COMPUTATIONAL THINKING

CSCU9A1: Savi Maharaj

UNIVERSITY OF STIRLING

Thinking before programming

- Computational Thinking is ***not***
  - Learning to think like a computer

- First part of lecture: general problem-solving
- Second part: designing computational solutions using pseudocode

How does one tackle a new problem?

Don’t Panic

Polya: How to Solve it (1945)

- Four principles:

1. First, you have to understand the problem.
2. After understanding, then make a plan
3. Carry out the plan
4. Look back on your work. How could it be better?

Understanding the problem

- What are you asked to find or show?
- Can you restate the problem in your own words?
- Can you draw a diagram to help understand the problem?
- Do you have enough information to find a solution?

Understanding the problem: Example

From the tutorials:

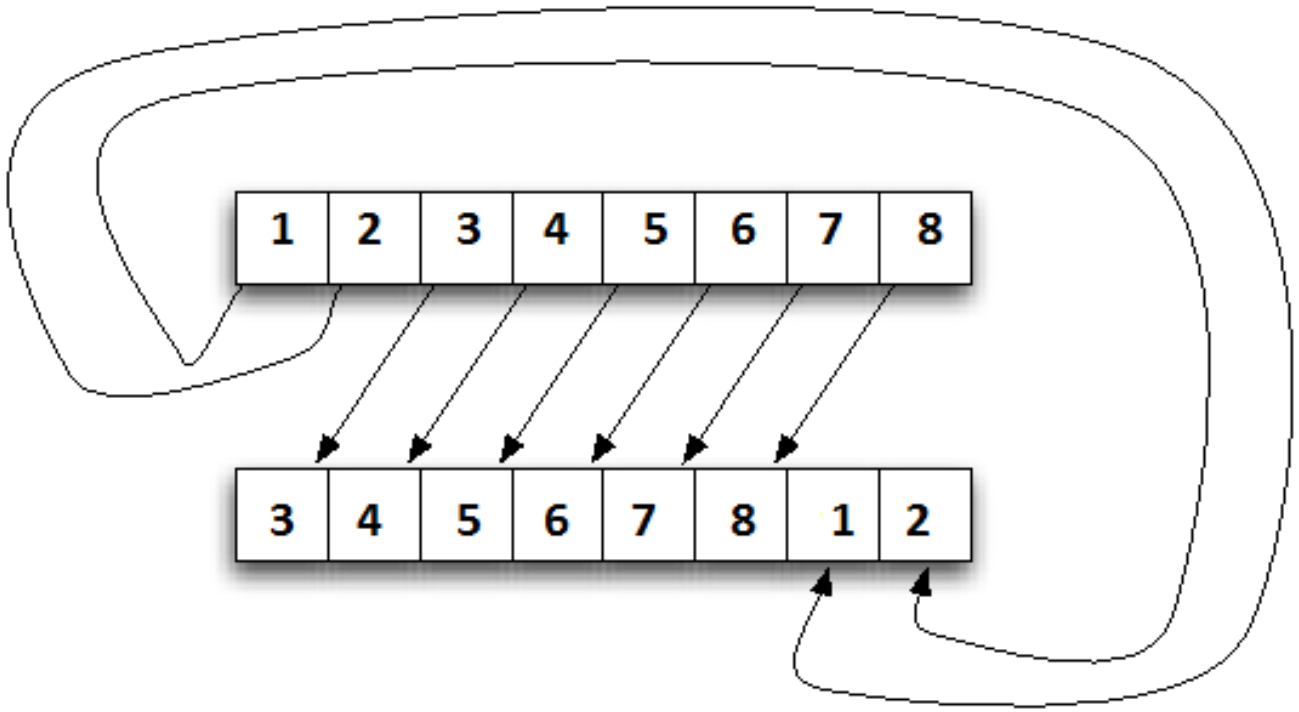
Write a method that is passed an array, *x*, of doubles and an integer rotation amount, *n*. The method creates a new array with the items of *x* moved forward by *n* positions. Elements that are rotated off the array will appear at the end. For example, suppose *x* contains the following items in sequence:

1 2 3 4 5 6 7 8

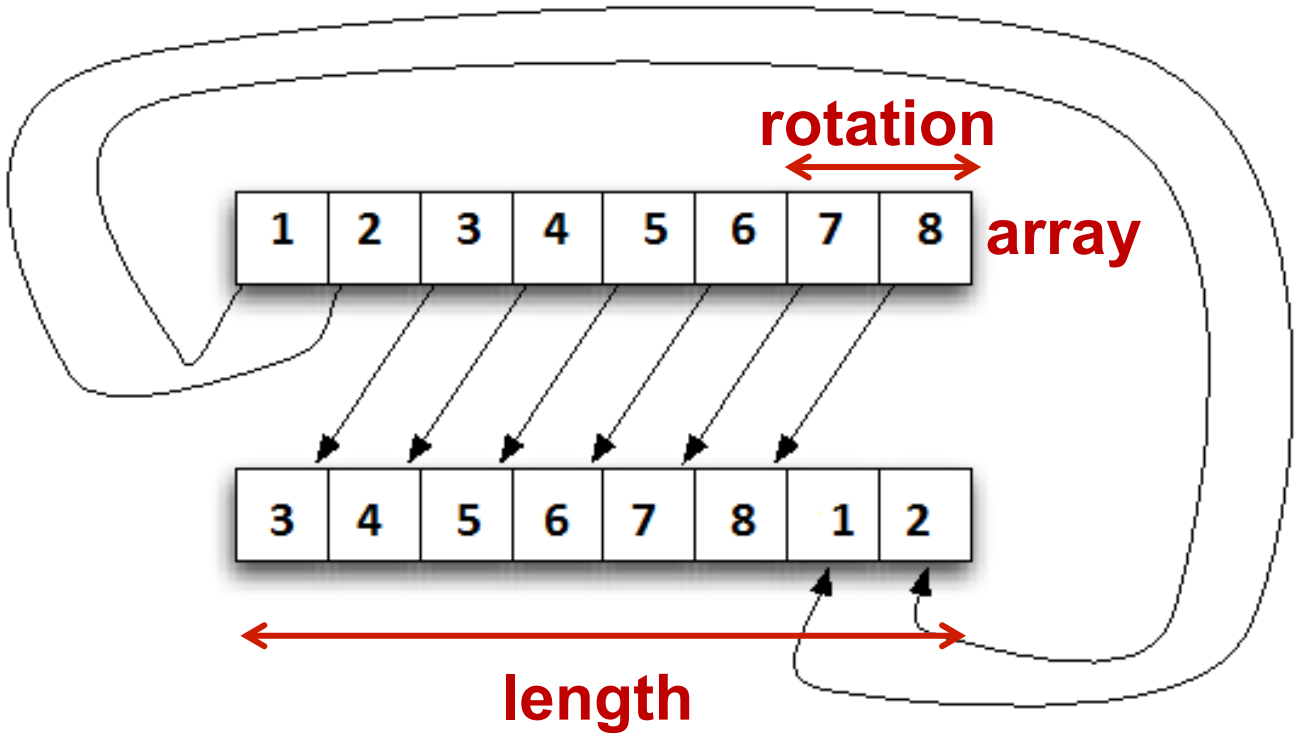
After rotating by 2, the elements in the new array will appear in this sequence:

3 4 5 6 7 8 1 2

Draw a picture



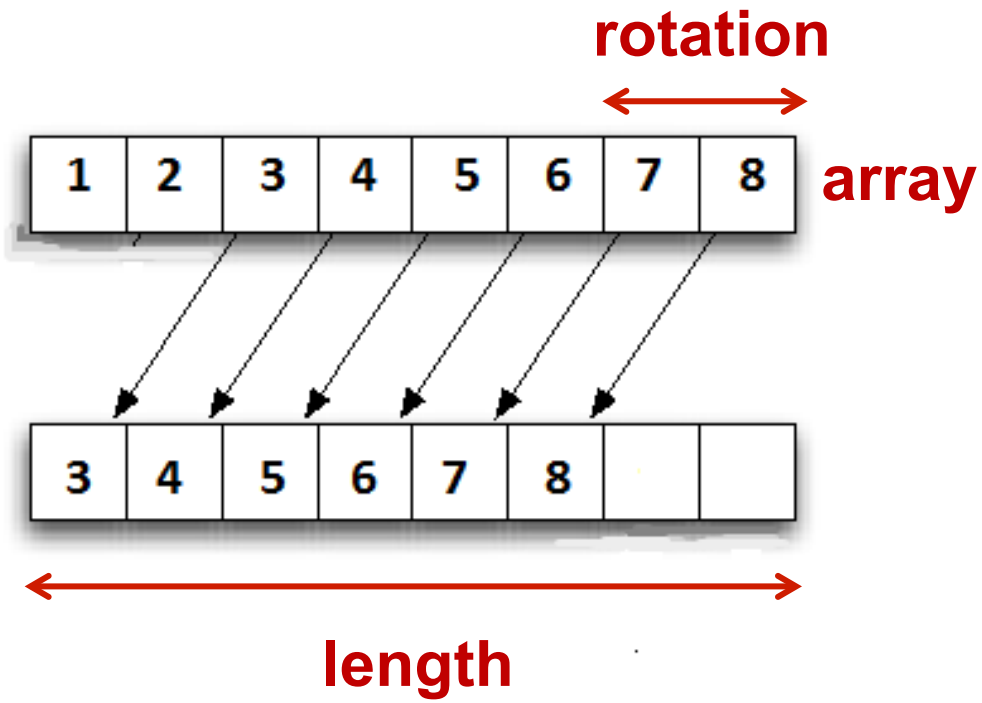
Identify the information you know



Make a plan

- Have you seen a *similar problem* before?
- Can you imagine an *easier related problem*?
- Could you solve a *part* of the problem?
- Do you have *enough information* to find a solution?
  - If not, what extra information do you need?
- Can you *restate the problem* yet again?
- Can you solve a *more general problem*?
- Can you solve a *more specific problem*?

Can you solve an easier problem?



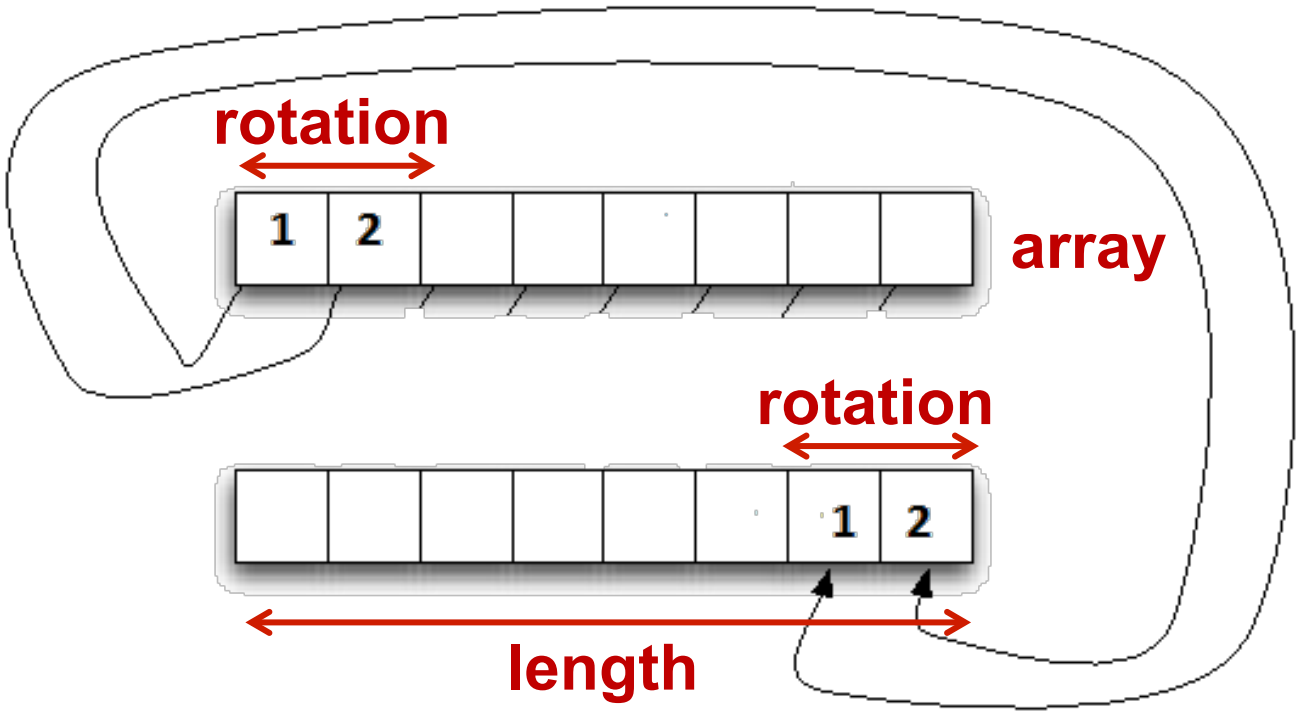
Write down your partial solution

An informal notation such as pseudocode is useful for this. (There is more about pseudocode in the second part of the lecture.)

```
Shift [array, rotation, newarray] ::=
    For index from rotation to length - 1
        Copy array[index] to newarray[index - rotation]
```

Does this solve the easier problem?

