COMPUTER SCIENCE (868)

Aims (Conceptual)

(1) To understand algorithmic problem solving using data abstractions, functional and procedural abstractions, and object based and object oriented abstractions.

(2) To understand: (a) how computers represent, store and process data by studying the architecture and machine language of a simple microprocessor and the different levels of abstraction that mediate between the machine and the algorithmic problem solving level and (b) how they communicate with the outside world.

(3) To create awareness of ethical problems and issues related to computing.

Aims (Skills)

To devise algorithmic solutions to problems and to be able to code, validate, document, execute and debug the solution using the Java programming system.

CLASS XI

There will be two papers in the subject:

Paper I: *Theory -* *3 hours ….100 marks*

Paper II: *Practical -* *3 hours ….100 marks*

PAPER I -THEORY

Paper 1 shall be of 3 hours duration and be divided into two parts.

Part I (30 marks): *This part will consist of compulsory short answer questions, testing knowledge, application and skills relating to the entire syllabus.*

Part II (70 marks): *This part will be divided into three Sections, A, B and C. Candidates are required to answer* ***three*** *questions out of* ***four*** *from Section A and* ***two*** *questions out of* ***three*** *in each of the Sections B and C. Each question in this part shall carry 10 marks.*

SECTION A

Basic Computer Hardware and Software

1. Numbers

Representation of numbers in different bases and interconversion between them (e.g. binary, octal, decimal, hexadecimal). Addition and subtraction operations for numbers in different bases.

Introduce the positional system of representing numbers and the concept of a base. Discuss the conversion of representations between different bases using English or pseudo code. These algorithms are also good examples for defining different functions in a class modelling numbers (when programming is discussed). For addition and subtraction use the analogy with decimal numbers, emphasize how carry works (this will be useful later when binary adders are discussed).

2. Encodings

(a) Binary encodings for integers and real numbers using a finite number of bits (sign-magnitude, twos complement, mantissa-exponent notation). Basic operations on integers and floating point numbers. Limitations of finite representations.

Signed, unsigned numbers, least and most significant bits. Sign-magnitude representation and its shortcomings (two representations for 0, addition requires extra step); twos-complement representation. Operations (arithmetic, logical, shift), discuss the basic algorithms used for the arithmetic operations. Floating point representation: normalized scientific notation, mantissa-exponent representation, binary point (discuss trade-off between size of mantissa and exponent). Single and double precision. Arithmetic operations with floating point numbers. Properties of finite representation: overflow, underflow, lack of associativity (demonstrate this through actual programs).

(b) Characters and their encodings (e.g. ASCII, Unicode).

Discuss the limitations of the ASCII code in representing characters of other languages. Discuss the Unicode representation for the local language. Java uses Unicode, so strings in the local language can be used (they can be displayed if fonts are available) – a simple table lookup for local language equivalents for Latin (i.e. English) character strings may be done. More details on Unicode are available at www.unicode.org.

3. High level structure of computer

Block diagram of a computer system with details of (i) function of each block and (ii) interconnectivity and data and control flow between the various blocks

Develop the diagram by successive refinement of blocks till all the following have been covered: ALU, RAM, cache, the buses (modern computers have multiple buses), disk (disk controller and what it does), input/output ports (serial, parallel, USB, network, modem, line-in, line-out etc.), devices that can be attached to these ports (e.g keyboard, mouse, monitor, CDROM, DVD, audio input/output devices, printer, etc.). Clearly describe the connectivity and the flow of data and control signals.

4. Basic architecture of a simple processor and its instruction set

Simple Hypothetical Computer.

The simple hypothetical computer abbreviated as (SHC) is meant to introduce the basic structure of a processor in particular registers, basic instruction set, structure of an instruction, program counter addressing modes (immediate, direct, register, register-indirect). Simple programs should be written in the SHC instruction set (e.g. max./min. of set of nos.)

5. Propositional logic, hardware implementation, arithmetic operations

(a) Propositional logic, well formed formulae, truth values and interpretation of well formed formulae, truth tables.

*Propositional variables; the common logical connectives (~ (not)(negation), $\wedge$ (and)(conjunction), $\vee$ (or)(disjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence)); definition of a well-formed formula (wff); representation of simple word problems as wff (this can be used for motivation); the values **true** and **false**; interpretation of a wff; truth tables; satisfiable, unsatisfiable and valid formulae.*

(b) Logic and hardware, basic gates (AND, NOT, OR) and their universality, other gates (NAND, NOR, XOR); inverter, half adder, full adder.

Show how the logic in (a) above can be realized in hardware in the form of gates. These gates can then be combined to implement the basic operations for arithmetic. Tie up with the arithmetic operations on integers discussed earlier in 2 (a).

