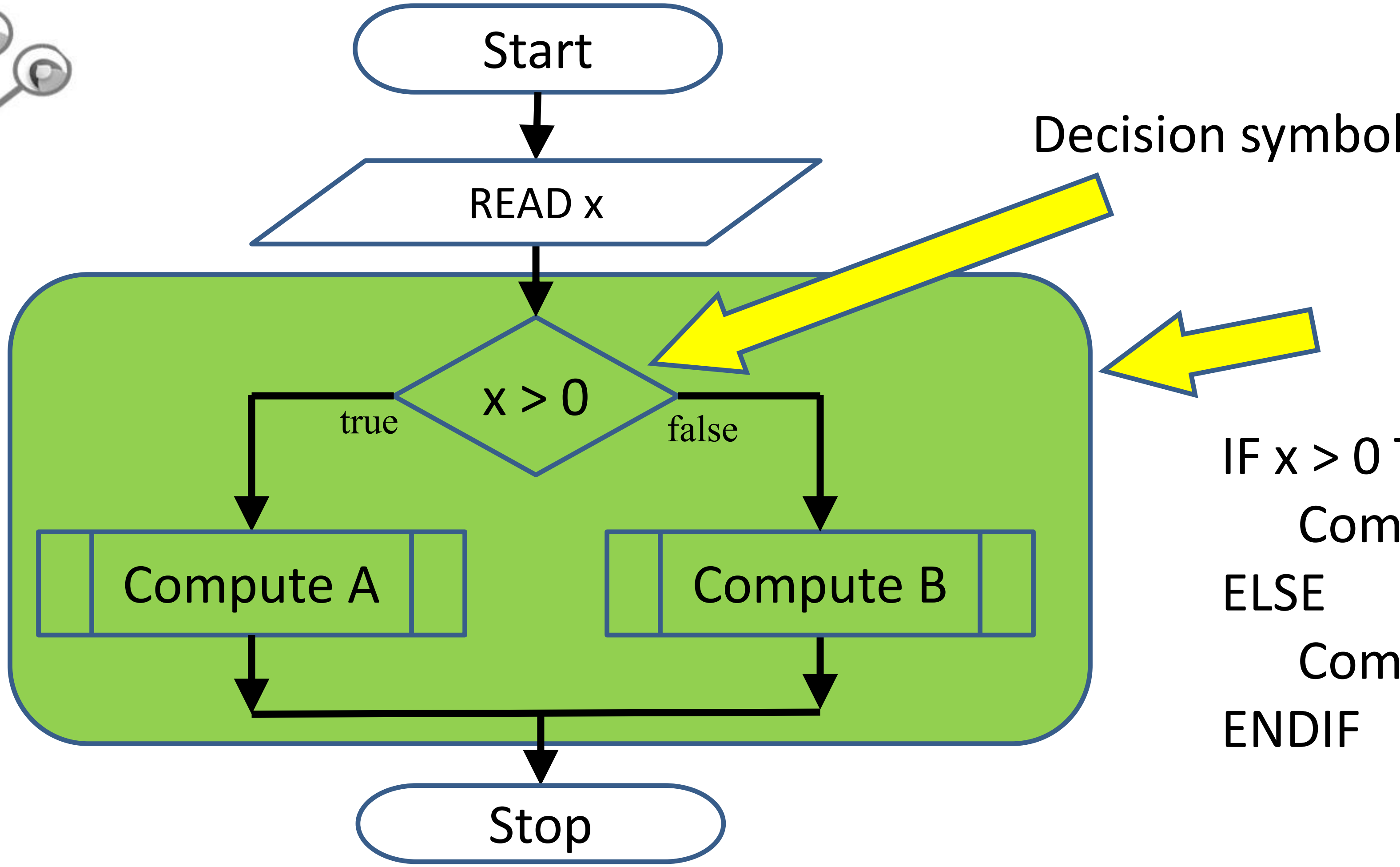
Predefined Processing Symbol

- Using an existing module



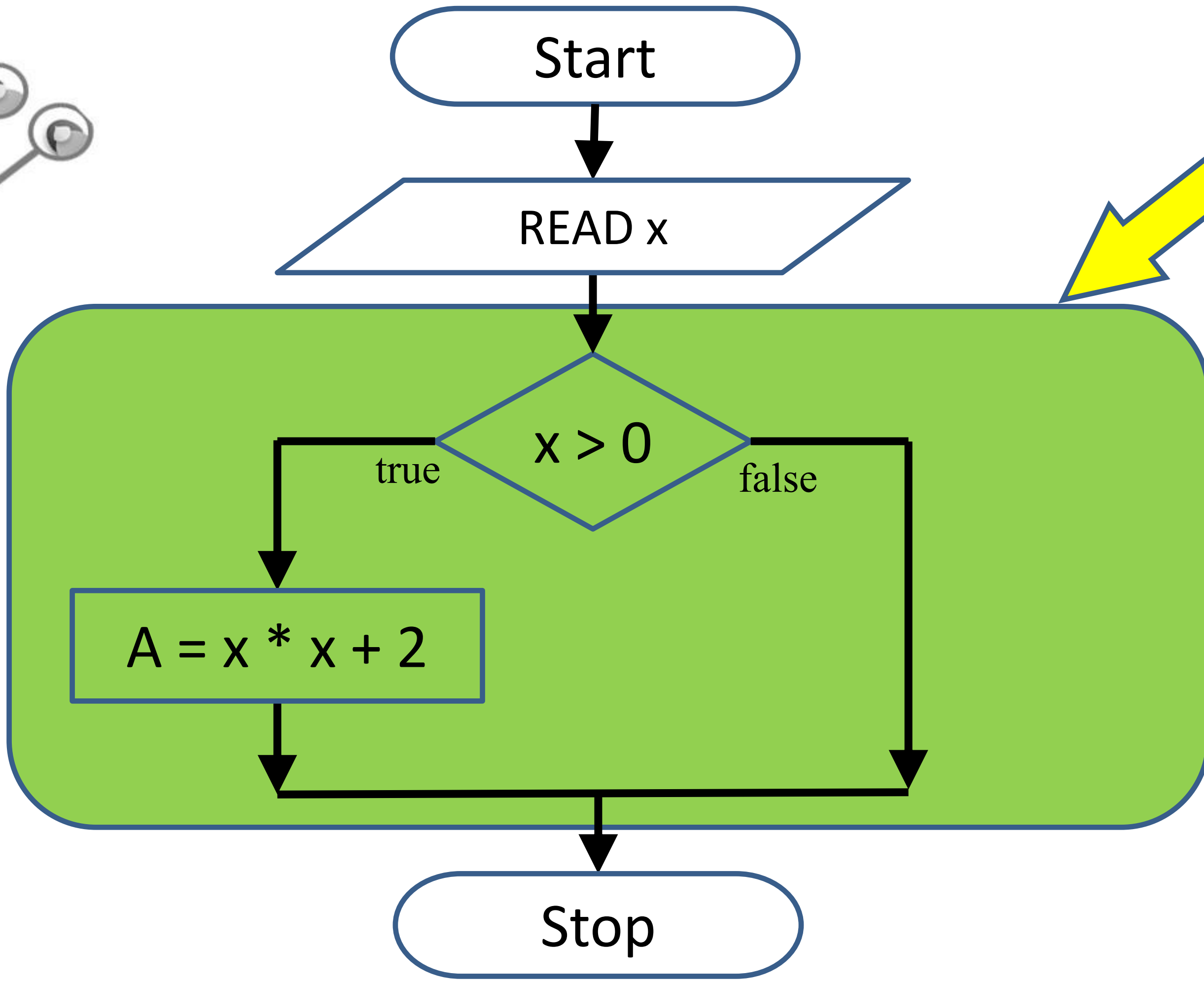
Decision Symbol - IF

- Choose a particular path



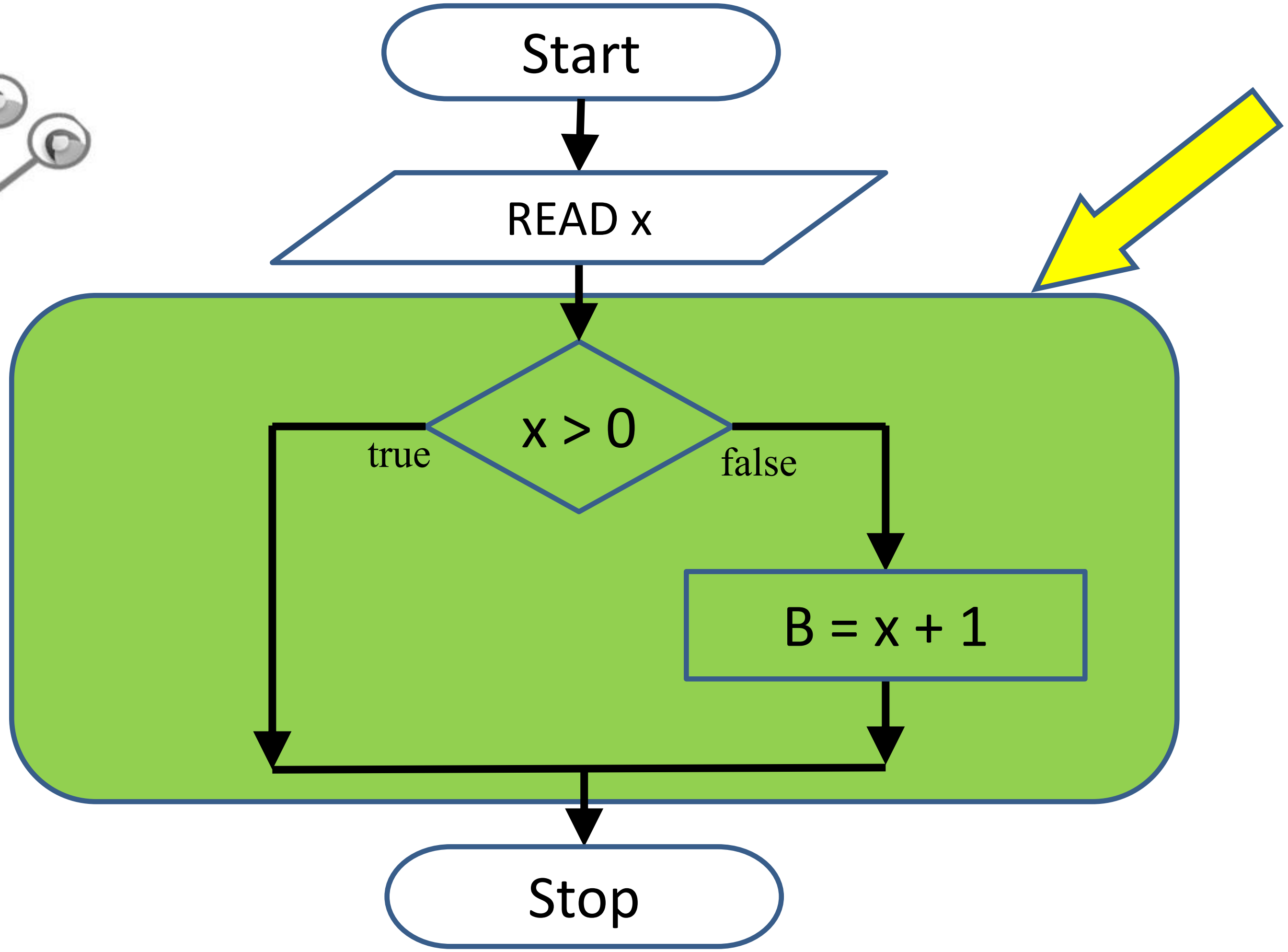
