Note: `System.out.println()` always ends the line with the platform dependent **line separator**, which on Linux is a new line character but on Microsoft Windows is a **carriage return character** followed by a new line character. In practice you may not notice the difference, but the above code is not strictly the same as using three separate `System.out.println()` calls and is not 100% portable.

```
    System.out.println("Your salary:\t" + salary
                       + "\nYour mortgage:\t" + mortgage
                       + "\nYour bills:\t" + bills
                       + "\nDisposable:\t" + disposableIncome);
  }
}
```

3.7.1 The full DisposableIncome code

```
public class DisposableIncome
{
  public static void main(String[] args)
  {
    int salary   = Integer.parseInt(args[0]);
    int mortgage = Integer.parseInt(args[1]);
    int bills    = Integer.parseInt(args[2]);
    int disposableIncome = salary - (mortgage + bills);
    System.out.println("Your salary:\t" + salary
                       + "\nYour mortgage:\t" + mortgage
                       + "\nYour bills:\t" + bills
                       + "\nDisposable:\t" + disposableIncome);
  }
}
```

3.7.2 Trying it

After we have **compile**d the program, we can **run** it.

Console Input / Output

```
$ java DisposableIncome 38356 24317 4665
Your salary:    38356
Your mortgage:  24317
Your bills:     4665
Disposable:     9374
$ _
```

You'll survive. ;-) But the guy below needs a better job – perhaps Java programming?

Console Input / Output

```
$ java DisposableIncome 19178 12875 3665
Your salary:    19178
Your mortgage:  12875
Your bills:     3665
Disposable:     2638
$ _
```

In later examples we shall see two ways of addressing the **line separator** portability issue in places where we don't want to, or cannot, use `System.out.println()` to get it right.

3.7.3 Coursework: ThreeWeights

In the days before accurate mechanical spring weighing scales (let alone digital ones), gold merchants were quite clever in their use of a small number of brass or lead weights, and a balance scale. (Indeed, many still use these in preference to inferior modern technology!) They would place the gold to be weighed in the left pan of the balance scale, and then place known weights in the right pan, and maybe also in the left pan, until the scales balanced. For example, suppose an unknown

6.5 Example: Printing a triangle

AIM: To reinforce the idea of nesting a **for loop** within a **for loop**.

This next program is very similar to the previous, except this time to make it trickier, we want an isosceles right angled triangle of a height given as the **command line argument**. The first line of text has one cell, the second has two, and so on, until the last line has as many cells as the height. For example, a triangle of height four would be printed as follows.

Console Input / Output

```
$ java PrintTriangle 4
[_]
[_][_]
[_][_][_]
[_][_][_][_]
$ _
```

Here is the code, which you should compare with that for printing a rectangle.

```
// Program to print out an isosceles right angled triangle.
// The height is given as an argument.
// We assume the argument represents a positive integer.
public class PrintTriangle
{
  public static void main(String[] args)
  {
    // The height of the triangle.
    int height = Integer.parseInt(args[0]);

    // Print out height number of rows.
    for (int row = 1; row <= height; row++)
    {
      // Print out row number of cells, on the same line.
      for (int column = 1; column <= row; column++)
        System.out.print("[_]");
      // End the line.
      System.out.println();
    } // for
  } // main

} // class PrintTriangle
```

6.5.1 Trying it

Console Input / Output

```
$ java PrintTriangle 10
[_]
[_][_]
[_][_][_]
[_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_][_]
[_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_]
$ _
```

Console Input / Output

```
$ java PrintTriangle 15
[_]
[_][_]
[_][_][_]
[_][_][_][_]
[_][_][_][_][_]
[_][_][_][_][_][_]
[_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_][_][_][_]
[_][_][_][_][_][_][_][_][_][_][_][_][_][_][_]
$ _
```

Coffee time: 6.5.1
What would happen if we changed the outer **for loop** to the following?

```
  for (int row = 0; row < height; row++)
```

Coffee time: 6.5.2
What would happen if we changed the inner **for loop** to the following?

```
  for (int column = 1; column <= height - row + 1; column++)
```

program causes an **exception** during its execution (p.24), and **logical error** when everything seems to work fine, but the program produces the wrong result (p.29). Syntactic and semantic errors are collectively known as **compile time error**s (p.22).

