8. Arrays and ArrayLists

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

So far the data we've seen consists of variables for storing *single* items. But this is often not convenient (or even possible in some cases). Here's an example of a program that is a bit tricky with only the types of data we have so far (the line to import Scanner is left out to make it shorter). The program counts the number of times you throw each possible number (1 to 6) on a normal dice (or 'die' if you're being picky).

```
public class SillyDice
{
   public static void main(String[] args)
   {
      int side1 = 0;
      int side2 = 0;
      int side3 = 0;
      int side4 = 0;
      int side5 = 0;
      int side6 = 0;

      System.out.println("Type values, non-integer to quit:");
      Scanner in = new Scanner(System.in);
      while (in.hasNextInt()) {
         int value = in.nextInt();

         if (value == 1) {
           side1++;
         } else if (value == 2) {
           side2++;
         } else if (value == 3) {
           side3++;
         } else if (value == 4) {
           side4++;
         } else if (value == 5) {
           side5++;
         } else if (value == 6) {
           side6++;
         } else {
           System.out.println("Not a valid value.");
         }
      }

      System.out.println("1:  " + side1);
      System.out.println("2:  " + side2);
      System.out.println("3:  " + side3);
      System.out.println("4:  " + side4);
      System.out.println("5:  " + side5);
      System.out.println("6:  " + side6);
   }
}
```

The program reads in integers until the user types a non-integer (when it exits the loop). It then checks the value of the integer and counts them. This program has one integer variable for each possible side of the dice.

There are problems with this:

- **It's Long** – We have quite long code to: (a) declare all the variables; (b) check which number has been typed in (the if statements); and (c) print them out.
- **It's Hard to Change** – not all dice only have six sides (though they are the most common). So if we want to change it for more or less sides, we have to add new variables, extend the if statement and add to the print statements. These changes would make it even longer.

The data we're dealing with here is conceptually related – it's a *collection* of data about the same thing and it would be easier if we could treat it like that. Here's a new program:

```
public class SimpleDice
{
   public static void main(String[] args)
   {
       final int SIDES = 6;
       // counters[0] is not used
      int[] counters = new int[SIDES+1];

      System.out.println("Type values, non-integer to quit:");
      Scanner in = new Scanner(System.in);

      //Read in the values
      while (in.hasNextInt()) {
         int value = in.nextInt();

         // Increment the counter for the input value
         if (1 <= value && value <= SIDES) {
            counters[value]++;
                  System.out.println("Count of " +
                     value + " is " + counters[value]);
         } else {
            System.out.println(value +
                     " is not a valid input.");
         }
      }

      // Print out the counts of the entered values
      for (int j = 1; j < counters.length; j++) {
         System.out.println(j + ": " + counters[j]);
      }
   }
}
```

This version is much shorter and uses an *array* to hold the counts of the number of throws. The key first line is:

```
int[] counters = new int[SIDES+1];
```

This declares an *array* (of integers) with enough space to hold SIDES+1 integers. An array is basically a list of numbered 'boxes' each of which can hold a *single* data item – in this case the data items are integers, but arrays can hold any kind of data (though each box in an array must hold *the same* type of data). We use the notation `name[x]` to refer to a data item in an array, where `name` is the name of the array, and `x` is the *index* - an integer between zero (arrays always start at element zero in Java) and *max* – 1, where *max* is the size of the array when we create it. In this case, max is equal to SIDES + 1, and SIDES is 6. So the largest value of x we can use is SIDES + 1 – 1 = 6. Slightly confusingly, because array indexes start at zero, for an array of size *n*, the indexes run from zero to *n-1* – **not** to *n*. Getting this wrong is a common mistake (everyone does it from time to time) and is another example of an *off-by-one* error.

The if statement:

```
if (1 <= value && value <= SIDES) {
```

Checks to see if the user has typed a number in the correct range (from 1 to SIDES). If the number is in the right range, it updates the count (`counters[value]++;` - see below) and prints a message; otherwise it prints an error message.

For example, if we have so far thrown three 1's, two 2's, four 3's and 4's, one 5 and two 6's, then the array will have these values:

Array Element	**Value**
counters[0]	0
counters[1]	3
counters[2]	2
counters[3]	4
counters[4]	4
counters[5]	1
counters[6]	2

Notice in this case we are not using `counters[0]` because a dice does not have a 'side zero'. We could instead have chosen to store the number of 1's thrown in `counters[0];` the number of 2's in `counters[1]` and so on – but for this example (and in some others) it's easier just to not use `counters[0]`.
We can use the notation `name[x]` to both set the value of array elements and to read them too. So the line:

```
counters[value]++
```

increases the value of `counters[value]` by one – and is exactly the same as `counters[value] = counters[value] + 1;` or `counters[value] += 1;` - just shorter.
Notice also the line

```
System.out.println(j + ": " + counters[j]);
```

which reads (and prints) the value of `counters[j].`

The other point to note about the program is the statement

```
for (int j = 1; j < counters.length; j++)
```

which loops from element 1 to (in this case) 6. The expression `counters.length` returns the length of the array (in this case `SIDES`+1 = 7), so this loop ends when `j` = 6 (because it only continues as long as `j < counters.length`). For any array, we can always use `name.length` to tell us how long it is.