IF x > 0 THEN
    Compute A
ELSE
    Compute B
ENDIF

Decision Symbol - IF(continued)



IF x > 0 THEN
    A = x*x + 2
ENDIF

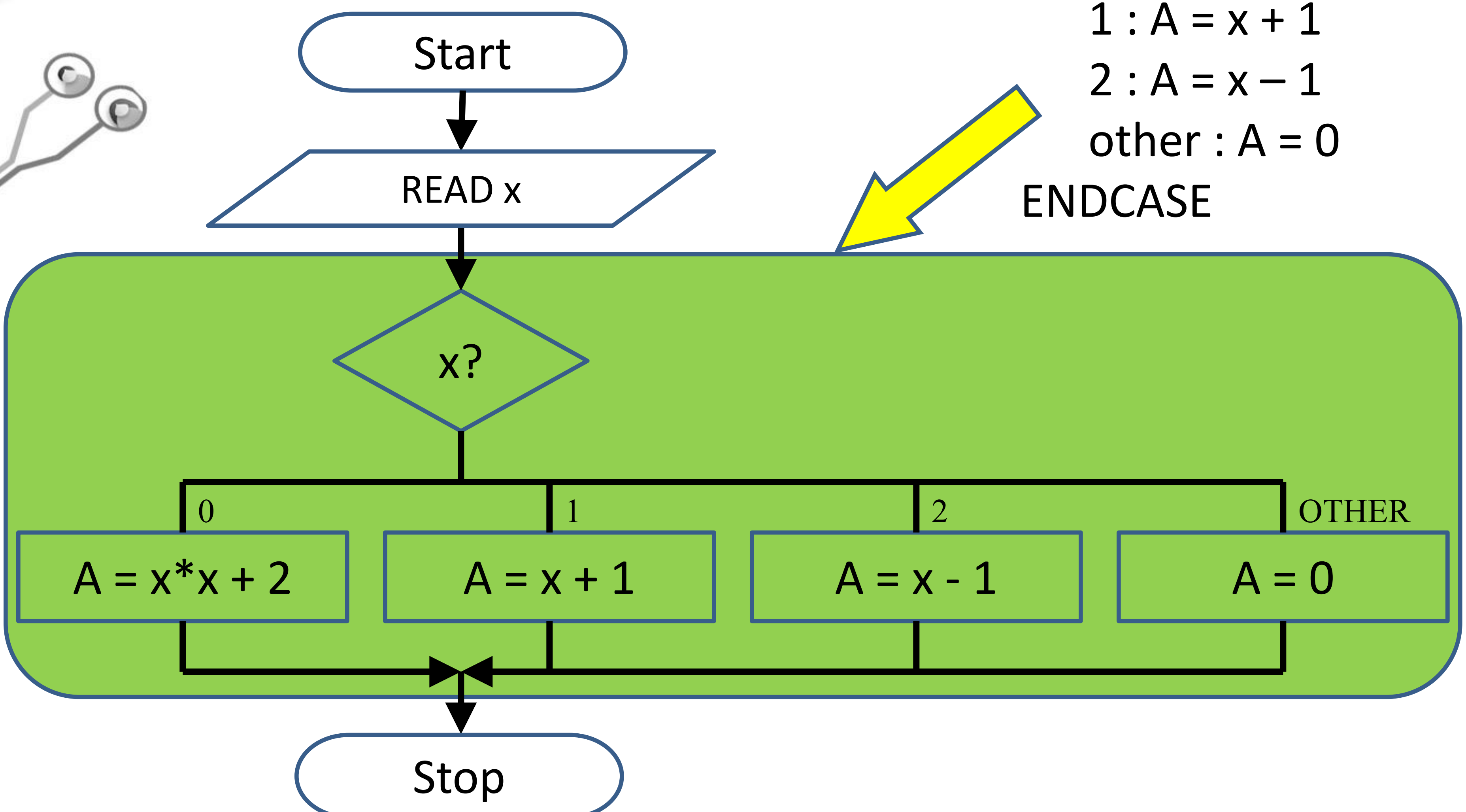
Decision Symbol - IF(continued)



