COMPSCI 105 SS

Principles of Computer Science

It is with great pleasure that we welcome you all to COMPSCI 105 SS. This course is conducted during the summer of 2004.

This is an intensive course that will be taught in six weeks. It has twelve week duration of lectures when taught during the first and second semesters. As we are covering the same material, it is important that you do your work consistently and not fall behind.

This booklet contains the course outline and tutorial questions. While every effort was made to provide accurate information in this course book, it is possible that some small mistakes may not have been corrected, or that some policies need to be changed. Be sure to monitor the course webpage for updates to the course book. We wish to acknowledge Dr. Kevin Novins for providing invaluable advice, lecture notes and other course material from last year's COMPSCI 105 SS.

Santokh Singh
Ivan Surya
Jonathan Teutenberg
Kelly Ting Yang

20 December 2003

Table of Contents

Course Overview and Policies **3**

Course Schedule *4*
Components of the course and learning resources *5*
Assessment *7*
Assignment Policy *8*
Policy on Cheating and Plagiarism *9*
Required Reading Materials *10*
How to seek assistance *14*
What to do about missed lectures/labs *14*
Course Personnel *14*

Lecture Notes **15**

Tutorials **42**

Numbers in the Computer **63**

Big and Small numbers *64*
Representing information in a machine *64*

Data Representation **66**

Data Representation in Memory *67*
Arithmetic *73*
Using bits as bits (not as parts of numbers) *81*
Representation of Characters *84*
Floating Point numbers *90*

Course Overview and Policies

Course schedule

	Monday	**Tuesday**	**Wednesday**	**Thursday**	**Friday**
10.00 – 11.00	Lec	Lec	Lec	Lec	Lec
11.00 – 12.00	Tut1	Tut1	Tut2	Tut2	
12.00 -13.00					
13.00 – 14.00					
14.00 – 15.00	Tut1		Tut2		
15.00 – 16.00					

Note:

1. The lectures are to be held in Eng3404.
2. Tutorials are to be held in the GTL (Ground floor Tutorial Lab).
3. *Tut1* and *Tut2* correspond to *Lab* and *Tut* in nDeva.
4. Each tutorial session is two hours with a ten minute break in the middle.
5. Each student should attend 2 different tutorials in accordance to their enrolment every week except in weeks 1 and 4, which are disrupted by Orientation Day and a Public Holiday.

Components of the course and learning resources

Textbook

The primary reference for the course is the textbook "Data abstraction and Problem Solving with Java: Walls and Mirrors" by Carrano and Prichard. The exact page numbers in the textbook covered in the course are listed on Page 13 in this course book. You are expected to have easy access to a textbook, and to bring a textbook to all tutorials. We expect most people to buy the textbook, but it may be practical for you to share a copy with a classmate.

Lectures

Lectures are designed to help you to understand the readings by providing explanations and by giving you a chance to ask questions and to work on simple problems. Outlines for each lecture and related book pages are given on Pages 15 in this course book. You will probably get the most out of a lecture if you try reading the relevant pages before each lecture. However, you could instead use the lectures to prepare you for your reading.

Tutorials

Tutorials provide a more personal and flexible environment for learning. You will mostly be working on your own, but tutors will be available to answer questions. There will be at least one tutor for every fifteen students.

Every tutorial has new readings associated with it. At tutorials, we will provide exercises for you to work on to help you to gauge your level of understanding. These exercises will also help you to develop your practical skills – there is a computer at every desk in the tutorial room. Tutorial exercises are published in the course book on Pages 42.

Tutorials are different from lectures in that they are assessed. See Page 7 for details of assessment.

The Forum

An electronic discussion forum for the course is provided at http://forums.cs.auckland.ac.nz. Staff and students participate. For students, the forum can be a powerful aid to learning. Participate by reading the posts, asking questions, and helping others in appropriate ways.

Posts to the forum must be course-related and must not be abusive. You also must be careful not to give away answers or partial answers to assessments before their final due dates. See the course plagiarism policy on Page 9 for details. **Disciplinary action will be taken if these rules are violated.**

If you are extremely uncomfortable with divulging your identify on the forum, you can post with an alias. However, we hope that most people will be willing to sign their names to their postings.

Sample Questions

Each tutorial and lecture has sample questions associated with them. It is best to do the related readings and attempt to understand them before trying the questions. That way, the questions are a good test of your understanding. If you go about it the other way, and choose what to read and understand based on the questions, you are likely to end up with significant gaps in your knowledge and understanding.

Sample answers to sample questions will not be released. However, if there is intelligent discussion of sample questions on the course forum, course staff will join in. Do not discuss tutorial questions on the forum until the last time that tutorial is offered. See the course timetable for details.

If you need additional questions with answers, check the course text book. Every chapter has "Self-Test Exercises". The answers to these are published on Pages 775-789 of the text book.

Sample Student Solutions

We won't be releasing model solutions to assessed material – tutorials, tests, and assignments. We will, however, be posting samples of student solutions on the web after each assignment is due. These can be the starting point for discussion of solutions on the forum.

Assessment

The course is assessed with nine tutorial sessions, a mid-term test, three assignments and an exam. It is officially classed as a practical course by the Science Faculty and in order to pass a student must pass both the practical component of the course (the tutorial sessions and the assignments) and the theoretical component (the test and exam). Pass rates are not set until after the final exam, so you should sit the exam even if you are unsure whether or not you have passed the practical part of the course.

The details of the different components are listed as follows:

- Tutorials: 8 %
  Assessment of each tutorial is based on participation. You have to make an honest attempt to complete all the exercises of each tutorial in order to get the mark. Your answers to the exercises need not be completely correct. At the end of each tutorial, the tutors will hand out a single-page answer sheet. You need to fill in selected answers as required in the sheet and hand it back to the tutors to record your mark. You have to finish all the exercises within your own tutorial session. No late attempts are accepted. There are 9 tutorials in total but you only have to complete 8 of them for full marks. The marks are allocated as in the following table.

Tutorial Mark Allocation

Completed Tutorials	9/9	8/9	7/9	6/9	5/9	4/9	3/9	2/9	1/9	0/9
Mark	8	8	7	6	5	4	3	2	1	0

- Assignments: 17%
  - Assignment 1: 3%,
    due at 4.00pm Friday, 9th January,
  - Assignment 2: 7%,
    due at 4.00pm Friday, 23th January,
  - Assignment 3: 7%,
    due at 4.00pm Thursday, 5th February,
- Test: 10%
- Exam: 65%

Assignment policy

- Read the assignment handout carefully and understand what is required, and what is **NOT** required to accomplish in the assignment. Be aware that some assignments provide skeleton code that contains detailed explanations and sample outputs. Make sure you download the specified code from the course assignment web page, and read through it carefully.
- All the assignments are related to the contents covered in the lectures and tutorials. However, some may require extra reading, self-learning, and research. Make sure you provide detailed citations when you refer to other resources.
- If you feel unsure about some part of the assignment, you can refer to the teaching staff or write down any reasonable assumptions in a README.txt file or as required.
- You should always produce your own test cases to test your implementations. Write them down in the README.txt file or as required. Note that some assignments allocate marks for this.
- Make sure your program compiles without modification on the lab machines and that you include all necessary files with your submission. You will lose marks if the markers cannot compile your code. In extreme cases you may lose all the marks for the program tasks.
- Only submit the required files. The written tasks should be typed in a plain text document or a Microsoft Word document.
- All the assignments are submitted through the Assignment Drop Box electronically. Make an early attempt to avoid problems that might happen near the due time.
- All assignments are individual assignments. See Page 9 for the policy on cheating and plagiarism.
- All assessment takes place after assignments are handed in. Do not ask course staff what mark you will get for a particular answer before the due date.
- You are responsible for protecting your assignment. Take reasonable steps to avoid theft: Don't leave your workstation unattended and be extremely careful with floppy disks and ZIP disks.

Policy on cheating and plagiarism

There are no group assignments in this course. All assignments are to reflect your own understanding of the course material. You may discuss high-level aspects of the assignments with others, but you may not share code or particular answers.

You will be given a zero for an assignment if you are caught copying all or part of your assignment work, or for showing all or part of your work to others, thus enabling them to copy (this includes postings on the forum). Two pieces of code that are the same except for insignificant details, such as variable and method names will still be recognised as copies. Paraphrased lines of text will also be recognised as copies.

Severe or repeated instances of cheating will result in further disciplinary action.

The test and exam are closed book. You may not bring notes or calculators. The test and exam are designed to assess your individual understanding. You may not copy from another student's paper or arrange your materials to make it easy for others to copy. If you are caught, you will be given a zero. See the University Calendar for further examination regulations.

All cheating incidents will be recorded in the department's register of suspicious activity.

Required reading materials

a) By Week

Week 1:

Lecture 1. Numbers in Computer
- Course overview and policies on Pages 3-14
- Numbers in the Computer and Data Representation on Pages 63-65, 66-68

Lecture 2. Bit Twiddling
- Data Representation on Pages 68-71, 73-75, 81-85

Lecture 3. Software Engineering and Abstraction
- Data Representation on Pages 81-85
- Carrano and Prichard, Pages 4-15

Lecture 4. Programming Style
- Carrano and Prichard, Pages 16-18, 22-40

Tutorial 1. COMPSCI101 Revision

Week 2:

Lecture 5. Introduction to Recursion
- Carrano and Prichard, Pages 48-55

Lecture 6. Recursion
Carrano and Prichard, Pages 55-68, 85-91

Lecture 7. Searching via Recursion
Carrano and Prichard, Pages 76-85

Lecture 8. Abstract Data Types
Carrano and Prichard, Pages 106-122

Lecture 9. Implementing ADTs
Carrano and Prichard, Pages 124-137

Tutorial 2. Data Representation.
Numbers in the Computer on Pages 64, Data Representation on Pages 67-78, 81-83

Tutorial 3. Data Representations, Exceptions and Input/Output.
Data Representation on Pages 84-93, 135-137,728-739
Note: also read how to convert binary fractions to decimal and vice versa, i.e. Data Representation on Pages 78-79

Week 3:

Lecture 10. References and Objects
Carrano and Prichard, Pages 152-159

Lecture 11. Linked Lists
Carrano and Prichard, Pages 159-178, 181-186

Lecture 12. Variations of Linked Lists
Carrano and Prichard, Pages 178-181, 186-191

Lecture 13. Stacks
Carrano and Prichard, Pages 248-266

Lecture 14. Queues
Carrano and Prichard, Pages 298-315

Tutorial 4. Software life-cycle, Abstraction and Recursion
Carrano and Prichard, Pages 5-18, 22-40, 48-95

Tutorial 5. ADTs, Exception, Interface and an array-based implementation of ADT list.
Carrano and Prichard, Pages 16-18, 106-144

Week 4:

Lecture 15. Big O Notation
Carrano and Prichard, Pages 372-380

Lecture 16. Searching and Sorting
Carrano and Prichard, Pages 380-388

Lecture 17. Merge Sort
Carrano and Prichard, Pages 393-398

Lecture 18. Quicksort
Carrano and Prichard, Pages 398-410

Tutorial 6. Linked lists and Doubly linked lists
Carrano and Prichard, Pages 151-203

Week 5:

Lecture 19. Trees
Carrano and Prichard, Pages 422-427, 429-432, 437-444, 482-483

Lecture 20. Binary Search Tree
Carrano and Prichard, Pages 432-434, 452-461

Lecture 21. Binary Search Tree Delete
Carrano and Prichard, Pages 461-474

Lecture 22. Efficiency of BST Operations
Carrano and Prichard, Pages 427-429, 474-482

Tutorial 7. Stacks, Queues, Sorting and Introduction to Efficiency.
Carrano and Prichard, Pages 247-332,388-393

Tutorial 8. Big-O and Complexity, Trees, Binary Trees and Tree Traversals.
Carrano and Prichard, Pages 372-384, 421-452

Week 6:

Lecture 23. Tables and Priority Queues
Carrano and Prichard, Pages 498-521

Lecture 24. Heap Implementation
Carrano and Prichard, Pages 521-530, 436-437

Lecture 25. Introduction to Hashing
Carrano and Prichard, Pages 578-586, 589-592

Lecture 26. Analysis of Hashing
Carrano and Prichard, Pages 586-588, 595-598

Tutorial 9. Binary Search Trees, Heaps and Heapsort.
Carrano and Prichard, Pages 452-482, 517-535

b) By Chapter

Numbers in the Computer 63-65

Data Representation 66-71, 73-79, 81-93

Chapter 1: 4-18, 22-40

Chapter 2: 48-95

Chapter 3: 106-122, 124-144

Chapter 4: 152-194

Chapter 5: Nil

Chapter 6: 248-266

Chapter 7: 298-315

Chapter 8: 352-359

Chapter 9: 372-410

Chapter 10: 422-434, 436-447, 452-483

Chapter 11: 498-534

Chapter 12: 578-592, 595-598

Appendix A: 728-736

How to seek assistance

If you have questions about the course material, you should first talk to your classmates or approach the tutors during tutorials or the lecturer during the office hours listed on the course webpage. You can also ask questions on the course's electronic forum.

If you have concerns about the marking of an assignment, see your marker.

If you have concerns about the course in general, feel free to approach the lecturer or one of the tutors, either in office hours or via e-mail. We appreciate constructive feedback and will respond as soon as possible and as best as we can.

If you don't feel comfortable talking with the course staff directly, you can approach one of the class representatives. Their names and contact details will be posted on the course web page once they are elected. Further information on sources of assistance is available in the Department's Handbook and from the Student Association.

What to do about missed lectures/labs

If you miss a lecture, you can catch up by doing the reading associated with it. If you have trouble following the reading, you should find a classmate who attended the lecture and who can tell you what you missed. You also should ask a classmate if you missed any important announcements.

If you miss a tutorial, you should try to complete it on your own. Again, ask classmates who did attend for help. Unfortunately, you won't be able to make up the assessment component on missed labs.

Course Personnel

Santokh Singh
Course Coordinator and Lecturer
santokh@cs.auckland.ac.nz

Ivan Surya
Tutor
isur004@ec.auckland.ac.nz

Jonathan Teutenberg
Tutor
jteu004@cs.auckland.ac.nz

Kelly Ting Yang
Tutor
tyan023@ec.auckland.ac.nz

Lecture Notes

Numbers in the Computer

Lecture 1
6 January 2004

Lecturer: Santokh Singh

Topics:

- Introduction to the Course
  - Lecturer
  - Course overview
  - Teaching Style
  - Policies
- Numbers in the Computer
  - Magnitude prefixes (kilo-, mega-, etc.)
  - Representing information in a machine
  - Binary representation of integers
- * Input/Output and Exception Handling (Assignment 1).

Related Reading:

Course Overview and Policies on Pages 3-14
Numbers in the Computer on Pages 63-65
Data Representation on Pages 66-68

Sample Questions:

L1.1 Approximately, how fast does ultra-violet electromagnetic waves travel in cm per hour through vacuum?

L1.2 If there were 3,652,797,543 birds in the largest country on earth, how many bits would it take to give each of these birds a unique identifier? Using that many bits, how many left over bit patterns would we have?

L1.3 During a test, a student was given 100 marks to the base of 2 instead of in the decimal system. How many marks did he actually get in the decimal system?

L1.4 A 19 year old bridegroom weighed 217 kg on his wedding day. What are these values in binary?

Bit Twiddling

Lecture 2
7 January 2004

Lecturer: Santokh Singh

Topics:

- Numbers in the Computer
  - Converting from decimal to binary and vice-versa
  - Octal representation of integers
  - Hexadecimal representation of integers
- Binary Arithmetic
  - Addition
  - Subtraction
- Using Bits as Bits
  - Logical Operations

Related Reading:

Data Representation on Pages 68-71, 73-75, 81-85

Sample Questions:

L2.1 An octopus has 3 spots on each of its tentacles. If it looses a tentacle during a dispute with a shark, state the number of spots remaining in the octal system?

