List of Slides

Title

Java Just in Time

John Latham

November 27, 2018

Chapter 14

Arrays

Chapter aims

- An **array** is **list** of items
  - store collection of values in one place
  - arbitrary order or specific order.
- Explore definition and use of arrays
  - **array creation**
  - **array element access**
  - **sort**ing items
  - **partially filled array**s
  - **array extension**
  - **two-dimensional array**s.
- Also reinforce **reference**s
  - have several arrays containing references to same **object**s.

Section 2

Example:

Salary analysis

Aim

AIM: To introduce the basic concepts of **array**s, including **array type**, **array variable**s, **array creation**, **array element access**, **array length** and **empty array**s. We also meet `Math.round()` and revisit `System.out.printf()` and **division** by zero.

Salary analysis

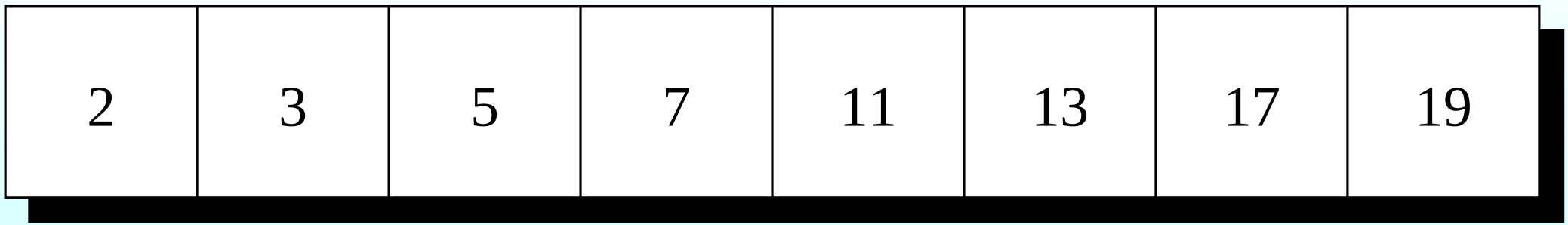
```
001: import java.util.Scanner;
002:
003: /* This program analyses integer salaries entered by the user.
004:    It outputs each salary together with its difference from the
005:    mean of the salaries. There must be at least one salary.
006:  */
007: public class SalaryAnalysis
008: {
009:   public static void main(String[] args)
010:   {
011:     // A Scanner for getting data from the user.
012:     Scanner salariesScanner = new Scanner(System.in);
013:
014:     System.out.print("Enter the number of salaries: ");
015:     int numberOfSalaries = salariesScanner.nextInt();
```

Salary analysis

- Problem: need to have all salaries before compute average
  - but need average before output analysis of each one.
- How can we store them all?

Array

- An **array** is ordered collection (**list**) of items
  - of some particular **type**
  - fixed size.
- Items stored in adjacent **computer memory** locations at **run time**.
- E.g. an array of 8 `int` values.



Array

- Each **array element** contains a value
  - can be changed
  - i.e. each element is separate **variable**.
- But array as a whole is single entity.
- Similar to idea of **object**s having **instance variable**s
  - but elements of array must all be same type.
- Indeed, arrays in Java *are* **object**s.

Type: array type

- Java **array**s are **object**s
  - but treated differently from **instance**s of **class**es.
- To denote **array type** write **type** of **array element**s followed by `[]`.
- Type of elements known as **array base type**.
- E.g. `int[]` – arrays with **`int`** as base type
  - each element is an **`int`** variable.
- E.g. `String[]`
  - each element can contain **reference** to a `String` object.

Salary analysis

Coffee time: You have actually met arrays before in this book, in fact we have been using them frequently since the very beginning! Where, and what for?

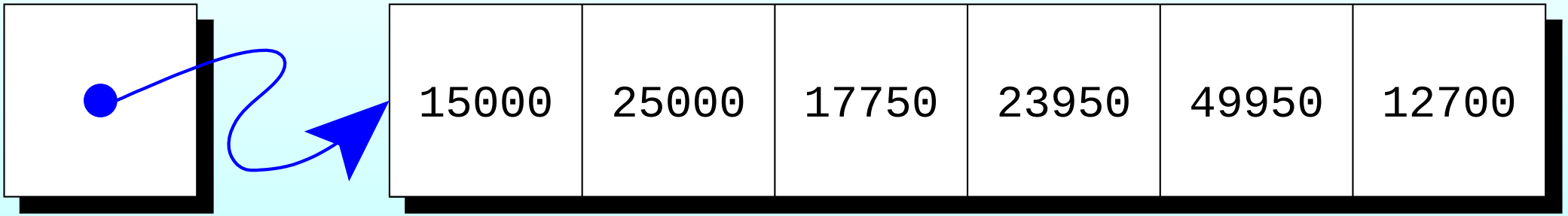
Variable: of an array type

- Can have **variable**s of **array type**.
- E.g.

```
int[] salaries;
```

- As **array**s are **object**s, are accessed via **reference**s
  - variable of array type at **run time** holds *reference* to an array
  - or **null reference**.
- E.g.

`int[] salaries`



Array: array creation

- Create **array**s using **reserved word** `new`
  - just like for other **object**s.
- Give **array base type** then array size in square brackets.
- E.g.

```
new double[10]
```

  At **run time** creates **new** array and yields **reference** to it.

- E.g.

```
double[] myFingerLengths = new double[10];
```

Array: array creation

- Thanks to references, size of array does not need to be known at **compile time**
  - **compiler** does not need to allocate memory for it
  - can create array of right size for actual **data** being processed.
- E.g.

```
int noOfEmployees = Integer.parseInt(args[0]);

String[] employeeNames = new String[noOfEmployees];
```

Salary analysis

- Have already asked user how many salaries there are.

```
017:     // Salaries are ints, stored in an array.
018:     int[] salaries = new int[numberOfSalaries];
```

Array: element access

- The **array element**s are accessed via **array index**
  - whole number **greater than or equal** to zero
  - first element indexed by `0`, second by `1`,. . . .
- To access element, use **reference** to array
  followed by index in square brackets.
- E.g.

```
double[] myFingerLengths = new double[10];
```

  Somehow get lengths of my fingers and thumbs in array, then:

```
double myTotalFingerLength = 0;
for (int index = 0; index < 10; index++)
  myTotalFingerLength += myFingerLengths[index];
```

Array: element access

- Arrays are like ordinary **object**s
  - array elements are like **instance variable**s
  - but number of them chosen when array created
  - and all same **type**
  - and 'named' by index rather than names
  - and accessed via different **syntax**.
- Not much difference then! ;-)

Salary analysis

- Obtain salaries one by one in **for loop**.

```
020:     // Obtain the salaries from the input.
021:     for (int index = 0; index < numberOfSalaries; index++)
022:     {
023:       System.out.print("Enter salary # " + (index + 1) + ": " );
024:       salaries[index] = salariesScanner.nextInt();
025:     } // for
```

- Then **loop** through to get sum.

```
027:     // Now compute the sum of the salaries.
028:     int sumOfSalaries = 0;
029:     for (int index = 0; index < numberOfSalaries; index++)
030:       sumOfSalaries += salaries[index];
```

Salary analysis

Coffee time: Could we have combined the above two for loops into one? Would that make the program clearer? Faster?

- Want mean average salary rounded to nearest pound. . . .

Standard API: `Math`: `round()`

- The **class** `java.lang.Math` has **class method** `round()`
  - takes **`double` method argument**
  - **returns** **`long`**
  - nearest whole number to given one.
- Often want to turn result into **`int`**
  - via **cast**.
- E.g.

```
int myPennies = ... Obtain this somehow.
int myNearlyPounds = (int) Math.round(myPennies / 100.0);
```

Salary analysis

```
032:     // Compute the mean, which is a double, not an integer.
033:     double meanSalary = sumOfSalaries / (double)numberOfSalaries;
034:
035:     // But we also want to round it to simplify the results.
036:     int meanSalaryRounded = (int) Math.round(meanSalary);
037:
038:     // Produce the results.
039:     System.out.println();
040:     System.out.println("The mean salary is:\t" + meanSalary);
041:     System.out.println("which rounds to:\t" + meanSalaryRounded);
042:     System.out.println();
```

Standard API: System: out.printf(): string item

- System.out.printf() can print a String
  - use s as conversion **character** in **format specifier**.
- E.g.

```
System.out.println("123456789012345");
System.out.printf("%15s%n", "Hello World");
```

produces:

```
123456789012345
     Hello World
```

Standard API: System: out.printf(): string item

- If item is reference to some other **object** (i.e. not a string)
  - `toString()` is used.
- E.g. assume `Point` **class** is defined as expected:

```
System.out.println("123456789012345");
System.out.printf("%15s%n", new Point(3, 4));
```

produces:

```
123456789012345
        (3.0,4.0)
```

Standard API: System: out.printf(): fixed text and many items

- System.out.printf() can have format string with
  - more than one **format specifier**
  - more than one value to be printed
  - other text not part of a format specifier.
- Also, if no width given in format specifier then natural width is used.
- E.g.

```
Point p1 = new Point(3, 4);
Point p2 = new Point(45, 60);
System.out.printf("The distance between %s and %s is %1.2f.%n",
                  p1, p2, p1.distanceFromPoint(p2));
```

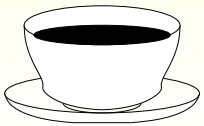
produces:

```
The distance between (3.0,4.0) and (45.0,60.0) is 70.00.
```

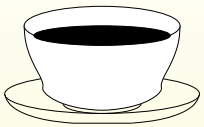
Salary analysis

```
044:     for (int index = 0; index < numberOfSalaries; index++)
045:     {
046:       int differenceFromMean = salaries[index] - meanSalaryRounded;
047:       String comparisonToMean = differenceFromMean == 0
048:                                 ? "zero difference from"
049:                                 : (differenceFromMean < 0
050:                                    ? "less than" : "greater than");
051:       System.out.printf("Person %2d earns %5d, which is %5d %s the mean%n",
052:                         (index + 1), salaries[index],
053:                         Math.abs(differenceFromMean), comparisonToMean);
054:     } // for
055:   } // main
056:
057: } // class SalaryAnalysis
```

Salary analysis



Coffee time: Notice the **nested conditional expressions** in the code above. Do you think that is an acceptable style, or is it pushing the boundaries of code clarity?



Coffee time: Before reading on, predict what the results will look like for the following input salaries. `15049, 49959, 25750, 24627, 12523, 19852`

Trying it

- Carefully ensured one salary exactly **equal** to rounded mean.

Console Input / Output

```
$ java SalaryAnalysis
Enter the number of salaries: 6
Enter salary # 1: 15049
Enter salary # 2: 49959
Enter salary # 3: 25750
Enter salary # 4: 24627
Enter salary # 5: 12523
Enter salary # 6: 19852

The mean salary is:     24626.666666666668
which rounds to:        24627

Person  1 earns 15049, which is  9578 less than the mean
Person  2 earns 49959, which is 25332 greater than the mean
Person  3 earns 25750, which is  1123 greater than the mean
Person  4 earns 24627, which is     0 zero difference from the mean
Person  5 earns 12523, which is 12104 less than the mean
Person  6 earns 19852, which is  4775 less than the mean
$ _
```

Run

Trying it

- Two salaries.

Console Input / Output

```
$ java SalaryAnalysis
Enter the number of salaries: 2
Enter salary # 1: 15000
Enter salary # 2: 25000

The mean salary is:     20000.0
which rounds to:        20000

Person  1 earns 15000, which is  5000 less than the mean
Person  2 earns 25000, which is  5000 greater than the mean
$ _
```

Run

Trying it

- Odd, but works with just one salary.