9.2.6 Standard classes

Java comes with lots of **class**es ready to use in its **application program interface** (**API**). We have met some of the features of a few so far.

The class `System` contains **method**s for printing results on **standard output** (p.7).

Name	Return	Parameter	Description	Page
`System.out.println`		`String`	Print the given string and a **new line** on the output.	(p.18)
`System.out.println`		(none)	Produce a new line on the output.	(p.98)
`System.out.println`		**`int`**	Print the decimal representation of the given **`int`** and a new line on the output.	(p.38)
`System.out.print`		`String`	Print the given string with no new line on the output.	(p.98)
`System.out.printf`		`String`, value	Prints a formatted representation of the given value, according to the given **format specifier** string (e.g. `"%010.2f%n"`).	(p.126) (p.140)

In fact there is a version of `System.out.print()` and `System.out.println()` for all the **type**s we have met so far. `System.out.println()` produces a new line using the platform dependent **line separator**, which is a **new line character** on Linux and a **carriage return character** followed by a new line character on Microsoft Windows.

The classes `Integer` and `Double` contain methods to convert a given `String` into the number it represents.

Name	Return	Parameter	Description	Page
`Integer.parseInt`	**`int`**	`String`	Convert the given string into the **`int`** it represents, or cause an **exception** if it cannot.	(p.41)
`Double.parseDouble`	**`double`**	`String`	Convert the given string into the **`double`** it represents, or cause an exception if it cannot.	(p.54)

The class `Math` contains methods for various mathematical **function**s including the following.

Name	Return	Parameter	Description	Page
`Math.pow`	**`double`**	**`double`**, **`double`**	Returns the first parameter raised to the second.	(p.73)
`Math.abs`	**`double`**	**`double`**	Returns the **absolute value** of the parameter.	(p.87)
`Math.sin`	**`double`**	**`double`**	Returns the sin of the given value, which is expressed in radians.	
`Math.toRadians`	**`double`**	**`double`**	Returns the radians equivalent of the given degrees value.	

There is also the constant `Math.PI` (p.87).

9.3 Program design concepts

AIM: To look more formally at the process of **design**ing an **algorithm** and writing a program. In particular, we look closely at **designing variables**.

We have seen lots of example programs in the previous chapters, and by a process of osmosis, especially if you have done the coursework too, you will have started to pick up the skill of programming. Now is a good time to try and formalize this

All you have to do is write the other classes.

The following are example **run**s of the program to help clarify the requirements.

Console Input / Output

```
$ java ShapeShift
Choose circle (1), triangle (2), rectangle (3): 1
Enter the centre as X Y: 0 0
Enter the radius: 1
Enter the offset as X Y: 2 2

Circle((0.0,0.0),1.0)
has area 3.141592653589793, perimeter 6.283185307179586
and when shifted by X offset 2.0 and Y offset 2.0, gives
Circle((2.0,2.0),1.0)
$ _
```

Console Input / Output

```
$ java ShapeShift
Choose circle (1), triangle (2), rectangle (3): 2
Enter point A as X Y: 0 0
Enter point B as X Y: 10 0
Enter point C as X Y: 0 20
Enter the offset as X Y: 5 10

Triangle((0.0,0.0),(10.0,0.0),(0.0,20.0))
has area 100.0, perimeter 52.3606797749979
and when shifted by X offset 5.0 and Y offset 10.0, gives
Triangle((5.0,10.0),(15.0,10.0),(5.0,30.0))
$ _
```

Console Input / Output

```
$ java ShapeShift
Choose circle (1), triangle (2), rectangle (3): 3
Enter one corner as X Y: 0 0
Enter opposite corner as X Y: 10 20
Enter the offset as X Y: 0 0

Rectangle((0.0,0.0),(10.0,0.0),(10.0,20.0),(0.0,20.0))
has area 200.0, perimeter 60.0
and when shifted by X offset 0.0 and Y offset 0.0, gives
Rectangle((0.0,0.0),(10.0,0.0),(10.0,20.0),(0.0,20.0))
$ _
```

Start by designing your **test data** in your logbook.