This program is much better than the previous one – it's shorter but more importantly it's very easy to change. If you want to use a new dice with, say, 10 sides – just change the value of the final variable `SIDES` from 6 to 10 – *and that's all you need to do.* By changing the value of SIDES we will automatically:

- Create an array the right size (`new int[SIDES+1];`).
- Check that the value typed in is in the right range (`if (1 <= value && value <= SIDES) {`)
- Loop the right number of times to print all the values (`for (int j = 1; j < counters.length; j++)`)

What if we try to look up an element of an array that is outside the range of 0 to max – 1? For example, in the case above, if we tried to look up element -5; or element 20 (where `SIDES` is < 20). This causes a *runtime error* – the program will crash with a message like this:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 44

        at SimpleDice.main(SimpleDice.java:26)
```

`ArrayIndexOutOfBounds` means that the number we are using to look up an element in the array (the *index*) is not within the range of allowed values.

Creating Arrays

Looking at the line that creates the array in a bit more detail:

```
int[] counters = new int[SIDES+1];
```

The first part declares the array and the second initializes it to be of size SIDES+1. As with all variable declarations, you can split it into two parts if you want:

```
int[] counters;
counters = new int[SIDES+1];
```

But as always it’s better if you can to combine the two together. When we declare a variable, we can make it an array by putting ‘[]’ after the type. So we can make an array of strings like this:

```
String[] names = new String[6];// An array of 6 strings.
```

An alternative way to create an array makes sense when you know what data you want is this:

```
int[] primes = {1, 3, 5, 7, 11, 13, 17, 19};
```

which lists the first eight prime numbers. In this case, we don’t have to create the array and then individually put the elements we want in it – we just list them and Java works out how big the array needs to be and put the data in for us. This often makes sense – especially for data that will not change. But you can’t always do it even for that – for example, suppose you wanted an array of the first million prime numbers… Also, our example with counting dice throws won’t work because we don’t know the numbers when we create the array.

Shape-of-the-Day – Another Array Example

Here’s another example of a program that’s tricky without arrays (or at least clumsy). This program pritns out the name of a random shape:

```
import java.util.Random;

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

          Random rnd = new Random();

          final int MAX_SHAPES = 6;
          final int TRIANGLE = 3;

          int sides = rnd.nextInt(MAX_SHAPES)+TRIANGLE;

          System.out.println("Sides is " + sides);
          System.out.print("The shape of the day is a ");
          switch (sides) {
               case 3: System.out.println("Triangle.");
                       break;
               case 4: System.out.println("Square.");
                       break;
               case 5: System.out.println("Pentagon.");
                       break;
               case 6: System.out.println("Hexagon.");
                       break;
               case 7: System.out.println("Heptagon.");
                       break;
               case 8: System.out.println("Octagon");
                       break;
          }
     }
}
```

This program creates a random number between 3 and 8, and prints out the name of the shape with that number of sides (I’ll explain how to do random numbers below). It’s tricky to extend it – we’d need to add more lines to the `switch` statement, and change the value of `MAX_SHAPES`. Here’s a better one:

```
import java.util.Random;

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

           Random rnd = new Random();

           final String[] shapeNames = {"Triangle",
                  "Square","Pentagon","Hexagon",
                  "Heptagon", "Octogon","Nonagon",
                  "Decagon"};
           final int[] shapeSides = {3,4,5,6,7,8,9,10};

           int sides = rnd.nextInt(shapeNames.length);

           System.out.print("The shape of the day is a ");
           System.out.println(shapeNames[sides]);
           System.out.println("It has " +
                          shapeSides[sides] +" sides");
     }
}
```

This one is shorter – despite doing more shapes *and* telling you how many sides they have. We still generate a random number, but this time we use it to look up the shape in an array. The array shapeNames (of Strings) contains the names of the shapes; and shapeSides contains the number of sides (integers). The random number we generate is in the range of – in this case – 0 to 9; and we use that number to look up the right elements in the array (shapeNames[sides] and shapeSides[sides]). The way we generate random numbers in Java (or at least one of them) is to use the Random library – which we include in the import at the top. Then we need to 'make' a random number generator (much like we make a Scanner):

```
import java.util.Random;
```

And then to generate a random number, we call nextInt(x) – which works by generating a number from 0 to x - 1, where x is an integer. Our first program generates a number beween 3 and 8:

```
final int MAX_SHAPES = 6;
final int TRIANGLE = 3;

int sides = rnd.nextInt(MAX_SHAPES)+TRIANGLE;
```

The call to nextInt generates a number between 1 and 5 (`MAX_SHAPES` – 1); and then we add 3 to it to get a number from 3 to 8.
The second one just generates a number from 1 to the size of the array `shapeNames`:

```
int sides = rnd.nextInt(shapeNames.length);
```

(we could have used the `shapeSides.length` – that would work too). This means that to change this program to print more shapes, we only have to add their details to the arrays:

```
final String[] shapeNames = {"Triangle",
                  "Square","Pentagon","Hexagon",
                  "Heptagon", "Octogon","Nonagon",
                  "Decagon" "hendecagon", "Icosagon"};
final int[] shapeSides = {3,4,5,6,7,8,9,10,11,20};
```

Limitations and Problems

Arrays are great for applications like the one above because:

- **We know in advance how many data items there will be** – the number of sides is fixed when the program is compiled.
- **We don't want to insert new data** – that is, we don't want to put in a new data item between, say element 3 and 4.
-

Neither of these things are a problem (or even make sense as an idea) for this example. But suppose you are trying to create a list of, say, names of users of a system. You probably:

- **Don't know how many there will be to start with** – quite often you would not have a clear idea of how many potential users there would be. But you need to say how big the array is going to be when you create it. You could create a *really* big array to be safe but: (a) that would potentially waste a lot of memory (though we're not too bothered by that); and (b) you could still be wrong (and run out of space).
- **Inserting in Alphabetical Order** – if you do the sensible thing (probably) and store the names in alphabetical order (which makes it easier to for example search), then you will probably need to, at some point, insert a name *between* two existing ones. This is difficult with arrays, as we'll see below – you have to *move* the data to put the new item in.
- **Deleting Names** – the same kind of problem arises when you want to delete a name: you are going to have a gap, which you need to 'close' somehow.
- **Empty Array Elements** – although `name.length` is handy, it doesn't always help because it returns the size of the array – not the number of elements we are using. Suppose you create an array to hold 100 usernames and then only insert 10 names to start with. the `.length` value will always return 100 and it won't tell you how many 'real' data items you have stored.

We can solve these problems, and the following sections show how. Be warned, they are fiddly!

Making an Array Bigger

To deal with the problem of your array not being big enough, you have to create a new one and *copy* your data across. You need code like this (assuming you have declared final variables `TOO_SMALL` and `BIG_ENOUGH`

```
//Make our first – too small - list
int[] list = new int[TOO_SMALL];

...

//We get to here and find we need to make list bigger
//Create our new array
int[] tempList = new int[BIG_ENOUGH];

//Loop to copy our data across
for(int i = 0; i < list.length; i++) {
	tempList[i] = list[i];
}

//Make the 'old' point to the 'new' one
list = tempList;
```

We have to:

- **Declare a new array** – the array `tempList` is bigger than the old one
- **Copy the data** – notice we need a loop to copy the data items individually
- **Make the old array 'point' to the new one** – the last line makes the old array name (which is the one our code is using) refer to the data in the new array.

A common mistake is to leave out the loop assuming the line:

```
list = tempList;
```

will be enough to copy the data*. But doing that will only make the name list refer to the (still empty) array* `tempList`. This will *throw away* all our data!

Inserting Elements

Here's a program to handle inserting elements in an array (assuming the array is in total big enough to start with).

```
class Insert{
	public static void main(String[] args) {
		int[] data = new int[10];
		int insertNumber;
		int value;

		Scanner in = new Scanner(System.in);

		//Keep track of amount of data in the array
		int count = 0;
```

```
          //Infinite loop – bit sloppy
          while(true){

                //Read in where we're going to insert
                System.out.print("Enter element no. to “ +
                 “insert between 0 and " + count + ": ");

                do{
                      insertNumber = in.nextInt();
                } while ((insertNumber < 0) ||
                        (insertNumber > count));

                //Read in the actual value to insert
                System.out.print("Enter value: ");
                value = in.nextInt();

                //Increase count of the size of the array
                count++;

                /*Copy the elements beyond the one we
                insert 'forward' one place in the array
                Notice we have to count from the back of
                the array forward to avoid
                overwriting data. This is a *serious* nuisance
                 – imagine if the array was really long */

                for(int i = count; i > insertNumber; i--){
                      data[i] = data[i-1];
                }
                //Actually insert the element
                data[insertNumber] = value;

                //Print out the contents
                System.out.println("Array:");
                for(int i = 0; i < count; i++){
                      System.out.println(data[i]);

                }
          }
     }
}
```

This is a bit longer than it needs to be because of the long comment in bold – but basically you have to:

- **Copy the data one place forward** – you need to move all the data *after* the insertion point one element further on to open up a space.
- **Insert the new item** – only then can you put the new data item in

This is annoying – suppose you had an array of a million elements and wanted to insert right at the start: you'd have to move one million array elements along one place.

Exercise

Make sure you understand we we have to start copying data from the end of the array and count down. Try entering the program and changing it so that the copying is done by counting up from the insertion point:

```
for(int i = insertNumber; i < count; i++){
```

Look at the result and work out what's gone wrong (then you'll understand why we do it by counting down from the end).

Deleting an Element

Deleting is quite similar to inserting – the difference is you need to copy the array elements back one place in the array (and you need to start at the front of the block of elements you are moving, not the back like inserting). Here's the code for you to read without comment – note there are several versions of of the removal program on Blackboard for you to look at.

```
class Remove {
     public static void main(String[] args) {
          //Dummy data to show what’s happening
          double[] data = {1.5, 2.3, 4.7, 5.2, 6.0, 7.9};

          int deleteNumber;

          //Print out the starting contents
          System.out.println("Before:");
          for(double element : data){
               System.out.println(element);
          }

          /*Keep track of the length of the
          array excluding the
          element we are going to "delete"*/
          int count = data.length;

          /*Read in the index of the array element
          to delete notice the use or OR (||) in the
          do-while loop*/
          Scanner in = new Scanner(System.in);
          System.out.print("Enter element no. “ +
               “to remove between 0 and " +
               (data.length - 1) + ": ");

          do{
               deleteNumber = in.nextInt();
          } while ((deleteNumber < 0) ||
                    (deleteNumber > data.length - 1));

          count--;

          /*Copy the elements beyond the one we
          delete 'back' one place in the array*/
          for(int i = deleteNumber; i < count; i++){
               data[i] = data[i+1];
          }

          //Print out the final contents
          System.out.println("After:");
          for(int i = 0; i < count; i++){
               System.out.println(data[i]);