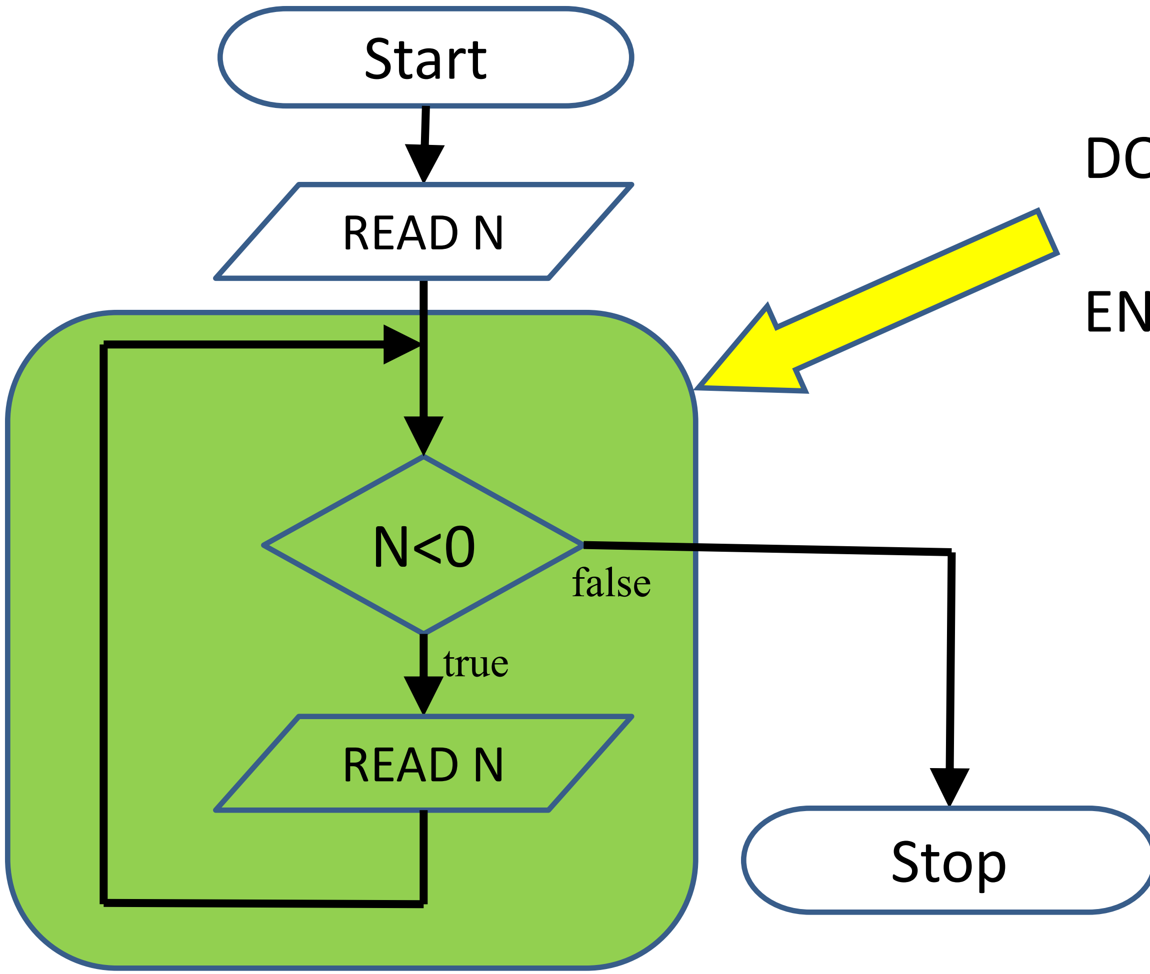

```
IF x <= 0 THEN
    B = x + 1
ENDIF
```

Decision Symbol - CASE

CASE x
  0 : A = x*x + 2
  1 : A = x + 1
  2 : A = x – 1
  other : A = 0
ENDCASE

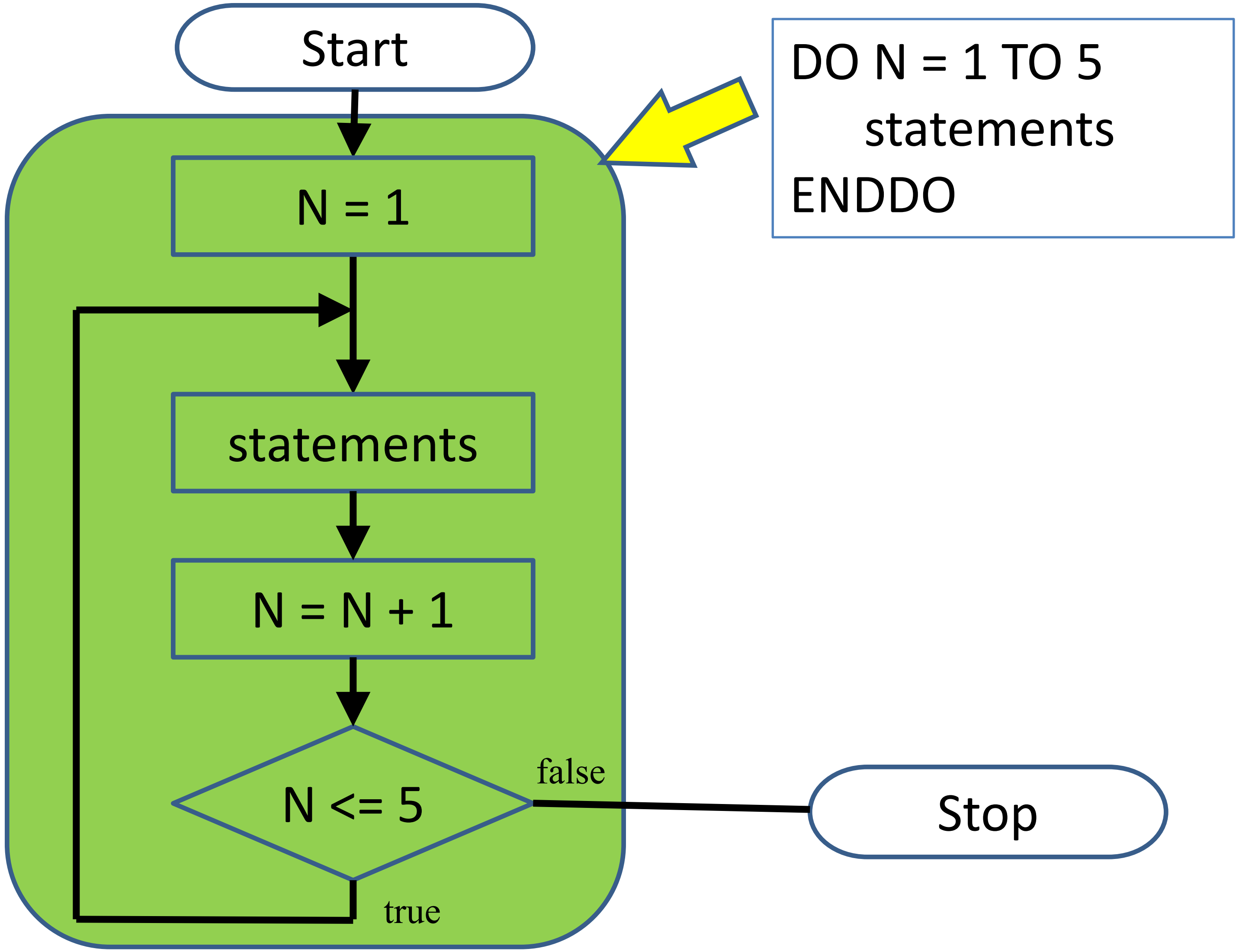


Decision Symbol - DOWHILE




```
DOWHILE N < 0
    READ N
ENDWHILE
```

Decision Symbol – DO



You might have guessed by now flowcharts can be really *useful!*

Are you someone who suffers from allergies or not really sure if you have allergies or something else entirely?

