Computer Project

(2017-2019)

Satvik Saha

Class: XII B

Roll number: 34

"Writing code a computer can understand is science. Writing code other programmers can understand is an art."

— **Jason Gorman**

"I am rarely happier than when spending an entire day programming my computer to perform automatically a task that would otherwise take me a good ten seconds to do by hand."

— **Douglas Adams**

Problem 1 An n digit integer $(a_1 a_2 \dots a_n)$, where each digit $a_i \in \{0, 1, \dots, 9\}$, is said to have *unique digits* if no digits are repeated, i.e., there is no i, j such that $a_i = a_j$ $(i \neq j)$.

Verify whether an inputted number has *unique digits*.

Solution The problem involves simply counting the number of occurrences of each digit in the given number and checking whether any of them exceed 1.

`main (number:Integer)`

1. Initialize an integer array `digits` of length `10`, indexed with integers from `[0]` to `[9]` with all elements set to `0`.
2. If `number` exceeds `0`, proceed. Otherwise, jump to (3).
   (a) Store the last digit[1] of `number` in a temporary variable `d`.
   (b) Increment the integer at the `d` index of `digits`.
   (c) If `digits[d]` exceeds `1`, the number does not have *unique digits*. Display a suitable message, and **exit**.
   (d) Discard the last digit of `number` by performing an integer division by `10` and storing the result back in `number`.
   (e) Jump to (2).
3. The number has *unique digits*. Display a suitable message.
4. **Exit**

[1]The last digit of an integer n is simply n mod 10

Source Code

```java
public class Unique {
	public static void main (String[] args) {
		try {
			/* Parse the first command line argument as the number
			   to check for unique digits */
			long number = Long.parseLong(args[0]);
			if (isUnique(number)) {
				System.out.println("Unique Number!");
			} else {
				System.out.println("Not a Unique Number!");
			}
		} catch (NumberFormatException | IndexOutOfBoundsException e) {
			/* Handle missing or incorrectly formatted arguments */
			System.out.println("Enter 1 argument (number[integer])!");
		}
	}

	public static boolean isUnique (long number) {
		/* Keep track of the number of occurrences of each digit */
		int[] count = new int[10];
		for (long n = Math.abs(number); n > 0; n /= 10) {
			/* Extract the last digit of the number */
			int digit = (int) n % 10;
			count[digit]++;
			if (count[digit] > 1){
				return false;
			}
		}
		return true;
	}
}
```

Variable Description

`Unique::main(String[])`		
`long`	`number`	The inputted number

`Unique::isUnique(long)`		
`long`	`number`	The number to check for uniqueness
`int[]`	`count`	The number of occurrences of each digit
`long`	`n`	Counter, temporarily stores the value of `number`
`int`	`digit`	The last digit in `n`

"Elegance is not a dispensable luxury but a factor that decides between success and failure."

— **Edsger W. Dijkstra**

Problem 2 A *partition* of a positive integer n is defined as a collection of other positive integers such that their sum is equal to n. Thus, if $(a_1, a_2, \ldots, a_k)$ is a partition of n,

$$n \;=\; a_1 + a_2 + \cdots + a_k \qquad (a_i \in \mathbb{Z}^+)$$

Display every *unique partition* of an inputted number.

Solution This problem can be solved elegantly using *recursion*[2]. Note that when partitioning a number n, we can calculate the partitions of $(n-1)$ and append 1 to each solution. Similarly, we can append 2 to partitions of $(n-2)$, 3 to partitions of $(n-3)$, and so on. By continuing in this fashion, all cases will be reduced to the single *base case*[3] of finding the partitions of 0, of which there are trivially none.[citation needed]

There is a slight flaw in this algorithm — partitions are often repeated. This can be overcome by imposing the restriction that each new term has to be of a lesser magnitude than the previous. In this way, repeated partitions will be automatically discarded.

`main (target:Integer)`

1. Call `partition(target, target, "")`.
2. **Exit**

`partition (target:Integer, previousTerm:Integer, suffix:String)`

1. If `target` is `0`, display `suffix` and **return**.
2. Initialize a counter `i` to `1`.
3. If `i` is less than or equal to both the `target` and `previousTerm`, proceed. Otherwise, jump to (4).
   (a) Call `partition(target - i, i, suffix + " " + i)`.
   (b) Increment `i` by `1`.
   (c) Jump to (3).
4. **Return**

[2]Recursion occurs when a thing is defined in terms of itself or of its type.

[3]A base case is a case for which the answer is known and can be expressed without recursion.

Source Code

```java
public class Partition {
	public static void main (String[] args) {
		try {
			/* Parse the first command line argument as the target sum */
			int target = Integer.parseInt(args[0]);
			if (target < 1) {
				throw new NumberFormatException();
			}
			partition(target);
		} catch (NumberFormatException | IndexOutOfBoundsException e) {
			/* Handle missing or incorrectly formatted arguments */
			System.out.println("Enter 1 argument (number[natural
			    number])!");
		}
	}

	/* Wrapper method for displaying partitions of a number */
	public static void partition (int target) {
		partition(target, target, "");
	}

	/* Display the partitions of the target */
	public static void partition (int target, int previousTerm, String suffix) {
		/* Base case : '0' has no partitions */
		if (target == 0)
			System.out.println(suffix);
		/* Recursively solve for partitions by diminishing the target,
		   adding that difference to the solution, and partitioning the
		   remaining sum */
		for (int i = 1; i <= target && i <= previousTerm; i++)
			partition(target - i, i, suffix + " " + i);
	}
}
```

Variable Description

`Partition::main(String[])`		
`int`	`target`	The inputted number

`Partition::partition(int)`		
`int`	`target`	The number to be partitioned

`Partition::partition(int, int, String)`		
`int`	`target`	The number to be partitioned
`int`	`previousTerm`	The previous term in the partition sequence
`String`	`suffix`	Terms in the sequence calculated so far
`int`	`i`	Counter variable, stores the next term in the sequence

"Simplicity is the ultimate sophistication."

— **Leonardo da Vinci**

Problem 3 A *Caesar cipher* is a type of monoalphabetic substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. The positions are circular, i.e., after reaching Z, the position wraps around to A. For example, following is some encrypted text, using a right shift of 5.

```
Plain:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:  FGHIJKLMNOPQRSTUVWXYZABCDE
```

Thus, after mapping the alphabet according to the scheme $A \mapsto 0, B \mapsto 1, \ldots, Z \mapsto 25$, we can define an encryption function E_n, in which a letter x is shifted rightwards by n as follows.

$$E_n(x) \;=\; (x + n) \mod 26$$

The corresponding decryption function D_n is simply

$$D_n(x) \;=\; (x - n) \mod 26$$

Implement a simple version of a *Caesar cipher*, encrypting capitalized plaintext by shifting it by a given value. Interpret positive shifts as rightwards, negative as leftwards.

Solution This problem can be solved simply by exploiting the fact that Unicode characters are already arranged in order, with successive alphabets encoded by consecutive numbers. In addition, the encryption function can be defined exactly as given in the question — characters can be converted to their corresponding codes, manipulated by addition of the `shift`, and converted back into alphabetic form.

`main (shift:Integer, plainText:String)`

1. Normalize `plainText` to uppercase.
2. Normalize `shift` by replacing it with `shift` mod 26.
3. Initialize an empty String `cipherText`.
4. Initialize a counter `i` to `0`.
5. If `i` is less than the length of `plainText`, proceed. Otherwise, jump to (6).
   (a) Store the character in `plainText` at position `i` in a variable `plain`.
   (b) Initialize an empty character `crypt`.
   (c) If `plain` is not an alphabet, assign `plain` to `crypt` and jump to (5g).
   (d) Convert `plain` into a number, such that `A` is mapped to `0`, `B` to `1` and so on. Store this in a temporary variable `n`.

(e) Add `shift` to `n`, calculate its least residue modulo 26[4], and store the result in `n`.
(f) Convert `n` back into a character and store the result in `crypt`.
(g) Append `crypt` to `cipherText`.
(h) Increment `i` by `1` and jump to (5).

6. Display `cipherText`.
7. **Exit**

Source Code

```
public class CaesarShift {
	public static void main (String[] args) {
		try {
			/* Parse the first command line argument as the shift */
			int shift = Integer.parseInt(args[0]) % 26;
			/* Parse the second command line argument as the text to
			    encrypt */
			String plaintext = args[1].toUpperCase();
			String ciphertext = "";
			for (int i = 0; i < plaintext.length(); i++) {
				char plain = plaintext.charAt(i);
				char crypt = ' ';
				if ('A' <= plain && plain <= 'Z') {
					/* Only shift letters of the alphabet */
					crypt = numToChar(charToNum(plain) + shift);
				} else {
					/* Keep special characters intact */
					crypt = plain;
				}
				/* Append the encrypted character to the cipherText */
				ciphertext += crypt;
			}
			System.out.println(ciphertext);
		} catch (NumberFormatException | IndexOutOfBoundsException e) {
			/* Handle missing or incorrectly formatted arguments */
			System.out.println("Enter 2 arguments (shift[integer],
			    plaintext[text])!");
		}
	}

	/* Map letters to numbers */
	public static int charToNum (char letter) {
```

[4]The set of integers $K = \{0, 1, 2, \ldots, n-1\}$ is called the least residue system modulo n. The number k such that $k \in K$ and $a \equiv k \pmod{n}$ is called the least residue of a modulo n.

```
31                return Character.toUpperCase(letter) - 'A';
32        }
33
34        /* Map numbers to letters */
35        public static char numToChar (int number) {
36                return (char) ('A' + Math.floorMod(number, 26));
37        }
38  }
```

Variable Description

CaesarShift::main(String[])		
int	shift	The inputted 'shift'
String	plainText	The text to encrypt
String	cipherText	The encrypted text
int	i	Counter variable, stores the position in plainText
char	plain	The character to encrypt
char	crypt	The encrypted form of plain

CaesarShift::charToNum(char)		
char	letter	The character to convert to an integer

CaesarShift::numToChar(int)		
int	number	The number to convert to a character

"There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors."

— Leon Bambrick

Problem 4 A *palindrome* is a sequence of characters which reads the same backwards as well as forwards. For example, `madam`, `racecar` and `kayak` are words which are palindromes. Similarly, the sentence "`A man, a plan, a canal -- Panama!`" is also a palindrome.

Analyze a sentence of input and display all *words* which are palindromes. If the entire *sentence* is also a palindrome, display it as well.

(A word is an unbroken sequence of characters, separated from other words by whitespace. Ignore single letter words such as I *and* a*. Ignore punctuation, numeric digits, whitespace and case while analyzing the entire sentence.)*

Solution The main challenge here is intelligently dividing a *sentence* into its component *words*. Verifying whether a sequence of characters is a palindrome is fairly simple — extracting those characters from a string of alphabets, numbers, punctuation and whitespace is not.

The main idea behind isolating words from sentences is to define two *markers* — a `start` to keep track of the boundary between whitespace and letters, and an `end` to mark the boundary between letters and whitespace. In this way, the markers can inch their way along the sentence, isolating words in the process. Managing the order of condition checking and incrementing of counters does require some careful manoeuvring in order to avoid any *off-by-1 errors*[5] — any of which would inevitably result in incorrect, hence undesirable output.[citation needed]

`main ()`

1. Accept a string as input, store it in a variable `sentence`.
2. Call `checkWords(sentence)` and `checkSentence(sentence)`. Store the returned values in booleans.
   (a) If either of them is `true`, set a boolean `foundPalindrome` to `true`, otherwise set it to `false`.
3. Display a suitable message if `foundPalindrome` is `false`.
4. **Exit**

[5] An off-by-one error often occurs in computer programming when an iterative loop iterates one time too many or too few.

`checkWords (sentence:String)`

1. Initialize a boolean `foundPalindrome` to false.
2. Initialize two integer counters: `start` to `-1`, `end` to `0`.
3. If `end` is less than the length of `sentence`, proceed. Otherwise, jump to (4).
   (a) Increment `start` as long as the character at the `[start + 1]` position in `sentence` is whitespace.
   (b) Assign `end` to `start`.
   (c) Increment `end` as long as it does not exceed the length of `sentence` and the character at the `[end]` position in `sentence` is not whitespace.
   (d) Assign the string of characters between `start` and `end` from `sentence` (inclusive, exclusive) to a variable `word`.
   (e) Call `isPalindrome(word)`. If `word` is a palindrome:
      i. Set `foundPalindrome` to `true`.
      ii. Display `word`.
   (f) Assign `end - 1` to `start`.
   (g) Jump to (3)
4. **Return** `foundPalindrome`

`checkSentence (sentence:String)`

1. Call `isPalindrome(sentence)`. If `sentence` is a palindrome:
   (a) Display `word`.
   (b) **Return** `true`.
2. **Return** `false`.

`isPalindrome (text:String)`

1. Normalize `text` by converting it into uppercase and removing all non-alphabetic characters.
2. Let the length of `text` be labeled temporarily as `t`.
3. Initialize two integer counters: `i` to `0`, `j` to `l - 1`.
4. If `i` is less than `j`, proceed. Otherwise, jump to (5).
   (a) If the characters at positions `i` and `j` in `text` are not equal, **return** `false`.
   (b) Increment `i` by `1`.
   (c) Decrement `j` by `1`.
   (d) Jump to (4)
5. **Return** `true` only if `text` is longer than one character. Otherwise, **return** `false`.

Source Code

```
import java.util.Scanner;

public class Palindrome {
	public static void main (String[] args) {
		System.out.print("Enter your sentence : ");
		String sentence = (new Scanner(System.in)).nextLine().trim();
		/* Keep track of whether palindromes have been found */
		boolean foundPalindrome = false;
		System.out.println("Palindromes : ");
		foundPalindrome |= checkWords(sentence);
		foundPalindrome |= checkSentence(sentence);
		if (!foundPalindrome) {
			System.out.println("(No palindromes found!)");
		}
	}

	/* Slice a sentence into words and check each individually */
	public static boolean checkWords (String sentence) {
		boolean foundPalindrome = false;
		int start = -1;
		int end = 0;
		while (end < sentence.length()) {
			while (Character.isWhitespace(sentence.charAt(++start)));
			end = start;
			while (end < sentence.length() &&
			    !Character.isWhitespace(sentence.charAt(end++)));
			String word = sentence.substring(start, end).trim();
			if (isPalindrome(word)) {
				foundPalindrome = true;
				System.out.println(getAlphabets(word));
			}
			start = end - 1;
		}
		return foundPalindrome;
	}

	/* Check the sentence as a whole */
	public static boolean checkSentence (String sentence) {
		if (isPalindrome(sentence)) {
			System.out.println("The sentence '" + sentence + "' is a
			    palindrome.");
			return true;
		}
		return false;
```

```
43          }
44
45          /* Check whether a piece of text is identical forward as well as backwards */
46          public static boolean isPalindrome (String text) {
47                  String rawText = getAlphabets(text).toUpperCase();
48                  for (int i = 0, j = rawText.length() - 1; i < j; i++, j--) {
49                          if (rawText.charAt(i) != rawText.charAt(j)) {
50                                  return false;
51                          }
52                  }
53                  /* Make sure that the text is not just one letter */
54                  return (rawText.length() > 1);
55          }
56
57          /* Strip a piece of text of all characters except alphabetic ones */
58          public static String getAlphabets (String text) {
59                  String rawText = "";
60                  for (int i = 0; i < text.length(); i++) {
61                          if (Character.isAlphabetic(text.charAt(i))) {
62                                  rawText += text.charAt(i);
63                          }
64                  }
65                  return rawText;
66          }
67  }
```

Variable Description

Palindrome::main(String[])		
String	sentence	Stores the text to check for palindromes
boolean	foundPalindrome	Stores whether palindromes have been found

Palindrome::checkWords(String)		
String	sentence	Stores the sentence to divide into words
boolean	foundPalindrome	Stores whether palindromes have been found
int	start	Counter variable, stores the index of the start of a word
int	end	Counter variable, stores the index of the end of a word
String	word	Stores words in sentence, extracted between start and end

Palindrome::checkSentence(String)		
String	sentence	Stores the sentence to divide into words

Palindrome::isPalindrome(String)		
String	text	Stores the text to check
String	rawText	Stores only alphabets from text
int	i	Counter variable, stores the current index in text

Palindrome::getAlphabets(String)		
String	text	Stores the text to extract alphabets from
String	rawText	Stores only alphabets from text
int	i	Counter variable, stores the current index in text

"In programming the hard part isn't solving problems, but deciding what problems to solve."

— **Paul Graham**

Problem 5 A *prime number* (or a *prime*) is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Display all primes upto a given limit, along with their number.

Solution This problem can be tackled in a multitude of ways.[citation needed] We could define a function for checking the primality of a given number, then iterate through all numbers in the required range. A common way of checking for primality is *trial division*. It consists of testing whether the number n is a multiple of any integer between 2 and $\sqrt{n}$. Although this works well enough for small numbers, repeating this consecutively for very large inputs is tedious and inefficient. Since the problem consists of identifying primes in a *range*, and not individually, we can make use of more efficient methods.

The *Sieve of Eratosthenes* is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite the multiples of each prime, starting with the first prime number, 2. As a result, when a prime p is found, none of its multiples will be tested further for primality — they are eliminated early on. In comparison, *trial division* has worse theoretical complexity than that of the *Sieve of Eratosthenes* in generating ranges of primes. When testing each prime, the optimal trial division algorithm uses all prime numbers not exceeding its square root, whereas the Sieve of Eratosthenes produces each composite only from its prime factors.

`main (upperLimit:Integer)`

1. Create a new `SieveOfEratosthenes`, pass it `upperLimit` and assign it to `sieve`.
2. Call `sieve->sievePrimes()`.
3. Display the indices which correspond to `true` in the boolean array `sieve->primes`.
4. **Exit**

`SieveOfEratosthenes (upperLimit:Integer)`

1. Initialize a boolean array `primes`, indexed with integers from `[0]` to `[upperLimit - 1]`, with all elements set to `true`.
2. Set `primes[0]` and `primes[1]` to `true`.
3. **Define** the function `SieveOfEratosthenes::sievePrimes()` and **return** the resultant object.

`SieveOfEratosthenes::sievePrimes ()`

1. Initialize an integer variable `prime` to 2.
2. If `prime` is less than the square root of `upperLimit`, proceed. Otherwise, **return**.
   (a) Initialize an integer variable `multiple` to the square of `prime`.
   (b) If `multiple` is less than `upperLimit`, proceed. Otherwise, jump to (2c).
      i. Set `primes[multiple]` to `false`.
      ii. Increment `multiple` by `prime`.
      iii. Jump to (2b)
   (c) Increment `prime` until `primes[prime]` is `true`.
   (d) Jump to (2).
3. **Return**

Source Code

```
1  public class SieveOfEratosthenes {
2         private final int upperLimit;
3         private boolean[] primes;
4
5         /* Initialize the list of numbers using an upper limit */
6         public SieveOfEratosthenes (int upperLimit) {
7                 this.upperLimit = upperLimit;
8                 this.initPrimes();
9         }
10
11        public boolean[] getPrimes () {
12                return primes;
13        }
14
15        /* Initialize all value to 'prime' by default */
16        public void initPrimes () {
17                this.primes = new boolean[upperLimit];
18                /* Mark known values as 'not prime' */
19                primes[0] = false;
20                primes[1] = false;
21                for (int i = 2; i < upperLimit; i++)
22                        primes[i] = true;
23        }
24
25        /* Iteratively sieve the numbers to leave primes behind */
26        public void sievePrimes () {
27                /* Start with the first prime */
28                int prime = 2;
```

```
29              while ((prime * prime) < upperLimit) {
30                      /* Start with the first multiple not crossed off */
31                      int multiple = prime * prime;
32                      while (multiple < upperLimit) {
33                              /* Cross multiples of a prime off the list */
34                              primes[multiple] = false;
35                              multiple += prime;
36                      }
37                      /* Skip forward to the next prime */
38                      while (!primes[++prime]);
39              }
40      }
41  }
```

```
1   public class Primes {
2       public static void main (String[] args) {
3               try {
4                       /* Parse the first command line argument as the upper limit
5                          on primes to calculate */
6                       int upperLimit = Integer.parseInt(args[0]);
7                       if (upperLimit < 2) {
8                               throw new NumberFormatException();
9                       }
10                      SieveOfEratosthenes sieve = new
                            SieveOfEratosthenes(upperLimit);
11                      sieve.sievePrimes();
12                      showPrimes(sieve.getPrimes());
13              } catch (NumberFormatException | IndexOutOfBoundsException e) {
14                      /* Handle missing or incorrectly formatted arguments */
15                      System.out.println("Enter 1 argument (limit[integer, >1])!");
16                      System.out.println("(Primes will be displayed up to, not
                            including 'limit')");
17              }
18      }
19
20      /* Display all primes calculated */
21      public static void showPrimes (boolean[] primes) {
22              int primeCount = 0;
23              /* Format all number to the same width */
24              int maxLength = Integer.toString(primes.length).length();
25              for (int i = 0; i < primes.length; i++) {
26                      /* If 'i' is prime, primes[i] will be marked 'true' */
27                      if (primes[i]) {
28                              System.out.printf("%" + maxLength + "d ", i);
29                              primeCount++;
```

```
30                      }
31                }
32                System.out.println("\nTotal number of primes : " + primeCount);
33        }
34  }
```

Variable Description

SieveOfEratosthenes		
int	upperLimit	The number of integers to sieve
boolean[]	primes	Primes, with contents indicating the primality of the index

SieveOfEratosthenes::initPrimes()		
int	i	Counter variable

SieveOfEratosthenes::sievePrimes()		
int	prime	Counter variable, stores current primes found
int	multiple	Counter variable, stores the multiples of prime

Primes::main(String[])		
int	upperLimit	The highest integer to check for primality (exclusive)
SieveOf Eratosthenes	sieve	An object capable of sieving primes

Primes::showPrimes(boolean[])		
boolean[]	primes	Primes, with contents indicating the primality of the index
int	primeCount	The number of primes found
int	maxLength	The length of the longest number to display
int	i	Counter variable, stores the current integer to check for primality

"Any fool can use a computer. Many do."

— **Ted Nelson**

Problem 6 Design a simple interface for an examiner which can format and display marks scored by a group of students in a particular examination. Calculate the percentage scored by each candidate and display the list of students and percentages in an ASCII bar chart, arranged alphabetically.

Solution This problem calls for a fairly straightforward flow of logic. The main goal is to present the user with a simple way of providing input, along with nicely formatted output.

`main (upperLimit:Integer)`

1. Input the maximum marks allotted for the examination as a floating point. Store it as `maxMarks`.
2. Input the total number of students whose marks are to be recorded as an integer. Store it as `numberOfStudents`.
3. Create a new `Marksheet`, pass it `maxMarks`, `numberOfStudents` and assign it to `sheet`.
4. Initialize an integer counter `i` to `0`;
5. If `i` is less than `numberOfStudents`, proceed. Otherwise, jump to (6).
   (a) Input a student's name as a string. Store it as `name`.
   (b) Input the student's marks as a floating point. Store it as `marks`.
   (c) Call `sheet->addMarks(name, marks)`.
   (d) Jump to (5).
6. Call `sheet->sortByName()`.
7. Call `sheet->displayChart()`.
8. Call `sheet->sortMaxScorers()`.
9. **Exit**

`Marksheet (maxMarks:FloatingPoint, numberOfStudents:Integer)`

1. Initialize a string array `names`, indexed with integers from `[0]` to `[numberOfStudents - 1]`.
2. Initialize a floating point array `marks`, indexed with integers from `[0]` to `[numberOfStudents - 1]`.
3. Initialize an integer counter `lastStudent` to `-1`.
4. **Define** the functions:

(a) `Marksheet::addMarks(name, score)`
(b) `Marksheet::sortByName()`
(c) `Marksheet::displayChart()`
(d) `Marksheet::displayMaxScorers()`

5. **Return** the resultant object.

`Marksheet::addMarks (name:String, score:FloatingPoint)`

1. Increment `lastStudent` by `1`.
2. Set the `names[lastStudent]` to `name`.
3. Set the `marks[lastStudent]` to `score`.
4. **Return**

`Marksheet::sortByName ()`

1. Assign `lastStudent` to `right`.
2. If `right` exceeds `0`, proceed. Otherwise, **return**.
   (a) Initialize an integer counter `i` to `1`.
   (b) If `i` is less than or equal to `right`, proceed. Otherwise, jump to (2c).
      i. If `names[i-1]` comes lexicographically after `names[i]`:
         A. Swap the elements at `names[i-1]` and `names[i]`.
         B. Swap the elements at `marks[i-1]` and `marks[i]`.
      ii. Jump to (2b).
   (c) Jump to (2).

