Chapter 41 - Passing parameters using the stack

41.1 The use of the stack to call procedures and functions and pass parameters

A common use of the stack is when a program calls a function or procedure. When a function or procedure is called, it may also need to be given some parameters, or data, to work with. This is known as '**parameter passing**'. Consider this example.

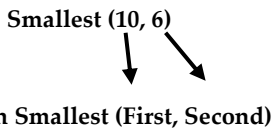
```
Begin
    Write ('Please enter the value of x')
    Read (x)
    Write ('Please enter the value of y')
    Read (y)
    .
    .
    c:= Smallest (x, y)          \\ The variable c is assigned to the result of calling the function Smallest (x, y).
    Print c
    .
    End

    Function Smallest (First, Second : integer) : integer      \\This function expects to be passed two integers
                                                               when it is called. It will return one integer value.
    Begin
      If (First < Second) then
         Smallest = First
      Else
        Smallest = Second
      EndIf
      Return Smallest
End
```

41.1.1 How does this program work?

This program asks a user to enter in the value of x. The user enters a number and this is assigned to the variable x. The program then asks the user to enter in the value of y. The user enters a number and this is assigned to the variable y.
Then the program comes across the instruction **c:= Smallest (x, y)** This means that the variable c is assigned to whatever value is returned when the function **Smallest (x, y)** is called. Once c has been assigned a value, it is printed out.

Suppose when the program is run that the user enters 10 for the first number and 6 for the second number. That means that x:= 10 and y:= 6. This now means that c is assigned the value of **Smallest (10, 6)**. Now the function Smallest is called. It is passed the parameters 10 and 6. 10 and 6 are known as the **actual parameters**. The first number, 10, is given to the first variable in the function and the second number, 6, is given to the second variable in the function, like this:



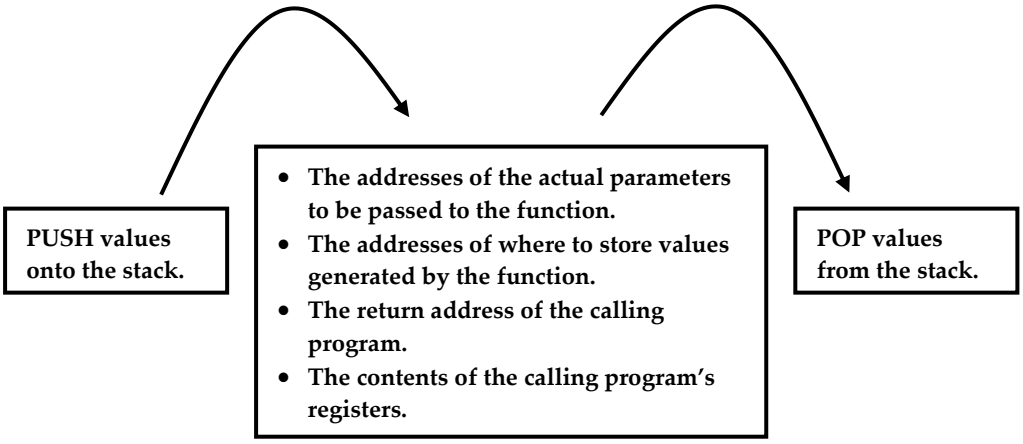
First and second are known as the **formal parameters**. Notice how the parameters 10 and 6 are passed from the call in the program, Smallest (10, 6), to the actual function Smallest (First, Second) **in the same order**. Now the function Smallest can be run using First:= 10 and Second:= 6 When the Function is run, we can see that the result is that the ***variable*** Smallest is assigned the value held in the variable Second (because Second is less than First, or 6 is less than 10). The variable Smallest is therefore assigned the value of 6. Remember that with functions, there is a variable that holds the result of the function that **has the same name** as the actual function itself!! The value **6** is therefore passed back to the main program and replaces the call, Smallest (10, 6). To summarise:

- We had **c:= Smallest (x, y)** to start with in the main program.
- This became **c:= Smallest (10, 6)** once the two values were entered by the user.
- This resulted in **c:= 6** once the function Smallest(10, 6) had been called and run.

41.1.2 How is the stack used in this program?

The steps in using the stack to call a function and to pass parameters to it can be summarised as follows:

1) The contents of the registers in the CPU are pushed onto the stack. The CPU needs to save the contents of these registers so that when it returns to the original program after the call to the function has finished, it can be set up in exactly the same way as before the call happened.
2) The return address is then pushed onto the stack. The return address is the address of the instruction in the main calling program that must be done once the function has finished. In this case, the return address would be the address for the instruction **Print c**.
3) Next, we need to push the arguments we want the function to use onto the stack. We can do this in one of two ways, either by using 'call by reference' or by using 'call by value'. With call by reference, we push onto the stack **references** to the variables that hold the data we want to pass to the function. If the function then changes these variables, these changes are reflected in the variables in the main program. If we use call by value, we push onto the stack **copies** of variables of the data we want to pass to the function. If the function then changes the data in these variables, it only changes the copy inside the function! It doesn't change the original data held in the variables outside of the function.
4) The address of the first instruction of the function is then placed in the Program Counter. When the next instruction is fetched, it will be the start of the function.
5) The function gets the actual parameters that are being passed to it by popping them off the stack and assigning them to its formal parameters. The function than continues until completion.
6) When the function has finished, the value that needs to be passed back to the calling program is stored by pushing it onto the stack.
7) The return address, register values and the value generated by the function are then popped from the stack and program control returns to the main program.



How the stack is used to call a function, pass it parameters and return a value.

Q1. What is a parameter?
Q2. What is an argument?
Q3. What is meant by the phrase 'passing parameters'?
Q4. What is a stack?
Q5. What does 'push' and 'pop' mean when talking about stacks?
Q6. What register holds the address of the next instruction that must be carried out by the CPU?
Q7. Research the Internet. What is meant by 'call by reference'?
Q8. Research the Internet. What is meant by 'call by value'?
Q9. Why does a return address have to be pushed onto the stack when a function is called?
Q10. What is meant by a subroutine?

Chapter 42 - Searching and sorting files

42.1 A serial search

A serial search involves starting at the beginning of a file and checking each record in turn. You would need to check if the first record is the one you are looking for. If it is, then you can stop searching and report that you have found the record. If it isn't the one you are looking for then you go on to the next record. You repeat this process until either you find the record you want or you come to the end of the file. If you do come to the end of the file without finding the record you want then you simply report that the record you are looking for isn't in the file and stop searching! It doesn't actually matter if the records are in a particular order or not when you sort through them when you do a serial search.

42.2 A binary search

This is a different approach to finding a particular record. The idea is that you divide all the records into two, a top half and a bottom half. You then test to see which half the record you want is in. Whichever half it isn't in, you discard! So you are now left with only **half** the original records. You then split that half into two and repeat the process, until you eventually find the record you want. This is explained in more detail with an example later in this chapter. **Binary searching will not work unless the records are all in order!** Of course, you could take a **serial file** (one whose records are not in any particular order) and sort them (so the serial file becomes a **sequential file**) and then perform a binary search on that file!

42.3 Advantages and disadvantages of serial searching compared to a binary search

There are some advantages and disadvantages to both approaches.

- Compared to a binary search, serial searching is a very easy system to write a program for (start at the beginning and compare each record in turn until you find it).
- Serial searching isn't very efficient. That's because it takes lots of comparisons to find a particular record in big files, compared to binary searching, so serial searching takes longer.
- If you only have a very few records in a file then it makes little difference which method you use!
- Serial searching is excellent if your records are unordered. In binary searching, you have to write code that puts records in order before the binary search can be done, or you won't be able to use binary searching at all. This will slow down processing and increase the complexity of the code, compared to serial searching.

42.4 Binary searching

Binary searching is a programming method used to search through data to find a particular item or record.

- The data must be in order for binary searching to work.
- It takes fewer comparisons than serial searching to find an item.
- Binary search algorithms are more complicated to write and understand than serial search algorithms.

42.4.1 Searching for a record using binary searching

To illustrate how binary searching works, we will look at an example. How would you search for record number 3 within these records using the binary search method? Remember, the records **must** be in order. If they are not, then they must be put in order first using some code. In the example below, the records are already in a sequential order (using the records' key values).

	Index	Course name	Points
Low →	1	Computing	20
	2	Maths	18
	3	Physics	12
	4	Media	14
Middle →	5	Multimedia	17
	6	Sociology	17
	7	Chemical Engineering	20
	8	Production management	18
	9	History of Art	24
High →	10	Teaching	16

A file of records that we wish to search through.

STEP 1. We set a flag to indicate the first and last records. We'll call these Low and High.
STEP 2. We then work out the Middle position. We can do this by calculating **Trunc (Low + High) / 2**
In our example, (1 + 10) / 2 = 5.5. Truncated, 5.5 equals 5, so our middle record is 5.
STEP 3. We first of all test to see if Middle equals the value we are looking for. If it does, we got lucky! We can stop searching and report that the record has been found.
STEP 4. If we didn't get lucky, we then ask ourselves, is the record we are looking for (in our case, record 3), higher or lower than the record pointed to by Middle?

- If the record we want is above Middle, then we can discard all the records from Middle and below. We would then have about a half of the original number of records to search through.
- If the record we are looking for is below Middle, then we can discard all the records from Middle and above. We would then have about a half of the original number of records to search through.

In our case, record 3 is less than record 5. We can therefore discard all the records from Middle up to High. We are now left with records 1 to 4 to search through. We then set the new Low value to point to the first record. We then set the new High value to point to the last record as before, and work out the new Middle (Trunc (1 + 4) / 2 = 2).

	Index	Course name	Points
Low →	1	Computing	20
Middle →	2	Maths	18
	3	Physics	12
High →	4	Media	14

At this stage, we are now in exactly the same position as we were at the start of the problem. The only difference is that we have about half the records to search through after just one comparison! We can now call our binary search function again (recursion) and perform the exact same steps we have just done. The remaining logic for finding record 3 goes like this:

- Record 3 does not equal Middle (record 2).
- Record 3 is greater than Middle (record 2).
- Discard all records from Middle and below.

We are now left with the following records:

	Index	Course name	Points
Low →	3	Physics	12
High →	4	Media	14

- Middle now equals Trunc (3 + 4) / 2 = 3.
- Record 3 equals Middle. Stop and report found.

42.4.2 Searching for a record using binary searching – another example

Suppose we want to search for record 10 in our previous example.

	Index	Course name	Points
Low →	1	Computing	20
	2	Maths	18
	3	Physics	12
	4	Media	14
Middle →	5	Multimedia	17
	6	Sociology	17
	7	Chemical Engineering	20
	8	Production management	18
	9	History of Art	24
High →	10	Teaching	16

A file of records that we wish to search through.

The logic goes like this:

- Record 10 is not equal to middle.
- Record 10 is greater than middle. Therefore, discard all records from Middle and below.
- Reset Low, High and Middle.

We are now left with the following situation:

	Index	Course name	Points
Low →	6	Sociology	17
	7	Chemical Engineering	20
Middle →	8	Production management	18
	9	History of Art	24
High →	10	Teaching	16

- Record 10 is not equal to Middle.
- Record 10 is greater than Middle. Therefore, discard all records from Middle and below.
- Reset Low, High and Middle.

We are now left with this situation:

	Index	Course name	Points
Low →	9	History of Art	24
High →	10	Teaching	16

- Middle now equals Trunc (9 + 10) / 2 = 9.
- Record 10 is not equal to Middle.
- Record 10 is greater than Middle. Therefore, discard all records from Middle and below.

	Index	Course name	Points
Low, High, Middle →	10	Teaching	16

- Reset Low, High and Middle.
- Record 10 is equal to Middle. Stop and report found.

