Arrays

This lecture will

- Explain how Java arrays can store and manipulate collections of data
- Introduce the enhanced `for` loop
- Introduce simple algorithms for searching and sorting arrays
- Explain multi-dimensional arrays

Collections of data items

- We often need to refer to collections of elements of the same type, e.g. a table of employee details or salaries
- It is inconvenient to write a collection of 5 integers as:

```
int dataItem1;
int dataItem2;
int dataItem3;
int dataItem4;
int dataItem5;
```

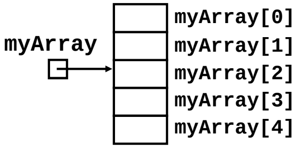
- Java allow us to store a collection of elements of the same type in an **array**

Declaring an array

- To declare an array of 5 integers called `myArray`, we write:

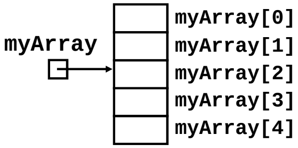
```
int[] myArray = new int[5];
```

- We pronounce `int[]` as “an array of int”
- `myArray` is a reference to an area of memory containing a collection of 5 integers:



Array indexing

- We specify an individual array element with an **index**, e.g. `myArray[3]`
- Indices are numbered from zero; the last index is one less than the number of elements in the array



Literal arrays

- We can initialise an array using a **literal array expression**, by specifying the elements in curly brackets:

```
int[] myArray = {1, 3, 5, 7, 9};
```

- The compiler calculates how many array elements there are (5 in this case, numbered from `myArray[0]` to `myArray[4]`)

How many elements in an array?

- We can find out the number of elements in `myArray` by writing `myArray.length`
- This is better than using a literal value, for reasons of software maintenance:

```
int[] myArray = {1, 3, 5, 7, 9};
for (int i=0; i<5; i++)
    myArray[i] = i * 10;
```

Bad

```
int[] myArray = {1, 3, 5, 7, 9};
for (int i=0; i<myArray.length; i++)
    myArray[i] = i * 10;
```

Good

How many elements in an array?

- The number of elements in an array can be determined at run time
- This creates an array that contain a user-specified number of elements:

```
int items=keyboard.readInt("How many? ");
int[] myArray=new int[items];
```

- But once the array has been created its size cannot be changed

Using a for loop to process an array

- We often use a **`for`** loop to process each array element:

```
for (int i=0; i<myArray.length; i++)
   myArray[i] = i * 10;
```

- Each element `myArray[i]` is processed in turn as **`i`** steps through from 0 to `myArray.length` – 1
- Be careful not to write

```
for (int i=0; i<=myArray.length; i++)
```

✗

A table of integers

```
Enter the number of items: 3
Enter number 1: 45
Enter number 2: 37
Enter number 3: 23
Your numbers were:
45
37
23
```

```
import sheffield.*;
public class SimpleTable {
   public static void main(String[] args) {

      EasyReader keyboard = new EasyReader();
      int items=keyboard.readInt("How many elements? ");
      int[] myArray = new int[items];

      for (int i=0; i<items; i++)
         myArray[i]=
            keyboard.readInt("Enter number "+(i+1)+": ");

      System.out.println("Your numbers were:");
      for (int i=0; i<items; i++)
         System.out.println(myArray[i]);
   }
}
```

Because people count from 1

A simpler table

```
import sheffield.*;
public class SimplerTable {
   public static void main(String[] args) {
      EasyReader keyboard = new EasyReader();
      int items=keyboard.readInt("How many words? ");

      String[] myArray = new String[items];
      for (int i=0; i<items; i++)
         myArray[i]= keyboard.
                  readString("Enter word "+(i+1)+": ");

      System.out.println("Your words were:");
      for (int element : myArray)
         System.out.println(element);
   }
}
```

An enhanced **for** loop

The enhanced for loop

- Is used to **access** values of an array in turn without a counter

```
for (type variable_name  :  array_name)
     loop_body;
```