`Marksheet::displayChart ()`

1. For every string `name` in `names`:
   (a) Calculate the length of the bar in the chart as a fraction of the screen width. Store the calculated number of characters to display as `points`.
   (b) Display `name`, a string of suitable characters for the bar of length `points`, along with the percentage scored.
2. **Return**

`Marksheet::displayMaxScorers ()`

1. Calculate the maximum floating point in `marks` and store it as `maxScore`.
2. For every integer `i` between `0` and `numberOfStudents` (inclusive, exclusive) such that `marks[i]` is equal to the `maxScore`, display `names[i]`.
3. **Return**

Source Code

```
public class Marksheet {
	public static final int SCREEN_WIDTH = 100;
	private final double maxMarks;
	private final int numberOfStudents;
	private int lastStudent;
	private String[] names;
	private double[] marks;

	/* Initialize some final data */
	public Marksheet (double maxMarks, int numberOfStudents) {
		this.maxMarks = maxMarks;
		this.numberOfStudents = numberOfStudents;
		this.names = new String[numberOfStudents];
		this.marks = new double[numberOfStudents];
		this.lastStudent = -1;
	}

	/* Add names and marks to the stack */
	public boolean addMarks (String name, double score) {
		try {
			names[++lastStudent] = name;
			marks[lastStudent] = score;
			return true;
		} catch (IndexOutOfBoundsException e) {
			return false;
		}
	}

	/* Display the names and percentages in a bar chart */
	public void displayChart () {
		System.out.println(Marksheet.multiplyString("-",
		    Marksheet.SCREEN_WIDTH));
		for (int i = 0; i <= lastStudent; i++) {
			/* Calculate the fraction of marks earned */
			double fraction = marks[i] / maxMarks;
			String name = (names[i].length() < 16)
				? names[i]
				: (names[i].substring(0,13) + "...");
			int points = (int) (fraction * (SCREEN_WIDTH - 34));
			/* Generate and pad the bar to display */
			String bar = multiplyString("*", points)
				+ multiplyString(" ", SCREEN_WIDTH - 34 - points);
			System.out.printf("| %16s | %s | %6.2f %% |%n"
					, name
```

```
                                , bar
                                , fraction * 100);
        }
        System.out.println(Marksheet.multiplyString("-",
            Marksheet.SCREEN_WIDTH));
    }

    /* Display the name of students with the highest score */
    public void displayMaxScorers () {
        String maxScorers = "";
        double maxScore = getMaxScore();
        for (int i = 0; i <= lastStudent; i++) {
            if (marks[i] == maxScore) {
                maxScorers += ", " + names[i];
            }
        }
        System.out.println(maxScorers.substring(1)
                + " scored the highest ("
                + maxScore + "/"
                + maxMarks + ")");
    }

    /* Sort the names and associated marks lexicographically */
    public void sortByName () {
        for (int right = lastStudent; right > 0; right--)
            for (int i = 1; i <= right; i++)
                if (names[i-1].compareToIgnoreCase(names[i]) > 0)
                    swapRecords(i, i - 1);

    }

    /* Get the value of the highest score */
    public double getMaxScore () {
        double max = Integer.MIN_VALUE;
        for (int i = 0; i <= lastStudent; i++) {
            max = Math.max(max, marks[i]);
        }
        return max;
    }

    /* Utility function to swap student records */
    private void swapRecords (int x, int y) {
        String tempName = names[x];
        double tempMark = marks[x];
        names[x] = names[y];
        marks[x] = marks[y];
```

```
89                names[y] = tempName;
90                marks[y] = tempMark;
91        }
92
93        /* Utility function for repeating strings */
94        public static String multiplyString (String s, int n) {
95                String out = "";
96                while (n --> 0)
97                        out += s;
98                return out;
99        }
100 }
```

```
1  import java.util.Scanner;
2  import java.util.InputMismatchException;
3
4  public class ScoreRecorder {
5         public static void main (String[] args) {
6                 /* Create an object capable of managing input */
7                 Scanner inp = new Scanner(System.in);
8                 double maxMarks = 0.0;
9                 int numberOfStudents = 0;
10                try {
11                        System.out.print("Enter the maximum marks allotted for each
                              student : ");
12                        maxMarks = inp.nextDouble();
13                        System.out.print("Enter the total number of students : ");
14                        numberOfStudents = inp.nextInt();
15                        /* Check for any erroneous data */
16                        if (maxMarks <= 0) {
17                               System.out.println("Maximum marks must be positive!");
18                               System.exit(0);
19                        }
20                        if (numberOfStudents <= 0) {
21                               System.out.println("Number of students must be
                                     positive!");
22                               System.exit(0);
23                        }
24                        /* Create an object capable of recording scoresheets */
25                        Marksheet sheet = new Marksheet(maxMarks, numberOfStudents);
26                        System.out.println("Enter " + numberOfStudents + " students'
                              names and marks : ");
27                        /* Accept student data */
28                        for (int i = 0; i < numberOfStudents; i++) {
29                               String name = "";
```

```
30                              while (!inp.hasNextDouble()) {
31                                      name += inp.next() + " ";
32                              }
33                              double marks = inp.nextDouble();
34                              if (marks <= 0 || marks > maxMarks) {
35                                      System.out.println("Marks must be within 0.0 and
                                            " + maxMarks + "!");
36                                      System.exit(0);
37                              }
38                              sheet.addMarks(name.trim(), marks);
39                      }
40                      /* Sort and display */
41                      sheet.sortByName();
42                      sheet.displayChart();
43                      sheet.displayMaxScorers();
44              } catch (InputMismatchException e) {
45                      /* Handle missing or incorrectly formatted arguments */
46                      System.out.println("Invalid Input!");
47                      System.exit(0);
48              }
49      }
50  }
```

Variable Description

Marksheet		
int	SCREEN_WIDTH	Number of characters to use in the display width
double	maxMarks	The maximum marks allotted for the examination
int	numberOf Students	The number of students whose marks are to be recorded
int	lastStudent	The index number of the last student added to the marksheet
String[]	names	The names of the students
double[]	marks	The marks of the students

Marksheet::addMarks(String, double)		
String	name	The name of the student to be added
double	score	The marks of the student to be added

Marksheet::displayChart()		
int	i	Counter variable
double	fraction	The fraction on marks scored over the maximum marks

String	name	Temporarily stores a formatted version of a student's name
int	points	The number of characters to display in the bar chart
String	bar	The bar in the chart, along with whitespace padding

Marksheet::displayMaxScorers()		
String	maxScorers	The list of highest scoring students
double	maxScore	The highest score
int	i	Counter variable

Marksheet::sortByName()		
int	right	Counter variable
int	i	Counter variable

Marksheet::getMaxScore()		
double	max	The maximum score in marks
int	i	Counter variable

Marksheet::swapRecords(int, int)		
int	x, y	The indices of the records to swap
String	tempName	Temporary storage of a name
double	tempMark	Temporary storage of a mark

Marksheet::multiplyString(String, int)		
String	s	The string to multiply
int	n	The number of times to multiply s
String	out	The string containing n copies of s

ScoreRecorder::main(String[])		
Scanner	inp	The input managing object
double	maxMarks	The maximum marks allotted for the examination
int	numberOf Students	The number of students whose marks are to be recorded
Marksheet	sheet	An object capable of managing student records
int	i	Counter variable
String	name	The name of the student to be added
double	marks	The marks of the student to be added

"To iterate is human, to recurse divine"

— L. Peter Deutsch

Problem 7 The *determinant* of a square matrix $A_{n,n}$ is defined recursively as follows.

$$det(A_{n,n}) \;=\; \begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} \;=\; \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} \cdot det(M_{i,j})$$

where $M_{i,j}$ is defined as the minor of $A_{n,n}$, an $(n-1) \times (n-1)$ matrix formed by removing the ith row and jth column from $A_{n,n}$.

The determinant of a (2×2) matrix is simply given by

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} \;=\; ad \;-\; bc$$

For example, the determinant of a (3×3) matrix is given by the following expression.

$$\begin{aligned} \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \;&=\; a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &=\; aei + bfg + cdh - ceg - bdi - afh \end{aligned}$$

Calculate the *determinant* of an inputted $(n \times n)$ square matrix.

Solution This problem offers the opportunity to showcase the power of recursive functions. Here, the complex task of calculating the determinant of a large matrix can be subdivided into multiple smaller tasks. In fact, each of these tasks is precisely the same as the larger one — the only difference is the size of the matrices. Eventually, the problem reduces to finding the determinants of multiple (2×2) matrices. The values thus obtained can be pieced together to form the final answer.

`main ()`

1. Input the size (number of rows/columns) of the square matrix. Store it as `size`.
2. Create a new `SquareMatrix`, pass it `size`, and assign it to `matrix`.
3. For each `i` $\in \{1, 2, \ldots,$ `size`$\}$:

(a) For each `j` $\in \{1, 2, \ldots,$ `size`$\}$:
   i. Input an integer as `n`.
   ii. Set the element at `[i, j]` of `matrix` to `n`.
4. Call `matrix->getDeterminant()` and display the returned value.
5. **Exit**

`Matrix (rows:Integer, columns:Integer)`
1. Initialize an integer array of integer arrays `elements`, indexed with integers from `[1]` to `[rows]`, with each contained integer array indexed with integers from `[1]` to `[columns]` .
2. **Return** the resultant object.

`SquareMatrix (size:Integer)`
1. **Define** the functions:
   (a) `SquareMatrix::getDeterminant()`
   (b) `SquareMatrix::getMinorMatrix(row, column)`
2. **Return** a `Matrix`, with both `rows` and `columns` set to `size`.

`SquareMatrix::getDeterminant ()`
1. If the `size` is `1`, **return** the only element (`elements[1, 1]`).
2. If the `size` is `2`, **return** (`elements[1, 1]`$\times$`elements[2, 2]`) $-$ (`elements[1, 2]`$\times$`elements[2, 1]`).
3. Initialize an integer variable `determinant` to `0`.
4. For each `i` $\in \{1, 2, \ldots,$ `size`$\}$:
   (a) Call `this->getMinorMatrix(i, i)->getDeterminant()`. Store the result in `d`.
   (b) Add $((-1)^{\texttt{i}+1} \times$ `matrix[1, i]` $\times$ `d`) to `determinant`.
5. **Return** `determinant`.

`SquareMatrix::getMinorMatrix (row:Integer, column:Integer)`
1. Create a new `SquareMatrix`, pass it `(size - 1)`, and assign it to `minor`.
2. Copy all elements from `this` to `minor`, except for those at position `[row, *]` or `[*, column]`.
3. **Return** `minor`.

Source Code

```
public class Matrix {
	protected final int rows;
	protected final int columns;
	protected int[][] elements;

	/* Initialize a matrix of a given order */
	public Matrix (int rows, int columns) {
		this.rows = rows;
		this.columns = columns;
		this.elements = new int[rows][columns];
	}

	public int getRows () {
		return this.rows;
	}

	public int getColumns () {
		return this.columns;
	}

	/* Set elements in the matrix using natural indices */
	public void setElementAt (int element, int row, int column) {
		if (row < 1 || row > rows || column < 1 || column > columns)
			return;
		elements[row-1][column-1] = element;
	}

	/* Get elements from the matrix using natural indices */
	public int getElementAt (int row, int column) {
		if (row < 1 || row > rows || column < 1 || column > columns)
			return Integer.MIN_VALUE;
		return elements[row-1][column-1];
	}
}
```

```
public class SquareMatrix extends Matrix {
	protected int size;

	/* Initialize the matrix with the same number of rows and columns */
	public SquareMatrix (int size) {
		super(size, size);
		this.size = size;
	}

```

```
10          public int getSize () {
11                  return this.size;
12          }
13
14          /* Recursively calculate the determinant of the matrix */
15          public int getDeterminant () {
16                  /* Base cases */
17                  if (this.size == 1)
18                          return getElementAt(1, 1);
19                  if (this.size == 2)
20                          return (getElementAt(1, 1) * getElementAt(2, 2))
21                                  - (getElementAt(1, 2) * getElementAt(2, 1));
22                  int determinant = 0;
23                  /* Accumulate the determinants of minors with alternating signs */
24                  for (int i = 1; i <= size; i++)
25                          determinant += ((int) Math.pow(-1, 1+i))
26                                  * getElementAt(1, i)
27                                  * getMinorMatrix(1, i).getDeterminant();
28                  return determinant;
29          }
30
31          /* Get the minor matrix by removing a row and a column */
32          public SquareMatrix getMinorMatrix (int row, int column) {
33                  /* Check bounds */
34                  if (row < 1 || row > size || column < 1 || column > size)
35                          return null;
36                  if (this.size <= 1)
37                          return new SquareMatrix(0);
38                  SquareMatrix minor = new SquareMatrix(this.size - 1);
39                  for (int i = 1, p = 1; p < size; i++, p++) {
40                          /* Skip 'row' */
41                          if (i == row)
42                                  i++;
43                          for (int j = 1, q = 1; q < size; j++, q++) {
44                                  /* Skip 'column' */
45                                  if (j == column)
46                                          j++;
47                                  /* Copy values into the new matrix */
48                                  minor.setElementAt(this.getElementAt(i, j), p, q);
49                          }
50                  }
51                  return minor;
52          }
53  }
```

```java
import java.util.Scanner;

public class Determinant {
	public static void main (String[] args) {
		/* Create an object for managing input */
		Scanner inp = new Scanner(System.in);
		try {
			System.out.print("Enter the size of the (size X size) square
			    matrix : ");
			int size = inp.nextInt();
			/* Create a square matrix which has suitable methods for
			    calculation */
			SquareMatrix matrix = new SquareMatrix(size);
			System.out.println("Enter " + (size * size) + " integers : ");
			for (int i = 1; i <= size; i++)
				for (int j = 1; j <= size; j++)
					matrix.setElementAt(inp.nextInt(), i, j);
			System.out.println("\nThe determinant is : " +
			    matrix.getDeterminant());
		} catch (Exception e) {
			/* Handle missing or incorrectly formatted arguments */
			System.out.println("Invalid Input!");
		}
	}

	/* Display the matrix in a neat format */
	public static void showMatrix (Matrix m) {
		for (int i = 1; i <= m.getRows(); i++) {
			for (int j = 1; j <= m.getColumns(); j++) {
				System.out.printf("%4d ", m.getElementAt(i, j));
			}
			System.out.println();
		}
	}
}
```

Variable Description

Matrix		
int	rows	Number of rows in the matrix
int	columns	Number of columns in the matrix
int[][]	elements	The array of integer arrays, storing the elements of the matrix

SquareMatrix		
int	size	Number of both rows and columns in the matrix

SquareMatrix::getDeterminant()		
int	determinant	The determinant of the SquareMatrix
int	i	Counter variable

SquareMatrix::getMinorMatrix(int, int)		
int	row	The row to remove from the matrix
int	column	The column to remove from the matrix
SquareMatrix	minor	The matrix obtained by removing row and column
int	i, j	Counter variables

Determinant::main(String[])		
Scanner	inp	The input managing object
int	size	Number of both rows and columns in the matrix
SquareMatrix	matrix	The matrix whose determinant is to be calculated
int	i, j	Counter variables

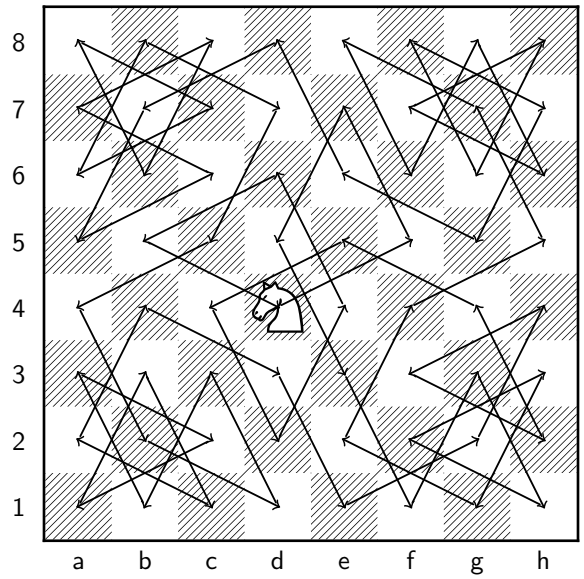
Determinant::showMatrix(Matrix)		
Matrix	m	The matrix to display
int	i, j	Counter variables

"My project is 90% done. I hope the second half goes as well."

— Scott W. Ambler

Problem 8 A *Knight's Tour* is a sequence of moves of a knight on a chessboard such that the *knight* visits every square only once. If the knight ends on a square that is one knight's move from the beginning square, the tour is *closed* forming a closed loop, otherwise it is *open*.

There are many ways of constructing such paths on an empty board. On an 8×8 board, there are no less than $26,534,728,821,064$ *directed*[6] *closed* tours. Below is one of them.

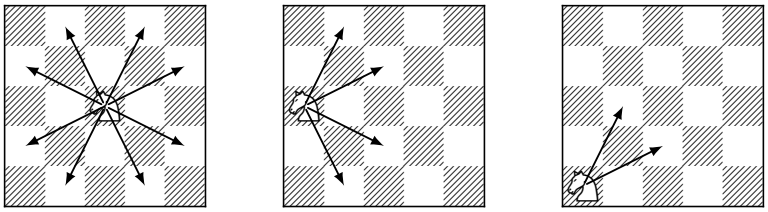


Construct a *Knight's Tour* (*open* or *closed*) on an $n \times n$ board, starting from a given square.

(Mark each square with the move number on which the knight landed on it. Mark the starting square 1*.)*

[6]Two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections.

Solution A knight on a chessboard can move to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally.



The mobility of a knight can make varies greatly with its position on the board — near the centre, it can jump to one of 8 squares while when in a corner, it can jump to only 2. On the other hand, the number of possible *sequences* of squares a knight can traverse grows extremely quickly. Although it may seem that a simple *brute force* search can quickly find one of *trillions* of solutions, there are approximately 4×10^{51} different paths to consider on an 8×8 board. For even larger boards, iterating through every possible path is clearly impractical.[citation needed]

This problem calls for implementing a *backtracking*[7] *algorithm*, coupled with some *heuristic*[8] to speed up the search. One such heuristic is *Warnsdorf's Rule*.

> The knight is moved so that it always proceeds to the square from which the knight will have the *fewest* onward moves.

This allows us to define a ranking algorithm for each possible path — the positions which result in the smallest number of further moves, or is furthest away from the board's centre will be investigated first. In case of a tie, we can either proceed without making any changes to the already existing positions, or introduce a random element. This has the effect of producing different results on successive executions, giving a variety of solutions.

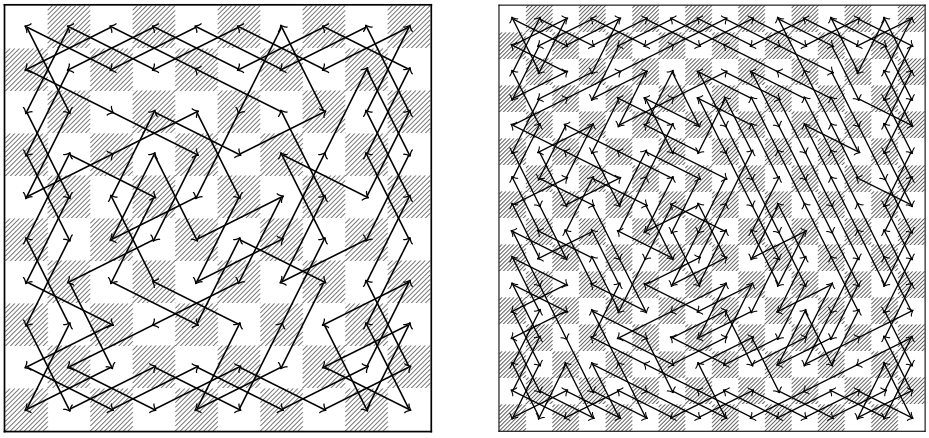
One drawback of resolving ties randomly is that an early "wrong" choice in the position tree can force the calculation of every resulting path without reaching a solution, effectively reducing the algorithm to a brute force search. This is especially problematic

[7] Backtracking is a general algorithm for finding some or all solutions to some computational problems that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

[8] A heuristic technique is any approach to problem solving that employs a practical method not guaranteed to be optimal or perfect, but sufficient for the immediate goals. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution.

for large boards, where it may take hours to backtrack and reach a solution. Thus, the "randomness factor" should be adjusted according to the board size.

A high randomness can be useful for searching specifically for *closed tours*, as a randomness of 0 simply produces the same solution every time (which may or may not be closed). Below are some tours generated by the program.



The tendency of the path to remain close to the edges of the board, where the mobility of the knight is restricted, is clearly evident.

`main (boardSize:Integer, initSquare:Position, randomness:FloatingPoint)`

1. Create a new `TourSolver`, pass it `boardSize`, `initSquare`, `randomness`, and assign it to `t`.
2. Call `t->getSolution()`. Store the returned move stack as `solution`.
3. Display the board obtained by calling `t->getBoard()` along with the moves in `solution`.
4. **Exit**

`TourSolver (size:Integer, initSquare:Position, randomness:FloatingPoint)`

1. Initialize an integer arrays of integer arrays indexed with integers from `[1]` to `[size]`, simulating a chessboard. Store it as `board`, which records the move numbers on which the knight lands on it.
2. Initialize a `Position` stack `path`, along with methods to add and remove `Position`'s from it.
3. Set an integer counter `numberOfMoves` to `0`, as part of the `path` stack.
4. **Define** the functions:
   (a) `TourSolver::solve(p)`

   (b) `TourSolver::getPossibleMoves(p)`
5. **Return** the resultant object.

`TourSolver::solve (p:Position)`

1. If the `path` stack is full, **return** `true`, indicating that the tour has been solved.
2. Call `this->getPossibleMoves(p)`. Store the returned list of possible legal moves as `moves`.
3. Sort `moves`, ranking each possible position according to *Warnsdorf's Rule*.
4. For every `move` in the list `moves`:
   (a) Push `move` onto the `path` stack and `board`.
   (b) If the call `this->solve(move)` returns `true`, **return** `true`. Otherwise, pop `move` from the `path` stack and `board` (*backtrack*).
5. If the list `moves` has been exhausted, **return** `false`, indicating that there are no solutions from the position `p` for that particular move stack.

`TourSolver::getPossibleMoves (p:Position)`

1. Initialize a list of moves `possibleMoves`.
2. For every possible square `move` a knight can jump to from `p` (on an empty board):
   (a) If `move` is currently a legal move, without falling outside the board or on a previously traversed square, add it to `possibleMoves`.
3. **Return** `possibleMoves`