42.4.3 Searching for a record using binary searching – another example

Suppose we want to search for record 20 in our previous example.

	Index	Course name	Points
Low →	1	Computing	20
	2	Maths	18
	3	Physics	12
	4	Media	14
Middle →	5	Multimedia	17
	6	Sociology	17
	7	Chemical Engineering	20
	8	Production management	18
	9	History of Art	24
High →	10	Teaching	16

A file of records that we wish to search through.

The logic goes like this:

- Record 20 is not equal to middle.
- Record 20 is greater than middle. Therefore, discard all records from Middle and below.
- Reset Low, High and Middle.

	Index	Course name	Points
Low →	6	Sociology	17
	7	Chemical Engineering	20
Middle →	8	Production management	18
	9	History of Art	24
High →	10	Teaching	16

- Record 20 is not equal to Middle.
- Record 20 is greater than Middle. Therefore, discard all records from Middle and below.
- Reset Low, High and Middle.

	Index	Course name	Points
Low →	9	History of Art	24
High →	10	Teaching	16

- Middle now equals Trunc (9 + 10) / 2 = 9.
- Record 20 is not equal to Middle.
- Record 20 is greater than Middle. Therefore, discard all records from Middle and below.

	Index	Course name	Points
Low, High, Middle →	10	Teaching	16

- Reset Low, High and Middle.
- Record 10 is not equal to Middle.
- Stop and report record not found.

42.5 Sorting and merging data

Taking some data such as pupil records from a school and organising them into some kind of order is called 'sorting' the data. You might sort data numerically in ascending order of the primary key, e.g. 2, 6, 8, 12, 14, 17 and so on. You might sort them in descending order, for example 17, 14, 12, 8, 6, 2. You could use a surname to sort the file, in ascending order: Adams, Best, Cartright, Drucker, Egbert and so on, or in a descending order: Egbert, Drucker, Cartright, Best, Adams. The choice of how to organise the data is up to the user and depends on what they want to do with it. The programmer is left with the job of deciding what programming technique to use to sort the data. There are a number of methods that could be chosen.

42.6 How does the programmer decide which programming approach to take?

Each sorting method has its own advantages and disadvantages. Whichever method is chosen will decide how long it takes to sort the data and from the user's point of view how long it will take to access sorted data. To select the right method of sorting, the programmer has to ask a number of questions:

- How much data is there?
- Can all the data fit into the primary memory at the same time?
- What kind of device is the data stored on? Tape? The hard drive? CD?

When the programmer knows the answers to these questions, they will be able to select the right method. They might choose an **Insertion sort**, a **Bubble sort**, or a **Quick sort, for example**.

42.7 Insertion sort algorithm

This is a straightforward algorithm to implement.

- You take a list of unsorted numbers that you want in ascending order. Unsorted, they start off in position one, position two, position three and so on.
- Start with the second number in the list (in position two) and compare it to the number in position one.
- If the number in position two is smaller than the number in position one, insert it in front of the number in position one. Otherwise leave the numbers where they are.
- Then go to the number in the next position, position three. Find the correct position for the number in position three by first comparing it to the number in position one, and then the number in position two. Insert the number into the correct position if necessary, or leave it in its current position if it does not require moving.

- Then check the number in the next position, position four. Find the correct place for it by checking it against the numbers in the lower positions in turn; position one first, then position two and finally position three.
- Continue until all the numbers in the list have been inserted into the correct positions.

42.7.1 An example of insertion sort

Suppose you want to sort these numbers in ascending order: 12, 5, 34, 33, 2, 90, 26

Step	Numbers	Comment
Step 1	12, 5, 34, 33, 2, 90, 26	Take a list of unsorted numbers.
Step 2	**5, 12,** 34, 33, 2, 90, 26	5 compared with the first position. 5 inserted before 12.
Step 3	**5, 12, 34,** 33, 2, 90, 26	34 compared with lower positions in turn. No change needed.
Step 4	**5, 12, 33, 34,** 2, 90, 26	33 compared with lower positions in turn. 33 inserted before 34.
Step 5	**2, 5, 12, 33, 34,** 90, 26	2 compared with lower positions in turn. 2 inserted before 5.
Step 6	**2, 5, 12, 33, 34, 90,** 26	90 compared with lower positions in turn. No change needed.
Step 7	**2, 5, 12, 26, 33, 34, 90**	26 compared with lower positions in turn. 26 inserted before 33.

42.8 Bubble sort algorithm

This is another method that can be used for sorting data items although it is slow if there are too many data items compared to alternative methods.

Suppose we have the following data that needs to be sorted:

P(1)	P(2)	P(3)	P(4)	P(5)	P(6)
21	3	65	13	83	2

The P(x) denotes what position the data is in and we also need an indicator (called a **flag**) to tell us whether the series is fully sorted or not. The FLAG is data type Boolean. We also need a temporary variable called TEMP. You'll see why in a minute.

We start by setting the flag to FALSE. Then we compare the data in position 1 with the data in position 2. If the data in the **first position** is greater than the data in the **second position** then the data items are swapped and the FLAG is set to true:

```
IF P(1) > P(2) THEN
BEGIN
        TEMP := P(1)
        P(1) := P(2)
        P(2) := TEMP
        FLAG := TRUE
END
```

The temporary value was used to allow you to swap numbers. If you didn't have it, you would overwrite a piece of data before you had swapped it! If at any time a swap does occur, the FLAG must be set to TRUE. If a swap is not needed, then the FLAG is left alone. It is not made TRUE or FALSE.

Once we have checked P(1) against P(2), we then go onto the next pair and repeat. We check P(2) against (P3). If the data in the **first position** P(2) is greater than the data in the **second position** (P3) then the data items are swapped and the FLAG is set to true, as before. Then we check P(3) against P(4), P(4) against P(5) and finally P(5) against P(6). Once we have checked the last pair, we examine the FLAG. If it is TRUE, we reset the FLAG and repeat the whole process. If it is FALSE, we stop because the data is now in a sorted state.

Let's see the whole process in action.

PASS	P(1)	P(2)	P(3)	P(4)	P(5)	P(6)	FLAG status at end of pass
	21	3	65	13	83	2	FALSE
1st	3	21	13	65	2	83	TRUE
2nd	3	13	21	2	65	83	TRUE
3rd	3	13	2	21	65	83	TRUE
4th	3	2	13	21	65	83	TRUE
5th	2	3	13	21	65	83	TRUE
6th	2	3	13	21	65	83	FALSE

A bubble sort.

At the end of the first pass, a swap had happened so the FLAG was set to TRUE. We therefore reset the flag and repeated the whole process. At the end of 2nd pass, a swap had happened so the FLAG was set to TRUE. We therefore reset the flag and repeated the whole process. We continued this until the end of the 6th pass. When we examined the flag, it was found to be FALSE. That meant that no swap had occurred, which meant the data was now sorted.

If you examine the data item 2 for a minute in the table of passes above. You can see that this number moved from position 6 towards the left, to position 1. It floated left, like a bubble. The higher numbers such as 65 floated to the right. The idea of lower numbers floating left and higher numbers floating right, like bubbles, is where the name **Bubble sort** came from!

We can generalise the pseudo-code we have been using for the bubble sort in the following way:

REPEAT

Reset FLAG.
Compare all pairs of numbers in turn, swapping if required.
IF a swap occurrs then set the FLAG.

UNTIL FLAG is FALSE.

42.9 Introduction

A very quick way to sort a set of data is to use the quicksort algorithm. The algorithm performs the following functions on a set of data:

- It selects one of the pieces of data in the set of data to be sorted, called the 'PIVOT'.
- It makes a pass through all the data items. At the end of the pass, three things will have happened:
  - The PIVOT will be positioned correctly.
  - Data items less than the PIVOT will be on the left of it.
  - Data items greater than the PIVOT will be on the right of it.
- The quicksort algorithm is then applied recursively to the left of the PIVOT.
- The quicksort algorithm is then applied recursively to the right of the PIVOT.

42.9.1 An example of the quicksort algorithm

One slightly confusing aspect of quicksort is that sometimes we need to look at the actual data and sometimes we need to look at pointers that point to the position of a data item. It is very important to know whether you are looking at the position of a piece of data (given by the pointer) or the actual data at the position pointed to by the pointer!!

The best way to understand how quicksort works is to do an example. Imagine you want to sort these nine numbers:

1	2	3	4	5	6	7	8	9
42	70	20	40	51	10	61	79	30

42.9.2 What variables and pointers do we need?

We are going to need two variables and two pointers.

- We will need a variable to hold the first value, called FIRST.
- We will need a pointer that points to the position of data as we move through the data from left to right. We will call this pointer UP.
- We will need a pointer that points to the position of data as we move from right to left through the data. We will call this pointer DOWN.
- We will need a variable to hold something called the PIVOT.

We now have:

1	2	3	4	5	6	7	8	9
42	70	20	40	51	10	61	79	30

FIRST = 42 UP = 1 DOWN = 9 PIVOT = 42

42.9.3 What actually happens when you call the quicksort procedure?

A slightly more detailed pseudo-code description of what quicksort does is as follows:

1. If the number of data items to be sorted is greater than one then
2. Make the PIVOT equal to the first piece of data in the set of data we want to sort.
3. Make FIRST equal the first piece of data in the set of data we want to sort.
4. Make UP the position of the first piece of data.
5. Make DOWN equal the position of the last piece of data.
6. REPEAT
7. Increment UP until you find the piece of data pointed to by UP that is greater than the PIVOT value (or until you get to the last position).
8. Decrement DOWN until you find the piece of data pointed to by DOWN that is less than or equal to the PIVOT. Always start by checking the actual value pointed to by DOWN first!
9. If the pointer UP is less than pointer DOWN then
10. Swap the data pointed to by those pointers.
11. End If
12. Until (UP meets DOWN) or (UP passes across DOWN).
13. Swap FIRST with the data item pointed to by DOWN.
14. Set PIVOT equal to the value pointed to by DOWN.
15. End If.

42.9.4 Let's work through the example!

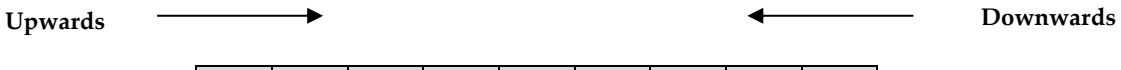
After lines 1 to 5 in out pseudo-code, we have:

1	2	3	4	5	6	7	8	9
42	70	20	40	51	10	61	79	30

FIRST = 42 UP = 1 DOWN = 9 PIVOT = 42

We now need to do the pseudo-code in lines 6 to 14. Increment UP until UP is greater than PIVOT. The first piece of data that is greater than 42 is 70. Decrement DOWN until the data pointed to by DOWN is less than or equal to PIVOT. In fact, 30 is the first piece of data that is less than or equal to 42.