Console Input / Output

```
$ java SalaryAnalysis
Enter the number of salaries: 1
Enter salary # 1: 15000

The mean salary is:     15000.0
which rounds to:        15000

Person  1 earns 15000, which is     0 zero difference from the mean
$ _
```

Run

Trying it

- What if no salaries?

Console Input / Output

```
$ java SalaryAnalysis
Enter the number of salaries: 0

The mean salary is:     NaN
which rounds to:        0
$ _
```

Run

- Program has **evaluate**d `0 / (double)0`
  - does *not* produce **exception**!

Expression: arithmetic: `double` division: by zero

- With **double division**, if denominator zero but numerator not zero
  - get model of **infinity**
  - represented by `System.out.println()` (etc) as `Infinity`.
- If both numerator and denominator zero
  - get model of **not a number**
  - represented by `System.out.println()` (etc) as `NaN`.
- Observe difference from **integer division**
  - produces **exception** if denominator zero.

Double division by zero: not a number

Coffee time: By looking at the **API** on-line documentation of the `Math.round()` **class method**, figure out why `NaN` gets 'rounded' to `0` in the `SalaryAnalysis` program when it is given no salaries.

An empty array is still an array!

- Running with no salaries shows can have **empty array**
  - no elements, **array length** zero.

Array: length

- Every **array** has **public instance variable** called `length`
  - **type** `int`
  - contains **array length**
  - **final variable** – we cannot change value.
- E.g.

```
int[] myArray = new int[25];
int myArrayLength = myArray.length;
```

variable `myArrayLength` has value 25.

Array: empty array

- When create **array** we give number of **array element**s
  - can be zero!
- An **empty array** may not seem much use?
  - But still exists
  - e.g. can access its **array length**.
- E.g. this outputs zero:

```
int[] myEmptyArray = new int[0];
System.out.println(myEmptyArray.length);
```

- But next code causes **run time error** (`NullPointerException`):

```
int[] myNonArray = null;
System.out.println(myNonArray.length);
```

  - there is no array, so cannot ask for its length!

Coursework: Mark analysis

(Summary only)

Write a program that analyses student coursework marks.

Section 3

Example:
Sorted salary analysis

Aim

AIM: To reinforce **array**s and introduce the idea of **sort**ing, together with one simple sorting **algorithm**. We also introduce the **for-each loop**, and have an array as a **method parameter** to a **method**.

Sorted salary analysis

- Same as last example, except report salaries in ascending order.
  - Have separate **class method** to **sort array**.

```
001: import java.util.Scanner;
002:
003: /* This program analyses integer salaries entered by the user.
004:    It outputs each salary together with its difference from the
005:    mean of the salaries. There must be at least one salary.
006:    The salaries are output in ascending order.
007:  */
008: public class SalaryAnalysis
009: {
010:   public static void main(String[] args)
011:   {
012:     // A Scanner for getting data from the user.
013:     Scanner salariesScanner = new Scanner(System.in);
014:
```

Sorted salary analysis

```
015:     System.out.print("Enter the number of salaries: ");
016:     int numberOfSalaries = salariesScanner.nextInt();
017:
018:     // Salaries are ints, stored in an array.
019:     int[] salaries = new int[numberOfSalaries];
020:
021:     // Obtain the salaries from the input.
022:     for (int index = 0; index < numberOfSalaries; index++)
023:     {
024:       System.out.print("Enter salary # " + (index + 1) + ": " );
025:       salaries[index] = salariesScanner.nextInt();
026:     } // for
```

- Code to calculate sum of salaries is better expressed using **for-each loop**. . . .

Statement: for-each loop: on arrays

- Since Java 5.0: **enhanced for statement**
  - commonly known as **for-each loop**.
- E.g.

```
double[] myFingerLengths = new double[10];

... Code here to assign values to the array elements.
```

find sum of them:

```
double myTotalFingerLength = 0;
for (double fingerLength : myFingerLengths)
  myTotalFingerLength += fingerLength;
```

Statement: for-each loop: on arrays

```
double myTotalFingerLength = 0;
for (double fingerLength : myFingerLengths)
  myTotalFingerLength += fingerLength;
```

- Loop over all elements in **array reference**d by `myFingerLengths`
  - store each in turn in `fingerLength`
  - add to `myTotalFingerLength`.
- 'For each `fingerLength` in `myFingerLengths`
  - add `fingerLength` to `myTotalFingerLength`'.

Statement: for-each loop: on arrays

```
double myTotalFingerLength = 0;
for (double fingerLength : myFingerLengths)
  myTotalFingerLength += fingerLength;
```

- Shorthand for:

```
double myTotalFingerLength = 0;
for (int index = 0; index < myFingerLengths.length; index++)
{
  double fingerLength = myFingerLengths[index];
  myTotalFingerLength += fingerLength;
} // for
```

Statement: for-each loop: on arrays

- General case (assume `SomeType[] anArray`):

```
for (SomeType elementName : anArray)
  ... Statement using elementName.
```

shorthand for:

```
for (int index = 0; index < anArray.length; index++)
{
  SomeType elementName = anArray[index];
  ... Statement using elementName.
} // for
```

Statement: for-each loop: on arrays

- For-each can and should be used instead of for loop
  - when wish to loop over all elements of single array
  - and **array index** is *only* used to access elements
    - i.e. where element values matter
      but position in array not directly used
  - and only one array.
- E.g. following cannot be replaced with for-each:

```
int weightedSum = 0;
for (int index = 0; index < numbers.length; index++)
  weightedSum += numbers[index] * index;
```

- Nor this:

```
for (int index = 0; index < numbers.length; index++)
  otherNumbers[index] = numbers[index];
```

Statement: for-each loop: on arrays

- Common error –
  think that for-each can be used to *change* array elements.
- E.g. following **compile**s without errors:

```
int[] numbers = new int[100];
for (int number : numbers)
  number = 10;
```

shorthand for:

```
for (int index = 0; index < numbers.length; index++)
{
  int number = numbers[index];
  number = 10;
} // for
```

which achieves nothing!

Sorted salary analysis

```
028:     // Now compute the sum of the salaries.
029:     int sumOfSalaries = 0;
030:     for (int salary : salaries)
031:       sumOfSalaries += salary;
032:
033:     // Compute the mean, which is a double, not an integer.
034:     double meanSalary = sumOfSalaries / (double)numberOfSalaries;
035:
036:     // But we also want to round it to simplify the results.
037:     int meanSalaryRounded = (int) Math.round(meanSalary);
038:
039:     // Sort the salaries into ascending order.
040:     sort(salaries);
041:
```

Sorted salary analysis

```
042:     // Produce the results.
043:     System.out.println();
044:     System.out.println("The mean salary is:\t" + meanSalary);
045:     System.out.println("which rounds to:\t" + meanSalaryRounded);
046:     System.out.println();
047:
048:     for (int index = 0; index < numberOfSalaries; index++)
049:     {
050:       int differenceFromMean = salaries[index] - meanSalaryRounded;
051:       String comparisonToMean = differenceFromMean == 0
052:                                 ? "zero difference from"
053:                                 : (differenceFromMean < 0
054:                                    ? "less than" : "greater than");
055:       System.out.printf("Person %2d earns %5d, which is %5d %s the mean%n",
056:                         (index + 1), salaries[index],
057:                         Math.abs(differenceFromMean), comparisonToMean);
058:     } // for
059:   } // main
```

Sorted salary analysis

Coffee time: Why did we not replace the other two for loops with a for-each loop? What do you think of the following code as an alternative to the first for loop?

```
int salaryNo = 1;
for (int salary : salaries)
{
  System.out.print("Enter salary # " + salaryNo + ": ");
  salary = salariesScanner.nextInt();
  salaryNo++;
} // for
```

Design: Sorting a list

- A **list** e.g. **array** has items in some order
  - perhaps arbitrary.
- Often want to rearrange into *specific* order
  - without losing or gaining any.
- Known as **sort**ing
  - E.g. list of numbers may be sorted into ascending numerical order
  - e.g. list of names may be sorted alphabetically
  - etc..
- Many different sort **algorithm**s: **bubble sort**, **insertion sort**, **selection sort**, **quick sort**, **merge sort**, **tree sort** . . . .

Design: Sorting a list: bubble sort

- A **sort algorithm** – **bubble sort**.
- Pass through **list**, look at adjacent items – swap if wrong order.
- One pass not enough to ensure the list completely sorted
  - more passes made until it is.
- After first pass, 'highest' item must be at end of list.
- E.g.

45	78	12	79	60	17

  - First pass, compare 45 with 78 – okay
  - then 78 with 12 – swap
  - then 78 with 79, etc.
  - 79 moves to end of list.

Design: Sorting a list: bubble sort

Start		45	78	12	79	60	17
$45 <= 78$	okay	45 <=	78	12	79	60	17
$78 > 12$	swap	45	12 <=	78	79	60	17
$78 <= 79$	okay	45	12	78 <=	79	60	17
$79 > 60$	swap	45	12	78	60 <=	79	17
$79 > 17$	swap	45	12	78	60	17 <=	79

- 79 is in place, preceding items still not sorted.
- After second pass, second highest item must be at penultimate place.
  - If N items then $N-1$ passes guarantee whole list sorted.
- First pass looks at $N-1$ pairs
  - next looks at $N-2$ pairs
  - last pass looks at one pair.

Design: Sorting a list: bubble sort

- Results at end of next passes:

Pass						
2	12	45	60	17	78	79
3	12	45	17	60	78	79
4	12	17	45	60	78	79
5	12	17	45	60	78	79

- Pass 5 unnecessary – became sorted after pass 4.

Design: Sorting a list: bubble sort

- Bubble sort **pseudo code**:

```
for passCount = 1 to anArray length - 1
  for pairLeftIndex = 0 to anArray length - 1 - passCount
    if items in anArray at pairLeftIndex and pairLeftIndex + 1
                                are out of order
      swap them over
```

Design: Sorting a list: bubble sort

- Improvement – list may be sorted before $N-1$ passes
  perhaps already sorted at start!

```
int unsortedLength = anArray length
boolean changedOnThisPass
do
  changedOnThisPass = false
  for pairLeftIndex = 0 to unsortedLength - 2
    if items in anArray at pairLeftIndex and pairLeftIndex + 1
                           are out of order
      swap them over
      changedOnThisPass = true
    end-if
  end-for
  unsortedLength--
while changedOnThisPass
```

Sorted salary analysis

Coffee time: Use the bubble sort **algorithm** (on paper) to sort the following numbers into *descending* order rather than ascending order which has already been explored.

45	78	12	79	60	17

Method: accepting parameters: of an array type

- The **method parameter**s of a **method** can be any **type**.
- If **array type** then **method argument** will be
  - **reference** to array of that type
  - or **null reference**.
- E.g. **main method** is given reference to `String[]` **command line argument** array.