Source Code

```
public class TourSolver {
	private final int size;
	private Position[] path;
	private int numberOfMoves;
	private int[][] board;
	private int[][] degreesOfFreedom;
	private Position initPosition;
	private double tieBreakRandomness;

	/* Store the list of possible changes in the 'x' and 'y' coordinates of
	   a knight on an empty board */
	private static final int[][] KNIGHT_MOVES = {
		{-1, -2}, {-1, 2}, {1, -2}, {1, 2},
		{-2, -1}, {-2, 1}, {2, -1}, {2, 1}
	};

	/* Initialize the board and move stack */
	public TourSolver (int size, Position initPosition, double randomness) {
		this.size = size;
		this.initPosition = initPosition;
		this.tieBreakRandomness = randomness / 2.0;
		this.path = new Position[size * size];
		this.numberOfMoves = 0;
		this.initBoard();
		this.initDegreesOfFreedom();
	}

	/* Reset the board */
	public void resetSolution () {
		this.path = new Position[size * size];
		this.numberOfMoves = 0;
		this.initBoard();
	}

	/* Initialize a blank board */
	private void initBoard () {
		board = new int[size][size];
		for (int i = 0; i < size; i++)
			for (int j = 0; j < size; j++)
				board[i][j] = 0;
	}

	/* Calculate the mobility of a knight on each square */
	private void initDegreesOfFreedom () {
```

```
45              degreesOfFreedom = new int[size][size];
46              for (int i = 0; i < size; i++)
47                      for (int j = 0; j < size; j++)
48                              degreesOfFreedom[i][j] = getPossibleMovesCount(new
                                    Position(i, j));
49      }
50
51      /* Push a move onto the move stack, add it to the board */
52      public boolean addMove (Position p) {
53              if (numberOfMoves < (size * size)) {
54                      path[numberOfMoves++] = p;
55                      board[p.getX()][p.getY()] = numberOfMoves;
56                      return true;
57              }
58              return false;
59      }
60
61      /* Pop a move from the move stack, remove it from the board */
62      public boolean removeMove () {
63              if (numberOfMoves > 0) {
64                      Position p = path[numberOfMoves - 1];
65                      /* Empty squares are marked '0' */
66                      board[p.getX()][p.getY()] = 0;
67                      path[--numberOfMoves] = null;
68                      return true;
69              }
70              return false;
71      }
72
73      public int[][] getBoard () {
74              return board;
75      }
76
77      /* Get the stack of moves comprising a knight's tour */
78      public Position[] getSolution () {
79              if (size < 5)
80                      return null;
81              addMove(initPosition);
82              if(solve(initPosition))
83                      return path;
84              return null;
85      }
86
87      /* Recursively solve a tour from a given position */
88      public boolean solve (Position p) {
89              /* If the move stack is full, the tour has been solved */
```

```
90              if (numberOfMoves == (size * size))
91                      return true;
92              /* Get every legal move and rank them using Warnsdorf's Rule */
93              Position[] possibleMoves = getPossibleMoves(p);
94              if (possibleMoves[0] == null)
95                      return false;
96              sortMoves(possibleMoves);
97              for (Position move : possibleMoves) {
98                      if (move != null) {
99                              /* Try a move */
100                             addMove(move);
101                             if (solve(move))
102                                     return true;
103                             /* Backtrack */
104                             removeMove();
105                     }
106             }
107             return false;
108     }
109
110     /* Sort a list of positions using Warnsdorf's Rule */
111     public void sortMoves (Position[] moves) {
112             int count = 0;
113             for (Position p : moves)
114                     if (p != null)
115                             count++;
116             for (int right = count; right > 0; right--)
117                     for (int i = 1; i < right; i++)
118                             if (compareMoves(moves[i-1], moves[i]) > 0)
119                                     swapMoves(i-1, i, moves);
120     }
121
122     /* Compare 2 moves using Warnsdorf's Rule */
123     public int compareMoves (Position a, Position b) {
124             /* Compare the mobilities of the knight */
125             int aCount = getPossibleMovesCount(a);
126             int bCount = getPossibleMovesCount(b);
127             if (aCount != bCount)
128                     return aCount - bCount;
129             /* Compare the mobilities of the knight on an empty board */
130             int aFree = degreesOfFreedom[a.getX()][a.getY()];
131             int bFree = degreesOfFreedom[b.getX()][b.getY()];
132             if (aFree != bFree)
133                     return aFree - bFree;
134             /* Resolve ties using a pre-decided element of randomness */
135             return (Math.random() < tieBreakRandomness)? 1 : -1;
```

```
136         }
137
138         /* Utility function to swap moves in the list of possible moves */
139         private static void swapMoves (int x, int y, Position[] moves) {
140                 Position t = moves[x];
141                 moves[x] = moves[y];
142                 moves[y] = t;
143         }
144
145         /* Get the list of all possible, legal moves not touching a previously
146            traveled square from a given position */
147         public Position[] getPossibleMoves (Position start) {
148                 Position[] possibleMoves = new Position[KNIGHT_MOVES.length];
149                 int i = 0;
150                 for (int[] move : KNIGHT_MOVES) {
151                         /* Generate a new */
152                         int x = start.getX() + move[0];
153                         int y = start.getY() + move[1];
154                         /* Check the legality of that move */
155                         if (isWithinBoard(x, y) && board[x][y] == 0) {
156                                 possibleMoves[i++] = new Position(x, y);
157                         }
158                 }
159                 return possibleMoves;
160         }
161
162         /* Get the number of legal moves */
163         public int getPossibleMovesCount (Position start) {
164                 int i = 0;
165                 for (Position p : getPossibleMoves(start))
166                         if (p != null)
167                                 i++;
168                 return i;
169         }
170
171         /* Check whether a position lies within the board */
172         public boolean isWithinBoard (int x, int y) {
173                 return (x >= 0 && x < size && y >= 0 && y < size);
174         }
175 }
```

```java
public class Position {
	private final int x;
	private final int y;

	/* Initialize using the coordinates on the board */
	public Position (int x, int y) {
		this.x = x;
		this.y = y;
	}

	/* Initialize using the position in algebraic notation */
	public Position (String s) {
		int x = 0;
		int i = 0;
		while (i < s.length() && Character.isAlphabetic(s.charAt(i))) {
			x = (x * 26) + Character.toLowerCase(s.charAt(i)) - 'a' + 1;
			i++;
		}
		int y = Integer.parseInt(s.substring(i));
		this.x = x - 1;
		this.y = y - 1;
	}

	public int getX () {
		return x;
	}

	public int getY () {
		return y;
	}

	public boolean equals (Position p) {
		return (p != null)
			&& (this.getX() == p.getX()) && (this.getY() == p.getY());
	}

	@Override
	public String toString () {
		return xToString(this.x) + (this.y + 1);
	}

	/* Convert a file number to its algebraic notation form */
	public static String xToString (int n) {
		int x = n + 1;
		String letters = "";
		while (x > 0) {
```

```
                letters = (char) ('a' + (--x % 26)) + letters;
                x /= 26;
            }
            return letters;
        }
}
```

```
public class KnightTour {
        public static void main (String[] args) {
            try {
                /* Parse the first command line argument as the size of the
                    board */
                int boardSize = Integer.parseInt(args[0]);
                if (boardSize <= 0)
                    throw new NumberFormatException();
                /* Parse the second command line argument as the starting
                    square
                  of the knight, written in algebraic notation */
                String initSquare = (args.length > 1)? args[1] : "a1";
                /* Parse the third command line argument as the degree of
                  randomness to be used while resolving ties */
                double randomness = (args.length > 2)?
                    Double.parseDouble(args[2])
                        : Math.pow(0.8, boardSize) * 2;
                /* Create an object capable of solving knight's tours */
                TourSolver t = new TourSolver(boardSize, new
                    Position(initSquare), randomness);
                Position[] solution = t.getSolution();
                if (solution != null) {
                    showBoard(t.getBoard());
                    showMoves(solution);
                    if (isClosed(solution))
                        System.out.println("\nThe tour is Closed!");
                } else {
                    System.out.println("No Knight's Tours found!");
                }
            } catch (Exception e) {
                /* Handle missing or incorrectly formatted arguments */
                System.out.print("Enter an integer (> 1) as the first
                    argument, ");
                System.out.println("and a well formed chessboard coordinate as
                    the second!");
                System.out.println("                          (size,
                    startSquare * , randomness * )");
                System.out.println();
```

```
32                  System.out.println("(size     -> Solve a Tour on a (size x
                        size) board)");
33                  System.out.println("(startSquare * -> A square in algebraic
                        chess notation of the form 'fr',");
34                  System.out.println("            where f = the letter
                        representing the file(column)");
35                  System.out.println("            and  r = the number
                        representing the rank(row).)");
36                  System.out.println("(startSquare is set to 'a1' by default)");
37                  System.out.println("(randomness * -> A number between 0(no
                        randomness) and 1(even chances),");
38                  System.out.println("            determining the randomness in
                        ranking positions of");
39                  System.out.println("            the same weightage while
                        searching. A randomness of 0 will");
40                  System.out.println("            produce the same tour every
                        time, for a specific size and");
41                  System.out.println("            startSquare. Keep extremely
                        small values of randomness for");
42                  System.out.println("            very large boards.)");
43                  System.out.println("(randomness is set to 2 * (0.8)^boardSize
                        by default)");
44                  System.out.println();
45                  System.out.println("                                        <
                        * = optional arguments >");
46          }
47      }
48
49      /* Display the board, with each square marked with the move number on which
50         the knight landed on it */
51      public static void showBoard (int[][] board) {
52              String hLine = "  " + multiplyString("+-----", board.length) + "+";
53              System.out.println(hLine);
54              for (int column = board.length - 1; column >= 0; column--) {
55                      System.out.printf(" %2d ", column + 1);
56                      for (int row = 0; row < board.length; row++) {
57                              System.out.printf("| %3d ", board[row][column]);
58                      }
59                      System.out.printf("|%n%s%n", hLine);
60              }
61              System.out.print("  ");
62              for (int i = 0; i < board.length; i++) {
63                      System.out.printf(" %2s ", Position.xToString(i));
64              }
65              System.out.println();
66      }
```

```
67
68          /* Display the list of moves in the tour in algebraic notation */
69          public static void showMoves (Position[] moves) {
70                  System.out.print("\nMoves : ");
71                  String movesOut = "";
72                  for (int i = 1; i < moves.length; i++) {
73                          movesOut += (moves[i-1] + "-" + moves[i] + ", ");
74                  }
75                  System.out.println(movesOut.substring(0, movesOut.length() - 2));
76          }
77
78          /* Utility function for repeating strings */
79          public static String multiplyString (String s, int n) {
80                  String result = "";
81                  while (n --> 0)
82                          result += s;
83                  return result;
84          }
85
86          /* Check whether a tour is closed or not */
87          public static boolean isClosed (Position[] path) {
88                  int l = path.length - 1;
89                  int dX = Math.abs(path[0].getX() - path[l].getX());
90                  int dY = Math.abs(path[0].getY() - path[l].getY());
91                  return (dX == 1 && dY == 2) || (dX == 2 && dY == 1);
92          }
93  }
```

Variable Description

TourSolver		
int	size	Number of files/ranks in the chessboard
Position[]	path	Stack of moves which are part of the solved tour
int	numberOfMoves	Counter variable, number of moves made in the solved tour
int[][]	board	An integer array of integer arrays, representing a chessboard, with each square marked with the move number at which the knight lands on it
int[][]	degreesOf Freedom	An integer array of integer arrays, representing a chessboard, with each square marked with the number of possible knight moves from it (on an empty board)
Position	initPosition	The position on the board the knight starts from
double	tieBreak Randomness	The degree to which a move in the path is randomly decided
int[][]	KNIGHT_MOVES	List of legal changes in the x and y positions of a knight
TourSolver::initBoard()		
int	i, j	Counter variables
TourSolver::initDegreesOfFreedom()		
int	i, j	Counter variables
TourSolver::addMove(Position)		
Position	p	The new position to add to the path stack
TourSolver::removeMove()		
Position	p	The position popped from the path stack
TourSolver::solve()		
Position[]	possible Moves	List of possible moves that can be added to the path stack
Position	move	Current move to evaluate in the path
TourSolver::sortMoves(Position[])		
Position[]	moves	List of moves to rank using Warnsdorf's heuristic
int	count	Total number of moves in moves
int	right	Counter variable
int	i	Counter variable

TourSolver::compareMoves(Position, Position)		
Position	a, b	Positions/moves to compare using Warnsdorf's heuristic
int	aCount, bCount	Respective number of possible legal moves for a and b
int	aFree, bFree	Respective number of possible legal moves on an empty board for a and b

TourSolver::swapMoves(int, int, Position[])		
int	x, y	The indices of the moves to swap
Position[]	moves	Array of moves containing the moves to be swapped

TourSolver::getPossibleMoves(Position)		
Position	start	Position from where possible moves are to be generated
int	i	Counter variable
int[]	move	Pair of legal changes in the x and y positions of a knight
int	x, y	New x and y positions of the knight

TourSolver::getPossibleMovesCount(Position)		
Position	start	Position from where possible moves are to be generated
Position	p	Possible position

TourSolver::isWithinBoard(int, int)		
int	x, y	The x and y positions on the board to verify

Position		
int	x, y	The x and y coordinates on the board encoded by the Position

Position::this(String)		
String	s	Chess position written in algebraic notation
int	x, y	The x and y coordinates on the board
int	i	Counter variable

Position::xToString(int)		
int	n	File (x position) to convert to algebraic notation
String	letters	n expressed as a base 26 number, digits starting from (a)

`int`	`x`	Counter variable, temporarily stores the file to convert

`KnightTour::main(String[])`		
`int`	`boardSize`	Number of files/ranks in the chessboard
`String`	`initSquare`	The position on the board the knight starts from (algebraic notation)
`double`	`randomness`	The degree to which a move in the path is randomly decided
`TourSolver`	`t`	An object capable of generating *knight's tours*
`Position[]`	`solution`	The solved sequence of moves in the *knight's tour*

`KnightTour::showBoard(int[][])`		
`int[][]`	`board`	An integer array of integer arrays, representing a chessboard, with each square marked with the move number at which the knight lands on it
`String`	`hline`	Horizontal line drawn to represent board squares
`int`	`row, column, i`	Counter variables

`KnightTour::showMoves(Position[])`		
`Position[]`	`moves`	The sequence of moves to display
`int`	`i`	Counter variable

`KnightTour::multiplyString(String, int)`		
`String`	`s`	The string to multiply
`int`	`n`	The number of times to multiply `s`
`String`	`out`	The string containing `n` copies of `s`

`KnightTour::isClosed(Position[])`		
`Position[]`	`path`	The solved sequence of moves in the *knight's tour*
`int`	`l`	Index of last move in `path`
`int`	`dX, dY`	Differences in x and y coordinates of the knight between the first and last moves

"Curiosity begins as an act of tearing to pieces, or analysis."

— **Samuel Alexander**

Problem 9 Calculate the *square root* of a given positive number, using only *addition*, *subtraction*, *multiplication* and *division*.

Solution The problem of finding the *square root* of a positive real number k is equivalent to finding a positive root of the function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$

$$f(x) \;=\; x^2 - k$$

This problem can be solved using *Newton's method*. *Newton's method* is an iterative process for finding a root of a general function $f : \mathbb{R} \to \mathbb{R}$ by creating an initial guess, then improving upon it.

Let f' denote the derivative of the function f. Thus, the equation of the tangent to the curve $f(x)$, drawn through the point $(x_n, f(x_n))$ is given by the following equation.

$$y \;=\; f'(x_n)(x - x_n) + f(x_n)$$

The idea here is that the *x-intercept* of this tangent will be a better approximation to the root of the function f. Setting $y = 0$, solving for x and renaming it to x_{n+1} yields the following expression.

$$x_{n+1} \;=\; x_n \;-\; \frac{f(x_n)}{f'(x_n)}$$

Plugging in the required function for this problem, we have

$$x_{n+1} \;=\; x_n \;-\; \frac{x_n^2 - k}{2x_n}$$

Simplifying, we arrive at our expression for the term x_{n+1} in our iterative process.

$$x_{n+1} \;=\; \frac{1}{2}\left(x_n \;+\; \frac{k}{x_n}\right)$$

This is the sort of simple expression we have been looking for, involving only one addition and two multiplications per iteration. As n becomes very large, the term x_n approaches the *square root* of k.

`main (number:FloatingPoint, maxIterations:Integer)`

1. Call `squareRoot(number, maxIterations)`. Store the result in `root`.
2. Display `root`, along with the error from the value calculated by the library function `Math->sqrt(number)`.
3. **Exit**

`squareRoot (n:FloatingPoint, maxIterations:Integer)`

1. Store the initial guess `n / 2` in the variable `x`.
2. For `maxIterations` times:
   (a) Calculate `0.5 * (x + (n / x))`. Store the result back in `x`.
3. **Return** `x`

Source Code

```
1  public class SquareRoot {
2        public static void main (String[] args) {
3              /* Parse the first command line argument as the number to square root
                   */
4              double number = Double.parseDouble(args[0]);
5              /* Parse the second conmand line argument as the number of iterations.
6                Default to 100 */
7              int maxIterations = (args.length > 1)? Integer.parseInt(args[1]) :
                   100;
8
9              double root = squareRoot(number, maxIterations);
10             double library_root = Math.sqrt(number);
11
12             /* Display the calculated root, along with a comparison with the
13               library calculated value */
14             System.out.printf("Calculated square root :  %f%n", root);
15             System.out.printf("System library square root : %f%n", library_root);
16             System.out.printf("Error : %f%n", (root - library_root));
17       }
18
19       public static double squareRoot (double n, int maxIterations) {
20             /* Handle edge cases, ignore negative values */
21             if (n < 0)
22                   return Double.NaN;
23             if (n == 0)
24                   return 0.0;
25             /* Start by guessing half of the number */
26             double x = n / 2;
```

```
27              for (int i = 0; i < maxIterations; i++) {
28                      x = 0.5 * (x + (n / x));
29              }
30              return x;
31      }
32  }
```

Variable Description

SquareRoot::main(String[])		
double	number	Stores the number whose square root is to be extracted
int	maxIterations	Stores the number of iterations for which Newton's method is to be applied
double	root	Stores the calculated square root of number
double	library_root	Stores the square root of number given by the Java library

SquareRoot::squareRoot(double, int)		
double	x	Stores the results of successive iterations of Newton's method
int	i	Counter variable

"Objects are abstractions of processing. Threads are abstractions of schedule."

— **James O. Coplien**

Problem 10 Let a *fraction* here be restricted to the ratio of two integers, m and n, where $n \neq 0$. Thus, a fraction $\frac{m}{n}$ is said to be reduced its *lowest terms* when m and n are relatively prime.

Implement this model of *fractions*, such that they are *immutable* and reduced to their *lowest terms* by default. Also implement a simple method for adding two *fractions*.

Solution The problem of reducing a fraction $\frac{m}{n}$ to its lowest terms can be solved simply by dividing the numerator and the denominator by their *greatest common divisor*, i.e., $\gcd(m, n)$. This works as $\gcd(p, q) = 1$ if and only if p and q are relatively prime. Fraction addition can also be implemented using the following formula.

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

The *greatest common divisor* of two integers can be calculated recursively using *Euclid's algorithm*.

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

`main ()`

1. Create 2 `Fraction` objects `a` and `b` using data supplied by the user.
2. Call `Fraction->addFractions(a, b)`. Store the result in another `Fraction` object `sum`.
3. Display `a`, `b` and `sum`.
4. **Exit**

`Fraction (numerator:Integer, denominator:Integer)`

1. Set internal variables `numerator` and `denominator`, keeping them private.
2. Reduce the fraction to its lowest form.
   (a) Calculate the *greatest common divisor* of `numerator` and `denominator`, then divide each by the result.
   (b) Shift any negative sign in `denominator` to `numerator`.
3. **Define** the function `Fraction::addFractions(fraction1, fraction2)`, and **return** the resultant object.

`Fraction::addFractions (fraction1:Fraction, fraction2:Fraction)`

1. Calculate the `numerator` and `denominator` of the sum using the formula discussed above.
2. Create a new `Fraction` object using the calculated `numerator` and `denominator`, then **return** it.

Source Code

```
1  public class Fraction {
2
3         /* Store the numerator and denominator */
4         protected int numerator;
5         protected int denominator;
6
7         public Fraction (int numerator, int denominator) {
8                 /* Handle invalid fractions */
9                 if (denominator == 0)
10                        throw new ArithmeticException("Division by zero!");
11
12                this.numerator = Math.abs(numerator);
13                this.denominator = Math.abs(denominator);
14
15                if (numerator != 0) {
16                        /* Reduce to lowest terms */
```

```
17                      int g = gcd(this.numerator, this.denominator);
18                      this.numerator /= g;
19                      this.denominator /= g;
20                      /* Make sure that the sign is on the numerator */
21                      this.numerator *= Math.signum(numerator) *
                            Math.signum(denominator);
22              } else {
23                      /* Make sure all 'zero fractions' are the same */
24                      this.denominator = 1;
25              }
26      }
27
28      public int getNumerator () {
29              return this.numerator;
30      }
31
32      public int getDenominator () {
33              return this.denominator;
34      }
35
36      /* Return a String representation of the Fraction for display */
37      public String toString () {
38              /* Format all fractions with denominator '1' as simple integers */
39              if (this.denominator == 1)
40                      return this.numerator + "";
41              return this.numerator + " / " + this.denominator;
42      }
43
44      /* Add 2 Fraction objects */
45      public static Fraction addFractions (Fraction a, Fraction b) {
46              int sumNumerator = (a.getNumerator() * b.getDenominator()) +
                    (a.getDenominator() * b.getNumerator());
47              int sumDenominator = a.getDenominator() * b.getDenominator();
48              return new Fraction(sumNumerator, sumDenominator);
49      }
50
51      /* Calculate the greatest common divisor of integers, using Euclid's method
            recursively */
52      private static int gcd (int p, int q) {
53              return (p < q)? gcd(q, p) : ((p % q) == 0)? q : gcd(q, p % q);
54      }
55  }
```

```java
import java.util.Scanner;

public class FractionAdder {
	public static void main (String[] args) {
		Scanner inp = new Scanner(System.in);
		try {
			/* Get the two fractions from user input */
			System.out.print("Enter the numerator and denominator [integer
			    integer] of the first fraction : ");
			Fraction a = new Fraction(inp.nextInt(), inp.nextInt());
			System.out.print("Enter the numerator and denominator [integer
			    integer] of the second fraction : ");
			Fraction b = new Fraction(inp.nextInt(), inp.nextInt());

			/* Calculate and display the sum of the fractions.
			   Here, we take advantage of the toString() method defined for
			       Fractions */
			Fraction sum = Fraction.addFractions(a, b);
			System.out.printf("%n(%s) + (%s) = (%s) %n", a, b, sum);
		} catch (ArithmeticException e) {
			System.out.println("Invalid fraction - division by zero!");
		}
	}
}
```

Variable Description

`Fraction`		
`int`	`numerator`	Stores the numerator of the fraction
`int`	`denominator`	Stores the denominator of the fraction

`Fraction(int, int)`		
`int`	`g`	Stores the greatest common divisor of `numerator` and `denominator`

`Fraction::addFractions(Fraction, Fraction)`		
`Fraction`	`a, b`	The two fractions to be added
`int`	`sumNumerator`	The numerator of the sum
`int`	`sumDenominator`	The denominator of the sum

`FractionAdder::main(String[])`		
`Scanner`	`inp`	The input managing object
`Fraction`	`a, b`	The two fractions to be added
`Fraction`	`sum`	The sum of the fractions `a` and `b`

"Dividing one number by another is mere computation; knowing what to divide by what is mathematics."

— **Jordan Ellenberg**

Problem 11 A rational number q can be broken down into a *simple continued fraction* in the form given below.

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

This may be represented by the abbreviated notation $[a_0; a_1, a_2, \ldots, a_n]$. For example, $[0; 1, 1, 2, 1, 4, 2]$ is shorthand for the following.

$$\frac{42}{73} = 0 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{2}}}}}}$$

Calculate the *simple continued fraction* expression for a given, positive fraction.

Solution We can thus solve this problem recursively by noting that the following holds.

$$\frac{p}{q} = \underbrace{\left\lfloor \frac{p}{q} \right\rfloor}_{\text{Integer part}} + \underbrace{\frac{p \bmod q}{q}}_{\text{Fractional part}}$$

Thus, by defining $f(\frac{p}{q})$ as the continued fraction representation of the fraction $\frac{p}{q}$, we can write

$$f\left(\frac{p}{q}\right) = \left\lfloor \frac{p}{q} \right\rfloor + f\left(\frac{q}{p \bmod q}\right)$$

Here, we are going to use the `Fraction` class defined in the solution to **Problem 10**, in order to take advantage of the reduced form and sign checks it carries out.

`main (numerator:Integer, denominator:Integer)`

1. Pack `numerator` and `denominator` into a `Fraction` object. Store it as `f`.
2. Call `getContinuedFraction(f)`. Display the returned `String`.
3. **Exit**

`getContinuedFraction (Fraction f)`

1. Unpack `numerator` and `denominator` from `f`.
2. Call `getContinuedFraction(numerator, denominator)`. Store the returned `String` in the variable `expansion`.
3. Replace the first comma (`,`) in `expansion` with a semicolon (`;`).
4. **Return** `expansion`

`getContinuedFraction (numerator:Integer, denominator:Integer)`

1. **If** `denominator` is `1`, **return** `numerator`.
2. Calculate the integer part of `numerator / denominator`. Store it in `x`.
3. Call `getContinuedFraction(denominator, numerator % denominator)`. Store the result in `y`.
4. **Return** `x + y`

Source Code

```
1  public class ContinuedFraction {
2        public static void main (String[] args) {
3              try {
4                     /* Parse command line arguments as the numerator and
                          denominator
5                        of the fraction */
6                     int numerator = Integer.parseInt(args[0]);
7                     int denominator = Integer.parseInt(args[1]);
8                     System.out.println(getContinuedFraction(new
                          Fraction(numerator, denominator)));
9              } catch (ArithmeticException e) {
10                    System.out.println("Invalid fraction - division by zero!");
11             } catch (Exception e) {
12                    System.out.println("Enter 2 arguments! ([numerator]
                          [denominator])");
13             }
14       }
15
16       /* Return the String representation of the continued fraction */
17       public static String getContinuedFraction (Fraction f) {
```

```
18              String expansion = "[" + getContinuedFraction(f.getNumerator(),
                    f.getDenominator());
19              /* By convention, the first comma is replaced with a semicolon */
20              return expansion.replaceFirst(",", ";");
21      }
22
23      /* Recursively calculate the continued fraction representation */
24      public static String getContinuedFraction (int numerator, int denominator) {
25              /* Base case : the fraction is now irreducible */
26              if (denominator == 1)
27                      return numerator + "]";
28              /* Pull out the integer part, invert the fraction and recurse */
29              return (numerator / denominator) + ", " +
                    getContinuedFraction(denominator, numerator % denominator);
30      }
31  }
```

Variable Description

ContinuedFraction::main(String[])		
int	numerator	Stores the numerator of the fraction to evaluate
int	denominator	Stores the denominator of the fraction to evaluate

ContinuedFraction::getContinuedFraction(Fraction)		
Fraction	f	Stores the fraction to evaluate
String	expansion	Stores the continued fraction representation of f

"Intelligence is the ability to avoid doing work, yet getting the work done."

— **Linus Torvalds**

Problem 12 The *binomial coefficient* [9] of two integers $n \geq k \geq 0$ is defined as follows.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Here, $n!$ is the *factorial* of n, defined as follows.

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-2) \times (n-1) \times n$$

Compute the binomial coefficient for two given integers.

Solution Note that we can rewrite the definition of the binomial by cancelling out common factors from the factorials.

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-(k-1))}{k(k-1)(k-2)\cdots 1}$$

Now that we have this definition, it is easy to see that we can separate the term $\frac{n}{k}$ and leave behind a smaller binomial coefficient. Thus, we arrive at the recursive formula

$$\binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1}$$

Coupled with the observation that $\binom{n}{0} = 1$, we can solve this problem recursively.

We can introduce a small optimisation by observing that $\binom{n}{k} = \binom{n}{n-k}$. Thus, for $k > \frac{n}{2}$, we can replace k with $n-k$ to reduce the number of recursive calls.

[9] They are given this name as they describe the coefficients of the expansion of powers of a binomial, according to the *binomial theorem*.

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

`main (n:Integer, k:Integer)`

1. Call and display `binomial(n, k)`.
2. **Exit**

`binomial (n:Integer, k:Integer)`

1. **If** `k` is zero, **return** 1.
2. **If** `k` exceeds half of `n`, call `binomial(n, n - k)`.
3. **Return** `binomial(n - 1, k - 1) * (n / k)`.