Turn your partial solution into a full solution



Write down your full solution

```
Shift [array, rotation, newarray] ::=
    For index from rotation to length - 1
        Copy array[index] to newarray[index - rotation]
    next = length – rotation
    For index from 0 to rotation – 1
        Copy array[index] to newarray[index + next]
```

Carry out your plan

- Implement your solution in Java
- Are you sure that every step is correct?
  - This will require testing your solution thoroughly.
- Do you notice anything that you overlooked before?

Implementing the rotate method

```
public static double[ ] rotate (double[ ] array, int rotation)
   {
      double[ ] result = new double[array.length];
      for (int index = rotation; index < array.length; index++)
      {
         result[index - rotation] = array[index];
      }
      int next = array.length - rotation;
      for (int index = 0; index < rotation; index++)
      {
         result[index + next] = array[index];
      }
      return result;
   }
```

Testing and correcting the solution

- Test the code with typical values. Does it work correctly?
  - Example: apply it to the array {1,2,3,4,5,6,7,8} with rotation 2
- Try less typical values:
  - empty array
  - rotation greater than the length of the array
  - zero rotation
  - negative rotation
- Make any necessary adjustments (exercise for you to do after class)

Revise and extend

- Reflect on your solution thoroughly.
- Could you have done anything better?
- Are there alternative solutions that might work better?

A simpler rotate method

```
public static double[ ] rotate (double[ ] array, int rotation)
   {
      double[ ] result = new double[array.length];
      for (int index = 0; index < array.length; index++)
      {
         result[index] = array[(index + rotation) % array.length];
      }
      return result;
   }
```

Using pseudocode to describe computational solutions

- What are the basic elements of a computer program?

- **Control structures** and **data**
  - Perhaps control structures operating on data

- Control structures define what should be done, and in what order
- Data (structures) define how the data is stored.

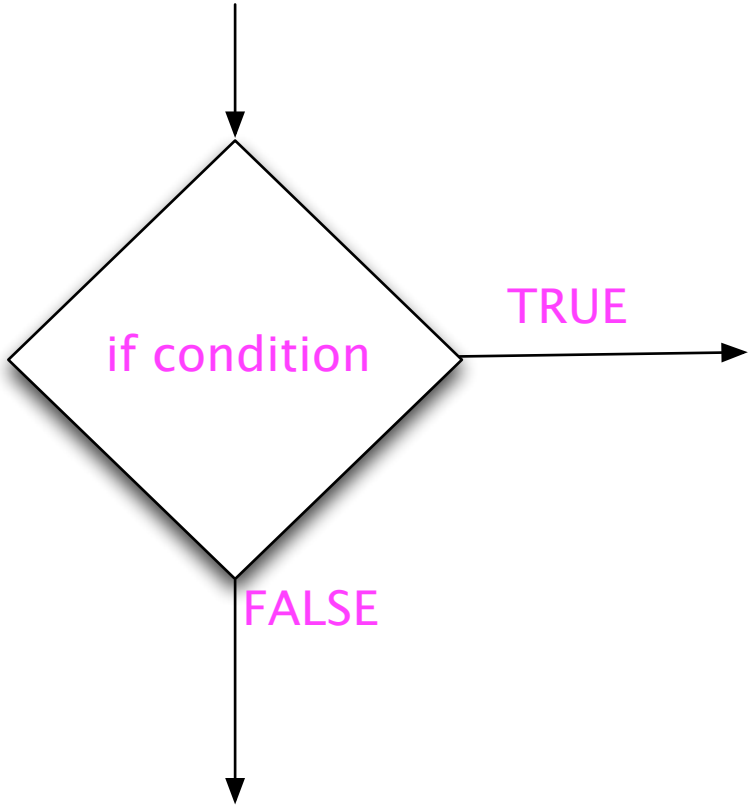
- Both are important: indeed often inter-related