6. Memory

Memory organization and access; parity; memory hierarchy - cache, primary memory, secondary memory.

The access time differences between the different kinds of memory; size differences; locality of reference and cache memory.

7. System and other software

Boot process. Operating system as resource manager, command processing, files, directories and file system. Commonly available programs (editors, compilers, interpreters, word processors, spread sheets etc.).

Boot process step-by-step from power on till the prompt. In OS discuss: (i) all the resources (processor, memory, i/o) that need to be managed in a computer; (ii) what is meant by managing these resources. Logical structure of data storage on disk using logical disks, hierarchical directories and files. Distinguish between interpreters and compilers. In particular discuss the javac and java programs.

SECTION B

The programming element in the syllabus is aimed at algorithmic problem solving and **not** merely rote learning of Java syntax. The Java version used should be 1.5 or later. For programming, the students can use any text editor and the javac and java programs or any development environment: for example, BlueJ, Eclipse, NetBeans etc. BlueJ is strongly recommended for its simplicity, ease of use and because it is very well suited for an 'objects first' approach.

8. Introduction to algorithmic problem solving using Java

Note that topics 9 to 13 will get introduced almost simultaneously when classes and their definitions are introduced.

9. Objects

(a) Objects as data (attributes) + behaviour (methods or functions); object as an instance of a class. Constructors.

Difference between object and class should be made very clear. BlueJ (www.bluej.org) and Greenfoot (www.greenfoot.org) can be used for this purpose. Constructor as a special kind of function; the new operator; multiple constructors with different argument structures; constructor returns a reference to the object.

(b) Analysis of some real world programming examples in terms of objects and classes.

Use simple examples like a calculator, date, number etc. to illustrate how they can be treated as objects that behave in certain well-defined ways and how the interface provides a way to access behaviour. Illustrate behaviour changes by adding new functions, deleting old functions or modifying existing functions.

10. Primitive values, wrapper classes, types and casting

Primitive values and types: int, short, long, float, double, boolean, char. Corresponding wrapper classes for each primitive type. Class as type of the object. Class as mechanism for user defined types. Changing types through user defined casting and automatic type coercion for some primitive types.

Ideally, everything should be a class; primitive types are defined for efficiency reasons; each primitive type has a corresponding wrapper class. Classes as user defined types. In some cases types are changed by automatic coercion or casting – e.g. mixed type expressions. However, casting in general is not a good idea and should be avoided, if possible.

11. Variables, expressions

Variables as names for values; expressions (arithmetic and logical) and their evaluation (operators, associativity, precedence). Assignment operation; difference between left hand side and right hand side of assignment.

Variables denote values; variables are already defined as attributes in classes; variables have types that constrain the values it can denote. Difference between variables denoting primitive values and object values – variables denoting objects are references to those objects. The assignment operator = is special. The variable on the lhs of = denotes the memory location while the same variable on the rhs denotes the contents of the location e.g. i=i+2.

12. Statements, scope

Statements; conditional (if, if-then-else, switch-break, ?: ternary operator), looping (for, while-do, do-while, continue, break); grouping statements in blocks, scope and visibility of variables.

Describe the semantics of the conditional and looping statements in detail. Evaluation of the condition in conditional statements (esp. difference between || and | and && and &). Emphasize fall through in switch statement. Many small examples should be done to illustrate control structures. Printing different kinds of

patterns for looping is instructive. When number of iterations are known in advance use the for loop otherwise the while-do or do-while loop. Express one loop construct using the others. For e.g.:

for (<init>; <test>; <inc>) <stmt>; is equivalent to:

(i) Using while

<init>; while <test> {<stmt>; <inc> }

(ii) Using do-while

<init>; if !<test> do <stmt>; <inc> while <test>;

Nesting of blocks. Variables with block scope, function scope, class scope. Visibility rules when variables with the same name are defined in different scopes.

13. Functions

Functions/methods (as abstractions for complex user defined operations on objects), functions as mechanisms for side effects; formal arguments and actual arguments in functions; different behaviour of primitive and object arguments. Static functions and variables. The **this** variable. Examples of algorithmic problem solving using functions (various number theoretic problems, finding roots of algebraic equations).

*Functions are like complex operations where the object is implicitly the first argument. Variable **this** denotes the current object. Functions typically return values, they may also cause side-effects (e.g. change attribute values of objects) – typically functions that are only supposed to cause side-effects return void (e.g. Set functions). Java passes argument by value. Illustrate the difference between primitive values and object values as arguments (changes made inside functions persist after the call for object values). Static definitions as class variables and class functions visible and shared by all instances. Need for static functions and variables. Introduce the main method – needed to begin execution.*

14. Arrays, strings

(a) Structured data types – arrays (single and multi-dimensional), strings. Example algorithms that use structured data types (e.g. searching, finding maximum/minimum, sorting techniques, solving systems of linear equations, substring, concatenation, length, access to char in string, etc.).