L2.2 Convert $1{,}372_{10}$ to base 9.

L2.3 Perform the addition $291_{10} + 316_{10}$ in binary.

L2.4 Perform the subtraction $FAB_{16} - BED_{16}$ in binary.

L2.5 Convert 283_{10} to (i) base 2, (ii) base 8, (iii) base 16.

L2.6 Perform the addition $121_8 + 143_8$ in binary.

L2.7 Perform the addition $AB_{16} + BA_{16}$ in binary.

Software Engineering and Abstraction

Lecture 3
8 January 2004

Lecturer: Santokh Singh

Topics:

- Using Bits as Bits
  - Logical Operations
  - Masking and Shifting
- Parity
- Representation of Characters
- Software Life Cycle
  - Basic Stages
  - Specification and Design via Preconditions and Postconditions
  - Verification via Loop Invariants
- Costs Associated with Software

Related Reading:

- Data Representation on Pages 81-85
- Carrano and Prichard, Pages 4-15

Sample Questions:

L3.1 Explain how to divide a positive binary integer by 16 using only masking and shifting. Explain how to compute the remainder of the division using only masking and shifting.

L3.2 A friend transmits the number 10011 and says the corresponding even parity bit is 1. Can you tell if the data has been corrupted? Explain.

L3.3 In ASKME code, the numbers 1-26 represent the letters 'A'-'Z' (case is ignored) and the number 0 represents all other characters. Write Java code to convert ASCII code to ASKME code.

L3.4 Carrano and Prichard, Chapter 1, Exercise 2, Page 42.

L3.5 On Page 12 of the textbook, Carrano and Pritchard write that "software does not wear out if you neglect it." Is this really true? Can you think of an instance in which software really does wear out when neglected?

Programming Style

Lecture 4
9 January 2004

Lecturer: Santokh Singh

Topics:

- Abstraction
  - Modular Design
  - Procedural Abstraction
  - Data Abstraction
  - Abstract Data Types (ADT's)
  - Information Hiding
- Facilitating Changes
  - Modularity
  - Methods
  - Named Constants
  - Readability
  - Documentation
  - Fail-Safe Programming
- Debugging with `System.out.println`

Related Reading:

Carrano and Prichard, Pages 16-18, 22-40

Sample Questions:

L4.1 Carrano and Prichard, Chapter 1, Exercise 4, Page 43.

L4.2 Carrano and Prichard, Chapter 1, Exercise 8, Page 43.

L4.3 Dig up a program that you wrote for COMPSCI 101. Can you still make sense of it? If not, why not? If so, see if you can write the preconditions and postconditions for one of the methods and the invariants for one of the loops.

L4.4 Sometimes too much of a good thing is a bad thing. Think of ways that you could carry the "rules" of style described in Chapter 1 to extremes, resulting in a shockingly bad program. See if you can follow the rules of style, but still make the program in L4.2 very hard to read.

L4.5 If you change a method so that it tests its preconditions, are its preconditions still preconditions? Explain.

Introduction to Recursion

Lecture 5
12 January 2004

Lecturer: Santokh Singh

Topics:

- Introduction to Recursion
  - Searching in a dictionary
  - Computing factorials
- Aspects of Recursive Algorithms
- Recurrence Relations

Related Reading:

Carrano and Prichard, Pages 48-55

Sample Questions:

L5.1 Describe a recursive algorithm for calculating n! (the factorial of n), where n is a whole number.

L5.2 Compare the code on Page 53 of Carrano and Prichard to the code on Page 32. Which do you think is easier to understand? Why?

L5.3 Let sum(k) be the sum of all integers 0..k, where k is a positive integer. Write a recurrence relation that defines sum(k). What is the base case? Now write recursive Java code to compute sum(k).

Recursion

Lecture 6
13 January 2004

Lecturer: Santokh Singh

Topics:

Tracing Recursive Programs
Recursion for `void` methods
The Towers of Hanoi

Related Reading:

Carrano and Prichard, Pages 55-68, 85-91

Sample Questions:

L6.1 Carrano and Prichard, Chapter 2, Exercise 1, Page 97.

L6.2 Carrano and Prichard, Chapter 2, Exercise 5, Page 97.

L6.3 Carrano and Prichard, Chapter 2, Exercise 6, Page 97. (Hint: At every step you'll need to divide the integer by 10 and compute the remainder of the integer when divided by 10).

L6.4 Carrano and Prichard, Chapter 2, Exercise 7a, Page 97.

Searching via Recursion

Lecture 7
14 January 2004

Lecturer: Santokh Singh

Topics:

- Searching in a sorted array
  - Binary Search
- Searching in an unsorted array
  - Finding the smallest item
  - Finding the k^{th} smallest item

Related Reading:

Carrano and Prichard, Pages 76-85

Sample Questions:

L7.1 Carrano and Prichard, Chapter 2, Exercise 13, Page 100.

L7.2 Carrano and Prichard, Chapter 2, Exercise 14, Page 100.

L7.3 Carrano and Prichard, Chapter 2, Programming Problem 2, Page 103.

Abstract Data Types

Lecture 8
15 January 2004

Lecturer: Santokh Singh

Topics:

- Abstract Data Types (ADTs)
  - Definition
  - Advantages
  - Examples
  - Specifying ADTs
  - Combining ADTs

Related Reading:

Carrano and Prichard, Pages 106-122

Sample Questions:

L8.1 Carrano and Prichard, Chapter 3, Exercise 4, Page 147.

L8.2 Carrano and Prichard, Chapter 3, Exercise 10, Page148.

L8.3 Think of situations in the real world in which you pay someone to do something for you and you care not only *that* they do it, but also *how* they do it. Do such situations occur in computing? If so, might ADTs be a bad idea?

Implementing Abstract Data Types

Lecture 9
16 January 2004

Lecturer: Santokh Singh

Topics:

- Implementing Abstract Data Types
  - Interfaces
  - Classes
    - Private vs. Public
    - Constructors
    - Other Methods
  - Exceptions and Inheritance

Related Reading:

Carrano and Prichard, Pages 124-137

Sample Questions:

L9.1 Carrano and Prichard, Chapter 3, Programming Problem 3, Page 149.

L9.2 Carrano and Prichard, Chapter 3, Programming Problem 5, Page 150.

References and Objects

Lecture 10
19 January 2004

Lecturer: Santokh Singh

Topics:

- References and Objects
  - Object Storage
  - Object References
  - Diagramming Object and References
  - Garbage Collection
  - Example: Equality
  - Example: Parameter Passing
  - Example: Resizable Arrays

Related Reading:

Carrano and Prichard, Pages 152-159

Sample Questions:

L10.1 What is the output of the following sequence of statements?

```
int a = 3;
int b = a;
b = 4;
System.out.println(a);
int[] c = {3};
int[] d = c;
d[0] = 4;
System.out.println(c[0]);
```

L10.2 Draw a series of diagrams, similar to those found on Page 155 of Carrano and Prichard, to explain Sample Question L10.1.

L10.3 Explain how to change the code on Pages 156 and 157 of the textbook to fix the `changeNumber` method.

Linked Lists

Lecture 11
20 January 2004

Lecturer: Santokh Singh

Topics:

- Linked Lists
  - The Node class
  - Lists as Recursive Structures
  - Displaying a Linked List
  - Inserting and deleting elements
  - Implementing the ADT List

Related Reading:

Carrano and Prichard, Pages 159-178, 181-186

Sample Questions:

L11.1 Carrano and Prichard, Chapter 4, Exercise 1, Page 204.

L11.2 Carrano and Prichard, Chapter 4, Exercise 2, Page 204.

L11.3 Counting the number of items in a linked list can be done either iteratively or recursively. Which method do you prefer? Explain why.

L11.4 Carrano and Prichard, Chapter 4, Exercise 6, Page 205.

Variations of Linked Lists

Lecture 12
21 January 2004

Lecturer: Santokh Singh

Topics:

- Linked Lists
  - Compared to Arrays
  - Passing Linked Lists as Parameters
- Variations of the Linked List
  - Tail References
  - Circular Linked Lists
  - Dummy Head Nodes

Related Reading:

Carrano and Prichard, Pages 178-181, 186-191

Sample Questions:

L12.1 Carrano and Prichard, Chapter 4, Exercise 8, Page 205.

L12.2 Explain why the question "Which is better, an array implementation of a List ADT or a linked list implementation of an ADT" does not have a simple answer.

L12.3 Carrano and Prichard, Chapter 4, Exercise 9, Page 205.

L12.4 Carrano and Prichard, Chapter 4, Exercise 10, Page 205.

L12.5 Carrano and Prichard, Chapter 4, Exercise 12, Page 205.

Stacks

Lecture 13
22 January 2004

Lecturer: Santokh Singh

Topics:

- Stacks
  - Basic ADT Operations
  - Application: Undo Operations
  - Application: Balancing Parenthesis
  - Array Implementation
  - Linked List Implementation
  - Implementing a ADT Stack with a ADT List

Related Reading:

Carrano and Prichard, Pages 248-266

Sample Questions:

L13.1 Carrano and Prichard, Chapter 6, Exercise 3, Page 288.

L13.2 Carrano and Prichard, Chapter 6, Exercise 4, Page 288.

L13.3 Carrano and Prichard, Chapter 6, Exercise 6, Page 289.

L13.4 An ADT List can do everything an ADT Stack can do, and more. Why, then, is it useful to define an ADT Stack at all?

Queues

Lecture 14
23 January 2004

Lecturer: Santokh Singh

Topics:

- Queues
  - Basic ADT Queue Operations
  - "Circular" Array Implementation
  - Linked List Implementation
  - "Circular" Linked List Implementation
  - Implementing an ADT Queue with an ADT List

Related Reading:

- Carrano and Prichard, Pages 298-315

Sample Questions:

- L14.1 Carrano and Prichard, Chapter 7, Exercise 1, Page 327.
- L14.2 Carrano and Prichard, Chapter 7, Exercise 4, Page 327.
- L14.3 Carrano and Prichard, Chapter 7, Programming Problem 1, Page 329.
- L14.4 Carrano and Prichard, Chapter 7, Programming Problem 3, Pages 329.

Big O Notation

Lecture 15
27 January 2004

Lecturer: Santokh Singh

Topics:

- Algorithm Efficiency
  - Challenges in Measuring Algorithm Efficiency
  - Counting Operations
  - Growth Rates of Functions
  - Big O Notation
    - Examples
    - General Rules

Related Reading:

Carrano and Prichard, Pages 372-380

Sample Questions:

L15.1 Carrano and Prichard, Chapter 9, Exercise 1(a) - 1(f), Page 415.

L15.2 Carrano and Prichard, Chapter 9, Exercise 2, Page 416.

Searching and Sorting

Lecture 16
28 January 2004

Lecturer: Santokh Singh

Topics:

- Algorithm Efficiency
  - Analysing Search Algorithms
  - Limits of Big O Analysis
- Sorting
  - Selection Sort

Related Reading:

Carrano and Prichard, Pages 380-388

Sample Questions:

L16.1 Carrano and Prichard, Chapter 9, Exercise 1(g), Page 415.

L16.2 Carrano and Prichard, Chapter 9, Exercise 7, Page 415.

L16.3 Carrano and Prichard, Chapter 9, Exercise 9, Page 415 (selection sort only).

L16.4 Carrano and Prichard, Chapter 9, Exercise 11, Page 415 (selection sort only).

Merge Sort

Lecture 17
29 January 2004

Lecturer: Santokh Singh

Topics:

- Sorting
  - Merge Sort
    - Algorithm
    - Analysis

Related Reading:

Carrano and Prichard, Pages 393-398

Sample Questions:

L17.1 Carrano and Prichard, Chapter 9, Exercise 13, Page 417.

L17.2 Write a version of merge sort that operates on linked lists rather than arrays. How does its efficiency compare to the array version?

Quicksort

Lecture 18
30 January 2004

Lecturer: Santokh Singh

Topics:

- Sorting
  - Quicksort
    - Algorithm
    - Analysis

Related Reading:

Carrano and Prichard, Pages 398-410

Sample Questions:

L18.1 Carrano and Prichard, Chapter 9, Exercise 14, Page 417.

L18.2 Carrano and Prichard, Chapter 9, Exercise 16, Page 417. With this modification, show that it is still possible to get $O(n^2)$ behaviour with quicksort.

L18.3 Starting with the code given in the textbook, implement the quicksort variation described in Carrano and Prichard, Chapter 9, Exercise 17, Page 417. Run timing tests for large random arrays and see if the variation is any faster in practice.

Trees

Lecture 19
2 February 2004

Lecturer: Santokh Singh

Topics:

- Trees
  - Introduction
  - General Tree Structures
  - Height
  - Binary Trees
  - Reference-Based Implementations

Related Reading:

Carrano and Prichard, Pages 422-427, 429-432, 437-444, 482-483

Sample Questions:

L19.1 Carrano and Prichard, Chapter 10, Programming Problem 3, Page 492.

L19.2 Give a non-recursive definition of the height of a tree.

L19.3 Assume you know that at most k nodes can be stored in a binary tree of height h. What is the maximum number of nodes that can be stored in a binary tree of height h+1?

L19.4 What is the maximum number of nodes that can be stored in a binary tree of height 9?

Binary Search Trees

Lecture 20
3 February 2004

Lecturer: Santokh Singh

Topics:

- Trees
  - Preorder Traversal
  - PostorderTraversal
  - Inorder Traversal
- Binary Search Trees
  - Introduction
  - Insertion
  - Searching

Related Reading:

Carrano and Prichard, Pages 432-434, 452-461

Sample Questions:

L20.1 Carrano and Prichard, Chapter 10, Exercise 2, Page 486.

L20.2 Carrano and Prichard, Chapter 10, Exercise 3, Page 486.

L20.3 Carrano and Prichard, Chapter 10, Exercise 4, Page 486.

L20.4 Carrano and Prichard, Chapter 10, Exercise 25, Pages 490-491.

L20.5 How many binary search trees have the preorder traversal A D B E C? Sketch them.

Binary Search Tree Delete

Lecture 21
4 February 2004

Lecturer: Santokh Singh

Topics:

Binary Search Trees
Delete

Related Reading:

Carrano and Prichard, Pages 461-474

Sample Questions:

L21.1 Insert the numbers 3, 1, 2, 6, 4, 5 into a binary search tree in the order listed. Draw the final result. Now delete the nodes in the same order that they were inserted, drawing the result after every deletion.

L21.2 Carrano and Prichard, Chapter 10, Exercise 10, Page 487.

L21.3 Carrano and Prichard, Chapter 10, Exercise 11, Page 488.

L21.4 Carrano and Prichard, Chapter 10, Exercise 28, Page 491.

L21.5 Some people argue that the binary search tree delete algorithm should alternate between calling `deleteLeftmost` and `deleteRightmost` rather than always calling one or the other. Why?

Efficiency of BST Operations

Lecture 22
5 February 2004

Lecturer: Santokh Singh

Topics:

- Trees
  - Full Trees
  - Complete Trees
  - Balanced Trees
- Binary Search Trees
  - Efficiency of Operations
  - Treesort
  - Rebalancing Trees

Related Reading:

Carrano and Prichard, Pages 427-429, 474-482

Sample Questions:

L22.1 Draw all possible binary trees with three nodes in them. What per cent of these have minimum height?

L22.2 Carrano and Prichard, Chapter 10, Exercise 5, Page 487.

L22.3 Suppose you have a binary search tree of minimum height with n nodes. What is the big *O* complexity of adding a new node and rebalancing the tree so that it still has minimum height?

L22.4 Sketch a binary tree that is balanced, but not complete.