Source Code

```
1  public class Binomial {
2  	public static void main (String[] args) {
3  		try {
4  			/* Parse the command line arguments as the terms in the
   			    binomial coefficient */
5  			long n = Long.parseLong(args[0]);
6  			long k = Long.parseLong(args[1]);
7  			System.out.println(binomial(n, k));
8  
9  		} catch (NumberFormatException | IndexOutOfBoundsException e) {
10 			System.out.println("Enter 2 arguments! ([+integer]
   			    [+integer])");
11 		} catch (Exception e) {
12 			System.out.println("Invalid 'k'! (0 <= k <= n)");
13 		}
14 	}
15 
16 	/* Recursively calculate the binomial coefficient n choose k */
17 	public static long binomial (long n, long k) throws Exception {
18 		/* Invalid case */
19 		if (k > n)
20 			throw new Exception();
21 		/* Base case : n choose 0 is 1 */
22 		if (k == 0)
23 			return 1;
24 		/* Optimisation to reduce the number of recursive steps by reflecting
25 		   k along the middle of n */
26 		if (k > (n / 2))
27 			return binomial(n, n - k);
28 		/* Recurse by unfolding the multiplication */
29 		return (n * binomial(n - 1, k - 1) / k);
30 	}
31 }
```

Variable Description

`Binomial::main(String[])`		
`long`	`n, k`	The arguments for calculating the binomial coefficient

`Binomial::binomial(long, long)`		
`long`	`n, k`	The arguments for calculating the binomial coefficient

"If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is."

— **John von Neumann**

Problem 13 Palindromes can be generated in many ways. One of them involves picking a number, reversing the order of its digits and adding the result to the original. For example, we have

$$135 + 531 \;=\; 666$$

Not all numbers will yield a palindrome after one step. Instead, we can repeat the above process, using the sum obtained as as the new number to reverse.

$$\begin{aligned} 963 + 369 &= 1332 \\ 1332 + 2331 &= 3663 \end{aligned}$$

This process is often called the *196-algorithm*. Some numbers seem never to yield a palindrome even after millions of iterations. These are called *Lychrel numbers*. The smallest of these in base 10 is conjectured to be the number 196, although none have been mathematically proven to exist.

Generate the steps and final palindrome of the *196-algorithm*, given a natural number as a *seed* [10].

Solution This problem can be solved without much complication. We can either create a loop, or use *tail recursion* [11] to roll up the process. The only problem here is that the numbers involved grow very large, very fast. Thus, care must be taken while dealing with such cases. Here, a library method for addition has been used to identify integer overflow.

[10] A *seed* is an initial number, from which subsequent numbers are generated.

[11] *Tail recursion* involves the use of *tail calls*. These are simply recursive function calls which appear as the last statement of the function body. Most programming languages can optimize tail recursion internally into a simple loop, thus avoiding the addition of stack frames on each recursive call.

`main (number:Integer)`

1. Call `generatePalindrome(number, 0)`.
2. **Exit**

`generatePalindrome (n:Integer, step:Integer)`

1. Reverse the digits in `n`. Store the result in `r`.
2. **If** `n` is equal to `r`:
   (a) Display `n` as a palindrome, along with `step`.
   (b) **Return**
3. Add `n` and `r`. Store the sum in the variable `sum`.
4. Call `generatePalindrome(sum, step + 1)`

Source Code

```
class PalindromeGenerator {
	public static void main (String[] args) {
		/* Parse the first command line argument as the seed */
		long n = Long.parseLong(args[0]);
		generatePalindrome(n, 0);
	}

	public static void generatePalindrome (long n, int step) {
		long r = reverse(n);
		if (n == r) {
			/* Base case : palindrome reached */
			System.out.printf("%d is a palindrome (%d step%s)%n", n, step,
			    ((step == 1)? "" : "s"));
		} else {
			try {
				/* Use a library method to add. This will throw an
				   Exception in case of overflow, which would have
				   otherwise been ignored */
				long sum = Math.addExact(n, r);
				System.out.printf("%d + %d = %d%n", n, r, sum);
				/* Recurse via tail recursion, simply incrementing the
				    step value */
				generatePalindrome(sum, step + 1);
			} catch (ArithmeticException e) {
				/* Stop if the numbers become too big */
				System.out.printf("Long Overflow - Sum exceeded maximum
				    size at step %d%n", step);
			}
```

```
26              }
27      }
28
29      /* Reverse the integer supplied */
30      public static long reverse (long n) {
31              long r = 0;
32              while (n > 0) {
33                      /* Pull out the last digit and accumulate it on another
                            variable */
34                      r = (r * 10) + (n % 10);
35                      n /= 10;
36              }
37              return r;
38      }
39  }
```

Variable Description

PalindromeGenerator::main(String[])		
long	n	Stores the *seed* for the palindrome generation

PalindromeGenerator::generatePalindrome(long, int)		
long	n	Stores the current number to generate a palindrome from
long	r	Stores the reverse of n
int	step	Stores the step of the generation currently executing
long	sum	Stores the sum of n and r

PalindromeGenerator::reverse(long)		
long	r	Stores the reverse of n

"Over thinking leads to problems that don't even exist in the first place."

— **Jayson Engay**

Problem 14 Compute the *prime factorization* of a given natural number.

Solution This solution is meant to showcase the drawbacks of using *recursion* in some problems.

Let $f(n)$ denote the expansion of the *prime factorization* of the natural number n. We *could* observe that if we can find naturals p and q such that $n = pq$, we can write

$$f(pq) \;=\; f(p) + f(q)$$

Using this, we can wrap up the iteration over the naturals into a recursive function.

The problem with this approach is that for moderately large numbers, the number of nested calls grows rapidly. For large enough numbers, the default memory allocated for the *call stack* by the *Java Virtual Machine* falls woefully short. As a result, it becomes necessary to manually set the size of the *thread stack size* by passing the `-Xss<size>` option to the *JVM* during program execution.

`main (number:Integer)`

1. Call and display `factorize(number, 2)`.
2. **Exit**

`factorize (n:Integer, next:Integer)`

1. **If** `n` is one, **return** an empty `String`.
2. **If** `next` exceeds, or is equal to, `n`, **return** `next`.
3. **If** `next` divides `n`:
   (a) Append `next` to the `String` returned by the call `factorize(n / next, next)`.
   (b) **Return** the above value.
4. **Return** `factorize(n, next + 1)`

Source Code

```
1 public class Factorize {
2       public static void main (String[] args) {
3             /* Parse the first command line argument as the number to factorize */
4             int number = Integer.parseInt(args[0]);
5             /* Start from 2 */
```

```
6                 System.out.println(factorize(number, 2));
7         }
8
9         /* Return the String representation of the prime factorization of an integer
              */
10        public static String factorize (int n, int next) {
11                /* Base case 1 : nothing to factorize */
12                if (n == 1)
13                        return "";
14                /* Base case 2 : reached a prime */
15                if (next >= n)
16                        return next + "";
17                /* Check for a factor */
18                if ((n % next) == 0)
19                        return next + " " + factorize(n / next, next);
20                /* Recurse by incrementing the next 'factor' to check */
21                return factorize(n, next + 1);
22        }
23  }
```

Variable Description

Factorize::main(String[])		
int	number	Stores the number to be factorized

Factorize::factorize(String[])		
int	n	Stores the current number to be factorized
int	next	Stores the next number to check for divisibility

"Meaning lies as much
in the mind of the reader
as in the Haiku."

— **Douglas Hofstadter**

Problem 15 A *codebook* is a document which stores a *lookup table* for coding and decoding text – each word has a different word, phrase or string to replace it. Design a system which, when given a *codebook* written in plaintext, translates a given sentence into its encoded form.

Solution Solving this problem requires careful reading of the supplied codebook. Here, the following format is assumed.

```
word          codeword
next_word     other_codeword
.             .
.             .
```

Thus, this data can be transformed into an *array*, which can then be searched for strings appearing in the supplied input.

`main (codebook:String)`

1. Create a `CodeSubstituter` object, pass it the filename `codebook`, and assign it to `cs`.
2. Get a line of user input. Store it in `sentence`.
3. Split `sentence` along whitespace into the `String` array `words`.
4. For each `word` in `words`:
   (a) Call `cs->getEncodedText(word)`. Store the result in `encodedText`.
   (b) Display `encodedText`.
5. **Exit**

`CodeSubstituter (codebook:String)`

1. Open the file pointed to by `codebook`. Start from the beginning in read mode.
2. On the first pass through `codebook`, count the number of lines and store the result in `numberOfLines`.
3. Close, and reopen `codebook`. Start at the beginning.
4. Initialize a `2` column `String` array, with `numberOfLines` as the number of rows. Assign it to `wordMap`.

5. Start reading `codebook` again. For each line, stored in `line` and each row in `wordMap` :
   (a) Split `line` along whitespace.
   (b) Store the first half in the first column of `wordMap`, and the second half in the second column of the same.
6. Close the file `codebook`.
7. **Define** the function `CodeSubstituter::getEncodedText(word)` and **return** the resultant object.

`CodeSubstituter::getEncodedText (word:String)`

1. For each row in `wordMap`:
   (a) If the first column entry matches `word`, **return** the second column entry.
2. **Return** `word`

Source Code

```
1  import java.io.IOException;
2  import java.io.FileReader;
3  import java.io.BufferedReader;
4
5  public class CodeSubstituter {
6         protected String filename;
7
8         protected int numberOfLines;
9         protected String[][] wordMap;
10
11        /* Create a codebook from a supplied file */
12        public CodeSubstituter (String filename) throws IOException {
13               this.filename = filename;
14               countNumberOfLines();
15               initWordMap();
16        }
17
18        /* Calculate the number of lines to store on the first pass */
19        private void countNumberOfLines () throws IOException {
20               FileReader fileReader = new FileReader(filename);
21               BufferedReader bufferedReader = new BufferedReader(fileReader);
22
23               numberOfLines = 0;
24               /* Keep incrementing the accumulator while lines are available */
25               while (bufferedReader.readLine() != null)
26                      numberOfLines++;
27
```

```
28              bufferedReader.close();
29              fileReader.close();
30      }
31
32      /* Initialize the map/dictionary by reading the file on the second pass */
33      private void initWordMap () throws IOException {
34              wordMap = new String[numberOfLines][2];
35
36              FileReader fileReader = new FileReader(filename);
37              BufferedReader bufferedReader = new BufferedReader(fileReader);
38
39              for (int i = 0; i < numberOfLines; i++) {
40                      /* Split a line along whitespace */
41                      String[] words = bufferedReader.readLine().split("\\s+");
42                      if (words.length >= 2) {
43                              wordMap[i][0] = words[0];
44                              wordMap[i][1] = words[1];
45                      } else {
46                              /* Ignore empty lines */
47                              wordMap[i][0] = wordMap[i][1] = "";
48                      }
49              }
50
51              bufferedReader.close();
52              fileReader.close();
53      }
54
55      /* Returns the codeword, given a plain word */
56      public String getEncodedText (String word) {
57              /* Iterate through all entries */
58              for (int i = 0; i < numberOfLines; i++) {
59                      if (wordMap[i][0].equalsIgnoreCase(word)) {
60                              return wordMap[i][1];
61                      }
62              }
63              /* Reflect the original back if not found in the codebook */
64              return word;
65      }
66  }
```

```java
import java.util.Scanner;
import java.io.IOException;
import java.io.FileNotFoundException;

public class TextEncoder {
	public static void main (String[] args) throws Exception {
		try {
			/* Parse the first command line argument as the path to the
			    codebook */
			CodeSubstituter cs = new CodeSubstituter(args[0]);

			/* Get a sentence to encode, and extract the individual words
			    */
			System.out.print("Enter a sentence to encode : ");
			String sentence = (new Scanner(System.in)).nextLine();
			String[] words = sentence.split("\\s+");

			System.out.print("Encoded sentence     : ");
			/* Iterate through each word, replacing it with the codeword
			    in the codebook */
			for (int i = 0; i < words.length; i++) {
				String encodedText =
				    cs.getEncodedText(words[i].toLowerCase().replaceAll("[^a-z]",
				    ""));
				System.out.print(encodedText + " ");
			}
			System.out.println();
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("Enter 1 argument ([codebook_filename])");
		} catch (FileNotFoundException e) {
			System.out.println("Codebook not found! Enter a valid
			    filename.");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Variable Description

CodeSubstituter		
String	filename	Stores the path of the file containing the codebook
int	numberOfLines	Stores the number of lines in the file filename
String[][]	wordMap	A table of plain words and their corresponding codewords

CodeSubstituter::countNumberOfLines()		
FileReader	fileReader	An object for reading character based files
Buffered Reader	bufferedReader	An object for buffering character streams

CodeSubstituter::initWordMap()		
FileReader	fileReader	An object for reading character based files
Buffered Reader	bufferedReader	An object for buffering character streams
String[]	words	Temporarily stores the parts of a line in the codebook

TextEncoder::main(String[])		
Code Substituter	cs	An object for accessing a codebook
String	sentence	Stores a line of user input to be encoded
String[]	words	Stores the list of words in sentence

"Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law."

— Douglas Hofstadter

Problem 16 Analyse the frequency of each letter in the English alphabet appearing in a given file. Store the results in a different file.

Solution All that has to be done here is reading the contents of a file, counting the occurrences of each character, then tabulating the results before writing them to another file. Here, the characters have also been sorted based on their frequencies.

`main (fromFile:String, toFile:String)`

1. Create a `CharacterCounter` object, pass it `fromFile`, and assign it to `cc`.
2. Call `cc->writeReportToFile(toFile)`.
3. **Exit**

`CharacterCounter (fromFile:String)`

1. Read all the lines from the file `fromFile` and store the resultant `String` in `fileData`.
2. Initialize a `26` row `Character` array `letters`, as well as a `26` row `Integer` array `letterCount`.
3. For each letter `c` $\in \{a, b, \ldots, z\}$:
   (a) Store `c` in an empty row in `letters`.
   (b) Count the number of occurrences of `c` in `fileData`. Store the result in the corresponding row in `letterCount`.
   (c) Move to a new row in `letters` and `letterCount`.
4. Store the sum of all entries in `letterCount` in the variable `totalLetters`.
5. Sort the entries in `letters` and `letterCount`, in descending order of the entries in `letterCount` using *bubble sort*.
6. **Define** the function `CharacterCounter::writeReportToFile(toFile)` and **return** the resultant object.

`CharacterCounter::writeReportToFile (toFile:String)`

1. Open the file pointed to by `toFile`. Start from the beginning in write mode.
2. Write all entries in `letters` and `letterCount`, formatted to include the ratio of the entry in `letterCount` to `totalLetters`.

3. Write `totalLetters` to `toFile`, along with any entry in `letters` whose corresponding entry in `letterCount` is zero.
4. Close the file `toFile`.
5. **Return**

Source Code

```
1  import java.io.IOException;
2  import java.io.FileReader;
3  import java.io.FileWriter;
4  import java.io.BufferedReader;
5  import java.io.BufferedWriter;
6  import java.io.PrintWriter;
7
8  public class CharacterCounter {
9         protected String filename;
10
11        protected String fileData;
12        protected char[] letters;
13        protected int[] letterCount;
14        protected int totalLetters;
15
16        /* Create a table of letter counts in a given file */
17        public CharacterCounter (String filename) throws IOException {
18               this.filename = filename;
19               this.fileData = "";
20               this.letterCount = new int[26];
21               this.letters = new char[26];
22               this.totalLetters = 0;
23               getFileData();
24               countAllLetters();
25               sortLetters();
26        }
27
28        /* Read all lines in the file and store them in a String */
29        private void getFileData () throws IOException {
30               FileReader fileReader = new FileReader(filename);
31               BufferedReader bufferedReader = new BufferedReader(fileReader);
32
33               String line = "";
34               while ((line = bufferedReader.readLine()) != null)
35                      fileData += line.toLowerCase();
36
37               bufferedReader.close();
38               fileReader.close();
```

```
39          }
40
41          /* Return the number of occurrences of a character in the file */
42          public int getCountOf (char c) {
43                  int count = 0;
44                  for (int i = 0; i < fileData.length(); i++) {
45                          if (fileData.charAt(i) == c) {
46                                  count++;
47                          }
48                  }
49                  return count;
50          }
51
52          /* Compile the counts of all letters in the file */
53          public void countAllLetters () {
54                  for (char c = 'a'; c <= 'z'; c++) {
55                          letters[c - 'a'] = c;
56                          letterCount[c - 'a'] = getCountOf(c);
57                          totalLetters += letterCount[c - 'a'];
58                  }
59          }
60
61          /* Sort the entries by frequency (bubble sort) */
62          private void sortLetters () {
63                  for (int right = 26; right > 0; right--)
64                          for (int i = 1; i < right; i++)
65                                  if (letterCount[i] > letterCount[i-1])
66                                          swap(i, i-1);
67          }
68
69          /* Utility swapping method */
70          private void swap (int i, int j) {
71                  char tmpChar = letters[i];
72                  int tmpCount = letterCount[i];
73                  letters[i] = letters[i-1];
74                  letterCount[i] = letterCount[i-1];
75                  letters[i-1] = tmpChar;
76                  letterCount[i-1] = tmpCount;
77          }
78
79          /* Create and write the final report to a file */
80          public void writeReportToFile (String toFilename) throws IOException {
81                  FileWriter fileWriter = new FileWriter(toFilename);
82                  BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
83                  PrintWriter printWriter = new PrintWriter(bufferedWriter);
84
```

```
85              /* Make sure the frequencies all fit, aligned in the same column */
86              int l = (totalLetters + "").length();
87              String unusedLetters = "";
88              for (int i = 0; i < 26; i++) {
89                      /* Show the letter, frequency and percentage out of the total
                           */
90                      char c = letters[i];
91                      int count = letterCount[i];
92                      double percent = (count * 100.0) / totalLetters;
93                      if (count > 0) {
94                              printWriter.printf("%c : %5.2f%% (%" + l + "d) %n",
95                                              c, percent, count);
96                      } else {
97                              /* Separate unused letters */
98                              unusedLetters += c + " ";
99                      }
100             }
101             printWriter.printf("Total letters : %d%n", totalLetters);
102             if (unusedLetters.length() == 0)
103                     unusedLetters = "(none)";
104             printWriter.printf("Unused letters : %s%n", unusedLetters);
105
106             printWriter.close();
107             bufferedWriter.close();
108             fileWriter.close();
109     }
110 }
```

```
1  import java.io.IOException;
2  import java.io.FileNotFoundException;
3
4  public class AnalyseCharacterFrequency {
5          public static void main (String[] args) {
6                  try {
7                          /* Parse the commnd line arguments as the file to analyse and
                               the
8                            file to pipe the results into */
9                          String fromFile = args[0];
10                         String toFile = args[1];
11
12                         /* Create and write the report */
13                         CharacterCounter cc = new CharacterCounter(fromFile);
14                         cc.writeReportToFile(toFile);
15                 } catch (ArrayIndexOutOfBoundsException e) {
16                         System.out.println("Enter 2 arguments! ([filename_from]
```

```
                        [filename_to])");
17              } catch (FileNotFoundException e) {
18                      System.out.println("Enter a valid filename!");
19              } catch (IOException e) {
20                      e.printStackTrace();
21              }
22      }
23  }
```

Variable Description

CharacterCounter		
String	filename	Stores the path of the file to analyse
String	fileData	Stores all character data from the file
char[]	letters	The list of all letters, in order of frequency
int[]	letterCount	The frequencies of each corresponding letter in letters
int	totalLetters	Stores the total number of letters in fileData
CharacterCounter::getFileData()		
FileReader	fileReader	An object for reading character based files
BufferedReader	bufferedReader	An object for buffering character streams
String	line	Stores a line of text in the file
CharacterCounter::getCountOf(char)		
char	c	The character whose frequency is to be found in fileData
int	count	The frequency of c in fileData
CharacterCounter::countAllLetters()		
char	c	The character whose frequency is to be found
CharacterCounter::sortLetters()		
int	right, i	Counter variables
CharacterCounter::swap(int, int)		
int	i, j	Indices of letters and letterCount whose entries are to be swapped
CharacterCounter::writeReportToFile(String)		
String	toFilename	Stores the path of the file to write the report to
FileWriter	fileWriter	An object for writing character based files

`BufferedWriter`	`bufferedWriter`	An object for buffering character streams being written to a file
`PrintWriter`	`printWriter`	An object for writing data to an output stream
`int`	`l`	Stores the number of digits in `totalLetters`
`String`	`unusedLetters`	Stores the list of letters not present in `fileData`
`char`	`c`	Stores the current character being written
`int`	`count`	Stores the frequency of `c`
`double`	`percent`	Stores the percentage of `count` out of `totalLetters`

`AnalyseCharacterFrequency::main(String[])`		
`String`	`fromFile`	Stores the path of the file to analyse
`String`	`toFile`	Stores the path of the file to write the report to
`CharacterCounter`	`cc`	An object for analysing the frequencies of letters in files

"If Java had true garbage collection, most programs would delete themselves upon execution."

— **Robert Sewell**

Problem 17 The classical *Möbius function* $\mu(n)$ is an important function in number theory and combinatorics. For positive integers n, $\mu(n)$ is defined as the sum of the primitive n^{th} roots of unity. It attains the following values.

$\mu(1) = +1$
$\mu(n) = -1$ if n is a square-free positive integer with an odd number of prime factors.
$\mu(n) = \ \ 0$ if n has a squared prime factor.
$\mu(n) = +1$ if n is a square-free positive integer with an even number of prime factors.

Compute the $\mu(n)$ for positive integers n within a specified range.

Solution For any given $n \in \mathbb{N}$, all we have to do is search for factors by trial-division, and find their multiplicity. If this is greater than 1, we can stop here since we have found squared prime factors. Otherwise, we can reduce the problem by dividing out these factors from n and repeating. By trying factors in ascending order and then discarding them from n, we are guaranteed to hit only prime factors, and can thus skip primality checks.

`main (lo:Integer, hi:Integer)`

1. Assert that the integers in the range `[lo, hi)` are all positive.
2. For each `i` $\in \{$`lo`$, $`lo`$ + 1, \ldots,$ `hi`$ - 1\}$:
   (a) Call and display `mobius(i)`.
3. **Exit**

`mobius (n:Integer)`

1. If `n` is one, **return** `1`.
2. Initialize an integer variable `mob` to one.
3. For `i` $\in \{2, 3, \ldots, n\}$:
   (a) Initialize an integer `multiplicity` to zero.
   (b) While `i` divides `n`, assign `n / i` to `n` and increment `multiplicity`.
   (c) If `multiplicity` is one, flip the sign of `mob`.
   (d) If `multiplicity` is greater than one, **return** `0`.
4. **Return** `mob`

Source Code

```
public class Mobius {
	/* Elements of a basic graph */
	public static final String[] graph =
	{"*      ",
	 "   *   ",
	 "      *"};
	public static void main (String[] args) {
		try {
			/* Parse the first command line argument as the lower limit */
			int lo = Integer.parseInt(args[0]);
			/* Parse the second commmand line argument as the upper limit */
			int hi = Integer.parseInt(args[1]);
			/* Incorrct input */
			if (lo < 1 || hi <= lo)
				throw new NumberFormatException();
			for (int i = lo; i < hi; i++) {
				int m = mobius(i);
				System.out.printf(" (%d)\t\t = %2d%24s\n", i, m, graph[m + 1]);
			}
		} catch (NumberFormatException | IndexOutOfBoundsException e) {
			/* Handle missing or incorrectly formatted arguments */
			System.out.println("Enter 2 arguments (lower_limit[integer, >0], upper_limit[integer, >lower_limit])!");
		}
	}

	public static int mobius (int n) {
		/* Ignore negative numbers */
		if (n < 1)
			return 0;
		/* Trivial case */
		if (n == 1)
			return 1;
		/* Start with +1 */
		int mob = 1;
		for (int i = 2; i <= n; i++) {
			int multiplicity = 0;
			/* Count the number of times (i) appears */
			while ((n % i) == 0) {
				/* Reduce 'n' */
				n /= i;
				multiplicity++;
```

```
42                      }
43                      if (multiplicity == 1) {
44                              /* Flip the sign */
45                              mob = -mob;
46                      } else if (multiplicity > 1) {
47                              /* Squared factor found */
48                              return 0;
49                      }
50              }
51              return mob;
52      }
53  }
```

Variable Description

Mobius::main(String[])		
int	lo	Lower bound of integers to evalute
int	hi	Upper bound of integers to evalute
int	i	Counter variable, stores the integer to be evaluated

Mobius::mobius(int)		
int	n	The number where the mobius function is to be evaluated
int	mob	Sign of the value of the mobius function
int	i	Counter variable, stores the current factor to be tested
int	multiplicity	The power of i in the factorisation of n

"The mathematics is not there till we put it there."

— **Arthur Eddington**

Problem 18 A *set* is a collection of distinct objects. Implement a simple model of *sets*, capable of holding *integers*.

Solution This implementation uses *arrays* as the framework for storing elements. The set is sorted during insertion of elements, allowing for fast *binary searching*.

`Set (maxSize:Integer)`

1. Copy `maxSize` into the object data.
2. Initialize an array of integers `elements`, with length `maxSize`.
3. Initialize an integer `top` to `-1`.
4. **Define** the following functions:
   (a) `Set::updateMaxSize(newMaxSize)`
   (b) `Set::contains(n)`
   (c) `Set::add(n)`
   (d) `Set::remove(n)`
   (e) `Set::indexOfEqualOrGreater(n)`
5. **Return** the resultant object.

`Set::updateMaxSize (newMaxSize:Integer)`

1. Initialize an array of integers `temp`, with length `newMaxSize`.
2. Set `maxSize` to `newMaxSize`.
3. If the new size cannot accomodate the present elements of the set, discard them by setting `top` to `maxSize - 1`.
4. Copy all integers from indices `0` to `top` from `elements` to `temp`.
5. Set `elements` to `temp`.

`Set::contains (n:Integer)`

1. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
2. If `i` is a valid index within the set, and the element at that index is equal to `n`, **return** `true`, otherwise **return** `false`.

`Set::add (n:Integer)`

1. Assert that the set is large enough to hold the new element.
2. If the set already contains `n`, **return** `false`.

3. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
4. Shift all integers in `elements` from indices `i` to `top` one place to the right.
5. Set `elements[i]` to `n`.
6. **Return** `true`

`Set::remove (n:Integer)`

1. Assert that the set is not empty.
2. If the set does not already contain `n`, **return** `false`.
3. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
4. Shift all integers in `elemets` from indices `i + 1` to `top` one place to the left.
5. **Return** `true`

`Set::indexOfEqualOrGreater (n:Integer)`

1. Initialize an integer `hi` to `top + 1`.
2. Initialize an integer `lo` to `0`;
3. While `lo < hi`:
   (a) Set a temporary integer `mid` to `(lo + hi) / 2`.
   (b) If `n` is less than the element at mid, set `hi` to `mid`.
   (c) If `n` is greater than the element at mid, set `lo` to `mid + 1`.
   (d) If `n` is equal to the element at mid, **return** `mid`.
4. **Return** `hi`

`union (a:Set, b:Set)`

1. Create a new `Set`, capable of holding the combined elements of `a` and `b`. Call it `r`.
2. For each element `n` in `a`, call `r->add(n)`.
3. For each element `n` in `b`, call `r->add(n)`.
4. **Return** `r`.

`intersection (a:Set, b:Set)`

1. Create a new `Set`, with its `maxSize` equal to either of the sizes of `a` or `b`. Call it `r`.
2. For each element `n` in `a`, also contained in `n`, call `r->add(n)`.
3. **Return** `r`

`difference (a:Set, b:Set)`

1. Create a new `Set`, with its `maxSize` equal to either of the sizes of `a` or `b`. Call it `r`.

2. For each element `n` in `a`, not contained in `n`, call `r->add(n)`.
3. **Return** `r`

Source Code

