Queen Mary
University of London

Procedural Programming
ECS401

Workbook
20016-17

Part 3
Unit 8-11

Paul Curzon

**School of Electronic Engineering
and Computer Science**

Unit 8 : References, the heap and the stack

Learning Outcomes

Once you have done the reading and the exercises you should be able to:

- compare and contrast the way integers and arrays (and other complex datatypes) are stored
- explain how information is stored in memory in the heap or stack
- explain how array information is passed to methods
- explain and compare and contrast pass by reference and pass by value

From last week …

Before moving on to this section you should be able to:

- write programs that are split into methods
- write programs that pass information to methods using parameters
- explain what is meant by a method and parameters
- explain what is meant by an abstract data type
- explain how abstract data types can be implemented from existing types
- write programs that create and use abstract data types
- discuss the importance of abstract data types

A fragment of code from last week

```
Queue q = createEmptyQueue(10);
System.out.println(firstInQueue(q));
q = joinQueue(q,"Alistair Brownlee");
System.out.println(firstInQueue(q));
q = leaveQueue(q);
q = joinQueue(q,"Mo Farah");
System.out.println(firstInQueue(q));
```

And then elsewhere in the program..definitions like:

```
public static Queue joinQueue(Queue q, String newentry)
   {
       if (q.numberqueueing < q.entries.length)  // Still
space
       {
            q.entries[q.numberqueueing] = newentry;
            q.numberqueueing = q.numberqueueing + 1;
        }
        return q;
   }
```

Strange things with arrays (and records, and strings)

- Strange things can appear to happen when you use arrays...
- To work out what's happening we need to understand how Java handles arrays and other similar datastructures.

Exercise 1

What does the following program fragment print?

```
int a = 1;
int b = 2;
b = a;
System.out.println(a);
System.out.println(b);
a = 5;
System.out.println(a);
```

```
System.out.println(b);
```

What does assignment *do*?

- For integers an assignment
  `b=a;`
- makes a *copy* of what was in one variable and puts it in the other.
- It takes the *contents* of a and puts it in b.
- a and b are still *different*.

Exercise 2

What does the following (very similar) program fragment print?

```
int [] a = {1};
int [] b = {2};
b = a;
System.out.println(a[0]);
System.out.println(b[0]);
a[0] = 5;
System.out.println(a[0]);
System.out.println(b[0]);
```

What does array assignment do?

- After an array assignment : b=a;
- a and b now contain the same things. But also...
- Change an entry of a ... and b changes.
- Change an entry of b ... and a changes.
- **So it looks as if a and b here refer to the *same* bits of storage - the same boxes.**
- but if a and b were *integers not arrays* they would refer to different bits of storage.

Storage for arrays

- So it looks as if there's something different about how arrays are stored.
- What's the story?
- I will start with a simplified view then give a bit more detail...

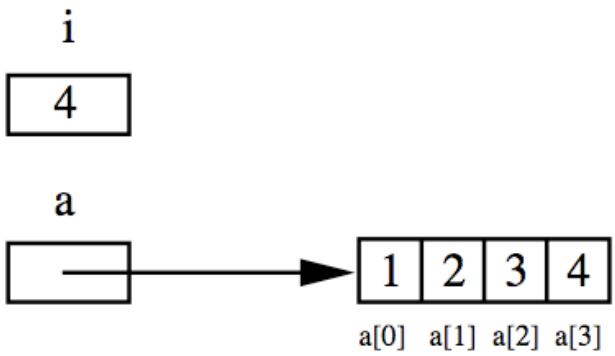
Variables as boxes

- We have talked of variables as being like boxes.
- Different variables are different boxes.
- Assignment just puts copies of values in the appropriate box.

Array variables are a *pair* of boxes

- An array is more like a pair of distinct boxes.
- The array variable does not actually hold the array.
- Instead it holds information (known as a *Reference*) of where to find the array data.

```
int i = 4;
int [] a = {1,2,3,4};
```



Explaining assignment

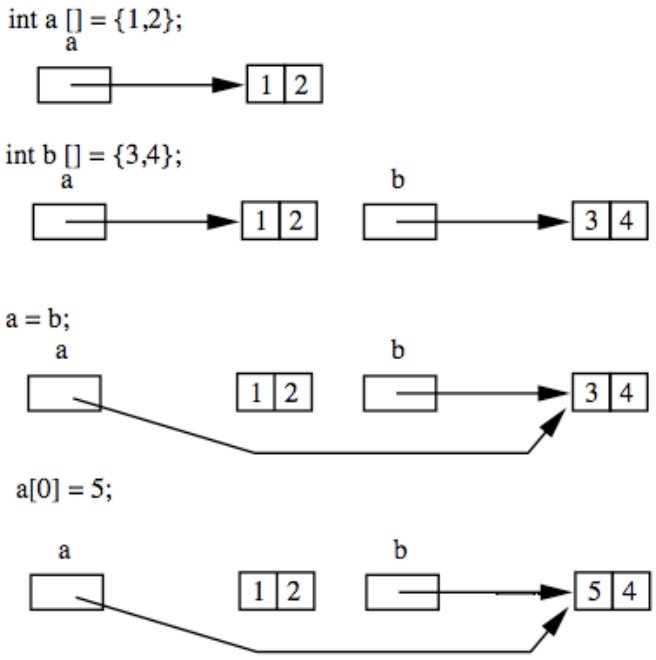
Using this picture of how arrays are stored we can explain the way assignment works:

- Assignments to the array variable, a, change what is in the box a.
- They make the arrow point somewhere else.
- Assignments to array *elements* of a *follow* the arrow to where the actual data is stored and change that.

Exercise

Trace the execution of the following code as a series of box and arrow pictures.

```
int a[] = {1,2};
int b[] = {3,4};
a = b;
a[0] = 5;
```



Exercise

Trace the execution of the following code as a series of box and arrow pictures.

```
int a[] = {1,2};
int b[] = {3,4};
int t[];
t = a;
a = b;
```

```
b = t;
```

Exercise

- Understanding how arrays are implemented also explains an otherwise weird result when they are tested.
- What does the following print to the screen?

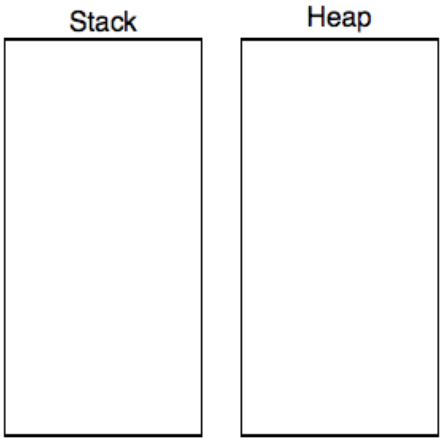
```
int a[] = {1,2};
int b[] = {1,2};
if (a==b)
System.out.println("a and b are equal");
else
System.out.println("a and b are NOT equal");
a = b;
if (a==b)
System.out.println("a and b are equal");
else
System.out.println("a and b are NOT equal");
```

Equality of arrays

- Initially the two arrays are not "equal" but then become "equal" after the assignment.
- So it looks as if `a==b` checks if a and b refer to the *same* bits of storage, not if the contents are the same.
- `a==b` tests the contents of the array variables (the references).
- If both a and b are pointing to the same place they are equal otherwise they are not.

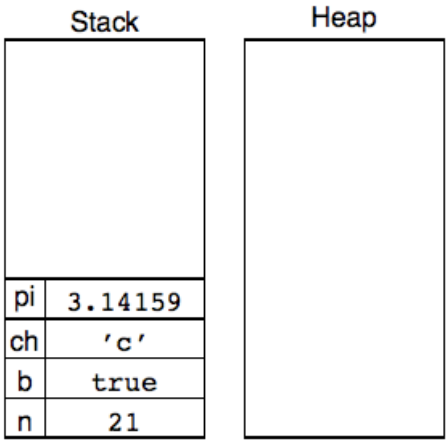
Java Storage

- Why are arrays stored like this?
- Java's storage is divided into two parts.
- They are called the *stack* and the *heap*.
- They contain different sorts of things.



The Stack

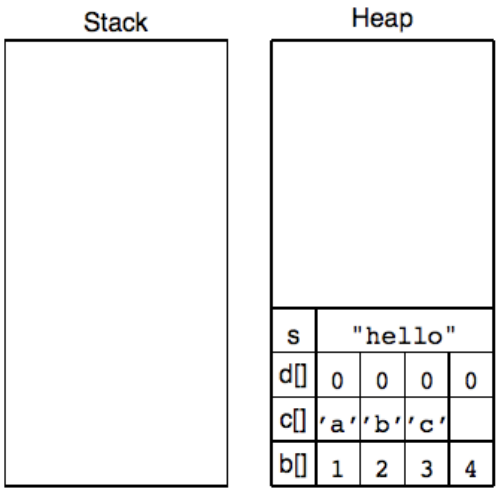
- The stack holds local variables and method parameters.
- When the program enters a block with a local variable declaration, storage for that variable is created on the stack.
- When the program leaves the block that storage is freed.



Note the names of variables are not stored in the stack or heap. (We include them in the diagrams as labels to show what is stored there). The Java system creates a lookup table that tells it where names are stored – in essence by the time a program is run the names have been replaced by the addresses in the code being executed.

The Heap

- The Heap holds arrays and other "objects" (e.g. Strings).
  - that is the actual data NOT the references to them.
- You will learn next semester how to create your own new kinds of objects in you programs



new

- The heap holds storage that is allocated through the use of the keyword "new", as opposed to some kind of variable declaration.
- (The use of new can sometimes be implicit. Eg String s = "hello"; does a hidden new)
- Things on the heap can live longer than the block in which they are created.