Your program will consist of five **class**es, `Point`, `Circle`, `Triangle`, `Rectangle` and the already given `ShapeShift`. Next identify and record the **public instance method**s and **class method**s for each of the four classes you will write. Endeavour to associate behaviour (i.e. **method**s) with the most appropriate classes. Here are some hints.

- Which classes should have a `toString()` instance method?
- Should shape classes have methods to find the area and perimeter of a shape?
- Should they additionally have a method to create a shifted shape from an existing one?
- Shifting shapes requires creating **new** points which are shifts of old ones. Where is that shifting best done?
- Perimeters of certain shapes are based on distances between points – does that suggest an instance method in the `Point` class?
- Are the points **mutable object**s or **immutable object**s? What about the shapes?
- All **instance variable**s should be **private**, so you may need some instance methods in some classes, to give read access to the instance variables. For example, `Point` might have `getX()` and `getY()`.

Next you should write **stub**s for the three shape classes, so that you can **compile** and try out the main class.

To use a layout manager, we make an **instance** of whichever type we desire to have, and then tell the `Container` that we wish it to use that layout manager, via its `setLayout()` **instance method**.

Concept **GUI API: `Container: setLayout()`.** The **class** `java.awt.Container` has an **instance method** called `setLayout` which takes an **instance** of one of the **layout manager** classes, and uses that to lay out its **graphical user interface** (**GUI**) components each time a lay out is needed, for example, when the window it is part of is **pack**ed.

```
    // We want the planet names to appear in one line.
    contents.setLayout(new FlowLayout());
```

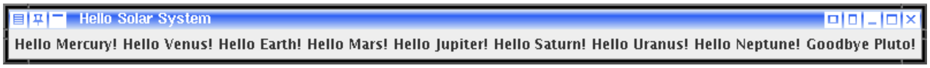
Now we add nine `JLabel` objects, and we know that these will appear in the final window, in a single row, in the order we add them.

```
    contents.add(new JLabel("Hello Mercury!"));
    contents.add(new JLabel("Hello Venus!"));
    contents.add(new JLabel("Hello Earth!"));
    contents.add(new JLabel("Hello Mars!"));
    contents.add(new JLabel("Hello Jupiter!"));
    contents.add(new JLabel("Hello Saturn!"));
    contents.add(new JLabel("Hello Uranus!"));
    contents.add(new JLabel("Hello Neptune!"));
    contents.add(new JLabel("Goodbye Pluto!"));

    setDefaultCloseOperation(EXIT_ON_CLOSE);
    pack();
  } // HelloSolarSystem
```

Finally we have the **main method**, which simply creates an instance and makes it visible.

```
  // Create a HelloSolarSystem and make it appear on screen.
  public static void main(String[] args)
  {
    HelloSolarSystem theHelloSolarSystem = new HelloSolarSystem();
    theHelloSolarSystem.setVisible(true);
  } // main

} // class HelloSolarSystem
```

13.3.1 Trying it



Hello Mercury! Hello Venus! Hello Earth! Hello Mars! Hello Jupiter! Hello Saturn! Hello Uranus! Hello Neptune! Goodbye Pluto!