Tables and Priority Queues

Lecture 23
9 February 2004

Lecturer: Santokh Singh

Topics:

- ADT Table
  - Basic Operations
  - Linear Implementation
  - Nonlinear Implementation
  - Comparing Implementations
- ADT Priority Queue
  - Basic Operations
  - Heaps

Related Reading:

Carrano and Prichard, Pages 498-521

Sample Questions:

L23.1 Carrano and Prichard, Chapter 11, Exercise 2, Page 536.

L23.2 Carrano and Prichard, Chapter 11, Exercise 3, Page 536.

L23.3 Carrano and Prichard, Chapter 11, Exercise 6, Page 537.

L23.4 Draw all possible maxheaps with the values 1, 2, and 3. How many are there? How many of them are valid binary search trees?

Heap Implementation

Lecture 24
10 February 2004

Lecturer: Santokh Singh

Topics:

- ADT Priority Queue
  - Basic Operations
  - Heaps
    - Complete Binary Tree Representation
    - Implementing Insert
    - Implementing Delete
- Trees
  - Storing Complete Binary Trees in Arrays

Related Reading:

Carrano and Prichard, Pages 521-530, 436-437

Sample Questions:

L24.1 Show the state of a maxheap at each stage as the elements 3, 1, 2, 4, 5 are added.

L24.2 Carrano and Prichard, Chapter 11, Exercise 11, Page 537.

L24.3 Redraw the rightmost heap in Carrano and Prichard Figure 11-12 on Page 525 after a delete operation has been performed on it.

L24.4 Carrano and Prichard, Chapter 11, Exercise 15, Page 538.

Introduction to Hashing

Lecture 25
11 February 2004

Lecturer: Santokh Singh

Topics:

- Hashing
  - Basic Concepts
  - Hash Functions
  - Collision Resolution
    - Open Addresing
      - Linear Probing
    - Separate Chaining

Related Reading:

Carrano and Prichard, Pages 578-586, 589-592

Sample Questions:

L25.1 Insert the numbers 123, 343, 125, 342, 333 into a hash table with 10 elements and the hash function "$h(x) = x \bmod 10$" and sketch the results using
(a) Linear Probing
(b) Separate Chaining

L25.2 Carrano and Prichard, Chapter 12, Exercise 10, Page 607.

L25.3 Carrano and Prichard, Chapter 12, Exercise 11, Page 607.

Analysis Of Hashing and Revision

Lecture 26
12 February 2004

Lecturer: Santokh Singh

Topics:

- Hashing
  - Open Addressing
    - Quadratic Probing
    - Double Hashing
  - What constitutes a good hash function?
  - Implementing the ADT Table with Hashing

Related Reading:

Carrano and Prichard, Pages 586-588, 595-598

Sample Questions:

L26.1 Insert the numbers 123, 343, 125, 342, 333 into a hash table with 10 elements and the hash function "$h(x) = x \bmod 10$" and sketch the results using
(c) Quadratic Probing
(d) Double Hashing with "$h_2(x) = 17x^2 \bmod 10$"

L26.2 Carrano and Prichard, Chapter 12, Exercise 12, Page 607.

Tutorials

Department of
Computer Science

COMPSCI 105 SS – Tutorial One

Question & Answer Sheet

Introduction:

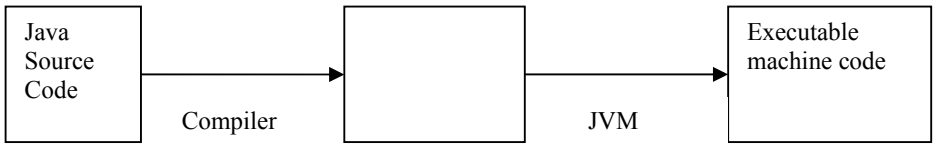
In this paper, you will use Java as the programming language to implement most of the ideas introduced in the lectures and also tutorials. Hence, you need to refresh your mind on how to write programs using Java programming language, which is taught in COMPSCI 101, which is a prerequisite for COMPSCI 105.

This tutorial is designed to refresh your mind on some of the important topics on COMPSCI 101, which will be necessary for you to know in order to understand the material that will be taught in COMPSCI 105.

Questions:

A. How Java works.

1. What is a compiler?
2. What is a Java Virtual Machine?
3. Complete the following diagram:



4. In a Java application, the code in which method will first be executed when the program is run?
5. What is a class?

B. Primitive and Object Variables.

1. What kinds of primitive variables are there?
2. What is the difference between a primitive variable and an Object variable?

C. Simple Methods .

1. Complete the following method which return the sum of the two integers a and b passed as parameters to the method:

```
public ___  calculateSum( ___ a, ___ b){
        return _______;
}
```

D. Conditional Statements.

1. Complete the following method which assign a grade to a student's mark. The following grades are assigned to the following marks:
   Less than 50 = D.
   More than or equal to 50 and less than 70 = C.
   More than or equal to 70 and less than 90 = B.
   More than or equal to 90 = A.

   ```
   public String computeGrade(int mark){

   }
   ```

E. Loops.

1. Change the following for loop into a while loop.

   ```
   for (int i = 0; i<30; i++){
         System.out.println(i);
   }
   ```

F. Collection of Elements (ArrayList and Arrays), String and StringTokenizer.

1. The words that follows the name of the Java program are stored in the String array args[] and can be accessed from the main() method.
   For example, if the name of the program is StringCollection and you type "java StringCollection Principles of Computer Science"
   You can refer to the Strings that follow the program name as follow:
   args[0] = "Principles"
   args[1] = "of"
   args[2] = "Computer"
   args[3] = "Science"
   You can also use args.length to find out the length of the array, i.e. the number of words that follow the program name.
   Complete the following main method which should store these Strings that follow the program name in an ArrayList called stringArrayList and in an array called stringArray.

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

   }
   ```

2. Complete the following method which prints out the length of the Strings in an ArrayList.

   ```
   public void printLength(ArrayList al){

   }
   ```

3. Write a method that takes a sentence in a String and print out each word to a line. You should use a StringTokenizer.

   public void printWord(String sentence){

   }

G. Debugging.

For this section, you have to:

1. Identify the error in the following pieces of code as either:
   a. Correct (if there is no error).
   b. Compile time error.
   c. Runtime error.
   d. Logic error.
2. Fix these simple errors.

1. String number = 23;
   Error Type:_____
   Fix:_____

2. String v = "The sum of 5 and 9 is "+ 5 + 9;
   Error Type:_____
   Fix:_____

3. int k = Integer.parseInt("35");
   Error Type:_____
   Fix:_____

4. int j = 35 / (4/5);
   Error Type:_____
   Fix:_____

5. String s = new String "5";
   Error Type:_____
   Fix:_____

H. Desk-checking and output.

Desk-check the following code and indicate the output. You should indicate clearly how each method is called, the passed parameters and the returned value of each method. You can use the technique from 101 lectures.

```
public class Test{

    public static void main(String[] args){
        System.out.println(method1(10));
    }

    public static int method1(int i){
        System.out.println("Method1");
        int a = 1;
        if (i%2==0){
            a = method2(i);
        }
        return a;
    }

    public static int method2(int i){
        System.out.println("Method2");
        return doubleIt(i);
    }

    public static int doubleIt(int i){
        System.out.println("Double");
        return 2*i;
    }
}
```

I. Writing a simple Java class.

Write a Java class called Student that store the following data about students:

1. Name.
2. ID number.
3. Degree.
4. Address (stored as a string).

The constructor should set the value of all of the above variables.
You should provide methods for:

1. Changing the degree of a student.
2. Changing the address of a student.
3. Printing out the details of a student (i.e. toString());
4. Retrieving the ID number of a student.
5. Retrieving the degree of a student.
6. Retrieving the address of a student.
7. Retrieving the name of a student.

Department of
Computer Science

COMPSCI 105 SS – Tutorial Two

Question & Answer Sheet

Introduction:

This tutorial covers the following topics:

1. Magnitude Prefixes.
2. Information Representation in a Machine.
3. Conversion of Numbers among Different Bases.
4. Signed Number.
5. Simple Arithmetic.
6. Bits Operations.

Questions:

A. Magnitude Prefixes.

1. A computer monitor displays 800 lines of 1,000 pixels. Each pixel needs 4 bytes to represent its colour and density.
   i. How many MegaBytes of memory are needed to hold one full screen image?
   ii. A full screen movie with no compression requires 30 images to be displayed each second. How many Gbytes of data are needed for a movie 1 hour and 40 minutes long?
2. Videos stored on hard disk or DVD need to be highly compressed. To see why, consider the following:
   A high resolution video system uses frame of 1,000 x 500 pixels and displays them at the rate of 50 frames per second. Each pixel needs 3 bytes to represent its colour. How many megabytes of data are displayed per second?
   A typical consumer computer hard disk contains 30 Gigabytes of data. What is the longest running time (in seconds) for an uncompressed high-resolution video stored on such a disk?

B. Information representation in a machine.

Complete the following paragraph:

In a computer, the memory is addressed so that every _____ of data is addressable. Every byte of data consists of __ bits, each of which can have 2 possible values, __ and __.

C. Conversion of Numbers among Different Bases.

1. Convert the following numbers to binary and please show your working.
   465
   154
2. Convert the following binary numbers to decimals. Show your working.
   10000_2
   11111_2
   1101011_2
3. Convert the following binary number to octal, to hexadecimal:
   11110101_2 = $\_\_\_\_\__8$ = $\_\_\_\_\__{16}$
   1110110110_2 = $\_\_\_\_\__8$ = $\_\_\_\_\__{16}$
4. Convert the following numbers to decimal:
   567_8
   ABC_{16}
5. Convert the following base 8 number to a base 13 number. You can convert them to decimal first if you wish.
   742_8 = $\_\_\_\_\__{13}$

D. Signed Number.

1. What is the 1's complement and the 2's complement of the following number?
   11001010_2
   1's complement: $\_\_\_\_\__2$
   2's complement: $\_\_\_\_\__2$
2. What do the following 2's complement number represent in decimal?
   i. 01101111_2
   ii. 10011011_2

E. Simple Arithmetic.

1. Perform the following binary addition and indicate if there is an overflow.
   11001001_2
   $\underline{01101101_2+}$
   Has there been an overflow? Why or why not?
2. Perform the following subtraction by adding the 2's complement of the subtrahend. Indicate if there is an overflow.
   $01100011_2 - 01010110_2$
   Has there been an overflow? Why or why not?

F. Bits Operations.

1. Truncate the following numbers to 4 bits where possible.

   11111001_2 = _____
   10011110_2 = _____
   00000011_2 = _____
   01101001_2 = _____

2. Show the output of the following pieces of Java code. Work it out on paper first, and then write a Java file to test it in a computer.

```
int x = 0xFFFFABCD;
int y = (x << 16) >> 16;
int z = x ^ y;
System.out.println(z);
```

```
int x = 0x55555555;
int y = x << 1;
int z = x|y;
System.out.println(z);
```

```
int x = 169;
int y = x << 7;
int z = ((byte)y)>>7;
System.out.println(z);
```

Department of Computer Science

COMPSCI 105 SS – Tutorial Three

Question & Answer Sheet

Introduction:

This tutorial covers the following topics:

1. ASCII.
2. UTF-8/UCS-2 Conversion.
3. IEEE Floating Point Numbers.
4. Exception.
5. Input/Output.

Questions:

A. ASCII.

1. What is the ASCII coding of the word “happy”?

B. UTF-8/UCS-2 Conversion.

1. A UTF-8 code sequence is 0D EF 80 8D D6 AD C3 BF. Derive the equivalent UCS-2 coded characters.
2. The eight bytes coded in hexadecimal as 00 68 4E 10 04 31 are interpreted as UCS-2 coded characters. (The first UCS-2 character is therefore, in hexadecimal, 0068). Convert the UCS-2 characters to UTF-8 coding.

C. IEEE Floating Point Numbers.

1. Express the following signed decimal numbers as IEEE 754 32-bit floating point words. Show the words hexadecimally.
    a) 3.75
    b) -1/9
2. What do the following hexadecimal words represent when regarded as 32-bit IEEE-754 floating point numbers?
    a) 0x41F00000
    b) 0xBE200000
3. Given the IEEE floating point representation of +0.55 is 3F0CCCCD, give the representation of +0.275 and of -2.2, in hexadecimal.

D. Exception.

1. Complete the definition of the Exception below that could be used to indicate that a "ratio has a zero denominator".

```
public class RatioZeroDenominator extends _________________{

    public RatioZeroDenominator(){
        _______________
    }

    public RatioZeroDenominator(_______ message){
        _________ _____________
    }
}
```

2. Use the above Exception in the constructor of the Ratio class to indicate a bad Ratio. Complete the constructor of Ratio in the class definition below:

```
public class Ratio{
    protected int numerator;
    protected int denominator;
    public Ratio (int top, int bottom) _______ _____________________{
        ____________________________________
        ____________________________________
        numerator = top;
        denominator = bottom;
    }
}
```

Note: Any method or constructor which may throw an Exception must be declared as such.

3. Fix the following in 2 ways:

```
public static Ratio makeRatio (int num, int den) {
    return new Ratio(num,den);
}
```

2 ways in which the above code could be fixed:

1. The exception is handled within that method. A sensible error message should be printed when an Exception is thrown.
2. The exception is passed to the method which call the method makeRatio(int num, int den).

E. Input/Output.

1. Write a simple program that takes in a filename as a parameter and copy the file into a new file with the following property:
    a) Each line in the new file is numbered with a line number starting at 1. For example, if the following line is the 2nd line on the original file:
    "here is an example."
    The 2nd line in the new file should be:
    "2: here is an example."
    The line number is followed by 2 spaces, then the text in that line number.
    b) The name of the output file is the same as the input file, with the word "line" added to the front of the original file name. The format of the output file should be unchanged (the extension of the filename is unchanged.

Skeleton file:

```
public class NumberLines{

        public static void main(String[] args){

        }

}
```

Department of
Computer Science

COMPSCI 105 SS – Tutorial Four

Question & Answer Sheet

Introduction:

This tutorial covers the following topics:

1. Software Lifecycle.
2. Abstraction.
3. General Notions of Recursion.
4. Differences between Recursion and Iteration.
5. Conversion of Tail-Recursive Solution into Iterative One.
6. Efficiency of Recursion.

Questions:

A. Software Lifecycle.

1. Why is writing modular programs useful? Look at page 6 and 7 of the textbook.
2. How should you specify a module and its interactions with other modules in the program?
3. What is a loop invariant?
4. Write the pre-condition and the post-condition for the following methods:

```
public static double sqrt(int number){
    return Math.sqrt(number);
}
```

B. Abstraction.

1. Why would you make some details of a module private rather than public?
2. What is fail-safe programming and what types of 'failure' are there?

C. General Notions of Recursion.

1. What does it mean by solving a problem recursively?
2. Write a simple recursive Java method printRange(int low, high) which prints out the number from low to high (inclusive), one number to a line.
   For example:
   printRange(3,7); should print out:
   3
   4
   5
   6
   7
3. What is the base case (basis) of recursion and how can you find it?

4. Write a simple recursive Java method which reverse a String. The method should return a String which is the reverse of the original String which is passed as a parameter.
   The method header should be:
   public String reverseString(String input){}
   i.e. the method should return a String.
   For example, reverseString("cool"); should return a String "looc".

D. Differences between Recursion and Iteration.

1. When is it better to use recursion rather than iteration?

E. Conversion of Tail-Recursive Solution into Iterative One.

1. What is tail recursion?
2. Describe the steps for conversion of tail-recursive solution into an iterative one.

3. Consider the following method that converts a positive decimal number to base 8 and displays the result.