Storing many data elements of the same type requires structured data types – like arrays. Access in arrays is constant time and does not depend on the number of elements. Sorting techniques (bubble, selection, insertion), Structured data types can be defined by classes – String. Introduce the Java library String class and the basic operations on strings (accessing individual characters, various substring operations, concatenation, replacement, index of operations).

(b) Basic concept of a virtual machine; Java virtual machine; compilation and execution of Java programs (the javac and java programs).

The JVM is a machine but built as a program and not through hardware. Therefore it is called a virtual machine. To run, JVM machine language programs require an interpreter (the java program). The advantage is that such JVM machine language programs (.class files) are portable and can run on any machine that has the java program.

(c) Compile time and run time errors; basic concept of an exception, the Exception class, catch and throw.

Differentiate between compile time and run time errors. Run time errors crash the program. Recovery is possible by the use of exceptions. Explain how an exception object is created and passed up until a matching catch is found. This behaviour is different from the one where a value is returned by a deeply nested function call. It is enough to discuss the Exception class. Sub-classes of Exception can be discussed after inheritance has been done in Class XII.

SECTION C

15. Elementary data structures and associated algorithms, basic input/output

(a) Class as a contract; separating implementation from interface; encapsulation; private and public.

Class is the basic reusable unit. Its function prototypes (i.e. the interface) work as a visible contract with the outside world since others will use these functions in their programs. This leads to encapsulation (i.e. hiding implementation information) which in turn leads to the use of private and public for realizing encapsulation.

(b) Interfaces in Java; implementing interfaces through a class; interfaces for user defined implementation of behaviour.

Motivation for interface: often when creating reusable classes some parts of the exact implementation can only be provided by the final end user. For example in a class that sorts records of different types the exact comparison operation can only be provided by the end user. Since only he/she knows which field(s) will be used for doing the comparison and whether sorting should be in ascending or descending order be given by the user of the class.

Emphasize the difference between the Java language construct interface and the word interface often used to describe the set of function prototypes of a class.

(c) Basic data structures (stack, queue, dequeue); implementation directly through classes; definition through an interface and multiple implementations by implementing the interface. Basic algorithms and programs using the above data structures.

A data structure is a data collection with well defined operations and behaviour or properties. The behaviour or properties can usually be expressed formally using equations or some kind of logical formulae. Consider for e.g. a stack with operations defined as follows:

void push(Object o)

Object pop()

boolean isEmpty()

Object top()

Then, for example the LIFO property can be expressed by (assume s is a stack):

if s.push(o); o1=pop() then o ≡ o1

What the rule says is: if o is pushed on the stack s and then it is popped and o1 is the object obtained then o, o1 are identical.

Another useful property is:

if s.isEmpty() == true then s.pop() = ERROR

It says that popping an empty stack gives ERROR.

Similarly, several other properties can also be specified. It is important to emphasize the behavioural rules or properties of a data structure since any implementation must guarantee that the rules hold.

Some simple algorithms that use the data structures:

i) For stack: parentheses matching, tower of Hanoi, nested function calls; solving a maze.

ii) For queue: scheduling processes, printers, jobs in a machine shop.

(d) Basic input/output using Scanner and Printer classes from JDK; files and their representation using the File class, file input/output; input/output exceptions. Tokens in an input stream, concept of whitespace, extracting tokens from an input stream (StringTokenizer class).

The Scanner class can be used for input of various types of data (e.g. int, float, char etc.) from the standard input stream or a file input stream. The File class is used model file objects in the underlying system in an OS independent manner. Similarly, the Printer class handles output. Only basic input and output using these classes should be covered.

Discuss the concept of a token (a delimited continuous stream of characters that is meaningful in the application program – e.g. words in a sentence where the delimiter is the

blank character). This naturally leads to the idea of delimiters and in particular whitespace and user defined characters as delimiters. As an example show how the StringTokenizer class allows one to extract a sequence of tokens from a string with user defined delimiters.

(e) Concept of recursion, simple recursive functions (e.g. factorial, GCD, binary search, conversion of representations of numbers between different bases).