We now have the following:



1	2	3	4	5	6	7	8	9
42	70	20	40	51	10	61	79	30

FIRST = 42 UP = 2 DOWN = 9 PIVOT = 42

UP is less than DOWN (2 < 9) so we swap the data in position 2 with the data in position 9, to give:

1	2	3	4	5	6	7	8	9
42	30	20	40	51	10	61	79	70

FIRST = 42 UP = 2 DOWN = 9 PIVOT = 42

42.9.5 Continue!

The pointers UP and DOWN haven't met and neither have they crossed over. We are therefore still inside the REPEAT construct. We need to continue as before. We continue to increment UP until we find the next piece of data where UP is greater than PIVOT. UP is at position 5 (data 51) when this occurs. We decrement DOWN until we find the first piece of data where DOWN is less than or equal to PIVOT. DOWN is at position 6 when this occurs (data 10). We then exchange their values because UP is less than DOWN. We now have this:

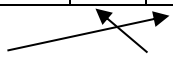
1	2	3	4	5	6	7	8	9
42	30	20	40	10	51	61	79	70

FIRST = 42 UP = 5 DOWN = 6 PIVOT = 42

42.9.6 And continue again!

The pointers UP and DOWN haven't met and neither have they crossed over. We are therefore still inside the REPEAT construct. We need to continue as before. We increment UP until we find the first piece of data where UP is greater than PIVOT. UP is at position 6 (data 51) when this occurs. We decrement DOWN until we find the first piece of data where DOWN is less than or equal to PIVOT. DOWN is at position 5 when this occurs (data 10). We do <u>not</u> exchange the data values at positions UP and DOWN this time because UP is not less than DOWN. We now have this:

1	2	3	4	5	6	7	8	9
42	30	20	40	10	51	61	79	70



FIRST = 42 UP = 6 DOWN = 5 PIVOT = 42

UP has actually passed across DOWN. This is one of the exit conditions of the REPEAT construct in line 12 of our pseudo-code. We therefore drop out of the REPEAT loop. Line 13 of the pseudo-code says to swap FIRST with the data item pointed to by DOWN. Line 14 of the pseudo-code says that PIVOT should then be given the value pointed to by DOWN. This gives us:

1	2	3	4	5	6	7	8	9
10	30	20	40	42	51	61	79	70

PIVOT = 42 (which, in this case, just so happens to be what it was before!)

42.9.7 Something important to note!

All the data values to the left of the PIVOT value are less than the PIVOT value. All of the values to the right of the PIVOT value are greater than the PIVOT value. This means:

1) The PIVOT value is now in exactly the correct place if, for a moment, we assumed that all data items were now ordered correctly.
2) There is an unordered set of data items to the left of the PIVOT. These data items are all less than the PIVOT value.
3) There is an unordered set of data items to the right of the PIVOT. These data items are all more than the PIVOT value.

We can represent this as:

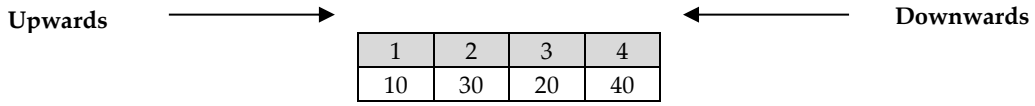
1	2	3	4
10	30	20	40

5
42

6	7	8	9
51	61	79	70

42.9.8 Now work on the left hand side.

What we can now do is treat the left hand set of data items as a completely independent set of data that needs sorting using the quicksort algorithm again! We call the quicksort algorithm again (recursion) and continue.



1	2	3	4
10	30	20	40

FIRST = 10 UP = 1 DOWN = 4 PIVOT = 10

Increment UP until it is greater than PIVOT. UP equals 2 (data 30). Decrement DOWN until it is less than or equal to PIVOT, as before. DOWN therefore equals 1 (data 10).

1	2	3	4
10	30	20	40

FIRST = 10 UP = 2 DOWN = 1 PIVOT = 10

42.9.9 Continue!

We don't exchange the data items pointed to by UP and DOWN because the pointer UP is not less than the pointer DOWN. We have met one of the exit conditions for the REPEAT loop and so we exit the loop. We exchange FIRST and DOWN (so in this particular case they stay the same because FIRST and DOWN have the same value). PIVOT is then given the value of the data pointed to by DOWN, which is 10 (the same as it was before). PIVOT is now in the correct place if *all* of the data were in order.

There aren't any data items to the left of the PIVOT so there isn't a left hand side of the PIVOT to work on. However, there is more than one piece of data to the right of the PIVOT. We therefore quicksort the right hand side. We call the quicksort algorithm again (recursion) and continue. We now have:

1	2	3
30	20	40

FIRST = 30 UP = 1 DOWN = 3 PIVOT = 30

Incrementing UP until the data item pointed to by UP is greater than PIVOT, we get UP equals 3 (data 40). Decrementing DOWN until the data item pointed to by DOWN is less than or equal to PIVOT, we get DOWN equals 2 (data 20). We don't exchange the data items pointed to by UP and DOWN because the pointer UP is not less than the pointer DOWN. Because UP and DOWN have crossed, however, we must swap FIRST with the data item pointed to by DOWN and then PIVOT is given the data item pointed to by DOWN. PIVOT is now in the correct place if *all* the data were in the correct order.
We now have:

1	2	3
20	30	40

FIRST = 20 UP = 3 DOWN = 2 PIVOT = 30

42.9.10 Remember!

Just to remind you, after every quicksort pass

1) The PIVOT value is in the right place.
2) All data to the left of the PIVOT are less than the PIVOT.
3) All data to the right of the PIVOT are greater than the PIVOT.

The left hand side of the PIVOT has only one data item and so doesn't need quicksorting. This also applies to the right hand side of the PIVOT. These three data items must therefore be sorted in order!

So, we started with:

1	2	3	4
10	30	20	40

and quicksorted them into:

1	2	3	3
10	20	30	40

42.9.11 A summary so far.

To summarise what we have done so far, we started with this:

1	2	3	4	5	6	7	8	9
42	70	20	40	51	10	61	79	30

And quicksorted it into this:

1	2	3	4
10	30	20	40

5
42

6	7	8	9
51	61	79	70

We then recursively called quicksort to sort the left hand side of the PIVOT, to get this:

1	2	3	4
10	20	30	40

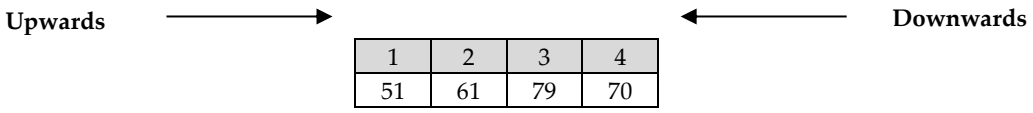
5
42

6	7	8	9
51	61	79	70

We now need to quicksort the right hand side.

42.9.12 Quicksorting the right hand side.

Remember to follow the algorithm! It is a very mechanical process.



1	2	3	4
51	61	79	70

FIRST = 51 UP = 1 DOWN = 4 PIVOT = 51

UP is incremented until the data it is pointing to is greater than the PIVOT. UP is incremented until it equals 2. Down is decremented until it points to a piece of data that is less than or equal to the PIVOT. Down is decremented until it equals 1.

The pointer UP is not less than the pointer DOWN so they are not swapped. UP and DOWN have crossed over so you exit the REPEAT loop. FIRST is swapped with the data item pointed to by DOWN (they both are 51 in this particular case). PIVOT is set to DOWN, which is 51 (which just happens to be what it was before).

42.9.13 Continue!

We now have a PIVOT value of 51. There aren't any data items to the left of the PIVOT so there isn't a left hand side of the PIVOT to work on. However, there is more than one piece of data to the right of the PIVOT. We therefore quicksort the right hand side.

1	2	3
61	79	70

FIRST = 61 UP = 1 DOWN = 3 PIVOT = 61

UP is incremented until the data it is pointing to is greater than the PIVOT. UP is incremented until it equals 2. Down is decremented until it points to a piece of data that is less than or equal to the PIVOT. Down is decremented until it equals 1.

The pointer UP is not less than the pointer DOWN so they are not swapped. UP and DOWN have crossed over. FIRST is swapped with the data item pointed to by DOWN (they both are 61 in this particular case). PIVOT is set to DOWN, which is 61 (which just happens to be what it was before). We now have a PIVOT value of 61. There isn't a left hand side so we quicksort the right hand side by calling the quicksort algorithm recursively again.

1	2
79	70

FIRST = 79 UP = 1 DOWN = 2 PIVOT = 79

UP is incremented until the data it is pointing to is greater than the PIVOT (or you get to the last position). UP is incremented until it equals 2 (the last position). DOWN is decremented until it points to a piece of data that is less than or equal to the PIVOT. In fact, DOWN stays at the value 2 because 70 is less than or equal to the PIVOT. Both UP and DOWN now equal 2.

The pointer UP is not less than the pointer DOWN (they are both the same). The pointer UP has met the pointer DOWN. This is an exit condition for the REPEAT loop. FIRST is now swapped with the data item pointed to by DOWN so is given the value 70. PIVOT is then set to the data item pointed to by DOWN, which is 70. We now have the data in this order:

1	2
70	79

We have a PIVOT value 70. There isn't a left hand side of the PIVOT and the right hand side only has one piece of data, so we have finished quicksorting this data.

42.9.14 Now recombine all of the recursive calls.

We started with this:

1	2	3	4	5	6	7	8	9
42	70	20	40	51	10	61	79	30

And we have ended with this:

1	2	3	4	5	6	7	8	9
10	20	30	40	42	51	61	70	79

42.10 Merging data

Merging two files is the name given to taking both files and creating one new file out of them. For example, File one might contain: 1, 6, 9, 16 and 22, and file two might contain: 2, 5, 23, 33, 100. If you merge the files, you end up with one file containing: 1, 2, 5, 6, 9, 16, 22, 23, 33, and 100. The programmer would write a procedure to merge two files. One possible merge pseudo-code, trying hard to use the sequence, selection and iteration constructs, is as follows:

```
READ first data in file A
READ first data in file B
WHILE not (end of file A) and not (end of file B)
        IF A < B
        WRITE A to file C
        IF not (end of file A)
                READ next data in file A
        ENDIF
        ELSE
        WRITE B to file C
        IF not (end of file B)
                READ next data in file B
        ENDIF
        ENDIF
ENDWHILE
WRITE remaining records from file that is not empty to file C.
```

42.10.1 An example of merge using the pseudo-code

See if you can follow the pseudo-code to do the following merge. We have used a trace table to help us. Two small colleges have just merged. They currently each have a file that holds details about the courses they offer. We require a program that can merge these two files into one file for the new 'super college'. We will use the Primary Key of each course in the merging algorithm. Merge the following two sorted files:

File A: 3, 7, 18, 80
File B: 1, 10, 19, 70, 100, 280, 500, 600, 800

File A	*File B*	*Merged file C*
3	1	1
3	10	1, 3
7	10	1, 3, 7
18	10	1, 3, 7, 10
18	19	1, 3, 7, 10, 18
80	19	1, 3, 7, 10, 18, 19
80	70	1, 3, 7, 10, 18, 19, 70
80	100	1, 3, 7, 10, 18, 19, 70, 80
-	100	1, 3, 7, 10, 18, 19, 70, 80, 100, 280, 500, 600, 800

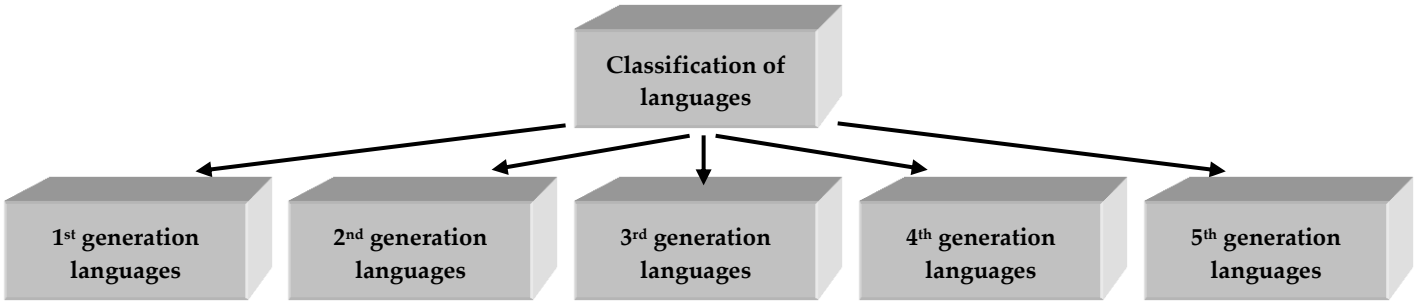
Q1. How many comparisons would it take on average to find a record in a file of size X records using serial searching?
Q2. Outline the difference between a serial file and sequential file.
Q3. What is a 'flag' in programming? What data type does it usually take?
Q4. What is the result of truncating the number 7.4?
Q5. Write down the algorithm for doing a binary search.
Q6. Do some research on the Internet. Write down an algorithm for doing an insertion sort.
Q7. Why is a bubble sort known as a bubble sort?
Q8. What is meant by merging two files together?
Q9. What do you do when you get a data item that appears in both files, when you are trying to merge them together?
Q10. Write down an algorithm for merging two files together.

