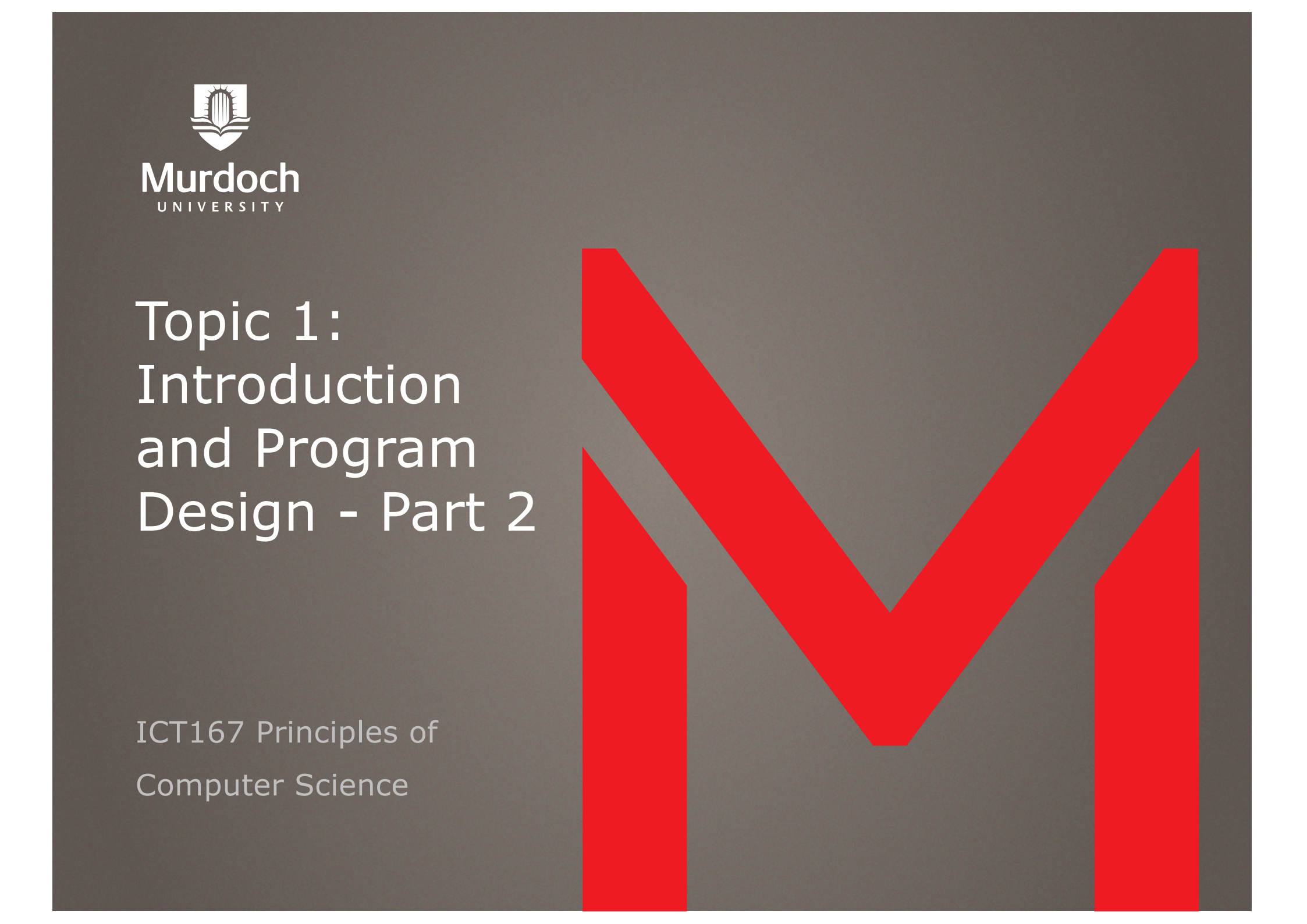


Topic 1: Introduction and Program Design - Part 2

ICT167 Principles of Computer Science

Murdoch UNIVERSITY

Learning Objectives

- Understand the need for various sorts of documentation to accompany the delivery of code
- Know basic types of internal and external documentation
- Understand the need for use of and documentation of a systematic test strategy and its results
- Explain the terms design and design methodology in software development