```
1  import java.util.Iterator;
2
3  public class Set implements Iterable<Integer> {
4         protected int maxSize;
5
6         /* Simple list setup */
7         protected int[] elements;
8         protected int top;
9
10        /* Let the maximum capacity be specified during instantiation */
11        public Set (int maxSize) {
12               this.maxSize = maxSize;
13               this.elements = new int[maxSize];
14               this.top = -1;
15        }
16
17        /* Returns the number of elements in the set */
18        public int getSize () {
19               return top + 1;
20        }
21
22        /* Returns the maximum capacity of the set */
23        public int getMaxSize () {
24               return maxSize;
25        }
26
27        /* Expands or contracts the set as necessary, discards elements if
28          they cannot be accomodated */
29        public void updateMaxSize (int newMaxSize) {
30               int[] temp = new int[newMaxSize];
31               this.maxSize = newMaxSize;
32               /* Make sure that the top index isn't out of bounds */
33               this.top = Math.min(top, newMaxSize - 1);
34               /* Copy data to the new list */
35               for (int i = 0; i <= top; i++)
36                      temp[i] = elements[i];
37               this.elements = temp;
38        }
39
40        /* Checks whether an element is present in the set */
```

```
41          public boolean contains (int n) {
42                  int i = indexOfEqualOrGreater(n);
43                  return ((i >= 0) && (i <= top) && (elements[i] == n));
44          }
45
46          /* Checks whether the set is empty */
47          public boolean isEmpty () {
48                  return top < 0;
49          }
50
51          /* Clears all elements from the set */
52          public void clear () {
53                  /* Only the top index has to be updated, since values byond it
54                     cannot be accessed */
55                  this.top = -1;
56          }
57
58          /* Adds an element to the set. Returns 'false' if it is already
59             present, or there isn't enough space. */
60          public boolean add (int n) {
61                  if (getSize() >= getMaxSize())
62                          return false;
63                  /* Find the breakpoint to shift elements */
64                  int i = indexOfEqualOrGreater(n);
65                  if ((i >= 0) && (i <= top) && (elements[i] == n))
66                          return false;
67                  /* Shift elements greater than 'n' to make room for it */
68                  for (int j = top; j >= i; j--)
69                          elements[j + 1] = elements[j];
70                  elements[i] = n;
71                  top++;
72                  return true;
73          }
74
75          /* Removes an element from the set. Returns 'false' if it isn't
76             already present. */
77          public boolean remove (int n) {
78                  if (isEmpty())
79                          return false;
80                  /* Find the location of the element */
81                  int i = indexOfEqualOrGreater(n);
82                  if ((i < 0) || (i > top) || (elements[i] != n))
83                          return false;
84                  /* Shift elements into the desired element, erasing it */
85                  for (int j = i; j < top; j++)
86                          elements[j] = elements[j + 1];
```

```
87              top--;
88              return true;
89      }
90
91      /* Returns the union of two sets */
92      public static Set union (Set a, Set b) {
93              Set r = new Set(a.getSize() + b.getSize());
94              /* The 'add' methods take care of duplicates */
95              for (int n : a)
96                      r.add(n);
97              for (int n : b)
98                      r.add(n);
99              return r;
100     }
101
102     /* Returns the intersection of two sets */
103     public static Set intersection (Set a, Set b) {
104             Set r = new Set(a.getSize());
105             for (int n : a)
106                     if (b.contains(n))
107                             r.add(n);
108             return r;
109     }
110
111     /* Returns the difference of two sets */
112     public static Set difference (Set a, Set b) {
113             Set r = new Set(a.getSize());
114             for (int n : a)
115                     if (!b.contains(n))
116                             r.add(n);
117             return r;
118     }
119
120     /* Finds the index of the element equal to or greater than
121        the desired element via binary search */
122     private int indexOfEqualOrGreater (int n) {
123             int hi = top + 1;
124             int lo = 0;
125             while (lo < hi) {
126                     int mid = (lo + hi) / 2;
127                     if (n < elements[mid])
128                             hi = mid;
129                     else if (n > elements[mid])
130                             lo = mid + 1;
131                     else
132                             return mid;
```

```
133                 }
134                 return hi;
135         }
136
137         /* Format the set elements as a list */
138         @Override
139         public String toString () {
140                 if (getSize() == 0)
141                         return "[]";
142                 String s = "";
143                 for (Integer n : this)
144                         s += n + " ";
145                 return "[" + String.join(", ", s.split("\\s+")) + "]";
146         }
147
148         /* Allow 'Set' to be iterable, providing easy access to elements
149            without indexing */
150         @Override
151         public Iterator<Integer> iterator () {
152                 return new Iterator<Integer>() {
153                         private int currentIndex = 0;
154
155                         @Override
156                         public boolean hasNext () {
157                                 return currentIndex <= top;
158                         }
159
160                         @Override
161                         public Integer next () {
162                                 return elements[currentIndex++];
163                         }
164
165                         @Override
166                         public void remove () {
167                                 throw new UnsupportedOperationException();
168                         }
169                 };
170         }
171 }
```

```
1  public class SetDemo {
2          public static void main (String[] args) {
3                  /* Create 3 sets with random elements */
4                  Set a = new Set(10);
5                  Set b = new Set(10);
```

```
        Set c = new Set(10);
        for (int i = 0; i < 10; i++)
               a.add((int) (Math.random() * 10));
        for (int i = 0; i < 10; i++)
               b.add((int) (Math.random() * 10));
        for (int i = 0; i < 10; i++)
               c.add((int) (Math.random() * 10));

        /* Demonstrate simple output formatting */
        System.out.printf("A [%2d] = %s\n", a.getSize(), a);
        System.out.printf("B [%2d] = %s\n", b.getSize(), b);
        System.out.printf("C [%2d] = %s\n", c.getSize(), c);
        System.out.println();

        /* Demonstrate set operations */
        System.out.printf("A union B [%2d] = %s\n",
                     Set.union(a, b).getSize(),
                     Set.union(a, b));
        System.out.printf("B union C [%2d] = %s\n",
                     Set.union(b, c).getSize(),
                     Set.union(b, c));
        System.out.printf("C union A [%2d] = %s\n",
                     Set.union(c, a).getSize(),
                     Set.union(c, a));
        System.out.printf("A union B union C [%2d] = %s\n",
                     Set.union(Set.union(a, b), c).getSize(),
                     Set.union(Set.union(a, b), c));
        System.out.println();
        System.out.printf("A intersection B [%2d] = %s\n",
                     Set.intersection(a, b).getSize(),
                     Set.intersection(a, b));
        System.out.printf("B intersection C [%2d] = %s\n",
                     Set.intersection(b, c).getSize(),
                     Set.intersection(b, c));
        System.out.printf("C intersection A [%2d] = %s\n",
                     Set.intersection(c, a).getSize(),
                     Set.intersection(c, a));
        System.out.printf("A intersection B intersection C [%2d] = %s\n",
                     Set.intersection(Set.intersection(a, b), c).getSize(),
                     Set.intersection(Set.intersection(a, b), c));
        System.out.println();
        System.out.printf("A - B [%2d] = %s\n",
                     Set.difference(a, b).getSize(),
                     Set.difference(a, b));
        System.out.printf("B - C [%2d] = %s\n",
                     Set.difference(b, c).getSize(),
```

```
52                              Set.difference(b, c));
53                System.out.printf("C - A [%2d] = %s\n",
54                              Set.difference(c, a).getSize(),
55                              Set.difference(c, a));
56        }
57  }
```

Variable Description

Set		
int	maxSize	The maximum number of elements the set can hold
int[]	elements	The collection of elements contained in the set
int	top	The index of the topmost element in elements

Set::Set(int)		
int	maxSize	The maximum number of elements the set can hold

Set::updateMaxSize(int)		
int	newMaxSize	The maximum number of elements the set is to hold
int[]	temp	The new copy of elements with the updated size

Set::add(int)		
int	n	The element to be added to the set
int	i	The index of the breakpoint from which elements have to be shifted

Set::remove(int)		
int	n	The element to be removed from the set
int	i	The index of the breakpoint from which elements have to be shifted

Set::indexOfEqualOrGreater(int)		
int	n	The element to be searched for
int	hi	The upper index where n can be
int	lo	The lower index where n can be
int	mid	The midpoint of hi and lo

"Mathematics is the art of giving the same name to different things."

— **Henri Poincaré**

Problem 19 A *vector space* is a collection of objects called *vectors*, which may be added together and multiplied (scaled) by *scalars*. One way of implementing a *vector* is to describe the space $\mathbb{R}^n$, i.e. all possible ordered tuples of n real numbers. For example, the vector $(1, 7, 0, 1)$ belongs to the vector space $\mathbb{R}^4$ – it is a four-dimensional vector.

Addition, scalar multiplication, the dot product and the magnitude of vectors is defined as follows. $(a_i, b_i, k \in \mathbb{R})$

$$\begin{aligned}
(a_1, a_2, \ldots, a_n) + (b_1, b_2, \ldots, b_n) &= (a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n) && \text{(Addition)} \\
k\,(a_1, a_2, \ldots, a_n) &= (ka_1, ka_2, \ldots, ka_n) && \text{(Scalar Multiplication)} \\
(a_1, a_2, \ldots, a_n) \cdot (b_1, b_2, \ldots, b_n) &= a_1b_1 + a_2b_2 + \cdots + a_nb_n && \text{(Dot Product)} \\
\|(a_1, a_2, \ldots, a_n)\| &= \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2} && \text{(Magnitude)}
\end{aligned}$$

Implement a simple model of *vectors* as defined above.

Solution

`Vector (components:FloatingPoint[])`

1. Set a constant integer `dimension` to the length of `components`.
2. Copy `components` into the object data as a constant.
3. **Define** the functions:
   (a) `Vector::getComponent(index)`
   (b) `Vector::getAbsoluteValue()`
4. **Return** the resultant object.

`Vector::getComponent (index:Integer)`

1. **Return** `components[index - 1]`

`Vector::getAbsoluteValue ()`

1. Initialize a floating point `abs` to zero.
2. For each `component` in `components`, add `component * component` to `abs`.
3. **Return** the square root of `abs`.

`add (a:Vector, b:Vector)`

1. Assert that `a` and `b` have the same `dimension`.

2. Create an array of floating points `sum`, with length equal to their common `dimension`.
3. For each `i` $\in \{1, 2, \ldots,$ `dimension`$\}$:
   (a) Set `sum[i-1]` to `a->getComponent(i) + b->getComponent(i)`.
4. Create a new `Vector`, pass it `sum` and **return** the resultant object.

`multiplyByScalar (v:Vector, k:FloatingPoint)`

1. Create an array of floating points `t`, with length equal to the `dimension` of `v`.
2. For each `i` $\in \{1, 2, \ldots,$ `dimension`$\}$:
   (a) Set `t[i-1]` to `v->getComponent(i) * k`.
3. Create a new `Vector`, pass it `t` and **return** the resultant object.

`dotProduct (a:Vector, b:Vector)`

1. Assert that `a` and `b` have the same `dimension`.
2. Initialize a floating point `dotProduct` to zero.
3. For each `i` $\in \{1, 2, \ldots,$ `dimension`$\}$:
   (a) Add `a->getComponent(i) * b->getComponent(i)` to `dotProduct` .
4. **Return** `dotProduct`

Source Code

```
1  public class Vector {
2        /* Vector data is immutable */
3        protected final int dimension;
4        protected final double[] components;
5
6        /* Use varargs to create an arbitrary dimensional vector */
7        public Vector (double ... components) {
8              this.dimension = components.length;
9              this.components = new double[dimension];
10             for (int i = 0; i < dimension; i++)
11                   this.components[i] = components[i];
12       }
13
14       /* Returns the dimensionality of the vector */
15       public int getDimension () {
16             return this.dimension;
17       }
18
19       /* Returns the component at the specified index.
20          This uses indexing starting at '1' per mathematical convention */
21       public double getComponent (int index) {
22             return this.components[index - 1];
```

```
23          }
24
25          /* Returns the absolute value/magnitude of the vector */
26          public double getAbsoluteValue () {
27                  double abs = 0.0;
28                  for (int i = 0; i < dimension; i++)
29                          abs += (components[i] * components[i]);
30                  return Math.sqrt(abs);
31          }
32
33          /* Wrapper methods which call static ones */
34
35          public Vector multiplyByScalar (double k) {
36                  return Vector.multiplyByScalar(this, k);
37          }
38
39          public Vector add (Vector v) {
40                  return Vector.add(this, v);
41          }
42
43          public double dotProduct (Vector v) {
44                  return Vector.dotProduct(this, v);
45          }
46
47          public double angleBetween (Vector v) {
48                  return Vector.angleBetween(this, v);
49          }
50
51          public boolean equals (Vector v) {
52                  return Vector.equals(this, v);
53          }
54
55          /* Format vector components neatly */
56          @Override
57          public String toString () {
58                  String s = "(";
59                  for (double component : components)
60                          s += component + ", ";
61                  return s.replaceAll(", $", ")");
62          }
63
64          /* Checks for equality between two vectors */
65          public static boolean equals (Vector a, Vector b) {
66                  /* Dimensionalities must be equal */
67                  if (a.getDimension() != b.getDimension())
68                          return false;
```

```
69              /* Corresponding components must be equal */
70              for (int i = 1; i <= a.getDimension(); i++)
71                      if (a.getComponent(i) != b.getComponent(i))
72                              return false;
73              return true;
74      }
75
76      /* Multiplies a vector by a scalar to return a vector */
77      public static Vector multiplyByScalar (Vector v, double k) {
78              double[] t = new double[v.getDimension()];
79              for (int i = 0; i < t.length; i++)
80                      t[i] = v.getComponent(i+1) * k;
81              return new Vector(t);
82      }
83
84      /* Adds two vectors to return a vector */
85      public static Vector add (Vector a, Vector b) {
86              double[] sum = new double[a.getDimension()];
87              /* Add corresponding components */
88              for (int i = 0; i < sum.length; i++)
89                      sum[i] = a.getComponent(i+1) + b.getComponent(i+1);
90              return new Vector(sum);
91      }
92
93      /* Adds multiple vectors to return a vector */
94      public static Vector add (Vector ... vectors) {
95              Vector v = vectors[0];
96              /* Repeatedly use the binary addition method */
97              for (int i = 1; i < vectors.length; i++)
98                      v = Vector.add(v, vectors[i]);
99              return v;
100     }
101
102     /* Returns the dot product of two vectors */
103     public static double dotProduct (Vector a, Vector b) {
104             double dotProduct = 0.0;
105             /* Multiply corresponding components */
106             for (int i = 1; i <= a.getDimension(); i++)
107                     dotProduct += a.getComponent(i) * b.getComponent(i);
108             return dotProduct;
109     }
110
111     /* Returns the angle between two vectors in radians.
112        If 'u' and 'v' are vectors, with an angle 'A' between them,
113        u.v = |u||v| cos(A) */
114     public static double angleBetween (Vector a, Vector b) {
```

```
115                 return Math.acos(Vector.dotProduct(a, b) / (a.getAbsoluteValue() *
                        b.getAbsoluteValue()));
116         }
117 }
```

```
1  public class VectorDemo {
2         public static void main (String[] args) {
3                 /* Simple 2D vector with magnitude sqrt(2) */
4                 Vector a = new Vector(1, 1);
5                 System.out.printf("Magnitude of %s is %f\n", a, a.getAbsoluteValue());
6
7                 /* Create 3 random 3D vectors */
8                 Vector b = new Vector(random(-10, 10), random(-10, 10), random(-10,
                      10));
9                 Vector c = new Vector(random(-10, 10), random(-10, 10), random(-10,
                      10));
10                Vector d = new Vector(random(-10, 10), random(-10, 10), random(-10,
                      10));
11
12                /* Demonstrate addition, dot products, angle measurement */
13                System.out.printf("Sum of vectors %s, %s, %s is %s\n", b, c, d,
                      Vector.add(b, c, d));
14                System.out.printf("Dot product of %s and %s is %d\n", b, c, (int)
                      Vector.dotProduct(b, c));
15                System.out.printf("The angle between %s and %s is %f degrees\n", b, c,
16                                Math.toDegrees(Vector.angleBetween(b, c)));
17        }
18
19        /* Returns random integers in a specified range */
20        public static int random (int lo, int hi) {
21                return (int) (lo + ((hi - lo) * Math.random()));
22        }
23 }
```

Variable Description

Vector		
int	dimension	The dimension of the vector
double[]	components	The ordered list of components of the vector

Vector::Vector(double[])		
double[]	components	The ordered list of components of the vector

Vector::getComponent(int)		
int	index	The index of the component to be retrieved

Vector::getAbsoluteValue()		
double	abs	Stores the square of the magnitude of the vector
int	i	Counter variable, counts through components of the vector

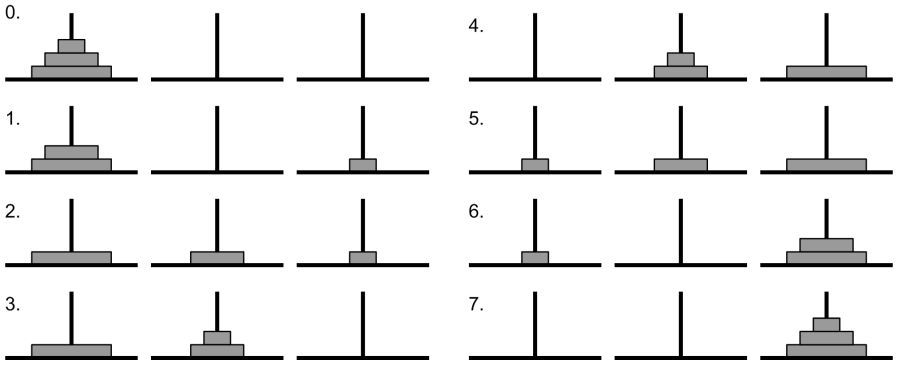
Vector::multiplyByScalar(double)		
double	k	The scalar to multiply the vector by

"In order to understand recursion, one must first understand recursion."

— **Anonymous**

Problem 20 The *Tower of Hanoi* is a mathematical puzzle, consisting of three rods and a number of disks of different sizes which can slide onto any rod. The puzzle starts with all disks, in ascending order of size, on one rod. The objective of the puzzle is to move the entire stack to another rod, obeying the following rules.

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one stack and placing it on the top of another stack or empty rod.
3. No disk can be placed on a smaller disk.



Solution to the Towers of Hanoi with 3 disks.

Solve the *Tower of Hanoi* puzzle for an arbitrary number of disks, enumerating the required moves.

Solution The main insight here is that the problem involving n disks can be reduced to one with $n-1$ disks. Labelling the rods A, B and C, and the disks with numerals 1 through n (smallest to largest), our aim is to move the entire stack from A to C. If we can solve the problem with $n-1$ disks, all we have to do is to move the topmost $n-1$ disks from A to B, move the remaining disk on A to C, and again move the $n-1$ disks on B to C. The base case for this recursive solution is moving 1 disk, which is trivial.

Clearly, if the problem with n disks takes k_n number of moves, the problem with $n+1$ moves will take $k_n + 1 + k_n = 2k_n + 1$ moves. For the base case with one disk,

$k_1 = 1$. With this infromation, we see that the *Tower of Hanoi* with n disks can be solved in exactly $2^n - 1$ moves.

`main (disks:Integer)`

1. Call `solveHanoi(disks, "A", "C", "B")`.
2. **Exit**

`solveHanoi (disk:Integer, source:String, destination:String, spare:String)`

1. If `disk` is zero, **return**.
2. Call `solveHanoi(disk - 1, source, spare, destination)`.
3. Move disk number `disk` has to be moved from `source` to `destination`.
4. Call `solveHanoi(disk - 1, spare, destination, source)`.
5. **Return**

Source Code

```
1  public class TowersOfHanoi {
2        public static void main (String[] args) {
3              try {
4                     /* Parse the first command line argument as the number of
                          disks */
5                     int disks = Integer.parseInt(args[0]);
6                     /* Make sure there is at least one disk */
7                     if (disks < 1)
8                           throw new NumberFormatException();
9                     /* Initiate the recursive steps */
10                    solveHanoi(disks, "A", "C", "B");
11             } catch (NumberFormatException | IndexOutOfBoundsException e) {
12                    /* Handle missing or incorrectly formatted arguments */
13                    System.out.println("Enter 1 argument (number_of_disks[integer,
                          >0])!");
14             }
15       }
16
17       /* Displays moves to solve the Towers of Hanoi problem with 3 pegs */
18       public static void solveHanoi (int disk, String source, String destination,
             String spare) {
19             /* Base case - nothing to do */
20             if (disk == 0)
21                    return;
22             /* Move the stack of (n-1) disks to the spare peg */
23             solveHanoi(disk - 1, source, spare, destination);
24             /* Move the largest disk to the destination */
```

```
25                System.out.printf("(%d) : %s -> %s%n", disk, source, destination);
26                /* Move the stack of (n-1) disks back on top of the largest
27                   disk, on the destination peg */
28                solveHanoi(disk - 1, spare, destination, source);
29        }
30  }
```

Variable Description

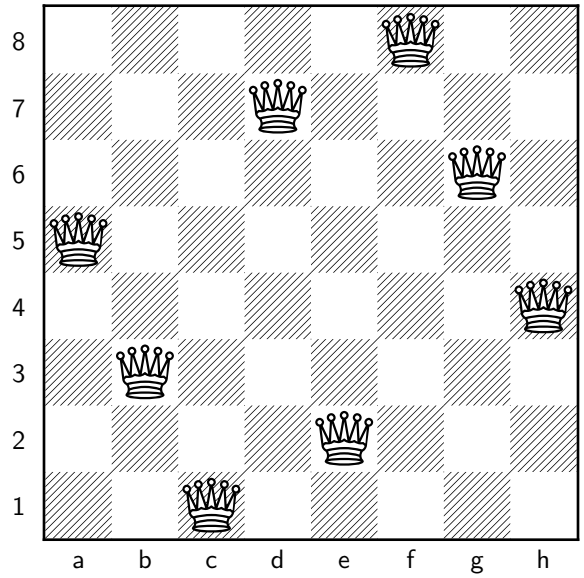
TowersOfHanoi::main(String[])		
int	disks	The number of disks in the problem

TowersOfHanoi::solveHanoi(int, String, String, String)		
int	disk	The current disk to be moved
String	source	The rod from which the stack is to be moved
String	destination	The rod to which the stack is to be moved
String	spare	The additional rod, where the remaining n-1 disks are temporarily moved

"Chess is the gymnasium of the mind."

— Blaise Pascal

Problem 21 The *8 queens puzzle* involves placing 8 queens on an 8×8 chessboard such that no two queens threaten each other, i.e. no two queens share the same rank, file or diagonal. It was first published by the chess composer *Max Bezzel* in 1848. This puzzle has 92 solutions, including reflections and rotations. Below is one of them.



The *n queens puzzle* is an extension of this puzzle, involving n queens on an $n \times n$ chessboard. Count the total number of solutions for the *n queens puzzle*, including reflections and rotations.

Solution This problem can be solved with *recursion* and *backtracking*. Starting from the topmost row of the chessboard, we can place a queen and for each available choice, place a queen on the next row, and so on, recursively shrinking the chessboard to solve. Invalid solutions can thus be discarded as they are formed without brute-forcing every possible permutation of queens on the board.

Finally, by noting that exactly one queen must occupy each row, we can optimize the board by storing only the column numbers of queens on each row in an array, instead of simulating a full 2D board.

`main (size:Integer, drawSolutions:Boolean)`

1. Create an `NQueens` object by passing it `size` and `drawSolutions`. Call it `q`.
2. Call `q->countSolutions()` and display the result.
3. **Exit**

`NQueens (size:Integer, drawSolutions:Boolean)`

1. Copy `size` and `drawSolutions` into the object data.
2. Initialize an integer `numberOfSolutions` to zero.
3. Initialize an array of integers with length `size`. Call it `board`.
4. **Define** the functions:
   (a) `NQueens::countSolutions()`
   (b) `NQueens::solveNQueens(row)`
   (c) `NQueens::isThreatened(row)`
5. **Return** the resultant object.

`NQueens::countSolutions ()`

1. Call `this->solveNQueens(0)`.
2. **Return**

`NQueens::solveNQueens (row:Integer)`

1. If `row` is equal to `size`:
   (a) Increment `numberOfSolutions`.
   (b) If `drawSolutions` is set to `true`, display the current state of `board`.
   (c) **Return**
2. For each `i` $\in \{0, 1, \ldots,$ `size` $- 1\}$:
   (a) Place a queen at row `row`, column `i`, i.e. set `board[row]` to `i`.
   (b) Call `this->isThreatened(row)`. If this returns `false`, call `this->solveNQueens(row + 1)`.
3. **Return**

`NQueens::isThreatened (row:Integer)`

1. For each `i` $\in \{0, 1, \ldots,$ `size` $- 1\}$:
   (a) If there are two queens on the same column in rows `row` and `i`, or the columns in which those two queens are on are on the same diagonal, **return** `true`.
2. **Return** `false`

Source Code