Chapter 43 - History of programming paradigms

43.1 A history of the development of programming languages

Before you begin, note that the strange word 'paradigms' simply means 'methods'. 'Programming paradigms' are the 'programming methods' available to you. The paradigms discussed here are OOP, procedural, declarative and functional languages. To understand how these languages have come about, you'll need to do some background reading. You should use your textbooks and search the Internet for different points of view.

In this section, we are going to classifying languages according to their historical and functional development. Then we will classify languages according to their paradigm-type: OOP, Procedural, Declarative and Functional.



Classifying languages by generation.

43.2 First generation languages

Computers are digital devices. They use patterns of binary numbers (ones and zeros). This is known as 'machine code'. Originally, you would write programs by writing patterns of ones and zeros! You can imagine how boring that was; it was difficult to write, easy to make errors, difficult to debug and took ages! Slightly (only just) better than this was to write programs using hex codes rather then binary! Hex codes were simply groups of bits. Instead of typing in 0111 1111, you could type in 7F. Great! These programming languages, while having many drawbacks, didn't need translation because they were written in the language of the processor - machine code. They have applications in areas such as control, where fast processing is needed. Binary and Hex programming languages are known as 'first generation languages'. Programs written in these languages run very quickly compared to programs written in high-level languages.

43.3 Second generation languages

It was recognised that writing in binary or hex was far from ideal. Languages were written to get over some of the problems of first generation languages. They were much closer to English but further from machine code. These 'second generation' languages were known as **assembly languages** or **low-level languages**. They contained shortened English instructions called '**mnemonics**. Examples of instructions include ADD, SUB, NOP, LDA, STA and so on. When a program was written, however, you couldn't just run it as before, because it wasn't in machine code, the language that the processor used. You had to convert (or 'translate') it first back to machine code and then you could run it. The type of program that translated assembly code into machine code was called an '**assembler**'. Second generation languages are used for real-time applications and where the programmer needs to manipulate the hardware inside a computer.

43.4 Third generation languages

Second generation languages were an improvement on what existed previously, but were still far from ideal. They were not particularly user-friendly and excluded people who had little training from using them. There was also a need to make programming focus more on specific applications, with an instruction set that helped get specific tasks done easily (such as file processing, for example). A whole range of third generation languages (also known as **high-level languages**) were written to do this; BASIC and PASCAL for teaching, COBOL for data processing, FORTRAN for scientists and so on. These languages require the user to specify all the steps needed to solve a problem. Think of your own experience of writing in a third generation language. Keywords are closely related to English. Your 'Integrated Development Environment' (IDE) enabled code to be developed easily, debugged and finally translated into machine code (now known as 'object code') quickly and effectively.

43.5 Fourth generation languages (4GL)

As computers became more powerful, software emerged that reduced the time it took to produce programs for users. Report Program Generator is an example of this type of program. This language is used to increase the speed of report production. It takes the actual format of the data and combines it with instructions on how to format a report. It then produces the report. Later on, you will learn about CASE tools. These are used to help an analyst analysis and design new systems. 4GL have been used to turn CASE designs into actual software or parts of the final product.

43.6 Fifth generation languages

Expert systems are considered 5th generation programs. They could be written in PROLOG, for example, which is considered a 5th generation programming language. You write applications by giving your computer a set of facts and then telling the computer how those facts relate to each other. You can then query the program by asking questions. It will search for facts based upon the rules you have declared and the search you are doing. What you don't do with an expert system is to tell it how to solve the problem (by writing a program that tells it what to do step-by-step, as in procedural languages such as COBOL or Pascal). You simply tell expert systems facts and rules and then ask it a question!
For an expert system to work, you need:

- An interface to let you enter facts and rules.
- A method of storing facts and rules.
- An interface to allow you to update facts and rules periodically.
- An interface to allow you to ask questions.
- An 'inference engine' - software that can actually perform queries.

43.7 Procedural, declarative, OOP and functional paradigms

Languages can also be classified according to their **type** as well as their **generation**. We will consider in detail procedural, declarative, OOP and functional paradigms.

43.7.1 Procedural languages (also known as Imperative languages)

With these kinds of languages, you must tell the computer exactly what to do, instruction by instruction. You will see later that this is not the only way to write a program! The computer will start at the first instruction, carry out the command and then move on to the next one. The programmer writes programs by using only three basic building blocks. These are sequence, selection and iteration.

In Pascal, conditional branching (selecting which code to do) is possible using the IF statement and the CASE statement. Iteration is possible using FOR, WHILE and REPEAT. Procedural languages also make heavy use of functions and procedures. This promotes modular programming, which allows code to be reused, better understood and better designed amongst other benefits. This approach isn't perfect, however, because the variables that are used within procedures and functions can always be accidentally changed elsewhere in the program and finding these kinds of bugs can be very difficult and time-consuming! PASCAL was written as a teaching language, to teach the elements of structured programming. COBOL has commands that make it excellent at data processing applications. FORTRAN focuses on scientific and engineering applications and so on. Other procedural languages have also been written. Each one provides the programmer with the instructions and facilities that enable programs for particular application areas to be written.

43.7.2 Declarative languages

We will introduce declarative languages here. There is a chapter later that shows some examples using PROLOG. A declarative language is very different from a procedural one. It is written in a completely different way. With procedural languages, you have to tell the computer exactly what to do, step-by-step. With declarative languages, you don't. You write a program by stating facts and relationships between facts. Then you ask the program questions. You don't state how to find information, only what information you want.

- You tell the computer facts and how those facts relate to each other.
- Running a program involves you stating a goal and letting the language work out whether the goal can be achieved or not by looking at the facts and their relationships.
- The order of statements is really important in procedural languages such as Pascal. They are not at all important in declarative languages.
- The route through a program needs to be carefully thought through with procedural languages. In declarative languages, the program works through a route of facts as it finds them. If it comes up against a dead-end, it ‘backtracks’ and tries a different route, until it achieves the goal you set it, or it can't achieve the goal at all.

Declarative languages are suited to artificial intelligence applications. Programs can be written to suggest illnesses from symptoms or to produce probable minerals present in a particular environment, for example.

43.7.3 Object-oriented languages

Good programming using a procedural language will employ a modular approach. There are many good reasons for this. One of them is that procedures and functions can be written which can then be stored in a library and used in other programs. That will save time and money developing future programs. Unfortunately, problems can still arise when you try to use library modules. Variables used in a module, for example, may get changed in an unexpected way by a program, or vice versa. This may cause a bug and if it does, they can be very hard indeed to find.

A different way of programming is the object-oriented approach. In this kind of paradigm, objects are defined. An object is a real-world thing. It might be a car, a person or a command button in Visual Basic. The programmer, before writing the program, asks what objects exist in the problem area, what **events** can happen to those objects and what should happen, **the methods**, when a particular event occurs.

The data needed to define an object, the things that can happen to that object (**events**) and the actions that need to take place when a particular event occurs (**methods**) are bound together (called **encapsulation**). Once an object has been encapsulated, it cannot fail as a result of the actions of other parts of the program, as procedural programs can.

OOP is conceptually much closer to the real world than procedural programming. It is much easier to think about what objects there are in a program and what might happen to an object than to try to break down a problem into modules and then write down a sequence of instructions that work through the modules. A later chapter deals with the concepts behind OOP in much more detail.

43.7.4 Functional languages

These types of languages are geared towards science and engineering applications. Writing a functional program involves breaking a maths problem down to the smallest possible units, then writing a set of functions for those units. Each function then receives parameters from outside the function, manipulates the parameters mathematically and then outputs parameters to other functions. Characteristics that make them suitable for science, maths and engineering application areas include:

- An instruction set geared towards handling maths problems.
- Being excellent at handling recursion.
- You can write mathematically-reliable programs.

43.7.4.1 Getting started with Functional languages

As with other paradigms, you will not fully understand the fundamentals of functional programming unless you get some experience in it. You will need to put aside a week or two to get a basic understanding. Download the functional programming language compiler called HUGS from **http://www.haskell.org/hugs/** (HUGS is based on a functional language called Haskell). If you are using Windows, you want to find and download the file called hugs98-Dec2001.msi. The other files are useful only if you want to really get interested in HUGS and explore it!

When you have downloaded the file, double-click on it to set it up. When you've set it up, open the HELP files and work your way through the **HUGS for Beginners** section! If you get stuck, use the HUGS homepage and use the Internet to find forums and discussion groups where you can get help. There are many other online guides to help you get started. For example, **http://gnosis.cx/publish/programming/Haskell.pdf**

Q1. What is meant by a first generation language?
Q2. What is a 'hex digit'?
Q3. Give an example of a mnemonic and say why it is a mnemonic.
Q4. Give a use for an assembly language.
Q5. What is meant by a high-level language and a low-level language?
Q6. Why are third generation languages an improvement over second generation languages
Q7. State three examples of third generation languages.
Q8. What is meant by a procedural language?
Q9. Describe what a declarative language looks like.
Q10. What are the key characteristics of an object oriented language?

Chapter 44 - Jackson Structured Programming (JSP)

44.1 Introduction

When developing programs in a third-generation language, the traditional way of approaching the problem has been to use a top-down approach. This is where the problem is written down very simply. Then it is split up into a number of smaller modules. Each of those smaller modules is then broken down into ever-smaller modules until each module does one and only one job. They are then given to the programmers to code.

44.2 Jackson Structured Programming (JSP) diagrams

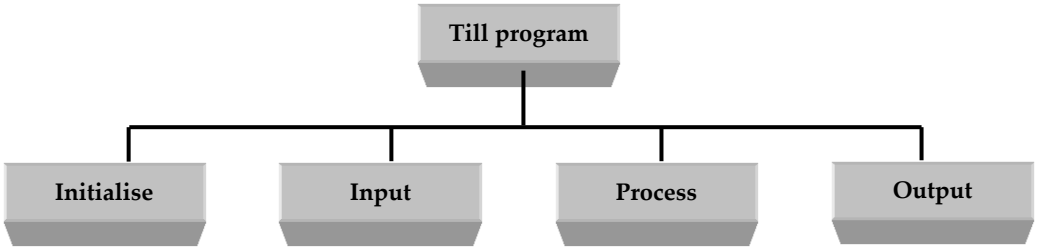
Top-down diagrams in programming are often drawn using Jackson Structured Programming (JSP) diagrams. These seek to break down problems into manageable chunks, to help people better understand a problem and to help them plan a solution.

44.3 An example of top-down design using JSP diagrams

Consider the following problem. You need a program that calculates the checkout total for a customer at a cash-and-carry. To shop at this cash-and-carry, you need to be a member and memberships come in three types: bronze, silver and gold. Each membership type gets its own discount! A customer at a checkout first gives details of their membership card. The checkout assistant then scans in each item. The price is retrieved from the stock database. All the items are added up and the appropriate discount subtracted from the total. The checkout assistant then tells the customer how much is due. They receive cash from the customer, then calculate and give out the change as well as an itemised receipt.