Sorted salary analysis

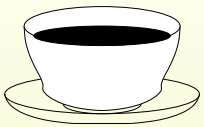
```
062:  // Sort a given array of int into ascending order.
063:  private static void sort(int[] anArray)
064:  {
065:    // Each pass of the sort reduces unsortedLength by one.
066:    int unsortedLength = anArray.length;
067:    // If no change is made on a pass, the main loop can stop.
068:    boolean changedOnThisPass;
```

Sorted salary analysis

```
069:     do
070:     {
071:       changedOnThisPass = false;
072:       for (int pairLeftIndex = 0;
073:            pairLeftIndex < unsortedLength - 1; pairLeftIndex++)
074:         if (anArray[pairLeftIndex] > anArray[pairLeftIndex + 1])
075:         {
076:           int thatWasAtPairLeftIndex = anArray[pairLeftIndex];
077:           anArray[pairLeftIndex] = anArray[pairLeftIndex + 1];
078:           anArray[pairLeftIndex + 1] = thatWasAtPairLeftIndex;
079:           changedOnThisPass = true;
080:         } // if
081:       unsortedLength--;
082:     } while (changedOnThisPass);
083:   } // sort
084:
085: } // class SalaryAnalysis
```

Sorted salary analysis

Coffee time: Suppose we decided we wanted the output to be sorted in *descending* order of salary. What change would we need to make to our program?



Coffee time: Sorting an array is quite a common thing we wish to do in our programs. Clearly it is good for you to see how we can write our own code for sorting, but do you think it is likely that there is in fact some code in a standard **class** somewhere? See if you can find it! Does it allow us to choose which order to sort into?

Trying it

Console Input / Output

```
$ java SalaryAnalysis
Enter the number of salaries: 6
Enter salary # 1: 15049
Enter salary # 2: 49959
Enter salary # 3: 25750
Enter salary # 4: 24627
Enter salary # 5: 12523
Enter salary # 6: 19852

The mean salary is:     24626.666666666668
which rounds to:        24627

Person  1 earns 12523, which is 12104 less than the mean
Person  2 earns 15049, which is  9578 less than the mean
Person  3 earns 19852, which is  4775 less than the mean
Person  4 earns 24627, which is     0 zero difference from the mean
Person  5 earns 25750, which is  1123 greater than the mean
Person  6 earns 49959, which is 25332 greater than the mean
$ _
```

Run

Trying it

Coffee time: Does the output still make sense? For example, what is the meaning of `Person 1`? While they are only numbers, perhaps it does not matter that the output numbers bear no correspondence to the input ones. Or perhaps it should! How could we modify our program so that the person numbers produced in the output were the position of that salary in the input **list**?

Coursework: Mark analysis with sorting

(Summary only)

Write a program that analyses student coursework marks, and presents the results in a **sort**ed order.

Section 4

Example:

Get a good job

Aim

AIM: To examine **array**s in which the **array element**s are **reference**s to **object**s. In particular, we see how this impacts on **sort**ing with the use of a `compareTo()` **instance method**. We also revisit `System.out.printf()` and meet `String.format()`.

Get a good job

- Variation of previous program:
  - input is **list** of pairs: name of firm, typical salary.
- Analysis as before
  - but keep name of firm and salary together while **sort**ing.

Get a good job

- Two **class**es:

Class list for JobAnalysis

Class	Description
`JobAnalysis`	The main class containing the **main method**. It will read the input data, and make **instance**s of `Job`.
`Job`	An instance of this will represent a firm's name together with their typical salary.

Get a good job

Public method interfaces for class `JobAnalysis`.

Method	Return	Arguments	Description
`main`		`String[]`	The main method for the program.

- Also **private class method** to **sort** `Jobs`.

Get a good job

Public method interfaces for class `Job`.

Method	Return	Arguments	Description
Constructor		`String, int`	Constructs a job with the given employer and salary.
`getEmployer`	`String`		Gives the employer.
`getSalary`	`int`		Gives the salary.
`compareTo`	`int`	`Job`	Compare this job with the given other, to support ordering by ascending salary.
`toString`	`String`		Returns a string representation of the job.

The Job class

```
001: // A class for representing a Job,
002: // comprising a firm's name and their typical salary.
003: public class Job
004: {
005:   // The name of the firm for this instance.
006:   private final String employer;
007:
008:   // Their typical salary.
009:   private final int salary;
010:
011:
012:   // The constructor method.
013:   public Job(String requiredEmployer, int requiredSalary)
014:   {
015:     employer = requiredEmployer;
016:     salary = requiredSalary;
017:   } // Job
```

The Job class

```
020:   // Get the employer.
021:   public String getEmployer()
022:   {
023:     return employer;
024:   } // getEmployer
025:
026:
027:   // Get the salary.
028:   public int getSalary()
029:   {
030:     return salary;
031:   } // getSalary
```

The Job class

```
034:   // Compare this Job with a given other,
035:   // basing the comparison on the salaries, then the employers.
036:   // Returns -ve(<), 0(=) or +ve(>) int. -ve means this one is the smallest.
037:   public int compareTo(Job other)
038:   {
039:     if (salary == other.salary)
040:       return employer.compareTo(other.employer);
041:     else
042:       return salary - other.salary;
043:   } // compareTo
```

Standard API: System: out.printf(): left justification

- System.out.printf() can print things left justified rather than right
  - just place hyphen in front of **format specifier** width.
- E.g.

```
System.out.println("123456789012345X");
System.out.printf("%-15sX%n", "Hello World");
```

produces:

```
123456789012345X
Hello World    X
```

Standard API: `String`: `format()`

- `java.lang.String` has **class method** called `format`
  - introduced in Java 5.0
  - works with **format specifier**s just like `System.out.printf()`
  - but **return**s formatted string rather than prints it.
- E.g.

```
System.out.println(String.format("The distance between %s and %s is %1.2f.",
                                 p1, p2, p1.distanceFromPoint(p2)));
```

- precisely same effect as (observe %n):

```
System.out.printf("The distance between %s and %s is %1.2f.%n",
                  p1, p2, p1.distanceFromPoint(p2));
```

The Job class

```
046:   // Return a string representation.
047:   public String toString()
048:   {
049:     return String.format("%-15s pays %5d", employer, salary);
050:   } // toString
051:
052: } // class Job
```

`JobAnalysis` class

- The **main method**
  - takes input **data** in pairs
    - creates `Job` **object**s
    - stores in **array**
  - sorts array
  - outputs results.

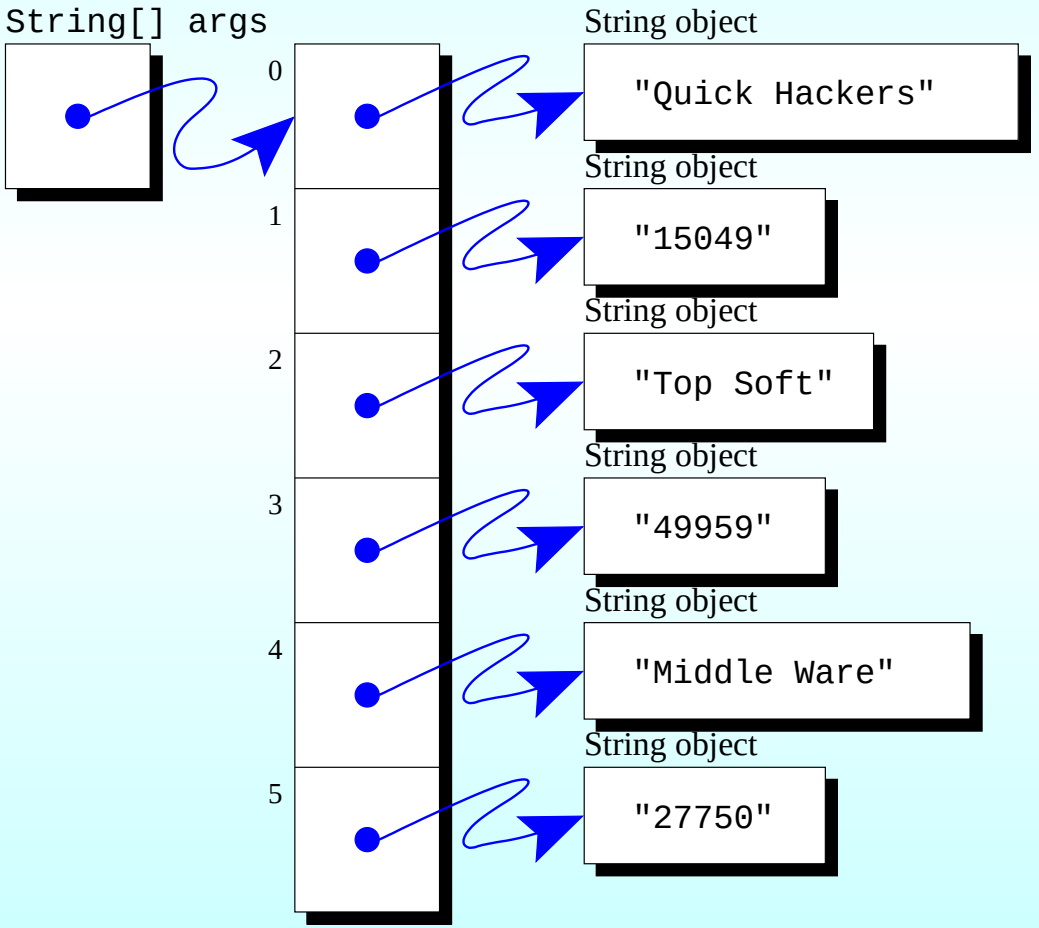
Array: of objects

- An **array base type** can be any **type**
  - including a **class**
  - if so, **array element**s contain **reference**s to objects
    - or **null reference**.

Array: of objects

- E.g. **command line argument**s for `JobAnalysis` (if were not using **standard input**):

```
public static void main(String[] args)
```



JobAnalysis class

```
001: import java.util.Scanner;
002:
003: /* Program to analyse Job information supplied by the user. Each Job comprises
004:    a firm name and their typical salary. Output is mean salary and ascending
005:    sorted list of jobs. There must be at least one job.
006:  */
007: public class JobAnalysis
008: {
009:    public static void main(String[] args)
010:    {
011:      // A Scanner for getting data from the user.
012:     Scanner inputScanner = new Scanner(System.in);
013:
```