13.3.2 Coursework: `HelloFamily` GUI

The coursework in Section 2.5.2 on page 24, asked you to produce a program called `HelloFamily` which greeted a number of your relatives. In this task you will write a version of that program which produces a window and greets the same relatives using labels. Each greeting should use a separate label. Use a `FlowLayout` **object** to manage the layout of the components in the window.

```
// Their typical salary.
private final int salary;
```

The **constructor method** sets the instance variables.

```
// The constructor method.
public Job(String requiredEmployer, int requiredSalary)
{
  employer = requiredEmployer;
  salary = requiredSalary;
} // Job
```

We have an **accessor method** for each instance variable.

```
// Get the employer.
public String getEmployer()
{
  return employer;
} // getEmployer


// Get the salary.
public int getSalary()
{
  return salary;
} // getSalary
```

We have a `compareTo()` **instance method** for comparing this job against a given other one with the usual **`int`** result which is negative, zero or positive. This provides an ordering based on ascending salary. However, if the salaries are the same, then we compare the employers instead, and you will recall from Section 12.4 on page 234 that `String` has a `compareTo()` instance method.

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

Finally, `toString()` provides a representation of the job, showing the firm's name and their salary.

Concept **Standard API: `System`: `out.printf()`: left justification.**

If we wish an item printed by `System.out.printf()` to be left justified, rather than right justified, then we can place a hyphen in front of the width in the **format specifier**. For example,

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

produces the following.

```
123456789012345X
Hello World    X
```

Concept **GUI API: `Color`.** The **class** `java.awt.Color` implements colours to be used in **graphical user interface**s. Each `Color` **object** comprises four values in the range 0 to 255, one for each of the primary colours red, green and blue, and a fourth component (alpha) for opacity.

For convenience, the class includes a number of **class constant**s containing **reference**s to `Color` objects which represent some common colours.

```
public static final Color black     = new Color(0,     0,   0, 255);
public static final Color white     = new Color(255, 255, 255, 255);
public static final Color red       = new Color(255,   0,   0, 255);
public static final Color green     = new Color(0,   255,   0, 255);
public static final Color blue      = new Color(0,   0,   255, 255);

public static final Color lightGray = new Color(192, 192, 192, 255);
public static final Color gray      = new Color(128, 128, 128, 255);
public static final Color darkGray  = new Color(64,   64,  64, 255);

public static final Color pink      = new Color(255, 175, 175, 255);
public static final Color orange    = new Color(255, 200,   0, 255);
public static final Color yellow    = new Color(255, 255,   0, 255);
public static final Color magenta   = new Color(255,   0, 255, 255);
public static final Color cyan      = new Color(0,   255, 255, 255);
```

Coffee time: 16.9.1
From these examples, can you work out the definition of the **constructor method** for `Color`?

Among many other features, there is an **instance method** `getRGB()` which **return**s a unique `int` for each **equivalent** colour, based on the four component values.

The `Ball` class is fairly straightforward.

```
import java.awt.Color;

// Representation of a lottery ball, comprising colour and value.
public class Ball
{
  // The numeric value of the ball.
  private final int value;

  // The colour of the ball.
  private final Color colour;


  // A ball is constructed by giving a number and a colour.
  public Ball(int requiredValue, Color requiredColour)
  {
    value = requiredValue;
    colour = requiredColour;
  } // Ball


  // Returns the numeric value of the ball.
  public int getValue()
  {
    return value;
  } // getValue
```

```
  // Returns the colour of the ball.
  public Color getColour()
  {
    return colour;
  } // getColour
```

Section	Aims	Associated Coursework
18.6 Numbering lines from and to anywhere (p.467)	To illustrate that reading from **text file**s and from **standard input** is essentially the same thing, as is writing to **text file**s and to **standard output**. We also look at testing for the existence of a **file** using the `File` **class**, and revisit `PrintWriter` and `PrintStream`.	Write a program to delete a field in tab separated text either from **standard input** or a **file**, with the results going to either **standard output** or another file. (p.471)
18.7 Text photographs (p.471)	To see an example of reading **binary file**s, where we did not choose the **file format**. This includes the process of turning **`byte`**s into **`int`**s, using a **shift operator** and an **integer bitwise operator**.	Write a program to encode a **binary file** as an **ASCII text file**, so that it can be sent in an email. (p.477)
18.8 Contour points (p.479)	To show an example of writing and reading **binary file**s where we choose the **data** format, using `DataOutputStream` and `DataInputStream` **class**es.	Add features to some existing model **class**es so they can be written and read back from **binary file**s. (p.483)

18.2 Example: Counting bytes from standard input

AIM: To introduce the principle of reading **byte**s from **standard input** using `InputStream`, meet the **try finally statement** and see that an **assignment statement** is actually an **expression** – and can be used as such *when appropriate*. We also meet `IOException` and briefly talk about initial values of **variable**s.