Many problems can be solved very elegantly by observing that the solution can be composed of solutions to 'smaller' versions of the same problem with the base version having a known simple solution. Recursion can be initially motivated by using recursive equations to define certain functions. These definitions are fairly obvious and are easy to understand. The definitions can be directly converted to a program. Emphasize that any recursion must have a base case. Otherwise, the computation can go into an infinite loop. Illustrate this by removing the base case and running the program. Examples:

(i) Definition of factorial:

factorial(0) = 1 //base case

*factorial(n) = n * factorial(n-1)*

(ii) Definition of GCD:

gcd(m, n) =

if (m==n) then n //base case

else if (m>n) then gcd(m-n, n)

else gcd(m, n-m)

(iii) Definition of Fibonacci numbers:

fib(0) = 1 //base case

fib(1) = 1 //base case

fib(n) = fib(n-1)+ fib(n-2)

The tower of Hanoi is a very good example of how recursion gives a very simple and elegant solution where as non-recursive solutions are quite complex. Discuss the use of a stack to keep track of function calls. The stack can also be used to solve the tower of Hanoi problem non-recursively.

(f) Concrete computational complexity; concept of input size; estimating complexity in terms of functions; importance of dominant term; best, average and worst case.

Points to be given particular emphasis:

(i) Algorithms are usually compared along two dimensions – amount of space (that is memory) used and the time taken. Of the two the time taken is usually considered the more important. The motivation to study time complexity is to compare different algorithms and use the one that is the most efficient in a particular situation.

(ii) Actual run time on a particular computer is not a good basis for comparison since it depends heavily on the speed of the computer, the total amount of RAM in the computer, the OS running on the system and the quality of the compiler used. So we need a more abstract way to compare the time complexity of algorithms.

(iii) This is done by trying to approximate the number of operations done by each algorithm as a function of the size of the input. In most programs the loops are important in deciding the complexity. For example in bubble sort there are two nested loops and in the worst case the time taken will be proportional to n(n-1) where n is the number of elements to be sorted. Similarly, in linear search in the worst case the target has to be compared with all the elements so time taken will be proportional to n where n is the number of elements in the search set.

(iv) In most algorithms the actual complexity for a particular input can vary. For example in search the number of comparisons can vary from 1 to n. This means we need to study the best, worst and average cases. Comparisons are usually made taking the worst case. Average cases are harder to estimate since it depends on how the data is distributed. For example in search, if the elements are uniformly distributed it will take on the average n/2 comparisons when the average is taken over a statistically significant number of instances.

(v) Comparisons are normally made for large values of the input size. This means that the dominant term in the function is the important term. For example if we are looking at bubble sort and see that time taken can be estimated as: $a*n^2 + b*n + c$ where n is the number of elements to be sorted and a, b, c are constants then for large n the dominant term is clearly n^2 and we can in effect ignore the other two terms.

16. Implementation of algorithms to solve problems

The students are required to do lab assignments in the computer lab concurrently with the lectures. Programming assignments should be done such that each major topic is covered in at least one assignment. Assignment problems should be designed so that they are non-trivial and make the student do algorithm design, address correctness issues, implement and execute the algorithm in Java and debug where necessary.

Self explanatory.

17. Social context of computing and ethical issues

(a) Intellectual property and corresponding laws and rights, software as intellectual property.

(b) Software copyright and patents and the difference between the two; trademarks; software licensing and piracy.

(c) Free software foundation and its position on software, open source software, various types of licensing (e.g. GPL, BSD).

(d) Privacy, email etiquette, spam, security issues, phising.

Social impact and ethical issues should be discussed and debated in class. The important thing is for students to realise that these are complex issues and there are multiple points of view on many of them and there is no single 'correct' or 'right' view.

PAPER II - PRACTICAL

This paper of three hours duration will be evaluated internally by the school.

The paper shall consist of three programming problems from which a candidate has to attempt any one. The practical consists of the two parts:

(1) Planning Session

(2) Examination Session

The total time to be spent on the Planning session and the Examination session is three hours. After completing the Planning session the candidates may begin with the Examination session. A maximum of 90 minutes is permitted for the Planning session. However, if the candidates finish earlier, they are to be permitted to begin with the Examination session.

Planning Session

The candidates will be required to prepare an algorithm and a hand written Java program to solve the problem.

Examination Session

The program handed in at the end of the Planning session shall be returned to the candidates. The candidates will be required to key-in and execute the Java program on seen and unseen inputs individually on the Computer and show execution to the examiner. A printout of the program listing, including output results should be attached to the answer script containing the algorithm and handwritten program. This should be returned to the examiner. The program should be sufficiently documented so that the algorithm, representation and development process is clear from reading the program. Large differences between the planned program and the printout will result in loss of marks.

Teachers should maintain a record of all the assignments done as part of the practical work through the year and give it due credit at the time of cumulative evaluation at the end of the year. Students are expected to do a **minimum** of twenty assignments for the year.