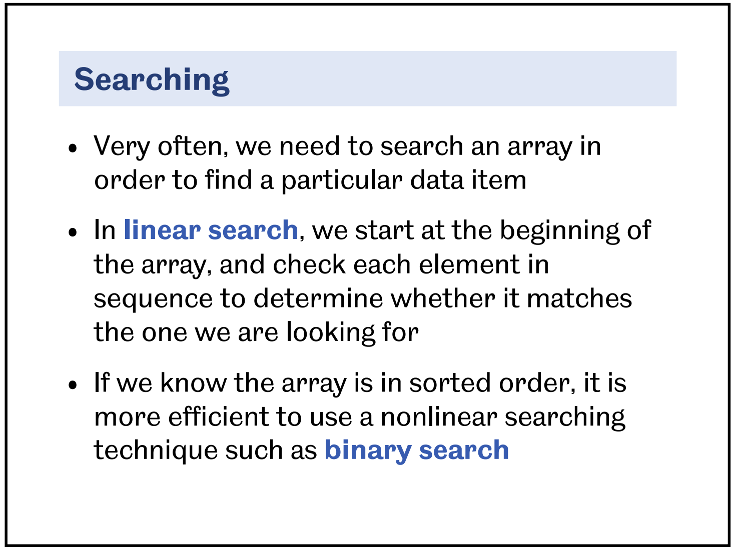
- The *type* is the type of the elements in the array
- It steps through the elements from 0 to the end in that order
- The *variable_name* takes the value of each element in turn

Using an expression as an array index

```
public class TestArrayExpressions {
  public static void main (String[] args) {
    int x=1, y=10;
    int[] dataItem = new int[5];
    dataItem[2] = 5;
    dataItem[0] = dataItem[2] * 2;
    dataItem[x+2] = 3*4;
    dataItem[3-2] = 65;
    dataItem[2+(x*6+98)/52] = 2+x+y;
    for (int d : dataItem)
      System.out.println(d);
  }
}
```

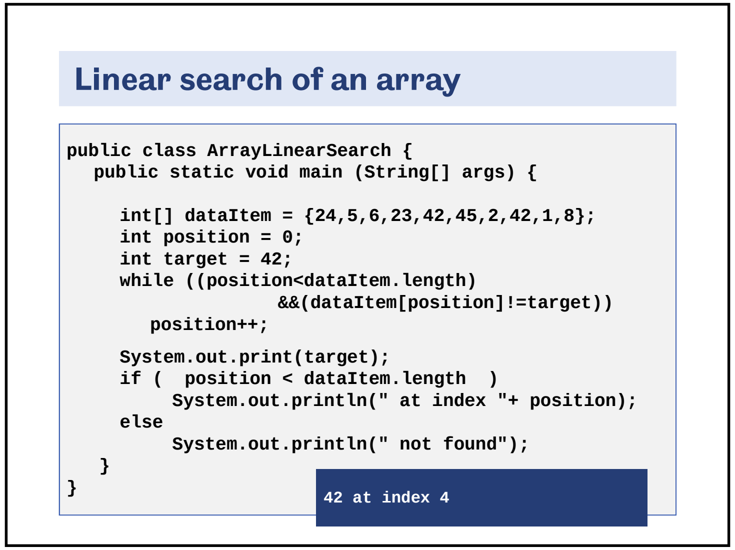
10	dataItem[0]
65	dataItem[1]
5	dataItem[2]
12	dataItem[3]
13	dataItem[4]

```
10
65
5
12
13
```



Searching

- Very often, we need to search an array in order to find a particular data item
- In **linear search**, we start at the beginning of the array, and check each element in sequence to determine whether it matches the one we are looking for
- If we know the array is in sorted order, it is more efficient to use a nonlinear searching technique such as **binary search**