We begin with a program that reads the **standard input** until it is finished, and then reports how many **byte**s it contained, and how many of each byte value, for those that appeared at least once. This feature could be useful in an **operating environment** in which the user can redirect standard input, so that it comes from a **file**, or from the output of a **run**ning program, and so see the profile of the bytes in that file or output.

We start by observing that file operations are prone to all sorts of **exception**al circumstances.

Concept **File IO API: `IOException`.** When processing **file**s, there is much potential for things to go wrong. For example, attempting to read a file that does not exist, or the end user running out of file space while writing a file, or the **operating system** experiencing a disk or network filestore problem, and so on. As a result, most of the operations we can perform on files in Java are capable of **throw**ing an **exception**, of the **type** `java.io.IOException`. As you might expect, there are many **subclass**es of `IOException`, including `java.io.FileNotFoundException`.

`IOException` is itself a direct **subclass** of `java.lang.Exception`, rather than `java.lang.RuntimeException` and thus **instance**s of it are **checked exception**s, that is, we must write **catch clause**s or **throws clause**s for them. This is because the errors which cause them are not generally avoidable by writing code.

Our program will read the **data** from the standard input, byte by byte, and process them. This will require the use of an `InputStream`, and the typical way we use it appropriately exploits the fact that an **assignment statement** is an **expression**.

Concept **Statement: assignment statement: is an expression.** In Java, the **assignment statement** is actually an **expression**. The = symbol is an **operator**, which takes a **variable** as its left **operand**, and an expression as its right operand. It evaluates the expression, assigns it to the variable, *and then* yields the value of the expression as its result.

19.5.4 The `TestConversationOops` class

Let’s see what happens if we put the wrong kind of `Person` in a `Conversation`.

```java
001: // Create conversations of people and make them speak.
002: public class TestConversationOops
003: {
004:   public static void main(String[] args)
005:   {
006:     // A conversation of AudienceMembers.
007:     Conversation<AudienceMember> audienceChat
008:       = new Conversation<AudienceMember>();
009:     audienceChat.addPerson(new AudienceMember("AM 1"));
010:     audienceChat.addPerson(new TVHost("TVH 1"));
011:     System.out.printf("%s%n%n", audienceChat);
012:     for (int count = 1; count <= audienceChat.getSize(); count++)
013:     {
014:       audienceChat.speak();
015:       System.out.printf("%s%n%n", audienceChat);
016:     } // for
017:   } // main
018:
019: } // class TestConversationOops
```

Console Input / Output

```
$ javac TestConversationOops.java
TestConversationOops.java:10: addPerson(AudienceMember) in Conversation<Audience
Member> cannot be applied to (TVHost)
    audienceChat.addPerson(new TVHost("TVH 1"));
                ^
1 error
$ _
```

Coffee time: 19.5.3 Recall the full `Person` hierarchy from Section 16.13 on page 416. How could we have a `Conversation` in which all the persons must be `MoodyPersons`, but can be any kind of moody person?

Coffee time: 19.5.4 Recall that within the `Conversation` **class**, we had an **array** of **type** `Person[]`, in which only `PersonType` **object**s were stored. It would have been nicer to declare the array as `PersonType[]`. So, why didn’t we? Try it to find out!

19.5.5 Coursework: A moody group

This coursework is set in the context of the Notional Lottery game from Section 16.2 on page 372.