Marks (out of a total of 100) should be distributed as given below:

Continuous Evaluation

Candidates will be required to submit a work file containing the practical work related to programming assignments done during the year.

Programming assignments done throughout the year

(Internal evaluation) - 20 marks

Terminal Evaluation

Solution to programming problem on the computer - 60 marks

(Marks should be given for choice of algorithm and implementation strategy, documentation, correct output on known inputs mentioned in the question paper, correct output for unknown inputs available only to the examiner.)

Viva-voce - 20 marks

(Viva-voce includes questions on the following aspects of the problem attempted by the student: the algorithm and implementation strategy, documentation, correctness, alternative algorithms or implementations. Questions should be confined largely to the problem the student has attempted).

CLASS XII

There will be two papers in the subject:

***Paper I:** Theory- 3 hours …100 marks*

***Paper II:** Practical- 3 hours …100 marks*

PAPER I-THEORY

Paper 1 shall be of 3 hours duration and be divided into two parts.

***Part I (30 marks):** This part will consist of compulsory short answer questions, testing knowledge, application and skills relating to the entire syllabus.*

***Part II (70 marks):** This part will be divided into three Sections, A, B and C. Candidates are required to answer **three** questions out of **four** from Section A and **two** questions out of **three** in each of the Sections B and C. Each question in this part shall carry 10 marks.*

SECTION A

1. Boolean Algebra

(a) Propositional logic, well formed formulae, truth values and interpretation of well formed formulae (wff), truth tables, satisfiable, unsatisfiable and valid formulae. Equivalence laws and their use in simplifying wffs.

*Propositional variables; the common logical connectives (~ (not)(negation), ∧ (and)(conjunction), ∨ (or)(disjunction), ⇒ (implication), ⇔ (biconditional); definition of a well-formed formula (wff); representation of simple word problems as wff (this can be used for motivation); the values **true** and **false**; interpretation of a wff; truth tables; satisfiable, unsatisfiable and valid formulae.*

*Equivalence laws: commutativity of ∧, ∨; associativity of ∧, ∨; distributivity; de Morgan's laws; law of implication ($p \Rightarrow q \equiv \sim p \vee q$); law of biconditional ($(p \Leftrightarrow q) \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$); identity ($p \equiv p$); law of negation ($\sim(\sim p) \equiv p$); law of excluded middle ($p \vee \sim p \equiv$ **true**); law of contradiction ($p \wedge \sim p \equiv$ **false**); tautology and contingency simplification rules for ∧, ∨. Converse, inverse and contra positive. Chain rule(Modus ponen).*

$p \vee p \equiv p$ $\quad$ $p \wedge p \equiv p$

$p \vee$ ***true*** $\equiv$ ***true*** $\quad$ $p \wedge$ ***true*** $\equiv p$

$p \vee$ ***false*** $\equiv p$ $\quad$ $p \wedge$ ***false*** $\equiv$ ***false***

$p \vee (p \wedge q) \equiv p$ $\quad$ $p \wedge (p \vee q) \equiv p$

The equivalence rules can be used to simplify propositional wffs, for example:

1) $(p \Rightarrow q) \wedge (p \Rightarrow r)$ to $p \Rightarrow (q \wedge r)$

*2) $((p \Rightarrow q) \wedge p) \Rightarrow q$ to **true***

etc.

(b) Binary valued quantities; basic postulates of Boolean algebra; operations AND, OR and NOT; truth tables.

(c) Basic theorems of Boolean algebra (e.g. Duality, idempotence, commutativity, associativity, distributivity, operations with 0 and 1, complements, absorption, involution); De Morgan's theorem and its applications; reducing Boolean expressions to sum of products and product of sums forms; Karnaugh maps (up to four variables).

Verify the laws of boolean algebra using truth tables. Inputs, outputs for circuits like half and full adders, majority circuit etc., SOP and POS representation; Maxterms & Minterms, Canonical and Cardinal representation, reduction using Karnaugh maps and boolean algebra.

2. Computer Hardware

(a) Elementary logic gates (NOT, AND, OR, NAND, NOR, XOR, XNOR) and their use in circuits.

(b) Applications of Boolean algebra and logic gates to half adders, full adders, encoders, decoders, multiplexers, NAND, NOR as universal gates.

Show the correspondence between boolean functions and the corresponding switching circuits or gates. Show that NAND and NOR gates are universal by converting some circuits to purely NAND or NOR gates.

SECTION B

The programming element in the syllabus (Sections B and C) is aimed at algorithmic problem solving and **not** merely rote learning of Java syntax. The Java version used should be 1.5 or later. For programming, the students can use any text editor and the javac and java programs or any development environment: for example, BlueJ, Eclipse, NetBeans etc. BlueJ is strongly recommended for its simplicity, ease of use and because it is very well suited for an 'objects first' approach.