          }
     }
}
```

Keeping Track of Elements

The last problem we listed was keeping track of how many ‘real’ data items are present in an array. We’ve actually solved that problem in the insert and delete examples – we introduced a variable `count` that keeps track: we just need to remember to add/subtract one when we add/delete elements. In the case of the delete example, we initially set `count` to `data.length` (because Java will have created an array exactly the right size for the data we've listed). After that, every time we delete an element, we reduce the size of `count` by 1 – *but* `data.length` *will always remain the same as it was at the start* because we are not reducing the size of the array (just taking elements out and leaving blanks – but Java does not know/care they are 'blank').

KEY POINT: Arrays are Good for Some Things

Although we managed to solve the problems we listed, the solutions are quite fiddly, and it's easy to make mistakes. In practice arrays are best suited to situations where:

- **You know how much data you have to start with** – so you don't need to resize the array.
- **You are only ever adding (and maybe deleting) data from the end** – so you don't have to move data elements when you insert or remove items.
- **All the data is available at the start (or at least the same time)** – so you don't have to worry about keeping track of how much 'real' data is in an array.

In practice, the first of these is the most important (I would pretty-much never consider using an array if I had to worry about possibly having to make it bigger at some point); the next is a bit less important (I would not really want to move data around in arrays and would only consider it if the array was small); and the last is least important (I wouldn't worry too much about that one personally). The *ideal* situation is when you can create an array with all the data you need right at the start – for example, like the prime numbers one near the start of the chapter.

However, it's clear we would like a better alternative – and there is one: *ArrayLists*. We'll get onto those in a bit, but first we need to mention multi-dimensional arrays.

Multi-Dimensional Arrays.

The arrays we've seen so far (and the ones you would use most often) are one-dimensional – just lists. However, some data is inherently two dimensional – for example, consider information representing the colours of the pixels in an image. We can also consider data that is three dimensional (or even four dimensional or more). In practice, one dimensional arrays are by far the most common; and higher dimensions are rare. However, two dimensions are reasonably common so it makes sense to know a bit about them.

Technical Aside: Multiple Dimensions of ‘Arrays of Arrays’?

You can actually get into technical arguments about whether programming languages really support multiple dimensional arrays or not – for example, is a two-dimensional array really just an ‘array of arrays’; or a three-dimensional one an ‘array of arrays of arrays’. However, in practice, it doesn’t’ make any difference when you are using them

Here’s a simple example of a program that uses a two-dimensional array.

```
public class WorldPopulation {
   public static void main(String[] args) {
      final int ROWS = 6;
      final int COLUMNS = 7;

      int[][] data = {
            { 106, 107, 111, 133, 221, 767, 1766 },
            { 502, 635, 809, 947, 1402, 3634, 5268 },
            { 2, 2, 2, 6, 13, 30, 46 },
            { 163, 203, 276, 408, 547, 729, 628 },
            { 2, 7, 26, 82, 172, 307, 392 },
            { 16, 24, 38, 74, 167, 511, 809 }
         };

      String[] continents = {
            "Africa",
            "Asia",
            "Australia",
            "Europe",
            "North America",
            "South America"
         };

      System.out.println("                Year 1750 “ +
         “1800 1850 1900 1950 2000 2050");

      // Print data
      for (int i = 0; i < continents.length; i++) {
         // Print the ith row
         System.out.printf("%20s", continents[i]);
         for (int j = 0; j < COLUMNS; j++) {
            System.out.printf("%5d", data[i][j]);
         }
         System.out.println(); // New line at end of row
      }

      // Print column totals
      System.out.print("               World");
      for (int i = 0; i < COLUMNS; i++) {
         int total = 0;
         for (int j = 0; j < ROWS; j++) {
            total += data[j][i];
         }
         System.out.printf("%5d", total);
      }
      System.out.println();
   }
}
```

The array `continents` is an normal one-dimensional array; but the array `data (int[][] data)` is two-dimensional – to access an element of an two-dimensional array, we use the notation `name[row][column]` where `row` is the row we want to access and `column` is the column.

Aside: Formatted Strings

One other new thing in this program – the lines:

```
System.out.printf("%20s", continents[i]);
System.out.printf("%5d", data[i][j]);
System.out.printf("%5d", total);
```

These prints a formatted string (`printf` = 'print formatted') where the bit in `“”` specifies how the data will be formatted on output. The string `“%20s”` means print the argument (in this case `continents[i]`) as a 20-character long formatted string and if the string is shorter than that, pad it with spaces; similarly the string `“%5d”` means print the argument (in this case `data[i][j]` and `total`) as a 5-digit number. There are many other possible format strings you can use but we're not going into this further here (there is much more information in the course text *Java for Everyone*).

ArrayLists: An Alternative to Arrays

Given the problems listed above with Arrays, and given that we do need to be able to create lists of data where we can insert, delete, and keep adding items, we need an alternative – and the one we're going to use is `ArrayList`.

An ArrayList is just one of a large number of ways of structured data types in Java which, together, are called *Collections*. Examples of other `Collection` data types include:

- `HashSet`
- `LinkedList`
- `ConcurrentHashMap`
- `TreeSet`
- `Stack`
- `Vector`

Some of these organize data in different ways (e.g. `TreeSet`); some provide more sophisticated ways of accessing data (e.g. `HashSet`); some have built-in support for dealing with *parallel* programs (multiple programs, or parts of programs, dealing with data at the same time – e.g. `Vector` and `ConcurrentHashMap`). You will see and use some of these later in your degree – but for now `ArrayList` is the most basic (and also probably the most commonly used).

Here's a very simple `ArrayList` program – it simply copies data from an ordinary array to an `ArrayList`:

```
import java.util.ArrayList;
class ArrayListAddToEnd2 {
     public static void main(String[] args) {
          ArrayList<String> data =
               new ArrayList<String>();
          String[] fruit =
               {"lemon", "orange",
               "pineapple", "kiwi", "apple"};

          System.out.println("Array Contents");
          for(int i = 0; i < fruit.length; i++) {
               System.out.println(fruit[lemon]);
          }

          //Copy Array to ArrayList
          for(int i = 0; i < fruit.length; i++) {
               data.add(fruit[i]);
          }

          //\n is newline
          System.out.println("\nArrayList Contents");
          for(int i = 0; i < data.size(); i++) {
               System.out.println(data.get(i));
          }
     }
}
```

We'll go through the key lines of this to explain what is going on.
The line:

```
ArrayList<String> data =new ArrayList<String>();
```

declares and creates the ArrayList. Just like ordinary arrays, or any variable in Java, when you declare it you need to say what type it is – or in this case, what type of data the ArrayList will store. That's what the `<String>` bit does – the first one (`ArrayList<String> data`) declares an `ArrayList` containing String data; the second one (`new ArrayList<String>()`) creates an `ArrayList` containing String data. Notice we don't have to say how 'big' the `ArrayList` is – that's because it will grow and shrink automatically as we add/remove data.

Advanced Aside: Skip if New to Programming

Actually, if you know that you will need at least a certain number of data items in an `ArrayList` you can tell it to create that many empty 'slots' to start with – and there are minor efficiency gains in doing this. But the `ArrayList` will *still* automatically grow if you add more items than you originally asked for.

The Diamond Operator

Looking at the line

```
ArrayList<String> data =new ArrayList<String>();
```

it seems that the second `<String>` is redundant – why do we need it twice? The reason is that sometimes you write *different* things in each case – the details are a bit beyond this module. However, it's very common to write the same thing in both cases, so from Java 7 the *diamond operator* <> was introduced and you can shorten it to:

```
ArrayList<String> data =new ArrayList<>();
```

We'll use this notation from now on.

ArrayList Operations

Going on with our program, we get to this line:

```
data.add(fruit[i]);
```

more generally:

```
data.add(dataItem);
```

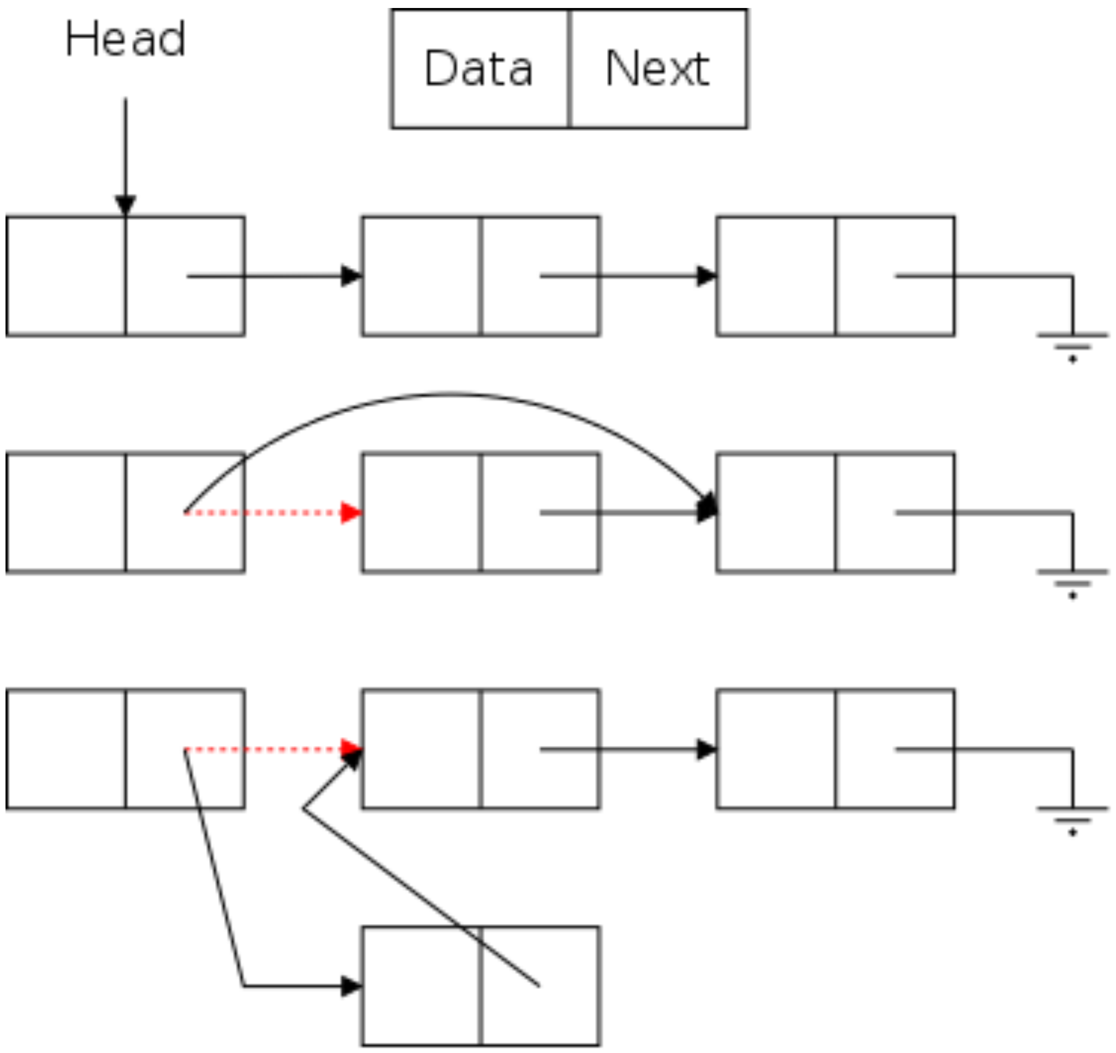
which adds `dataItem` to the *end* of the `ArrayList.` Obviously we need to be able to add at points other than the end – and we can do that with:

```
data.add(index, dataItem);
```

which adds `dataItem` at location index. So `data.add(0,dataItem);` puts it at location 0 (the start); `data.add(5, dataItem);` puts it at location 5 (or causes an error if there are not already 5 or more items in the `ArrayList`. There is no need to 'shift' data out of the way – the `ArrayList` handles that automatically.

Structure of an ArrayList

To see how this *can* work (note the actual implementation of an `ArrayList` is more sophisticated than this), consider the following picture:



Each element of an `ArrayList` consists of two parts – the first is the data (Data in the diagram); and the second is a *reference* (Next in the diagram) which links to the next item in the `ArrayList` (if there is one). The first line shows how three items are linked together in an `ArrayList`; the second shows how you delete an element – you just make the one before it refer to the one after it; the final line shows how to insert an element – you make the one before the insertion refer to the new data item, and make the new item refer to the next one in the sequence. **All the details of *how* this works are hidden from you – you will never have to deal with anything other than the actual data you put in/take out of the ArrayList.** All the details of making it happen are dealt with by the system.

Advanced Aside: Reference vs Pointer

I'm using the term *reference* to describe links between data items. You may have also heard the term *pointer*. They are very similar with subtle differences – for example, a pointer can generally 'point' to any kind of data item while a reference can (a) only point to data items of the right type; and (b) can't usually point to 'simple' data – like an int. Both are really, internally, just *memory addresses*.

Advanced Aside 2: Garbage Collection

In the second line, we seem to be left with a ‘disconnected’ element of the `ArrayList` – the one we deleted isn’t joined to the list any more so what happens to it? In Java and some other languages, the system periodically runs a *Garbage Collection* algorithm, that looks for bits of data that are no longer referred to by anything else, and returns the memory to the system so it can be reused. **Warning:** there are other languages (like C, C++ and Objective C) that *don’t* do this – and you have to do it manually. Failing to do this is a common error that leads to *memory leaks* – the amount of memory a program using growing over time (because the ‘dead’ memory is not released). It can cause serious problems especially for programs that are expected to run for a long time. Fortunately, they can’t happen in Java.

ArrayList Operations – Continued

The next operation in the example above is:

```
data.get(i)
```

which returns the data item at location i. `Fairly straightforward I hope – and equivalent to, in array terms, data[i].` Notice that `data.get(i)` *returns* a value: in this case we print it out, but we could also assign it to a variable:

```
String valueAtLocationEye = data.get(i);
```

Also in the example above is `data.size()` which returns the number of items in the `ArrayList` – this is *not* exactly the same as `data.length` because `data.size()` is the number of items actually in the list at that time.

Other ArrayList Operations

So far we’ve only done adding, reading and checking size – but there are lots of other ArrayList operations too. The most useful ones are listed below (with some comment about how to use them after the list) including the ones we’ve see already for completeness. There are example programs showing how to use most of these on Blackboard and they won’t be repeated here. We assume that we have an ArrayList:

```
ArrayList<String> listData = new ArrayList<>();
```

Operation	**Meaning**
`listData.get(index)`	return the item at location ‘index’
`listData.add(item)`	insert ‘item’ at the end of the list
`istData.add(index, item)`	insert ‘item’ at location ‘index’
`listData.set(index, item)`	replace ‘item’ at location ‘index’
`listData.size()`	return the number of items in the list
`listData.isEmpty()`	returns true if the list is empty
`listData.contains(item)`	returns true if ‘item’ is in the list
`listData.indexOf(item)`	returns the position of the first instance of item in the list (may be more than one) or -1 if it’s not there
`listData.lastIndexOf(item)`	As above but the last instance
`listData.clear()`	Empty the complete list
`listData.remove(index)`	remove the item at location ‘index’
`listData.remove(item)`	remove ‘item’ if present – returns true if it was

Searching ArrayLists

Note that unlike arrays, ArrayList allows you to search for items *without checking each one individually.* For example, here’s a program that returns the location of an item in an array (or -1 if not present):

```
import java.util.ArrayList;
import java.util.Scanner;
import java.util.Arrays;

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

          String[] arrayData = {"lemon", "orange",
                               "pineapple", "kiwi",
                               "apple"};
          ArrayList<String> data =
             new ArrayList<>(Arrays.asList(arrayData));


          Scanner in = new Scanner(System.in);
          System.out.print("What do you want to find? ");
          String fruit = in.nextLine();


          System.out.println(data.indexOf(fruit));
     }
}
```

Notice the line:

```
ArrayList<String> data =
             new ArrayList<>(Arrays.asList(arrayData));
```

which converts an array directly into an `ArrayList` – our earlier example used a loop for this. To use this, we need to import `java.util.Arrays`. Once the data is in an `ArrayList`, we can just directly ask if the item we want is present:

```
data.indexOf(fruit)
```

If we were to do this with an ordinary array we would need a loop to check each item in turn:

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

          String[] arrayData = {"lemon", "orange",
                                "pineapple", "kiwi",
                                "apple"};


          Scanner in = new Scanner(System.in);
          System.out.print("What do you want to find? ");
          String fruit = in.nextLine();

          int count = 0;
          while (count < arrayData.length) {
               if (arrayData[count].equals(fruit) {
                    break; //note use of break
               }
               count++;
          }
          //used a while because we need to access
          //count here
          System.out.println(count);
     }
}
```

Notice also this will only work with an array that is 'full' and has data in all it's elements. Otherwise, we would still need *another* counter to keep track of the actual data – making it even *more* complex.

KEY POINT: ArrayList is Very Flexible and Powerful

In most cases, when dealing with lists of data where you want to do things like add, remove (especially when not at the end) data, and search for things – `ArrayList` is *much* better than arrays. There are a number of examples on Blackboard illustrating other operations (like removing elements and replacing them).

The For-each Loop and Abstraction

Arrays and ArrayLists let us make sense of a 'new' kind of loop – the *for-each* loop. It's really a variation on a for loop but it's more *abstract* (which is a good thing as we'll explain).

One thing you quickly discover in programming is that you often have to go through structured data (like an array or `ArrayList`; or one of the others like those listed above) and do something to each element. Suppose for example you want to add one to each element in an array of integers called `data`. The 'traditional' code for an array (assuming it's full and we can use `data.length`) looks like this:

```
for (int i = 0; i < data.length; i++) {
     data[i]++;
}
```

and for an ArrayList it looks something like this:

```
for (int i =0; i < data.size(); i++) {
     data.set(i, data.get(i) + 1);
}
```

Apart from the fact that in this case the `ArrayList` one is more complex, they are fairly different. Suppose we'd started off using arrays and then changed to an `ArrayList` – we would have to go through the code and rewrite it. But *conceptually* we are doing the same thing in each case and the only reason the code is different is the details of the way we are storing the data. If we could 'step back' from the detail and just tell Java – 'add one to every element in my structured data' then maybe we could eliminate the differences. We can using a *for-each loop*:

```
for(int item  : data) {
     item++;
}
```

In this loop, the value of item is successively set to the elements of the array (or ArrayList) in turn. So, for example:

```
int data = {1, 1, 2, 3, 5, 8, 13};
for(int item  : data) {
     item++;
}
```

changes the array data to contain 2, 2, 3, 4, 6, 9, 14 – the original array with one added to each element.

The format of a for-each loop is:

```
for (type name : list)
```

where:

- `type` is the data type of the items in the array/`ArrayList` (e.g. `int`, `String` etc.)
- `name` is an identifier *you choose* to represent each data item in your list – you use this name to represent each item in your array/`ArrayList` in the body of your for-each loop.
- `list` is the name of the list of data (array/`ArrayList`) you are processing.

The for-each loop is a more *abstract* way of representing the same loop – **and it's exactly the same for arrays and `ArrayLists`.** In fact, *it's the same for all the ways of representing structured data in Java* (and there are many more ways of doing it than we have seen).

Key Point: Use For-Each Loops When Processing Arrays or ArrayLists

If you're systematically processing an array or `ArrayList` you should us a for-each loop instead of a 'normal' for loop.

Key Point: Abstraction

For-each loops are good because:

- **They are simpler** – there are less things you can do wrong.
- **They are more abstract** – they say *what* you want to do not *how* it's done.

Abstraction is what makes it possible for them to be the same for both arrays and `ArrayLists`. Abstraction is a key concept in Computer Science – we try to develop abstract solutions because they are easier to understand and apply to a wider range of problems without change.

Foreach Loop Example

We're going to look at a few examples of programs that use for loops and foreach loops, and the differences between them using arrays and `ArrayLists`. The first program just prints out a list of shapes from an array (leaving out the import statements):

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

          final String[] SHAPE_NAMES = {"Triangle",
                "Square","Pentagon", "Hexagon",
                 "Heptagon", "Octogon",
                 "Nonagon", "Decagon"};

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

Now suppose we change this so it uses an `ArrayList`:

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

          Random rnd = new Random();

          final String[] DATA = {"Triangle",
                "Square","Pentagon", "Hexagon",
                 "Heptagon", "Octogon",
                 "Nonagon", "Decagon"};
          final ArrayList<String> SHAPE_NAMES =
               new ArrayList<>(Arrays.asList(DATA));

          for(int i = 0; i < SHAPE_NAMES.size(); i++) {
               System.out.println(SHAPE_NAMES.get(i));
          }
     }
}
```

The parts of the *loop only* that need to be changed are in bold. Now here's the first example with a foreach loop:

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

          Random rnd = new Random();
          final String[] SHAPE_NAMES = {"Triangle",
                "Square","Pentagon", "Hexagon",
                 "Heptagon", "Octogon",
                 "Nonagon", "Decagon"};

          for(String shape : SHAPE_NAMES) {
               System.out.println(shape);
          }
     }
}
```