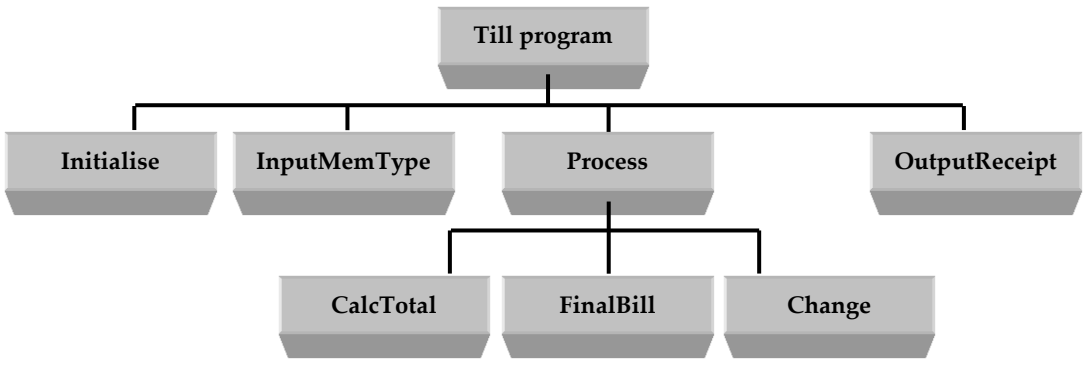
STEP 1

Define the problem simply. If you initially start by breaking down **every** problem into the four modules shown above, **initialise**, **input**, **process** and **output**, it will help you get started!



STEP 2

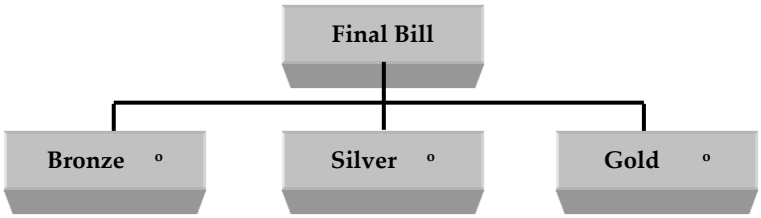
Break the problem down into smaller modules. You read this diagram by reading the top level of the diagram first, from left to right in **sequence**. Then read the second level of the diagram, left to right in **sequence**, then the third level and so on.



STEP 3
Break the problem down into further modules wherever possible.

44.4 Showing 'selection'

Before we can go any further with this, we are going to have to introduce some drawing conventions. Consider for a moment the module 'Final bill'. The final bill will be worked out by taking the total from the 'CalcTotal' module, taking the discount due from the 'InputMemType' module and then subtracting a discount from the total. We can represent this in the following way:

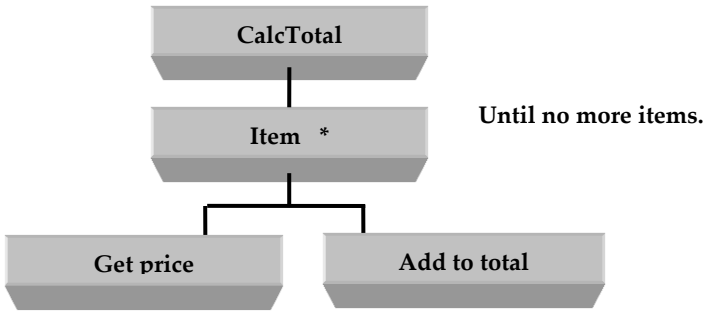


Showing 'selection' in JSP.

You can read the above as "If the customer has a bronze membership, then do the module Bronze. Else if they have a Silver membership, do the module Silver, Else if they have a Gold membership, do the module Gold." Note the little circle in the top right hand corner of each of the possible modules that could be done. That is how you show choice!

44.5 Showing 'iteration'

There is one other drawing convention we need to know about. To do the 'CalcTotal' module, you will need to read in an item, add it to a running total, then read in the next item and add it to the running total and keep repeating this until there are no more items. How do we show a repetition (otherwise known as an iteration)? We show it like this:

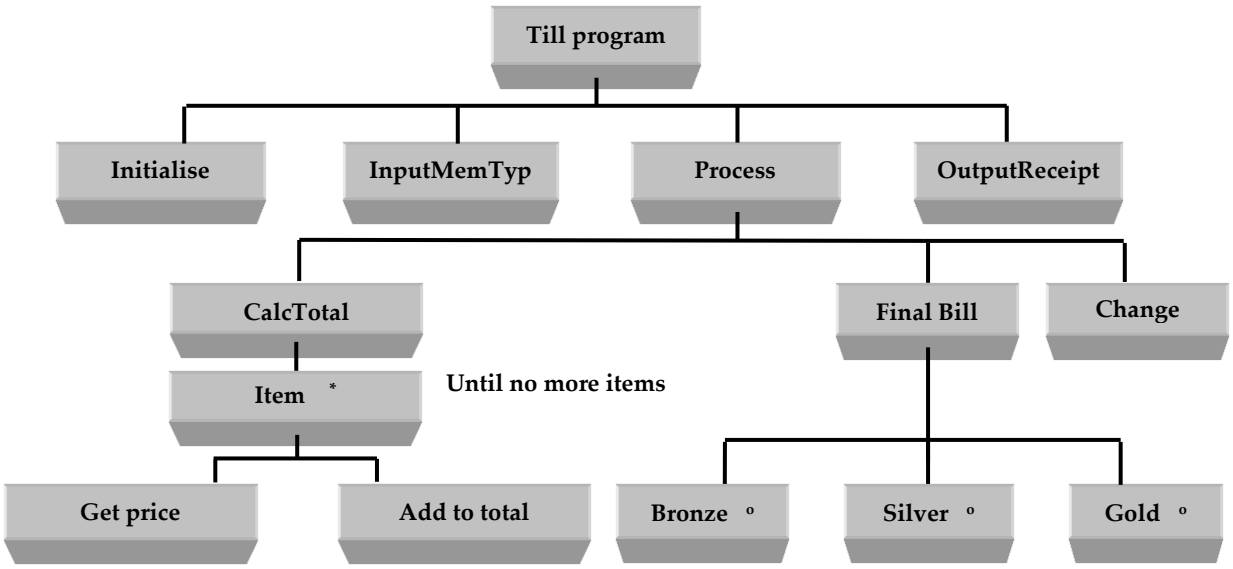


Showing 'iteration' in JSP.

Notice the use of what appears to be an extra box with a star in it! Omitting it is a common mistake when you are first learning about this design method. Don't forget the comment just outside the box, in this case "Until no more items".

44.6 Back to STEP 3

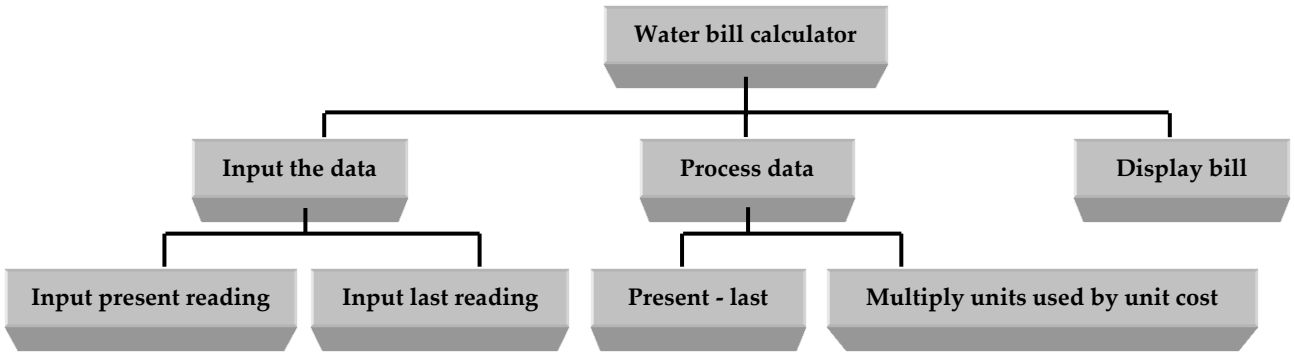
We can now show the final JSP diagram, including the structures that are needed to show selection and iteration.



The final JSP diagram for our problem.

The diagram you have produced is called a **JACKSON STRUCTURED PROGRAMMING** diagram, or JSP diagram for short. It is only made up of three types of constructs: SEQUENCE, SELECTION and ITERATION and you have seen examples of how to represent each type in the above diagram. You will need to do two or three examples yourself before you are confident at representing problems using this method.

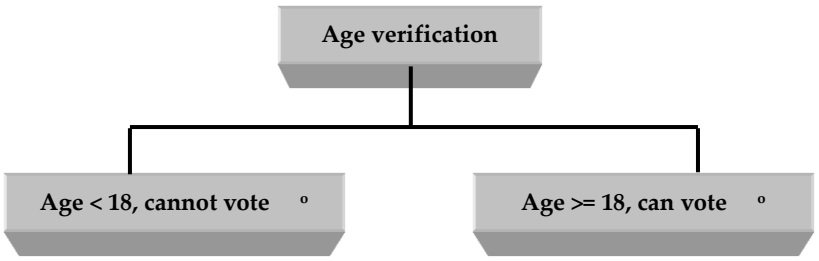
44.7 More examples of JSP – sequence



Another example of sequence.

On the first level is the problem. This is represented by one box. The one box has been broken down into three modules. These are each represented by a box and are all in line on the next level. We read these three boxes in sequence, starting on the left. We can break down some of these boxes further, as shown in the diagram. These are shown on the next level. Again, we read the boxes in sequence from left to right. In this JSP diagram, we have only used the construction sequence.

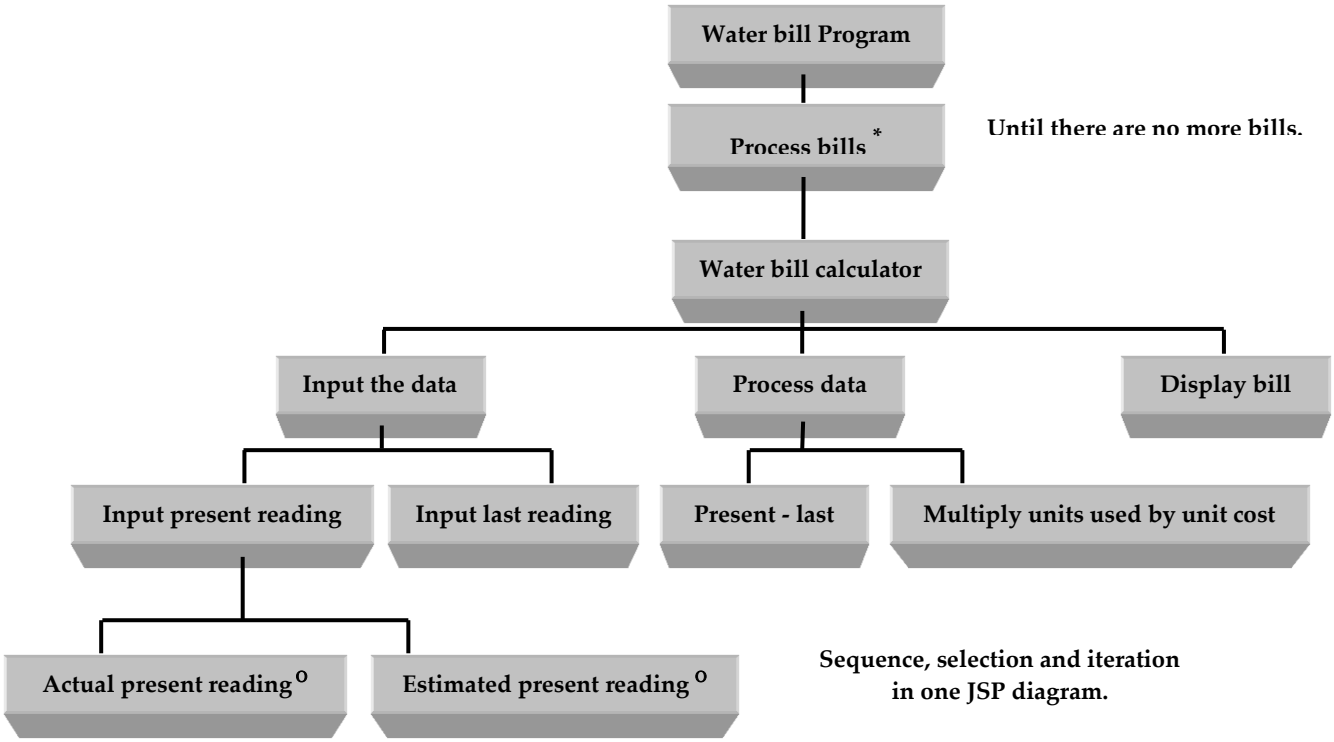
44.8.1 More examples of JSP – selection



Another example of selection.