```
public class NQueens {
	private final int size;
	private int[] board;
	private int numberOfSolutions;
	private final boolean drawSolutions;

	/* Sets the size of the board and the number of queens */
	public NQueens (int size, boolean drawSolutions) {
		this.size = size;
		this.drawSolutions = drawSolutions;
		this.initBoard();
	}

	/* Returns the number of solutions to a board of given size */
	public int countSolutions () {
		solveNQueens(0);
		return numberOfSolutions;
	}

	/* Initializes the board */
	private void initBoard () {
		this.board = new int[size];
		this.numberOfSolutions = 0;
		for (int i = 0; i < size; i++)
			board[i] = -1;
	}

	/* Determines whether the queen on a specified row is threatened
	   by a queen on a previous row */
	private boolean isThreatened (int row) {
		for (int i = 0; i < row; i++) {
			if ((board[row] == board[i])
					|| ((board[row] - board[i]) == (row - i))
					|| ((board[row] - board[i]) == (i - row))) {
				return true;
			}
		}
		return false;
	}

	/* Recursively solves the n-queens problem */
	private void solveNQueens (int row) {
		if (row == size) {
			/* Reached maximum recursion depth - found a solution */
```

```
45                      numberOfSolutions++;
46                      if (drawSolutions) {
47                              drawBoard();
48                              System.out.println();
49                      }
50                      return;
51              }
52              /* Place queens on all possible columns on the row */
53              for (board[row] = 0; board[row] < size; board[row]++) {
54                      if (!isThreatened(row)) {
55                              /* Recurse if the board is valid so far */
56                              solveNQueens(row + 1);
57                      }
58              }
59      }
60
61      /* Displays the current configuration of the board */
62      public void drawBoard () {
63              for (int i = 0; i < size; i++) {
64                      for (int j = 0; j < size; j++) {
65                              System.out.print(((board[i] == j)? "Q" : "-") + " ");
66                      }
67                      System.out.println();
68              }
69      }
70
71      public static void main (String[] args) {
72              try {
73                      /* Parse the first command line argument as the size of the
                           board */
74                      int size = Integer.parseInt(args[0]);
75                      /* Parse the second command line argument as a boolean,
76                         indicating whether to draw the solved boards.
77                         Defaults to not showing the solutions */
78                      boolean drawSolutions = (args.length > 1)?
                           Boolean.parseBoolean(args[1]) : false;
79                      /* Make sure the board exists */
80                      if (size < 1)
81                              throw new NumberFormatException();
82                      /* Create a 'NQueens' object */
83                      NQueens q = new NQueens(size, drawSolutions);
84                      /* Display the number of solutions */
85                      System.out.println(q.countSolutions());
86              } catch (NumberFormatException | IndexOutOfBoundsException e) {
87                      /* Handle missing or incorrectly formatted arguments */
```

```
88                      System.out.println("Enter at least 1 argument
                            (size_of_board[integer], <show_solutions>[true/false])!");
89                      System.out.println("(show_solutions defaults to false)");
90              }
91      }
92  }
```

Variable Description

NQueens		
int	size	The number of rows and columns in the chessboard
int[]	board	The list of positions of queens in columns, with their rows corresponding to their index.
int	numberOfSolutions	Counts the number of solutions found
boolean	drawSolutions	Stores whether to display solved boards or not

NQueens::isThreatened(int)		
int	row	The row of the queen to test
int	i	Counter variable, stores the row of the queen to test against

NQueens::solveNQueens(int)		
int	row	The current row on which a queen is to be placed

NQueens::drawBoard()		
int	i, j	Counter variables, store the row and column to be currently displayed

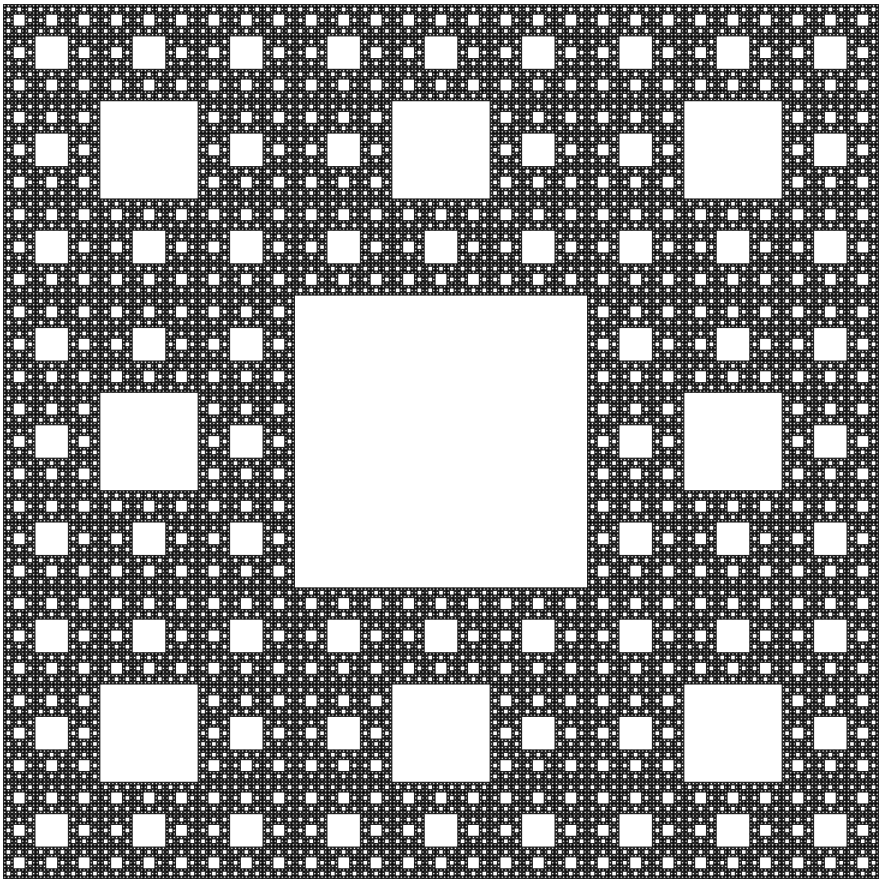
NQueens::main(String[])		
int	size	The number of rows and columns in the chessboard
boolean	drawSolutions	Stores whether to display solved boards or not
NQueens	q	Object capable of solving the *n queens* problem

"In the mind's eye, a fractal is a way of seeing infinity."

— **James Gleick**

Problem 22 The *Sierpinski Carpet* is a plane fractal. It can be produced iteratively by taking a solid square, dividing it into 9 congruent squares in a 3-by-3 grid, removing the centre square, and recursively applying the same procedure on each of the remaining squares *ad infinitum*.

Display the *Sierpinski Carpet* to a specified number of iterations.



The Sierpinski Carpet

Solution In an ASCII terminal, we can only display a rough representation of the *Sierpinski Carpet*, a few levels deep. A level n carpet will have a width and height of 3^n. Within this grid, every character lies either in the centre of a 3-by-3 square, in which case it is not in the carpet, or it lies on the edge, in which case it is in the carpet. If neither can be determined, we can scale up the search square to the next level, and repeat recursively.

Here, points in the carpet are drawn, while points not in the the carpet are left as whitespace.

`main (level:Integer)`

1. For each pair (`i` , `j`) $\in \{0, 1, \ldots, 3^n - 1\} \times \{0, 1, \ldots, 3^n - 1\}$:
   (a) Call `isInSierpinskiCarpet(i, j)`. If it returns `true`, display a solid block at (`i`, `j`), otherwise, leave a blank space there.
2. **Exit**

`isInSierpinskiCarpet (x:Integer, y:Integer)`

1. If either of `x` or `y` is zero, the point (`x`, `y`) is on the edge of a square of some level. **Return** `true`.
2. If both `x` and `y` leave a remainder of one on division by 3, the point (`x`, `y`) is at the centre of a square of some level. **Return** `false`.
3. Call `isInSierpinskiCarpet(x / 3, y / 3)`, and **return** the returned value.

Source Code

```
1  public class SierpinskiCarpet {
2        public static void main (String[] args) {
3              try {
4                    /* Parse the first command line argument as the level of
                        detail of the carpet */
5                    int level = Integer.parseInt(args[0]);
6                    /* Make sure that the level is positive */
7                    if (level < 0)
8                          throw new NumberFormatException();
9                    /* Iterate over every 'point' in the carpet */
10                   for (int i = 0; i < Math.pow(3, level); i++) {
11                         for (int j = 0; j < Math.pow(3, level); j++) {
12                               /* Display a full block for points 'in' the
                                    carpet */
13                               System.out.print(isInSierpinskiCarpet(i, j)?
                                    "\u2588\u2588" : " ");
14                         }
```

```
15                          System.out.println();
16                  }
17          } catch (NumberFormatException | IndexOutOfBoundsException e) {
18                  /* Handle missing or incorrectly formatted arguments */
19                  System.out.println("Enter 1 argument
                        (order_of_carpet[integer])!");
20          }
21      }
22
23      /* Determines whether a point is in the carpet */
24      public static boolean isInSierpinskiCarpet (int x, int y) {
25          /* Blocks are in the carpet if they are on the edge */
26          if (x == 0 || y == 0)
27                  return true;
28          /* Blocks at the centres of 3-by-3 squares on any level are
29             not in the carpet */
30          if (((x % 3) == 1) && ((y % 3) == 1))
31                  return false;
32          /* Recurse to the next, larger level */
33          return isInSierpinskiCarpet(x / 3, y / 3);
34      }
35  }
```

Variable Description

SierpinskiCarpet::main(String[])		
int	level	The depth to which to render the carpet
int	i, j	Counter variables, represent a point on the screen to be displayed

SierpinskiCarpet::isInSierpinskiCarpet(int, int)		
int	x, y	Counter variables, represent the point in question

"Computers are useless. They can only give you answers."

— Pablo Picasso

Problem 23 *Reverse Polish Notation (RPN)* or *postfix notation* is a mathematical notation for writing arithmetic expresssions in which operators follow their operands. Thus, as long as each operator has a fixed number of operands, the use of parentheses or rules of precedence are no longer required to write unambiguous expressions. For example, the expression `2 3 * 3 2 ^ 2 - *` evaluates to `42`.

Create a program capable of evaluating *RPN* expressions which use the following operators.

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	Exponentiation

Solution The nature of *RPN* lends itself to a very simple implementation with a stack for pushing operands into as they appear in an expression. When an operator is encountered, the required number of operands are popped from the stack, the operation is carried out, and the result is popped back into the stack. This continued until the entire expression has been parsed, leaving only the evaluated result in the stack.

`main (expression:String)`

1. Call `evaluateRPNExpression(expression)` and display the returned value.
2. **Exit**

`evaluateRPNExpression (expression:String)`

1. Split `expression` along whitespace into an array of tokens. Call it `tokens`.
2. Create a stack of floating points large enough to hold all elements in `tokens`. Call it `operandStack`.
3. For each string `token` $\in$ `tokens`:
   (a) If `token` is a floating point:
      i. Push `token` onto `operandStack`.
      ii. Get the next `token` from `tokens`.
      iii. Jump back to (3a).
   (b) Pop an operand from `operandStack` and call it `rightOperand`.
   (c) Pop another operand from `operandStack` and call it `leftOperand`.

(d) Depending on which operator `token` represents, evaluate the operation with `token` as the operator and `leftOperand` and `rightOperand` as the respective operands. Call it `result`.

(e) Push `result` onto `operandStack`.

4. Pop and operand from `operandStack` and **return** it.

Source Code

```
1  import java.util.Scanner;
2
3  public class RPNCalculator {
4         /* Simple stack setup */
5         private static double[] operandStack;
6         private static int top;
7
8         public static void main (String[] args) {
9                /* Prompt an RPN expression from the terminal */
10               System.out.printf("Reverse Polish Expression : ");
11               String expression = (new Scanner(System.in)).nextLine();
12               /* Evaluate the expression and display the result */
13               double result = evaluateRPNExpression(expression);
14               System.out.printf("Evaluated Expression  :   %s %n",
                     Double.toString(result));
15        }
16
17        /* Evaluates expression in RPN */
18        public static double evaluateRPNExpression (String expression) {
19               /* Split the expression into tokens */
20               String[] tokens = expression.split("\\s+");
21               /* Initialize the stack with an appropriately large capacity */
22               top = -1;
23               operandStack = new double[tokens.length];
24
25               /* Iterate through all tokens in the expression */
26               for (String token : tokens) {
27                      /* Push operands into the stack and continue */
28                      if (isDouble(token)) {
29                             pushOperand(Double.parseDouble(token));
30                             continue;
31                      }
32
33                      /* Pop operands from the stack */
34                      double rightOperand = popOperand();
35                      double leftOperand = popOperand();
```

```
36                      /* Determine the operator encountered and calculate the
                            appropriate result */
37                      double result = 0.0;
38                      switch (token.charAt(0)) {
39                              case '+' :    result = leftOperand + rightOperand;
40                                            break;
41                              case '-' :    result = leftOperand - rightOperand;
42                                            break;
43                              case '*' :    result = leftOperand * rightOperand;
44                                            break;
45                              case '/' :    result = leftOperand / rightOperand;
46                                            break;
47                              case '^' :    result = Math.pow(leftOperand,
                                    rightOperand);
48                                            break;
49                              default :     System.out.printf("Unknown operator
                                    (%s)!\n", token);
50                                            System.exit(0);
51                      }
52                      /* Push the result onto the stack */
53                      pushOperand(result);
54              }
55              /* Return the last item in the stack */
56              return popOperand();
57      }
58
59      /* Pushes an operand onto the stack */
60      private static void pushOperand (double n) {
61              operandStack[++top] = n;
62      }
63
64      /* Pops an operand from the stack. Exits on failure. */
65      private static double popOperand () {
66              if (top < 0) {
67                      System.out.println("Insufficient operands!");
68                      System.exit(0);
69              }
70              return operandStack[top--];
71      }
72
73      /* Determines whether a token is a number */
74      private static boolean isDouble (String n) {
75              try {
76                      Double.parseDouble(n);
77                      return true;
78              } catch (NumberFormatException e) {}
```

```
79                  return false;
80          }
81  }
```

Variable Description

RPNCalculator		
double[]	operandStack	The stack of operands in order of appearance.
int	top	The index of the topmost element of operandStack

RPNCalculator::main(String[])		
String	expression	The expression in RPN to be evaluated
double	result	The evaluated form of expression

RPNCalculator::evaluateRPNExpression(String)		
String	expression	The expression in RPN to be evaluated
String[]	tokens	The individual tokens in expression, separated by whitespace
String	token	An individual token from tokens
double	rightOperand	The operand to be taken on the right side of the operator
double	leftOperand	The operand to be taken on the left side of the operator
double	result	The result on evaluating the operator token on rightOperand and leftOperand

RPNCalculator::pushOperand(double)		
double	n	The operand to be pushed into operandStack

RPNCalculator::isDouble(String)		
String	n	The string to be tested on whether it is a floating point or not

"Computer Science is no more about computers than astronomy is about telescopes."

— **Edsger W. Dijkstra**

Problem 24 A *queue* is a linear data structure which allows storage and retrieval of elements in accordance with the *First In First Out (FIFO)* principle. Thus, elements exit a *queue* in the same order they entered it.

Implement a *queue* capable of holding an arbitrary number of elements of a specified type.

Solution The use of *linked lists*[12] is apropriate here. *Generics* ensure that once a queue is declared with a data type, only elements of that data type can be added to it, as opposed to merely storing `Objects`.

`Node<T> (item:T)`

1. Copy `item` as an object variable.
2. Declare two variables `left` and `right`, both of type `Node<T>`.
3. **Return** the resultant object.

`link (left:Node<T>, right:Node<T>)`

1. Set `left->right` to `right`.
2. Set `right->left` to `left`.

`LinkedQueue<T> ()`

1. Declare two constants `HEAD` and `TAIL`, both of type `Node<T>` with arbitrary data items.
2. Link `TAIL` and `HEAD`.
3. **Define** the functions:
   (a) `LinkedQueue<T>::enqueue(item)`
   (b) `LinkedQueue<T>::dequeue()`
   (c) `LinkedQueue<T>::peek()`
   (d) `LinkedQueue<T>::clear()`
   (e) `LinkedQueue<T>::isEmpty()`
   (f) `LinkedQueue<T>::size()`

[12] A linked list is a linear data structure where each element is a separate object, or *node*. Each *node* contains both *data* and *addresses* of the surrounding nodes.

4. **Return** the resultant object.

`LinkedQueue<T>::enqueue (item:T)`

1. Create a new `Node<T>`, pass it `item`, and call it `newNode`.
2. Link `HEAD->left` and `newNode`.
3. Link `newNode` and `HEAD`.

`LinkedQueue<T>::dequeue ()`

1. If the queue is empty, return `null`.
2. Temporarily store the node `TAIL->right` as `lastNode`.
3. Link `TAIL` and `lastNode->right`.
4. **Return** the item contained in `lastNode`.

`LinkedQueue<T>::peek ()`

1. **Return** the item in the node `TAIL->right`.

`LinkedQueue<T>::clear ()`

1. Link `TAIL` and `HEAD`.

`LinkedQueue<T>::isEmpty ()`

1. If the `TAIL->right` is `HEAD`, **return** `true`, otherwise **return** `false`.

`LinkedQueue<T>::size ()`

1. Initialize an integer `n` to zero.
2. Set a variable `current` to `TAIL`.
3. While `current->right` is not `HEAD`, set `current` to `current->right` and increment `n`.
4. **Return** `n`.

Source Code

```
1  public class Node<T> {
2        /* Item data is immutable */
3        protected final T item;
4
5        /* References to other nodes */
6        protected Node<T> left;
7        protected Node<T> right;
8
9        /* Set the data item */
```

```
10          public Node (T item) {
11                  this.item = item;
12          }
13
14          /* Get the data item */
15          public T getItem () {
16                  return item;
17          }
18
19          /* Use the data item's 'toString()' method */
20          @Override
21          public String toString () {
22                  return item.toString();
23          }
24
25          /* Doubly link two nodes */
26          public static <T> void link (Node<T> left, Node<T> right) {
27                  left.right = right;
28                  right.left = left;
29          }
30  }
```

```
1   import java.util.Iterator;
2
3   /* Use generics to allow arbitrary data typed queues, with type checking
4      enforced at compile-time */
5   public class LinkedQueue<T> implements Iterable<T> {
6          /* Special nodes surrounding data nodes */
7          protected final Node<T> HEAD = new Node<T>(null);
8          protected final Node<T> TAIL = new Node<T>(null);
9
10         public LinkedQueue () {
11                 Node.<T>link(TAIL, HEAD);
12         }
13
14         /* Enqueues a data item of generic type into the head */
15         public void enqueue (T item) {
16                 Node<T> newNode = new Node<T>(item);
17                 Node.<T>link(HEAD.left, newNode);
18                 Node.<T>link(newNode, HEAD);
19         }
20
21         /* Dequeues a data item from the tail */
22         public T dequeue () {
23                 if (this.isEmpty())
```

```
24                      return null;
25              Node<T> lastNode = TAIL.right;
26              Node.<T>link(TAIL, lastNode.right);
27              return lastNode.getItem();
28      }
29
30      /* Returns the data item at the tail without removing it */
31      public T peek () {
32              return TAIL.right.getItem();
33      }
34
35      /* Clears the queue */
36      public void clear () {
37              /* Garbage collection takes care of orphaned nodes */
38              Node.<T>link(TAIL, HEAD);
39      }
40
41      /* Checks if the queue is empty */
42      public boolean isEmpty () {
43              return TAIL.right == HEAD;
44      }
45
46      /* Returns the size of the queue */
47      public int size () {
48              int n = 0;
49              /* Start at the tail */
50              Node<T> current = TAIL;
51              /* Iterate through all nodes until the head */
52              while ((current = current.right) != HEAD)
53                      n++;
54              return n;
55      }
56
57      /* Formats the elements of the queue neatly */
58      @Override
59      public String toString () {
60              String[] elements = new String[this.size()];
61              Node<T> current = TAIL;
62              int n = 0;
63              while ((current = current.right) != HEAD)
64                      elements[n++] = current.toString();
65              return "[" + String.join(", ", elements) + "]";
66      }
67
68      /* Allow the elements of the queue to be iterated over simply */
69      @Override
```

```
70          public Iterator<T> iterator () {
71                  return new Iterator<T>() {
72                          private Node<T> current = TAIL.right;
73
74                          @Override
75                          public boolean hasNext () {
76                                  return current != HEAD;
77                          }
78
79                          @Override
80                          public T next () {
81                                  T item = current.getItem();
82                                  current = current.right;
83                                  return item;
84                          }
85
86                          @Override
87                          public void remove () {
88                                  throw new UnsupportedOperationException();
89                          }
90                  };
91          }
92  }
```

```
1   public class QueueDemo {
2           public static void main (String[] args) {
3                   /* Create an integer queue */
4                   LinkedQueue<Integer> q = new LinkedQueue<Integer>();
5
6                   /* Enqueue random numbers to the queue */
7                   for (int i = 0; i < (10 + (int) (10 * Math.random())); i++) {
8                           Integer n = (int) (100 * Math.random());
9                           System.out.printf("Enqueuing : %s\n", n);
10                          q.enqueue(n);
11                  }
12                  /* Demonstrate simple output formatting */
13                  System.out.printf("Queue[%2d] : %s\n", q.size(), q);
14
15                  /* Demonstrate peeking */
16                  System.out.printf("Number about to be dequeued : %s\n", q.peek());
17
18                  /* Demonstrate the FIFO principle in effect */
19                  System.out.println("(Dequeuing 10 numbers)");
20                  for (int i = 0; i < 10; i++)
21                          System.out.printf("Dequeuing : %s\n", q.dequeue());
```

```
22                System.out.printf("Queue[%2d] : %s\n", q.size(), q);
23
24                /* Demonstrate iteration until empty */
25                System.out.println("(Dequeueing until empty)");
26                while (!q.isEmpty())
27                       System.out.printf("Dequeuing : %s\n", q.dequeue());
28                System.out.printf("Queue[%2d] : %s\n", q.size(), q);
29        }
30 }
```

Variable Description

Node<T>		
T	item	The data stored in the node
Node<T>	left	Reference to the node to the left of this
Node<T>	right	Reference to the node to the right of this
LinkedQueue<T>		
Node<T>	HEAD	Special node, marks the point of entry of new data
Node<T>	TAIL	Special node, marks the point of exit of data
LinkedQueue<T>::enqueue(T)		
T	item	The data to be enqueued
Node<T>	newNode	The node containing the data to be enqueued
LinkedQueue<T>::dequeue()		
Node<T>	lastNode	The node containing the data to be dequeued
LinkedQueue<T>::size()		
int	n	Stores the number of elements in the queue
LinkedQueue<T>::toString()		
String[]	elements	Temporary array, stores the string representations of the data items in the queue
int	n	Counter variable

"A good way to have good ideas is by being unoriginal."

— Bram Cohen

Problem 25 A *double ended queue*, or *DEqueue* is a linear data structure which allows the insertion and deletion of data items from both the front and rear.

Implement a *double ended queue* capable of holding an arbitrary number of elements of a specified type.

Solution This problem can be solved by extending the funcitonality of the *queue* defined in the previous problem. The algorithms for insertion and deletion at one end mirror those for the other.

`LinkedDEQueue<T> ()`

1. Call the constructor of the superclass `LinkedQueue`.
2. **Define** the functions:
   (a) `LinkedDEQueue<T>::enqueueRear(item)`
   (b) `LinkedDEQueue<T>::dequeueFront()`
3. **Return** the resultant object.

`LinkedDEQueue<T>::enqueueRear (item:T)`

1. Create a new `Node<T>`, pass it `item`, and call it `newNode`.
2. Link `newNode` and `TAIL->right`.
3. Link `TAIL` and `newNode`.

`LinkedDEQueue<T>::dequeueFront ()`

1. If the queue is empty, return `null`.
2. Temporarily store the node `HEAD->left` as `firstNode`.
3. Link `firstNode->left` and `HEAD` .
4. **Return** the item contained in `firstNode`.

Source Code

```
import java.util.Iterator;

/* Extend LinkedQueue<T> to build on existing functionality */
public class LinkedDEQueue<T> extends LinkedQueue<T> {