JobAnalysis class

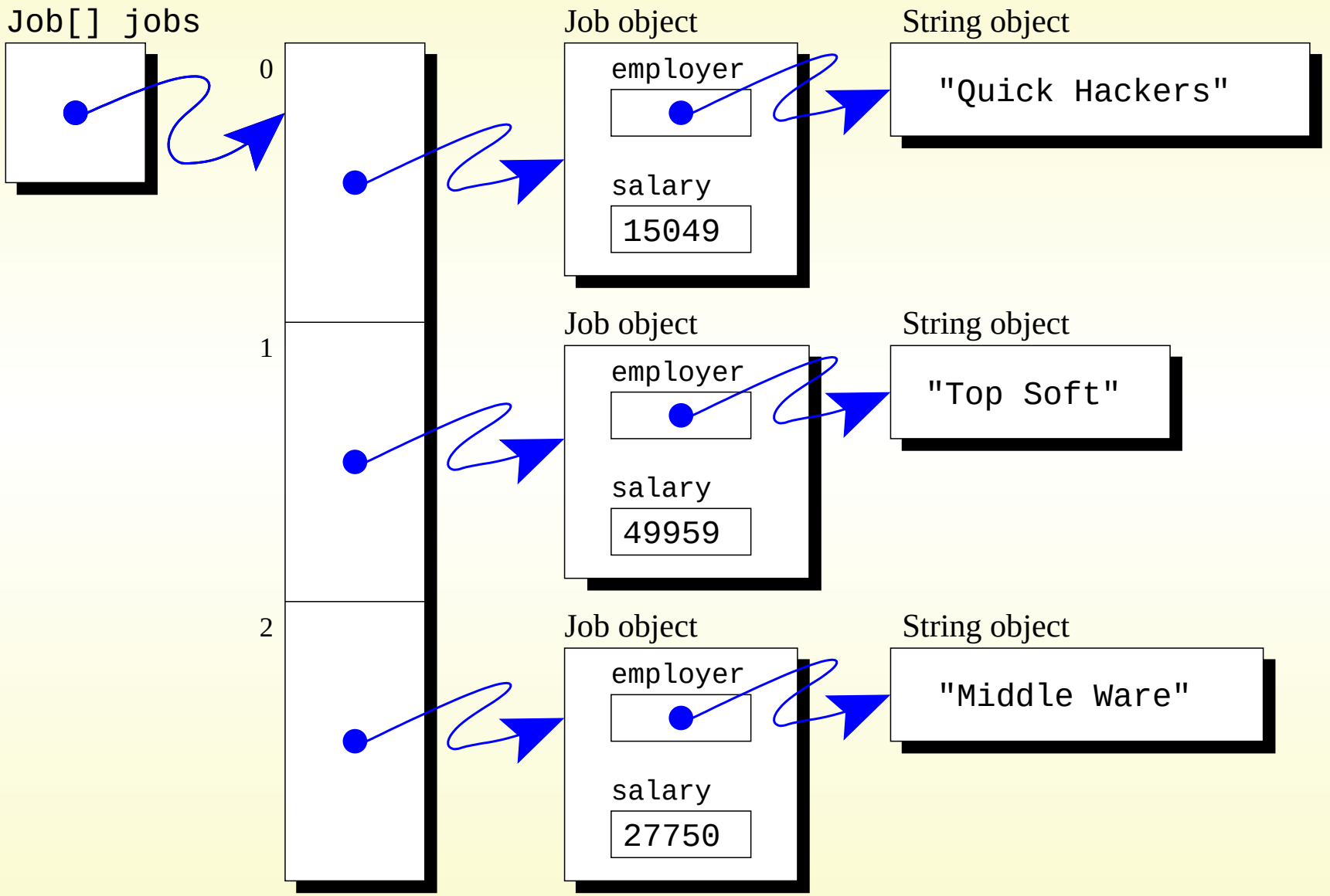
```
014:    System.out.print("Enter the number of jobs: ");
015:    int noOfJobs = inputScanner.nextInt();
016:    // Skip past the end of that line.
017:    inputScanner.nextLine();
018:
019:    // We keep the jobs in an array.
020:    Job[] jobs = new Job[noOfJobs];
```

JobAnalysis class

```
022:    // Read the data in pairs,
023:    // build Job objects and store them in jobs array.
024:    for (int jobCount = 1; jobCount <= noOfJobs; jobCount++)
025:    {
026:      System.out.print("Enter the name of employer " + jobCount + ": ");
027:      String employer = inputScanner.nextLine();
028:      System.out.print("Enter the salary for '" + employer + "': ");
029:      int salary = inputScanner.nextInt();
030:      // Skip past the end of that line.
031:      inputScanner.nextLine();
032:      jobs[jobCount - 1] = new Job(employer, salary);
033:    } // for
```

- Diagram ....

JobAnalysis class



JobAnalysis class

```
035:     // Now compute the sum of the salaries.
036:     int sumOfSalaries = 0;
037:     for (Job job : jobs)
038:       sumOfSalaries += job.getSalary();
039:
040:     // Compute the mean, which is a double, not an integer.
041:     double meanSalary = sumOfSalaries / (double)noOfJobs;
042:
043:     // But we also want to round it to simplify the results.
044:     int meanSalaryRounded = (int) Math.round(meanSalary);
045:
046:     // Sort the jobs by salary into ascending order.
047:     sort(jobs);
048:
049:     // Produce the results.
050:     System.out.println();
051:     System.out.println("The mean salary is:\t" + meanSalary);
052:     System.out.println("which rounds to:\t" + meanSalaryRounded);
053:     System.out.println();
```

JobAnalysis class

- No longer need **array index** appearing in output
  - so can use **for-each loop**.

```
055:    // Output each job.
056:    for (Job job : jobs)
057:    {
058:      int differenceFromMean = job.getSalary() - meanSalaryRounded;
059:      String comparisonToMean = differenceFromMean == 0
060:                              ? "zero difference from"
061:                              : (differenceFromMean < 0
062:                                 ? "less than" : "greater than");
063:      System.out.printf("%s, which is %5d %s the mean%n",
064:                        job, Math.abs(differenceFromMean), comparisonToMean);
065:    } // for
066:  } // main
```

JobAnalysis class

- Class method to sort array uses `compareTo()` **instance method** of `Job` objects.

```
069:   // Sort the given array of Jobs using compareTo on the Job objects.
070:   private static void sort(Job[] anArray)
071:   {
072:     // Each pass of the sort reduces unsortedLength by one.
073:     int unsortedLength = anArray.length;
074:     // If no change is made on a pass, the main loop can stop.
075:     boolean changedOnThisPass;
```

JobAnalysis class

```
076:     do
077:     {
078:       changedOnThisPass = false;
079:       for (int pairLeftIndex = 0;
080:            pairLeftIndex < unsortedLength - 1; pairLeftIndex++)
081:         if (anArray[pairLeftIndex].compareTo(anArray[pairLeftIndex + 1]) > 0)
082:         {
083:           Job thatWasAtPairLeftIndex = anArray[pairLeftIndex];
084:           anArray[pairLeftIndex] = anArray[pairLeftIndex + 1];
085:           anArray[pairLeftIndex + 1] = thatWasAtPairLeftIndex;
086:           changedOnThisPass = true;
087:         } // if
088:       unsortedLength--;
089:     } while (changedOnThisPass);
090:   } // sort
091:
092: } // class JobAnalysis
```

Trying it

Console Input / Output

```
$ java JobAnalysis
Enter the number of jobs: 6
Enter the name of employer 1: Quick Hackers
Enter the salary for 'Quick Hackers': 15049
Enter the name of employer 2: Top Soft
Enter the salary for 'Top Soft': 49959
Enter the name of employer 3: Middle Ware
Enter the salary for 'Middle Ware': 25750
Enter the name of employer 4: Mean Media
Enter the salary for 'Mean Media': 24627
Enter the name of employer 5: OK Coral
Enter the salary for 'OK Coral': 12523
Enter the name of employer 6: Cheaper Cheers
Enter the salary for 'Cheaper Cheers': 19852

The mean salary is:     24626.666666666668
which rounds to:        24627

OK Coral        pays 12523, which is 12104 less than the mean
Quick Hackers   pays 15049, which is  9578 less than the mean
Cheaper Cheers  pays 19852, which is  4775 less than the mean
Mean Media      pays 24627, which is     0 zero difference from the mean
Middle Ware     pays 25750, which is  1123 greater than the mean
Top Soft        pays 49959, which is 25332 greater than the mean
$ _
```

Run

Coursework: Mark analysis with student names and sorting

(Summary only)

Write a program that analyses named student coursework marks, and presents the results in a **sort**ed order.

Section 5

Example:

Sort out a job share?

Aim

AIM: To introduce **partially filled array**s with **array extension**, **array copying** to make a **shallow copy** and **return**ing an **array** from a **method**. We also look at **object sharing** as we have three arrays containing **reference**s to the same **object**s. Along the way we meet the use of a `Scanner` on a **file**, **enum type**s and `split()` on a `String`.

Sort out a job share?

- Elaboration of previous example
  - each **command line argument** is name of **text file**
    - containing employer and salary **data** for kind of job.
  - Data presented twice:
    - **sort**ed by name of employer
    - again by salary.

Sort out a job share?

- Three **class**es.

Class list for JobSurvey

Class	Description
`JobSurvey`	The main class containing the **main method**. It will make an **instance** of `JobList` for each command line argument.
`JobList`	This holds a collection of `Jobs`, one for each data pair in the associated data **file**.
`Job`	An instance of this will represent a firm's name together with their typical salary.

Sort out a job share?

Public method interfaces for class `JobSurvey`.

Method	Return	Arguments	Description
`main`		`String[]`	The main method for the program.

Sort out a job share?

Public method interfaces for class `JobList`.

Method	Return	Arguments	Description
Constructor		`Scanner`	Constructs a job list, reading the information from the given `Scanner`.
`toString`	`String`		Returns a string representation of the job list including the jobs sorted by employer and again by salary.

Sort out a job share?

Public method interfaces for class `Job`.

Method	Return	Arguments	Description
Constructor		`String, int`	Constructs a job with the given employer and salary.
`getEmployer`	`String`		Gives the employer.
`getSalary`	`int`		Gives the salary.
`compareTo`	`int`	`Job, SortOrder`	Compare this job with the given other, to support ordering by employer or salary, as specified by the second argument.
`toString`	`String`		Returns a string representation of the job.

The JobSurvey class

- The **main method** in `JobSurvey`: for each **text file command line argument**:
  - create `Scanner` with access to file
    - pass to `JobList` **constructor method**.

Standard API: Scanner: for a file

- `java.util.Scanner` can be used to read **file**, e.g.

```
import java.io.File;
import java.util.Scanner;
  ...
  Scanner input = new Scanner(new File("my-data.txt"));
```

- `java.io.File` is standard class used to represent file names.
- E.g. reading all lines of file, with help of `hasNextLine()`:

```
while (input.hasNextLine())
{
  String line = input.nextLine();
  ...
} // while
```

The JobSurvey class

- Creating Scanner for file could cause **run time error**
  - more serious kind than so far met: Java won't let us ignore it!
- Handling **exception**s is topic of next chapter – here do minimum:
  - add `throws Exception` to main method heading.

```
001: import java.io.File;
002: import java.util.Scanner;
003:
004: /* Program to report jobs and their salaries.
005:    Each command line argument is the name of a text file containing:
006:      The first line is a name or description of the jobs.
007:      Subsequent lines describe one job, in the format:
008:        Employer (including spaces but not tabs) <TAB> salary
009:    Output is a report for each file containing:
010:      Name or description of the jobs, average salary
011:      Job details in name order and again in salary order.
012:  */
```

The JobSurvey class

```
013: public class JobSurvey
014: {
015:   public static void main(String[] args) throws Exception
016:   {
017:     for (String fileName : args)
018:     {
019:       JobList jobList = new JobList(new Scanner(new File(fileName)));
020:       System.out.println(jobList);
021:       System.out.println();
022:     } // for
023:   } // main
024:
025: } // class JobSurvey
```

The Job class

```
001: // A class for representing a Job,
002: // comprising a firm's name and their typical salary.
003: public class Job
004: {
005:   // The name of the firm for this instance.
006:   private final String employer;
007:
008:   // Their typical salary.
009:   private final int salary;
010:
011:
012:   // The constructor method.
013:   public Job(String requiredEmployer, int requiredSalary)
014:   {
015:     employer = requiredEmployer;
016:     salary = requiredSalary;
017:   } // Job
018:
019:
```

The Job class

```
020:   // Get the employer.
021:   public String getEmployer()
022:   {
023:     return employer;
024:   } // getEmployer
025:
026:
027:   // Get the salary.
028:   public int getSalary()
029:   {
030:     return salary;
031:   } // getSalary
```

- `compareTo()` needs to be able to order by employer or salary....

Variable: final variables: class constant: a set of choices

- Can use **class constant**s to define set of options for users of **class**
  - don't have to know what values are used to model each option.
- E.g. possible directions in game?

```
public static final int UP = 0;
public static final int DOWN = 1;
public static final int LEFT = 2;
public static final int RIGHT = 3;
```

- Code more readable than if numbers are used directly.
- Also more flexible for **source code** maintainer – can change values.