In this example, we can see how selection is represented in part of a program that deals with age verification. You must be 18 or older to vote. If you are less than 18 then the block of code on the left is run. If you are 18 or older then the block of code on the right is run. There is a little zero in the top left of each choice to indicate that the box is one choice of a number of choices. This is how selection is represented using JSP.

44.8.2 More examples of JSP



Sequence, selection and iteration in one JSP diagram.

In this diagram, we can see that **iteration** has been used. We should note the use of what appears to be an extra box with a star in it. The star denotes that there is an iteration going on and the iteration should continue until the condition that is written just under the box is met. In this case, it's 'Until there are no more bills'. This JSP diagram also shows the sequence construction in use and there is one example of a selection construction.

44.9 The benefits of JSP diagrams

Once you have got a top-down design, you can then write a program. There are a number of benefits associated with the production of a JSP diagram:

- One benefit is that you can see the whole program as one diagram.
- JSP diagrams give you a tool (the diagram) to discuss the program with other designers and programmers, for example, to see if there are any improvements that can be made.
- The diagram helps the management of building a program because a team leader can see what modules need to be distributed.

- JSP diagrams can help with the identification of the more complicated modules. These can be given to the more experienced programmers.
- One important advantage JSP diagrams have is that they are language-independent although they are really meant for programs that will be written using procedural languages. Once a design has been done, it can be implemented in any procedural programming language.

44.10 Bottom-up programming

You may also hear about problems being solved using a 'bottom-up approach'. This simply means that you start by programming lots of smaller modules before considering the overall design. Once you have a set of individual modules, you then think about the design of the program. You take modules 'off-the-shelf' and stick them together into a larger program. You test that as if it were a working system. Then you add in more modules to the system so far and get that working. This is repeated until all of the modules you need have been recombined into one program.

44.10.1 An analogy for bottom-up programming

Consider making a computer, the best computer there has ever been! You might start by designing and building various components such as the motherboard, the DVD drive, the hard drive, the monitor and so on. Once you have the design for all of the components, you then consider the design of your PC. You select the first few components you want, connect them together and test them together. You then have a basic, but working computer. You then add another component and test your system again. You keep adding components and testing them, one at a time, until you have a final system. This is an example of you building a computer using a 'bottom-up' method.

Q1. What is meant by a top-down program design?
Q2. When should you stop breaking down a module of code?
Q3. What are other names for a module of code?
Q4. What is meant by initialisation?
Q5. What are the three programming constructions in procedural languages?
Q6. How do you show 'sequence' using Jackson Structured Programming diagrams?
Q7. How do you show 'selection' using Jackson Structured Programming diagrams?
Q8. How do you show 'iteration' using Jackson Structured Programming diagrams?
Q9. What are the benefits of using JSP diagrams?
Q10. What is meant by bottom-up programming?

Chapter 45 - Object oriented programming

45.1 Object oriented programming

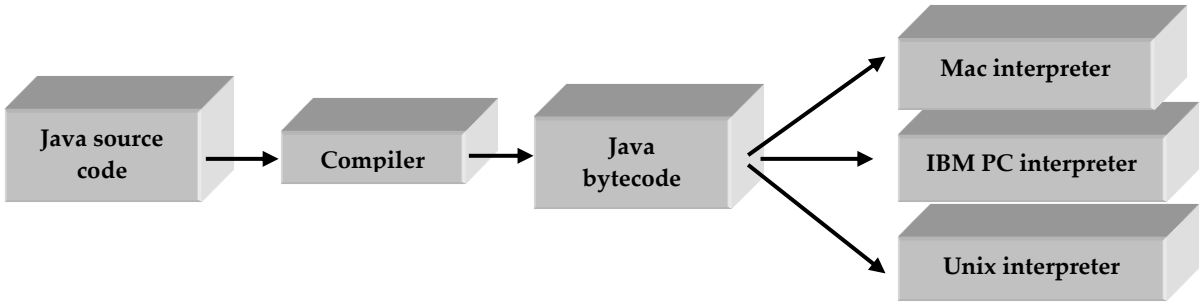
Object oriented programming (OOP) is a programming language that constructs programs in a way that is different to the top-down programming methodology used in PASCAL or COBOL, for example. Examples of OOP languages include Visual Basic and Java. In the following sections, we will briefly look at JAVA and then some of the concepts behind OOP. It is difficult to see how you can fully understand the concepts behind OOP without using them. I strongly recommend that you get some experience! It will take you a weekend of work.

Go to this site: **http://www.bluej.org/tutorial/tutorial.pdf** and printout the BlueJ introductory tutorial. The tutorial also gives you instructions on how to download and set up Java on your machine, with an additional piece of software called BlueJ. BlueJ is written for beginners and will cover the basic concepts of OOP. It is thoroughly recommended! You will need to download the Sun Java compiler from **http://java.sun.com/j2se/** and also the BlueJ programming environment from **http://bluej.monash.edu**. Install Java first, then BlueJ - not the other way around! Set-up in most cases will be very easy and should take a couple of minutes once the files have been downloaded. If you do have any problems then use the BlueJ tutorial information. Once you have installed the software, do the tutorial. It's worth it. Then search the Internet for some 'Java beginner' programs. See if you can enter one or two programs and get them working. As a minimum, try to get a 'Hello World' program working. Greenfoot is also a popular learning environment for Java with some excellent tutorials for beginners.

45.2 Java

Computers such as an IBM clone (a 'normal' PC) or a Macintosh each have their own CPUs that use their own machine code. If you write a program in PASCAL, for example, you can run it on a school PC only after you have translated it using a compiler into machine code. You couldn't, however, take that object code and run it on a Macintosh - because it has a different CPU that has a different instruction set. You would have to retranslate the source code using a different compiler.

Java is an OO high level language. It was designed so that the code can run on any machine! How does it do this? When it is compiled, it is compiled into code known as Java **bytecode** for a machine that doesn't exist, called a Java virtual machine!! The bytecode can then be distributed to different types of computers. Each of these types of computers will need to have their own type of interpreter (rather than a compiler). These interpreters can take bytecode and run it!



Java is both compiled and interpreted.

Why not simply miss out the 'Java bytecode' stage and distribute the source code and have a compiler for each type of machine rather than an interpreter? Amongst other reasons, compilers are more complex programs compared to interpreters. If you have a new type of CPU it is far easier to write a new interpreter than a new compiler.

45.2.1 Java and the Internet

Java is used extensively on the Internet. Small programs called **applets** are written by programmers and transmitted with html code across the Internet. If you have a browser that has a Java bytecode interpreter (most of the latest ones have!) and you have enabled your browser to accept Java applets, then they will run when downloaded. Suddenly, very boring html web pages can be turned into anything the programmer wants to turn them into! Not everyone likes the idea of downloading and running programs not guaranteed to be virus-free and which may compromise personal privacy. As a result, some people disable Java applets on their PC!

45.3 An introduction to object oriented programming (OOP)

In Pascal and other third-generation languages, a top-down programming approach is used. A problem is constantly broken down into modules until each module can be broken down no more. Then a set of procedures and functions is written, one for each module. Finally the actual program is written. The actual program will simply be a call to all of the procedures and functions. When you call a procedure, you simply type the name of the procedure. You may even pass some parameters to it. What actually goes on inside the procedure is of little importance (assuming it works). From the program's point of view, if it needs to use a procedure, it just calls it! This is a kind of **information-hiding** approach. It doesn't need to know about the data and variables inside the procedure, just how to call it. Unfortunately, in third-generation languages, things can still go wrong. It is possible for other procedures and functions to have a direct effect on the insides of a particular procedure. This is one way that difficult-to-find bugs come about. The principal of 'information hiding' (or more technically known as '**data encapsulation**') has been taken one step further with the OOP approach. In OOP, data can only be accessed via **methods** provided by **objects**! These terms will be explained in the next section.

45.4 Objects and methods

The central idea of object-oriented programming is that any program is made up of 'objects'. An object is an **instance** of a class (see below) and relates to an actual object that can be found in the real world. Here are some examples of classes and objects.

- You might have a class 'shape'. An instance of that class (in other words, an 'object') might be 'square_1'. Another might be 'square_2'. Another might be 'Circle_1'. Each of these 'objects' will have their own data.
- You might have a class 'pupil'. An instance of that class might be 'David Smith'. Another object might be 'Mary Jones'. Each of these 'objects' will store the data about that particular pupil.
- You might have a class 'Manufacturer'. Objects might include 'Ford', 'Saab' or 'Skoda' for example. Again, each object contains data about that particular manufacturer.

45.5 Classes can be viewed as templates

Classes can be viewed as templates from which instances of that class (objects) can be produced. Producing an object or instance from a class is known as '**instantiation**'. A class states what data any object will need and what can be done to that data (known as the '**methods**'). For example, if you had a class Pupil, any object produced from that class might need data such as name, address, contact number and timetable. Any object would need ways of accessing that data so you would need some methods. These might include PrintAddressLabel(), GetEmergencyNumber() and PrintTimetable(), for example. The brackets after the name of each method indicate that it is a method and not a piece of data.

45.6 Objects inherit what data to store and what methods they can use from their class

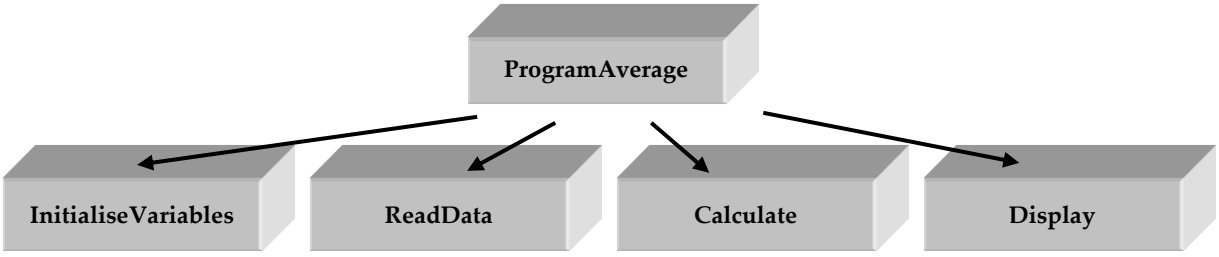
We have said that when an instance of a class is produced, for example, when a new pupil joins a school, they will have the ability to store the data that was defined in the class they came from. They will also **inherit** the methods from that class. You can define one class and use it to produce many objects. For example, you could have a Pupil class and use it to create objects called Smith, Jones, Cooper and Taylor. Each object would store the data specific to that person and the list of data that could be stored came from the class.