```
public static void displayOctal( int n ){
    if ( n > 0 ){
        if ( n/8 > 0 ){
            displayOctal( n/8 );
        }
        System.out.println(n%8);
    }
}
```

   Describe how the algorithm works.
   Trace the method with n = 100 and show the output. Number the actions done in each method call to show the sequence of the execution of these actions.

4. Transform the recursive method from the previous question into iterative one.

F. Efficiency of Recursion.

1. Why are some of recursive solutions not effective? Explain.

Department of Computer Science

COMPSCI 105 SS – Tutorial Five

Question & Answer Sheet

Introduction:

This tutorial covers the following topics:

1. Data Abstraction and Information Hiding
2. The ADT List and ADT Sorted List
3. Java Classes and Interfaces
4. Java Exception
5. Array-based Implementation of ADT List

Questions:

A. Data Abstraction and Information Hiding

1. What is information hiding? Why is it generally a good idea in programming?
2. Find an abstract data structure defined in Java API. Use it to explain how ADTs are defined. (Hint: You are expected to show what information is hidden and how to access the hidden information.)
3. What is the difference between ADTs and data structures? Can you give an example of a data structure implementing an ADT?

B. The ADT List and ADT Sorted List

1. What is the biggest difference between ADT List and ADT Sorted List?
2. Define your own class *myList* that contains a list of integers, by implementing *java.util.Collection* interface. Write a method *getMax(),* which returns the biggest integer in the list.
3. Add necessary methods to change your ADT List *myList* into an ADT Sorted List *mySortedList*.

C. Java Classes and Interfaces

1. For each pair of classes, indicate which class extends the other:
   i. java.util.ArrayList
   ii. java.lang.Number.java.lang.Integer
   iii. java.lang.Number.java.lang.Object
   iv. java.util.Stack.java.util.Vector
   v. java.util.AbstractCollection
2. What is the restriction of using the keyword “extends”? Can you think of any reason why there is such a restriction?
3. In what situation extending a class would be more preferable than implementing an interface?

D. Java Exceptions

1. Explain the difference between the two types of exceptions in Java. Name two examples of each type of exceptions besides the ones in the textbook.
2. Carrano and Prichard, Chapter 3, Exercise 2, Page 147. You need to define the exception ListIndexOutOfBoundsException first (refer to Appendix A on Page 719).

E. Array-based Implementation of ADT List

1. Design and implement an ADT called Complex that represents a complex number, which contains a real part and an imaginary part. Its operations should include addition, subtraction, and multiplication with another complex number. You should write an interface, IComplex, first, and then make your Complex class which implements IComplex interface.
   The constructor can accept no parameter (default), one double (for the real part) or two doubles. When you print out the complex number, e.g. new Complex (2,3), it should print out "2.0 + 3.0i".

Department of Computer Science

COMPSCI 105 SS – Tutorial Six

Question & Answer Sheet

Introduction:

This tutorial covers the following topics:

1. Implementation Issues of Linked Lists
2. Linked List Variations
3. Introduction to Doubly Linked Lists

Questions:

A. Implementation Issues of Linked Lists

1. When would you use a reference-based implementation of a list instead of an array-based implementation?
2. Why are dummy heads useful for implementing linked lists?
3. Write code to reverse the elements in a linked list, first iteratively, and then recursively. Compare those two implementations.

B. Linked List Variations

1. What are the two most important operations of stack and queue?
2. Write pseudo-code of using linked list to implement both stack and queue.
3. List the advantages and disadvantages of using singly linked list, and doubly linked list.
4. When traversing a circular linked list, how could you know that the traversal had ended?

C. Introduction to Doubly Linked Lists

1. Write pseudo-code to show how to delete the node pointed to by *current* in a doubly liked list without dummy heads.
2. Define a circular doubly linked list with dummy heads *myDoubList*. Implement *insert(), remove()*, *display()*methods for it.
3. Implement *swap()* method, which takes two indexes of elements in the list as parameters and swap their values. Call *display()* to show the results.

Department of
Computer Science

COMPSCI 105 SS –Tutorial Seven

Question Sheet

Introduction:

This tutorial covers the following topics:

1. Stacks
2. Queues
3. Insertion and Bubble sort
4. Introduction to efficiency

Questions:

A. Stacks

1. What are the differences between the following 3 implementations of stacks: Array, Linked-list and ADT List. In what situations is one better than the others?

2. Explain how you would make methods to
   a) count the number of items in a stack, and
   b) print a stack from bottom to top.
   Both should leave the stack in its original state when finished. (Hint: you can use a second stack to help).

B. Queues

1. What are the differences between Array and Linked-list implementations of Queues? What is the difficulty of the Array-based implementation?

2. Write a method to print the contents of a queue
   a) as an external method, using only Queue ADT operations.
   b) as an internal method, inside a linked-list based Queue.
   Both should leave the queue in its original state when finished.

C. Insertion and Bubble sort

1. Write a recursive method that performs insertion sort.

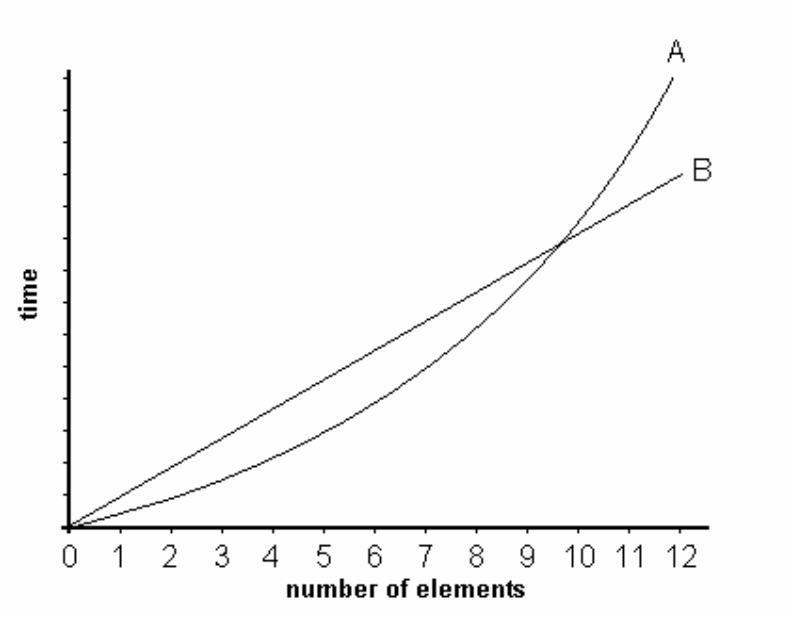
2. The following is a list of birthdays in DD/MM format.
   21/04
   13/02
   01/05
   26/12
   03/11
   a) Using bubble sort, first sort this list by day, then by month.
   b) Using insertion sort, first sort this list by month, then by day.

3. A sorting algorithm is called "stable" when:
   IF two items are equal THEN after sorting they will be in the same order as they started in.
   Is Insertion sort stable? Is Bubble sort stable?

D. Efficiency of Algorithms

1. The running time of two sorting algorithms are shown in the graph below.
   Which algorithm would you choose to use
   1. for sorting arrays with fewer than 10 elements
   2. for sorting arrays with more than 10 elements
   3. for sorting arrays of unknown size

   Explain your answer.



Department of Computer Science

COMPSCI 105 SS – Tutorial Eight

Question Sheet

Introduction:

This tutorial covers the following topics:

1. Big-O and complexity
2. Trees, binary trees
3. Tree traversal

Questions:

A. Big-O and Complexity

1. The textbook claims that algorithm analysis should be independent of specific programming languages, computers and data. Explain why.

2. Sort the following orders of growth from slowest to fastest.
   $O(\log_2 n), O(n), O(1), O(n^3), O(n*\log_2 n), O(n^2), O(2^n), O(n\sqrt{n})$

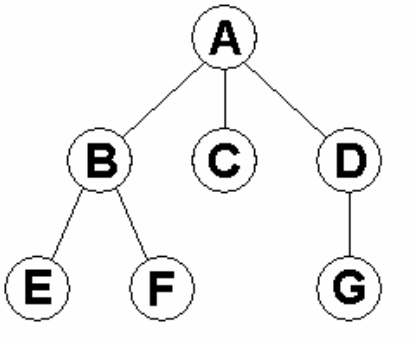
3. Find an example of an algorithm for each of the orders in the previous question.

4. What is the Big-O complexity of the method given below?

```
public int m(int n){
   int s = 0;
   for(int i = 0; i < n; i++){
        int k = n;
        while(k > 0){
           s = s + k;
           k--;
        }
   }
   return s;
}
```

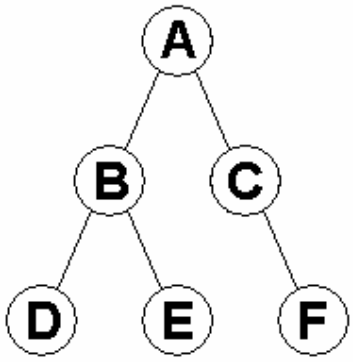
B. Trees

The questions in this section all refer to the tree given below.



1. Which nodes are children of node A?
2. What is the height of the tree?
3. How many ancestors does G have?
4. How many descendants does C have?
5. What are the leaves of the tree?

C. Tree Traversals and Binary Trees



1. List the nodes of this binary tree in-, post-, level- and pre-order.

2. How would you transform this into a complete binary tree with 6 nodes?

3. How would you transform this into a full binary tree?

4. Using only pre- and post-order values, make new definitions for parent, ancestor, descendant, root and right child in binary trees.
   For example: A node *x* is a **left child** of a node *y* iff the pre-order of *x* is one larger than the pre-order of *y*.

Department of
Computer Science

COMPSCI 105 SS – Tutorial Nine

Question Sheet

Introduction:

This tutorial covers the following topics:

1. Binary Search Trees
2. Heaps
3. Heapsort

Questions:

A. Binary Search Trees

1. What is the difference between a binary tree and a binary search tree?
2. Carrano and Prichard, Chapter 10, Exercise 6, Page 487
3. Carrano and Prichard, Chapter 10, Exercise 9, Page 487

B. Heaps

1. What is the difference between a priority queue and a heap?
2. Draw the following array as a heap:

9	7	6	5	1	2	5	4

3. Is this a max or min heap?
4. Draw the heap after the number 8 is inserted.
5. Does the order of insertion of elements into a heap affect their positions in the heap? Prove your answer.

C. Heapsort

1. Which heap operation is used most by heapsort?
2. Sort the two arrays below using heapsort. Which was faster? Explain why.

5	2	1	7	4	6	3

1	2	3	4	5	6	7

Numbers in the Computer

(adapted from Bob Doran's lecture notes of COMPSCI105 S2, 2002)

Big and Small numbers

Computers are so fast and powerful, made of such tiny components, that we need to have a notation for big and small numbers in order to describe them. We do this by multiplying detailed numbers by powers of 10 which range from very large to very small:

Peta 10^{15}
Tera 10^{12}
Giga 10^{9}
Mega 10^{6}
Kilo 10^{3}
Milli 10^{-3}
Micro 10^{-6}
Nano 10^{-9}
Pico 10^{-12}
Femto 10^{-15}

So:

6 nanometres is 6×10^{-9} metres
150 megayears is 150×10^{6} years = 15×10^{7} years

Representing information in a machine

We have to take the pieces of information that we are dealing with - numbers, characters, strings - and represent them with the physical quantities of the machine. This is true even for simple examples like finger counting or an abacus. However, it has been found that 2-state devices are easier to make, faster, more reliable, so much so that representation of information used in computers by more than two states is very rare nowadays. One of the advantages of having a device in only two states is that makes it much easier to decide which state the device is in.

Any particular device might have specific names for the two states, such as left and right or north and south. However, we always call the states *bits* collectively, and use *0* or *1* as names for the two states. Why will become clear later. Now, rather than thinking about how to represent data in devices we can concern ourselves about how we represent information as bits.

How many different things can be distinguished with n bits?

One bit: We could represent a dog by 0 and a cat by 1. So we can only distingish two different things with one device capable of being in two states. This is the number of different patterns can be formed by one thing selected from 0 or 1.

Two bits: There are four 2-bit patterns 00 01 10 11, so we can represent four things.

Continuing in this way we see that n bits can represent 2^n different things.

This is why the powers of two are so important and so often encountered when dealing with computers or digital information. Here are the first 12 powers. You need to know them (really and truly):

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096

In fact, we need them immediately to answer the opposite question:

How many bits are needed to represent m different things?

Example: m=1000. From our powers of 2 we know that 9 bits can represent 512 things so 9 bits are not enough. But 10 bits allows 1024 different things to be represented. So 10 bits is the answer, although some of the 10-bit patterns are not used.

If $a = b^c$, we write the other relationship as $c = \log_b a$. So we can say that roundup($\log_2 m$) bits are needed to represent m different things. For the above example $\log_2 1000 = 9.9$ approximately and this rounds up to 10. In Computer Science we mean $\log_2 m$ when we write log m, mostly.

Data Representation

Computer Science 105 1996-2004
(prepared originally by Dr Peter Fenwick)

Data Representation in Memory

The "*Memory*" of a computer is usually a numbered set of "*bytes*". For an "64 Megabyte" memory the numbers, or "addresses", range from 0 to about 67,000,000 (actually 67,088,863). Each address identifies one unique byte out of the 67 million (or whatever is the memory size). Each byte consists of 8 "*bits*", which are the fundamental unit of data representation. A bit is usually written as having the two possible values "0" or "1".

A single byte can hold one character (letter, decimal digit, punctuation, etc) or a small integer (< 256). As a number limited to 256 is seldom of much use, it is usual to collect bytes together in groups. This larger grouping of bytes is called a *word*. The size of a word depends on the computer but is usually 16 or 32 bits, but many other sizes have been used.

A very important point (it is difficult to over-emphasise it) is that a pattern of bits in memory is just a pattern of bits - no more and no less - with no intrinsic meaning at all. Its meaning depends entirely on how we (the programmer) or the computer interprets it.

Thus a group of 32 bits may be

four 8-bit characters
one 32-bit integer
one 32-bit instruction
one 32-bit address
two 16-bit integers
one 16 bit integer, one 8-bit integer, and one 8-bit character,
etc, etc ...

The meaning depends entirely on its use by the computer's programs.

In Java the different sizes correspond to different primitive data types -

bytes	bits	Java	maximum positive value
1	8	`byte`	127
2	16	`short`	32 767
4	32	`int`	2 147 483 647
8	64	`long`	9 223 372 036 854 775 807

Why use bits, 0 or 1?

Why not use say decimal digits, 0...9?

Logic. Computers are built with devices which use "Boolean logic", or "2-valued logic". The *logical values* (0 & 1) can be regarded as equivalent to the *data values* (also 0 & 1), making it easy to use logic devices to perform arithmetic.

Engineering. It is much easier to make devices which can reliably handle only two values (0, 1) than, say, 10 values (0...9).

Efficiency. With bits as the data units we are forced to use *binary* representation (or *base*-2) for numbers. It can be shown that a binary representation is much more efficient than the more usual decimal number representation.[1]

Number representation

In most of our work with numbers and computers we must be very careful to distinguish between a *value* and its *representation*. There is usually little distinction between a value, say 13, and its representation in decimal.

[1] For example, a value less than 1,000 requires 3 decimal digits, each digit requiring 10 states (0 - 9) for a total of 30 states, while with binary it requires about 10 binary digits, each with 2 states, for a total of 20 states. The optimal base to minimise the number of states is $e = 2.71818285$, but that is hardly practical. Base 3 representation is slightly more efficient than base 2 and has been used in a few (experimental) computers.

Consider however MCMXCVIII, which is a different way of representing 1998 (and MCMXCVIII+IV= MMII - this should be obvious to you!).

As far as we are concerned (but not in Roman numbers!) values are always represented as a sequence of digits ($x_{n-1}, x_{n-2}, \ldots, x_1, x_0$) and a base b.

A value N with base b and n digits is given by

$$N = x_{n-1}b^{n-1} + x_{n-2}b^{n-2} + \ldots + x_1 b^1 + x_0$$
$$= \sum_{i=0}^{n-1} x_i b^i$$

The value is represented by a polynomial in the *base*, with the *digits* of the representation being the *coefficients* of the polynomial. coefficient x_i is in the range $0 \le x_i < b$.

If the base is 10, things aren't very interesting. A number such as 56432 means

$$5\times10^4 + 6\times10^3 + 4\times10^2 + 3\times10 + 2$$

In base 2 though, and doing the arithmetic in our familiar base 10, the value 10101_2 is

$$1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1 = 16 + 4 + 1 = 21$$

(A value written as $ddd...dd_b$, means that it is represented as a base b number. The base b is omitted if it is 10.)

So to convert a binary number to decimal we can write, in decimal, the values of the powers of two corresponding to 1's in the binary representation and add these powers together.

Powers of two.

These must be learnt, certainly up to 2^8 (256) and preferably up to 2^{16} (65,536).

	Binary (to 13 bits)	Decimal
2^0	...0 0000 0000 0001	1
2^1	...0 0000 0000 0010	2
2^2	...0 0000 0000 0100	4
2^3	...0 0000 0000 1000	8
2^4	...0 0000 0001 0000	16
2^5	...0 0000 0010 0000	32
2^6	...0 0000 0100 0000	64
2^7	...0 0000 1000 0000	128
2^8	...0 0001 0000 0000	256
2^9	...0 0010 0000 0000	512
2^{10}	...0 0100 0000 0000	1,024
2^{11}	...0 1000 0000 0000	2,048

Converting between number bases.

In converting numbers we must perform arithmetic in some base (usually base 2 on computers, base 10 for humans) - call this the *native base*.

To convert into the native base.
There are two ways of converting *into* the native base, based on different ways of evaluating the polynomial.

The first assumes that we know the powers of the old base expressed in the native base. We multiply each of the powers by its appropriate digit and add the values, as was done in the example -

$$1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1 = 16 + 4 + 1 = 21$$

The second writes the polynomial in a better form for computer evaluation.

$$ax^4+bx^3+cx^2+dx+e = e + x(d + x(c + x(b + xa)))$$

The number here would appear as *abcde*. Assume a "value so far" (*V*), which is initially set to 0. Working from the left-most (most significant) digit, multiply *V* by the base and add in the next digit.

To convert from the native base.

Set the working value V to the number to convert. Then calculate

d := *V* **mod** *base*; and *V* := *V* **div** *base*; or

d = *V* % *base*; and *V* = *V* / *base*(*V* = 0). The successive values of *d* are the digits in order, from least-significant (right most) to most-significant (left most). [`V%base` returns the *remainder* on division, and `V/base` returns the *quotient*., where `base` is the new base represented in the native base.]

Converting a decimal number to binary

With decimal native base, to convert 237 to binary