Array allocation

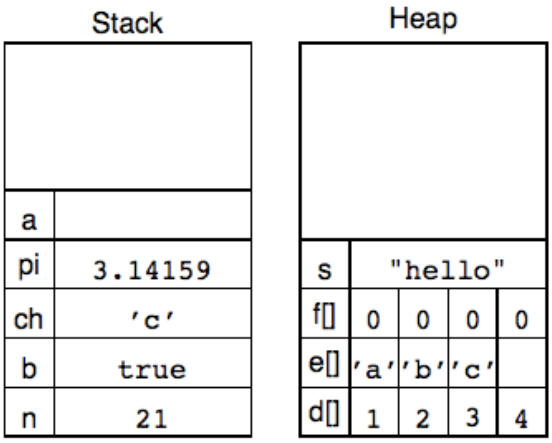
- So what happens when we declare and allocate an array?

```
int a[] = new int[2];
```

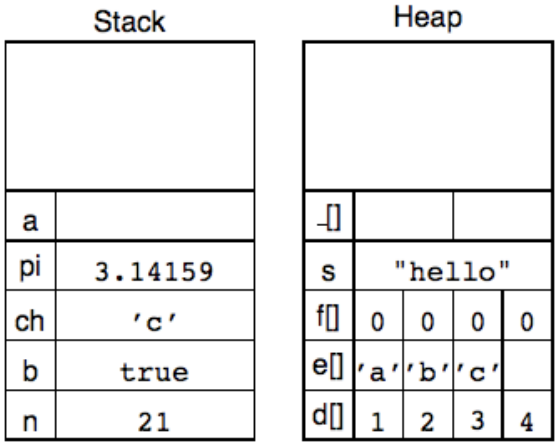
is the same as

```
int a[];
a = new int[2];
```

- int a[]; is the declaration of a new local variable a.
- The rules say it goes on the stack.



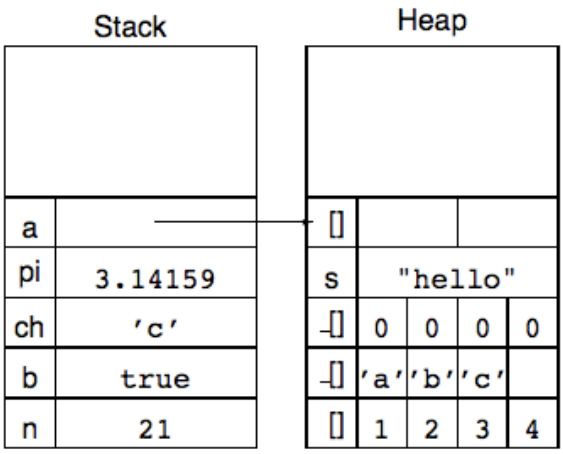
- new int[2] allocates a new array of length 2.
- The rules say it lives on the heap.



- But this means that the array variable is storage on the *stack*, and the array is storage on the *heap*!!!
- But fortunately we haven't finished.
- That is where the references come in.

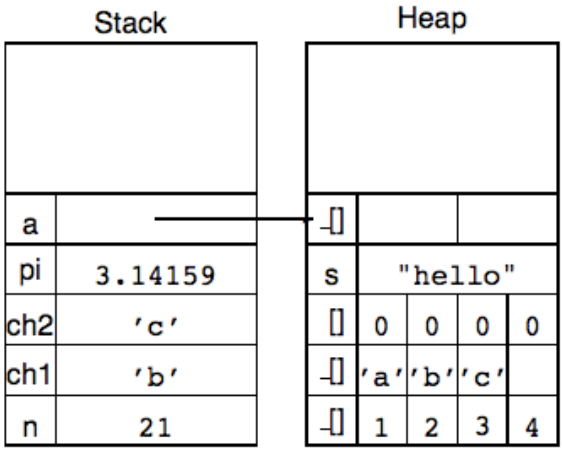
References

- a = new int[2]; puts something in the variable a.
- What?
- The location (address) of the array we've just created on the heap.



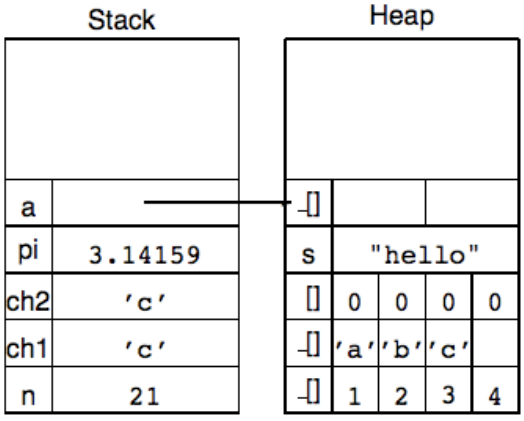
Strange?

- Now we can see why we got that strange behaviour from arrays.
- Let's look at assignment of, say, characters.

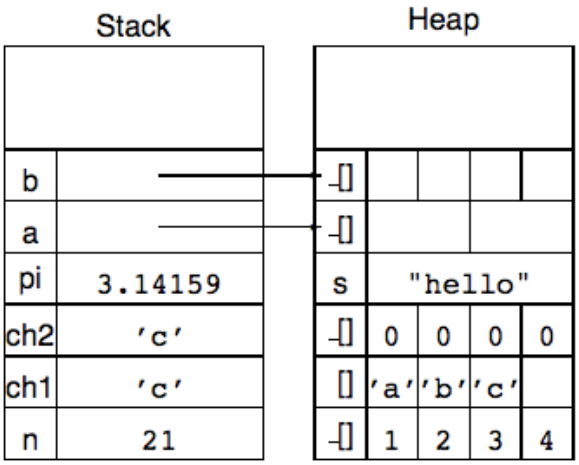



```
ch1=ch2;
```

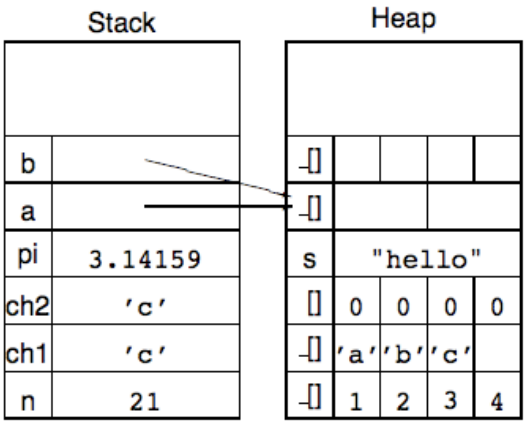
- puts the value (the contents) of ch2 in ch1.
- So we get



Now, if we have two arrays:



- `b=a;`
- takes the contents of a (the address) and puts it in b.
- So both a and b contain the location of the same array.



- This explains everything we observed in our experiments tracing code fragments.
- In Java things stored in stack variables are very simple:
  - either they are values of basic type (char, int, float, boolean, double, long, . . .)
  - or they are locations of things that are on the heap.
  - They are short and have known size.
- Things stored on the heap can be much more complicated (arrays, Strings, objects).

- This is a somewhat simplified account:
  - things on the heap can contain the locations of other things on the heap.
  - For example, if you have an array of arrays, or an array of Strings.

String equality again

- Strings are stored on the heap.

- Their equality is determined by that.
- Some storage optimisation happens so that you get equalities you shouldn't expect.
- See experiment 5.java on the web

Strings don't change

- By the way, . . .
- You can't change Strings in Java.
- That is you can't change the bit of the String that is on the heap, in the way that you can change a cell in an array.
- Operations that look as if they're changing Strings are actually creating new String values (and assigning their locations to String variables).

Records

- Records are stored in the same way to so essentially the same issues apply to them.
- You should get in the habit in your abstract data types of providing equality methods (essentially .equals) to compare if two abstract data types are the same by comparing the relevant values as opposed to the references.

Explaining Methods

- What does this have to do with methods?
- Quite a lot.
- Let's start with procedures…

Array parameters

Consider the two following methods:

```
static void array_inc(int[] array, int index)
{
      array[index]++;
}
static void int_inc (int x)
{
      x++;
}
```

What do they do?

Working it out

- Remember the rules about a call being equivalent to substituting the method body in its place...
- The rules say that a call to int inc:

```
int n=1;
int_inc(n);
```

- is equivalent to:

```
int n=1;
{ int x=n;
  x++;
}
```

This does not change the final value of x or n.

Let's step through it and see what happens on the stack.

In the following line executing is marked in bold at each step and what it does to the stack is then shown.

The first line executing:

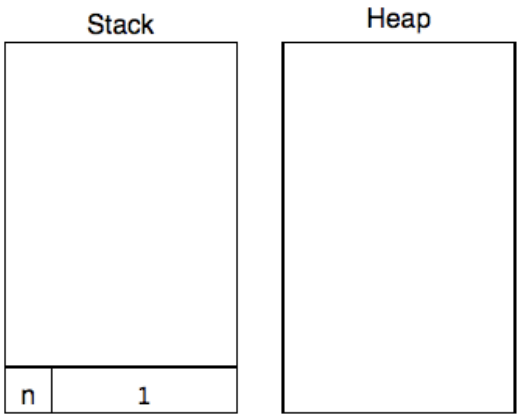
```
int n=1;
```

```
int_inc(n);
```

which is equivalent to:

```
int n=1;
{ int x=n;
  x++;
}
```

creates a new variable called n on the stack and stores 1 there. It thus has the following effect on the stack:



We continue execution. The call is made and the value that is in n passed to the method. It creates a new variable on the stack called x and stores the value passed in it.