3. Implementation of algorithms to solve problems

The students are required to do lab assignments in the computer lab concurrently with the lectures. Programming assignments should be done such that each major topic is covered in at least one assignment. Assignment problems should be designed so that they are non-trivial and make the student do algorithm design, address correctness issues, implement and execute the algorithm in Java and debug where necessary.

Self explanatory.

4. Programming in Java (Review of Class XI Sections B and C)

Note that items 4 to 8 will get introduced almost simultaneously when classes and their definitions are introduced.

5. Objects

(a) Objects as data (attributes) + behaviour (methods or functions); object as an instance of a class. Constructors.

Difference between object and class should be made very clear. BlueJ (www.bluej.org) and Greenfoot (www.greenfoot.org) can be profitably used for this purpose. Constructor as a special kind of function; the new operator; multiple constructors with different argument structures; constructor returns a reference to the object.

(b) Analysis of some real world programming examples in terms of objects and classes.

Use simple examples like a calculator, date, number, etc. to illustrate how they can be treated as objects that behave in certain well-defined ways and how the interface provides a way to access behaviour. Illustrate behaviour changes by adding new functions, deleting old functions or modifying existing functions.

6. Primitive values, wrapper classes, types and casting

Primitive values and types: int, short, long, float, double, boolean, char. Corresponding wrapper classes for each primitive type. Class as type of the object. Class as mechanism for user defined types. Changing types through user defined casting and automatic type coercion for some primitive types.

Ideally, everything should be a class; primitive types are defined for efficiency reasons; each primitive type has a corresponding wrapper class. Classes as user defined types. In some cases types are changed by automatic coercion or casting – e.g. mixed type expressions. However, casting in general is not a good idea and should be avoided, if possible.

7. Variables, expressions

Variables as names for values; expressions (arithmetic and logical) and their evaluation (operators, associativity, precedence). Assignment operation; difference between left hand side and right hand side of assignment.

Variables denote values; variables are already defined as attributes in classes; variables have types that constrain the values it can denote. Difference between variables denoting primitive values and object values – variables denoting objects are references to those objects. The assignment operator = is special. The variable on the lhs of = denotes the memory location while the same variable on the rhs denotes the contents of the location e.g. i=i+2.

8. Statements, scope

Statements; conditional (if, if-then-else, switch-break, ?: ternary operator), looping (for, while-do, do-while, continue, break); grouping statements in blocks, scope and visibility of variables.

Describe the semantics of the conditional and looping statements in detail. Evaluation of the condition in conditional statements (esp. difference between || and | and && and &). Emphasize fall through in switch statement. Many small examples should be done to illustrate control structures. Printing different kinds of patterns for looping is instructive. When number of iterations are known in advance use the for loop otherwise the while-do or do-while loop. Express one loop construct using the others. For e.g.:

for (<init>; <test>; <inc>) <stmt>; is equivalent to:

Using while

<init>; while <test> {<stmt>; <inc> }

Using do-while

<init>; if !<test> do <stmt>; <inc> while <test>;

Nesting of blocks. Variables with block scope, function scope, class scope. Visibility rules when variables with the same name are defined in different scopes.

9. Functions

Functions/methods (as abstractions for complex user defined operations on objects), functions as mechanisms for side effects; formal arguments and actual arguments in functions; different behaviour of primitive and object arguments. Static functions and variables. The **this** variable. Examples of algorithmic problem solving using functions (various number theoretic problems, finding roots of algebraic equations).

*Functions are like complex operations where the object is implicitly the first argument. Variable **this** denotes the current object. Functions typically return values, they may also cause side-effects (e.g. change attribute values of objects) – typically functions that are only supposed to cause side-effects return void (e.g. Set functions). Java passes argument by value. Illustrate the difference between primitive values and object values as arguments (changes made inside functions persist after the call for object values). Static definitions as class variables and class functions visible and shared by all instances. Need for static functions and variables. Introduce the main method – needed to begin execution.*

10. Arrays, strings

(a) Structured data types – arrays (single and multi-dimensional), strings. Example algorithms that use structured data types (e.g. searching, finding maximum/minimum, sorting techniques, solving systems of linear equations, substring, concatenation, length, access to char in string, etc.).

Storing many data elements of the same type requires structured data types – like arrays. Access in arrays is constant time and does not depend on the number of elements. Sorting techniques (bubble, selection, insertion). Structured data types can be defined by classes – String. Introduce the Java library String class and the basic operations on strings (accessing individual characters, various substring operations, concatenation, replacement, index of operations). The Class StringBuffer should be introduced for those applications that involve heavy manipulation of strings.