	/* Enqueues a data item of generic type into the tail */
	public void enqueueRear (T item) {
		Node<T> newNode = new Node<T>(item);
		Node.<T>link(newNode, TAIL.right);
		Node.<T>link(TAIL, newNode);
	}

	/* Dequeues a data item from the head */
	public T dequeueFront () {
		if (this.isEmpty())
			return null;
		Node<T> firstNode = HEAD.left;
		Node.<T>link(firstNode.left, HEAD);
		return firstNode.getItem();
	}

	/* Descending iterator */
	public Iterator<T> descendingIterator () {
		return new Iterator<T>() {
			private Node<T> current = HEAD.left;

			@Override
			public boolean hasNext () {
				return current != TAIL;
			}

			@Override
			public T next () {
				T item = current.getItem();
				current = current.left;
				return item;
			}

			@Override
			public void remove () {
				throw new UnsupportedOperationException();
			}
		};
	}
```

```
45 }
```

```
1  public class DEQueueDemo {
2         public static void main (String[] args) {
3                 /* Create an integer DEqueue */
4                 LinkedDEQueue<Integer> dq = new LinkedDEQueue<Integer>();
5
6                 /* Enqueue random numbers to the front of the DEqueue */
7                 for (int i = 0; i < (7 + (int) (5 * Math.random())); i++) {
8                         Integer n = (int) (100 * Math.random());
9                         System.out.printf("Enqueuing (Front) : %s\n", n);
10                        dq.enqueue(n);
11                }
12                /* Enqueue random numbers to the rear of the DEqueue */
13                for (int i = 0; i < (7 + (int) (5 * Math.random())); i++) {
14                        Integer n = (int) (100 * Math.random());
15                        System.out.printf("Enqueuing ( Rear) : %s\n", n);
16                        dq.enqueueRear(n);
17                }
18                /* Demonstrate simple output formatting */
19                System.out.printf("DEQueue[%2d] : %s\n", dq.size(), dq);
20
21                /* Dequeue items from the front of the DEQueue */
22                System.out.println("(Dequeuing 10 numbers (Front))");
23                for (int i = 0; i < 10; i++)
24                        System.out.printf("Dequeuing : %s\n", dq.dequeueFront());
25                System.out.printf("Queue[%2d] : %s\n", dq.size(), dq);
26
27                /* Dequeue items from the rear of the DEQueue until empty */
28                System.out.println("(Dequeueing until empty (Rear))");
29                while (!dq.isEmpty())
30                        System.out.printf("Dequeuing : %s\n", dq.dequeue());
31                System.out.printf("DEQueue[%2d] : %s\n", dq.size(), dq);
32        }
33 }
```

Variable Description

LinkedDEQueue<T>::enqueueRear(T)		
T	item	The data to be enqueued

LinkedDEQueue<T>::dequeueFront()		
Node<T>	firstNode	The node containing the data to be dequeued

"You can't trust code that you did not totally create yourself."

— Ken Thompson

Problem 26 Arrange the words in a given sentence of input in alphabetical order. *(Ignore case, duplicated words.)*

Solution This problem can be solved using a data structure called a *binary tree*.

A *binary tree* consists of multiples *nodes*, each of which holds a data item. Ideally, these items can be *ordered*, i.e., there is a way to compare them, using a value called a *key*. Each node is connected to two nodes below it — the *left child* and the *right child*. The left child has lower *key*, while the right child has a higher *key* than the parent node. The node at the top of a given binary tree is called its *root*.

Binary trees have a nice recursive form, in that the left and right children of the root can be regarded as roots of individual binary trees — the *left* and *right subtrees* of the root. This makes it easy to write recursive algorithms for searching, inserting, and deleting nodes from a binary tree.

Searching and insertion in a binary tree containing n nodes have an average time complexity $O(\log n)$.

`TreeNode<T> (item:T)`

1. Copy `item` as an object variable.
2. Declare two variables `left` and `right`, both of type `Node<T>`.
3. **Return** the resultant object.

`BinaryTree<T> (root:TreeNode<T>)`

1. Copy `root` as an object variable.
2. **Define** the functions:
   (a) `BinaryTree<T>::contains(item)`
   (b) `BinaryTree<T>::search(item)`
   (c) `BinaryTree<T>::add(item)`
3. **Return** the resultant object.

`BinaryTree<T>::contains (item:T)`

1. If `this->search(item)` returns a non-null object, **return** `true`, otherwise **return** `false`.

`BinaryTree<T>::search (item:T)`

1. **Return** `search(this->root, item)`

`BinaryTree<T>::add (item:T)`

1. Set `this->root` to the `TreeNode` returned by `add(this->root, item)`.

`search (root:TreeNode<T>, item:T)`

1. If `item` $<$ `root->item`, **return** `search(root->left, item)`
2. If `item` $>$ `root->item`, **return** `search(root->right, item)`
3. **Return** `root`

`add (root:TreeNode<T>, item:T)`

1. If `root` is `null`, set it to a new `TreeNode<T>` containing `item` and **return** `root`.
2. If `item` $<$ `root->item`, set `root->left` to `add(root->left, item)`.
3. If `item` $>$ `root->item`, set `root->right` to `add(root->right, item)`.
4. **Return** `root`

`traverseInOrder (node:TreeNode<T>)`

1. If `node` is `null`, **return** an empty string.
2. **Return** `traverseInOrder(node->left)` + `node` + `traverseInOrder(node->right)` *(with spacing as necessary)*.

Source Code

```
1  public class TreeNode<T extends Comparable<T>> {
2         /* Item data is immutable */
3         protected final T item;
4
5         /* References to child nodes */
6         public TreeNode<T> left;
7         public TreeNode<T> right;
8
9         /* Set the data item */
10        public TreeNode (T item) {
11                this.item = item;
12                this.left = null;
13                this.right = null;
14        }
15
16        /* Get the data item */
17        public T getItem () {
18                return item;
```

```
        }

        /* Use the data item's 'toString()' method */
        @Override
        public String toString () {
                return item.toString();
        }
}
```

```
public class BinaryTree<T extends Comparable<T>> {
        /* The root node is at the top of all other nodes */
        protected TreeNode<T> root;

        public BinaryTree (TreeNode<T> root) {
                this.root = root;
        }

        /* Default to a 'null' root node */
        public BinaryTree () {
                this(null);
        }

        /* Checks whether the tree contains a given item */
        public boolean contains (T item) {
                return this.search(item) != null;
        }

        /* Returns the node containing a given item. If not found, returns 'null' */
        public TreeNode<T> search (T item) {
                return BinaryTree.<T>search(root, item);
        }

        /* Adds an item to the tree in order, if not already present */
        public void add (T item) {
                root = BinaryTree.<T>add(root, item);
        }

        /* Formats the items in the tree neatly, in order */
        @Override
        public String toString () {
                return BinaryTree.<T>traverseInOrder(this.root).trim();
        }

        /* Recursive binary search */
        public static <T extends Comparable<T>> TreeNode<T> search (TreeNode<T>
```

```
            root, T item) {
37              if (item.compareTo(root.item) < 0)
38                      return BinaryTree.<T>search(root.left, item);
39              if (item.compareTo(root.item) > 0)
40                      return BinaryTree.<T>search(root.right, item);
41              return root;
42      }
43
44      /* Recursive insertion of a node in a binary tree */
45      public static <T extends Comparable<T>> TreeNode<T> add (TreeNode<T> root, T
            item) {
46              if (root == null)
47                      root = new TreeNode<T>(item);
48              else if (item.compareTo(root.item) < 0)
49                      root.left = BinaryTree.<T>add(root.left, item);
50              else if (item.compareTo(root.item) > 0)
51                      root.right = BinaryTree.<T>add(root.right, item);
52              return root;
53      }
54
55      /* Recursive in order traversal of a binary tree */
56      public static <T extends Comparable<T>> String traverseInOrder (TreeNode<T>
            node) {
57              if (node == null)
58                      return "";
59              return traverseInOrder(node.left) + " "
60                      + node + " "
61                      + traverseInOrder(node.right);
62      }
63  }
```

```
1   import java.util.Scanner;
2
3   public class BinaryTreeDemo {
4       public static void main (String[] args) {
5               /* Create a binary tree which holds strings */
6               BinaryTree<String> bTree = new BinaryTree<String>();
7
8               /* Get a line of input */
9               System.out.print("Enter a sentence : ");
10              String sentence = (new Scanner(System.in)).nextLine();
11
12              /* Only retain letters */
13              sentence = sentence.toUpperCase().replaceAll("[^A-Z]", " ");
14
```

```
15              /* Insert each word into the tree. This implicitly sorts them. */
16              for (String word : sentence.split("\\s+"))
17                      bTree.add(word);
18
19              /* In order traversal of the tree */
20              System.out.print("Sorted words : ");
21              System.out.println(bTree);
22      }
23  }
```

Variable Description

TreeNode<T>		
T	item	The data stored in the node
TreeNode<T>	left	Reference to the left child of this
TreeNode<T>	right	Reference to the right child of this
BinaryTree<T>		
TreeNode<T>	root	The root node of the binary tree
BinaryTree<T>::contains(T)		
T	item	The item to check for
BinaryTree<T>::search(T)		
T	item	The item to search for
BinaryTree<T>::add(T)		
T	item	The item to be added
BinaryTree<T>::search(TreeNode<T>, T)		
TreeNode<T>	root	The current node being checked
T	item	The item to search for
BinaryTree<T>::add(TreeNode<T>, T)		
TreeNode	root	The current node being compared
T	item	The item to be added

"One should always play fairly when one has the winning cards."

— **Oscar Wilde**

Problem 27 Simulate a deck of playing cards.

Solution A deck of cards can be simulated by a list of 'Card' objects. A playing card is wholly defined by its *suit*, of which there are 4, and its *rank*, of which there are 12. A standard deck contains 52 cards, such that every permutation of suit and rank is present. Cards can only be dealt from a deck, or shuffled in the deck.

There are many algorithms for shuffling a list, but the simplest is the *Knuth shuffle*, also known as the *Fisher-Yates shuffle*. It involves choosing a random card from the list, putting it aside, then repeating until the list is exhausted. This generates an *unbiased permutation* of the list.

`Card (suit:Suit, rank::Rank)`

1. Copy `suit` and `rank` as constants into the object.
2. **Return** the resultant object.

`Deck ()`

1. Create a stack of `Card` objects of capacity 52.
2. For each ordered pair (`s`,`r`) ∈`Suit->values()`×`Rank->values()`:
   (a) Create a new `Card`, pass it `s` and `r`, and add it to the card stack.
3. **Define** the functions:
   (a) `Deck::deal()`
   (b) `Deck::shuffle()`
4. **Return** the resultant object.

`Deck::deal ()`

1. If there are no cards in the stack, **return** a `null` object.
2. Pop a card from the stack and **return** it.

`Deck::shuffle ()`

1. Let there be n cards in the stack.
2. For each `i` $\in \{n-1, n-2, \ldots, 1\}$:
   (a) Let `j` be a random integer such that $0 \leq$`j`$\leq$`i`.
   (b) Swap the cards at indices `i` and `j` in the stack.

Source Code

```
/* List all possible suits */
public enum Suit {
	SPADES,
	HEARTS,
	DIAMONDS,
	CLUBS;
}
```

```
/* List all possible ranks, along with their equivalent numeric values */
public enum Rank {
	ACE    (1),
	TWO    (2),
	THREE  (3),
	FOUR   (4),
	FIVE   (5),
	SIX    (6),
	SEVEN  (7),
	EIGHT  (8),
	NINE   (9),
	TEN    (10),
	JACK   (11),
	QUEEN  (12),
	KING   (13);

	protected int value;

	Rank (int value) {
		this.value = value;
	}

	public int getValue () {
		return this.value;
	}
}
```

```
/* Abstraction of a standard playing card */
public class Card {
	/* Each card has an immutable suit and rank */
	public final Suit suit;
	public final Rank rank;

	/* Short names of cards */
	public static final String rankShort = " A 2 3 4 5 6 7 8 910 J Q K";
```

```
9
10          public Card (Suit suit, Rank rank) {
11                  this.suit = suit;
12                  this.rank = rank;
13          }
14
15          /* Formats the card details neatly */
16          @Override
17          public String toString () {
18                  return rank + " of " + suit;
19          }
20
21          /* Formats the card as a 2-character string */
22          public String toStringShort () {
23                  int r = rank.getValue();
24                  String rs = rankShort.substring(2 * r, 2 * (r + 1)).trim();
25                  char ss = suit.toString().charAt(0);
26                  return rs + ss;
27          }
28  }
```

```
1   /* Abstraction of a deck of cards */
2   public class Deck {
3           /* Setup a simple stack */
4           protected Card[] cards;
5           protected int top;
6
7           public Deck () {
8                   cards = new Card[52];
9                   top = -1;
10                  /* Initialize a full deck */
11                  for (Suit suit : Suit.values())
12                          for (Rank rank : Rank.values())
13                                  cards[++top] = new Card(suit, rank);
14          }
15
16          /* Checks if the deck is empty */
17          public boolean isEmpty () {
18                  return top < 0;
19          }
20
21          /* Returns the number of cards in the deck */
22          public int size () {
23                  return top + 1;
24          }
```

```
25
26          /* Pops the topmost card from the deck */
27          public Card deal () {
28                  if (this.isEmpty())
29                          return null;
30                  return cards[top--];
31          }
32
33          /* Shuffles the deck using the Fisher-Yates, or Knuth shuffle */
34          public void shuffle () {
35                  for (int i = top; i > 0; i--) {
36                          int j = random(0, i + 1);
37                          swap(i, j);
38                  }
39          }
40
41          /* Utility method for swapping cards in the deck */
42          private void swap (int i, int j) {
43                  Card t = cards[i];
44                  cards[i] = cards[j];
45                  cards[j] = t;
46          }
47
48          /* Format the cards in the deck neatly */
49          @Override
50          public String toString () {
51                  if (this.isEmpty())
52                          return "[]";
53                  String s = "[";
54                  for (int i = top; i >= 0; i--)
55                          s += cards[i].toStringShort() + ", ";
56                  return s.substring(0, s.length() - 2) + "]";
57          }
58
59          /* Utility method for generating random integers in a given range */
60          private static int random (int lo, int hi) {
61                  return (int) (lo + (Math.random() * (hi - lo)));
62          }
63  }
```

```
1   public class DeckDemo {
2           public static void main (String[] args) {
3                   /* Create a new deck of cards in standard order */
4                   Deck d = new Deck();
5                   System.out.println(d);
```

```
6
7               /* Shuffle the deck */
8               d.shuffle();
9               System.out.println(d);
10
11              /* Deal out 26 cards */
12              for (int i = 0; i < 26; i++)
13                      System.out.println(d.deal());
14
15              /* Show the deck */
16              System.out.println(d);
17      }
18  }
```

Variable Description

Card		
Suit	suit	The suit of the playing card
Rank	rank	The rank of the playing card

Deck		
Card[]	cards	The stack of cards making up the deck
int	top	The index of the card at the top of the stack
Suit	suit	The suit of the playing card being added
Rank	rank	The rank of the playing card being added

Deck::shuffle()		
int	i, j	The indices of the cards to be swapped

Deck::swap(int, int)		
int	i, j	The indices of the cards to be swapped

"Code never lies, comments sometimes do."

— **Ron Jeffries**

Problem 28 Remove all comments from given source code.

Solution Java comments can be classified into two broad types — single line comments beginning with the sequence '`//`' and ending with a newline, and multiple line comment beginning with the sequence '`/*`' and ennding with the sequence '`*/`'. Care must be taken to ignore such sequences within quotes *both single and double*, as well as within other comments. Escape sequences also have to be dealt with.

While parsing the given source code character by character, it becomes necessary to keep track of a *state variable*. This will store information about what is currently being parsed, and different sets of checks are executed accordingly. Java *enums*, or *enumerated lists*, are ideal for this purpose.

`main (filename:String)`

1. Create a `ReadSourceFile` object called `s`, and pass it `filename` and a buffer size of 10.
2. Declare a state variable called `currentState`, and set it to `SOURCE`.
3. Declare a character called `matchingQuotes`, and set it to a black space.
4. **While** `s->hasNextChar()`:
   (a) Store the character returned by `s->getChar()` as `c`.
   (b) If `c` is a backslash, display it, get another character from `s->getChar()`, display that, and jump to (4).
   (c) If `currentState` is `SOURCE`:
      i. If `c` is a quotation mark, set `currentState` to `QUOTES`, set `matchingQuotes` to `c`. Display `c` and jump to (4).
      ii. If `c` is a forward slash, get another character called `n` from `s->getChar()`.
         A. If `n` is an asterisk, set `currentState` to `MULTIPLE_LINE_COMMENT`.
         B. If `n` ia another forward slash, set `currentState` to `SIMGLE_LINE_COMMENT`.
         C. If none of the above, call `s->putChar(n)`, display `c` and jump to (4).
      iii. If none of the above, display `c` and jump to (4).
   (d) If `currentState` is `SIMGLE_LINE_COMMENT` and `c` is a newline, set `currentState` to `SOURCE`, display `c` and jump to (4).
   (e) If `currentState` is `MULTIPLE_LINE_COMMENT` and `c` is an asterisk:
      i. Get another character called `n` from `s->getChar()`.

ii. If `n` is a forward slash, set `currentState` to `SOURCE`.
iii. Jump to (4).

(f) If `currentState` is `QUOTES` and `c` is equal to `matchingQuotes`, set `currentState` to `SOURCE`, `matchingQuotes` to an blank space. Display `c` and jump to (4).

(g) If none of the above, display `c`.

`ReadSourceFile (filename:String, bufferSize:integer)`

1. Initialize a new `FileReader` *unbuffered* called `fileReader` and pass it `filename`.
2. Create a simple *buffer* of integers, implemented using a stack. *This will store characters, but the* ***`char`*** *data type cannot store special characters, such as the character which indicates the end of a file.*
3. **Define** the functions:
   (a) `ReadSourceFile::hasNextChar()`
   (b) `ReadSourceFile::getChar()`
   (c) `ReadSourceFile::putChar(c)`
4. **Return** the resultant object.

`ReadSourceFile::hasNextChar ()`

1. Read a new character from `fileReader`, and call it `c`.
2. If `c` is equal to -1, **return** `false`.
3. Call `this->putChar(c)`
4. **Return** `true`

`ReadSourceFile::getChar ()`

1. If the buffer has some characters, pop one off and **return** it.
2. Read a character from `fileReader` and **return** it.

`ReadSourceFile::putChar (c:Integer)`

1. If the buffer has space, push `c` onto it and **return** `true`. Otherwise, **return** `false`.

Source Code

```
/* List of possible states */
public enum State {
	SOURCE, SINGLE_LINE_COMMENT, MULTIPLE_LINE_COMMENT, QUOTES;
}
```

```
import java.io.IOException;
import java.io.FileReader;

public class ReadSourceFile {
	protected String filename;

	/* Setup a simple stack as a buffer */
	protected int[] buffer;
	protected int top;

	/* Use a FileReader to collect input */
	protected FileReader fileReader;

	/* Sets the filename and the buffer size */
	public ReadSourceFile (String filename, int bufferSize) throws IOException {
		this.filename = filename;
		this.buffer = new int[bufferSize];
		this.top = -1;
		this.fileReader = new FileReader(filename);
	}

	/* Checks whether there are more characters */
	public boolean hasNextChar () throws IOException {
		/* Read a character */
		int c = fileReader.read();
		if (c == -1)
			return false;
		/* Push the character onto the buffer */
		putChar(c);
		return true;
	}

	/* Returns the next character in the file */
	public int getChar () throws IOException {
		/* Pop from the buffer */
		if (top >= 0)
			return buffer[top--];
		/* Read directly from file */
		return fileReader.read();
```

```
        }

        /* Pushes a character onto the buffer */
        public boolean putChar (int c) {
                /* Check for stackoverflow */
                if (top == (buffer.length - 1))
                        return false;
                buffer[++top] = c;
                return true;
        }

        /* Close all resources */
        public void close () throws IOException {
                fileReader.close();
        }
}
```

```
import java.io.IOException;

public class RemoveComments {
        public static void main (String[] args) throws IOException {
                /* Parse first command line argument as the file to read from.
                   Allocate a small buffer */
                ReadSourceFile s = new ReadSourceFile(args[0], 10);

                /* Initialize the current state to plain source */
                State currentState = State.SOURCE;
                /* Initialize the current matching quote to empty */
                char matchingQuotes = ' ';

                /* Loop through all characters */
                while (s.hasNextChar()) {
                        /* Get a character from the file */
                        char c = (char) s.getChar();

                        /* Escaped characters - display the backslash and the
                            following character */
                        if (c == '\\') {
                                System.out.print(c + "" + ((char) s.getChar()));
                                continue;
                        }
                        switch (currentState) {
                                case SOURCE:
                                        switch (c) {
                                                /* Single and double opening quotes */
```

```
                        case '\"':
                        case '\'':
                                /* Set the new state */
                                currentState = State.QUOTES;
                                /* Set the matching closing quote */
                                matchingQuotes = c;
                                System.out.print(c);
                                break;
                                /* Possible comment */
                        case '/':
                                char n = (char) s.getChar();
                                if (n == '*')
                                        currentState =
                                            State.MULTIPLE_LINE_COMMENT;
                                else if (n == '/')
                                        currentState =
                                            State.SINGLE_LINE_COMMENT;
                                else {
                                        s.putChar(n);
                                        System.out.print(c);
                                }
                                break;
                        default:
                                System.out.print(c);
                }
                break;
        case SINGLE_LINE_COMMENT:
                /* Exit state to plain source on newline */
                if (c == '\n') {
                        currentState = State.SOURCE;
                        System.out.print(c);
                }
                break;
        case MULTIPLE_LINE_COMMENT:
                /* Exit state to plain source on closing
                    characters */
                if (c == '*') {
                        char n = (char) s.getChar();
                        if (n == '/')
                                currentState = State.SOURCE;
                }
                break;
        case QUOTES:
                /* Exit state on encountering closing quote */
                if (c == matchingQuotes) {
                        currentState = State.SOURCE;
```

```
71                                          matchingQuotes = ' ';
72                                      }
73                                      /* Display anything in quotes verbatim */
74                                      System.out.print(c);
75                                      break;
76                                  default:
77                                      System.out.print(c);
78                          }
79                  }
80                  s.close();
81          }
82  }
```

Variable Description

ReadSourceFile		
String	filename	The file containing the source code to be read
int[]	buffer	The stack of characters read from the file
int	top	The index of the character at the top of the buffer

RemoveComments::main(String[])		
ReadSource File	s	The source file reader
State	currentState	Indicates the type of code currently being parsed
char	matchingQuotes	Indicates the type of ending quote which pairs with the opening quote, if currently inside a string in the source code
char	c, n	Stores the current and next characters in the source code being parsed

"A program that produces incorrect results twice as fast is infinitely slower."

— **John Ousterhout**

Problem 29 Compare the runtimes of the following sorting algorithms — *bubble sort*, *insertion sort* and *quicksort*.

Solution *Bubble sort* is a sorting algorithm which repeatedly steps through an unsorted list, compares adjacent elements and swaps them if they are in the wrong order. It has an average time complexity of $O(n^2)$.

Insertion sort is a sorting algorithm which builds a sorted list one element at a time by repeatedly selecting an unsorted element and inserting it into the correct position in the sorted portion. It too has an average time complexity of $O(n^2)$

Quicksort is a *divide and conquer* sorting algorithm which splits an unsorted list along a pivot, with elements less than it shifted before and elements greater than it shifted after. The two halves are then sorted recursively. This algorithm has an average time complexity of $O(n \log n)$.

Each of these algorithms have different strengths and weaknesses. *Insertion sort* and *bubble sort* perform progressively slower than *quicksort* on long lists with a large spread of randomly shuffled numbers. On the other hand, *insertion sort* performs faster than *bubble sort*, which in turn performs faster than quicksort on shorter lists with randomly shuffled numbers. Again, *bubble sort* performs faster than *insertion sort*, which performs significantly faster on long lists with a small spread of numbers, i.e., almost sorted lists.

`BubbleSorter::sort (a:Integer[])`

1. Initialize an integer `right` to the length of `a`.
2. Initialize a boolean `swapped` to `true`.
3. **While** `swapped`:
   (a) Set `swapped` to `false`
   (b) For $\texttt{i} \in \{1, 2, \ldots, right - 1\}$:
      i. If `a[i - 1]` $>$ `a[i]`:
         A. Swap the elements in `a` at indices `i-1` and `i`.
         B. Set `swapped` to `true`.
   (c) Decrement `right`.

`InsertionSorter::sort (a:Integer[])`

1. Let n be thenumber of elements in `a`.
2. For `i` $\in \{1, 2, \ldots, n-1\}$:
   (a) Set an integer `k` to `a[i]`.
   (b) Set an integer `j` to `i - 1`.
   (c) **While** (`j` ≥ 0) and (`a[j]` $>$ `k`):
      i. Set `a[j + 1]` to `a[j]`.
      ii. Decrement `j`.
   (d) Set `a[j + 1]` to `k`.

`QuickSorter::sort (a:Integer[])`

1. Let `l` be the number of elements in `a`.
2. Call `this->sort(a, 0, l - 1)`

`QuickSorter::sort (a:Integer[], lo:Integer, hi:Integer)`

1. If `hi` $\leq$ `lo`, **return**.
2. Call `this->partition(a, lo, hi)`, and store the returned integer as `pivot`.
3. Call `this->sort(a, lo, pivot - 1)`
4. Call `this->sort(a, pivot + 1, hi)`

`QuickSorter::partition (a:Integer[], lo:Integer, hi:Integer)`

1. Set an integer `pivotValue` to `a[hi]`.
2. Set an integer `pivot` to `lo - 1`.
3. For `i` $\in$ {`lo`,`lo + 1`,...,`hi - 1` }:
   (a) If `a[i]` $\leq$ `pivotValue`:
      i. Increment `pivot`.
      ii. Swap the elements in `a` at indices `i` and `pivot`.
4. Increment `pivot`.
5. Swap the elements in `a` at indices `i` and `pivot`.
6. **Return** `pivot`

Source Code