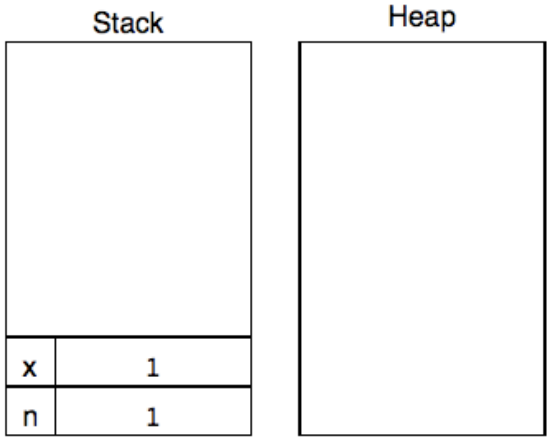
So…

```
int n=1;
int_inc(n);
```

which is equivalent to:

```
int n=1;
{ int x=n;
  x++;
}
```

has the following effect on the stack as the call starts.



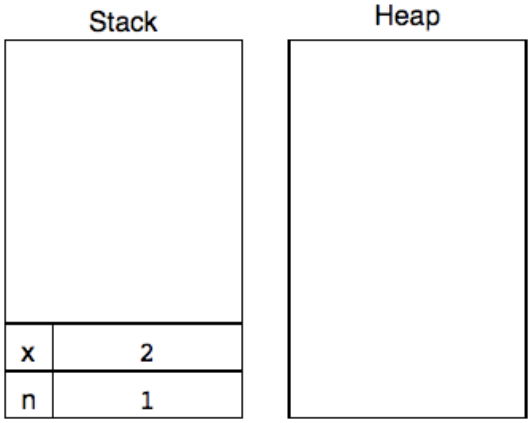
We now execute the commands in the body of the method. That just says add 1 to whatever is in x and store it back into x

```
int n=1;
int_inc(n);
```

which is equivalent to:

```
int n=1;
{ int x=n;
  x++;
}
```

The stack now looks like this:



Finally we get to the end of the method body. That ends the method…but that tells the system to tidy the stack up. Variable x was local to the method and when the method ends all its variables are cleared away. The stack returns to just having the variables on it that were there before the method was called.

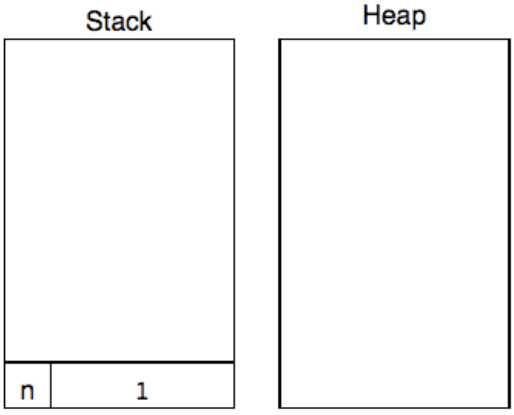
So in executing the final step of the method in

```
int n=1;
int_inc(n);
```

which is equivalent to:

```
int n=1;
{ int x=n;
  x++;
}
```

the final state of the stack (and heap) ends up as:



Array parameters

On the other hand let us now consider a call to array inc:

```
int[] a = new int[1];
a[0] = 1;
array_inc(a,0);
```

is equivalent to:

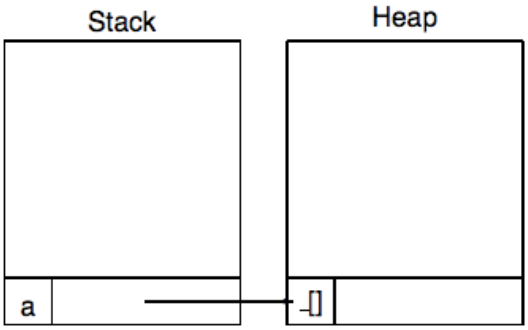
```
int[] a = new int[1];
a[0]=1;
{ int[] array = a;
  int index = 0;
  array[index]++;
}
```

- Even though it is doing something similar just on an array entry rather than on a simple integer variable, **this does change a[0].**
- Again the step being executed is marked in bold.
- First we declare a new array of size 1. It is allocated on the heap with a reference to it on the stack.

```
int [] a = new int[1];
a[0] = 1;
array_inc(a,0);
```

is equivalent to:

```
int[] a = new int[1];
a[0]=1;
{ int array[] = a;
  int index = 0;
  array[index]++;
}
```



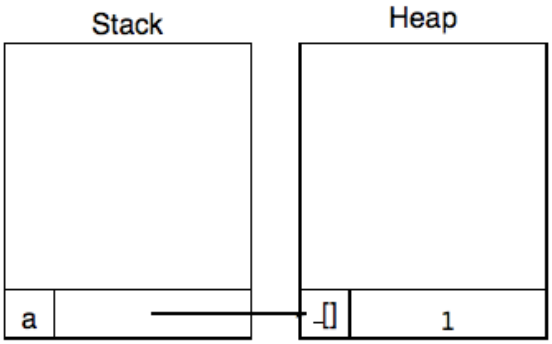
Next we update the cell of the array at position 0.

```
int[] a = new int[1];
a[0] = 1;
array_inc(a,0);
```

is equivalent to:

```
int[] a = new int[1];
a[0]=1;
{ int[] array = a;
  int index = 0;
  array[index]++;
}
```

This follows the pointer that is on the stack to the right place on the heap and updates the cell there.



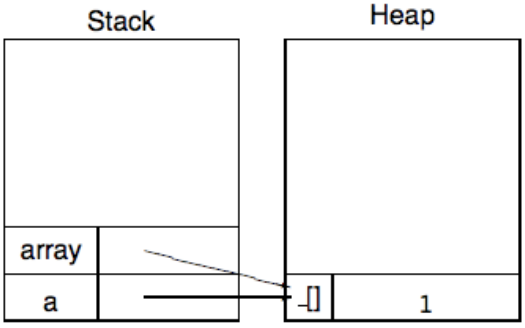
In the next step we do the call so set up new variables for the parameters

```
int[] a = new int[1];
a[0] = 1;
array_inc(a,0);
```

is equivalent to:

```
int[] a = new int[1];
a[0]=1;
{ int[] array = a;
  int index = 0;
  array[index]++;
}
```

We create a new array reference on the stack. The value of the array is actually just a reference so we copy that reference (the contents of variable a on the stack). That is we make a copy of the pointer and put it in the new position on the stack.

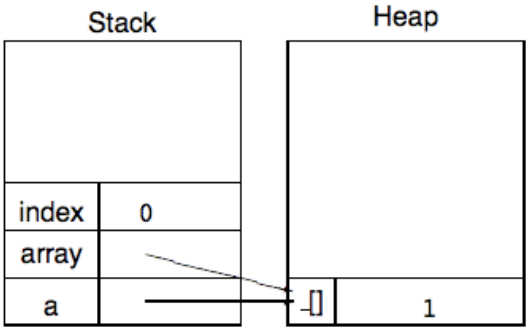



```
int[] a = new int[1];
a[0] = 1;
array_inc(a,0);
```

is equivalent to:

```
int[] a = new int[1];
a[0]=1;
{ int [] array = a;
  int index = 0;
  array[index]++;
}
```

The second parameter is just an integer so the value 1 is copied into the new stack position for variable index.



Next we update the array.

```
int []a = new int[1];
a[0] = 1;
array_inc(a,0);
```

is equivalent to:

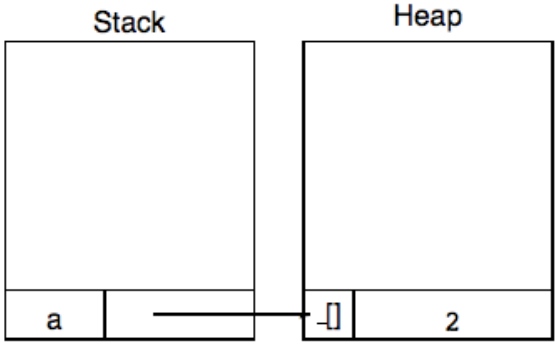
```
int a = new int[1];
a[0]=1;
{ int [] array = a;
  int index = 0;
  array[index]++;
}
```

What does this do. Index holds value 0 so we are saying

array[0]++;

We go to the stack and find array, follow the reference to the heap and then go to the 0th position on the heap.

We take the value that is there (It is the number 1) and we add one to it. We store the answer 2 back in the same position.



Similarly a method like:

```
static void int_swap(int x, int y)
{
      int tmp = x;
      x = y;
      y = tmp;
}
```

does nothing.

It only changes local values on the stack that are then thrown away.

Whereas

```
static void array_swap(int[] a, int x, int y)
{
      int tmp = array[x];
```

```
            array[x] = array[y];
            array[y] = tmp;
      }
```

does swap the elements of the array.

This means you can write procedures that sort an array (as we will see).

Arrays and functions

Finally you can of course still return arrays from functions.

```
      static int[] create(int lngth, int init)
      {
            int[] a = new int[lngth];
            for (int i=0; i<lngth; i++)
            {
                  a[i] = init;
            }
            return a;
      }
      ...
      int [] myarray = create(5,1);
      ...
```

Note that this creates the array, and initialises it.

You can run through the storage model for yourselves!

An new style of programming

This however gives a new style of programming with arrays and abstract data types.