The Job class

- Would have:

```
public static final int SORT_BY_EMPLOYER = 1;
public static final int SORT_BY_SALARY = 2;
```

and

```
public int compareTo(Job other, int sortOrder)
{
  switch (sortOrder)
  {
    case SORT_BY_EMPLOYER: ...
    case SORT_BY_SALARY:   ...
    default:               ...
  } // switch
} // compareTo
```

Variable: final variables: class constant: a set of choices: dangerous

- Use of **`int` class constant**s to model options has two dangers
  - Constants could be used for other purposes
    - e.g. could be used inappropriately in some **arithmetic expression**.
  - Could accidentally use another **`int`** value, not one of constants,

Type: enum type

- Since 5.0 Java has **enum type**s
  - allows us to identify enumeration of named values as **type**.
- E.g. four possible directions in some game:

```
private enum Direction { UP, DOWN, LEFT, RIGHT }
```

- A bit like defining **class** called `Direction`
  - with four **variable**s
    - each referring to a unique **instance** of `Direction`.
- So, e.g.:

```
private Direction currentDirection = Direction.UP;
private Direction nextDirection = null;
```

- Could declare as **public** if want to be available in other classes.

Type: enum type

- Enum types can be used in **switch statement**s.

```
switch (currentDirection)
{
  case UP:    ...
  case DOWN:  ...
  case LEFT:  ...
  case RIGHT: ...
  default:    ...
} // switch
```

The Job class

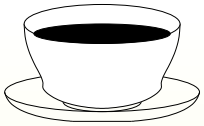
```
034:   // These are the possible sort orders.
035:   // If more are required, then add here and update compareTo.
036:   public enum SortOrder { BY_EMPLOYER, BY_SALARY }
```

The Job class

```
039:  // Compare this Job with a given other,
040:  // basing the comparison on the given sort order.
041:  // Returns -ve(<), 0(=) or +ve(>) int. -ve means this one is the smallest.
042:  public int compareTo(Job other, SortOrder sortOrder)
043:  {
044:    switch (sortOrder)
045:    {
046:      case BY_EMPLOYER: if (employer.equals(other.employer))
047:                          return salary - other.salary;
048:                        else
049:                          return employer.compareTo(other.employer);
050:      case BY_SALARY:   if (salary == other.salary)
051:                          return employer.compareTo(other.employer);
052:                        else
053:                          return salary - other.salary;
054:      default:          return 0;
055:    } // switch
056:  } // compareTo
```

The Job class

- `sortOrder` could be **`null`**.
- Plus other subtle reason why need to have default entry even though both enumeration values are covered. . . .



Coffee time: Imagine that the enum type is defined in one class, and the **switch statement** appears in another. What would happen if a third value was added to the enum type, without the second class being re**compile**d?

Coffee time: If we wished to alter the sorting by salary so that it was descending, instead of ascending, what change would we need to make to `compareTo()`?

The Job class

```
059:   // Return a string representation.
060:   public String toString()
061:   {
062:     return String.format("%-15s pays %5d", employer, salary);
063:   } // toString
064:
065: } // class Job
```

The JobList class

- First line of text from Scanner is description of JobList.

```
001: import java.util.Scanner;
002:
003: /* A JobList holds a list of Job objects, the data for which is read from a
004:    Scanner passed to the constructor. It sorts these by employer and by
005:    salary. The toString method returns a String showing both lists.
006:  */
007: public class JobList
008: {
009:   // The description of this JobList.
010:   private final String description;
```

The `JobList` class

- For convenience, do not require number of jobs stated up front
  - program counts them as they are read.
- But how big should **array** be?
  - Too big – waste memory
  - too small – can't process all jobs.

Array: partially filled array

- A **partially filled array** – not all **array element**s used
  - only leading portion
  - size of portion stored in another **variable**.
- E.g.

```
private final int MAX_NO_OF_ITEMS = 100;
private int noOfItemsInArray = 0;
private SomeType[] anArray = new SomeType[MAX_NO_OF_ITEMS];
```

add item into array – e.g. ignore if full:

```
if (noOfItemsInArray < MAX_NO_OF_ITEMS)
{
  anArray[noOfItemsInArray] = aNewItem;
  noOfItemsInArray++;
} // if
```

Array: array extension

- What if **partially filled array** full, but more data to be added?
  - Use **array extension**:
    - make **new**, bigger **array**
    - copy existing items to it.
- E.g.

```
private static final int INITIAL_ARRAY_SIZE = 100;
private static final int ARRAY_RESIZE_FACTOR = 2;
private int noOfItemsInArray = 0;
private SomeType[] anArray = new SomeType[INITIAL_ARRAY_SIZE];
```

adding item...

Array: array extension

```
if (noOfItemsInArray == anArray.length)
{
  SomeType[] biggerArray
    = new SomeType[anArray.length * ARRAY_RESIZE_FACTOR];
  for (int index = 0; index < noOfItemsInArray; index++)
    biggerArray[index] = anArray[index];
  anArray = biggerArray;
} // if

anArray[noOfItemsInArray] = aNewItem;
noOfItemsInArray++;
```

- New array need not be twice as big
  - just bigger
  - but only one bigger would be slow. . . .

The JobList class

```
012:   // The number of Jobs.
013:   private int noOfJobs;
014:
015:   // The jobs in original order.
016:   // Only the first 0 to noOfJobs - 1 indices are used.
017:   private Job[] jobsInOriginalOrder;
018:
019:   // The jobs in ascending order by employer name.
020:   private final Job[] jobsSortedByEmployer;
021:
022:   // The jobs in ascending order by salary.
023:   private final Job[] jobsSortedBySalary;
024:
025:   // The mean and rounded mean salary.
026:   private final double meanSalary;
027:   private final int meanSalaryRounded;
```

The JobList class

```
030:   // The constructor is given a Scanner from which to read
031:   //     the description of the JobList
032:   //     and then the job data.
033:   public JobList(Scanner scanner)
034:   {
035:     description = scanner.nextLine();
036:     readJobsInOriginalOrder(scanner);
```

- Next copy elements into two more arrays
  - just big enough
  - one for each of the other two **array variable**s.

Method: returning a value: of an array type

- The **return type** of a **method** may be **array type**
  - value will be **reference** to **array**
  - or **null reference**.

The JobList class

```
038:     // Copy the jobs into two arrays.
039:     jobsSortedByEmployer = copyJobArray(jobsInOriginalOrder, noOfJobs);
040:     jobsSortedBySalary = copyJobArray(jobsInOriginalOrder, noOfJobs);
```

- Next sort these arrays
  - one by employer
  - one by salary.

Type: enum type: access from another class

- An **enum type** declared **public** can be used in other **class**es
  - accessed using dot like other kinds of access.
- E.g. if `Direction` defined in `Movement`:

```
Movement.Direction requestedDirection = Movement.Direction.UP;
```

The JobList class

```
042:     // Sort each array into its correct order.
043:     sort(jobsSortedByEmployer, Job.SortOrder.BY_EMPLOYER);
044:     sort(jobsSortedBySalary, Job.SortOrder.BY_SALARY);
045:
046:     // Now compute the sum of the salaries.
047:     int sumOfSalaries = 0;
048:     for (Job job : jobsSortedBySalary)
049:        sumOfSalaries += job.getSalary();
050:
051:     // Compute the mean, which is a double, not an integer.
052:     meanSalary = sumOfSalaries / (double)noOfJobs;
053:
054:     // But we also want to round it to simplify the results.
055:     meanSalaryRounded = (int) Math.round(meanSalary);
056:   } // JobList
```

The JobList class

- When reading jobs
  - start array with initial size
  - create new array for extension if/when required.
- For testing, set values small.

```
059:   // Initial size of the jobsInOriginalOrder array.
060:   private static final int INITIAL_ARRAY_SIZE = 2;
061:
062:   // When jobsInOriginalOrder is full, we extend it by this factor.
063:   private static final int ARRAY_RESIZE_FACTOR = 2;
```

- Have helper method to read one line from scanner and create `Job`....

The JobList class

```
066:   // Read job data from the given Scanner, count them using noOfJobs,
067:   // and store in jobsInOriginalOrder -- extending as required.
068:   private void readJobsInOriginalOrder(Scanner scanner)
069:   {
070:     jobsInOriginalOrder = new Job[INITIAL_ARRAY_SIZE];
071:     noOfJobs = 0;
072:     while (scanner.hasNextLine())
073:     {
074:       // Obtain the next Job.
075:       Job currentJob = readOneJob(scanner);
076:       // Extend the array if it is too small.
```

The JobList class

```
077:       if (noOfJobs == jobsInOriginalOrder.length)
078:       {
079:         Job[] biggerArray
080:           = new Job[jobsInOriginalOrder.length * ARRAY_RESIZE_FACTOR];
081:         for (int index = 0; index < jobsInOriginalOrder.length; index++)
082:           biggerArray[index] = jobsInOriginalOrder[index];
083:         jobsInOriginalOrder = biggerArray;
084:       } // if
085:       // Finally store the Job and update noOfJobs.
086:       jobsInOriginalOrder[noOfJobs] = currentJob;
087:       noOfJobs++;
088:     } // while
089:   } // readJobsInOriginalOrder
```

Standard API: `String`: `split()`

- The **instance method** `java.lang.String.split()`
  - **returns array** of `Strings`
  - each **array element** is portion of the `String`
  - split depending on **method argument**
    - **regular expression** describing what separates the portions.

Standard API: `String`: `split()`

- E.g.

String and regular expression	Resulting array
`"The-cat-sat-on-the-mat".split("-")`	`{ "The", "cat", "sat", "on", "the", "mat" }`
`"The--cat--sat--on--the--mat".split("-")`	`{ "The", "", "cat", "", "sat", "", "on", "", "the", "", "mat" }`
`"The--cat--sat--on--the--mat".split("-+")`	`{ "The", "cat", "sat", "on", "the", "mat" }`
`"The-cat--sat---on----the--mat".split("-+")`	`{ "The", "cat", "sat", "on", "the", "mat" }`

- `"-+"` means "one or more hyphens".

The JobList class

Coffee time: Read the Java **API** on-line documentation to find out more about **regular expressions**.

```
092:   // Read one line of text from the Scanner,
093:   // split it into employer name <TAB> salary,
094:   // create a corresponding Job and return it.
095:   private Job readOneJob(Scanner scanner)
096:   {
097:     String[] jobData = scanner.nextLine().split("\t");
098:     return new Job(jobData[0], Integer.parseInt(jobData[1]));
099:   } // readOneJob
```

- Next instance method to make copy of original array. . . .