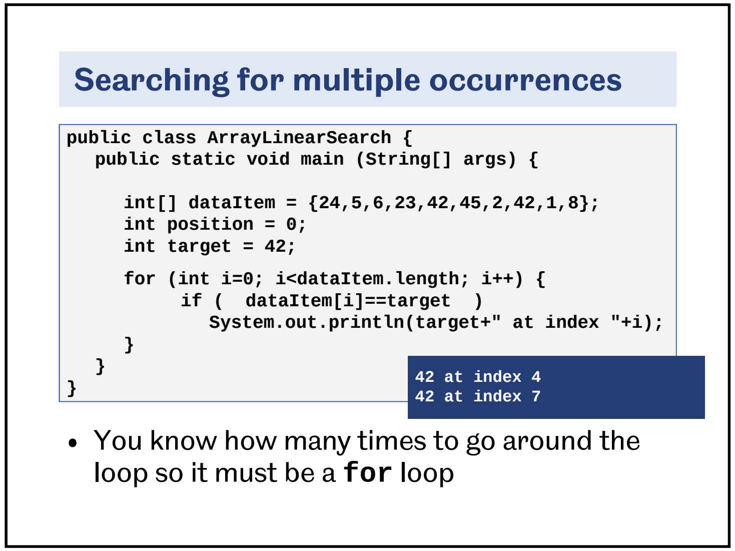
Linear search of an array

```
public class ArrayLinearSearch {
   public static void main (String[] args) {

      int[] dataItem = {24,5,6,23,42,45,2,42,1,8};
      int position = 0;
      int target = 42;
      while ((position<dataItem.length)
                      &&(dataItem[position]!=target))
         position++;

      System.out.print(target);
      if (  position < dataItem.length  )
           System.out.println(" at index "+ position);
      else
           System.out.println(" not found");
   }
}
```

```
42 at index 4
```



Searching for multiple occurrences

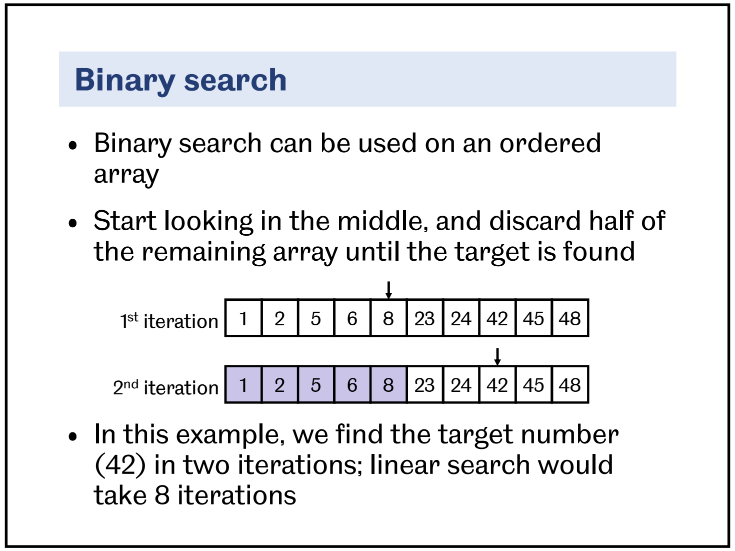
```
public class ArrayLinearSearch {
   public static void main (String[] args) {

      int[] dataItem = {24,5,6,23,42,45,2,42,1,8};
      int position = 0;
      int target = 42;

      for (int i=0; i<dataItem.length; i++) {
           if (  dataItem[i]==target  )
              System.out.println(target+" at index "+i);
      }
   }
}
```

```
42 at index 4
42 at index 7
```

- You know how many times to go around the loop so it must be a `for` loop

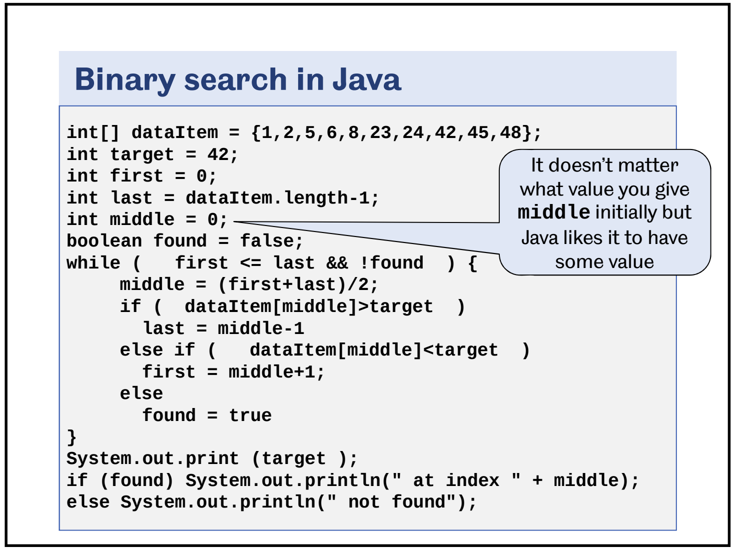


Binary search

- Binary search can be used on an ordered array
- Start looking in the middle, and discard half of the remaining array until the target is found