```
2)237  + 1 rem
2)118  + 0 rem
2)59   + 1 rem
2)29   + 1 rem
2)14   + 0 rem
2)7    + 1 rem
2)3    + 1 rem
2)1    + 1 rem
  0
```

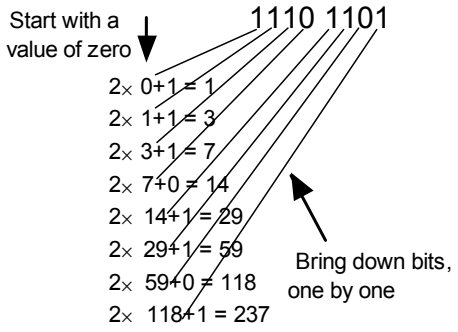
Collecting the remainders gives 1110 1101 as the result

Converting a binary number to decimal

To convert 1110 1101 to decimal, using the first method where decimal is the native base, we have

$1\times128 + 1\times64 + 1\times32 + 1\times8 + 1\times4 + 1 =$ **237**

Converting the same binary number, using the second method, gives -



Octal and hexadecimal numbers.

Pure binary numbers have so many digits that they are often difficult to handle. We usually collect bits in groups of 3 (base-8 or octal) or 4 (base-16, or hexadecimal, *NOT "hexidecimal"*) to get numbers with fewer digits that are easier for people to handle. binary value 0110110100110100_2 would be written in hexadecimal as $6D34_{16}$.

octal			hexadecimal			hexadecimal	
bits	digit		bits	digit		bits	digit
000	0		0000	0		1000	8
001	1		0001	1		1001	9
010	2		0010	2		1010	A
011	3		0011	3		1011	B
100	4		0100	4		1100	C
101	5		0101	5		1101	D
110	6		0110	6		1110	E
111	7		0111	7		1111	F

Converting between octal, binary and hexadecimal.

As octal and hexadecimal are just ways of rewriting binary, and the conversion is normally done by inspection,. e.g.:

Octal	5	6	4	5	2	0	1	7
Convert to binary	101	110	100	101	010	000	001	111
regroup bits	1011	1010		0101	0100		0000	1111
Hexadecimal	B	A		5	4		0	F

When converting binary integers to hexadecimal, start at the *right* and count off groups of 4 bits, filling out with high-order zeros as needed. (For octal, count off groups of 3 bits from the right.)

Example, with the "fill" bits shown smaller -

hex 1010111010110 → 0001 0101 1101 0110
1 5 d 6

oct 1010111010110 → 001 010 111 010 110
1 2 7 2 6

Java representation

Java usually represents "integer type" values (byte, short, int and long) in the conventional decimal form but converts them internally to binary. Long (64-bit) constants must be written with a following "L" (or "l") because a string of decimal digits is, by default, an integer.

Where the bit pattern is important (rather than a numerical value) Java allows a string of hexadecimal digits, preceded by 0x. Upper and lower case are equivalent. Thus

hexadecimal	corresponding decimal
0x7	7
0x2b	43
0xFF	255
0x7cf	1999
0xf4240	1000000L

Converting a number to binary, using hexadecimal

To convert from decimal to binary we can first, working in decimal, convert the decimal number to hexadecimal. Because the arithmetic is decimal the hexadecimal digits will initially be in decimal. We then convert them to true hexadecimal and re-express the hexadecimal number as binary.

To convert 237 to binary

```
16)237 + 13 rem = D16
 16)14 + 14 rem = E16
    0
```

The hexadecimal representation is ED_{16} in binary gives 1110 1101

Conversion by table lookup

This method, working in decimal, avoids multiplication by using a table of the decimal equivalents of selected hexadecimal numbers (or octal, but we will use hexadecimal.).Each table entry has the decimal value if its "row digit" is substituted for X in its column header. Thus 400_{16} (row 4, column X00) is 1,024, and $B000_{16}$ is 45,056.

To convert from hexadecimal to decimal, we use the table to get the decimal equivalent of each hexadecimal digit and add up those values. For example $ABCD_{16}$ is A000+B00+C0+D = 40,960+2,816+192+13=43,981.

To convert from decimal to hexadecimal, look for the largest table value which does not exceed the value. Write down its hexadecimal digit and subtract the decimal value. Repeat until the value has been reduced to zero. For example, to convert 45,678 to hexadecimal, we see that the first digit must be B (B000 = 45,056); subtracting this value gives a new value of 622. Repeating the operation gives successive digits of 2 (200=512), 6 and E. Thus 45678 = $B26E_{16}$.

	X	X0	X00	X000	X 0000	X0 0000
0	0	0	0	0	0	0
1	1	16	256	4,096	65,536	1,048,576
2	2	32	512	8,192	131,072	2,097,152
3	3	48	768	12,288	196,608	3,145,728
4	4	64	1,024	16,384	262,144	4,194,304
5	5	80	1,280	20,480	327,680	5,242,880
6	6	96	1,536	24,576	393,216	6,291,456
7	7	112	1,792	28,672	458,752	7,340,032
8	8	128	2,048	32,768	524,288	8,388,608
9	9	144	2,304	36,864	589,824	9,437,184
A	10	160	2,560	40,960	655,360	10,485,760
B	11	176	2,816	45,056	720,896	11,534,336
C	12	192	3,072	49,152	786,432	12,582,912
D	13	208	3,328	53,248	851,968	13,631,488
E	14	224	3,584	57,344	917,504	14,680,064
F	15	240	3,840	61,440	983,040	15,728,640

Hexadecimal conversion table.

Tables such as this are often used to perform number-base conversions within computers. The table shown above uses decimal arithmetic to convert hexadecimal to decimal. A similar table, with hexadecimal equivalents of decimal numbers can convert decimal values into hexadecimal (or binary). The table is interrogated with decimal values and gives the corresponding hexadecimal values. The table has columns giving the hexadecimal equivalents of tens, hundreds, thousands, etc. Arithmetic is now done in hexadecimal (or binary).

	1	10	100	1,000	10,000	100,000
0	00	00	000	0000	00000	00000
1	01	0A	064	03E8	02710	186A0
2	02	14	0C8	07D0	04E20	30D40
3	03	1E	12C	0BB8	07530	493E0
4	04	28	190	0FA0	09C40	61A80
5	05	32	1F4	1388	0C350	7A120
6	06	3C	258	1770	0EA60	927C0
7	07	46	2BC	1B58	11170	AAE60
8	08	50	320	1F40	13880	C3500
9	09	5A	384	2328	15F90	DBBA0

To convert 45,678 to hexadecimal (with hexadecimal arithmetic) the table gives -

40,000 = 09C40
5,000 = 01388
600 = 00258
70 = 00046
8 = 00008

value = 0B26E (= 45,678)

Arithmetic

Adding binary numbers.

The rules for adding numbers are very similar in all number bases, provided that we can add pairs of digits. The addition of pairs of digits may be defined by an addition table, or we can just use our familiar decimal addition with appropriate changes. Starting from the right (least significant) digit,

- add each pair of digits.
- if the sum is not less than the base, subtract the base and generate a "carry" to include in the next addition to the left.

Binary addition table

This table shows all possible values of the three inputs to a binary addition and the corresponding *Sum* and *Carry-out*.

Add-end	Aug-end	Carry In	Carry Out	Sum	Σ inputs	Σ inputs = 2	Σ inputs 1 or 3
0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	1
0	1	0	0	1	1	0	1
0	1	1	1	0	2	1	0
1	0	0	0	1	1	0	1
1	0	1	1	0	2	1	0
1	1	0	1	0	2	1	0
1	1	1	1	1	3	1	1

It also shows an alternative approach to the Sum and Carry-out, based on the number of inputs which are 1.

Addition example.

Add (0110 1111 0101 1001 + 0010 0000 1011 0011)

Take augend	0110	1111	0101	1001
and the addend	0010	0000	1011	0011
with the carries	1101	1111	1110	0110
then add, to get answer	1001	0000	0000	1100

Check that the answer is (28,505 + 8,371) = 36,876

Each carry digit is shifted left by one place from where it is generated, to be added in with the next-significant digits See the "carry propagation" through most of the middle 8 bits. For each position, proceeding right to left, we add the digits and the incoming carry. If the sum is not less than the base, enter a 1 as the carry-in to the next position to the left and subtract the base from the sum; enter the difference as the sum digit.

If we are adding two numbers x and y in base b, with the digits $x_{N-1}...x_3\, x_2\, x_1\, x_0$ and $y_{N-1}...y_3\, y_2\, y_1\, y_0$ to give a sum $z_{N-1}...z_3\, z_2\, z_1\, z_0$ we can hold each of the sets of digits in an integer array and add them with the program. The operation follows exactly from the description above.

```
Carry = 0;
for (i = 0; i < N; i++)// right to left scan
  {
  z[i] = x[i] + y[i] + Carry;  // the add
  if (z[i] >= base)   // digit overflow!!
    {
    Carry = 1;     // carry to next stage
```

```
      z[i] = z[i] - base;// correct overflow
      }
    else
      Carry = 0;  // no carry to next stage
    }
```

Terminology

If we add two numbers to produce a third, as $X + Y \rightarrow Z$, then

- X is the *augend* (that which is augmented)
- Y is the *addend* (that which is added)
- Z (the result) is the *sum*

Addition is completely symmetrical in the two inputs and we often call both inputs an "addend".

If we subtract one number from another to produce a third, as $X - Y \rightarrow Z$, then

- X is the *minuend* (that which is diminished)
- Y is the *subtrahend* (that which is subtracted)
- Z (the result) is the *difference*

Subtracting binary numbers.

Subtraction is similar to addition. We work in the same direction (right to left), but now must *borrow* if the subtraction "cannot be done" rather than *carry*. Again we can use a subtraction table, or we can just use our familiar decimal subtraction with appropriate changes. Starting from the right (least significant) digit,

- subtract each pair of digits.
- if the result is less than zero, add on the base to the result and generate a "borrow" to include in the next subtraction to the left.

Before examining binary subtraction it is best to consider decimal subtraction and especially the action of the "borrow". ("Borrowing" is an aspect which is seldom well-explained.) As an example, take 416 - 263.

- Remember always that any generated digit d of the difference must be such that $0 \leq d < 10$.
- The first subtraction, of the units digits, is 6 - 3 = 3 with no problem.
- The next, tens digit, subtraction yields 1 - 6 = -5, which is outside the valid range.
- To correct this "overdraw", *add 10* (the number base) to the tens digit of the minuend and compensate by *subtracting 1* from the hundreds digit of the minuend. (These cancel and have no overall effect.)
- The tens digit subtraction is now 11 - 6 = 5, which is a valid result.
- With the borrow of 1, the hundreds digit subtraction is no longer 4 - 2 but (4 - 1) - 2 = 3 - 2 = 1.
- The difference is then 153.

The actions in binary subtraction are identical; if the subtraction "won't go", add the base (10_2) to the minuend digit and decrement the next most-significant minuend digit by 1. (A better way of binary subtraction is given later!)

Subtracting two numbers x and y in base b, $x_{N-1}...x_3\, x_2\, x_1\, x_0$ and $y_{N-1}...y_3\, y_2\, y_1\, y_0$ to give a result $z_{N-1}...z_3\, z_2\, z_1\, z_0$ ($x - y \rightarrow z$):