Learning Objectives

- Define the terms algorithm, pseudocode, sequence, selection and iteration
- Be able to give acceptable pseudocode versions of simple algorithms
- Use Java constructs for controlling flow
- Explain concepts of high-level pseudocode, procedural abstraction and data abstraction
- Understand the basics of using procedures including return types, arguments, formal parameters

Learning Objectives

- Give a brief description of structured programming including top-down refinement and the use of call graphs;
- Be able to use simple procedural abstraction in good designs

Reading

Savitch: Chapters 2, 3, 4

Recommended self test questions:

Chapters 2.4, 3.1, 3.2, 4.1

Programmers Responsibility

- Whether working alone or as part of a team, the main concerns of a programmer are that at the end of their work they need to deliver:
  - Some code which works according to the expectations of their customer, or client (who could be an end user, a big organisation or other software developers)
  - Some documentation which, along with the code, helps make the code easy to install, use, understand (for modification, maintenance and extension), and gives the client confidence that the product works

Programmers Responsibility

- A programmer may also have to document the progress of their work so that:
  - They and their manager (or even client) can get an idea of how things are going and predict finish dates
  - They don't keep re-trying dead-ends
  - They and their manager can learn from the overall process
  - Someone else can take over if the programmer gets run over by a bus

ICT167 Required Documentation

- In later units, you will learn quite formal ways of presenting all this documentation
- In ICT167 we require:
  - Internal documentation (i.e. in the program)

    AND
  - External documentation (i.e. in separate text)

ICT167 Required Documentation

- For internal documentation we require:
  - A beginning comment clearly stating title, author, date, file name, purpose and any assumptions or conditions on the form of input and expected output
  - Other comments giving useful low-level documentation and describing each component
  - Well-formatted readable code with meaningful identifier names and blank lines between components (like functions, modules, methods and classes)

ICT167 Required Documentation

- In using Java it is also good practice to make use of Javadoc
- For practice work, you should follow these requirements
- For assignments, you should follow these requirements and submit source code as well as external documentation...

ICT167 Required Documentation

- Required External Documentation
  - **Title:** a paragraph clearly stating title, author, date, file name, and one-line statement of purpose
  - **Requirements/Specification**: a paragraph giving a more detailed account of what the program is supposed to do. State any assumptions or conditions on the form of input and expected output
  - **User Guide**: instructions on how to compile, run and use the program

ICT167 Required Documentation

- Required External Documentation continued..
  - **Structure/Design**: Outline the design of your program. Give a written description, use diagrams and use pseudocode
  - **Testing**: Describe your testing strategy (the more systematic, the better) and any errors noticed. Give copy of results of testing
  - **Limitations:** Describe program shortfalls (if any), eg, the features asked for but not implemented
  - **Listings**: Attach source code listings (source program text)

Design

- This can mean either:
  - The task of thinking up a good overall approach to the program (one of the phases of the software development cycle)
  - The actual overall arrangement of the program
  - The description of overall arrangement of the program as given in the documentation

Design

- A **Design Methodology** is a systematic approach to design, and supports good design by helping with the design task and possibly with the description of it

Design

- A design methodology may:
  - Suggest a tried and tested general way of coming up with a good design
  - Allow a programmer to re-use other designs by allowing easier understanding and communication of designs
  - (or may not) be supported by a programming language
- We will look at the OO design methodology later

Algorithms and Design

- During most of the 20th century, programmers did not have to be very sophisticated designers
- The most important design task was to come up with an algorithm to solve a problem
- Recall, an **algorithm** is a very precise, complete set of instructions to solve a problem
- The instructions may be expressed in a natural language (like Chinese or English) or a programming language (like Java or C) or a mixture