45.7 Accessing data via the methods of an object

If anybody wants to get access to a new pupil's data, they can ***only*** do it by calling a **method**. The method then accesses the pupil's data. This technique is fundamental to OOP and is known as '**data encapsulation**'. It means that once an object has been set up, the data in that object cannot be corrupted. This is in contrast to procedural languages, as we saw earlier.

45.8 A case study of the procedural approach to programming verses the object oriented approach

Consider a problem that requires the name of pupils to be read in along with each pupil's two exam scores from two modules. The program must then display the names of all the pupils, their two exam scores and their average marks. The first attempt at a program design using a traditional top-down design approach for procedural programs might look like this.



A traditional top-down approach to solving problems.

This design can be improved, but we will stick with this design. It tells us that we need to write 4 procedures. There are really two main problems with this design. The first is that the variables we use in this program have the potential to be changed from **anywhere** in the program. This might not seem a big problem but if the above program was a small part of a much bigger program, then we could see this problem appear. If it does, bugs will occur in the software and they may be very difficult to find! The second problem is that the program we are writing is a '**conceptual**' view of the problem. By this, we mean that it is made up of blocks of code that don't resemble 'things' that we can use our senses to appreciate. They are ideas that exist in our heads rather than something tangible. We may be able to understand 'Calculate' but this is just an idea in our minds, not something we can see and touch. Because of this, programs can be hard to understand.

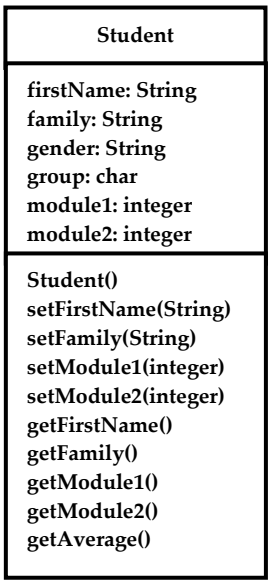
45.9 The object oriented approach

The object oriented approach would not begin by using the top-down design approach to write the program. It would seek to identify real objects. In this case, there is an obvious real world object - a student! Each student object is going to need some data. It might have some String variables to hold their first name and their family name and it may have some integer variables to hold the scores for module 1 and 2. It wouldn't need a variable to hold the average score because if we ever need it, we will calculate it at the time from the module 1 and 2 scores.

We also need some methods to **set** the variable values and also to retrieve (or **get**) the variable values. Remember, a program cannot access the variables of an object directly but must use the methods available. We will need some methods to 'set' the first name, the family name and the module scores. We also need some methods to retrieve the values held in the variables so we need some 'get' methods. We know we have lots of students. Our first job is to design a template, a **class**, from which we can create any number of student objects.

45.10 UML diagrams

We can represent the design of our Student class using a diagram known as a **UML diagram**. This stands for Unified Modelling Language diagram. A UML diagram is split into three. In the top portion, you have the name of the class. In the middle portion, you list the variables you need and what data type each one is. In the third section, you list the methods that you will include. You will always need a special method called a **constructor**. This has the same name as the class. The constructor is the method that is called when you want to create a new object, such as a real student, for example. Generally, you will need a method to set the value of each variable and a method to read the contents of each variable. Note that the set methods also include a 'parameter'. This is the value that you pass to an object when you want to set a variable. So for example, if you want to set the value of module 1 for a student called student1 to 56, then the instruction would be **student1.setModule1(56).** If you ever wanted to read the contents of this variable, then you would use the instruction **student1.getModule1().** Note there is no value within the brackets when you retrieve the contents of a variable. The UML diagram for the class Student looks like this:



The UML diagram for the class Student.

You should use an IDE (Integrated Development Environment) to write an OO program to make life easy for yourselves. An IDE is a software application that provides a programmer with the tools to write applications. Typical tools include:

- **A text editor**, so that you can write and edit programming code.
- **Dubugging tools** such as watch, trace, step and breakpoints, so you can debug programs.
- **Translators**, so you can turn source code into object code.
- **A runtime environment**, so you can run the code and see the results.
- **Auto-documentation tools**, so the associated documentation for a program can be generated automatically and to a standard form.

45.11 Now let's write an OO program!

Let's write some Java code! Remember that the aim of the following example is NOT to teach you Java but to help you understand OO concepts. The first step is to write the class. This is **not** a program, but a block of code that defines the variables and methods that every object of that type will have.

```
public class Student  //This is the name of the template or 'class'
{
        private String firstName; //The data is private - you can only get to it by the methods.
        private String family;
        private String group;
        private String gender;
        private int module1, module2;

/*The first method is a special one. It is used to create an object, a real student, using this class. It has a special
name. It is called a CONSTRUCTOR. We will discuss this in the polymorphism section.*/

        public Student()
        {
        }
        public void setFamilyName (String familyName) //This method allows you to store a family name.
        {
                family = familyName;
        }
        public void setGivenName (String given) //This method allows you to store a first name.
         {
                firstName = given;
        }
         public String getFamilyName() //This method allows you to retrieve a family name.
        {
                return family;
         }
        public String getGivenName() //This method allows you to retrieve a given name.
        {
                return firstName;
        }
        public void setModule1(int mark) //This method allows you to set the mark for the first module.
        {
                module1 = mark;
        }
        public int getModule1() //This method allows you to get the mark for the first module.

        {
                return module1;
        }
        public void setModule2(int mark) //This method allows you to set the mark for the second module.
        {
                module2 = mark;
        }
        public int getModule2() //This method allows you to get the mark for the second module.
        {
                return module2;
        }
        public double getAverage() //This method allows you to calculate then return the average.
        {
                return (module1+module2)/2;
        }
}
```

Do note that this block of code is not a program! It is simply a CLASS that defines the properties of real-world students. Each time we need a new student, we will call the **constructor method** in the Student class. This has the effect of creating a new area in memory for a new student, complete with methods and data. When you have copied the Student class code, compile it. If you get any error messages, you will need to debug it. The first thing to check is whether you have copied the code exactly. Java is very funny about capital letters and punctuation. Create a working folder for all the files you write in this section and save this

class in it. Then close the Student class. If we want to write a program that actually uses these classes, that actually creates objects, we need to write a new class which has a special bit of code called the 'main method'. Start a new java file.

```
public class StudentProgramTest
{
	public static void main(String args[])
	{
		Student student1 = new Student();  //This creates a new student called student1
		Student student2 = new Student();  //Another object created called student2
		String temp2;	//This creates a variable that can be used to hold strings.
		int temp; // This creates a variable that can be used to hold integers.

		System.out.println("Welcome to the Student mark program");
		System.out.println(); //This prints a line space.
		System.out.println("Please enter the first student's first name");
		temp2 = EasyIn.getString(); //EasyIn.getString() gets keyboard input and assigns it to temp2
		student1.setGivenName(temp2);
		System.out.println("Please enter the first student's family name");
		temp2 = EasyIn.getString();
		student1.setFamilyName(temp2);   //This uses student1's setFamilyName method.
		System.out.println("Please enter first student's first module mark");
		temp = EasyIn.getInt();
		student1.setModule1(temp); //This uses student1's setModule1 method.
		System.out.println("Please enter first student's second module mark");
		temp = EasyIn.getInt();
		student1.setModule2(temp);

/*The following code prints out the contents of the variables held in the object called student1.

		System.out.println("Marks for: ");
		System.out.println(student1.getGivenName() + " " + student1.getFamilyName());
		System.out.println("Module 1: " + student1.getModule1());
		System.out.println("Module 2: " + student1.getModule2());
		System.out.println("Average: " + student1.getAverage());

		EasyIn.pause(); //The program won't close until enter is pressed.
	}
}
```

Write, compile, debug if necessary and save this file. Before you try to run it, you need to find on the Internet a class called **EasyIn.java**. Someone else has written this class to help with the business of reading in from the keyboard. (Java doesn't do this automatically). Do a search for **EasyIn.java download**. When you have located this class, download it to your working folder. Now, if you have downloaded EasyIn.java, and you have successfully compiled Student and StudentProgramTest, and you have all of these files in one folder, you can run the program. Open StudentProgramTest, recompile and then run it. Enter data when prompted to. At the end of the program, press <ENTER> to quit.

The whole point of this code is to see that you cannot write directly to student one's variables, the variables that hold the name or the marks. You can only access them via the methods provided. You send a message to the method with the data you want to use and the method sets (or gets) the variables. This is known as 'information hiding' or '**data encapsulation**'.

45.12 Polymorphism – more about constructors

In the previous program, when we wanted to create a new real student, we used the line:

Student student1 = new Student();

This called the constructor method Student() in the Student class and a new student was created. Did you notice though that no details about the new student were entered when we first created the student? We set their name, gender etc afterwards using the methods provided. OO languages, however, give you the facility to create objects with data at the time of creation if you want to. For example, you may want to enter in the name of a student at the time he was created. This is easy to achieve in Java.

We just write another constructor. In our program, we need to modify the Student Class so that it has an extra constructor method. The start of the class should now look like this:

```
public Student ()
{
}

public Student(String name, String surname)
{
        firstName = name;
        family = surname;
}
```

Recompile it and save it. Now you need to see it in action. Modify StudentProgramTest. Change

Student student1 = new Student(); so it reads

Student student1 = new Student("Monty", "Python");

Finally, go to the **StudentProgramTest** code and delete the part of the code that asks you to enter in your student's first and second name and the calls to the methods that change the name variables. We won't need these anymore because we are setting the values when we create the new student.

Excellent! Now recompile the program and test it. When you run it, your program should not ask you for your first student's first and second name, only their marks. But it should display the name Monty Python and his marks! Your Student Class now has two constructors that the programmer can choose which one to use. If they know the student's name, they can pick the constructor that uses parameters, and if they don't want to enter in the pupil's name straight away, they can use the other one that needs no parameters. You can in fact have a range of different constructors but this is beyond the scope of this book! **Do note that both these constructors have the same name, although they do have different parameter requirements**.

The ability to create objects from a class using **different** constructor methods **which have the same name** but **different parameter requirements** is known as **polymorhism**.

45.13 Inheritance

We have created a class called Student. This was for a particular purpose. We needed to store the name of a student, store their two marks and be able to calculate their average mark when necessary.

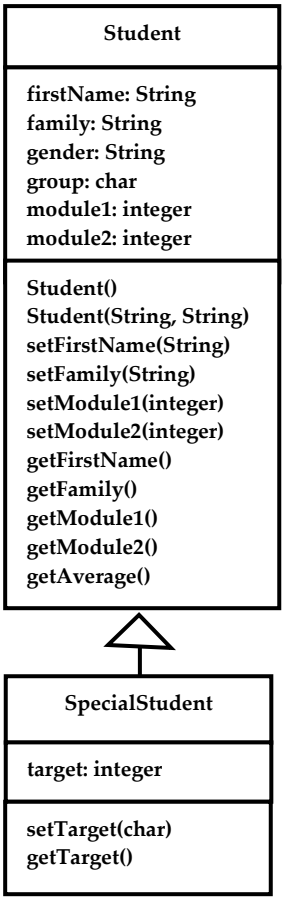
Suppose we now want to be able to store the details about a different kind of student. These 'special students' will need their name and marks recording but they will also have a target average score stored as well. You might reasonably think that we should write a completely new class, perhaps called SpecialStudent. We would define the variables that the class needed as before, including one to hold the target average score, and then define all the methods needed as before, including extra methods to set and get the target. Well we don't! OO languages use something called **inheritance**. In practise, this means we do write a new class but:

- the new class inherits the data and methods of another (already written) class.
- we add to it any extra bits of data and any extra methods we need!