Take our queue program from the last unit…

Since when a method changes a record (or array) passed to it we dont need an explicit return:

For our queue a fragment:

```
        Queue q = createEmptyQueue(10);
        q = joinQueue(q,"Alistair Brownlee");
        System.out.println(firstInQueue(q));
        q = leaveQueue(q);
        q = joinQueue(q,"Mo Farah");
        System.out.println(firstInQueue(q));
```

could be re-engineered to use procedures rather than functions

```
        Queue q = createEmptyQueue(10);
        joinQueue(q,"Alistair Brownlee");
        System.out.println(firstInQueue(q));
        leaveQueue(q);
        joinQueue(q,"Mo Farah");
        System.out.println(firstInQueue(q));
```

- Note we are changing the signature of the methods so our abstract data type doesn't help with this kind of change - we are changing the abstract data type!
- So abstract data types and abstraction don't solve all problems!
- We need new implementations of the methods eg

```
public static void joinQueue(Queue q, String newentry)
   {
       if (q.numberqueueing < q.entries.length)  // Still space
       {
            q.entries[q.numberqueueing] = newentry;
            q.numberqueueing = q.numberqueueing + 1;
        }
        return;
   }
```

- Similarly

```
public static void leaveQueue(Queue q)
   {
       if (q.numberqueueing != 0) // queue is not empty
       {
           for (int i = 0; i < q.numberqueueing; i++)
           {
              q.entries[i] = q.entries[i + 1];
                                   //Shuffle all entries down
           }
           q.numberqueueing = q.numberqueueing - 1;
       }
       return;
   }
```

As we aren't returning the queue we could now return the entry being removed and combine it with firstInQueue:

```
public static String leaveQueue(Queue q)
   {
       String first = q.entries[0];
       if (q.numberqueueing != 0) // queue is not empty
       {
           for (int i = 0; i < q.numberqueueing; i++)
           {
              q.entries[i] = q.entries[i + 1];
           }
           q.numberqueueing = q.numberqueueing - 1;
       }
       return first;
   }
```

With this implementation we could eg print the result of the call to leaveQueue as it now is the value removed. eg

```
System.out.println(leaveQueue(q));
```

so we would now have sequences like:

```
Queue q = createEmptyQueue(10);
joinQueue(q,"Alistair Brownlee");
System.out.println(leaveQueue(q));
joinQueue(q,"Mo Farah");
System.out.println(leaveQueue(q));
```

Example program

```
class queueprogram3
{
   public static void main(String []p)
   {
       Queue q = createEmptyQueue(10);
       joinQueue(q,"Alistair Brownlee");
       System.out.println(leaveQueue(q));
       joinQueue(q,"Mo Farah");
       System.out.println(leaveQueue(q));
       joinQueue(q,"Laura Trott");
       joinQueue(q,"Nicola Adams");
       joinQueue(q,"Amir Khan");
       System.out.println(leaveQueue(q));
       System.out.println(leaveQueue(q));
       System.out.println(leaveQueue(q));

       System.exit(0);
   }

    // Create an empty queue of a given size
   public static Queue createEmptyQueue(int size)
   {
       Queue q = new Queue ();
       String [] a = new String[size];
       q.entries = a;
       q.numberqueueing = 0;
       return q;
    }

   // add given string to the back of the queue
   // In this implementation it is just the nextfree position as
stored
   // Do nothing if queue full
   public static void joinQueue(Queue q, String newentry)
   {
        if (q.numberqueueing < q.entries.length)  // Still space
       {
            q.entries[q.numberqueueing] = newentry;
            q.numberqueueing = q.numberqueueing + 1;
        }
        return;
   }

   // remove the front of the queue
   // In this implementation it is just the first entry
   // return the element removed
   public static String leaveQueue(Queue q)
   {
       String front = q.entries[0]; // save so can return
       if (q.numberqueueing != 0) // queue is not empty
       {
           for (int i = 0; i < q.numberqueueing; i++)
           {
              q.entries[i] = q.entries[i + 1]; // Shuffle all entries
down
           }
```

```
            q.numberqueueing = q.numberqueueing - 1;
        }
        return front;
    }

} //END class QueueProgram

    /* ************************************************
        Create a new type (a record) called Queue that acts as a first
in first out queue
     */

class Queue
{
    String [] entries; // Where the queued values are stored
    int numberqueueing;   // The number of entries currently in the
queue
} // END class Queue
```

Unit 8 Programming Exercises

Arrays as References

Compile and run the programs from the ECS401 website – unit 8 area, experiment with them modifying them to do slightly different things (**experiment1.java-experiment5.java**)

- What does array assignment do? Try adding different assignments and printing of the arrays at the end of the program. Explain what is going on!
- Compare with what happens with integer assignment
- Arrays when we change single elements isn't the same as with integers - why not?
- What about testing for equality of arrays
- So what happens with strings - they are similar to arrays, stored as references (see experiment5.java)
- Now try similar experiments with records (eg the Student record we use originally). Try making assignments to records and testing their equality. Are they the same as integers? Or Arrays? Or Strings? Or do they behave differently again?

Fill array

Write a method to fill an array with even numbers from 0. It should expect to be passed a newly declared array to fill as an argument. Write a main method to test it.

Double array

Write a method that is passed an integer array as an argument and changes every value in it to double the original value. Write a main method to test it.

Arrays versus integers

Write a method that is passed two arguments: an integer array and a single integer, both passed as variables. The method should set both its local parameters to zero – ie zero all entries in the array and set the local version of the integer variable. Write a

main method that shows that whilst the array is zeroed the integer variable is unchanged by the method.

Go back to all previous programs you have written that used arrays and/or abstract data and modify them to no longer return the array or abstract data types from the methods they were passed to, using call by reference to side effect the changes. See **studentgettersetter3.java** as an example. It updates **studentgettersetter2.java** from unit 3 to this style of programming with records (and arrays). See how the records are no longer returned from getter methods which become procedures rather than functions.

Other Exercises

Compare and contrast the two different styles of manipulating abstract data types - returning them and reassigning versus making the change by side effect. What are the advantages and disadvantages of each?

Unit 8 Exam Style Question: Arrays, Method Call

Question 1 [25 marks]
a. [6 marks]
Explain what is meant by
i) the stack
ii) the heap
with respect to the Java runtime environment.

b. [9 marks]
Compare and contrast the following, **illustrating** your answer with examples:
i) method call with an array parameter
ii) method call with an integer parameter

c. [10 marks]
Write a program that allows a user to input a series of names and telephone numbers. Once the names and numbers have been input the program should allow the user to repeatedly search for numbers of people with a name given by the user. The program should use a separate method for inputting the details and for doing the search, both called from the main method. The arrays used to store the data should be declared in the main method and passed to the other methods as arguments.

Unit 9 : Recursion

Learning Outcomes

Once you have done the reading and the exercises you should be able to:

- write programs using recursion
- implement divide and conquer algorithms
- explain concepts related to recursion and divide and conquer algorithms

From last week …

Before moving on to this section you should be able to:

- compare and contrast the way integers and arrays are stored
- explain how information is stored in memory in the heap or stack
- explain how array information is passed to methods
- explain and compare and contrast pass by reference and pass by value

Recursive functions

- A function or method is called *recursive* if during a call to the function it is possible to get another call to the *same* function.
- *Recursion* is fine when the recursive call has arguments that are smaller in some sense (not always an obvious one).
- It causes your program to loop.

Step Case and Base Case

- Recursion is a form of repetition.
- It is an alternative to while loops.
- A while loop has a test to decide when to stop, something that says do it again and a thing to do repeatedly.
- A recursive function has the same bits; put together in a different way.

Sum-to-n

Here is a while loop way of adding the first n integers for any n: 1+2+3+ …+ n

```
static int sumton(int n)
{
     int sum = 0;
     while (n != 0)
     {
          sum = n + sum;
          n = n - 1;
     }
     return sum;
}
```

Sum-to-n

Here is the same thing done in a different way – using recursion.

The function calls itself passing a new value of n.

Notice there is no loop insight!

```
static int sumton(int n)
{
     if (n==0)
          return 0;
     else
          return (n + sumton(n-1));
}
```

The *base case* is the simple terminating case.

if (n==0) return 0;

The *step case* is the recursive call.
return (n + sumton(n-1));
It moves the solution closer to the base case, and calls the function on that simpler version of the problem.

Exercise

Write a program that multiplies the first n numbers using recursion.

Recursion and non-termination

- A problem with recursion is that sometimes it's possible to get into an infinite loop.
- This can't happen provided the function only calls another function if either
  - it's further down the food chain or
  - it's on a smaller input
- So in some sense it's always a "smaller call", and eventually things have to bottom out (unless you supply infinite input).
- That's not the case with every recursive function. . .

Recursion A Warning

- It's OK to have functions call themselves.
- But you MUST leave a get-out clause, or your program will go into an infinite loop.
- Forgetting this is the most common cause of bugs in writing recursive programs.

Unwinding the recursion

- It is important to understand how the recursive calls all return at the end. Take this method loop.
- Dry run it to see what it does when called with loop(10) before you read on.

```
public static void loop (int n)
   {
      report("Entering.." + n);
      if (n==0)
        return; // base case
      else
      {
        report("Do some work .." + n);
        loop(n-1); // step case
      }
      report("Leaving .." + n);
      return;
   }

   public static void main (String param[])
   {
      report("starting...");
      loop(10);
      report("finished!");
   }
```

What does it do?