Algorithms and Design

- In ICT167 we will use structured English or **pseudo-code** (a mixture of English and Java)
  - You can mix English and Java as you find convenient
- It is important to remember that your pseudo-code is supposed to convey to the reader, in an easily readable way, that you really have produced **a completely precise algorithm** which needs no further clever work to put into code

Algorithms and Design

- Although you might think that coding is hard, you will hopefully one day join the many people who can easily translate good precise pseudo-code into one of several programming languages
- The hard creative part is coming up with the algorithm in the first place
- So format your pseudo-code nicely (i.e. indent) and don’t use words like "it" unless it is very clear what "it" is

Algorithms and Control Flow

- Because algorithms should be complete instructions, their basic steps should consist in **small indivisible steps** like "input a character", add x to y and store the result as z, etc.

Algorithms and Control Flow

- You can then build up a bigger algorithm by putting together steps via:
  - **sequence:** one step after another
  - **selection:** choosing what to do on the basis of a simple test
  - **iteration:** keep doing the same thing over and over until some test holds
- Virtually every programming language allows easy expression of these ways of implementing an algorithm

Using Pseudo-code

- Today, only the most basic of programs directly represent low-level algorithms
- It is much more common for us to use pseudo-code in a more *high-level* way
- Still we use sequencing, selection and iteration but the nature of the individual steps is different

Using Pseudo-code

- Here is some pseudo-code to keep getting an input line from the user and display the first and last character until the first one is ‘q’:

```
firstchar = ‘x’
while ( firstchar != ‘q’)
    prompt user for input
    s = next input line
    firstchar = first character of s
    lastchar = last character of s
    display “first character =” firstchar
    display “last character =” lastchar
```

indicate that program is finished

Using Pseudo-code

- Before we think about why this is high-level, let us look at the Java version
- Note that there are extra bits and pieces in the Java code to make it more user friendly and set up the input, but the underlying algorithm is the same

Example

```
//Pseudo.java
//Displays the first and last characters of lines of
//text from standard input. Use 'q' to quit program.

//must import Scanner class
import java.util.Scanner;

public class Pseudo
{
   public static void main( String[] args)
   {
      Scanner input = new Scanner(System.in);
      char firstchar = 'x';
      char lastchar = 'x';
      System.out.println("Start a line with q to quit.");
```

Example

```
while (firstchar != 'q'){
   System.out.println("Enter a line:");
   String s = input.nextLine();

   //gets 1st character of s
   firstchar = firstCharOf(s);
   //gets last character of s
   lastchar = lastCharOf(s);

   System.out.println("1st character is "+firstchar);
   System.out.println("Last character is "+lastchar);
   System.out.println();
   System.out.println("Next.");
} //end of while
System.out.println("You quit.");
} //end of main
```

Example

```
   //Note:  this program is incomplete -
   //       some things are missing here
} //end of class Pseudo
```

High Level Pseudo-code

- The pseudo-code and Java code above counts as high-level for two reasons:
  1. The basic steps are not the single indivisible steps of machine code
     - They are sometimes words which summarize quite a complex operation
     - That is what we mean by high-level, i.e. ***using simple names to stand for something more complex***
  2. In programming design this technique is called **Abstraction**

High Level Pseudo-code

- Abstraction = dealing with the essential features of something while ignoring the details
- It allows us to give an overall description of some procedure (like that in Pseudo.java) without getting bogged down in details
- Abstraction of a tool:
  - What it is used for and how to use it, **not** how it works and what parts it consists of
- Note: procedure here is a generic term
  - When translated into Java, the procedure is called a method (and a function in C)

High Level Pseudo-code

- In software development, abstraction is used in two ways:
  1. **Procedural (functional) abstraction:** we have procedural abstraction when we have a simple name for a more or less complicated procedure. Eg: `firstchar` is the first character in `s` (the procedure finds the first character in a string)
  2. **Data abstraction:** we have data abstraction when we have a simple name for a more or less complicated piece of data, like the value of a variable. Eg: `s` is a whole string of characters