Write a **generic class** called `MoodyGroup` that contains a collection of some **subclass** of `MoodyPerson` **object**s, rather like the `Conversation` **class** does with `Person`. However, instead of a `speak()` **instance method**, `MoodyGroup` should have `setHappy()`. This will take a **`boolean`** and pass it to the instance method of the same name belonging to each of the `MoodyPersons` in the group. You will recall that only `MoodyPersons` have the `setHappy()` instance method, whereas the more general `Person` does not.

Test your class with a program called `TestMoodyGroup`. This will do the following.

- Create an **instance** of `MoodyGroup<Teenager>` and populate it with a small number of `Teenagers`.
- Invoke `setHappy()` with **`false`** and print out the group.
- Invoke `setHappy()` with **`true`** and print out the group again.
- Create a second moody group which can contain any kind of `MoodyPerson`, and populate it with a `Worker` and one of the *same* `Teenagers` which was put into the first group.
- Invoke `setHappy()` on the second group with **`true`** and print out the group.
- Invoke `setHappy()` on the second group with **`false`** and print out the group.
- Print out the first group one more time to show that the teenager which is in both groups stands out from the others.

How many **prime numbers** are there up to 1 thousand?

Console Input / Output

```
$ java Primes 1000
 (Output shown using multiple columns to save space.)
1 : 2          25 : 97        49 : 227       73 : 367       97 : 509       121 : 661      145 : 829
2 : 3          26 : 101       50 : 229       74 : 373       98 : 521       122 : 673      146 : 839
3 : 5          27 : 103       51 : 233       75 : 379       99 : 523       123 : 677      147 : 853
4 : 7          28 : 107       52 : 239       76 : 383       100 : 541      124 : 683      148 : 857
5 : 11         29 : 109       53 : 241       77 : 389       101 : 547      125 : 691      149 : 859
6 : 13         30 : 113       54 : 251       78 : 397       102 : 557      126 : 701      150 : 863
7 : 17         31 : 127       55 : 257       79 : 401       103 : 563      127 : 709      151 : 877
8 : 19         32 : 131       56 : 263       80 : 409       104 : 569      128 : 719      152 : 881
9 : 23         33 : 137       57 : 269       81 : 419       105 : 571      129 : 727      153 : 883
10 : 29        34 : 139       58 : 271       82 : 421       106 : 577      130 : 733      154 : 887
11 : 31        35 : 149       59 : 277       83 : 431       107 : 587      131 : 739      155 : 907
12 : 37        36 : 151       60 : 281       84 : 433       108 : 593      132 : 743      156 : 911
13 : 41        37 : 157       61 : 283       85 : 439       109 : 599      133 : 751      157 : 919
14 : 43        38 : 163       62 : 293       86 : 443       110 : 601      134 : 757      158 : 929
15 : 47        39 : 167       63 : 307       87 : 449       111 : 607      135 : 761      159 : 937
16 : 53        40 : 173       64 : 311       88 : 457       112 : 613      136 : 769      160 : 941
17 : 59        41 : 179       65 : 313       89 : 461       113 : 617      137 : 773      161 : 947
18 : 61        42 : 181       66 : 317       90 : 463       114 : 619      138 : 787      162 : 953
19 : 67        43 : 191       67 : 331       91 : 467       115 : 631      139 : 797      163 : 967
20 : 71        44 : 193       68 : 337       92 : 479       116 : 641      140 : 809      164 : 971
21 : 73        45 : 197       69 : 347       93 : 487       117 : 643      141 : 811      165 : 977
22 : 79        46 : 199       70 : 349       94 : 491       118 : 647      142 : 821      166 : 983
23 : 83        47 : 211       71 : 353       95 : 499       119 : 653      143 : 823      167 : 991
24 : 89        48 : 223       72 : 359       96 : 503       120 : 659      144 : 827      168 : 997
$ _
```

How fast is this **algorithm**? Let's find the primes up to 1 million. We can time it using the Unix `time` command to **run** the program and then tell us how long it took to run.[a] (In case you are interested, this was run on a 2Gig Hertz Athlon XP 2600+ processor.) We redirect the output to a **file**, using >, so that displaying the numbers does not seriously slow down the program.