- Here is the output it prints

```
starting...
Entering..10
Do some work ..10
```

```
Entering..9
Do some work ..9
Entering..8
Do some work ..8
Entering..7
Do some work ..7
Entering..6
Do some work ..6
Entering..5
Do some work ..5
Entering..4
Do some work ..4
Entering..3
Do some work ..3
Entering..2
Do some work ..2
Entering..1
Do some work ..1
Entering..0
Leaving ..0
Leaving ..1
Leaving ..2
Leaving ..3
Leaving ..4
Leaving ..5
Leaving ..6
Leaving ..7
Leaving ..8
Leaving ..9
Leaving ..10
finished!
```

- Notice how none of the methods are exited (so the leaving message printed) to the end.
- That is the recursion unwinding.
- The calls called another copy of themselves over and over again without any returning until the base case was hit.
- Copies of any variables declared in a recursive method like this are stored until the recursion unwinds (as when that invocation of the method is finally returned to it needs its copies of the variables back).

Tail Recursion

- One form of recursion is called **tail recursion.**
- **Tail recursion is generally bad programming!!**
- It is a recursion where the very last thing to happen in a method is a recursive call
- It means the recursive call is being used as a goto in that it just jumps to the start (and doesn't actually need to come back as there is nothing to do when it does.
- It does come back though! (as it was a method call and method calls return).

Why bad?

- This is bad for several reasons

    - it is unnecessary
    - setting up the call takes (so wastes) time
    - each call eats up (wastes) space on the stack storing local variables that will never be used again (as it does nothing with them when it returns).
    - You could run out of stack space so the program could crash
- If you end up writing a tail recursive method you would have faster code with a while loop
- However sometimes it makes life very easy in writing a program as we will see…
- Some compilers for some languages spot it and create code that doesn't actually recurse - turning it in to a jump
- You cannot rely on it though.

Example

- Here is a BAD method that uses tail recursion to print numbers from a given number down.

```
public static void printNto0(int n)
{
   if (n == 0)
        return;
   else
       System.out.println(n);

   printNto0(n-1);
   return;
 }
```

- If you ever find yourself putting a recursive call immediately before the return statement of a method ask yourself why you are doing it!
- If you are just trying to jump to the start, think again

Sumton was not tail recursive

- Notice that our sumton method above is NOT tail recursive
- After returning each call still has work to do adding n on to the result returned

```
return (n + sumton(n-1));
```

Head Recursion

- Head recursion is the 'opposite' of tail recursion.
- The recursive call is done BEFORE the method does its main work other than to work out eg that recursion is needed
- It saves the state of the method on the stack
- Sumton was an example. Here is another:

```
public static void print0toN(int n)
{
   if (n == -1)
        return;
   else
   {
       print0toN(n-1);
       System.out.println(n);
       return;
    }
 }
```

Divide and Conquer

- Divide and conquer is one of the most common and best-known problem-solving strategies.
- It has a recursive flavour.

How Divide and Conquer works

This is how it works:

To solve a problem:
if the solution is immediate, produce it
otherwise, we have a big problem:
break the big problem into a number of **similar but smaller** problems
solve the smaller problems **in exactly the same way**
use the solutions to the small problems to build a solution to the big one.

Divide and Conquer

In pseudo-code the code for solve(problem) might be:

```
solve (problem)
   if (solution immediate)
        return immediate-solution;
   else {
        break problem into small-problems;
        for each small-problem in small-problems {
            small-solution = solve(small-problem);
        }
        construct full-solution from small-solutions;
        return solution;
    }
```

Note the recursive call to solve in the above.

Binary Search

- Binary Search is a good example of a recursive divide and conquer algorithm.

```
To search an array that is sorted looking for key.

    if the middle value in the array is the key
    then return FOUND at that position
    else if the mid point of the array is smaller than key
    then
           return the result of doing a
                 binary search on the lower half of array,
    else
           return the result of doing a
                 binary search on the upper half of array,
```

- Note this only works if the array is sorted into order first. We've assumed sorted in to ascending order but it only takes a small change if the array starts sorted in descending order.
- Oops but notice it is tail recursive!
- This is an example where it is just very natural.

Exercise

Modify the algorithm to find something in an array that is sorted in descending order

Towers of Hanoi

There's a mythical story about Buddhist monks in a temple at Hanoi.
The monks have a number of heavy gold discs, each with a hole in the middle, and each of a different size. There are three pillars, and each disc is fitted onto one of the pillars. The monks are moving the discs from pillar to pillar, according to the following rules:

- only one disc can be moved at a time
- the disc must be moved from one pillar to another,
  it cannot be held or put on the ground
- no disc can be put on top of a smaller disc, only a larger one

When the monks get all the discs onto the final pillar, the world will come to an end.
Fortunately there are a lot of discs!

Towers of Hanoi

The problem is to write a computer program that will simulate a solution, either by displaying it or writing a list of instructions for the monks.

Call the pillars A, B and C.
If you only have to move one disc, the solution is immediate.
If you have n discs:
Move the top n -1 from A to B using C as a spare.
Then move the bottom disc from A to C.
Finally move the top n -1 from B to C using A as a spare.

This is a recursive algorithm (see code).

How long have we got?

If you look at the code you'll see that the number of moves taken to solve an n-disc Tower of Hanoi satisfies the following relation:
moves(n) = 2 * moves(n -1) + 1
This is exponential in the number of discs.
There were a lot of discs!
Executive summary:
heat death of the universe happens first!

Unit 9 Example Programs and Programming Exercises

Only try these exercises once you feel confident with those from the previous week/ unit.

Recursion Example Programs

Compile and run the following programs ECS401 website – week 9 area, experiment with them modifying them to do slightly different things.

- A very simple recursive counter Try editing out the report statements (they are there just to show you how the methods get called over and over and when they return. Explain what is going on
- A recursive solution to the Towers of Hanoi
- A dangerous non terminating recursive program
- Another dangerous non terminating recursive program

Recursive sum to n

Write a recursive version of a function to sum the first n integers for a given n: sum (n +1) = n + sum(n). Embed it in a test program to ensure it works.

Factorial, exponential, fibonacci, log
Many mathematical functions have simple recursive algorithms. Here are the usual culprits. Implement them:

factorial: if n=0 return 1, otherwise let m = factorial(n-1) and return n*m;

exponential (mn): if n=0 return 1, otherwise let e = exp(m,n-1) and return m*e;

fibonacci: if n=0 or 1 return 1, otherwise let a = fibonacci(n-1), b=fibonacci(n-2) and return a+b;

log: if n=1 return 0, otherwise let m = log(n/2) and return m+1;

The control flow in all these is quite simple and you can easily find efficient algorithms that use while loops. That is not so true for ...

Euclid's algorithm for greatest common divisor
This is a cunning algorithm for producing the greatest common divisor of two numbers by repeated subtraction:

```
gcd (0,n) = 0
gcd (m,n) = m,  if m=n
gcd (m,n) = gcd(n,m), if n<m   //ensures m,n right way round
gcd (m,n) = gcd(m,n-m)
```

The last clause is the one that does most of the work: the repeated subtraction bit.

Binary search (divide and conquer)
Write a recursive version of binary search.

```
public static int search (int target, int[] list, int from, int to)
```

should search the segment of list from from to to for target, returning the index of a place where target is, if it is in the list, and -1 if it is not.

Test it on arrays of data you set searching for things both that are there and not, ensuring it finds things that are there and reports them missing if not

Write your own version, but once completed compare it with the binary search program on the website.

Unit 9 Exam Style Question: Recursion

Question 1 [25 marks]

a. [6 marks]

Explain what is meant in programming by:

i) a base case

ii) a step case

b. [9 marks]

i) Write an equivalent method to the following that does not use a recursive call.

```
public static int f(int n)
{
   if (n == 0)
   {
           return 1;
   }
   else
   {
        return (n * f(n-2));
   }
}
```

ii) Explain how the above recursive program could fail to terminate.

c. [10 marks]

Discuss the truth or otherwise of the statement "Recursion is not necessary".

Unit 10 : Sorting and Matrices

Learning Outcomes

Once you have done the reading and the exercises you should be able to:

- explain different ways to sort arrays of data into order
- write programs to sort arrays
- write programs containing matrices
- explain how to declare and use matrices

From last week …

Before moving on to this section you should be able to:

- compare and contrast the way integers and arrays are stored
- explain how information is stored in memory in the heap or stack
- explain how array information is passed to methods
- explain and compare and contrast pass by reference and pass by value

Searching from a previous week

```
for (int i=0; i<length; i++)
{
   if (a[i]==target)
   { System.out.println(i); break; }
}
```

- We also saw you can search more quickly if the data was sorted!

Arrays

- Arrays are used to handle bulk data.
- Typical things you do with arrays are:
- apply some operation to a single element
- apply some operation to each element of the array
- find some particular element
- sort the elements into some order

Sorting Arrays

- There are lots of sorting algorithms.
- Which one is best depends on the kind of data you have (size, how nearly sorted, how easy to copy, . . .)
- Some algorithms change the array so that it becomes sorted (sorting in place)
- Others give you a copy of the array in sorted order.
- By using Divide and Conquer we can get very fast sort algorithms

Sorting: bubblesort

A rough outline of how bubblesort works

- Go through the whole array swapping pairs that are out of order (a pass)
- Go back and do it again.
- Keep going (doing passes) until the array is sorted.

This is essentially a loop inside a loop as follows. I will give an inefficient version of the code first then talk through how it works. We will then look at how to improve it.

Naïve Bubblesort

```
public static void naivebubblesort(int[]array)
{
  for (int pass=1; pass <= length-1; pass++)
  {
      for (int i=0; i < length-1; i++)
      {
```

```
            if (array[i] > array [i+1])
            {
                  swap(array, i, i+1);
            }
      }
  }
}
```

Developing a Bubble Sort method

Bubblesort involves doing lots of passes over the array. Let's program a single pass.