```
Borrow = 0;
for (i = 0; i < N; i++)
   {
   z[i] = x[i] - y[i] - Borrow;
   if (z[i] < 0)
     {
     Borrow = 1;
     z[i] = z[i] + base;
     }
   else
     Borrow = 0;
   }
```

Subtraction example.

subtract (0110 1111 0101 1001 - 0010 0000 1011 0011)

Take minuend	0110	1111	0101	1001
and the subtrahend	0010	0000	1011	0011
with the borrows	0000	0001	0100	1100
subtract, to get answer	0100	1110	1010	0110

Check that the answer is (28,505 - 8,371) = 20,134

Each borrow digit is shifted left by one place from where it is generated, to be subtracted the next-significant minuend digit.

Negative Numbers

Although positive integers (the "natural" numbers) are important, they are inadequate for practical arithmetic. To allow negative values as well we require a *signed number* representation. Following the earlier discussion, we now introduce ways of *representing* both positive and negative *values*.

- There are several ways of representing signed binary numbers.
- All representations are based on the unsigned (positive) number representation
- All use the left-most bit as the sign - usually
  $\Rightarrow$ positive, and 1 $\Rightarrow$ negative.
- Most represent positive integers exactly as with an unsigned representation.

An important operation is that of *complementing* a value, or changing its sign. The complement of +3 is -3, and of -46 is +46. We can complement a positive value (to get a negative value) or a negative value (getting a positive value).

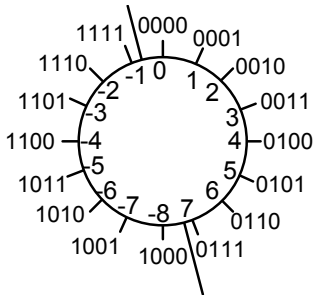
There are three important representations for signed binary numbers. All represent positive integers as for unsigned integers and all reserve the most-significant bit as the sign bit (0 $\Rightarrow$ +ve, and 1 $\Rightarrow$ -ve).

1. Sign and Magnitude. Only the sign bit changes when complementing the number. In 8 bits, +5 is 00000101, and -5 is 10000101. Sign and magnitude representation is used only in the significands of floating point numbers and when we display or print decimal numbers.

2. Ones complement. To complement a value of either sign, change all the bits, 0$\rightarrow$1 and 1$\rightarrow$0. So 00000101 (+5) becomes 11111010; recomplementing obviously recovers the original 00000101. Ones complement has two representations for zero - +0=000...000 and -0=111...111, but every value represented can be complemented. It was once an important number representation, but is now seldom used. An important property of 1's complement arithmetic is that adding a number and its complement always gives an all-1's result.

3. Twos complement. This is now the only way used to represent signed binary integers. To form the twos complement, take the ones complement and then add 1. Thus, if +5 is 00000101, then -5 is 11111011, and complementing -5 gives 00000101. Values represented in twos complement are added just as if they were simple unsigned quantities. There is only one representation for zero, but the most negative number has no complement. With 8 bits, the range is $-128 \le V \le 127$. If x is *any* number represented in 2s complement to N bits, then with unsigned addition and allowing the sum to grow longer $x + -x \rightarrow 2^N$. For example, to 8 bits, +5 is 00000101 and -5 is 11111011; adding the two gives 00000101+11111011 $\rightarrow$ 100000000 = 256 = 2^8. (But if the result is truncated to only 8 bits it is 00000000, just as expected.)

Twos complement numbers may be represented as points around a circle, shown here for 4-bit values.



Addition is performed by stepping clockwise around the circle and subtraction anticlockwise. A special axis, just anticlockwise from vertical, marks sign changes. Stepping across it near zero is a normal sign change. Stepping across remote from zero corresponds to an overflow (between -8 and +7). The symmetry of the diagram indicates why there is a -8 but not a +8.

Another way of representing signed values is to use an "Excess" or "biased" representation. In this a "bias" is added to the value so that all legitimate values appear, after adjustment, as positive, unsigned, integers. This unsigned integer is used as the representation of the value. The bias is usually about half of the number range; for 8-bit representations it is either 127 or 128. Excess representations are used mainly for the exponents of floating point numbers.

We often say "twos (or ones) complement a value". This means "change the sign of the value according to twos (or ones) complement rules". If the original value was positive it will end up negative; if it was negative the result will be positive (usually).

Sign extension

A positive or unsigned integer is always extended by prefixing it with zeros. (As in decimal, 1234 is *identical* to 00001234, but we usually suppress leading zeros.) Just as positive numbers extend to the left with 0 bits, so do negative numbers extend to the left with 1 bits (except for sign and magnitude). The operation of sign extension is important if we have say a signed 8-bit or 16-bit value and must extend to a 32-bit signed value. The sign bit of the old value is essentially "propagated through" the unused bits of the new value. For example, 8-bit to 16-bit extensions are

```
      0011 0101 → 0000 0000 0011 0101
and,  1101 1001 → 1111 1111 1101 1001
```

A longer value can be converted to a shorter value by discarding the high-order or left-hand bits, *provided that the discarded bits are all equal to the sign bit of the new, shorter, value.*

Overflow

As computers always represent integers in some small number of bits, usually 16 or 32, only a limited range of values can be represented. If two large positive values are added the result may exceed the maximum value, leading to an *overflow*. Overflows can also occur when adding two large negative values, but never when adding numbers of opposite sign.

An overflow shows itself as a number of the wrong apparent sign. For example, with 4 bits and twos complement, the range is $-8 \le V \le 7$. Adding 6 + 7 gives

$$\begin{array}{r} 0110 \\ +\ 0111 \\ \hline 1101 \end{array}$$

The result (13) is outside the valid range and appears as a negative sum of two positive values. Similarly, -6 + -7 gives an apparently positive result

$$\begin{array}{rl} 1010 & \\ +\ 1001 & \\ \hline 10011 & = 0011 \end{array}$$

Both results would be correct with more digits available. In general an overflow may be detected as a result of unexpected sign (+ve + +ve → -ve, or -ve + -ve → +ve)

Another way to detect overflow is to look at the carries into and out of the sign bit. If these are not equal, there is an overflow. (This method also works for subtraction by addition of the complement, as described later, and is the technique used in computer hardware.)

Remember, when adding *N*-bit twos complement numbers, always discard the carry coming out of the sign bit (truncate the result at *N* bits).

Summary of signed binary numbers

- To complement (change the sign of) a sign & magnitude number, change the sign bit to 0 (+ve result) or 1 (-ve result)
- To complement a one's complement number, change all the bits, 0 → 1 and 1 → 0 (the logical NOT operation).
- To complement a two's complement number, take the one's complement and then add 1,starting from the rightmost bit, copy all 0 bits and the rightmost 1; then complement all the more significant bits.

Subtraction by complement addition

Computers usually perform subtraction by adding the complement of the subtrahend - use X-Y = X+(-Y).

To calculate `0100 1011 0110` - 0011 0111 1001 ($1{,}206_{10}$-889_{10}) using 2's complement arithmetic.

```
Take subtrahend            0011 0111 1001
1's complement it          1100 1000 0110
add minuend                0100 1011 0110
with a carry-in for the +1 1001 0000 1101
then add, to get answer  1 0001 0011 1101          =  317
```

The answer has a carry out of the high order bit. Detail shows that there is also a carry *into* the high-order bit. A correct 2's complement addition requires that the carry *into the sign bit* must be equal to the carry *out of the sign bit*. (We can also detect overflow by the result having an unexpected sign as a positive number result when subtracting a positive number from a negative number.)

Decimal arithmetic by complementing

A very similar process can be used with decimal subtraction, retaining trailing zeros, 10's complementing the least-significant non-zero and 9's complementing all digits to its left. (If *y* is the 9's complement of *x*, then $x + y \equiv 9$, and if *z* is the 10's complement of *x*, then $x + z \equiv 10$.) A negative number now has leading 9's, or a leading digit of 5 or greater.

For example to calculate 235 - 164,

The subtrahend complement is 836 (1 → 8, 6 → 3 and 10's complementing 4 gives 6.)
add the 836 to the minuend 235 to give 235 + 836 = 1071.
Ignoring the carry out gives the correct answer of 71

With trailing zeros e.g. 7435 - 1250 (and extending with high-order zeros)

The complement of 1250 is 998750
Add 7435 (007435+998750 → 1006185)
Discard the carry-out to give the difference 6185.

To add *two* negative numbers by adding their decimal complements, consider -4512 + (-1200), which gives 995488 + 998800 = 1994288. Discarding the carry gives 994288, which is the complement representation of -5712.

[For another look at complementing, take the 9's complement of *every* digit, and add a 1 carry-in. Trailing 0's then revert to 0 and the carry propagates. The first complement digit which is not a 9 absorbs the carry; the addition of another 1 turns it into a 10's complement. More-significant digits stay as the 9's complement. This corresponds precisely to adding 1 to the binary 1's complement.]

Signed addition and subtraction

The basic rules given for addition and subtraction really apply only to unsigned (positive) numbers, although negative values have been mentioned at times. When we have genuinely signed numbers, the rules depend on the representation. (you need to know only the part for two's complement).

1. Sign and magnitude numbers are usually converted to one of the other representations and the result converted back if necessary. This means that S&M addition and subtraction is relatively complex and is one good reason for avoiding this representation.
2. Ones complement numbers are always added by adding the representations as unsigned values, with appropriate handling of the carry (see later). Subtraction is usually performed by complement addition. A detailed study of the one's complement representation shows that if an addition yields a carry out of the sign bit, the sum is too small and 1 must be added into the low-order position. This is handled by connecting the high-order carry out to the low-order carry-in, a so-called *end-around carry*.
3. Two's complement numbers are always added as unsigned values, with subtraction performed by complement addition. In other words there is virtually no special treatment needed, and that is one good reason for using two's complement.

Binary fractions.

The polynomial representation can be extended to handle negative powers of the base and therefore values less than 1. The rules are very similar - each bit corresponds to a power of 2 (a -ve power) and we get the decimal value by adding up the decimal values of these powers.

While integers can be always represented exactly in binary, *few decimal fractions have an exact representation*. There is little obvious difference between 0.01 and $1/\sqrt{2}$ even though one is a rational decimal and one an irrational number.

0.10_{10} ~ 0.0001 1001 1001 1001 1001 1001 ...
0.01_{10} ~ 0.0000 0010 1000 1111 0101 1100 ...
$1/\bullet 2_{10}$ ~ 0.1011 0101 0000 0100 1111 0011 ...

Negative powers of two.
While it may be useful to remember the first few of these values, they are generally less important than the positive powers.

n	2^{-n}
0	1
-1	0.5
-2	0.25
-3	0.125
-4	0.0625
-5	0.03125
-6	0.015625
-7	0.0078125
-8	0.00390625
-9	0.001953125
-10	0.0009765625

Converting fractions between number bases.

To convert from the native base. Successively multiply the fraction by the new base. At each stage take the integral part as the next digit and retain the fraction for the next stage. Thus, to convert a binary fraction to decimal, on a binary computer, repeatedly multiply by 1010_2, collecting the integral parts each time.

To convert into the native base. Working from the *least-significant* fractional digit, append the digit as an integer *prefix* to the fraction-so-far and divide the whole by the old base; repeat for each digit. Alternatively, convert the fraction as though an integer, simultaneously building an *input base* scaling factor. Finally, divide the converted fraction by the scale factor.

Extending binary fractions

Irrespective of the sign, binary fractions always extend to longer precision by adding zeros to the right.

From Binary fraction to decimal (binary arithmetic)

Take 0.00101 (= $2^{-3} + 2^{-5} = 0.125 + 0.03125 = 0.15625$)

(To multiply *x* by 1010, add *x* shifted left 1 and *x* shifted left 3)

0.00101 × 1010 = 0001.10010	int = 1
0.10010 × 1010 = 0101.10100	int = 5
0.10100 × 1010 = 0110.01000	int = 6
0.01000 × 1010 = 0010.10000	int = 2
0.10000 × 1010 = 0101.00000	int = 5

Stop when the fraction becomes zero.

Collecting digits, we get 0.15625 as the decimal fraction equivalent to 0.00101.

Alternatively, convert 00101 to positive decimal as 5. 00101 has to be shifted left 5 bit positions, multiplied by 2^5, to change it from a fraction to a whole number with value 5. So the value of the fraction is $5/2^5 = 5/32 = 0.15625$ if we perform the division.

Binary fraction into decimal (decimal native base and arithmetic)

Take 0.00101 (= $2^{-3} + 2^{-5} = 0.125 + 0.03125 = 0.15625$)

Take an initial working value, V = 0.0.

Then bring in the binary digits, from the right, as the integral part ahead of the previous value V and divide the whole value (integer.fraction) by 2.

Repeat until all of the binary digits are used.

Add bit 1	1.0000 / 2 = 0.50000
Add bit 0	0.5000 / 2 = 0.25000
Add bit 1	1.2500 / 2 = 0.62500
Add bit 0	0.6250 / 2 = 0.31250
Add bit 0	0.3125 / 2 = 0.15625

Mixed-Base numbers (not examinable)

Sometimes we have numbers with a mixture of bases. There used to be many more before the adoption of decimal money and metric measurements, because traditional weights and measures are full of strange relations between units. Now only time and angles are important (apart from in the USA where distances and weights are still non-decimal). Some examples are shown in the following table. In all cases the unit has its base below. The base is now a *vector*, such as {24, 60, 60} for time, or {360, 60, 60} for angles.

Addition is done from right to left, at each stage reducing the result by the corresponding component of the base vector, shown immediately below the unit name. Thus the pence total is divided by 12 to give the shillings carry, and the remainder is the pence result.

Time	days	hours	minutes	seconds
		24	60	60
Angles	revolutions	degrees	minutes	seconds
		360	60	60
Distance	miles	yards	feet	inches
		1760	3	12
Weight	tons	cwt	pounds	ounces
		20	112	16
Sterling	pounds	shillings	pence	
		20	12	

For example, if an angle of 24 degrees, 35 minutes and 25 seconds is written as 24° 35' 25", and we want to add

83° 42' 54"
+ 75° 23' 58"
+ 66° 45' 32"
+ 84° 23' 12"
+ 75° 56' 11".

- First add the seconds, for a total of 167". Divide 167 by 60, for remainder of 47 (which is the seconds result) and quotient of 2 (which carries into the minutes).
- Adding the minutes gives 191 minutes = 3°11', with carry-in of 2'. Save the 11' as the minutes sum and carry the 3° to the next (degrees).
- Now add the degrees to give a total of 386°, or 360° (1 revolution) + 26°
- The sum is then either 386° 11' 47", or (reducing to whole revolutions) 26° 11' 47".

Using bits as bits (not as parts of numbers)

Often we want to work with individual bits within a word, and must use operations of logic rather than arithmetic. The ones we need involve only one or two operands and it is possible to write a "*truth table*" listing all possible inputs and the corresponding results.

NOT ¬ Inverts its 1-bit argument
AND & Yields 1 (TRUE) if both inputs are 1
OR | Yields 1 if either input is true (1)
EOR ⊕ (exclusive OR) 1 if one input = 1, but not both

Inputs		NOT	AND	OR	EOR	NAND
X	Y	¬X	X & Y	X | Y	X ⊕ Y	¬ (X & Y)
0	0	1	0	0	0	1
0	1	1	0	1	1	1
1	0	0	0	1	1	1
1	1	0	1	1	0	0

The symbols for the logical operations are not completely standard - there are other conventions.

Using bit operations - masking and shifting

A computer word is often divided into several *fields*, or groups of bits, as in floating point representations. Sometimes we handle the word as a whole; sometimes we must get inside it and work on its internal fields. The fields can be extracted and inserted using the logical operations with suitable *masks*.

For example, later we will meet floating point numbers, where the 32 bits of a single word are divided up as -

```
sxxx xxxx xfff ffff ffff ffff ffff ffff
```

where *s* is the *sign bit*, *xx...xx* is the *exponent*, and ff...ff is the *significand*, *fraction*, or *mantissa*. The word can be "ANDed" with the following hexadecimal masks to isolate its fields -

7F800000 extracts the exponent
80000000 obtains the sign bit
007FFFFF obtains the significand
807FFFFF clears the exponent
FF800000 clears the significand

Fields must be moved left or right to bring them into the correct *alignment*. For example, the exponent may have to be shifted right by 23 bits to use it as in integer, or an integer shifted left to become an exponent. An OR can insert bits into a field first cleared by masking.

The Exclusive OR is an interesting operation. If a field is extracted by masking and the field is then EORed back into the original word, the field in the word is cleared.

In summary -

An AND clears to 0 wherever the mask is 0
An OR forces to 1 wherever the mask is 1
An EOR complements a bit wherever the mask is 1

Bit Operations in Java

We have already seen that Java allows hexadecimal constants to enter bit patterns into "integer type" variables. Java also provides a set of logical or bit-wise operations -

&	logical AND	`0x3c & 0x1f` → `0x1c`
|	logical OR	`0x3c | 0x1f` → `0x3f`
^	exclusive OR	`0x3c ^ 0x1f` → `0x23`
<<	shift left	`0x3c << 2` → `0xF0`
>>	shift right (sign fill)	`0xff23 >> 4` → `0xfff2`
>>>	shift right (zero fill)	`0xff23 >>> 4` → `0x0ff2`

In the "shift" operations, the right operand gives the number of bits to shift. In the last 2 examples, `0xff23` is assumed to represent a 16-bit "short" value; we see the difference between "sign fill" and "zero fill".

Note that while shift right with sign fill is the same as numeric division by a power of 2 for positive values, it is not the same for all negative values.

It is possible to convert between compatible numeric types provided that the destination type can contain the value.

- Assignments such as `longVar = intVar`, or `intVar = shortVar` "widen" the representation and are *always* possible because the right-hand side is a subset of the left-hand side.
- Conversions in the reverse direction which "narrow" the representation or reduce the precision use a "cast" such as `intVar = (int)longVar` or `shortVar = (short)intVar`.

Java programs can also use the wrapper classes to provide a wider range of conversions. We illustrate with the `Java.lang.Long` class; other integer types have corresponding methods.

The first group are Class methods which may be called as `longHexStr = Long.toHexString(longVal)`

- public static String toBinaryString(long i) converts the value i into the corresponding String of binary digits ("0" and "1").
- public static String toHexString(long i) converts the value i into the corresponding String of hexadecimal digits.
- public static String toOctalString(long i) converts the value i into the corresponding String of octal digits ("0" to "7").
- public static String toString(long i) converts the value i into the corresponding String of decimal digits ("0" to "9"), or to any radix.
- public static long parseLong(String s) converts the String s to the corresponding long value. (There are several other similar methods, some returning a Long object or allowing a non-decimal radix.)

The remaining, Instance, methods are of less value because many duplicate implicit conversions and casting. They "wrap around" an existing value, such as `Long(longVal).byteValue()`. Not all are given here.

- public byte byteValue() converts to a byte value
- public short shortValue() converts to a short value
- public int intValue() converts to an int value
- public long longValue() converts to a long value (!!)
- public float floatValue() converts to a float value (32-bit floating point)
- public double doubleValue() converts to a double value (64-bit floating point)
- public String toString() converts to a String

Using bit operations - masking and shifting

To see how the masks are generated, look again at the floating point number. If we want to copy a field, the mask must have 1's for every position of the field and 0's elsewhere.

Where the mask has 0's it *must* give a 0 result, but where it has 1's it gives 1's only where the original data has 1's and gives 0's elsewhere.

The mask to get the exponent field is then