Using Procedural Abstraction

- (We will look at data abstraction later)
- Coding at low-level is very boring and time consuming
- Productivity (and convenience and ease of design and ease of understanding and maintenance etc) is increased greatly when the programming language supports abstraction
- Much larger and more complex programs become feasible

Using Procedural Abstraction

- Early high-level languages supported procedural abstraction by allowing named functions, procedures, subroutines or methods
- The idea was perhaps most developed in the traditional teaching language Pascal
- Even modern OO languages like Java continue to do so (in a very slightly different way)

Using Procedural Abstraction

- The idea of using a procedure in design (and then in coding) is to capture a common and well-defined task
- It might be a task which is undertaken many times at many places in your program (or even in other programs too) or it might be a task which you want to think about separately

A Typical Procedure

- You must give some code (or pseudo-code) to show how the procedure works – that is, the procedure *definition* and its body
- You will also, separately, have the main program which *calls* the procedure (using its name)
- Eg: here's a pseudocode definition ...

```
Procedure char lastCharOf(String str)
  …pseudocode to work out last char of str…
```

A Typical Procedure

- Decide what input information (if any) the procedure needs to do its job
  - The procedure may in general get information from the program via parameters (or via global/instance variables)
- Decide what output information (if any) the procedure returns to the main program or another procedure
  - Information may come back via a return value, via changes to arguments or via changes to global/instance variables

A Typical Procedure

- On the other hand a procedure may just do something (like output something to the screen) and not return any values
- After that you need to try and work out the code for the body of the procedure – that is, the code that actually does the work

Return Types

- In many languages (as in Java) you can choose whether a procedure (called a **method** in O-O languages) has a return value or not
- If it does have a return value, then you will often see calls to it like

```
chr = firstCharOf( str);
```

so the return value of the procedure `firstCharOf()` is assigned to the variable `chr`

Or, calls like

```
print( firstCharOf( str ));
```

Return Types

- So the return value of `firstCharOf()` is given to the procedure `print()`
- In typed languages like Java (which require you to declare the type of variables) you will need to declare the return type of a procedure in its definition and make sure it makes sense where it is called
- In some languages procedures with return values are called **functions**

Return Types

- In the definition of a procedure with a return value you will need to indicate exactly which value to return to the main program or another procedure. For example, in Java you write:

```
return 2;
```

  to immediately send the value 2 back from the procedure
- Use "return *value*" also in pseudo-code. Make sure that the type of the value matches the declared return type of the procedure

Example

- Here is the possible pseudo-code for the whole of the procedure `lastCharOf()`

```
Procedure char lastCharOf( String str )
  n = lengthOf(str)
  lco = CharAt( str, n-1 )
  return lco
```

- Again this is high-level
  - However, I happen to know that it uses procedures which are (practically) built-in to Java
  - They find the length of a String and find the character at a given index location in a String

Example

- As in many languages, Java indexes String (and Array) locations starting from 0
  - So the first character is at index 0, etc.
- If you want to know how these procedures themselves are defined in terms of even more basic steps then you need to start thinking about the way Strings are implemented
  - But that is Data Abstraction and it is looking below Java anyway, so we won't

Example

- Note there is no official syntax (way of writing) procedures in pseudocode, but do try to be clear about names, types, etc.