How can you know?

This could be made into an app!

Here's A Simple Flowchart To Help You!

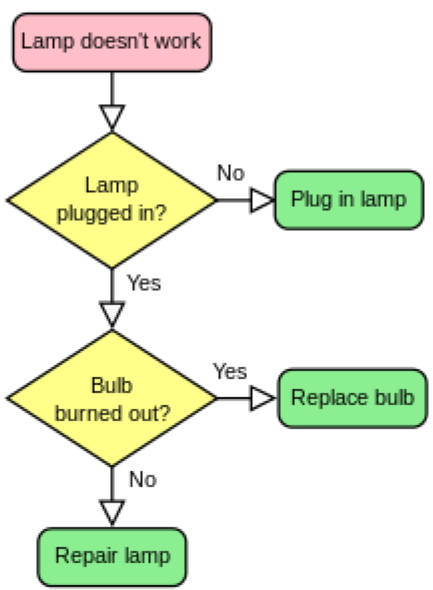
Article: http://www.huffingtonpost.com/2015/05/11/you-probably-have-allergies-flowchart_n_7129762.html

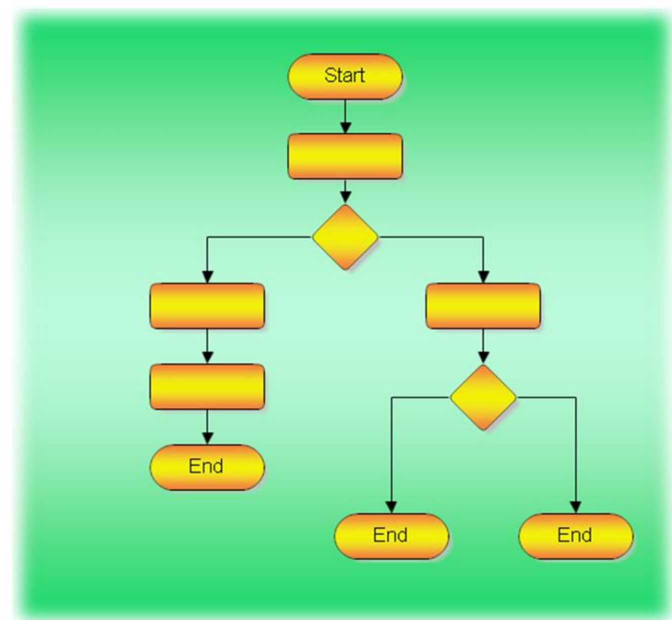
Full flowchart image here: http://big.assets.huffingtonpost.com/ProbablyAllergies.jpg

How To Draw Simple Flowchart Diagram in Visio

Flowcharts Summary!

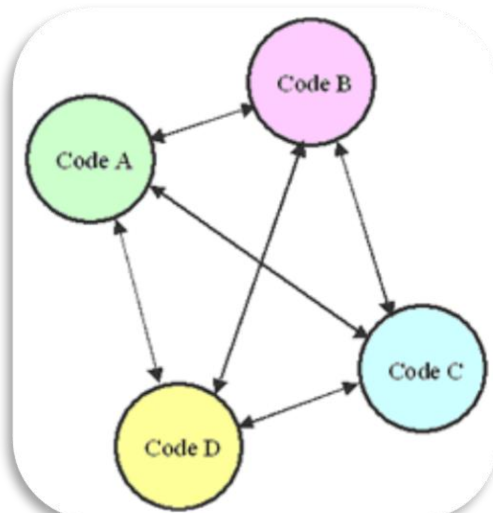
- Flowcharts are another way to:
  - Represent an algorithm
  - Develop an algorithm

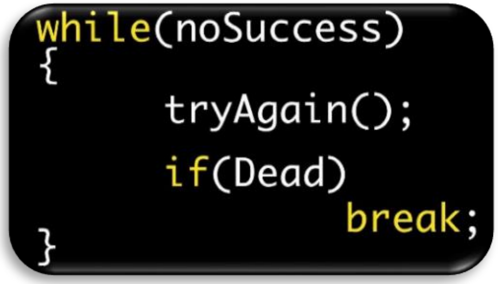




Part 2 Content

1. Introduction
2. Preconditions
3. Postconditions
4. Implementing pre and post conditions
5. Assertions
6. Examples





Introduction

- We are nearing the end of learning how to write algorithms using pseudocode!
  - We learned its not easy!
- The more statements we have, the harder the algorithm becomes to understand.
- We need to create more organisation and ‘beauty’, so it can become easier to maintain and work on.
- As we write more and more complex algorithms we always are looking for ways to ensure our pseudocode is reliable and robust (no bugs).
  - Preconditions and postconditions can help!

Introduction

Preconditions and postconditions are used to specify precisely what a module does.

When you write a complete module for example, you specify how it performs its tasks. If someone else is using your module, they do not need to know how the task is performed.

- More just need to know what the operation does without knowing how it works.
- **So more technically we describe these as:**

➢ **Precondition** is a condition or predicate that must always be true just prior to the execution

➢ **Postcodition** is a condition or predicate that must always be true just after the execution

…of some section of code or after an operation in a formal specification.

Introduction

- Frequently a programmer must communicate precisely **what** a function accomplishes, without any indication of **how** the function does its work.

What is a situation where this would occur ?

You are the head of a programming team and you want one of your programmers to write a function for part of a project.

They need the requirements

However don’t need to follow a specific method to achieve it!

Preconditions

1. A module is associated with a **set of preconditions**.

2. Preconditions must be **true prior to evaluating** a module.
   - A module is **dependant on true preconditions** to correctly carry out its tasks.
   - If any module **precondition is false**, the **results** of that module are said to be **undefined**.

Postconditions

1. How has the state of my program changed as a result of calling this method?
   - How have my local variables changed
   - What other objects have changed

1. A module has a **set of postconditions**.

2. Postconditions refer to what the module must **achieve**.
   - If any module **postcondition is false**, that module has **failed to carry out its tasks**.

Implementing pre and postconditions

- Preconditions and postconditions are **normally in the documentation** for a module and/or program.

- However, preconditions and postconditions are implemented in a real programming language using:
  1. **IF** statements
  2. **Assertions** (but what are they?... Next!)
  3. Programming language specific **statements**

Assertions? What are they?

- An assertion is a boolean expression at a specific point in a program which will be **true** unless there is a **bug** in the program.

- The role of assertions is to **identify bugs** in a program.
- Assertions do several useful things:
  - Detect subtle errors that might otherwise go undetected.
  - Detect errors sooner after they occur than they might otherwise be detected.

Assertions

- An assertion in an algorithm/program is a **condition** where it's value is always **expected to be true**.
  - If the value of an assertion is **true**, that algorithm is **assumed to be correct** to that point.
  - If the value of an assertion is **false**, there is **something wrong** with that algorithm.

Assertions

- **<u>Many</u>** assertions can be placed within an algorithm (or real program).
- Assertions can be placed **<u>anywhere</u>** in a module:
  1. at the **<u>beginning for preconditions</u>**
  2. <u>or **between**</u> the beginning and end
  3. or at the **<u>end for postconditions</u>**

Assertions can go anywhere!

```
moduleA(x, y)
        Preconditions
        Code
        Postconditions
END
```

Determine if data is correctly passed?

In order to test whether or not **data is correctly passed** from one module to another,

one can place the **same assertions just before** a module call and **just at the start** of the module.

```
mainModule
        …
        Assertion1
        moduleA(123, 789)
        …
END
```

```
moduleA(x, y)
        Assertion1
        Code
        …
END
```

First:

Real Life Example

1. Buying, with cash, a 2 litre bottle of milk at a supermarket

Question:

What will preconditions be, what will post conditions be and what are some assertions?