```
data sxxx xxxx xfff ffff ffff ffff ffff ffff
mask 0111 1111 1000 0000 0000 0000 0000 0000
```

Converting the mask to hexadecimal (done by inspection, as always between binary and hexadecimal) gives the mask as $7F800000_{16}$. The other masks follow similarly.

Example using Java .

Given a float value `fVal` (32 bits), we want to get its exponent, significand and sign as bits in integer values, using Java statements. So

Get the bits

```
bits = Float.floatToIntBits(fVal);
```

Get the significand

```
sigBits = bits & 0x7fffff;
```

Insert the significand hidden bit

```
sigBits = sigBits | 0x800000;
```

Get the exponent

```
expBits = bits & 0x7f800000;
```

and align the exponent

```
expBits = (expBits >> 23) - 127;
```

and get the sign bit, and align it

```
sgnBit = (bits >> 31) & 0x00000001;
```

Parity

An important use of Exclusive OR is in "parity checking" to detect corrupted data, whether in memory, disk storage, or in transmission.

In the simplest case, each word has an added *parity bit*, which is set to make the total number of 1's in the word either even (*even parity*) or odd (*odd parity*). If a word is found to have the wrong parity when read, then an error must have occurred.

If the bits of the word are $x_{n-1}, x_{n-2}, \ldots, x_1, x_0$,

for even parity $Pbit_{even} = x_{n-1} \oplus x_{n-2} \oplus \ldots \oplus x_0$, and

for odd parity $Pbit_{odd} = x_{n-1} \oplus x_{n-2} \oplus \ldots \oplus x_0 \oplus 1$.

Better error checking and error correction use more complex forms and patterns of parity checking.

Hamming Code (*not examinable*)

This is the oldest and simplest of the error-correcting codes. Take 4 data bits and 3 parity bits and arrange them in a 7-bit word as $d_7\, d_6\, d_5\, p_4\, d_3\, p_2\, p_1$, with the bits numbered from 7 on the left to 1 on the right.

Set the parity bits as

$p_1 = d_3 \oplus d_5 \oplus d_7$ (the bits with a "1" in the bit number)

$p_2 = d_3 \oplus d_6 \oplus d_7$ (the bits with a "2" in the bit number)

$p_4 = d_5 \oplus d_6 \oplus d_7$ (the bits with a "4" in the bit number)

Transmit the 7-bit word as data. On reception, calculate the *syndrome* $S = \{s_4\, s_2\, s_1\}$ from the equations

$s_1 = p_1 \oplus d_3 \oplus d_5 \oplus d_7$

$s_2 = p_2 \oplus d_3 \oplus d_6 \oplus d_7$

$s_4 = p_4 \oplus d_5 \oplus d_6 \oplus d_7$

If $S = 0$, then the bits are all correct. If S - 0, then it gives the number of the bit in error (assuming just one error). The Hamming code is easily extended to longer words, by using each bit number 2^k as a parity bit. It does not extend to correcting more than one error.

Representation of Characters

Graphics, or printing, characters or text are usually held within an 8-bit byte, one character per byte. The usual code is ASCII (American Standard Code for Information Interchange), which is also very similar to International Alphabet 5 (IA5).

- It is basically a 7-bit code. The 8th (most significant) bit may be 0, 1, or parity. Recent extensions fill other characters into the "top half" of the table space.
- 16-bit extensions allow for Arabic, Hebrew, Chinese etc. (These are discussed later)
- There are 32 "transmission control" and "formatting" characters.
- A "6-bit subset" includes the upper-case letters and the more frequent punctuation symbols.

Table of ASCII character codes

binary	hex	000	001	010	011	100	101	110	111
0000	0	NUL	DLE	SP	0	@	P	`	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS	-	=	M	]	m	}
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

ASCII character codings

The character code is {column: row} ('B' is *x*100 0010, and 'k' *x*110 1011) where *x* is usually 0. The ASCII codes then divide into several groups

1. 000*x xxxx* **Transmission control codes**. The only ones of these which are important for now are
   - CR Carriage Return
   - LF New Line
   - HT Horizontal Tab

The other codes are mostly used in data communications to surround messages and to signal between stations.

2. 001*x xxxx* **Numeric and "specials" or punctuation**.

3. 010*x xxxx* **Upper case letters** (and some punctuation)

4. 011*x xxxx* **Lower case letters**

We can usually assume that successive characters of text will be placed in adjacent bytes of memory and that the text "grows" to higher memory addresses. With 32-bit words (4 characters to a word), the string "A text sample." would be stored as -

	characters	hexadecimal
word 1	`A te`	41 20 74 65
word 2	`xt s`	78 74 20 73
word 3	`ampl`	61 6D 70 6C
word 4	`e.`	65 2E *xx xx*

16-bit codings, or Unicode

UniCode extends ASCII to allow the handling of many different alphabets. Instead of using a basic 8-bit code and escaping into versions for different alphabets, a single unified code covers all alphabets. Unicode is used for Java strings and in some modern operating systems. Unicode is based on "pages" or "blocks" of 128 symbols, where each block is typically allocated to a particular alphabet, as shown in the large table.

ASCII codes, zero-extended to 16 bits, are the first few values. Other 8-bit prefixes identify Arabic, Hebrew, Thai, various Indian alphabets and the accented letters for some central European languages. About half the total space (30,000 symbols) is used for Chinese, Japanese and Korean ideographs.

Code Range	Name	Code Range	Name
U+0000–U+007F	C0 Controls and Basic Latin	U+2190–U+21FF	Arrows
U+0080–U+00FF	C1 Controls & Latin-1 Supplement	U+2200–U+22FF	Mathematical Operators
U+0100–U+017F	Latin Extended-A	U+2300–U+23FF	Miscellaneous Technical
U+0180–U+024F	Latin Extended-B	U+2400–U+243F	Control Pictures
U+0250–U+02AF	IPA Extensions	U+2440–U+245F	Optical Character Recognition
U+02B0–U+02FF	Spacing Modifier Letters	U+2460–U+24FF	Enclosed Alphanumerics
U+0300–U+036F	Combining Diacritical Marks	U+2500–U+257F	Box Drawing
U+0370–U+03FF	Greek	U+2580–U+259F	Block Elements
U+0400–U+04FF	Cyrillic	U+25A0–U+25FF	Geometric Shapes
U+0530–U+058F	Armenian	U+2600–U+26FF	Miscellaneous Symbols
U+0590–U+05FF	Hebrew	U+2700–U+27BF	Dingbats
U+0600–U+06FF	Arabic	U+3000–U+303F	CJK Symbols and Punctuation
U+0900–U+097F	Devanagari	U+3040–U+309F	Hiragana
U+0980–U+09FF	Bengali	U+30A0–U+30FF	Katakana
U+0A00–U+0A7F	Gurmukhi	U+3100–U+312F	Bopomofo
U+0A80–U+0AFF	Gujarati	U+3130–U+318F	Hangul Compatibility Jamo
U+0B00–U+0B7F	Oriya	U+3190–U+319F	Kanbun
U+0B80–U+0BFF	Tamil	U+3200–U+32FF	Enclosed CJK Letters and Months
U+0C00–U+0C7F	Telugu	U+3300–U+33FF	CJK Compatibility
U+0C80–U+0CFF	Kannada	U+4E00–U+9FA5	CJK Ideographs
U+0D00–U+0D7F	Malayalam	U+AC00–U+D7A3	Hangul Syllables
U+0E00–U+0E7F	Thai	U+D800–U+DB7F	High Surrogates
U+0E80–U+0EFF	Lao	U+DB80–U+DBFF	High Private Use Surrogates
U+0F00–U+0FBF	Tibetan	U+DC00–U+DFFF	Low Surrogates
U+10A0–U+10FF	Georgian	U+E000–U+F8FF	Private Use Area
U+1100–U+11FF	Hangul Jamo	U+F900–U+FAFF	CJK Compatibility Ideographs
U+1E00–U+1EFF	Latin Extended Additional	U+FB00–U+FB4F	Alphabetic Presentation Forms
U+1F00–U+1FFF	Greek Extended	U+FB50–U+FDFF	Arabic Presentation Forms-A
U+2000–U+206F	General Punctuation	U+FE20–U+FE2F	Combining Half Marks
U+2070–U+209F	Superscripts and Subscripts	U+FE30–U+FE4F	CJK Compatibility Forms
U+20A0–U+20CF	Currency Symbols	U+FE50–U+FE6F	Small Form Variants
U+20D0–U+20FF	Combining Diacritical Marks	U+FE70–U+FEFF	Arabic Presentation Forms-B
U+2100–U+214F	Letterlike Symbols	U+FF00–U+FFEF	Halfwidth and Fullwidth Forms
U+2150–U+218F	Number Forms	U+FFF0–U+FFFF	Specials

The allocation of Unicode code pages to alphabets

UTF-8 Coding

The standard or "canonical" 16-bit coding is known as UCS-2. An alternative coding, UTF-8, gives a way of representing UCS-2 characters (16-bit) within an 8-bit code stream. ASCII characters are represented unchanged, while others are packed into groups of bytes. A standard ASCII character is emitted "as is" in UTF-8 with a high-order 0 bit. Larger values are broken into 6-bit groups, from the least significant bit. Each group except the most significant is prefixed by the bits "10" and emitted as a byte. The first byte starts with as many 1's as there are bytes in the code, followed by a 0 (sometimes called a unary code). Only 2-byte and 3-byte codes are used for UCS-2 characters. (UTF-8 can also handle 32-bit UCS-4 codes and some extended alphabets.)

data bits	Input bit pattern	coding into successive bytes		
7	0000 0000 0gfe dcba	0gfe dcba		
11	0000 0kji hgfe dcba	110k jihg	10fe dcba	
16	ponm lkji hgfe dcba	1110 ponm	10lk jihg	10fe dcba

UCS-2 and UTF-8 coding.

Some examples of UCS-2 to UTF-8 coding are shown in the table, coding into 1, 2 and 3 bytes respectively.

UCS-2	UTF-8		
(16 bits)	byte 1	byte 2	byte 3
0000 0000 0010 1101	0010 1101		
0000 0011 0101 1110	1100 1101	1001 1110	
0010 1011 0111 1010	1110 0010	1010 1101	1011 1010

To change UCS-2 to UTF-8, the details depend on the number of leading zeros in the UCS-2 coding.

- If there are 9 or more leading zeros (code ≤ 0x7F) the low-order 8 bytes or right-hand byte are taken as the UTF-8 code. This case, and this case only, may be interpreted as an ASCII character.
- If there are 5 - 8 leading zeros, divide the 16 bits of the UCS-2 coding as `0000 0xxx xxyy yyyy` and form the 2 bytes `110x xxxx` and `10yy yyyy`. The two bytes are the UTF-8 code. (Here, as before, the x's and y's may be any mixture of 0 and 1 bits.)
- If there are 4 or fewer leading zeros, divide the UCS-2 coding as `wwww xxxx xxyy yyyy`, and then encode into 3 bytes as `1110 wwww 10 xx xxxx 10yy yyyy`.

To change UTF-8 to UCS-2, the initial bits of each UTF-8 byte encoding determine the interpretation of that and following bytes -

- 0..... The character is encoded in a single byte, equivalent to standard ASCII (or International Alphabet No. 5 - IA5).
- 10..... The following 6 bits are used to continue whatever has been previously emitted for the partial UCS-2 coding. (This byte must be the second or third byte of a 2 or 3 byte group.)
- 110..... Emit 5 leading zeros and then the remaining 5 bits of this byte, as the first 10 bits of the UCS-2 code. One 10... byte must follow.
- 1110 ..... Emit the remaining 4 bits of this byte and then, in order, 6 bits from each of the two following bytes, both of which must start with 10..... .

The UTF-8 bytes 20 E6 98 AF 20 43 61 6C 69 73 20 C7 9A 44 E5 BB B6 convert to UCS-2 according to the following table. (The symbol ¶ denotes a non-ASCII character.) The final UCS2 encoded string is 0020 662F 0020 0043 0061 006C 0069 0073 0020 01DA 0044 5EF6.

UTF-8 Bytes	UCS-2 codes	ASCII
20	0020	sp
E6 98 AF	662F	¶
20	0020	sp
43	0043	C
61	0061	a
6C	006C	l
69	0069	i
73	0073	s
20	0020	sp
C7 9A	01DA	¶
44	0044	D
E5 BB B6	5EF6	¶

Java Code to convert Unicode.

The conversion of Unicode between UCS-2 and UTF-8 gives a good demonstration of Java's bit handling facilities, with combinations of AND, OR and shift operations.

We must note the precedence or priority of the logical operations -

shifts (highest)

== equality (be careful when used with shifts!)

& AND

^ Exclusive OR

| OR (lowest)

In actual programming, people tend to forget the priorities of these relatively rare operations and it is often much clearer to use parentheses to emphasise the groupings.

For these examples, assume the declarations -

```
int  ucs2; // (or short ucs2)
byte utf8[ ];
```

to provide obvious space for the two codings. The code examples will be worked through as a trace of the operation to show the actual bit manipulations.

1. UCS-2 to UTF-8.

The UTF code is put into the first 1, 2, or 3 bytes of the utf8 byte array. Check, in order, for 9 leading zeros (for 1 UTF-8 byte), for 5 leading zeros (for 2 UTF-8 bytes), or else use 3 bytes output. The code should be read in conjunction with the recoding rules above.