(b) Basic concept of a virtual machine; Java virtual machine; compilation and execution of Java programs (the javac and java programs).

The JVM is a machine but built as a program and not through hardware. Therefore it is called a virtual machine. To run, JVM machine language programs require an interpreter (the java program). The advantage is that such JVM machine language programs (.class files) are portable and can run on any machine that has the java program.

(c) Compile time and run time errors; basic concept of an exception, the Exception class, catch and throw.

Differentiate between compile time and run time errors. Run time errors crash the program. Recovery is possible by the use of exceptions. Explain how an exception object is created and passed up until a matching catch is found. This behaviour is different

from the one where a value is returned by a deeply nested function call. It is enough to discuss the Exception class. Sub-classes of Exception can be discussed after inheritance has been done in Class XII.

(d) Class as a contract; separating implementation from interface; encapsulation; private and public.

Class is the basic reusable unit. Its function prototypes (i.e. the interface) work as a visible contract with the outside world since others will use these functions in their programs. This leads to encapsulation (i.e. hiding implementation information) which in turn leads to the use of private and public for realizing encapsulation.

(e) Interfaces in Java; implementing interfaces through a class; interfaces for user defined implementation of behaviour.

Motivation for interface: often when creating reusable classes, some parts of the exact implementation can only be provided by the final end user. For example, in a class that sorts records of different types the exact comparison operation can only be provided by the end user. Since only he/she knows which field(s) will be used for doing the comparison and whether sorting should be in ascending or descending order be given by the user of the class.

Emphasize the difference between the Java language construct interface and the word interface often used to describe the set of function prototypes of a class.

(e) Basic input/output using Scanner and Printer classes from JDK; files and their representation using the File class, file input/output; input/output exceptions. Tokens in an input stream, concept of whitespace, extracting tokens from an input stream (StringTokenizer class).

The Scanner class can be used for input of various types of data (e.g. int, float, char etc.) from the standard input stream or a file input stream. The File class is used model file objects in the underlying system in an OS independent manner. Similarly, the Printer class handles output. Only basic input and output using these classes should be covered.

Discuss the concept of a token (a delimited continuous stream of characters that is meaningful in the application program – e.g. words in a sentence where the delimiter is the blank character). This naturally leads to the idea of delimiters and in particular whitespace and user defined characters as delimiters. As an example show how the StringTokenizer class allows one to extract a sequence of tokens from a string with user defined delimiters.

(g) Concept of recursion, simple recursive functions (e.g. factorial, GCD, binary search, conversion of representations of numbers between different bases). Recursive sorting techniques.

Many problems can be solved very elegantly by observing that the solution can be composed of solutions to 'smaller' versions of the same problem with the base version having a known simple solution. Recursion can be initially motivated by using recursive equations to define certain functions. These definitions are fairly obvious and are easy to understand. The definitions can be directly converted to a program. Emphasize that any recursion must have a base case. Otherwise, the computation can go into an infinite loop. Illustrate this by removing the base case and running the program. Examples:

(i) Definition of factorial:

factorial(0) = 1 //base case

*factorial(n) = n * factorial(n-1)*

(ii) Definition of GCD:

gcd(m, n) =

if (m==n) then n //base case

else if (m>n) then gcd(m-n, n)

else gcd(m, n-m)

(iii) Definition of Fibonacci numbers:

fib(0) = 1 //base case

fib(1) = 1 //base case

fib(n) = fib(n-1)+ fib(n-2)

The tower of Hanoi is a very good example of how recursion gives a very simple and elegant solution where as non-recursive solutions are quite complex. Discuss the use of a stack to keep track of function calls. A stack can also be used to solve the tower of Hanoi problem non-recursively. Merge sort and Quick sort on arrays.

SECTION C

Inheritance, polymorphism, data structures, computational complexity

11. Inheritance and polymorphism

Inheritance; base and derived classes; member access in derived classes; redefinition of variables and functions in subclasses; abstract classes; class Object; protected visibility. Subclass polymorphism and dynamic binding.

Emphasize the following:

- *inheritance as a mechanism to reuse a class by extending it.*
- *inheritance should not normally be used just to reuse some functions defined in a class but only when there is a genuine specialization (or subclass) relationship between objects of the base class and that of the derived class.*
- *Allows one to implement operations at the highest relevant level of abstraction.*
- *Freezes the interface in the form of abstract classes with abstract functions that can be extended by the concrete implementing classes. For example, an abstract class Shape can have an abstract function draw that is implemented differently in the sub-classes like Circle, Quadrilateral etc.*
- *how the exact function call at run time depends on the type of the object referenced by the variable. This gives sub-class polymorphism. For example in the code fragment:*

  Shape s1=new Circle(), s2=new Quadrilateral();

  s1.draw(); //the draw is the draw in Circle

  s2.draw(); //the draw is the draw in Quadrilateral

  the two draw function invocations on s1, s2 invoke different draw functions depending on the type of objects referenced by s1 and s2 respectively.