					↓					
1st iteration	1	2	5	6	8	23	24	42	45	48
								↓		
2nd iteration	1	2	5	6	8	23	24	42	45	48

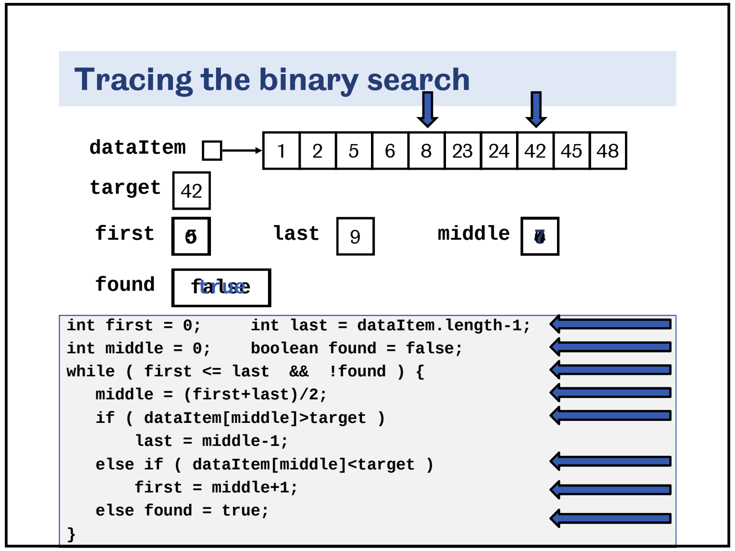
- In this example, we find the target number (42) in two iterations; linear search would take 8 iterations



Binary search in Java

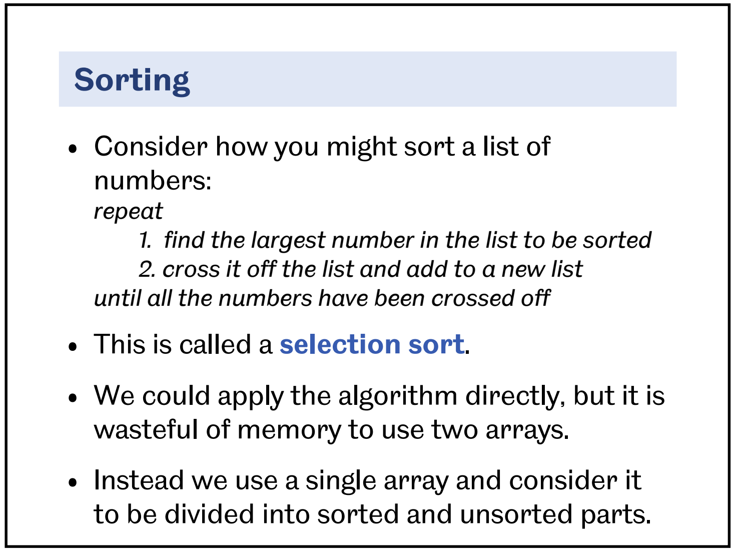
```
int[] dataItem = {1,2,5,6,8,23,24,42,45,48};
int target = 42;
int first = 0;
int last = dataItem.length-1;
int middle = 0;
boolean found = false;
while (   first <= last && !found   ) {
     middle = (first+last)/2;
     if (  dataItem[middle]>target   )
       last = middle-1
     else if (   dataItem[middle]<target   )
       first = middle+1;
     else
       found = true
}
System.out.print (target );
if (found) System.out.println(" at index " + middle);
else System.out.println(" not found");
```

It doesn't matter what value you give `middle` initially but Java likes it to have some value



Tracing the binary search

```
int first = 0;     int last = dataItem.length-1;
int middle = 0;    boolean found = false;
while ( first <= last  &&  !found ) {
   middle = (first+last)/2;
   if ( dataItem[middle]>target )
       last = middle-1;
   else if ( dataItem[middle]<target )
       first = middle+1;
   else found = true;
}
```