Control structures

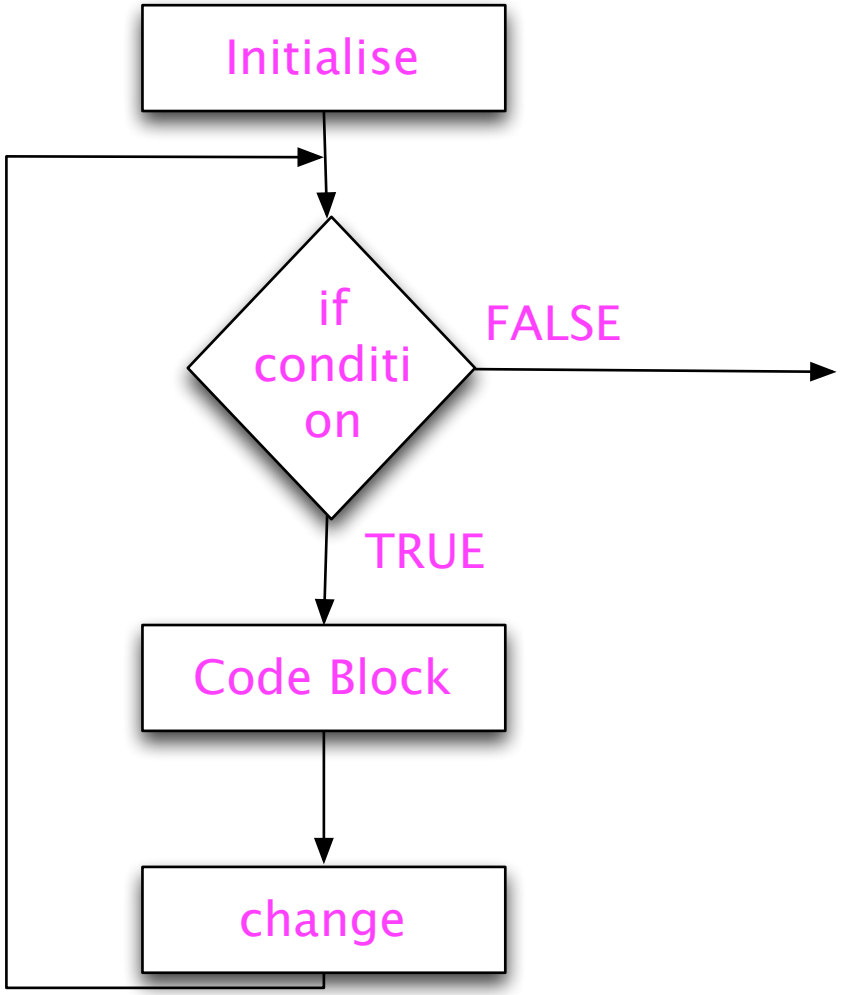
What are the basic control structures?

- Sequencing
  - Exemplified in Java by the semicolon, ;
- Choice
  - Exemplified by if … then … else
- Iteration
  - More than one type:
    - Conditional iteration: repeat some section of code until something changes (while and do-while loops)
    - Enumerated iteration: repeat some section of code a particular number of times (for loop)