Buying, with cash, a 2 litre bottle of milk at a supermarket

Preconditions

- supermarket is open for business
- supermarket has a fridge of 2 litre bottles of milk
- fridge temperature is cold
- bottle is cold
- bottle contains 2 litres of milk
- seal of bottle cap is unbroken
- expiry date is in the future
- I have sufficient cash at hand
- I can carry a 2 litre bottle of milk from fridge to check-out

Postconditions (what you expect)

- I received correct change
- I have a receipt of the transaction
- I have and own a bottle containing 2 litres of cold milk
- seal of bottle cap is unbroken
- expiry date is still in the future
- supermarket is open for business
- supermarket has a fridge with one less bottle of milk
- fridge temperature is cold

Buying, with cash, a 2 litre bottle of milk at a supermarket

Algorithm

- take milk bottle from fridge
- walk to check-out
- join queue at check-out
- place bottle on bench
- wait to be served
- wait for total cost
- validate total cost
- hand over sufficient cash
- take receipt and change
- remove bottle from bench

What assertions/conditions can you place after any one of these activities?

Do I still have sufficient cash? True/False?

I own the bottle of milk? True/False

Second
Real Life Example

What will preconditions be, what will post conditions be and what are some assertions?

1. Adding two time values (hh:mm:ss) (numeric)

10 : 30 : 30 + 3 : 40 : 30 = 14 : 11 : 00

10:30 and 30 seconds	3:40 and 30 seconds	14 hours : 11 minutes

- 30 seconds + 30 seconds = 1 minute 0 seconds
- 30 minutes + 40 minutes = 1 hour : 10 mins
- 10 hours + 3 hours + that extra hour (above) = 14 hours

Adding two time values (hh:mm:ss) - Algorithm

T1 + T2 inputs and T3 output

```
AddTimes(t1, t2, t3)
  sec = t1.sec + t2.sec
  IF sec < 60 THEN
    min = t1.min + t2.min
  ELSE
    sec = sec – 60
    min = t1.min + t2.min + 1
  ENDIF
```

```
  IF min < 60 THEN
    hr = t1.hr + t2.hr
  ELSE
    min = min – 60
    hr = t1.hr + t2.hr + 1
  ENDIF

  IF hr >= 24 THEN
    hr = hr – 24
  ENDIF
```

```
  t3.sec = sec
  t3.min = min
  t3.hr = hr
END
```

If the hours are above 24, then subtract 24 from hours. So we don't get say 25 on our watch.

If #seconds less than 60 then add up the minutes.

If its not (e.g. 70 seconds) change 70 to 1 minute and 10 seconds.

Then add up the minutes. E.g. 30 + 40 + 1= 71

If the minute value is less than 60 then calculate the hours.
But if it wasn't such as 71. Subtract 60 and store 11.
Then add up the hours + 1 more hour.

Adding two time values (hh:mm:ss)

Preconditions

1. 0 <= t1.sec < 60
2. 0 <= t1.min < 60
3. 0 <= t1.hr < 24
4. 0 <= t2.sec < 60
5. 0 <= t2.min < 60
6. 0 <= t2.hr < 24

In the algorithm there is no testing if it could work.

What would our pre/post/asserts be?

On the left these ensures our numbers are in the correct ranges, before the algorithm starts.

Before the algorithm starts our numbers are in correct ranges.

Adding two time values (hh:mm:ss)

Postconditions

1. 0 <= t3.sec < 60
2. 0 <= t3.min < 60
3. 0 <= t3.hr < 24
4. t3.sec = (t1.sec + t2.sec) **mod** 60
5. t3.min = ((t1.sec + t2.sec) / 60 + t1.min + t2.min) **mod** 60
6. t3.hr = (((t1.sec + t2.sec) / 60 + t1.min + t2.min) / 60 + t1.hr + t2.hr) mod 24
7. t1 did not change
8. t2 did not change

Whatever we store here has to equal, has to equal the following formula.
(t1.sec + t2.sec) mod 60

Mod produces the **remainder** of dividing the first value by the second value.

T1 and t2 should not change, as they are our inputs

Adding two time values - Assertions

- Where would you place assertions?

```
AddTimes(t1, t2, t3)
  sec = t1.sec + t2.sec
  IF sec < 60 THEN
    min = t1.min + t2.min
  ELSE
    sec = sec – 60
    min = t1.min + t2.min + 1
  ENDIF
  assert(0 <= sec < 60)
  IF min < 60 THEN
    hr = t1.hr + t2.hr
  ELSE
    min = min – 60
    hr = t1.hr + t2.hr + 1
  ENDIF
  assert(0 <= min < 60)

  IF hr >= 24 THEN
    hr = hr – 24
  ENDIF
  assert(0 <= hr < 24)
  t3.sec = sec
  t3.min = min
  t3.hr = hr
END
```

We expect the seconds, minutes and hours value to be in the correct range.

They all have to be true for the code to do its job.

What you expect to be true to do its job.

Summary!

Precondition

- The person who calls a module ensures that the precondition is valid.
- The programmer who writes a function can bank on the precondition being true when the module begins execution.

Postcondition

- The programmer who writes a module ensures that the postcondition is true when the function finishes executing.

Summary!

- Assertions specify what must be true at **particular places** midway through a module.

- An assertion violation indicates a **bug** in the program.
  - Thus, assertions are an effective means of improving the reliability of programs- in other words, they are a systematic debugging tool.

Questions?

C# Implementation of Assertions

1. System.Diagnostics.Debug
   - using System.Diagnostics;
   - Debug.Assert(…);
2. Run in Debug mode
3. Run in Release mode (What happens?)

Debug.Assert

Debug.Assert Method

Namespace: System.Diagnostics

Assemblies: System.Diagnostics.Debug.dll, System.dll, netstandard.dll

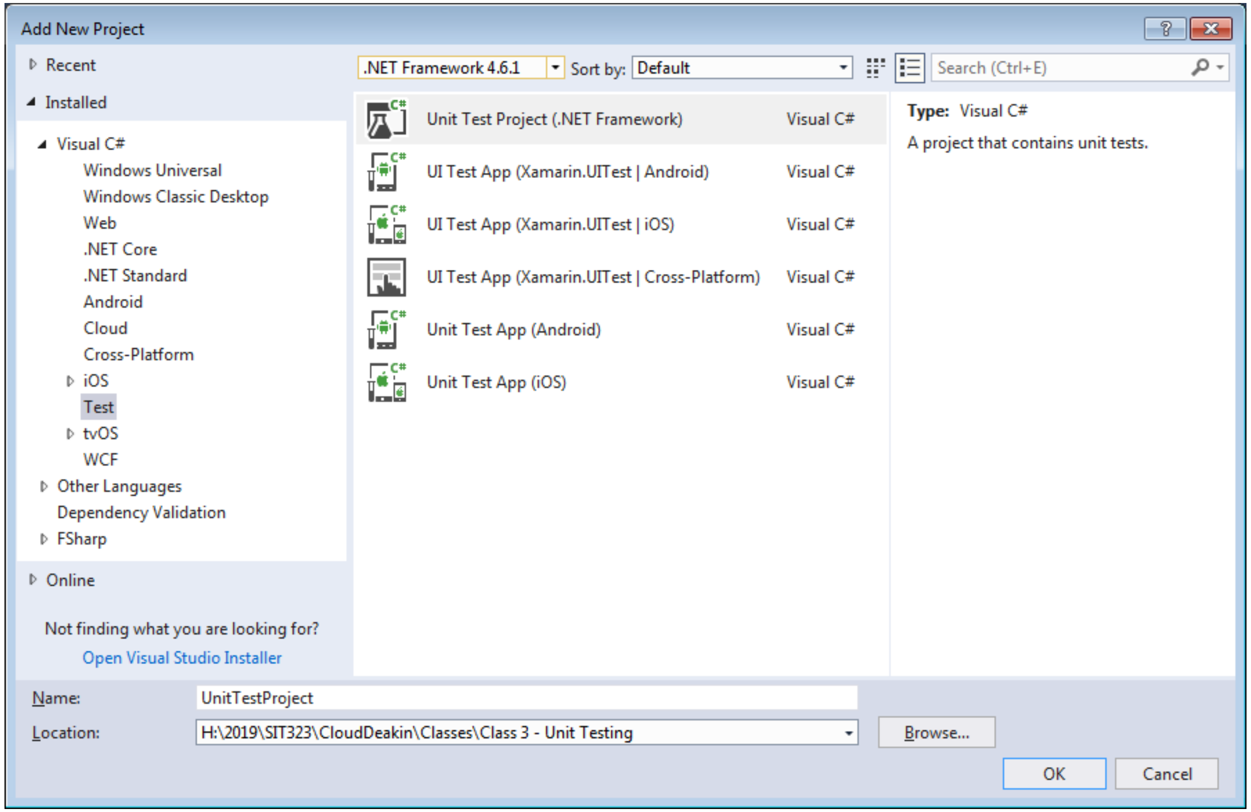
Checks for a condition; if the condition is `false`, outputs messages and displays a message box that shows the call stack.