If we change this one to use an `ArrayList`:

```
public class ShapeList3 {
     public static void main(String[] args) {

          Random rnd = new Random();

          final int MAXSHAPES = 8;
          final String[] DATA = {"Triangle",
                "Square","Pentagon", "Hexagon",
                 "Heptagon", "Octogon",
                 "Nonagon", "Decagon"};
          final ArrayList<String> SHAPE_NAMES =
               new ArrayList<>(Arrays.asList(DATA));

          for(String shape : SHAPE_NAMES) {
               System.out.println(shape);
          }
     }
}
```

Notice the loop is exactly the same – *we can change how we store the shape data and we do not have to change any of the rest of the code.*

Advanced Aside: Lambda Expression

Version 8 of Java has introduced a new concept called Lambda Expressions (named after the λ calculus – a key concept in *functional programming*.) Lambda expressions allow a wider range of options for various things in Java, including for-each loops. For example, we can re-write the loop that adds one to elements of an array or `ArrayList` above like this:

```
items.forEach(item -> item++);
```

They are handy, especially for *parallel programming* – but conceptually quite complex. If you want to research (and use) them, that's fine. But there's no need to at this stage.

Advanced Aside: The Name…

You may have noticed that except in the Lambda Expression version for-each loops actually just use the word 'for' and not 'for-each' (or 'foreach'). The reason they are called for-each loops is that the concept as it's used in Java originated in another language called C#, which did use the word 'foreach'.

DON'T DO THIS – BAD PRACTICE WITH ARRAYLISTS

When you declare an ArralyList, you can actually leave the <`String`> bit out altogether: So instead of, say:

```
ArrayList<String> names = new ArrayList<>();
```

You can write:

```
ArrayList names = new ArraryList();
```

And you will often see this in old text books (still in the library, unfortunately) and web sites about Java (from early version of Java, before you could say what type the data should be). **But you shouldn't do it**. Doing so says 'make an `ArrayList` that can contain *anything'* – which seems handy **but is very bad practice** (unless you are in the very rare case that this is what you actually need to do) *because it means a whole class of errors can no longer be caught early by the compiler*.

```
class ArrayListNoNo {
    public static void main(String[] args) {
        ArrayList arrayListData = new ArrayList();
        String[] arrayData = {"lemon", "orange",
                        "pineapple", "kiwi", "apple"};

        //Copy Array to ArrayList
        for(String elem : arrayData) {
            arrayListData.add(elem);
        }

        //Print out ArrayList
        for(int i = 0; i < arrayListData.size(); i++) {
            String elem = (String) arrayListData.get(i);
            System.out.println(elem);
        }
    }
}
```

The lines that are different are in **bold.** The first one should be:

```
ArrayList<String> arrayListData = new ArrayList<>();
```

And the second one:

```
String elem = arrayListData.get(i);
```

The first line in our 'wrong' program doesn't tell Java that our `ArrayList` will only contain strings so (a) it can contain anything; and (b) we need to tell it that it actually has `String` data in it when we read the data to print it – which is what the `(String)` in brackets does in the second line (this is a *type cast*). The problem is that we can put in this line
`arrayListData.add(5);`

And *the code will compile and run* – it will, however, crash when it tries to turn the integer 5 into a `String`. The trouble is that in real, complex, programs not all the lines of code are run every time – so it's easily possible for a bug like this to lay hidden and not be found.

Exercise

Try swapping the two lines in bold with the two 'correct' lines directly above; and then try putting in the line that adds an integer – you will get a compiler error because Java now knows that only `String` is allowed in the `ArrayList`. This is good because the compiler *always* checks *all the code.*

Compiler Warnings – Pay Attention to Them!

If you actually compile the "bad" code above, and pay attention to what the compiler output, you will see that it (a) compiles but it (b) gives you a *warning.* This warning says:

```
Note: ArrayListNoNo.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for details.
```

You have two choices at this point:

- You can say "hey, it's only a warning – it still compiles". This is the **wrong answer!**
- You can actually follow the instructions it's giving you. It's telling you to recompile your code like this:

```
javac -Xlint:unchecked ArrayListNoNo.java
```

`where -Xlint:unchecked` is a compiler switch or option, that tells the compiler (in this case) to tell you about any code that does not check the type of the data it deals with. Two things:
the name 'lint' is from an old in-joke - you'll find it refers to "… the name of the undesirable bits of fiber and fluff found in sheep's wool."
You can just type `-Xlint` and leave off the ":`unchecked`" part

If you do this, you'll find the compiler tells you:

```
ArrayListNoNo.java:5: warning: [rawtypes] found raw type:
ArrayList
          ArrayList arrayListData = new ArrayList();
          ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
ArrayListNoNo.java:5: warning: [rawtypes] found raw type:
ArrayList
          ArrayList arrayListData = new ArrayList();
                                        ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
ArrayListNoNo.java:16: warning: [unchecked] unchecked
call to add(E) as a member of the raw type ArrayList
               arrayListData.add(elem);
                                ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
ArrayListNoNo.java:18: warning: [unchecked] unchecked
call to add(E) as a member of the raw type ArrayList
          arrayListData.add(5);
                           ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
4 warnings
```

So it says (a) what's wrong and (b) what lines the problems are on so you can fix them. There are people who *every year* make this *exact mistake* in coursework – and lose credit as a result.

Always pay attention to compiler warnings; always follow the instructions; always fix them (there is credit in the coursework marking for not having any warnings). There are *some* advanced cases where you have to accept warnings – *but not in this module.*