Array: shallow copy

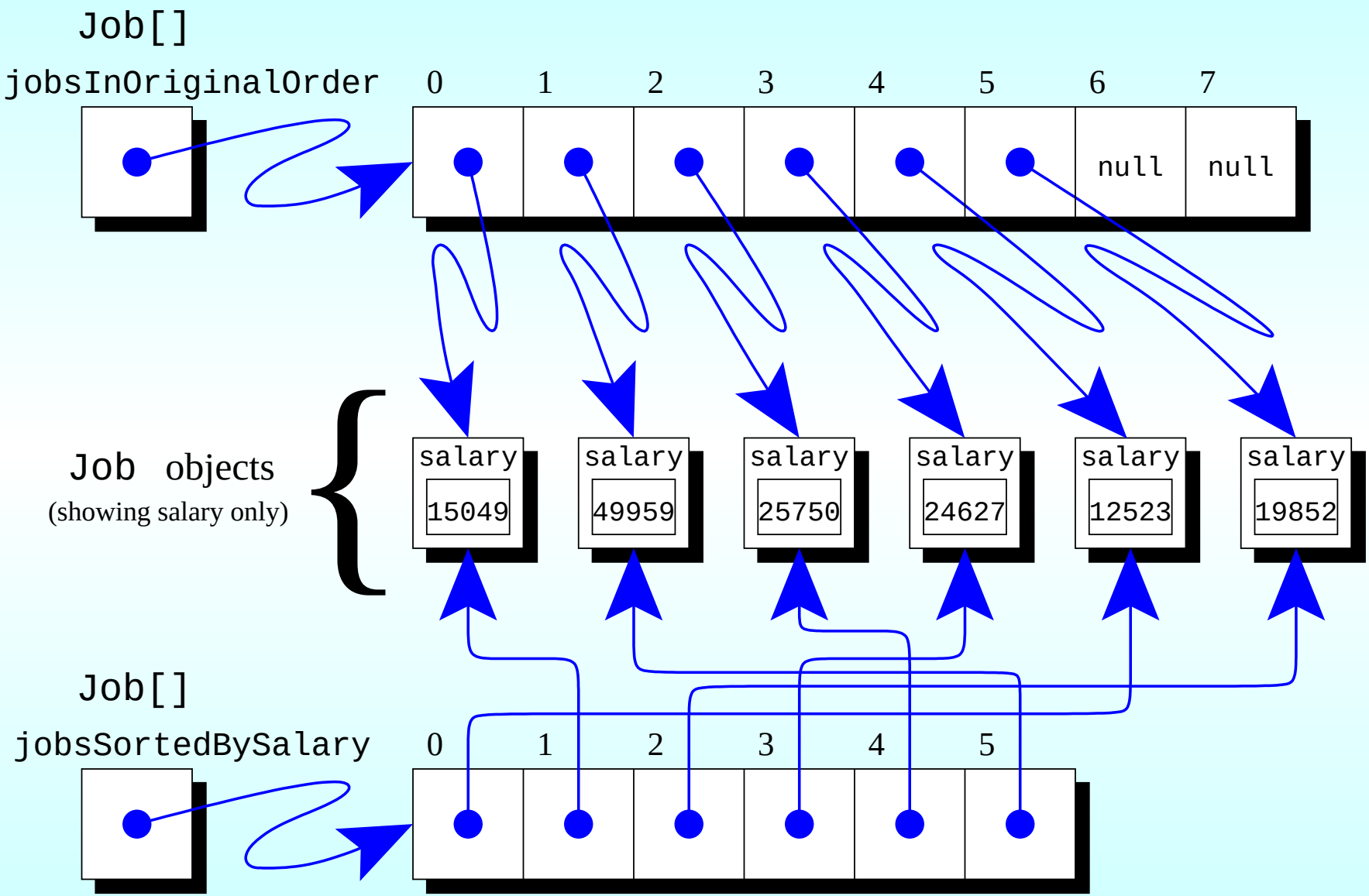
- When copy **array** containing **reference**s to **object**s
  - can make **shallow copy**
    - contains same references
      so objects end up shared by the two arrays
  - or **deep copy**
    - contains references to *copies* of original objects.

The JobList class

```
102:   // Return a shallow copy of given source,
103:   // but only the first dataLength elements.
104:   private Job[] copyJobArray(Job[] source, int dataLength)
105:   {
106:     Job[] result = new Job[dataLength];
107:     for (int index = 0; index < dataLength; index++)
108:       result[index] = source[index];
109:     return result;
110:   } // copyJobArray
```

Coffee time: Why can we not use a **for-each loop** in the above code?

The JobList class



The JobList class

```
113:   // Sort the given array of Jobs
114:   // using compareTo on the Job objects with the given sortOrder.
115:   private void sort(Job[] anArray, Job.SortOrder sortOrder)
116:   {
117:     // Each pass of the sort reduces unsortedLength by one.
118:     int unsortedLength = anArray.length;
119:     // If no change is made on a pass, the main loop can stop.
120:     boolean changedOnThisPass;
```

The JobList class

```
121:     do
122:     {
123:       changedOnThisPass = false;
124:       for (int pairLeftIndex = 0;
125:            pairLeftIndex < unsortedLength - 1; pairLeftIndex++)
126:         if (anArray[pairLeftIndex]
127:               .compareTo(anArray[pairLeftIndex + 1], sortOrder) > 0)
128:         {
129:           Job thatWasAtPairLeftIndex = anArray[pairLeftIndex];
130:           anArray[pairLeftIndex] = anArray[pairLeftIndex + 1];
131:           anArray[pairLeftIndex + 1] = thatWasAtPairLeftIndex;
132:           changedOnThisPass = true;
133:         } // if
134:       unsortedLength--;
135:     } while (changedOnThisPass);
136:   } // sort
```

The JobList class

```
139:   // Return job details sorted by employer name and then salary.
140:   public String toString()
141:   {
142:     return String.format("Job list: %s\tAverage: %f%n%n"
143:                          + "Sorted by employer%s%n%nSorted by salary%s",
144:                          description, meanSalary,
145:                          listOneJobArray(jobsSortedByEmployer),
146:                          listOneJobArray(jobsSortedBySalary));
147:   } // toString
148:
149:
```

The JobList class

```
150:   // Helper method for toString.
151:   private String listOneJobArray(Job[] jobArray)
152:   {
153:     String result = "";
154:     for (Job job : jobArray)
155:     {
156:       int differenceFromMean = job.getSalary() - meanSalaryRounded;
157:       String comparisonToMean = differenceFromMean == 0
158:                                 ? "zero difference from"
159:                                 : (differenceFromMean < 0
160:                                    ? "less than" : "greater than");
161:       result +=
162:         String.format("%n%s, which is %5d %s the mean",
163:                       job, Math.abs(differenceFromMean), comparisonToMean);
164:     } // for
165:     return result;
166:   } // listOneJobArray
167:
168: } // class JobList
```

The `JobList` class

Coffee time: Why did we not declare `noOfJobs` and `jobsInOriginalOrder` as being **final variable**s?

Coffee time: In a previous coffee time you were invited to find out about `StringBuffer`. Do you think the `toString()` instance method above would be a good place to use one?

Trying it

Console Input / Output

```
$ cat programmers.txt
Programmers
Quick Hackers   15049
Top Soft        49959
Middle Ware     25750
Mean Media      24627
OK Coral        12523
Cheaper Cheers  19852
$ cat testers.txt
Testers
Quick Hackers   13999
Top Soft        49059
Middle Ware     24049
Mean Media      23316
OK Coral        10999
Cheaper Cheers  18474
$ _
```

Run

Trying it

Console Input / Output

```
$ java JobSurvey programmers.txt testers.txt
Job list: Programmers   Average: 24626.666667

Sorted by employer
Cheaper Cheers  pays 19852, which is  4775 less than the mean
Mean Media      pays 24627, which is     0 zero difference from the mean
Middle Ware     pays 25750, which is  1123 greater than the mean
OK Coral        pays 12523, which is 12104 less than the mean
Quick Hackers   pays 15049, which is  9578 less than the mean
Top Soft        pays 49959, which is 25332 greater than the mean

Sorted by salary
OK Coral        pays 12523, which is 12104 less than the mean
Quick Hackers   pays 15049, which is  9578 less than the mean
Cheaper Cheers  pays 19852, which is  4775 less than the mean
Mean Media      pays 24627, which is     0 zero difference from the mean
Middle Ware     pays 25750, which is  1123 greater than the mean
Top Soft        pays 49959, which is 25332 greater than the mean

Job list: Testers       Average: 23316.000000

Sorted by employer
Cheaper Cheers  pays 18474, which is  4842 less than the mean
Mean Media      pays 23316, which is     0 zero difference from the mean
...
$ _
```

Run

Trying it

Coffee time: Did you notice a `%f` without a precision (number of decimal places) in the **format specifier** string in our `toString()`? Can you guess what the default number of decimal places is?

Coursework: Random order text puzzle

(Summary only)

Write a random order text line **sort**ing puzzle program.

Section 6

Example:
Diet monitoring

Aim

AIM: To reinforce ideas met so far, and introduce **array initializer** and **array searching**, for which we revisit the **logical operators**.

Diet monitoring

- Program to aid people monitoring their diet
  - record food name and how many grams eaten
  - (**tab character** separated) in text file `diet-diary.txt`.
- E.g.

Console Input / Output

```
$ cat diet-diary.txt
pizza           400
garlic bread    200
cheesecake      260
burger          200
fries           180
milkshake       400
fried chicken   360
wedges          270
$ _
```

Run

Diet monitoring

- Program produces summary table
  how much of each nutritional component was eaten:
  - protein, carbohydrate, etc..
- Reads file `food-details.txt` containing various foods breakdown
  - grams per *kilogram*.

Console Input / Output

```
$ cat food-details.txt
Food            Protein Carb    Fat     Fibre   Sodium
burger          150     200     100     25      12
cheesecake      50      300     200     6       5
fried chicken   225     115     190     15      10
fries           35      400     150     50      12
garlic bread    130     420     95      25      5
milkshake       28      202     32      1       2
pizza           140     300     119     25      8
wedges          210     390     99      41      12
$ _
```

Run

Diet monitoring

- Three **class**es.

Class list for Diet	
Class	**Description**
`Diet`	The **main method** makes an **instance** of `FoodList` from `food-details.txt`. It then accumulates nutritional components from food items listed in `diet-diary.txt`, and outputs those totals.
`FoodList`	This will make an **instance** of `Food` for each item specified in the text file, and store them in an **array**.
`Food`	An instance of this will store a food name together with its nutritional data.

Diet monitoring

Public method interfaces for class `Diet`.

Method	Return	Arguments	Description
`main`		`String[]`	The main method for the program.

Diet monitoring

Public method interfaces for class `FoodList`.

Method	Return	Arguments	Description
Constructor		`Scanner`	Constructs a food list, reading from the given `Scanner`.
`findFood`	`Food`	`String`	Return (a **reference** to) the `Food` **object** corresponding to the food named by the given `String`, or **`null`** if it is not recognized.

Diet monitoring

Public method interfaces for class Food.

Method	Return	Arguments	Description
Constructor		`String`	Constructs a `Food` with the details specified by the given `String`. This will be a line of text from the `food-details.txt` data text file.
`getName`	`String`		Returns the food name.

Diet monitoring

Public method interfaces for class `Food`.

Method	Return	Arguments	Description
`componentMilliGramsForWeight`	`int[]`	`int`	Produces an array of nutritional component amounts corresponding to a given number of grams of the food being consumed.

The Food class

- Have **class constant array** of nutritional component names.

Array: array creation: initializer

- Can create actual array at same time as declaring **array variable**
  - using **array initializer**
    - list **array element**s, instead of stating length.
- E.g.