Overloads

Assert(Boolean)	Checks for a condition; if the condition is `false`, displays a message box that shows the call stack.
Assert(Boolean, String)	Checks for a condition; if the condition is `false`, outputs a specified message and displays a message box that shows the call stack.
Assert(Boolean, String, String)	Checks for a condition; if the condition is `false`, outputs two specified messages and displays a message box that shows the call stack.
Assert(Boolean, String, String, Object[])	Checks for a condition; if the condition is `false`, outputs two messages (simple and formatted) and displays a message box that shows the call stack.

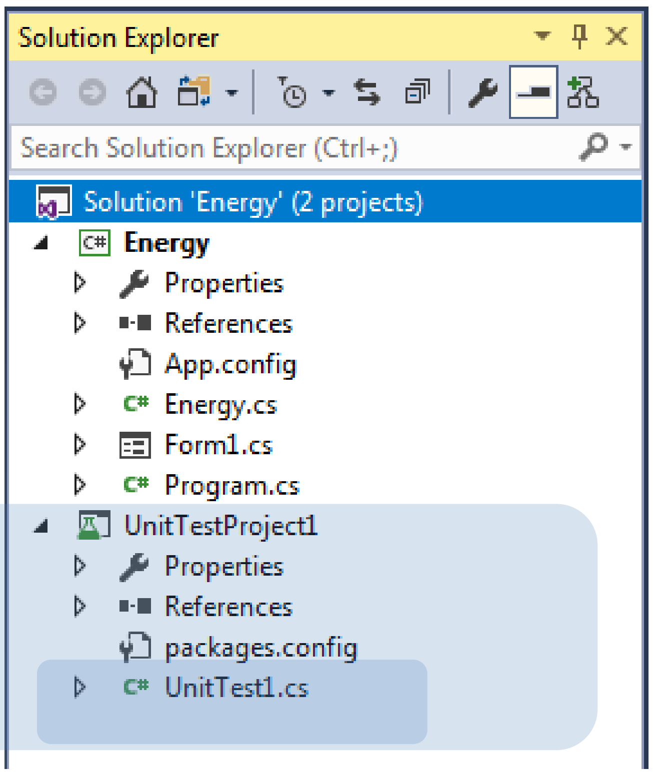
Unit Testing - Visual Studio

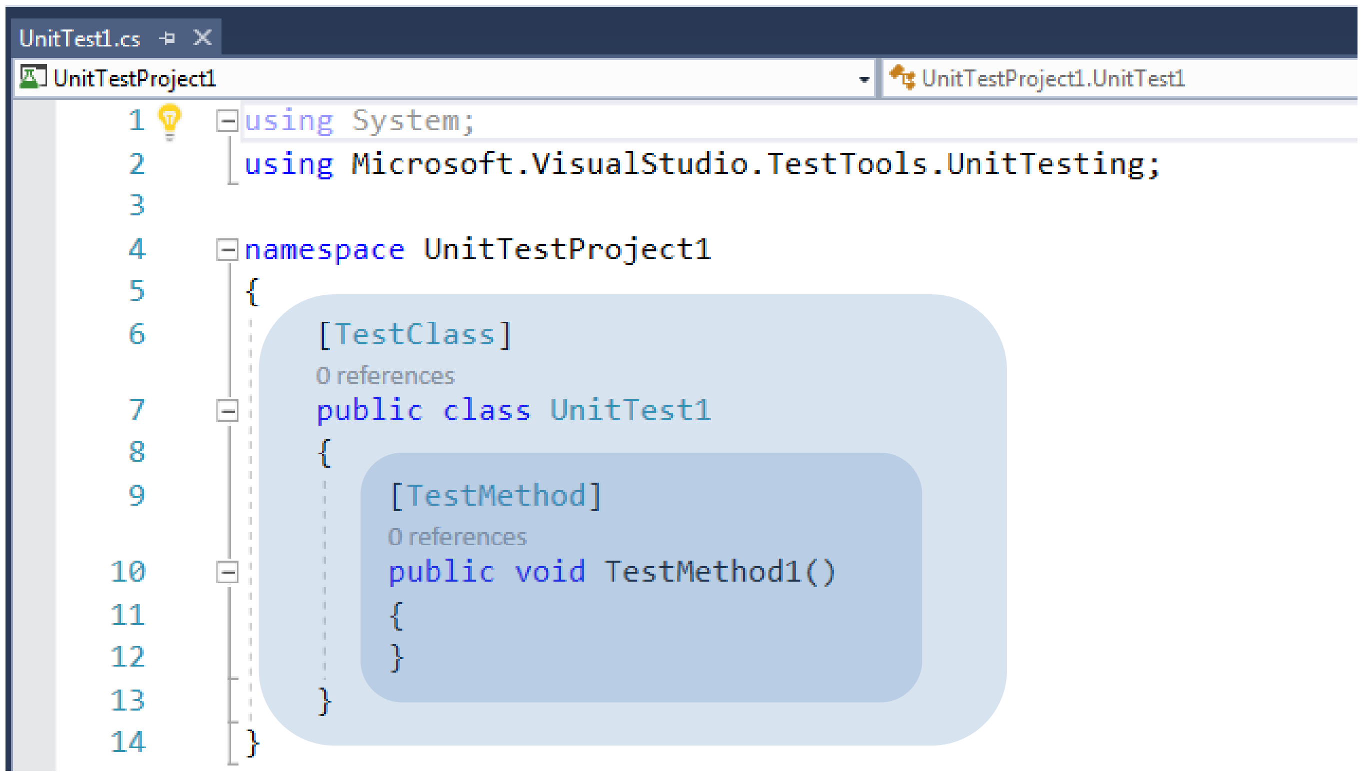
- Open a **Solution**
- Add a new **Unit Test Project**