- We need to run from one end of the array to the other
- That is just a for loop processing the elements one at a time
- At each position we compare the current entry with the one in the next cell along.
- If they are out of order we swap them

```
for (int i=0; i < length-1; i++)
{
      if (array[i] > array [i+1])
      {
            // swap them
      }
}
```

How do you do a swap?

- We've seen that in previous lectures

```
int tmp = array[i+1];
array[i+1] = array[i];
array[i] = tmp;
```

- We also saw last week how to package that up into a method called swap (left here as an exercise
- So doing one pass gives us:

```
for (int i=0; i < length-1; i++)
{
      if (array[i] > array [i+1])
      {
            swap(array, i, i+1);
      }
}
```

Note that to swap two positions in an array you have to give the array and the positions as arguments – you can't pass the values (Ie swap(array[i],array[i+1]) just wouldn't work as the method doesn't then have access to the array.

How much is sorted

- Will that guarantee to sort them? No
- Let's think about what we know.
- As we do the pass we always move the biggest along towards the end
- If we come across a bigger element we leave the other behind and move this one.
- Eventually we are at the end and have carried the biggest with us.
- So after one pass we are sure that one element (the biggest) is now in the right place (the end)
- The others could still be totally shuffled though.
- We need more passes!

How many passes?

- If we do it again we will take the next biggest with us towards the end, so it will end up in the right place (second from the end)
- A third pass and the third biggest will end up in the right place.
- How many passes do we need altogether?
- If we had 5 elements to sort then after 4 passes the 4 biggest would be in the right place. That would mean the fifth would be too (no where else for it to go)
- So for an array of length 5 we need 4 passes
- Similarly for an array of length 100 we need 99 passes.
- For an array of length n, we need n-1 passes.

How do we do lots of passes?
We put the code we saw already into a for loop (that how to repeat things)
That gives us a simple version of bubble sort.

Naïve Bubblesort

```
for (int pass=1; pass <= length-1; pass++)
{
      for (int i=0; i < length-1; i++)
      {
            if (array[i] > array [i+1])
            {
                  swap(array, i, i+1);
            }
      }
}
```

- Notice this is a nested loop.
- It's the same structure of program used to print a rectangle of stars!

Now to make it a method we can reuse we need the method header

```
public static void naivebubblesort(int[]array)
{
  for (int pass=1; pass <= length-1; pass++)
  {
      for (int i=0; i < length-1; i++)
      {
            if (array[i] > array [i+1])
            {
                  swap(array, i, i+1);
            }
      }
  }
}
```

It actually changes the array passed to rather than return a value so it's return type is void. It needs to be passed the array to sort of course and as we have to give it a type (here it is an integer array) that means this method can only sort arrays of that type.

A better version of bubblesort

- If you think about it we are doing a lot of wasted effort.
- Every pass goes all the way to the end of the array comparing things we know are in the right place.
- Why do that? If we didn't our sorting algorithm would be quicker!
- Each pass can stop one place earlier than the previous one

A better version of bubblesort

Let's assume the array is of length 5

```
for (int pass=1; pass <= 4; pass++)
      for (int i=0; i < 4; i++)
            if (array[i] > array [i+1])
                  swap(array, i, i+1);
```

On pass 1 we want to count all the way up to 4 (but 4 is just 5-1)
On pass 2 we want to count all the way up to 3 (but 3 is just 5-2)
On pass 3 we want to count all the way up to 2 (but 2 is just 5-3)
On pass 4 we want to count all the way up to 1 (but 1 is just 5-4)
What can we write as the test of the inner loop to make this happen?
We could add a variable that we change each time to count 4,3,2,1 but we don't need to.
The clue is in that subtraction.
Every time it is just (5-pass) as pass counts from 1 to 4

```
for (int pass=1; pass <= 4; pass++)
      for (int i=0; i < 5-pass; i++)
            if (array[i] > array [i+1])
                  swap(array, i, i+1);
```

If the array has length n it would be (n-pass)

```
for (int pass=1; pass <= length-1; pass++)
{
      for (int i=0; i < length-pass; i++)
      {
            if (array[i] > array [i+1])
            {
                  swap(array, i, i+1);
            }
      }
}
```

Bubble Sort program - a more sophisticated version still
We can do even better… It's left as an exercise for you to work out why this version is sometimes faster still, and how it does it.

```
boolean sorted=false;
while (!sorted)
{
      sorted = true; // array potentially sorted
      // traverse array switching ill-ordered pairs
      for (int i=0; i < length-1; i++)
      {
            if (array[i] > array [i+1]) // swap them
            {
                  int tmp = array[i+1];
                  array[i+1] = array[i];
                  array[i] = tmp;
                  sorted = false; // array wasn't sorted
            }
      }
}
```

Sometimes this is faster, but in fact sometimes this version can actually be slower – it can end up doing a whole pass more than it needs to.
Sorting: insertion sort
There are lots of different sort algorithms.
Here is a different way to sort

- Find the smallest element.
- Put it in first place.
- Find the second smallest element.
- Put it in second place.
- Keep going.

Sorting: insertion sort

- Put another way (recursively):
  Find the smallest element.
  Put it in first place.
  Sort the rest of the array (using insertion sort).

Exercise:

Write a method that does insertion sort.

Divide and conquer: Sorting

- With searching we saw that divide and conquer could make it massively faster.
- Can we do the same trick with sorting?
- Yes we can.
- Suppose you have a long array that you want to sort.
- Divide and conquer says:
  split it into smaller arrays and sort those (in the same way)
  then combine the sorted arrays into the sorted large array.

Divide and Conquer: Mergesort

- We could just split the array in the middle.
- Then we'd end up with two sorted arrays, and we'd have to shuffle them together.
- This leads to mergesort.
- Splitting the arrays is trivial, recombining them after sorting takes time.

Mergesort:
  split array in two halves
  Mergesort first half
  Mergesort second half
  Shuffle (merge) the two sorted halves together

Divide and Conquer: Quicksort

- In quicksort we also split the array.
- But we put all the elements smaller than some suitably chosen size at one end,
- and all the elements bigger at the other first.
- That way we don't have to do any shuffling at the end.

Quicksort:
  choose an element to partition round
  move elements smaller than it to its left
  move elements bigger than it to its right
  Split the array into two parts around the final position of the chosen element
  Quicksort first part
  Quicksort second half
  Put the two sorted halves together

Quicksort

- Quicksort is a famous algorithm for sorting arrays invented by Tony Hoare.

- It is one of the algorithms most commonly used in real systems.
- It is generally very fast.
- We gave a rational reconstruction, but not its most efficient form.

Multi-dimensional arrays / Matrices

- Sometimes the information you want to work with comes in two dimensions or more:
    a. pixels on a screen
    b. squares on chess board
    c. volume elements in space
- Then you need a two or three dimensional form of array.

Java handles this by allowing arrays of arrays

```
int matrix[][] = new int[m][n];
```

This lets you store a matrix as an array of rows (or an array of columns).

M00 M01 M02 . . . M0m

M10 M11 M12 . . . M1m

```
.........
```

Mn0 Mn1 Mn2 . . . Mnm

```
int M[][] = new int[n][m];
```

M[1][] is a row (Row order).

M[][1] is a column (Column order).

Processing a matrix

```
int M[][] = new int[n][m];
for (int i=0; i<n; i++)
{
 for (int j=0; j<m; j++)
 {
   PROCESS (a[i][j]);
  }
}
```

Processing a matrix – printing every element a row at a time

```
int M[][] = new int[n][m];
for (int i=0; i<n; i++)
{ for (int j=0; j<m; j++)
 {
   System.out.println(a[i][j]);
  }
}
```

Unit 10 Example Programs and Programming Exercises

Only try these exercises once you feel confident with those from the previous unit.

Sorting

Compile and run the following programs ECS401 website – week 10 area, experiment with them modifying them to do slightly different things.

- A Bubble sort program that constantly checks if sorted
- The same algorithm with a trace statement. Look at the pattern of how it sorts. Try changing the values in the array.
- Quicksort

Bubblesort

This 'bubble sort' variant algorithm repeatedly scans the array from left to right interchanging adjacent elements that are out of order. The scan is repeated until no out of order pairs are found on a scan.

Write a program that 'bubble sorts' into **descending** order an array of size 20 filled with random integers. Use a for loop inside a for loop to do this. Print out the array at the beginning and after every exchange of neighbouring elements. This will enable you to see the bubbling and will also help with debugging.

HINT: You will need a swap method to swap adjacent values. Get that working first. Then write a method that does a single pass of an array swapping adjacent values. After it has run the array will not be sorted but the biggest value should at least be in the correct position. Once that is working put it in a loop to do that pass repeatedly until it is sorted.

As an extension to your program, keep a count of both the number of comparisons and swaps of array elements that are made. Print out these counts at the end. Compare the counts with results from other algorithms below.

Improved Bubblesort

Write an improved version following the discussion in the lecture that doesn't run to the end of the array every pass, so avoids pointless comparisons.

Insertion/Selection sort

Write a program that 'insertion sorts' into non-descending order an array of size 20 filled with random integers. Print out the array after every exchange of elements.

As an extension to your program, keep a count of both the number of comparisons and swaps of array elements that are made. Print out these counts at the end.

Binary Search with sorting

Write a binary search program (see exercise from earlier week) that contains a sort method to first sort an array and a search method that does the actual searching of that array. Your program must sort any data first before it searches through it.