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13, 17, 19};
```

- Shorthand for:

```
int[] smallPrimes = new int[8];
...
smallPrimes[0]=2;  smallPrimes[1]=3;  smallPrimes[2]=5;
smallPrimes[3]=7;  smallPrimes[4]=11; smallPrimes[5]=13;
smallPrimes[6]=17; smallPrimes[7]=19;
```

The Food class

```
001: // Representation of a food, as a name
002: // together with nutritional data in grams per kilogram.
003: public class Food
004: {
005:   // This defines the spelling and order of nutritional components.
006:   public static final String[] NUTRITIONAL_COMPONENTS
007:     = { "Protein", "Carb", "Fat", "Fibre", "Sodium" };
008:
009:   // The name of this food.
010:   private final String name;
011:
012:   // Nutritional data in the same order as NUTRITIONAL_COMPONENTS.
013:   private final int[] nutrientGramsPerKilogram
014:     = new int[NUTRITIONAL_COMPONENTS.length];
```

The Food class

```
017:   // Constructor is given name and data as tab separated parts of a string.
018:   public Food(String details)
019:   {
020:     String[] detailParts = details.split("\t+");
021:     name = detailParts[0];
022:     for (int index = 0; index < NUTRITIONAL_COMPONENTS.length; index++)
023:       nutrientGramsPerKilogram[index]
024:         = Integer.parseInt(detailParts[index + 1]);
025:   } // Food
026:
027:
028:   // Accessor for name.
029:   public String getName()
030:   {
031:     return name;
032:   } // getName
```

The Food class

```
035:   // Returns the number of milligrams of each component
036:   // for the given number of grams consumed.
037:   public int[] componentMilliGramsForWeight(int grams)
038:   {
039:     int[] result = new int[NUTRITIONAL_COMPONENTS.length];
040:     for (int index = 0; index < NUTRITIONAL_COMPONENTS.length; index++)
041:       result[index] = nutrientGramsPerKilogram[index] * grams;
042:     return result;
043:   } // componentMilliGramsForWeight
044:
045: } // class Food
```

The Food class

Coffee time: By declaring the array `NUTRITIONAL_COMPONENTS` as a **final variable**, have we made it impossible for code (inside or outside of this class) to alter the order or spellings of the components?

The FoodList class

```
001: import java.util.Scanner;
002:
003: // Keeps a list of food items, and provides a search facility.
004: public class FoodList
005: {
006:   // For array extension of foodList.
007:   private static final int INITIAL_ARRAY_SIZE = 100, ARRAY_RESIZE_FACTOR = 2;
008:
009:   // The food details are stored in a partially filled array
010:   // with an associated count.
011:   private final int noOfFoodItems;
012:   private final Food[] foodList;
013:
014:
```

The FoodList class

```
015:   // The constructor reads the food details from the given scanner
016:   // and stores them in foodList, extending as necessary.
017:   public FoodList(Scanner scanner)
018:   {
019:     // The first line is just titles.
020:     scanner.nextLine();
021:     Food[] foodListSoFar = new Food[INITIAL_ARRAY_SIZE];
022:     int noOfFoodItemsSoFar = 0;
```

The FoodList class

```
023:     while (scanner.hasNextLine())
024:     {
025:       // Food constructor parses the whole line.
026:       Food latestFood = new Food(scanner.nextLine());
027:       // Extend the array if it is full.
028:       if (noOfFoodItemsSoFar == foodListSoFar.length)
029:       {
030:         Food[] biggerArray
031:           = new Food[foodListSoFar.length * ARRAY_RESIZE_FACTOR];
032:         for (int index = 0; index < foodListSoFar.length; index++)
033:           biggerArray[index] = foodListSoFar[index];
034:         foodListSoFar = biggerArray;
035:       } // if
036:       // Store the new item and count it.
037:       foodListSoFar[noOfFoodItemsSoFar] = latestFood;
038:       noOfFoodItemsSoFar++;
039:     } // while
```

The FoodList class

```
040:     noOfFoodItems = noOfFoodItemsSoFar;
041:     foodList = foodListSoFar;
042:   } // FoodList
```

Expression: boolean: logical operators: conditional

- The **logical operator** && is **conditional and**
  and || is **conditional or**
  - are lazy
    - if result determined by left **operand**, don't **evaluate** right one.
- I.e. if first **disjunct** of || is **`true`**
  or if first **conjunct** of && is **`false`**.
- Can safely write, e.g.:

```
data == null || data.length == 0
```

Design: Searching a list: linear search

- Simplest way to find item in **list**: **linear search**
  - start at front, look at each item in turn.
- E.g. **array search method**:

```
private int posOfInt(int[] anArray, int toFind)
{
  int searchPos = 0;
  while (searchPos < anArray.length && anArray[searchPos] != toFind)
    searchPos++;
  if (searchPos == anArray.length) return -1;
  else                             return searchPos;
} // posOfInt
```

Design: Searching a list: linear search

- But if swap **conjuncts**:

```
// Definitely silly code.
while (anArray[searchPos] != toFind && searchPos < anArray.length)
  searchPos++;
```

causes `ArrayIndexOutOfBoundsException`.

The FoodList class

```
045:   // Find the Food object corresponding to foodNameToFind
046:   // or return null if not found.
047:   public Food findFood(String foodNameToFind)
048:   {
049:     int foodIndex = 0;
050:     while (foodIndex < noOfFoodItems
051:            && ! foodList[foodIndex].getName().equals(foodNameToFind))
052:       foodIndex++;
053:     if (foodIndex == noOfFoodItems) return null;
054:     else                            return foodList[foodIndex];
055:   } // findFood
056:
057: } // class FoodList
```

The Diet class

```
001: import java.io.File;
002: import java.util.Scanner;
003:
004: /* This program reads food information from food-details.txt
005:    and diet information from diet-diary.txt
006:    and produces a table of how much nutritional component was eaten.
007:  */
008: public class Diet
009: {
010:   // The FoodList to be obtained from food-details.txt.
011:   private static FoodList foodList;
012:
013:
014:   // The main method.
015:   public static void main(String[] args) throws Exception
016:   {
017:     foodList = new FoodList(new Scanner(new File("food-details.txt")));
018:     readDietDiary(new Scanner(new File("diet-diary.txt")));
019:     printDietTable();
020:   } // main
```

The Diet class

```
023:   // An array of total nutritional component amounts:
024:   // Index is [component number]
025:   // and data is accumulated as number of milligrams of that component.
026:   private static int[] dietTable = new int[Food.NUTRITIONAL_COMPONENTS.length];
```

The Diet class

```
029:   // Read the diet information from the given Scanner
030:   // accumulating nutritional components in dietTable.
031:   private static void readDietDiary(Scanner scanner)
032:   {
033:     // First initialize the amounts to zero.
034:     for (int componentIndex = 0;
035:          componentIndex < Food.NUTRITIONAL_COMPONENTS.length; componentIndex++)
036:       dietTable[componentIndex] = 0;
037:     // Now read each line.
038:     while (scanner.hasNextLine())
039:     {
040:       String[] portionDetails = scanner.nextLine().split("\t+");
041:       // Food name is the first item.
042:       Food food = foodList.findFood(portionDetails[0]);
```

The Diet class

```
043:       if (food == null)
044:         System.out.println("Unrecognized food name: " + portionDetails[0]);
045:       else
046:       {
047:         // Food amount is the second item.
048:         int amount = Integer.parseInt(portionDetails[1]);
049:         // Obtain nutritional components from that amount.
050:         int[] foodComponents = food.componentMilliGramsForWeight(amount);
051:         // And accumulate them in dietTable.
052:         for (int componentIndex = 0;
053:              componentIndex < Food.NUTRITIONAL_COMPONENTS.length;
054:              componentIndex++)
055:           dietTable[componentIndex] += foodComponents[componentIndex];
056:       } // else
057:     } // while
058:   } // readDietDiary
```

The Diet class

```
061:   // Print the dietTable as grams (so divide by 1000).
062:   private static void printDietTable()
063:   {
064:     for (int componentIndex = 0;
065:          componentIndex < Food.NUTRITIONAL_COMPONENTS.length; componentIndex++)
066:       System.out.println(Food.NUTRITIONAL_COMPONENTS[componentIndex] + "\t"
067:                          + Math.round(dietTable[componentIndex] / 1000));
068:   } // printDietTable
069:
070: } // class Diet
```

Trying it

Console Input / Output

```
$ java Diet
Protein 280
Carb    621
Fat     273
Fibre   47
Sodium  17
$ _
```

Run

Trying it

Coffee time: Find out about `equalsIgnoreCase()` from the `String` **class**, and propose a change so that the user would not need to have consistent capitalization in the names of the food items.

Coffee time: Suppose you wish to delete an element from an arbitrarily ordered **partially filled array**. How can you, using only one assignment and one decrement?

Coffee time: How could we add saturated fat to the program? What extra issue would we have for Kcals?

Coursework: Viewing phone call details

(Summary only)

Write a program to allow the user to view certain phone call details.

Section 7

Example:

A weekly diet

Aim

AIM: To introduce **two-dimensional array**s.

A weekly diet

- Elaboration of previous:
  dieter also records day of week when food consumed.

Console Input / Output

```
$ cat diet-diary.txt
Mon     pizza           400
Mon     garlic bread    200
Mon     cheesecake      260
Tue     burger          200
Tue     fries           180
Tue     milkshake       400
Wed     fried chicken   360
Wed     wedges          270
Thu     pizza           650
Fri     burger          400
Fri     fries           360
Sat     fried chicken   360
Sat     wedges          540
Sun     garlic bread    800
Sun     cheesecake      260
$ _
```

Run

A weekly diet

- Program produces table of nutritional component amounts
  - row for each day of week.
- Much is same as previous
  - `food-details.txt`
  - **class**es `Food` and `FoodList`.
- Have `WeeklyDiet` instead of `Diet`.

The WeeklyDiet class

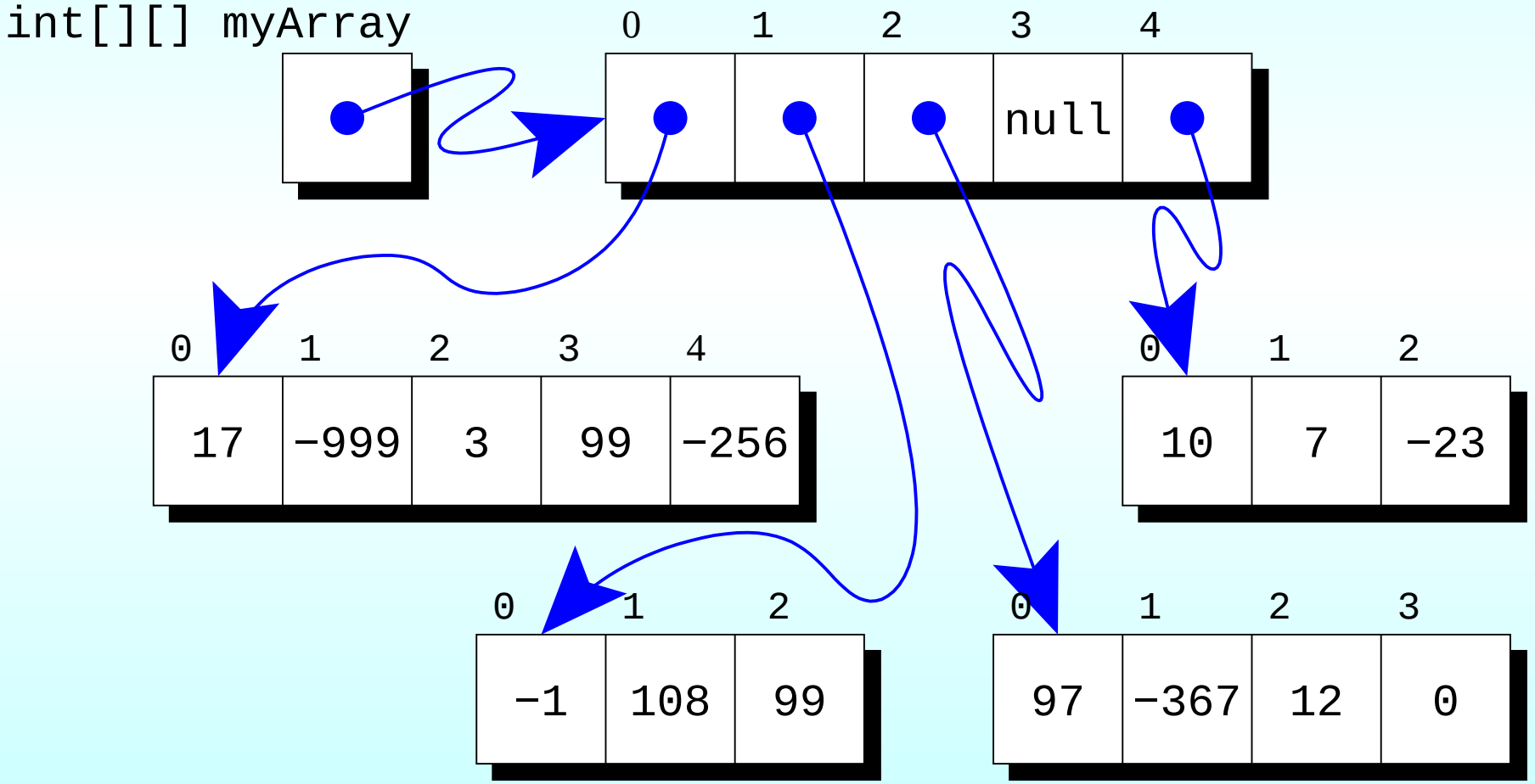
```
001: import java.io.File;
002: import java.util.Scanner;
003:
004: /* This program reads food information from food-details.txt
005:    and diet information from diet-diary.txt
006:    and produces a table of how much nutritional component was eaten
007:    on each day of the week.
008:  */
009: public class WeeklyDiet
010: {
011:    // The FoodList to be obtained from food-details.txt.
012:    private static FoodList foodList;
013:
014:
015:    // The main method.
016:    public static void main(String[] args) throws Exception
017:    {
018:      foodList = new FoodList(new Scanner(new File("food-details.txt")));
019:      readDietDiary(new Scanner(new File("diet-diary.txt")));
020:      printDietTable();
021:    } // main
```

The WeeklyDiet class