Unit Testing - Visual Studio

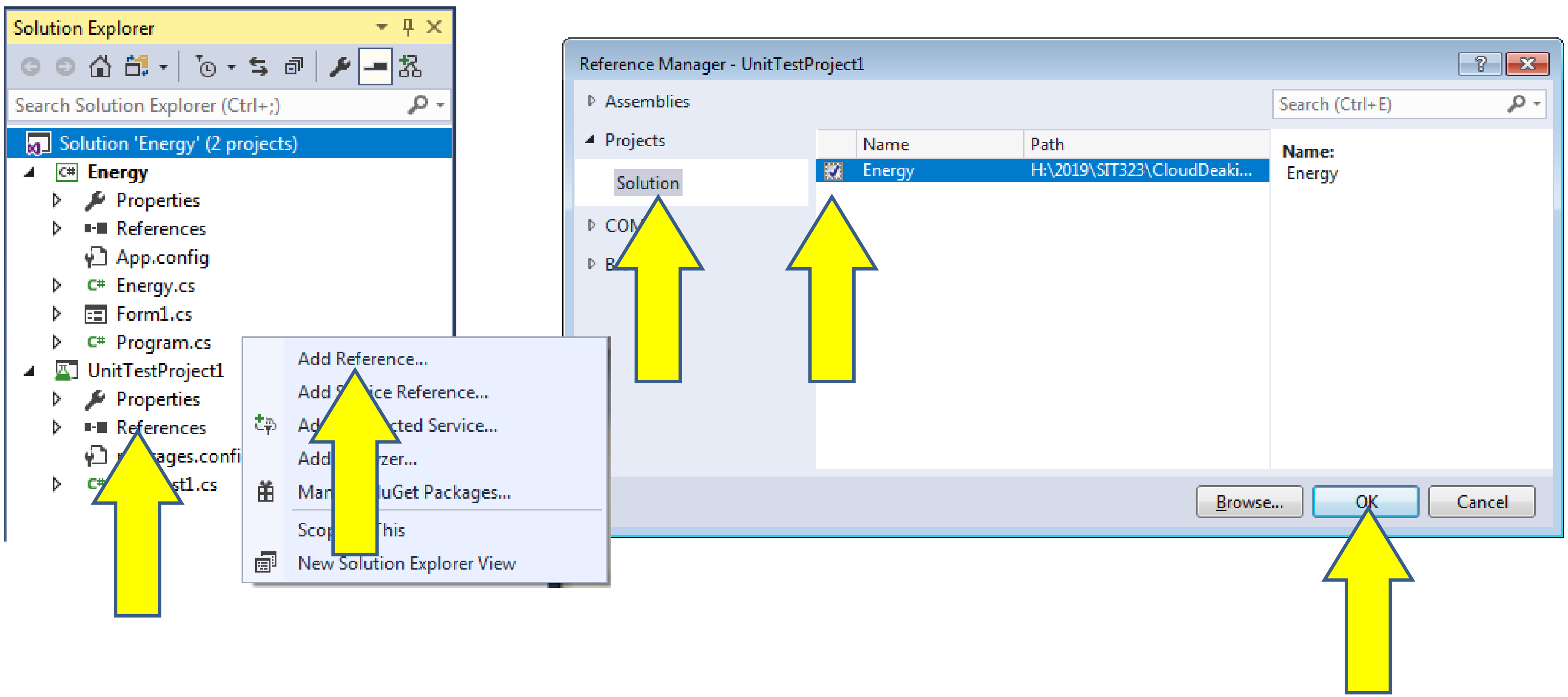
- A **new project** is created, within is a **C# file** containing a **test class** and a **test method**.





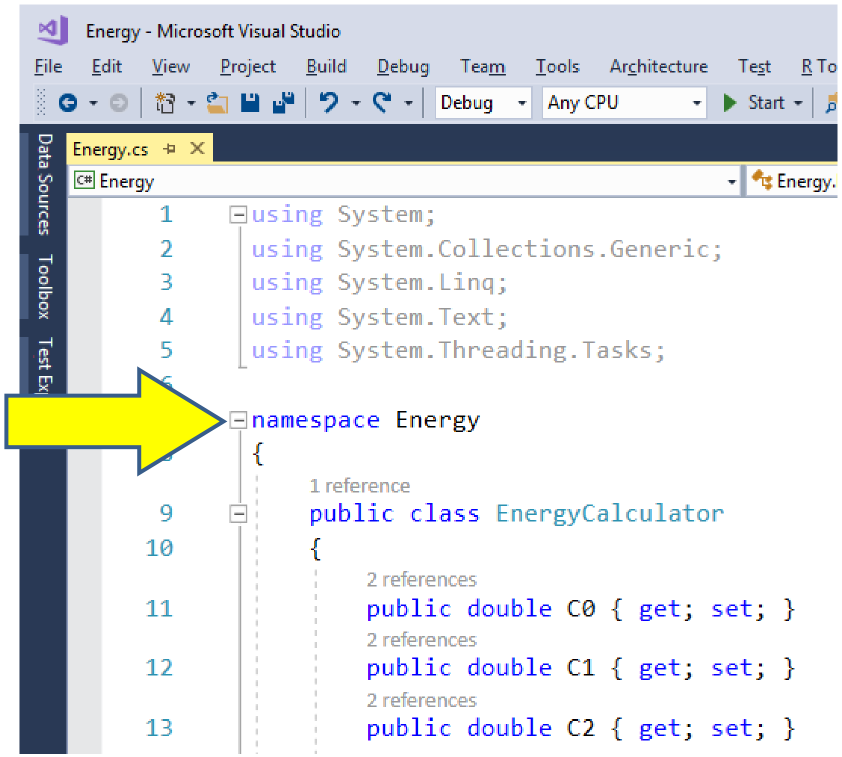
Unit Testing - Visual Studio

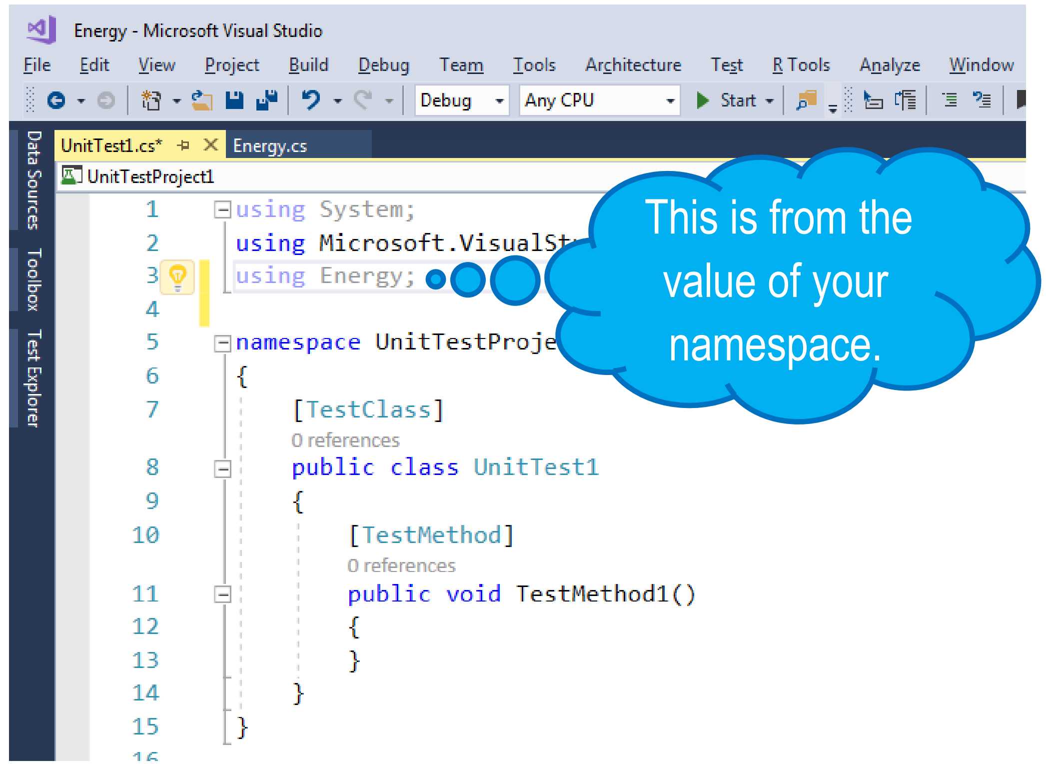
- **<u>Add a reference</u>** from your Unit Test project to the other project in your solution.