Dictionary ordering

Write a program that inputs words (at least 20) and stores them in a String array. Then sort the array into dictionary order using some sorting algorithm. To compare values of type String for dictionary ordering you will need to use the String method compareTo, e.g. s.compareTo(t). This returns the value 0 if the strings are equal, less than 0 if s comes before t in dictionary order and greater than 0 if s comes after t.

As an extension, read ahead to find out about file and read the word from a file first and save the sorted dictionary to a new file. You may use Hansen's method readword for reading words from the input dictionary.

Spell checker with an ordered dictionary

Redo the spell checker exercise from an earlier week with an ordered dictionary of data (since the above exercise enables you to generate one). When searching the array representing the dictionary you should now use linear search but make use of the dictionary ordering by making sure that the search terminates as soon as you know that the word is not present in the dictionary (i.e. when it comes before the word being examined in the dictionary). This will, on average, halve the search time for a word.

There are of course even better search methods for ordered lists like binary search. It doesnt just half the search time. Remember the guessing game from week 5? The method we indicated could be used by the player in the game was 'binary search'. Here binary search would work as follows. Compare your word against the middle word in the dictionary. If it comes before that word then repeat the search on the first half of the dictionary, if it comes after the middle word then repeat on the second half. And so on.... For a dictionary with a million words finding a given word (or finding its absence) will never take more than 20 comparisons using binary search whereas it would on average take half-a-million comparisons using the naive linear search method.

Mergesort

Write a recursive mergesort method.

Write a non-recursive version of mergesort. It can be done efficiently using two arrays copying from one to the other.

Unit 10 Exam Style Question: Sorting

Question 1 [25 marks]

a. [6 marks]

Explain how with simple changes the following algorithm to sort data can be improved, **justifying** your answer

```
for(int pass = 0; pass <= array.length-2; pass++)
   for(int i = 0; i <= array.length-2; i++)
   {
     if (a[i] < a[i+1]))
     {
           swap(a, i, i+1)
     }
   }
```

You may assume that swaps the values in an array at the given positions.

b. [9 marks]

Compare and contrast:

i) bubble sort

ii) mergesort

c. [10 marks]

Discuss the truth or otherwise of the statement '*Searching is faster if data is sorted'*.

Unit 11 : File Input and Output

Learning Outcomes

Once you have done the reading and the exercises you should be able to:

- write programs that store information in files
- write programs that retrieve information from files
- explain how data can be stored persistently
- explain different ways characters are stored

From last week …

Before moving on to this section you should be able to:

- explain different ways to sort arrays of data into order
- write programs to sort arrays
- write programs containing matrices
- explain how to declare and use matrices

Files

- Files are the operating systems mechanism for permanently storing data.
- ...storing data persistently so that it survives a program terminating
- Files are kept in directories, or folders. Together these make up a file system.
- Each operating system has its own file system.(Usually more than one).
- But (these days) each operating system also has to be able to handle “foreign” file systems as seamlessly as possible.

Data files and Text files

- Files can contain either data (data files) or text (text files).
- You can read text files in a text editor, but you cannot usually read data files.
- Each kind of file keeps its information in a particular layout (or format).
- Under Unix you can get a good guess at what kind of information (really what kind of format a file is in) by running the file command.

Text Files

- Not all text files are to be read or written by you.
- Some are supposed to be read or written by programs.
- When you write a java program it is supposed to be read by the java compiler (javac).
- A PostScript file (.ps) is usually written by a program (save as PostScript) and read by a printer or display program. It is a text file, but it’s rarely looked at by a human.
- Configuration files are often text files. They are generated by programs when you edit preferences, and read by programs to set them up to your liking
    - (gnome/gedit).

Text Files - XML

- These days databases and spreadsheets are increasingly being saved in text-based formats:
    - csv: comma-separated value
    - XML: Extensible Markup Language
- You will learn about XML in the Language course next term.

Accessing Files

- You can **read** from a file.
- You can **write** to a file.

- It is a poor idea to try and do both at the same time!
- Its also a bad idea to have more than one thing trying to write to the same file!
- (Having more than one thing reading is OK.)
- Typically you can only read a file sequentially, and anything you write goes at the end of it.

Text Files - what you see

- Here is a little text file:

There was a young lady of Niger,
Who went for a ride on a tiger.
They returned from the ride,
With the lady inside,
And a smile on the face of the tiger! (traditional limerick)

- The computer doesn't see it as nicely laid out in lines like this.

Text Files- What the computer sees

- What the computer sees is

ThereUwasUaUyoungUladyUofUNiger,↲WhoUwentUforUaUrideUonUaUtiger.↲TheyUreturnedUfromUtheUride,↲WithUtheUladyUinside,↲AndUaUsmileUonUtheUfaceUofUtheUtiger!↲↲(traditionalUlimerick)⊗

- The computer sees it as one long stream of characters. Some of which you can't really see on the screen.

  ∪ is a space
  ↲ is a new line
  ⊗ is an end-of-file character

Invisible Characters

- A space (which we show as ∪) is a standard character (ASCII 32)
- ↲ can be represented in different ways by different operating systems:
  - UNIX: newline (ASCII 10) separates lines
  - Microsoft: carriage return - linefeed (ASCII 13 10) separates lines (also used for some mail file attachments)
  - MacOS: carriage return (ASCII 13) separates lines
  - ⊗ can similarly be represented in different ways:
  - In UNIX it is the eof character - crtl-d (ASCII 4)

File Output

- There are many ways to do file IO, depending on which Java library you use. We will use the standard Java I/O library java.io
- You need to import the library at the start of any program

```
import java.io.*;
```

Write to a file

- **T**o write to a file you need to create an output stream targeted at that file:

```
PrintWriter outputStream =
      new PrintWriter(new FileWriter("mydata.txt"));
```

- This creates such an output stream connecting variable `outputStream` with the file called `mydata.txt`.
- You can use any variable name and any file name.

FileWriter and PrintWriter

- The output stream is created in two steps. First you create a new FileWriter
  `new FileWriter("mydata.txt")`
- It opens the file named (creating one if it doesnt already exist) and links it to a FileWriter object in the program, so the program can write to it. Any characters passed to the FileWriter go into the file.
- To give a series of easy to use methods for actually sending whole lines of text to the file we connect it to a PrintWriter object. It comes with a println method, giving a simple way to print lines of text (strings) to the file just as we would to the screen.
  `PrintWriter outputStream = new PrintWriter(...);`
- Because we have passed the FileWriter object to the PrintWriter, anything that we print to outputStream is passed on the the FileWriter and so into the file.

Writing to the file

- To write to it, you just use
  `outputStream.println("blah");`

Closing the file

- You must ALWAYS close a file once you have finished with it
  `outputStream.close();`
- This tidies up! It releases the file so others can write to it.

Exceptions

- Working with a file can go wrong (eg the filesystem might be full).
- If that happens an error occurs – the program will throw an exception
- You need to label your method to say you will pass any such error on
- You do this by adding throws IOException on the end of the header line
  ```
  public static void main(String[] params)
                                      throws IOException
  ```
- It is actually better practice to catch the exception, and handle it (ie do something about it so the program can carry on) but we will leave you to research that yourself.

Exercise: What does this do?

```
import java.io.*;
class savehello
{
    public static void main(String[] params)
                                      throws IOException
    {
        PrintWriter outputStream =
              new PrintWriter(new FileWriter("hello.txt"));

        outputStream.println("Hello World");
        outputStream.close();
        System.exit(0);
    }
}
```

Exercise

Write a program that saves your name into the file called wombat.txt

More on File Output

- The argument taken by FileWriter is a String
- You can put anything that gives a String there.
- eg a string variable filled using keyboard input

- so the user can specify the file.

```
import java.io.*;
class savehello2
{
    public static void main(String[] params)
                                    throws IOException
    {
        String filename = "file";
        PrintWriter outputStream = new PrintWriter(
                        new FileWriter(filename + ".txt"));
        outputStream.println("Hello World");
        outputStream.close();
        System.exit(0);
    }
}
```

DANGER

- If you close a file for output, and then open it again, you will **erase** the previous contents
- You will not simply add things at the end of the file!!

Reading from a file

- Reading from a file (either one your program created or one created some other way) is also similar to reading from the keyboard.
- In order to read from a file you need an input stream whose source is that file:

```
BufferedReader inputStream =
        new BufferedReader(new FileReader("hello.txt"));
```

- creates such an input stream connected to the file called (in this case) hello.txt.
- To read a line of text from it, you use

```
inputStream.readLine();
```

- It gives a string which you might for example store in a variable

```
String s = inputStream.readLine();
```

Think of a pointer

- Think of an input file as having a pointer pointing at where you are in the file.

```
There␣was␣a␣young␣lady␣of␣Niger,↲|Who␣went␣for␣a␣ride␣on␣a␣tiger.↲|T
hey␣returned␣from␣the␣ride,↲|With␣the␣lady␣inside,↲|And␣a␣smile␣on␣the␣
                    ⇑
face␣of␣the␣tiger!↲↲(traditional␣limerick)⊗
```

Read and Move on

- As characters are read the pointer moves on.

```
...the␣ride...
    ⇑
...the␣ride...
     ⇑
```

- readLine keeps going until it gets to the end of a whole line

Reading Lines

- We've just looked at reading whole lines at a time.
- There are methods that allow you much finer control – eg reading a character at a time, and checking to see if there is another character
- We won't go into the details here – find out about it yourself/

```
...theUride,↲WithUtheUlady...
          ⇑
...theUride,↲WithUtheUlady...
             ⇑
```

Move to end of line

```
infile.readln()
```

- moves the pointer just past the next newline character, but does not give you the string of characters you've just passed.

```
...theUride,↲WithUtheUlady...
          ⇑
...theUride,↲WithUtheUlady...
             ⇑
```

End of File?

```
readLine()
```

- returns a null reference if there are no more lines in the file.
- Strictly you should check therefore that null has not been returned

```
s = inputStream.readLine()
if (s != null)
      outputStream.println(s);
```

Newline characters

- editor gedit does not put a newline at the end of a text file by default.
- To check if you do have a newline:
- cat <file>
- (cat is short for "concatenate").