Procedures With No Return Types

- If the procedure has no return value then it can only be called via a whole statement like:

```
print('x');
```

or

```
print( firstCharOf( str ) );
```

i.e. it does **not** make sense to write

```
a = print('x');
```

- In Java these types of procedures are treated in a similar way to procedures with a return value. We just declare the return type to be **void** if we have no return value

Procedures With No Return Types

- You can put a **return** statement in the definition of the procedure to indicate where control passes back to the main program
- Remember that code might (or might not) be useless after a return statement:

```
void proCC( int  x) {
   if (x != 3) {
        x = 4;
   }
   else
        return;
```

Procedures With No Return Types


```
    printOut(x);
    return;
    printOut(x-1);
} //end of proCC
```

- Sometimes, you do not have to put a **return**
  - The procedure also returns when its end is reached

Formal Parameters and Arguments

- Information is also passed in and out of a procedure via its arguments
- You will need to specify the number of, types of and exact ordering of arguments
- You do this in the procedure definition by using some **formal parameters**, i.e. variables standing for the arguments (actual parameters) which you can then use in the body of the procedure

Formal Parameters and Arguments

- When you call the procedure from the main program (or another procedure) you need to supply an exactly matching set of arguments
- Eg: a definition

```
void addAndDisplay(String s, int x, int y){
   int z = x+y;
   System.out.println( s + " : " + z);
}
```

and a call ...

```
val = 7- 3;
addAndDisplay("Answer is", val, 3 );
```

Formal Parameters and Arguments

- Note that if a procedure has no arguments we still write eg: **name()** to define and call it

Parameter Passing

- The neatest, most standard, and commonly understood way of using parameters in pseudo-code or program code is:
  - Have either no output from a procedure or at most one output value which is the return value
  - Use parameters for input only
  - Do not change the value of the parameters inside the procedure
- Please stick to these rules in all pseudo-code

Parameter Passing

- In many programming languages, including Java, these rules can be broken and we will break them later
- However, what happens then depends very much on the detailed rules of the language
- Pseudo-code has no such tricky subtle rules
  Eg: what does the following mean?

```
//bad pseudocode
void swap(int x, int y){
  int t=x;  x=y;  y=t;
}
```

Parameter Passing

- The previous pseudo-code would be called by:

```
a=4;
b=3;
swap(a,b);
swap(a,6);
```

- This is bad because it is not unambiguous and so not precise

Parameters In O-O Languages

- Another problem arises with O-O languages
- Sometimes, you will see what looks like an argument in front of a procedure call. Eg:

```
char x = str.charAt(2);
```

  - (find the 3rd char in the String `str` and assign to `x`)
- Isn't the String `str` an argument to the function `charAt()`?
- In pseudocode, we write:

```
x = CharAt( str, 2)
```

Parameters In O-O Languages

- The nitty-gritty of parameter passing is tricky in a real programming language so we will come back to that later
- The use of parameters makes the design clear and the procedure re-usable

Parameters In O-O Languages

- It is useful to be able to look at the top line of the definition of a procedure and know that you are seeing all the information about what inputs and outputs there are
- You can copy the procedure and use it in another application

Structured Design

- The idea of procedural abstraction and, more importantly, its support in powerful languages like C and Pascal, allowed much more complex programs to be designed and implemented well
- Abstraction allows people to cope with complexity
- In fact, a whole design methodology called **Structured Design** based on procedural abstraction became popular

Structured Design

- The basic idea is to design and implement by **top-down refinement**: break the problem up into a series of steps involving the major sub-tasks and then implement each sub-task as a procedure in the same way
- Eventually you will just have procedures which can be directly implemented in basic statements in the programming language
- The methodology even came with its own diagrams: **call graphs**

Structured Design

- These are graphs showing which procedures call which procedures
  - They give a good overall picture of the design of a structured program
- We can and do program like this in Java
- It is easy to use procedures (i.e. methods) to break down a task into simpler tasks
- However, there is quite a bit more as well to O-O design as we will see

Example Problem

Procedural Abstraction in Java

- Consider the following problem:
  - Loop around, getting a floating point number from the user and keep a running total of the input numbers
  - Each time display the latest number and the running total, both correct to two decimal places
  - Stop when the user enters a number outside the range -100 to 100

Example Solution: Pseudo-code

- Here's some pseudo-code for the problem:

```
total = 0
flag = true
while (flag)
   get an input number d from the user
   if outOfRange(d) then set flag to false
   else
      dispTwoDPs(d)
      total = total + d
      dispTwoDps(total)
   end else
end while
```