Unit Testing - Visual Studio

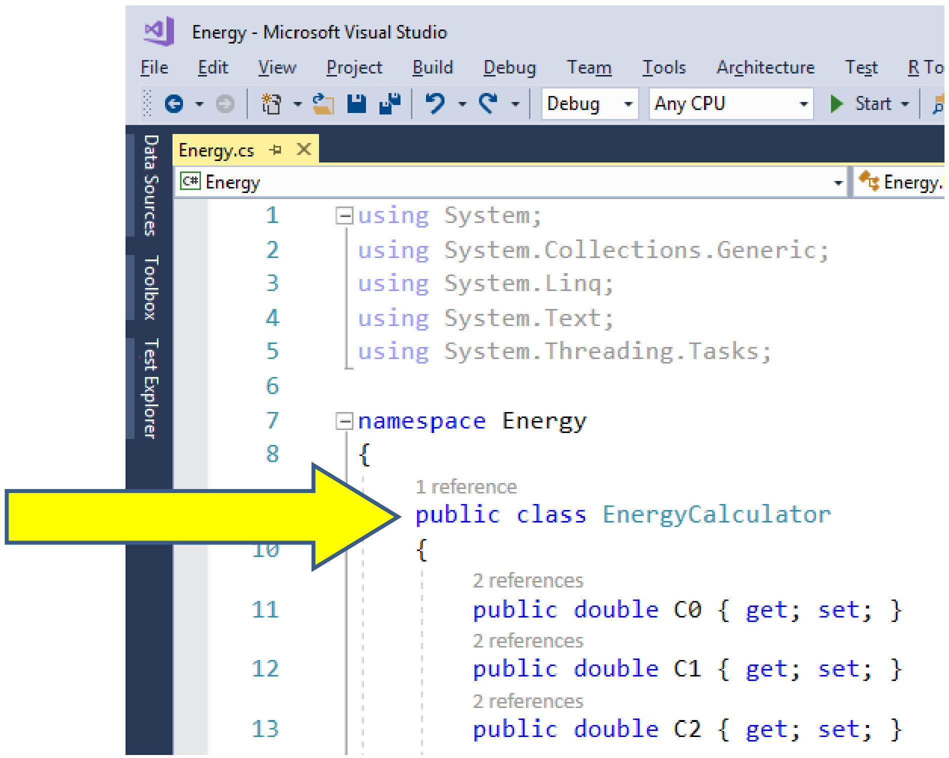
- Add a **using statement** with the appropriate namespace to access appropriate classes.





Unit Testing - Visual Studio

- Ensure the class being tested is **public**.



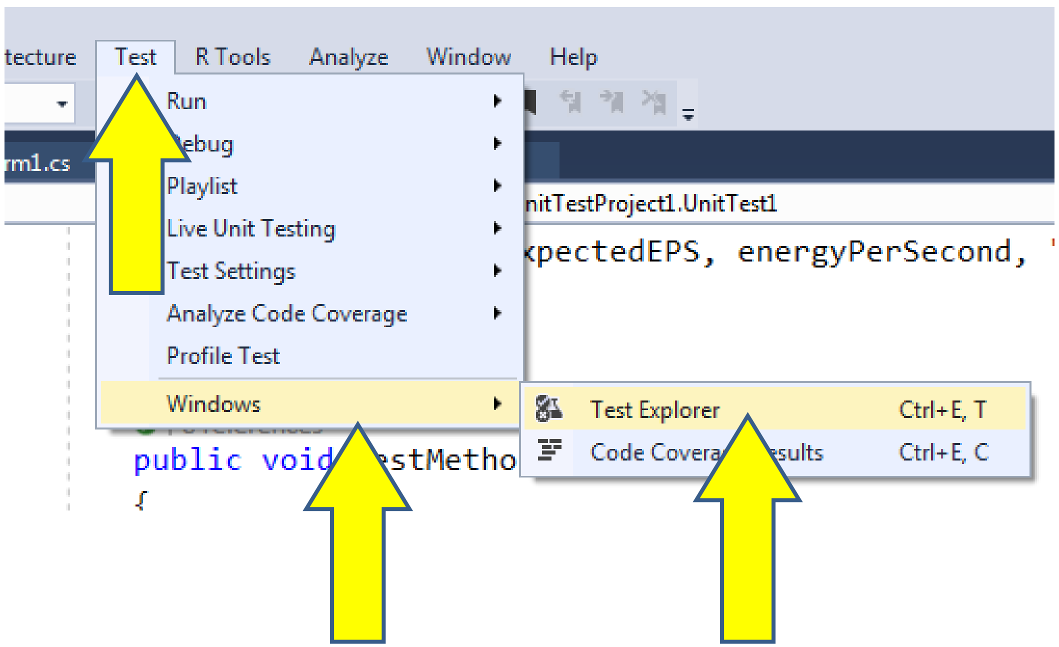
Unit Testing - Visual Studio

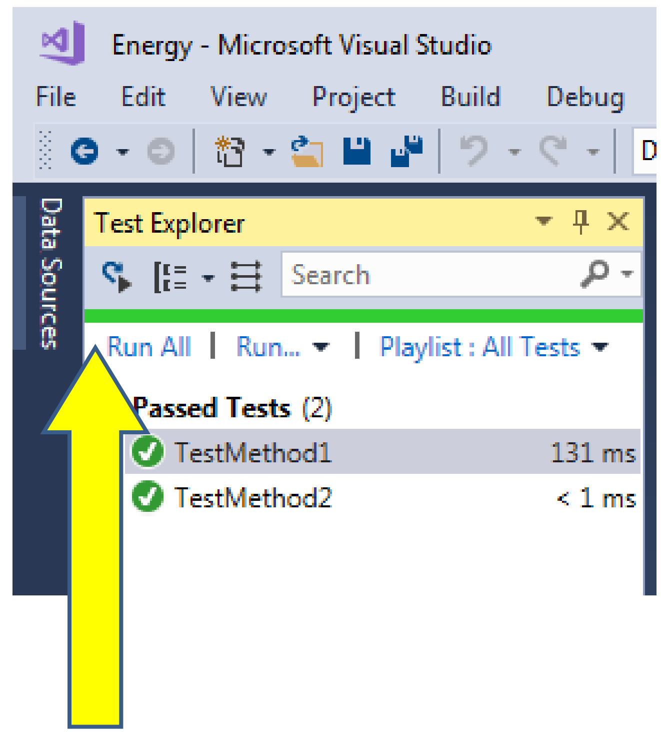
For each test method, **<u>write your test code</u>** using the **AAA** pattern.

1. **Arrange**
   - prepare object/values for the method being tested
   - prepare expected objects/values
2. **Act**
   - call the method being tested
3. **Assert**
   - check that the method worked as expected

Unit Testing - Visual Studio

- Use **<u>Test Explorer</u>** to run your tests, e.g., select **Run All**





Unit Testing - Visual Studio

Some MSDN Reading

- Unit Test Basics
  https://msdn.microsoft.com/en-au/library/hh694602(v=vs.120).aspx
- Creating and Running Unit Tests
  https://msdn.microsoft.com/en-us/library/ms182532(v=vs.120).aspx

- Unit Testing C# Code - Tutorial for Beginners

https://www.youtube.com/watch?v=HYrXogLj7vg

Case Study – Edge Workflow System

X. Liu et al., "FogWorkflowSim: An Automated Simulation Toolkit for Workflow Performance Evaluation in Fog Computing," 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 2019, pp. 1114-1117,
Jia Xu, Ran Ding, Xiao Liu, Xuejun Li, John Grundy, Yun Yang, EdgeWorkflow: One click to test and deploy your workflow applications to the edge, Journal of Systems and Software, Volume 193, 2022,