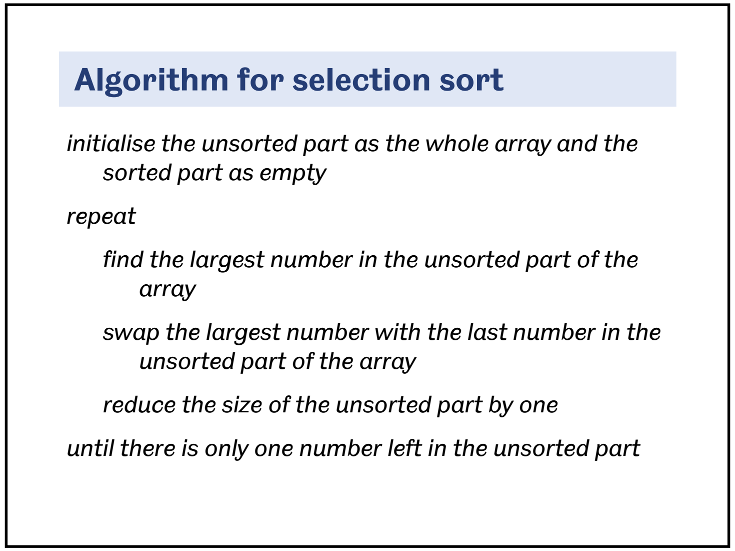


Sorting

- Consider how you might sort a list of numbers:

```
repeat
    1. find the largest number in the list to be sorted
    2. cross it off the list and add to a new list
until all the numbers have been crossed off
```

- This is called a **selection sort**.
- We could apply the algorithm directly, but it is wasteful of memory to use two arrays.
- Instead we use a single array and consider it to be divided into sorted and unsorted parts.



Algorithm for selection sort

```
initialise the unsorted part as the whole array and the sorted part as empty
repeat
    find the largest number in the unsorted part of the array
    swap the largest number with the last number in the unsorted part of the array
    reduce the size of the unsorted part by one
until there is only one number left in the unsorted part
```

Selection sort in Java

```
public class SelectionSort {
   public static void main(String[] args) {
      int[] dataItem = {24, 5, 6, 23, 42, 45, 2, 42, 1, 8};
      System.out.println("Unsorted data:");
      for (int d : dataItem) System.out.print(d + " ");
      System.out.println();
      for (int lastUnsorted=dataItem.length-1;
                                   lastUnsorted>0; lastUnsorted--) {
          int positionOfLargest=lastUnsorted;
          for (int i=0; i<lastUnsorted; i++) {
             if (dataItem[i] > dataItem[positionOfLargest])
                  positionOfLargest = i;
          }
          if  ( positionOfLargest != lastUnsorted  )  {
             int temp = dataItem[positionOfLargest];
             dataItem[positionOfLargest] = dataItem[lastUnsorted];
             dataItem[lastUnsorted] = temp;
         }
      }
      System.out.println("Sorted data:");
      for (int d : dataItem) System.out.print(d + " ");
      System.out.println();
   }
}
```

Selection sort in Java

```
for (int lastUnsorted = dataItem.length-1;
                  lastUnsorted>0; lastUnsorted--) {
     //Find largest in unsorted part
     int positionOfLargest=lastUnsorted;
     for (int i=0; i<lastUnsorted; i++) {
       if (dataItem[i] > dataItem[positionOfLargest])
             positionOfLargest = i;
     }

     // Swap largest with last unsorted
     if  ( positionOfLargest != lastUnsorted  )  {
        int temp = dataItem[positionOfLargest];
        dataItem[positionOfLargest] =
                      dataItem[lastUnsorted];
        dataItem[lastUnsorted] = temp;
     }
}
```

Multidimensional arrays

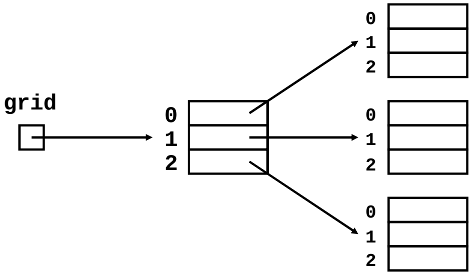
- Arrays can have more than one dimension.
- The most useful are two dimensional (2-D), which have **rows** and **columns**.

```
int[][] grid = new int[3][3];
```

- The array **`grid`** is of type **`int[][]`**, pronounced "array of array of int".
- So, **`grid`** is actually a one-dimensional array of one-dimensional arrays.