Example Solution: Pseudo-code

- Note the use of procedures above to display a number correct to 2 decimal places and test whether a number is out of range or not

Example Solution: Java Code

```
//TwoDPs.java
//Displays running total of numbers in lines of
   standard
//input correct to two decimal places.
//Uses an out of range number (<-100 or >100) to quit.

import java.util.Scanner;
public class TwoDPs {
  public static void main( String[] args) {
     Scanner input = new Scanner(System.in);
     double total=0;
     boolean flag=true;
```

Example Solution: Java Code

```
System.out.println("Use an out of range entry
                    < -100 or > 100 to quit.");
while (flag){
   System.out.println("Enter a number on a
                                    line:");
   double d = input.nextDouble();
   if (outOfRange(d)){
      flag=false;
   }
   else
   {
```

Example Solution: Java Code

```
            dispTwoDPs("The number value is", d);
            total = total + d;
            dispTwoDPs("The total is", total);
            System.out.println();
            System.out.println("Next.");
         }//end of else
      }//end of while
      System.out.println("You quit.");
   }//end of main
/* Note: put the outOfRange(…) and
   dispTwoDPs(…) method definitions here */
}//end of class
```

Example Solution: Pseudo-code

Out of Range procedure

- Here is some possible pseudocode:

```
procedure boolean outOfRange( double d )
if (d < -100) return true
if (d > 100) return true
return false
```

Example Solution: Java Code

- Here is some Java code:

```
static boolean outOfRange( double d ){
  if (d<-100) return true;
  if (d>100) return true;
  return false;
}
```

- And here is another possibility in Java:

```
static boolean outOfRange( double d ){
  return ((d < -100) || (d > 100));
}
```

Example Solution: Pseudo-code

Display to two decimal places

- The algorithm is a bit tricky
- Here is some very high-level pseudocode:

```
procedure DispTwoDPs( double d )
    record whether d is negative in neg
    make a positive version of d, store in posNum
    add 0.005 to posNum, store in nPlus
    record the whole number part of d in whole
```

Example Solution: Pseudo-code

let `rest` be `whole - nPlus`
let `temp` be the whole number part of `rest` to 2
    decimal places by `100 * rest + 100`
let `ss` be the last two characters of the string
    version of temp
display the sign of the original d,
    followed by the `whole` number part of d,
    followed by a decimal point
    followed by `ss`
return

Example Solution: Pseudo-code

- We could be even more specific with the pseudo-code but it would then look very much like the following Java code
- EXERCISE: to see where this 0.005 and 100 come in, try some examples
- Eg, try 0.861, 0.866, 0.995, -0.1, 0.05, 4, 45.0

Example Solution: Java Code

```
static void dispTwoDPs(String msg, double
  num){
// Display on screen the message msg followed by num
// correct to two decimal places with both decimal
// values showing even if they are zero
  //record whether the number is negative
  boolean neg = ( num < 0 );
  //make a positive version of the number
  double posNum = num;
  if (neg) posNum = -num;
```

Example Solution: Java Code

```
//add 0.005 to the posNum, so that
//truncating nPlus is equivalent to
//rounding posNum
double nPlus = posNum + 0.005;
//extract whole number part and the rest
int whole = (int) nPlus;
double rest = nPlus - whole;
//multiply the rest by 100
//truncate, cast and make sure there
//are some zeros in front of small numbers
int temp = (int )( 100.0 * rest + 100.0 );
```