If we are able to use inheritance, it means we can take a previously written class, even if it doesn't quite do what we wanted, and modify it to create a new class. We don't have to start completely from the beginning and code up an entirely new class. We can reuse proven code! This is a very powerful feature of OO programming.

45.14 Representing inheritance using UML diagrams

We can show how that one class inherits the attributes and methods of another class using UML diagrams. Here is an example.



SpecialStudent inherits the properties of Student, in addition to some extra ones!

The UML diagram above shows that Student is the **Superclass**. SpecialStudent is the **Subclass**. SpecialStudent is **derived** from Student. It inherits all of the variables of the Superclass and has an additional one called 'target'. It also **inherits** all of the methods of Student, and some additional ones to set and get the target. This diagram is also called an **inheritance diagram**.

Copy the following code into the folder you have been using for this Java exercise. Compile it and debug if necessary. Note the use of the keyword 'extends' in the title line. This is Java's way of saying that SpecialStudent is a subclass of Student, and therefore inherits all of Student's variables and methods. Some extra ones are then provided.

```
class SpecialStudent extends Student //SpecialStudent inherits the attributes and methods of Student
	{
		private int target;//This is the extra variable each special student will need.

		public SpecialStudent()    //This is the constructor.
		{
		}
		public void setTarget(int aimForThis) //This is the extra set method for level.
		{
			target = aimForThis;
		}
		public int getTarget() //This extra method is used to retrieve a special pupil's level.
		{
			return target;
		}
	}
```

Now you need to test the subclass SpecialStudent, to see if it did indeed inherit Student's variables and methods. You should modify your StudentProgramTest code so that it now looks like the following (or you may want to create a new StudentProgramTest2, if you want to keep the old one). We will just **instantiate** (create) one new student called student21.

```
public class StudentProgramTest
{
	public static void main(String args[])
	{
		SpecialStudent student21 = new SpecialStudent();	//This creates a new Special student.
		String temp2;	//This creates a variable that can be used to hold strings.
		int temp; // This creates a variable that can be used to hold integers.
		System.out.println("Welcome to the Student mark program");
		System.out.println(); //This prints a line space.

		System.out.println("Please enter the Special student's first name");
		temp2 = EasyIn.getString();
		student21.setGivenName(temp2);
		System.out.println("Please enter the Special student's family name");
		temp2 = EasyIn.getString();
		student21.setFamilyName(temp2);
		System.out.println("Please enter Special student's first module mark");
		temp = EasyIn.getInt();
		student21.setModule1(temp); //This uses student1's setModule method.
		System.out.println("Please enter Special student's second module mark");
		temp = EasyIn.getInt();
		student21.setModule2(temp);

		System.out.println("Please enter Special student's target average.");
		temp = EasyIn.getInt();
		student21.setTarget(temp);

		System.out.println("Marks for: ");
		System.out.println(student21.getGivenName() + " " + student21.getFamilyName());
		System.out.println("Module 1: " + student21.getModule1());
		System.out.println("Module 2: " + student21.getModule2());
		System.out.println("Average: " + student21.getAverage());
		System.out.println("Target average: " + student21.getTarget());
		EasyIn.pause(); //The program won't close until enter is pressed.
	}
}
```

You should compile, debug and then run the code! You will see once you have got the program working that SpecialStudent did indeed inherit all of the properties of Student, in addition to some extra ones.

45.15 How would you know an OOP language if you saw one?

As already mentioned, Java and Visual Basic are examples of Objected Oriented Programming languages. There are other languages! For a language to be called Object Oriented:

1) It **must contain objects** that have methods and data, with the data only being accessible via the methods.
2) The objects must be **instantiated from a class**, making use of **polymorphism**.
3) The classes must be able to **inherit** the properties of other classes.

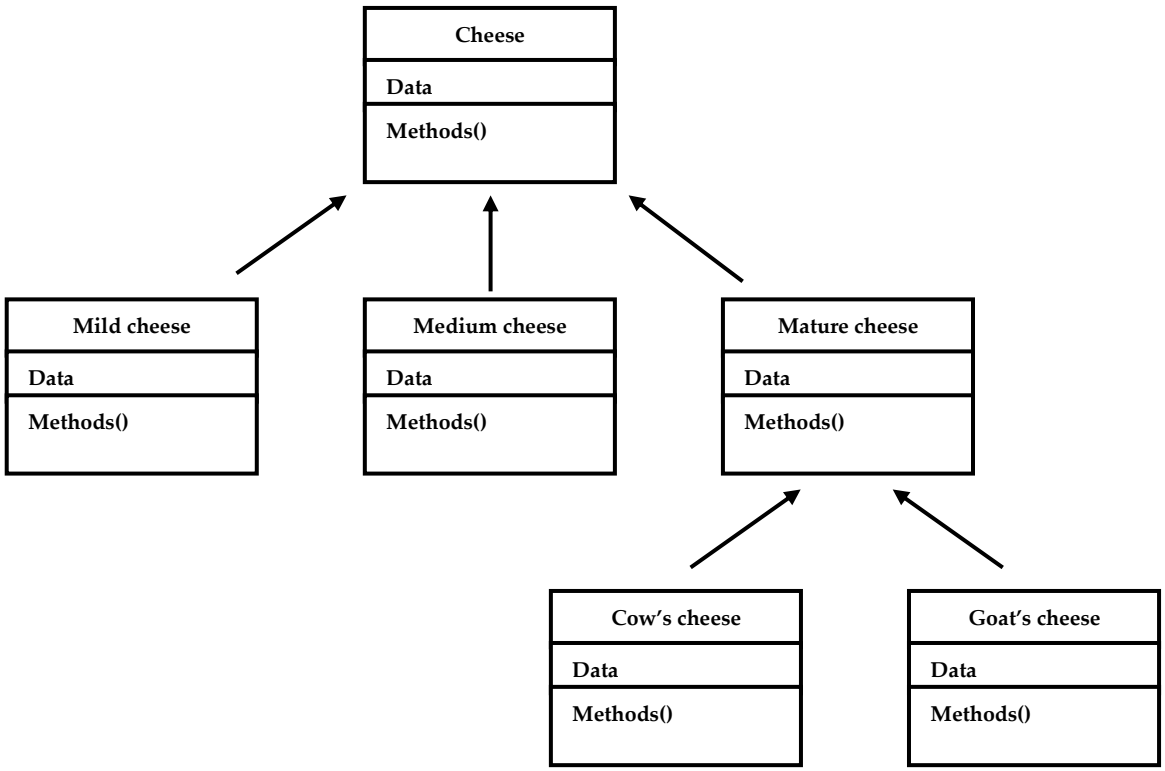
45.16 The advantages of the object oriented approach

Here are the advantages of the object oriented approach:

1) The object oriented approach uses objects. These are **real** and easier to view and understand than the conceptual approach of the structured programming.
2) Classes are **reusable** and you don't actually need to know how they work to reuse them! You just need to look at the **specification** for a class that gives a programmer who wants to use it information on how to do so.

3) If you can reuse classes, you can **save time and money** on developing new applications.
4) If you reuse code that is proven, any application you develop will have a **higher probability of working correctly**.
5) Classes have **fewer errors** in them compared to functions and procedures.
6) Classes are far more **robust** than modules of code produced in structured programming. They are less prone to errors because of the nature of **data encapsulation**. The data is accessible in very well-defined ways using the methods provided for the data.
7) If **modifications** to a program have to be made then it is far easier to do using the object oriented approach. You can, for example, remove a class and replace it without affecting the rest of the program.
8) It is relatively easy to **update** object oriented programs because you can add features by adding new classes that have been developed from already proven ones. This is why the property of **inheritance** is so important.

45.17 An inheritance diagram example



An inheritance diagram for the cheese stocked in the shop.

A shop stocks cheese. The cheese stocked can be one of three types: mild, medium or mature. Two types of mature cheese are stocked: cow's milk mature cheese and goat's milk mature cheese. Draw an inheritance diagram for this system.

The first thing to notice is that this shop sells cheese! Cheese is a class that will have properties. Any actual cheese will have data, such as a name and price per kilo and methods, such as a set and get method for the price. However, there are three types of cheese and each of these types will have their own particular properties ***in addition to*** those that the class Cheese has. There are two further classes because mature cheese can be broken down into cow's milk and goat's milk cheese. We have a total of six classes. The main class is Cheese. The superclass Cheese has three subclasses. One of those subclasses is itself a superclass to two subclasses. We can now draw an inheritance diagram for the cheese classes.

The class 'Mild cheese' has its own data and its own methods ***in addition to*** those it has inherited from the class 'Cheese'. 'Mild cheese' is a subclass of 'Cheese'. 'Mature cheese' is also a subclass of 'Cheese' and it inherits all of the data and properties of 'Cheese' as well. In addition, it has its own data and properties, those peculiar to 'Mature cheese'. 'Cow's cheese' inherits all of the data and properties of 'Mature cheese' (which includes the data and properties of 'Cheese'). 'Mature cheese' is a subclass of 'Cheese' but a superclass of 'Cow's cheese'.

45.18 More UML design - a case study

You are to design a class called RoadTransport. It will have three pieces of data: Length, which is a real number, NumberOfWheels, which is an integer, and NeedsToBeTaxed, which is Boolean. Each of these pieces of data will have a set method and a get method. There will also be two constructors. The first will be used when you want to create a new object that records the details about a new type of transport but you don't know any details about it. The second constructor will be used when you know the length, number of wheels and whether it needs to be taxed or not.

a) Sketch a URL for the above class.
b) Using examples from this scenario, clearly state what a class is and what an object is.
c) Using this scenario, explain the property of polymorphism in an OO language.
d) Using this scenario, state what is a constructor used for?
e) Using this scenario, explain what the property of encapsulation is and why it is so important in OO languages.

A new class is to be written called Car. It will be derived from the RoadTransport class. In addition to the data variables and methods inherited from the RoadTransport class, the Car class will also have a string for holding the Manufacturer of the car and a real number to show the EngineSize in litres. Both of these variables will need set and get methods and there will be only the default constructor.

f) Redraw the UML diagram to show both classes. Using labels, show which class is the superclass and which one is the subclass.
g) Using suitable names, add the data and methods to your UML diagram in the appropriate place.
h) Using this scenario, explain why the concept of inheritance is so important in an OO language. Ensure that you clearly explain the ways in which inheritance helps programmers produce new applications quickly and to a high quality (i.e. bug-free and reliable).

Another class is to be written that will also be derived from the RoadTransport class. It will be called Motorbike. It will have a string to hold the Manufacturer, a string to hold the Model, and a Date to hold the date that that model of motorbike was designed. The class will have a default constructor as well as a constructor that can be used when the Length, Manufacturer, Model and Date are known. In addition, there will be a set and get method for each piece of data.

i) Redraw the UML diagram to include the motorbike class. Label it as either a superclass or a subclass, as appropriate.

Another class is to be written that is to be derived from the class Car. It is to be called VintageCar. It will have a default constructor, and will have a string to hold the DesignersName and a variable called DateBuilt to hold the date it was built. These will need set and get methods.

j) Redraw the UML diagram as appropriate.
k) Write down a list of ***all*** the data variables and all of the set and get methods that VintageCar has.

Q1. What is bytecode and why is it used?
Q2. What is an applet?
Q3. Explain the terms 'class' and 'object' using examples.
Q4. Explain what is meant by 'data encapsulation'.
Q5. What are 'methods'?
Q6. What does a constructor method do?
Q7. What is meant by polymorphism?
Q8. What is meant by inheritance in an OO program?
Q9. What is meant by a 'sub-class' and a 'super-class'?
Q10. What are the key characteristics of an OO language?