Visualising a 2-dimensional array

```
int[][] grid = new int[3][3];
```



Visualising a 2-dimensional array as a matrix

```
int[][] grid = new int[3][3];
```

grid[0][0]	grid[0][1]	grid[0][2]
grid[1][0]	grid[1][1]	grid[1][2]
grid[2][0]	grid[2][1]	grid[2][2]

Processing a 2-dimensional array

- To process a 2-D array, we use a nested loop:

```
for (int r=0; r<grid.length; r++)
  for (int c=0; c<grid[r].length; c++)
      grid[r][c]=0;
```

- We use **r** to count rows (there are **grid.length** rows) and **c** to count columns in each row (there are **grid[r].length** columns).

Initialising a 2-dimensional array

- We can also initialise multidimensional arrays by writing the elements of each row in curly brackets:

```
int[][] grid = {{0,1,2},{3,4,5},{6,7,8}};
```

The enhanced for loop with 2D arrays

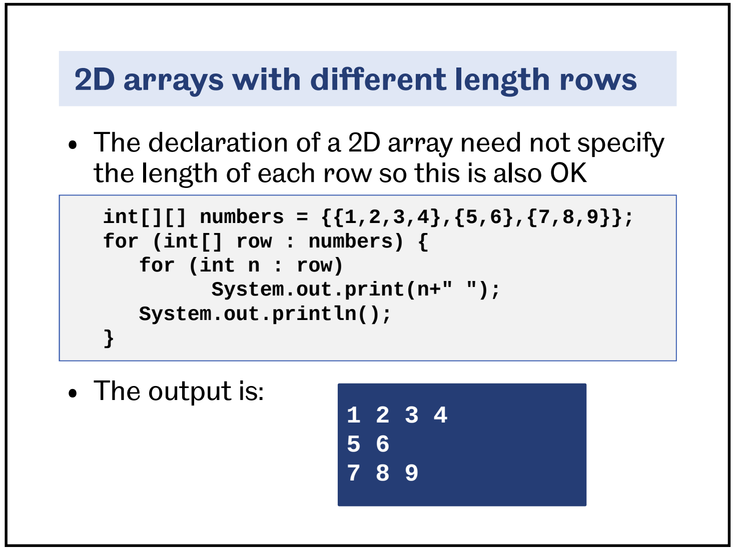
- The enhanced **for** loop works with 2D arrays too, if we remember that multidimensional Java arrays are represented as arrays of arrays:

```
int[][] numbers = {{1,2,3},{4,5,6},{7,8,9}};
for (int[] row : numbers) {
    for (int n : row)
          System.out.print(n+" ");
    System.out.println();
}
```

Don't use ' ' here

- The output is:

```
1 2 3
4 5 6
7 8 9
```



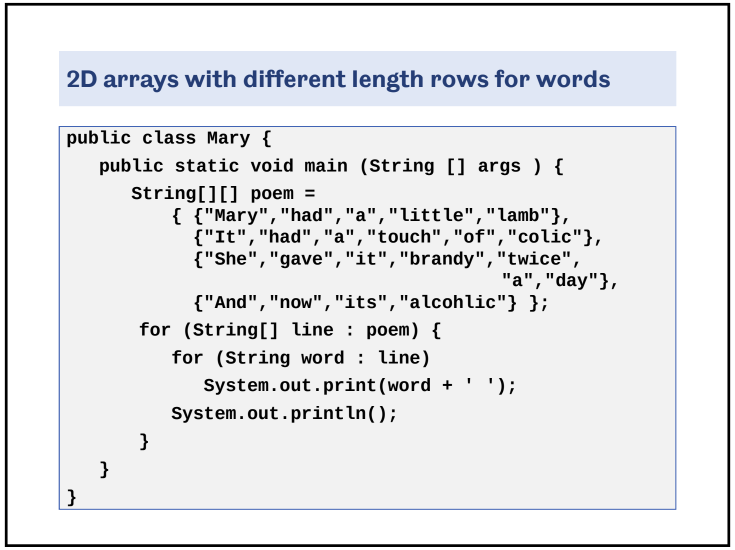
2D arrays with different length rows

- The declaration of a 2D array need not specify the length of each row so this is also OK

```
int[][] numbers = {{1,2,3,4},{5,6},{7,8,9}};
for (int[] row : numbers) {
   for (int n : row)
         System.out.print(n+" ");
   System.out.println();
}
```