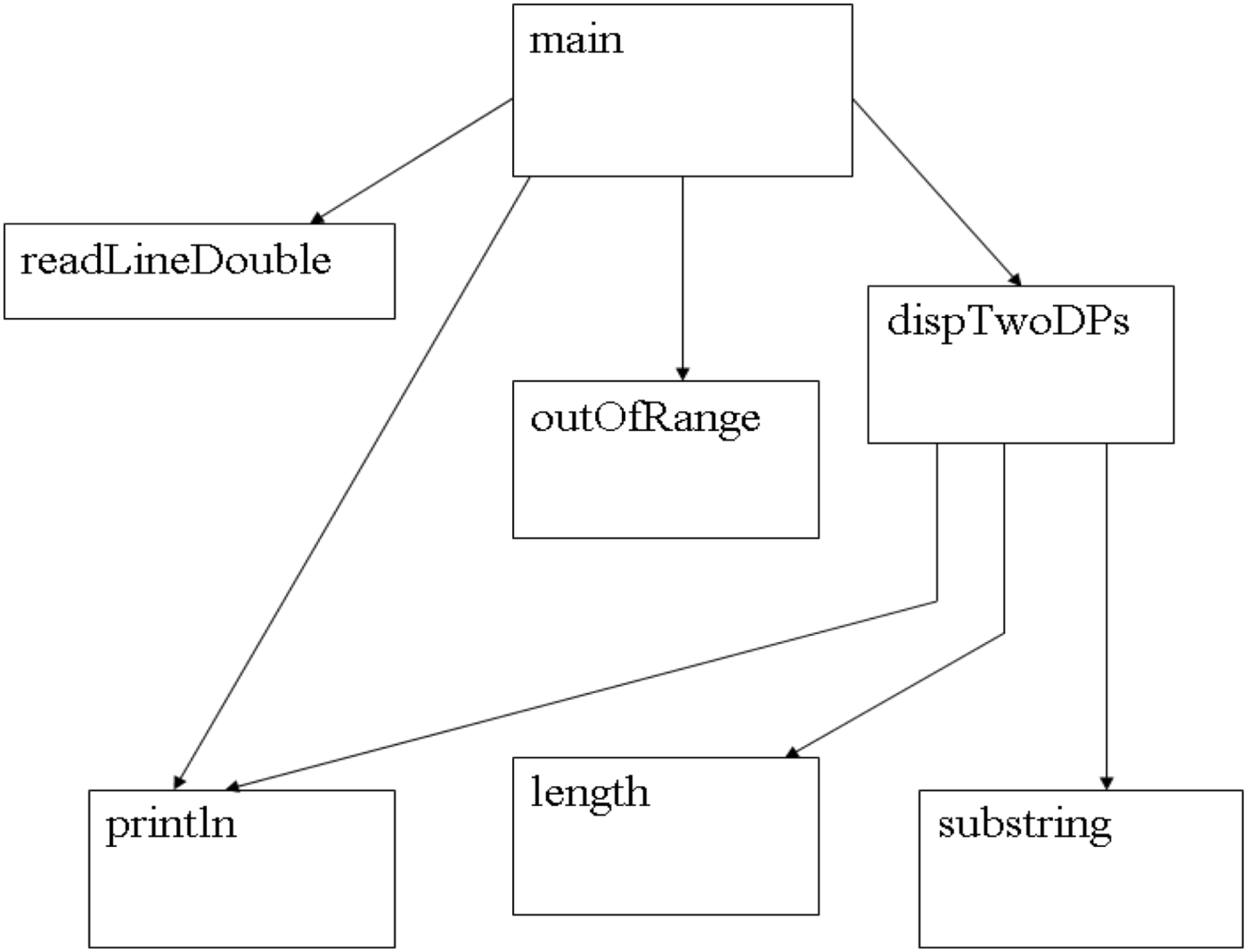
Example Solution: Java Code

```
//make a string version of temp
String ss = "" + temp;
int len = ss.length();
String sign ="";
if (neg) sign="-";
//display the message, sign, whole part
//and last two digits of ss
System.out.println(msg + " " + sign +
   whole + "." + ss.substring(len-2,len)
);
}//end of DispTwoDPs
```

Example Solution: Java Code

```
//Note: ss.substring(i,j) is a library
//procedure which finds the substring of
//string ss starting at index i and
//finishing at index j-1
```

A Call Graph



End of Topic 1 – Part 2

Murdoch UNIVERSITY