```
/* Abstract integer array sorter */
public abstract class IntegerArraySorter {
	/* Each sorter has a common sort method */
	public abstract void sort (int[] a);

	/* Utility method for swapping elements in an array */
	public static void swap (int[] a, int i, int j) {
		int t = a[i];
		a[i] = a[j];
		a[j] = t;
	}
}
```

```
public class BubbleSorter extends IntegerArraySorter {
	@Override
	public void sort (int[] a) {
		int right = a.length;
		boolean swapped = true;
		while (swapped) {
			swapped = false;
			for (int i = 1; i < right; i++) {
				if (a[i - 1] > a[i]) {
					swap(a, i - 1, i);
					swapped = true;
				}
			}
			right -= 1;
		}
	}

	@Override
	public String toString () {
		return "BubbleSort";
	}
}
```

```
public class InsertionSorter extends IntegerArraySorter {
	@Override
	public void sort (int[] a) {
		for (int i = 1; i < a.length; i++) {
			int k = a[i];
			int j = i - 1;
			while ((j >= 0) && (a[j] > k)) {
```

```
                        a[j + 1] = a[j];
                        j -= 1;
                    }
                    a[j + 1] = k;
                }
        }

        @Override
        public String toString () {
                return "InsertionSort";
        }
}
```

```
public class QuickSorter extends IntegerArraySorter {
        @Override
        public void sort (int[] a) {
                sort(a, 0, a.length - 1);
        }

        /* Recursive quicksort */
        private void sort (int[] a, int lo, int hi) {
                if (hi <= lo)
                        return;
                int pivot = partition(a, lo, hi);
                sort(a, lo, pivot - 1);
                sort(a, pivot + 1, hi);
        }

        /* Lomuto partition scheme */
        private int partition (int[] a, int lo, int hi) {
                int pivotValue = a[hi];
                int pivot = lo - 1;
                for (int i = lo; i < hi; i++)
                        if (a[i] <= pivotValue)
                                swap(a, i, ++pivot);
                swap(a, hi, ++pivot);
                return pivot;
        }

        @Override
        public String toString () {
                return "QuickSort";
        }
}
```

```
public class SortCompare {
	public static void main (String[] args) {
		/* Initialize the sorters */
		IntegerArraySorter[] sorters = {
			new BubbleSorter(),
			new InsertionSorter(),
			new QuickSorter()
		};

		/* Parse the first command line argument as the length of the list */
		int length = Integer.parseInt(args[0]);

		/* Parse the second command line argument as the upper bound of
		    integers in the list */
		int range = Integer.parseInt(args[1]);

		/* Create a random list */
		int[] a = randomArray(length, range);

		long t0 = 0, t1 = 0;
		for (IntegerArraySorter s : sorters) {
			/* Clone the list */
			int[] b = a.clone();

			/* Start the timer */
			t0 = System.nanoTime();

			/* Sort */
			s.sort(b);

			/* Stop the timer */
			t1 = System.nanoTime();

			System.out.printf("%16s : %16d ns\n", s, t1 - t0);
		}
	}

	/* Generates a random array of given length and given upper limit of random
	    numbers */
	public static int[] randomArray (int length, int hi) {
		int[] a = new int[length];
		for (int i = 0; i < length; i++)
			a[i] = (int) (Math.random() * hi);
		return a;
	}
}
```

Variable Description

`IntegerArraySorter::sort(int[])`		
`int[]`	`a`	The array whose elements are to be sorted

`IntegerArraySorter::swap(int[], int, int)`		
`int[]`	`a`	The array whose elements are to be swapped
`int`	`i, j`	The indices of the elements to be swapped

`BubbleSorter::sort(int[])`		
`int[]`	`a`	The array whose elements are to be sorted
`int`	`right, i`	Counter variables
`boolean`	`swapped`	Keeps track of whether any swaps were performed in the current iteration

`InsertionSorter::sort(int[])`		
`int[]`	`a`	The array whose elements are to be sorted
`int`	`i, j`	Counter variables
`int`	`k`	The element to be inserted

`QuickSorter::sort(int[])`		
`int[]`	`a`	The array whose elements are to be sorted

`QuickSorter::sort(int[], int, int)`		
`int[]`	`a`	The array whose elements are to be sorted
`int`	`lo, hi`	The lower and upper indices of the unsorted list
`int`	`pivot`	The index of the value about which the list is partitioned

`QuickSorter::partition(int[], int, int)`		
`int[]`	`a`	The array whose elements are to be sorted
`int`	`lo, hi`	The lower and upper indices of the unsorted list
`int`	`pivotValue`	The value about which the list is partitioned
`int`	`pivot`	The index of the value about which the list is partitioned
`int`	`i`	Counter variable

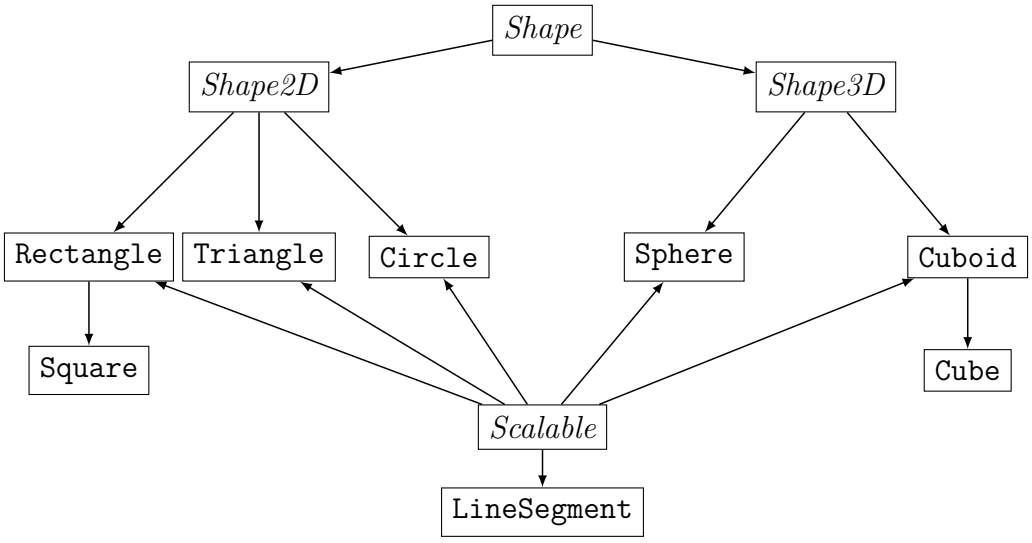
"Design is choosing how you will fail."

— Ron Fein

Problem 30 Showcase *class inheritance* in *Java*, by designing a hierarchy of geometric shapes.

Solution Here, we use an interface `Shape` as the superclass of the interfaces `Shape2D` and `Shape3D`, each of which has subclasses sharing common behaviour. For example, all $2D$ shapes have computable areas and perimeters, while all $3D$ shapes have computable volumes and surface areas. This structure illustrates *multilevel inheritance*.

All these shapes can be *scaled*, i.e., their dimensions can be changed by some factor. This behaviour is defined by the interface `Scalable`, which the above classes all implement. This structure illustrates *multiple inheritance*. The class `LineSegment` is also `Scalable`, despite not being a `Shape`.



Following is a general implementation of a shape '`MyShape`', which has a computable property '`myProperty`' and is `Scalable`.

`MyShape (parameters... :Number)`

1. Copy each `parameter` as an immutable constant into the object data.
2. **Define** the functions:
   (a) `MyShape::getMyProperty()`
   (b) `MyShape::scale(scaleFactor())`
3. **Return** the resultant object.

`MyShape::getMyProperty ()`

1. Compute `myProperty` using the `parameters`, and **return** the result.

`MyShape::scale (scaleFactor:FloatingPoint)`

1. Create a new `MyShape`, whose `parameters` are the parameters of `this` multiplied by `scaleFactor`, and `return` it.

Source Code

```
1  public interface Scalable<T> {
2         public T scale (double scaleFactor);
3  }
```

```
1  public class LineSegment implements Scalable<LineSegment> {
2         protected final double length;
3
4         public LineSegment (double length) {
5                this.length = length;
6         }
7
8         @Override
9         public LineSegment scale (double scaleFactor) {
10               return new LineSegment(length * scaleFactor);
11        }
12
13        @Override
14        public String toString () {
15               return String.format("LineSegment (length = %f)", length);
16        }
17 }
```

```
1  public interface Shape {}
```

```
1  public interface Shape2D extends Shape {
```

```
        public double getArea ();
        public double getPerimeter ();
}
```

```
public class Circle implements Shape2D, Scalable<Circle> {
        protected final double radius;

        public Circle (double radius) {
                this.radius = radius;
        }

        @Override
        public double getArea () {
                return Math.PI * radius * radius;
        }

        @Override
        public double getPerimeter () {
                return 2 * Math.PI * radius;
        }

        @Override
        public Circle scale (double scaleFactor) {
                return new Circle(radius * scaleFactor);
        }

        @Override
        public String toString () {
                return String.format("Circle (radius = %f)", radius);
        }
}
```

```
public class Triangle implements Shape2D, Scalable<Triangle> {
        protected final double a;
        protected final double b;
        protected final double c;

        public Triangle (double a, double b, double c) {
                this.a = a;
                this.b = b;
                this.c = c;
        }

        @Override
        public double getArea () {
```

```
14              double s = (a + b + c) / 2.0;
15              return Math.sqrt(s * (s - a) * (s - b) * (s - c));
16      }
17
18      @Override
19      public double getPerimeter () {
20              return a + b + c;
21      }
22
23      @Override
24      public Triangle scale (double scaleFactor) {
25              return new Triangle(a * scaleFactor, b * scaleFactor, c *
                    scaleFactor);
26      }
27
28      @Override
29      public String toString () {
30              return String.format("Triangle (sides = {%f, %f, %f})", a, b, c);
31      }
32  }
```

```
1   public class Rectangle implements Shape2D, Scalable<Rectangle> {
2       protected final double length;
3       protected final double breadth;
4
5       public Rectangle (double length, double breadth) {
6               this.length = length;
7               this.breadth = breadth;
8       }
9
10      @Override
11      public double getArea () {
12              return length * breadth;
13      }
14
15      @Override
16      public double getPerimeter () {
17              return 2 * (length + breadth);
18      }
19
20      @Override
21      public Rectangle scale (double scaleFactor) {
22              return new Rectangle(length * scaleFactor, breadth * scaleFactor);
23      }
24
```

```
25          @Override
26          public String toString () {
27                  return String.format("Rectangle (length = %f, breadth = %f)", length,
                        breadth);
28          }
29  }
```

```
1   public class Square extends Rectangle {
2           public Square (double side) {
3                   super(side, side);
4           }
5
6           @Override
7           public Square scale (double scaleFactor) {
8                   return new Square(length * scaleFactor);
9           }
10
11          @Override
12          public String toString () {
13                  return String.format("Square (side = %f)", length);
14          }
15  }
```

```
1   public interface Shape3D extends Shape {
2           public double getVolume ();
3           public double getSurfaceArea ();
4   }
```

```
1   public class Sphere implements Shape3D, Scalable<Sphere> {
2           protected final double radius;
3
4           public Sphere (double radius) {
5                   this.radius = radius;
6           }
7
8           @Override
9           public double getVolume () {
10                  return 4.0 * Math.PI * radius * radius * radius / 3.0;
11          }
12
13          @Override
14          public double getSurfaceArea () {
15                  return 4 * Math.PI * radius * radius;
16          }
17
```

```
18          @Override
19          public Sphere scale (double scaleFactor) {
20                  return new Sphere(radius * scaleFactor);
21          }
22
23          @Override
24          public String toString () {
25                  return String.format("Sphere (radius = %f)", radius);
26          }
27  }
```

```
1   public class Cuboid implements Shape3D, Scalable<Cuboid> {
2           protected final double length;
3           protected final double breadth;
4           protected final double height;
5
6           public Cuboid (double length, double breadth, double height) {
7                   this.length = length;
8                   this.breadth = breadth;
9                   this.height = height;
10          }
11
12          @Override
13          public double getVolume () {
14                  return length * breadth * height;
15          }
16
17          @Override
18          public double getSurfaceArea () {
19                  return 2.0 * ((length * breadth) + (breadth * height) + (height *
                        length));
20          }
21
22          @Override
23          public Cuboid scale (double scaleFactor) {
24                  return new Cuboid(length * scaleFactor, breadth * scaleFactor, height
                        * scaleFactor);
25          }
26
27          @Override
28          public String toString () {
29                  return String.format("Cuboid (length = %f, breadth = %f, height =
                        %f)", length, breadth, height);
30          }
31  }
```

```java
public class Cube extends Cuboid {
	public Cube (double side) {
		super(side, side, side);
	}

	@Override
	public Cube scale (double scaleFactor) {
		return new Cube(length * scaleFactor);
	}

	@Override
	public String toString () {
		return String.format("Cube (side = %f)", length);
	}
}
```

```java
public class ShapeDemo {
	public static void main (String[] args) {
		/* Shapes of all kinds can be stored under the same type - Shape */
		Shape[] shapes = {
			new Circle(1.0),
			new Cube(2.0),
			new Triangle(3.0, 4.0, 5.0)
		};

		/* Each shape overrides the toString() method */
		System.out.println("Shapes :");
		for (Shape s : shapes)
			System.out.println(s);
		System.out.println();

		/* 2D shapes can be stored under the same type - Shape2D */
		Shape2D[] flatShapes = {
			new Circle(1.0),
			new Triangle(1.0, 1.0, 1.0),
			new Square(1.0)
		};

		/* Each shape overrides the getArea() and getPerimeter() methods */
		System.out.println("2D Shapes :");
		for (Shape2D s2D : flatShapes)
			System.out.printf("%-66s area = %4f perimeter = %8f\n",
					s2D,
					s2D.getArea(),
					s2D.getPerimeter());
```

```
30              System.out.println();
31
32              /* Scalable shapes can be stored under the same type - Scalable */
33              Scalable[] scalable = {
34                      new LineSegment(1.0),
35                      new Sphere(1.0),
36                      new Cuboid(1.0, 2.0, 3.0)
37              };
38
39              /* Each scalable shape overrides the scale() method */
40              System.out.println("Scalable :");
41              for (Scalable sc : scalable)
42                      System.out.printf("%-66s scaled by 3 is %-66s\n", sc,
                            sc.scale(3));
43              System.out.println();
44      }
45  }
```

Variable Description

LineSegment		
double	length	The length of the line segment

Circle		
double	radius	The radius of the circle

Triangle		
double	a, b, c	The lengths of the sides of the triangle

Rectangle		
double	length, breadth	The dimensions of the rectangle

Sphere		
double	radius	The radius of the sphere

Cuboid		
double	length, breadth, height	The dimensions of the cuboid

"If brute force doesn't solve your problems, then you aren't using enough."

— **Anonymous**

Problem 31 Spell out a given number in words.

Solution In English, digits are grouped in sets of 3, with the first digit representing the number of 'hundreds', the second representing the number of 'tens', and the third representing the number of 'ones'. Each set is given a suffix such as 'thousand', 'million', 'billion', and so on. A special case exists for the two digit numbers 'eleven' to 'nineteen'.

Digits following a decimal point are simply spelt out in succession.

`main (number:String)`

1. Assert that `number` can be parsed as a floating point number.
2. Call and display `numberToWords(number)`.
3. **Exit**

`numberToWords (number:String)`

1. Split `number` into an `integerPart` and a `decimalPart` along the decimal point (`.`).
2. Replace `integerPart` with `stringToWords(integerPart)`.
3. If there is a decimal part, replace `decimalPart` with `stringToDigits(decimalPart)`. Otherwise, **return** `integerPart`.
4. **Return** `integerPart + "point" + "decimalPart"`

`stringToDigits (number:String)`

1. Initialize an empty string `s`.
2. For each character `c` in `number`:
   (a) Convert `c` to its corresponding digit `d`.
   (b) Append the English word for `d` to `s`.
3. **Return** `s`

`stringToWords (number: String)`

1. If `number` starts with a minus sign (`-`), remove it and **return** `"minus" + stringToWords(number)`.
2. Initialize an empty string `s`.
3. Initialize a counter `blockNumber` to zero.

4. **While** `number` is non-empty:
   (a) Remove a block of three characters from `number`, and store them as an integer `temp`.
   (b) If `temp` is non-zero, add `threeDigitsToWords(temp)` and the English word for the power of thousand correspinding to `blockNumber` to the beginning of `s`.
   (c) Increment `blockNumber`.
5. If `s` is empty, **return** `"zero"`.
6. **Return** `s`

`threeDigitsToWords (n:Integer)`

1. Store the first, second, and third digits of `n` as integers `h`, `t`, and `o` respectively.
2. Initialize an empty string `s`.
3. If `h` is non-zero, append its corresponding English word and the word `"hundred"` to `s`.
4. If `t` is `1`, append the corresponding English word for the last two digits of `n` *(which are in the 'teens')* to `s` and **return** it.
5. Append the English word for the multiple of ten corresponding to `t` to `s`.
6. If `o` is non-zero, append its corresponding English word to `s`.
7. **Return** `s`

Source Code

```
1  public class NumberToWords {
2         /* Map of single digits to words */
3         public static final String[] singleDigits = {
4                 " zero",
5                 " one",
6                 " two",
7                 " three",
8                 " four",
9                 " five",
10                " six",
11                " seven",
12                " eight",
13                " nine"
14        };
15
16        /* Map of numbers in the 'teens' to words */
17        public static final String[] twoDigits = {
18                " ten",
19                " eleven",
```

```
20              " twelve",
21              " thirteen",
22              " fourteen",
23              " fifteen",
24              " sixteen",
25              " seventeen",
26              " eighteen",
27              " nineteen"
28      };
29
30      /* Map of multiples of tens into words */
31      public static final String[] tenMultiples = {
32              "",
33              "",
34              " twenty",
35              " thirty",
36              " forty",
37              " fifty",
38              " sixty",
39              " seventy",
40              " eighty",
41              " ninety"
42      };
43
44      /* Map of suffixes of powers of thousand into words */
45      public static final String[] thousandPowerGroups = {
46              "",
47              " thousand",
48              " million",
49              " billion",
50              " trillion",
51              " quadrillion",
52              " quintillion",
53              " sextillion",
54              " septillion",
55              " octillion",
56              " nonillion",
57              " decillion"
58      };
59
60      public static void main (String[] args) {
61              try {
62                      /* Parse the first command line argument as the number
63                         to be spelt out */
64                      Double.parseDouble(args[0]);
65                      System.out.println(numberToWords(args[0]));
```

```
66              } catch (IndexOutOfBoundsException e) {
67                      System.out.println("Enter 1 argument! ([number])");
68              } catch (NumberFormatException e) {
69                      System.out.println("Invalid number!");
70              }
71      }
72
73      /* Convert a string of digits into words */
74      public static String numberToWords (String n) {
75              /* Deal with the integral and fractional parts separately */
76              String parts[] = n.split("\\.");
77              String integerPart = stringToWords(parts[0]);
78              /* Check for the fractional part */
79              if (parts.length == 1)
80                      return integerPart.trim();
81              String decimalPart = stringToDigits(parts[1]);
82              return (integerPart + " point" + decimalPart).trim();
83      }
84
85      /* Convert the digits of the fractional part into words */
86      public static String stringToDigits (String digits) {
87              String s = "";
88              for (int i = 0; i < digits.length(); i++) {
89                      /* Map digits to their corresponding words */
90                      int d = digits.charAt(i) - '0';
91                      s += singleDigits[d];
92              }
93              return s;
94      }
95
96      /* Convert the digits of the integral part into words */
97      public static String stringToWords (String n) {
98              /* Negative sign is simply read off as 'minus' */
99              if (n.charAt(0) == '-')
100                     return "minus" + stringToWords(n.substring(1));
101             String s = "";
102             int left = Math.max(0, n.length() - 3);
103             int blockNumber = 0;
104             /* Loop through blocks of three */
105             while (n.length() > 0) {
106                     String temp = n.substring(left);
107                     int blockOfThree = Integer.parseInt(temp);
108                     if (blockOfThree != 0) {
109                             s = threeDigitsToWords(blockOfThree)
110                                     + thousandPowerGroups[blockNumber]
111                                     + "," + s;
```

```
112                     }
113                     blockNumber++;
114                     /* Cut off evaluated part */
115                     n = n.substring(0, left);
116                     left = Math.max(0, left - 3);
117             }
118             /* Special case */
119             if (s.equals(""))
120                     return "zero";
121             return s.substring(0, s.length() - 1);
122     }
123
124     /* Convert a block of three digits into words */
125     public static String threeDigitsToWords (int n) {
126             /* Extract each digit */
127             int h = n / 100;
128             int t = (n / 10) % 10;
129             int o = n % 10;
130             String s = "";
131             /* Only convert the 'hundreds' if it is non-zero */
132             if (h > 0) {
133                     s += singleDigits[h] + " hundred";
134             }
135             /* Special case of 'teens' */
136             if (t == 1) {
137                     s += twoDigits[o];
138                     return s;
139             }
140             s += tenMultiples[t];
141             /* Only convert 'ones' if it is non-zero */
142             if (o > 0) {
143                     s += singleDigits[o];
144             }
145             return s;
146     }
147 }
```

Variable Description

NumberToWords		
String[]	singleDigits	Map of English words corresponding to single digits
String[]	twoDigits	Map of English words corresponding to two digit numbers in the 'teens'
String[]	tenMultiples	Map of English words corresponding to multiples of ten
String[]	thousand PowerGroups	Map of English words corresponding to powers of thousand

NumberToWords::numberToWords(String)		
String	n	The number to be spelt out
String[]	parts	Stores the integer and fractional parts of number
String	integerPart	The integer part in words
String	decimalPart	The digits after the decimal in words

NumberToWords::stringToDigits(String)		
String	digits	The string of digits to be spelt out
String	s	digits in words
int	i	Counter variable
int	d	The current digit to be spelt out

NumberToWords::stringToWords(String)		
String	n	The integer to be spelt out
int	left	The left index of the current block of three
int	blockNumber	Counter variable, stores the current block number
String	temp	Stores the current block of three
int	blockOfThree	Stores the current block of three as an integer

NumberToWords::threeDigitsToWords(int)		
int	n	The integer to be spelt out
int	h	The first digit of n
int	t	The second digit of n
int	o	The third digit of n

"Please, Oh please, publish me in your collection of self-referential sentences!"

— Douglas Hofstadter

Problem 32 A *quine* is a non-empty computer program which takes no input and produces a copy of its own source code as its only output.

Write a *quine* in *Java*.

(Note that a program which finds its source code file and displays it is not considered a quine, since it takes a file as input.)

Hofstadter (`xkcd.com/917`)

Solution The name *quine* was coined by *Douglas Hofstadter* in his brilliant book *Gödel, Escher, Bach: An Eternal Golden Braid*, in honour of the philosopher *Willard Van Orman Quine*, who extensively studied indirect self reference, in particular the following statement known as *Quine's paradox*.

> "Yields falsehood when preceded by its quotation" yields falsehood when preceded by its quotation.

Although writing a *quine* in *Java* seems impossible at first glance, it can be shown that *quines* exist in any *Turing complete* programming language.

We might start off by writing the following code.

```
1 public class Quine {
2       public static void main (String[] args) {
3             System.out.println("public class Quine {\n\t ... System.out.println(
                  ??? )}");
4       }
5 }
```

A problem arises — what can we write in place of `???` ? This part of the string must contain the entire string itself. Is this possible without the string being infinitely long?

The problem is that the string we seek must contain the characters to be printed, and also be able to be used to print itself. The following code snippet illustrates this.

```
1 String s = "???";
2 System.out.println(???);
```

What can replace `???` so that the entirety of line 1 is displayed?

A solution is as follows.

```
1 String s = "String s = ";
2 System.out.println(s + '"' + s + '"' + ';');
```

We can now use this template to move the entirety of the code into the string, including the print statement itself. This leads to another problem — double quotes are now inside double quotes, and must be escaped (`\"`). However, the backslashes themselves will not appear in the output. This can be solved by using the ASCII value for an double quote, which is `34`, in place of an escaped double quote. Discarding newlines and delcaring the string `s` as a global variable at the very end of the program minimizes the amount of code considerably.

The result is the following *quine*.

```
1 public class Quine { public static void main (String[] args) { char q = 34;
    System.out.println(s + q + s + q + ';' + '}'); } public static String s =
    "public class Quine { public static void main (String[] args) { char q = 34;
    System.out.println(s + q + s + q + ';' + '}'); } public static String s = ";}
```

Variable Description

Quine		
`String`	`s`	Stores the entire source code of the program
`Quine::main()`		
`char`	`q`	Stores a double quote

This project was compiled with XeLaTeX.

All files involved in the making of this project can be found at
https://github.com/sahasatvik/Computer-Project/tree/master/ISC

Satvik Saha
sahasatvik@gmail.com
https://sahasatvik.github.io