```
024:   // Days of the week -- this defines spelling for use in diet-diary.txt
025:   // and their order in dietTable.
026:   private static final String[] DAY_NAMES
027:     = { "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" };
028:
029:
030:   // Find the day index for the given day name, or -1 if not found.
031:   private static int findDayIndex(String dayName)
032:   {
033:     int dayIndex = 0;
034:     while (dayIndex < DAY_NAMES.length
035:            && ! DAY_NAMES[dayIndex].equals(dayName))
036:       dayIndex++;
037:     if (dayIndex == DAY_NAMES.length)
038:       return -1;
039:     else
040:       return dayIndex;
041:   } // findDayIndex
```

Array: array of arrays

- The **array element**s of an **array** may be any **type**: including arrays.
- E.g.



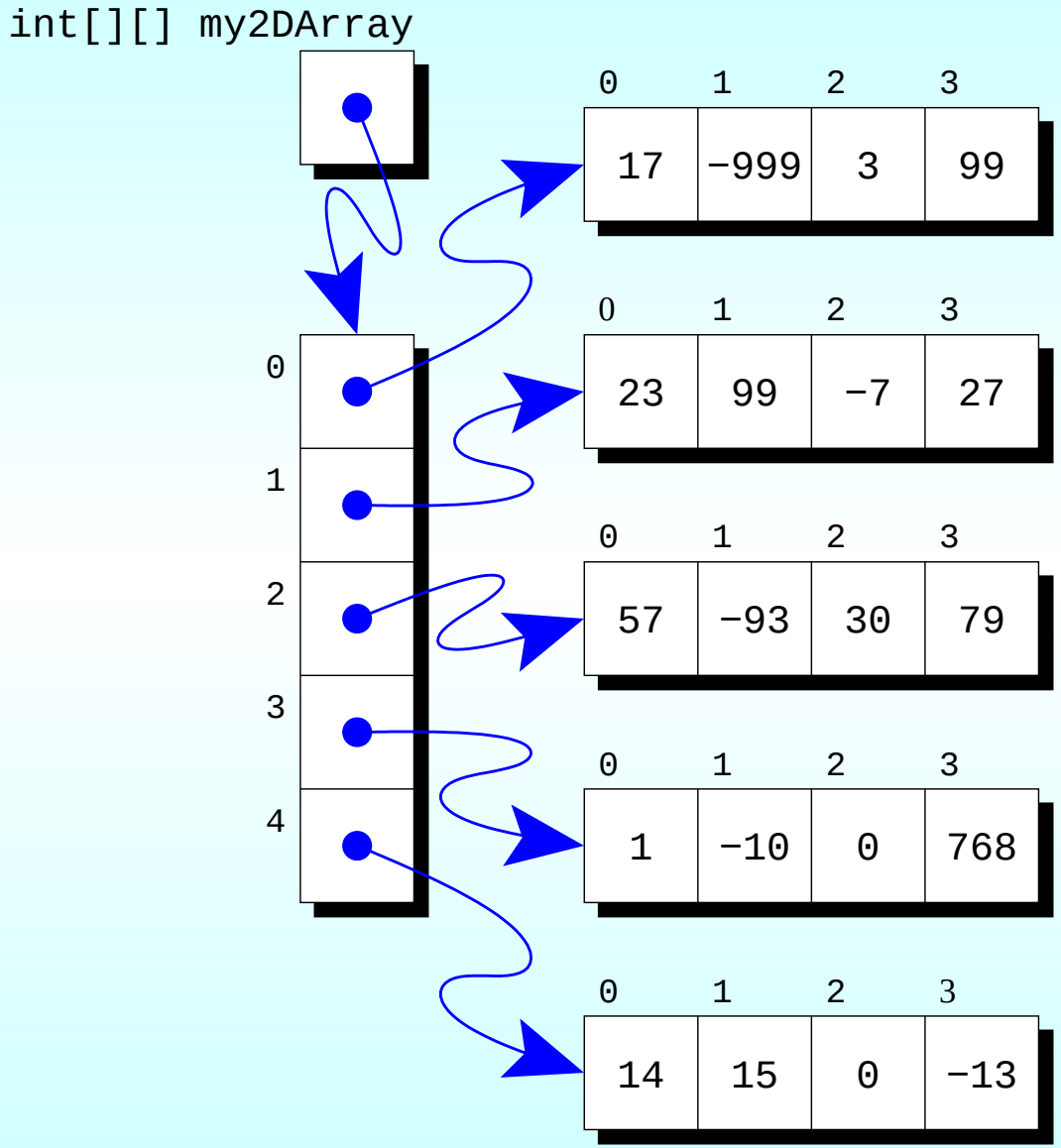
Array: array of arrays

- Created (not populated) via:

```
int[][] myArray = new int[5][];
myArray[0] = new int[5];
myArray[1] = new int[3];
myArray[2] = new int[4];
myArray[3] = null;
myArray[4] = new int[3];
```

Array: array of arrays: two-dimensional arrays

- Common situation with **array** of arrays
  - no **array element**s are **null reference**
  - all arrays **referenced** are same length.
- Known as **two-dimensional array**
  - essentially a grid.



Array: array of arrays: two-dimensional arrays

- Could be be created (not populated) via:

```
int[][] my2DArray = new int[5][];
my2DArray[0] = new int[4];
my2DArray[1] = new int[4];
my2DArray[2] = new int[4];
my2DArray[3] = new int[4];
my2DArray[4] = new int[4];
```

- But Java has shorthand:

```
int[][] my2DArray = new int[5][4];
```

The WeeklyDiet class

```
044:   // A two dimensional array of nutritional component amounts:
045:   // Index is [day number][component number]
046:   // and data is accumulated as number of milligrams of that component
047:   // eaten on that day.
048:   private static int[][] dietTable
049:     = new int[DAY_NAMES.length][Food.NUTRITIONAL_COMPONENTS.length];
```

Array: element access: in two-dimensional arrays

- Each grid element in **two-dimensional array** indexed by two indices
  - first **array index** accesses row **array**
  - second accesses **array element** within row.
- E.g.

```
int[][] my2DArray = new int[5][4];
```

  - `my2DArray[0]` is **reference** to first row
  - `my2DArray[0][0]` is first element in first row
  - `my2DArray[4][3]` is last element in last row.

The WeeklyDiet class

```
052:    // Read the diet information from the given Scanner
053:    // accumulating nutritional components in dietTable.
054:    private static void readDietDiary(Scanner scanner)
055:    {
056:      // First initialize the amounts to zero.
057:      for (int dayIndex = 0 ; dayIndex < DAY_NAMES.length; dayIndex++)
058:        for (int componentIndex = 0;
059:             componentIndex < Food.NUTRITIONAL_COMPONENTS.length;
060:             componentIndex++)
061:          dietTable[dayIndex][componentIndex] = 0;
```

The WeeklyDiet class

```
062:     // Now read each line.
063:     while (scanner.hasNextLine())
064:     {
065:       String[] portionDetails = scanner.nextLine().split("\t+");
066:       // Day name is the first item.
067:       int dayIndex = findDayIndex(portionDetails[0]);
068:       if (dayIndex == -1)
069:         System.out.println("Unrecognized day name: " + portionDetails[0]);
070:       // Food name is the second item.
071:       Food food = foodList.findFood(portionDetails[1]);
072:       if (food == null)
073:         System.out.println("Unrecognized food name: " + portionDetails[1]);
```

The WeeklyDiet class

```
074:      if (dayIndex != -1 && food != null)
075:      {
076:        // Food amount is the third item.
077:        int amount = Integer.parseInt(portionDetails[2]);
078:        // Obtain nutritional components from that amount.
079:        int[] foodComponents = food.componentMilliGramsForWeight(amount);
080:        // And accumulate them in dietTable.
081:        for (int componentIndex = 0;
082:             componentIndex < Food.NUTRITIONAL_COMPONENTS.length;
083:             componentIndex++)
084:          dietTable[dayIndex][componentIndex]
085:              += foodComponents[componentIndex];
086:      } // if
087:    } // while
088:  } // readDietDiary
```

The WeeklyDiet class

```
091:    // Print the dietTable as grams (so divide by 1000).
092:    private static void printDietTable()
093:    {
094:      // First print the column headings.
095:      for (String componentName : Food.NUTRITIONAL_COMPONENTS)
096:        System.out.print("\t" + componentName);
097:      System.out.println();
```

The WeeklyDiet class

```
099:     // Now print the rows, one for each day of the week.
100:     for (int dayIndex = 0; dayIndex < DAY_NAMES.length; dayIndex++)
101:     {
102:       System.out.print(DAY_NAMES[dayIndex]);
103:       for (int amountOfComponentEaten : dietTable[dayIndex])
104:         System.out.print(
105:           "\t" + Math.round(amountOfComponentEaten / 1000));
106:       System.out.println();
107:     } // for
108:   } // printDietTable
109:
110: } // class WeeklyDiet
```

Trying it

Console Input / Output

```
$ java WeeklyDiet
        Protein Carb    Fat     Fibre   Sodium
Mon     95      282     118     16      5
Tue     47      192     59      14      5
Wed     137     146     95      16      6
Thu     91      195     77      16      5
Fri     72      224     94      28      9
Sat     194     252     121     27      10
Sun     117     414     128     21      5
$ _
```

Run

Coffee time: What would happen if we declared DAY_NAMES as follows?

```
private static final String[] DAY_NAMES
    = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Coursework: Maze solver

(Summary only)

Write a program that finds the shortest path through a maze.

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.