File Formats

- If a program is storing data in a file for it to read later, so has control over the way the data is stored...
- Then it can use a special format to make it easier
- For example a program that is storing the contents of an array in a file to read back in later could store the size of the array in the first entry
- Then once that it read a simple for loop can read exactly the right number of entries from the file.
- There are many standard file formats that exist that allow programs to share data (eg comma separated files created by spreadsheets)

Characters

- Computers need some way of representing letters (and digits and punctuation) as bit patterns.
- You need to know about two of these:
  - ASCII
  - Unicode

ASCII

- American Standard Code for Information Interchange
- For many years this has been **the** standard. (It was originally set up when the only printers were line-printers).
- It will still be used by any computer you have.
- It has 128 characters (0-127: 7 bits):
  - 0-9 are numbers 48-57

- A-Z are numbers 65-90
- a-z are numbers 97-122

Drawbacks of ASCII

- ASCII has only 128 characters.
- It only has basic Anglo-Saxon characters.
- There's no room for other stuff.
- Could use full 8-bit, but then would have to use different 8-bit set for different languages. (This is done...)

Unicode

- Unicode is a newer system: www.unicode.org
- It uses 32bit characters: ¥uxxxx where each x is a digit in hexadecimal (base 16). (0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f)
- It extends ASCII (first 128 characters are as in ASCII).
- But it can represent huge numbers of characters. Nevertheless there are still problems.
- Standard English characters are sequential in alphabetical order.
- But some languages use some of the same characters as others.
- This means you have a choice.
- Either you have different representations of the same character
- Or you have alphabets that go all over the place.
- Or you have both.
- In ASCII some simple tricks work: newchar = mychar + 'A' - 'a'; capitalises mychar
- This won't work reliably in unicode (not across all scripts).
- Instead there is the function:

```
Character.toUpperCase(char c)
newchar = Character.toUpperCase(mychar);
```

Similarly:

```
Character.toLowerCase(char c)
Character.isUpperCase(char c)
Character.isLowerCase(char c)
Character.isDigit(char c)
Character.toLetter(char c)
```

If you want your program to work reliably internationally, you have to use these.

Standard character escapes

- You can get any character using its unicode form:
- \uxxxx where xxxx is its four-digit hexadecimal unicode number.

Examples:

- newline: \u000a'
- space: \u0020'
- tab: \u0009'
- But many of them have more convenient mnemonic escapes, in particular:
- newline: \n tab: \t

Exercise

Copy a file to another file removing all blank lines.

Typical Input:

Here is an

input text with just two

blank lines.
Typical Output
Here is an
input text with just two
blank lines.
Reading Computing Without Computers Chapter 8
Going Further

- We've looked at one simple way to do file i/o. There are many variations.
- Research how to read a file a character at a time, rather than a line at a time.

Unit 11 Example Programs and Programming Exercises

Only try these exercises once you feel confident with those from the previous unit.
Persistent storage - File Input and Output
Compile and run the following programs, experiment with them modifying them to do slightly different things.

- savehello.java
- savehello2.java
- readhello.java
- readhello2.java
- savenames.java
- readnames.java
- saveNnames.java
- readNnames.java
- readfile.java

Write to a file
Write a program that creates a file containing your name and address. The name of the file can be built in to your program. Check the file has been created by looking at it in an editor.
Display a file with line numbers
Write a program that displays a text file line by line on the screen, but displaying the line with line numbers. Again, the name of the file can be built into your program. (Test it on the file created by your exercise 1)
Write to a file given by the user
Write a program that creates a file containing your name and address. The program should ask the user for the name of the file.
Write from keyboard to a file
Write a program that writes a name supplied by the user to a file fixed in the program. What happens to the original information in the file if you run the program twice storing different information in the file.
Write repeatedly to a file
Write a program that repeatedly asks to input a name (10 times) and writes each to a file.

Copy a file
Copy a file a line at a time to another file. For a first attempt hardwire the names of the input and output files into your program. Then rewrite your program so that they are read from the keyboard. Compare the files afterwards using the Linux commands cmp and diff. Learn about these programs.

Additional Exercises
The following are additional, harder exercises. The more programs you write this week the easier you will find next week's work and the assessments and the quicker you will learn to program. If you complete these exercises easily, you may wish to read ahead and start thinking about next week's exercises.

Some of these involve reading files character by character – research how to do that yourself.

Rotate vowels
Copy a file named "history" to an output file named "history.copy" rotating the lower-case vowels. That is, change all occurrences of
'a' to 'e'
'e' to 'i'
'i' to 'o'
'o' to 'u'
'u' to 'a'

Typical input file

```
In an age when acronyms were popular, the Manchester Mark I was
sometimes referred to as MADM (Manchester Automatic Digital
Machine) or MUC (Manchester University Computer).
```

Typical output file

```
In en egi whin ecrunyms wiri pupaler, thi Menchistir Merk I wes
sumitomis rifirrid tu es MADM (Menchistir Aatumetoc Dogotel
Mechoni) ur MUC (Menchistir Unovirsoty Cumpatir).
```

Output file with line numbers
Copy a file character by character to another file adding line numbers to each line.

Compare two files
Write a program that compares two input files character by character (and so behaves like the Linux command cmp). The filenames should be read from the keyboard. If the files match exactly then output a suitable message. Otherwise output the character number at which they differ.

Simple version of diff
Compare two input files line by line (i.e. write a program which behaves a bit like diff - diff itself is much more sophisticated). The filenames should be read from the keyboard. You may use Hansen's readline method to read the lines of the two files. If the files match exactly then output a suitable message. Otherwise output the first pair of lines which differ along with the line number.

Remove blank lines

Copy a file to another file removing all blank lines.

Typical Input

```
    Here is an

    input text with just two

    blank lines.
```

Typical Output

```
    Here is an
    input text with just two
    blank lines.
```

Word count

A program is required which will print out the number of characters, words and lines in any given text stored in a file.

For the purposes of this program a 'word' is any sequence of characters not containing space (' '), tab ('\t') or newline ('\n').

The output should agree with that obtained using the Unix (or Linux) program wc. This means that newline, space and tab characters are counted as characters.

Typical input:

```
  This is an extract from a book called "Godel, Escher, Bach: an eternal
    golden braid" by D.R.Hofstadter.

    "One of the most central notions in this book is that of a formal
system. The type of formal system I use
    was invented by the American logician Emil Post in the 1920's, and is
often called a "Post production system".

    This Chapter [1] introduces you to a formal system and moreover, it is
my hope that you will want to explore
    this formal system at least a little; so to provoke your curiosity, I
have posed a little puzzle.

    "Can you produce MU?" is the puzzle.  To begin with, you will be
supplied with a string (which means a string
    of letters). Not to keep you in suspense, that string will be MI.  Then
you will be told some rules, with
    which you can change one string into another.  If one of those rules is
applicable at some point and you want
    to use it, you may, but - there is nothing that will dictate which rule
you should use, in case there are
    several applicable rules.
    ...
    The first thing to say about our formal system - the MIU-system - is
that it utilizes only three letters of
    the alphabet: M, I, U. That means that the only strings of the MIU-
system are strings which are composed of
    those three letters.
    ...
    Rule 1:    If you possess a string whose last letter is I, you can add
on a U at the end.

    Rule 2:    Suppose you have Mx. Then you may add Mxx to your collection.
[x stands for any string of letters MIU]

    Rule 3:    If III occurs in one of the strings in your collection, you
may make a new string with U in place of III.

    Rule 4:    If UU occurs inside one of your strings, you can drop it.
```

Typical output

```
    Number of lines  25
    Number of words  313
    Number of characters  1623
```

Checking an expression for "well-bracketedness"
Read a string of characters representing a bracketed expression from a file one character at a time until end of file. Maintain a count of how many opening parentheses '(' and closing parentheses ')' have been encountered.

Declare an integer variable 'level', initialized to zeroAt each opening parenthesis, increase its value by one, and print its new value. At each closing parenthesis, decrease its value by one, and print its new value. If the level becomes negative, print "Unmatched ')'", just once regardless of the number of unmatched ')'s. At end-of-file, if the level is positive print "Unmatched '('". If the level is zero and there have been no unmatched ')'s then print "Well-bracketed expression".
Typical input

```
(a*(b-c)-((d/e)+f))
```

Typical output

```
Level 1
Level 2
Level 1
Level 2
Level 3
Level 2
Level 1
Level 0
Well-bracketed expression
```

Unit 11 Exam Style Question: Persistent storage

Question 1 [25 marks]
a. [9 marks]
Explain what is meant by:
i) persistent storage
ii) a text file
iii) ASCII

b. [6 marks]
Compare and contrast file input with keyboard input using scanners.

c. [10 marks]
Write a Java program that reads the contents of a text file "wombat.txt" and counts the number of times the words “a” and “the” each occur in it. You may assume the file contains one word per line.

If you got this far, doing all the exercises along the way, hopefully you can now program! Well done! Keep learning though - there is a lot more to learn and the more you understand the better programmer you will be.

Over time I am adding material to the final “further challenges, going deeper” unit for anyone who runs out of things to do. Ask me if you want some new challenges