```
if ((ucs2 & 0xFF80 == 0)        // 1 byte
  utf8[0] = ucs2 & 0x7f;
else if (ucs2 & 0xf800) == 0)  // 2 bytes
  {
  utf8[0] = 0xc0 | ((ucs2 >> 6) & 0x1f);
  utf8[1] = 0x80 | (ucs2 & 0x3f);
  }
else                           // 3 bytes
  {
  utf8[0] = 0xe0 | ((ucs2 >> 12) & 0x0f);
  utf8[1] = 0x80 | ((ucs2 >>  6) & 0x3f);
  utf8[2] = 0x80 | (ucs2 & 0x3f);
  }
```

Example, UCS-2 to UTF-8

To illustrate this code, consider the conversion of the UCS-2 code 01DA to UTF-8 (the reverse of second to last line in the table earlier.

1. Take UCS2	0000 0001 1101 1010
AND with 0xFF80	1111 1111 1000 0000
Result not zero, try next	0000 0001 1000 0000
2. Take UCS2	0000 0001 1101 1010
AND with 0xF800	1111 1000 0000 0000
Result is zero, proceed	0000 0000 0000 0000
3. Take UCS2	0000 0001 1101 1010
Shift right 6 places	0000 0000 0000 0111
AND with 0x1F	0000 0000 0001 1111
Gives	0000 0000 0000 0111
OR in prefix 0xC0	0000 0000 1100 0000
Get UTF8[0] (to 8 bits)	1100 0111
4. Take UCS2	0000 0001 1101 1010
AND with 0x3F	0000 0000 0011 1111
Gives	0000 0000 0001 1010
OR in prefix 0x80	0000 0000 1000 0000
Get UTF8[1] (to 8 bits)	1001 1010
5. The two UTF-8 bytes are therefore	1100 0111
and	1001 1010
or, in hexadecimal	C7 9A

2. UTF-8 to UCS-2.
The input is now a sequence of 1, 2, or 3 bytes in the UTF-8 array, which must be assembled into the single UCS-2 word.

```
if ((utf8[0] & 0x80) == 0)      // 1 byte
   ucs2 = utf8[0];
else if ((utf8[0] & 0xE0) == 0xc0)// 2 bytes
   ucs2 = ((utf8[0] & 0x1f) << 6)
        | (utf8[1] & 0x3f);
else if ((utf8[0] & 0xf0) == 0xe0)// 3 bytes
   ucs2 = ((utf8[0] & 0x0f) << 12)
        | ((utf8[1] & 0x3f) << 6)
        | ( utf8[2] & 0x3f);
else
// we have an error !
```

Example, UTF-8 to UCS-2

Many of the UTF-8 bytes have values less than 80 (all values in hexadecimal); these are straightforward ASCII characters and are converted to UCS-2 by adding a prefix of 00. A space 20 becomes 0020, and 61 (‘a’) becomes 0061.

The UTF-8 bytes include the sequence E6 98 AF, which will be used as the first example. First, expand the hexadecimal into binary, given with the prefix bits underlined

1110 0110 10 01 1000 10 10 1111.

The first byte expands to the 16-bit value

1111 1111 1110 0110, which we will call UTF0

First and with the mask 0x80 as -

`UTF0`	1111 1111 1110 0110
mask 0x80	0000 0000 1000 0000
AND result	0000 0000 1000 0000is not zero, so the first test fails.

The second test ANDs with the mask 0xE0 as -

`UTF0`	1111 1111 1110 0110
mask 0xE0	0000 0000 1110 0000
AND result	0000 0000 1110 0000
compare with 0xc0	0000 0000 1100 0000is not equal, so this test fails too.

Finally use the mask 0xF0 -

`UTF0`	1111 1111 1110 0110
mask 0xF0	0000 0000 1111 0000
AND result	0000 0000 1110 0000
compare with 0xe0	0000 0000 1110 0000is equal, so this test succeeds.

Select the data bits of UTF0 -

`UTF0`	1111 1111 1110 0110
mask 0x0F	0000 0000 0000 1111
AND result	0000 0000 0000 0110
shift left 12, to get **A**	0110 0000 0000 0000 **A**

Take the second UTF-8 byte, expanded to 16 bits

`UTF1`	1111 1111 1001 1000
mask 0x3F	0000 0000 0011 1111
AND result	0000 0000 0001 1000
shift left 6, to get **B**	0000 0110 0000 0000 **B**

Take the third UTF-8 byte, expanded to 16 bits

`UTF2`	1111 1111 1010 1111
mask 0x3F	0000 0000 0011 1111
AND result, to get **C**	0000 0000 0010 1111 **C**

OR together the three values A, B and C for the final UCS-2 value

A	0110 0000 0000 0000
B	0000 0110 0000 0000
C	0000 0000 0010 1111
OR result, for UCS-2	0110 0110 0010 1111

Floating Point numbers

Ordinary binary integers cover a relatively limited range of values about ±32,767 for 16-bits and ±2,147,483,647 for 32 bits. Many real-world problems cover a wider range of values than this, from very small values to very large values. These "scientific" values are represented in most computers by "real" or "floating point" numbers.

Many calculations with only addition and subtraction can be done with integers. Multiplication quickly outgrows the integer range (overflow), while division almost always generates either a remainder or a number with a repeating fraction. Thus real values or floating point numbers arise naturally in many problems.

The representation is similar to "scientific" or "exponent" notation for numbers, which also allows very large and very small numbers to be written easily. For example, the velocity of light is 299,792,458 ms^{-1}, or 2.99792458×10^8 ms^{-1}. For most practical purposes we can use the approximate values 300,000,000 ms^{-1}, or 3×10^8 ms^{-1}. The decimal point is normally to the right of the last digit (the 8). To convert to scientific, the point is first shifted so that it follows the first digit, a shift of 8 places. To correct for this shift, a multiplier of 10^8 is included. For very large and very small numbers the scientific notation is very efficient. Thus the charge on the electron is about 0.000000000000000000016021892 C which is 1.6021892×10^{-19} C. Similarly, the number of atoms in one mole of gas is about 602204500000000000000000 or 6.022045×10^{23}. The scientific form makes things easier in several ways -

- The exponent tells very quickly how large the number is (+ve exponent) or how small (-ve exponent). It also reduces the problems in counting many following or preceding zeros (I hope the counting was correct in the two examples!).
- The number of digits tells how accurately the value is known. Thus a value of 98270 known to an accuracy of ±10 could be written 9.827×10^4 (the units digit is not certain), whereas if the last digit is certain, it would be written as 9.8270×10^4. Writing the speed of light as 3×10^8 ms^{-1} means that we worry about only that first digit, whereas writing it as 3.00×10^8 ms^{-1} means that the first 3 digits are correct. (Writing to another digit must be 2.998×10^8 ms^{-1} when the last digit is rounded.)

Computer real numbers are held in a similar way, except that all values are normally binary. A real value is held as two parts -

1. The significand, fraction or mantissa is usually about 24 or 50 bits and usually gives a value $0.5 \le V < 1.0$, with the binary point at or near the left-most bit.

2. The exponent or characteristic is a smaller 8 or 12 bit value which gives a multiplier for the significand. For most numbers the value for a significand S and an exponent E, is $S\times2^E$. The value could be written as a binary value with integral and fractional parts; the exponent tells by how many bits the binary point must be shifted.

Normalisation

The value 2.99792458×10^8 could be written as 0.299792458×10^9, perhaps $0.00299792458\times10^{11}$, or 29.9792458×10^7 - all are equivalent. By convention, scientific numbers are always written with one digit before the point, giving a "normalised" representation. Binary floating point numbers are similarly *normalised* to the binary value 0.1..., or 1. *xx*... by balancing a left shift of the digits with a decrease in the exponent (or a right shift of the digits with an increase in the exponent).

Floating point representation

Internally, most computers use a base of 2 (ie the fraction is multiplied by $2^{exponent}$), or less often 16 or 8. The significand is usually held in sign & magnitude form, with the exponent in say excess 127. A 0.0 value is an all-0 word, by convention

Important points to remember are

- Do not confuse the *range* and the *precision* of floating point numbers.
- The *range* is determined by the exponent and determines how close to zero or far from zero a number may be. It is closely connected to the exponent form of scientific notation. An 8-bit signed exponent can have values close to the range -128 to +127. The smallest representable number will be about 2^{-128} and the largest 2^{+127}. Remembering that $\log_2 10 \approx 1/0.3$, the number range is about from 10^{-38} to 10^{+38}. It is shown later that this range is quite inadequate for some calculations.
- The *precision* is governed by the significand (or fraction or mantissa) and gives the accuracy with which a number may be represented. Remember that N bits equals about $0.3 \times N$ decimal digits. A standard "32-bit real" has 23-bit precision, or not quite 7 decimal digits. A 64-bit "double" has 52-bits or 15 decimal digits. Even a 32 bit real can handle the accuracy of most physical measurements, but much of the precision is lost by rounding in lengthy calculations; this is the real justification for using 64-bit or 128-bit floating point numbers.

Floating point arithmetic is subject to rounding and truncation errors. The significand can represent only so many bits; any less significant bits must be discarded. Often, if the first discarded bit is a 1, we add 1 onto the significand to "round" the result. Thus 1.7 would round to 2, which is probably a better result than 1 (from just forgetting the bits).

Care is needed when using real-number arithmetic. Some of the problems seem to disappear with "long" numbers, but really stay there and are never more than reduced.

- The 32 bit floating point "real" on many computers is quite limited in comparison with many scientific calculators. Its range is about $10^{\pm 38}$, and its precision is not quite 7 decimal digits. Even short calculation sequences can overwhelm it.
- Arithmetic with floating point numbers is seldom exact and great care must be taken in long calculation sequences as "round-off" errors accumulate. For example a solution of a set of 40 simultaneous equations had the 3-4 least significant decimal digits quite meaningless.
- Beware of mathematical techniques which involve differences of large quantities. This is related to the previous point. Say we have two values close to 1000, both with the last decimal digit uncertain, such as 102*x* and 99*x* (both known to about 10 parts in 1000, or 1%). Subtracting gives a value 3*x*, where the last digit is still uncertain, but the error is now 10 parts in 30, or about 30%. Two moderately accurate values have combined to give a value which is nearly meaningless. Some types of statistical calculation are especially sensitive.
- The result is that floating point arithmetic is not exact. Because of possible disappearance of low-order bits we cannot guarantee that (A+B)+C = A+(B+C). In most cases it is very nearly true, if we are careful, but in extreme cases it is anything but true.
- Be very careful if using floating point arithmetic for financial calculations. Rounding errors may make it almost impossible to achieve reliable balances, especially if the number precision is barely adequate to represent the whole amount.

IEEE 754 Floating point representation

Most computer manufacturers used to develop their own floating point representations for their own computers. Not only were they different, but many had serious design errors. The IEEE 754 standard attempts to overcome these problems and has been adopted in most modern computers.

The IEEE 754 standard defines several number formats and precisions. The 32 bit format has a 1-bit sign, an 8-bit exponent with a bias of 127, and a 23-bit significand. The significand is always stored in "normalised" form with its most significant bit "1". As this bit is always a 1 it is redundant and can be omitted from the stored number and automatically inserted in the arithmetic unit when calculations are to be done. The bits are used as -

```
sxxx xxxx xfff ffff ffff ffff ffff ffff
```

where `s` is the sign bit, `x...x` are the exponent bits and `f...f` the significand bits. The value of a number is then

$$(-1)^{sign} \times (1.0+significand) \times 2^{(exponent-127)}$$

The IEEE 754 standard has quite complicated rules on the rounding of numbers. It also has ways of representing underflowed and overflowed numbers and special error values called "Not a Number" (NaN), from cases like 0/0 or √-3. Normalisation is also rather more complicated than is described here, to handle a "gradual underflow".

Double precision numbers

As described earlier, the 32-bit representation is barely adequate for serious computation; the precision is limited and "rounding" errors accumulate very quickly. Some computations lasting only a second or two can become quite meaningless. Also, the number range of $10^{\pm 38}$ is too small to handle some physical quantities, or formulas involving them. The 754 standard therefore includes a 64-bit representation to overcome these problems.

In Java 32-bit quantities are of type **`float`**, while 64-bit are **`double`**. Floating point results are normally produced with type **`double`** and a cast is necessary to store into a **`float`** variable.

IEEE 754 double precision uses a 53-bit significand (giving about 16 decimal digits of precision) and an 11-bit exponent with a bias of 1023 (a range of about $10^{\pm 300}$). The underlying principles are as for the 32-bit representation.

The following is an example to show the problem of exponent range; you need not worry what it means but should remember the lesson!

Problems of exponent range easily arise in physics, where many of the "fundamental" constants are very small quantities. In the Bohr model of the hydrogen atom, the energy E_n of the n'th stationary state is given by

$$E_n = \frac{m_e e^4}{8\varepsilon_0{}^2 h^2} \cdot \frac{1}{n^2}$$

where $m_e = 9.109534 \times 10^{-31}$ mass of electron

$e = 1.6021892 \times 10^{-19}$ charge of an electron

$\varepsilon_0 = 8.85419 \times 10^{-12}$ permittivity of space

$h = 6.626176 \times 10^{-34}$ Plank's constant

The numerator (top line) of the fraction is 6×10^{-106}, and the denominator (bottom line) is 2.75×10^{-88}. Both values are well outside the range of a 32-bit number and indeed many calculators, but are easily handled by a number in the 64-bit format.

Floating point in Java

Java provides both single (32-bit) and double (64-bit) floating point types, with the default being double. The java.lang.Double class provides methods similar to those for "integer type" values, but extended as necessary.

The Class methods include

- public static long doubleToLongBits(double value) returns the actual bit pattern of the IEEE 754 representation of the value. (It is useful for analysing or "dismantling" a floating point number.)
- public static boolean isInFinite(double value) returns TRUE if the value is an infinity.
- public static boolean isNaN(double value) returns TRUE if the value is a "Not a Number".
- public static double longBitsToDouble(long bits) takes the bit pattern in bits and returns it as a double value. (It is useful for building a floating point number.)
- public static String toString(double d) returns a String corresponding to the value of d.
- public static Double valueOf(String s) converts the String s to a double precision value (may throw an exception).

The Instance methods follow those for "integer type" values and again "wrap around" an existing value, such as `Double(doubleVal).byteValue()`. Two methods specific to floating point are added; both are duplicated from the class methods.

- public byte byteValue() converts to a byte value
- public short shortValue() converts to a short value
- public int intValue() converts to an int value
- public long longValue() converts to a long value (!!)
- public float floatValue() converts to a float value (32-bit floating point)
- public double doubleValue() converts to a double value (64-bit floating point)
- public String toString() converts to a String
- public boolean isInFinite() returns TRUE if the value is an infinity.
- public boolean isNaN() returns TRUE if the value is a "Not a Number".

Casting is also useful.

- Any integer-type can be cast to a float or double.example `floatVal = (float) i / j` performs a floating point division, rather than an integer division. (There is an explicit cast of `i` and an implicit cast of `j`.)
- A float or double can be cast to an integer-type provided that its integral part is within the range of the destination.

Floating Point number examples

Decimal value 1.25.

Written as a binary value with one bit to the left of the binary point, 1.25 becomes 1.0100000000.... Dropping the first bit gives .0100000000... as the significand.
Now 1.25 is already normalised, without any shifting or alignment necessary and the number has an exponent of zero. Adding the bias of 127 gives a field value of 127, or 01111111, in the representation of the number. With a positive sign of 0, the representation becomes
01111111 01000000000000000000. Converting to hexadecimal gives **3FA00000.**

Decimal value 14.0.

Written as a binary value, 14.0 is 1110.0000000000....

To normalise, the binary point must be shifted three places to the *left*, giving a true exponent of +3 and a fraction of 1.1100000000..., which with the leading bit omitted is represented as .1100000000....

Adding the bias of 127 to the exponent (+3) gives a represented value of 130, or a bit pattern of 10000010. With a sign of 0, the whole bit pattern becomes
10000010 11000000000000000000. to hexadecimal gives **41600000.**

Decimal value 0.1

In binary, 0.1 is 0.0001100110011001100110011001100...
To normalise, the binary point must be shifted four places to the *right*, giving a true exponent of -4 and a fraction of 1.10011001100110011001100..., .10011001100110011001100... with the leading bit omitted. Adding the bias of 127 to the exponent (-4) gives a represented value of 123, or a bit pattern of 01111011. With a sign of 0, the whole bit pattern becomes
01111011 10011001100110011001100.
Converting to hexadecimal gives 3DCCCCCC. As the bits immediately following this pattern are 1100..., the last digit is "rounded" up to the next digit, or D, giving a final representation **3DCCCCCD**.

The IEEE754 Applet shows the representations of floating point numbers, for both 32-bit and 64-bit precisions.