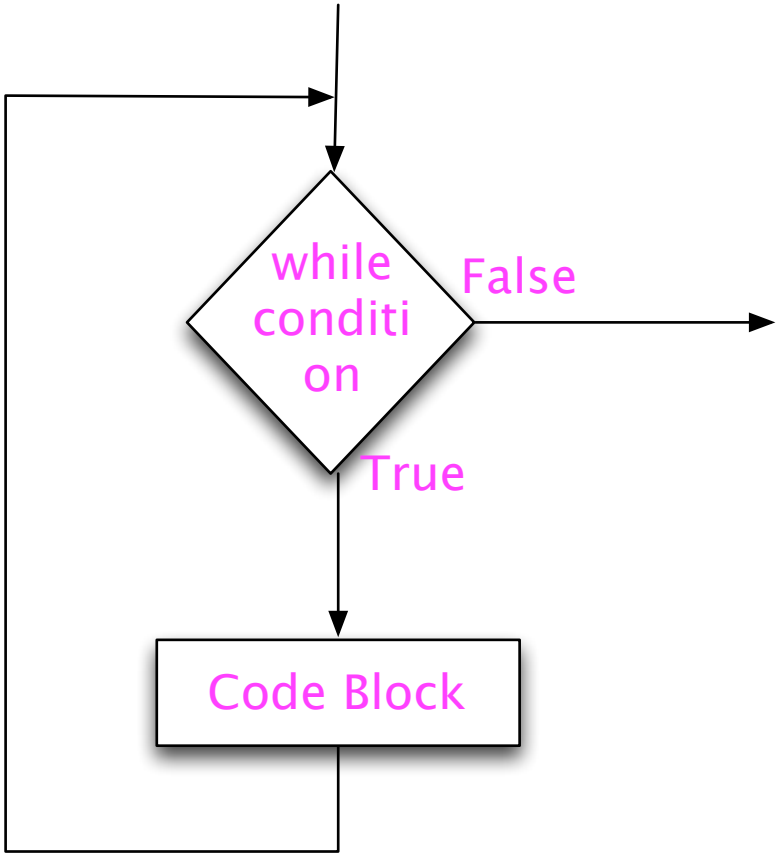
If statement



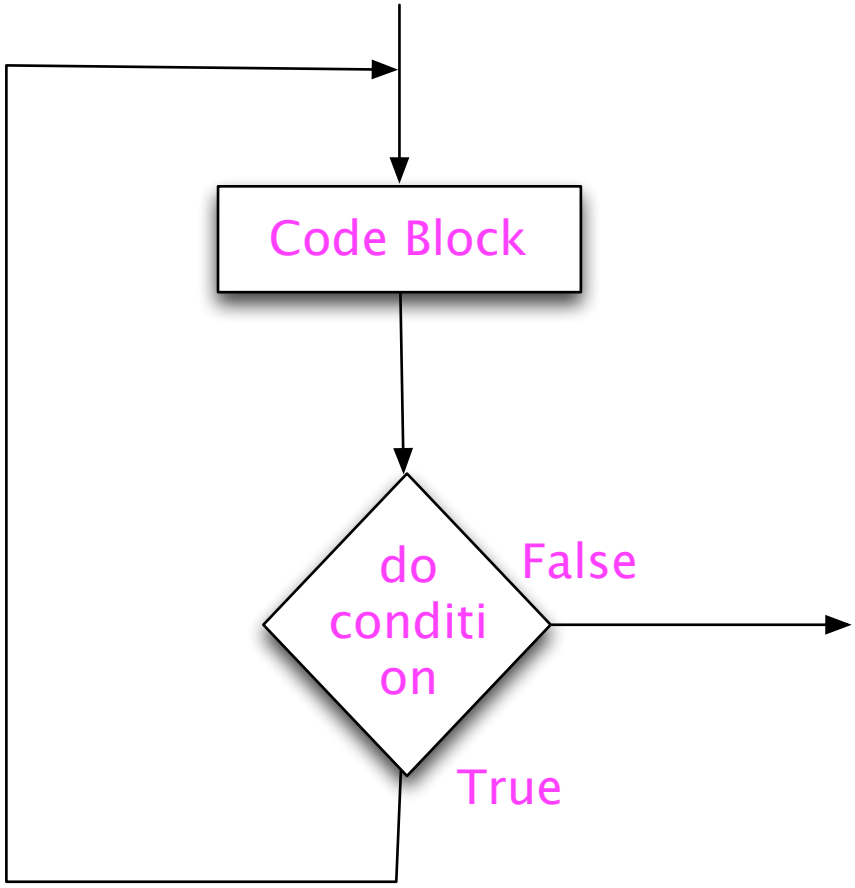
For loop



While loop



Do loop



Issues

- What's the difference between these?
  - Discuss
- How do we ensure that they terminate
  - Do we always want them to terminate?

Abstraction

- Very often we want to parcel up some code
  - Perhaps we want to re-use it often
  - Perhaps we simply want to think about it later

- Abstraction refers to writing a simple statement which will result in the execution of some larger piece of code.
  - In this case, a function, or method

- Abstraction can also be used to parcel up some code and some data structures.
  - This leads us to object-oriented programming, which you will cover in future modules.