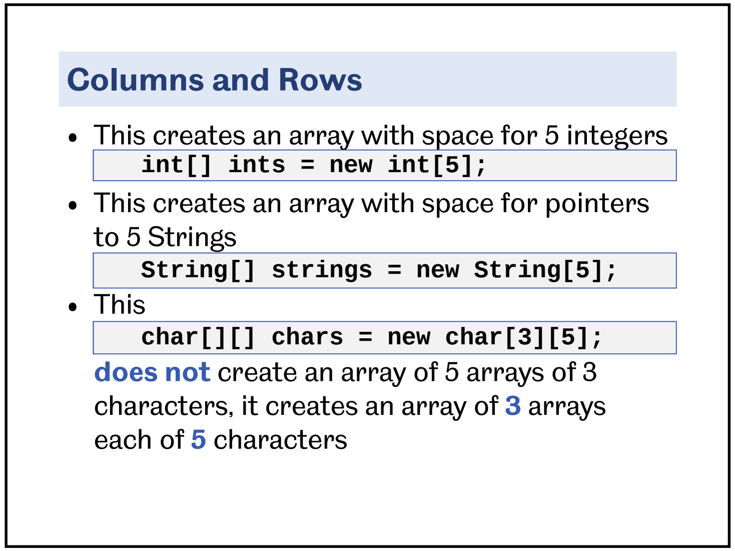
- The output is:

```
1 2 3 4
5 6
7 8 9
```



2D arrays with different length rows for words

```
public class Mary {
   public static void main (String [] args ) {
      String[][] poem =
         { {"Mary","had","a","little","lamb"},
           {"It","had","a","touch","of","colic"},
           {"She","gave","it","brandy","twice",
                                        "a","day"},
           {"And","now","its","alcohlic"} };
       for (String[] line : poem) {
          for (String word : line)
             System.out.print(word + ' ');
          System.out.println();
       }
   }
}
```



Columns and Rows

- This creates an array with space for 5 integers

```
int[] ints = new int[5];
```

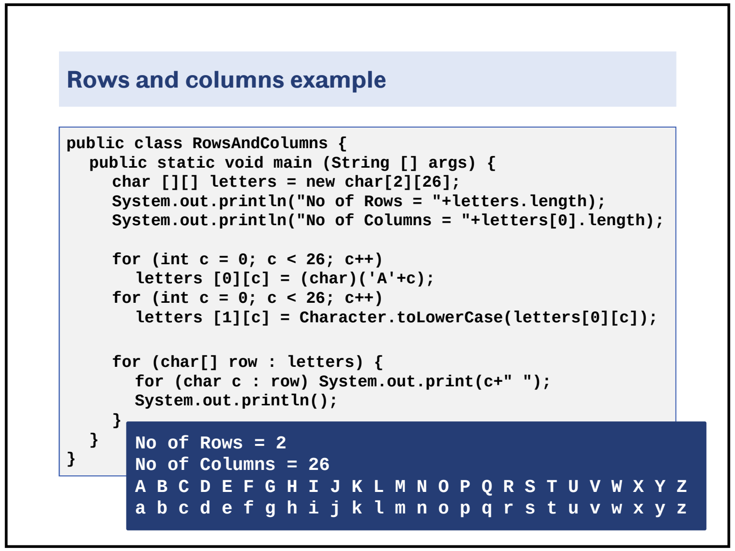
- This creates an array with space for pointers to 5 Strings

```
String[] strings = new String[5];
```

- This

```
char[][] chars = new char[3][5];
```

does not create an array of 5 arrays of 3 characters, it creates an array of **3** arrays each of **5** characters