Console Input / Output

```
$ time java Primes 1000000 > primes.txt

real    0m5.608s
user    0m4.690s
sys     0m0.860s
$ cat primes.txt
1 : 2
2 : 3
 (... lines removed to save space.)
78496 : 999961
78497 : 999979
78498 : 999983
$ _
```

[a] Unfortunately, there is no simple way of doing this using standard commands in a Microsoft Command Prompt.

Ah, but it does require a lot of space to store all those non-prime numbers – let's try up to 10 million.

Console Input / Output

```
$ time java Primes 10000000 > primes.txt
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.lang.Integer.valueOf(Integer.java:585)
        at Primes.main(Primes.java:30)

real    0m8.495s
user    0m7.910s
sys     0m0.560s
$ cat primes.txt
1 : 2
$ _
```

```
      // Put the asterisk back to restore the value,
      // which is needed for later calls past this point.
      inputChars[scanPosition] = '*';
    } // else
  } // outputVowelMovements

} // class VowelMovements
```

Coffee time: 22.9.1 What would happen if we forgot to replace the asterisk after the loop that goes through the five vowels? If we had two asterisks in the input, how many output 'words' would we get?

Our `outputVowelMovements()` recursive method does not use **tail recursion**, so it is not obvious how to implement it **iterative**ly.

Coffee time: 22.9.2 Have a go at finding an iterative solution! You can do it, if you approach the problem in a wholly different way – similar to what you did for the dice combinations. Is the iterative solution (significantly) more efficient? Is it shorter or longer code? Is it easier or harder to see that it is correct?

22.9.1 Trying it

Console Input / Output

```
$ java VowelMovements 'El*zabeth'
 (Output shown using multiple columns to save space.)
Elazabeth    | Elezabeth    | Elizabeth    | Elozabeth    | Eluzabeth
$ _
```

Console Input / Output

```
$ java VowelMovements Elizabeth
Elizabeth
$ _
```

Console Input / Output

```
$ java VowelMovements 'El*z*beth'
 (Output shown using multiple columns to save space.)
Elazabeth    | Elezabeth    | Elizabeth    | Elozabeth    | Eluzabeth
Elazebeth    | Elezebeth    | Elizebeth    | Elozebeth    | Eluzebeth
Elazibeth    | Elezibeth    | Elizibeth    | Elozibeth    | Eluzibeth
Elazobeth    | Elezobeth    | Elizobeth    | Elozobeth    | Eluzobeth
Elazubeth    | Elezubeth    | Elizubeth    | Elozubeth    | Eluzubeth
$
```

Console Input / Output

```
$ java VowelMovements '*****' | java LineNumber
 (Output shown using multiple columns to save space.)
00001 aaaaa           | 00018 aaaoi           | 03109 uuueo
00002 aaaae           | 00019 aaaoo           | 03110 uuueu
00003 aaaai           | 00020 aaaou           | 03111 uuuia
00004 aaaao           | 00021 aaaua           | 03112 uuuie
00005 aaaau           | 00022 aaaue           | 03113 uuuii
00006 aaaea           | 00023 aaaui           | 03114 uuuio
00007 aaaee           | 00024 aaauo           | 03115 uuuiu
00008 aaaei           | 00025 aaauu           | 03116 uuuoa
00009 aaaeo           |  (...lines removed to save space.) | 03117 uuuoe
00010 aaaeu           | 03101 uuuaa           | 03118 uuuoi
00011 aaaia           | 03102 uuuae           | 03119 uuuoo
00012 aaaie           | 03103 uuuai           | 03120 uuuou
00013 aaaii           | 03104 uuuao           | 03121 uuuua
00014 aaaio           | 03105 uuuau           | 03122 uuuue
00015 aaaiu           | 03106 uuuea           | 03123 uuuui
00016 aaaoa           | 03107 uuuee           | 03124 uuuuo
00017 aaaoe           | 03108 uuuei           | 03125 uuuuu
$ _
```