Abstraction 2

- Very often we want to parcel up some code *and some data structures*

- Perhaps we want to re-use it again
- Perhaps we are using it to model some entity that the program is describing
  - Perhaps we simply want to think about it later

- Abstraction refers to writing a simple statement which will result in the execution of some larger piece of code.
  - In this case, an object
    - And very probably methods that act on/interrogate/modify that object.

Pseudocode for designing programs

- Pseudocode is an informal notation – there is no standard syntax.
- The statements of pseudocode often resemble program code but may be more abstract and hide details that are to be developed later.
- Pseudocode statements often include:
  - Conditionals: **if** … **then** … **else**
  - Iterations:
    - **While** (condition) **do** (statements) **end while**
    - **Repeat** (statements) **until** (condition) **end repeat**
    - **For** (iterator description) **do** (statements) **end for**

Example: make a pot of tea

```
Make tea::=
    Boil kettle
    Put tea in teapot
    Pour kettle into teapot
        Repeat
            Wait
        Until tea is correct strength
```

Comments?

Example: make a pot of tea

```
Make tea::=
    Boil kettle // Needs expanded
    Put tea in teapot
    Pour kettle into teapot // needs expanded
        Repeat
            Wait
        Until tea is correct strength // and this condition?
```

More comments?

Boil Kettle

Boil Kettle ::=

Put water in kettle

Switch kettle on

Repeat

Wait a short time

Until kettle has boiled

Comments?

Boil Kettle

```
Boil Kettle ::=
    Put water in kettle // this might be a loop
    Switch kettle on
    Repeat
        Wait a short time // busywaiting: inefficient?
    Until kettle has boiled
```

Comments?

Pour kettle into teapot

```
Pour kettle into teapot ::=
    While teapot not full do
        Pour water into teapot
```

Comments?

Making N cups of tea?

```
Make cups of tea (N) ::=
    Make tea
    For i = 1: N
        Pour cup(i)
```

Easy? Possible problems?

Making N cups of tea?

```
Make cups of tea (N) ::=
    Make tea
    For i = 1: N
        Pour cup(i)
```

What if N is big? What if the teapot is a 4 person teapot, but you want to make 10 cups of tea?

- In general, pseudocode is good as an *aunt sally* or *straw man*: something that can be criticised, discussed, and improved on
- AND: there much less momentum: much less has been invested in generating the pseudocode,
  - Making starting again/severe modification much easier!

Events

- Some programs begin, run through some data, and end
- But many programs need to respond to things that happen
  - Inside the program (unexpected conditions, for example)
  - Outside the program (key is pressed, mouse moved; or external sensors indicate a change that needs action)
- Generically, these “things that happen” are called ***events***.
- You will learn more about event-driven programming in semester 2.

Events and boiling a kettle (!)

```
Boil Kettle ::=
    Put water in kettle // this might be a loop
    Switch kettle on
    Repeat
        Wait a short time // busy-waiting: inefficient?
    Until kettle has boiled
```

- Busy-waiting is very inefficient
- We can use events to help

Boil Kettle (again)

```
Boil Kettle ::=
    Put water in kettle
    Set up kettle-has-boiled event
    Switch kettle on
    Await kettle-has-boiled event
```

- The process running Boil Kettle is *suspended* by the await statement
- It continues only after the event occurs
- (Of course, if the event fails to occur, it may wait forever)

Events concluded (for now!)

- Real events may be anything from pressing a button, to a sensor detecting a temperature change, to a mouse movement, to a disk transfer finishing, …
- What do they have in common?
  - They need processed quickly.
- How to do this?
  - Old answer: suspend main stream of processing, start event-handling processing (interrupt processing), altering some data somewhere, restart main stream of processing
  - Apparently, but not really asynchronous
  - Apparently, but not really parallel

Events concluded: modern systems.

- More recent machines have
  - Many cores per processor
  - Multiple processors as well
- Event handling remains conceptually the same
  - But is often really is asynchronous and parallel
  - And this can lead to issues
    - Parallel updating of data structures, for example.

Concluding

- Computational thinking is an attempt to consider what sort of thinking underlies the ability to implement …
  - Concepts, applications, ideas etc.

    … on computer systems.
- It is
  - part general problem solving (which certainly not unique to computing, but equally certainly needs thinking)
  - and part applied problem solving (aimed at automating the eventual implementation)