Rows and columns example

```
public class RowsAndColumns {
   public static void main (String [] args) {
      char [][] letters = new char[2][26];
      System.out.println("No of Rows = "+letters.length);
      System.out.println("No of Columns = "+letters[0].length);

      for (int c = 0; c < 26; c++)
         letters [0][c] = (char)('A'+c);
      for (int c = 0; c < 26; c++)
         letters [1][c] = Character.toLowerCase(letters[0][c]);

      for (char[] row : letters) {
         for (char c : row) System.out.print(c+" ");
         System.out.println();
      }
   }
}
```

```
No of Rows = 2
No of Columns = 26
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

2D Arrays

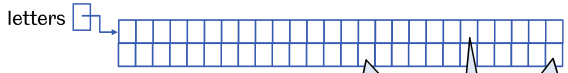
```
char [][] letters;
```

- Creates space for a pointer to an array

letters □

```
char [][] letters = new char[2][26];
```

- Creates space for a pointer to an array and the array itself



- Any component of the array can be referred to by the array name and indices

Notice that the first index refers to the row and the second to the column both here

and here

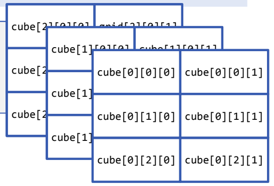
letters [1][14]

letters [0][20]

letters [1][25]

3D Arrays

```
char[][][] cube =
          new char[3][3][2];
```



- Used in many applications including computer graphics.

Rows and Columns and Layers

```
char [][][] letters = new char[2][3][5];
System.out.println("No of Layers = "+letters.length);
System.out.println("No of Rows = "+letters[0].length);
System.out.println("No of Columns = "+letters[0][0].length);
char next = 'A';
for (int a = 0; a<2; a++)
  for (int b=0; b<3; b++)
    for (int c=0; c<5; c++) {
      letters[a][b][c]=next;
      next = (char)(next+1);
    }
System.out.println("-------------------");
for (char[][] first : letters) {
  for (char[] second : first) {
    for (char third : second)
      System.out.print(third+" ");
    System.out.println();
  }
  System.out.println("-------------------");
}
```

```
No of Layers = 2
No of Rows = 3
No of Columns = 5
-------------------
A B C D E
F G H I J
K L M N O
-------------------
P Q R S T
U V W X Y
Z [ \ ] ^
-------------------
```

Remember this..

Change the print statement in the **Simple.java** program to:

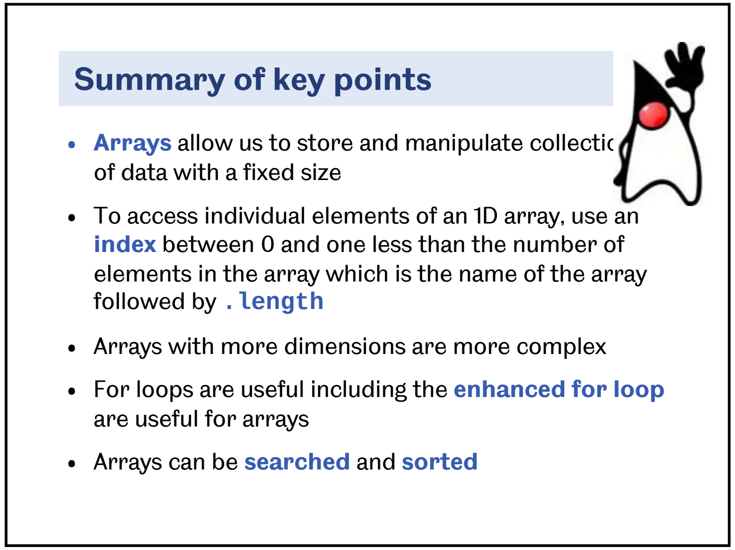
```
System.out.println("Hello " + args[0]);
```

Compile the program as usual but run it with

```
U:...\myjava> java Simple XXXX
```

```
public class Exercise1c {
     public static void main(String[] args) {
        System.out.println("Hello "+args[0]);
     }
}
```

```
U:…\myjava>java Exercise1g Siobhan
Hello Siobhan
U:…\myjava>java Exercise1g
Exception in thread main java.lang.ArrayIndexOutOfBoundsException: 0
            at Exercise1g.main(Exercise1g.java:3)
```



Summary of key points

- **Arrays** allow us to store and manipulate collectic of data with a fixed size
- To access individual elements of an 1D array, use an **index** between 0 and one less than the number of elements in the array which is the name of the array followed by `.length`
- Arrays with more dimensions are more complex
- For loops are useful including the **enhanced for loop** are useful for arrays
- Arrays can be **searched** and **sorted**