12. Data structures

(a) Basic data structures (stack, queue, dequeue); implementation directly through classes; definition through an interface and multiple implementations by implementing the interface. Basic algorithms and programs using the above data structures.

A data structure is a data collection with well defined operations and behaviour or properties. The behaviour or properties can usually be expressed formally using equations or some kind of logical formulae. Consider for e.g. a stack with operations defined as follows:

void push(Object o)

Object pop()

boolean isEmpty()

Object top()

Then, for example the LIFO property can be expressed by (assume s is a stack):

if s.push(o); o1=pop() then o ≡ o1

What the rule says is: if o is pushed on the stack s and then it is popped and o1 is the object obtained then o, o1 are identical.

Another useful property is:

if s.isEmpty() == true then s.pop() = ERROR

It says that popping an empty stack gives ERROR.

Similarly, several other properties can also be specified. It is important to emphasize the behavioural rules or properties of a data structure since any implementation must guarantee that the rules hold.

Some simple algorithms that use the data structures:

(i) For stack: parentheses matching, tower of Hanoi, nested function calls; solving a maze.

(ii) For queue: scheduling processes, printers, jobs in a machine shop.

(b) **Recursive data structures: single linked list (Algorithm and programming), binary trees, tree traversals (Conceptual)**

Data structures should be defined as abstract data types with a well defined interface (it is instructive to define them using the Java interface construct) – see the comments in (a) above. Emphasize that algorithms for recursive data structures are themselves recursive and that algorithms are usually the simplest and most elegant. The following should be covered for each data structure:

Linked List (single): insertion, deletion, reversal, extracting an element or a sublist, checking emptiness.

Binary trees: apart from the definition the following concepts should be covered: external and internal nodes, height, level, size, degree, completeness, balancing, Traversals (pre, post and in-order).

13. Complexity and big O notation

Concrete computational complexity; concept of input size; estimating complexity in terms of functions; importance of dominant term; best, average and worst case. Big O notation for computational complexity; analysis of complexity of example algorithms using the big O notation (e.g. Various searching and sorting algorithms, algorithm for solution of linear equations etc.).

Points to be given particular emphasis:

(i) Algorithms are usually compared along two dimensions – amount of space (that is memory) used and the time taken. Of the two the time taken is usually considered the more important. The motivation to study time complexity is to compare different algorithms and use the one that is the most efficient in a particular situation.

(ii) Actual run time on a particular computer is not a good basis for comparison since it depends heavily on the speed of the computer, the total amount of RAM in the computer, the OS running on the system and the quality of the compiler used. So we need a more abstract way to compare the time complexity of algorithms.

(iii) This is done by trying to approximate the number of operations done by each algorithm as a function of the size of the input. In most programs the loops are important in deciding the complexity. For example in bubble sort there are two nested loops and in the worst case the time taken will be proportional to n(n-1) where n is the number of elements to be sorted. Similarly, in linear search in the worst case the target has to be compared with all the elements so time taken will be proportional to n where n is the number of elements in the search set.

(iv) In most algorithms the actual complexity for a particular input can vary. For example in search the number of comparisons can vary from 1 to n. This means we need to study the best, worst and average cases. Comparisons are usually made taking the worst case. Average cases are harder to estimate since it depends on how the data is distributed. For example in search, if the elements are uniformly distributed it will take on the average n/2 comparisons when the average is taken over a statistically significant number of instances.

(v) Comparisons are normally made for large values of the input size. This means that the dominant term in the function is the important term. For example if we are looking at bubble sort and see that time taken can be estimated as: $a*n^2 + b*n + c$ where n is the number of elements to be sorted and a, b, c are constants then for large n the dominant term is clearly n^2 and we can, in effect, ignore the other two terms.

All the above motivates the big O notation. Let f(n), g(n) be positive functions, then f(n) is said to be O(g(n)) if there exists constants c, n_0 such that $f(x) \leq c*g(n)$ whenever $n > n_0$. What this means is that g(n) asymptotically dominates f(n). Expressing time complexity using the big O notation gives us an abstract basis for comparison and frees us from bothering about constants. So the estimated time complexity $a*n^2+b*n+c$ is $O(n^2)$.

Analyse the big O complexity of the algorithms pertaining to the data structures in 11 (a) and (b) above.

PAPER II - PRACTICAL

This paper of three hours duration will be evaluated by the Visiting Examiner appointed locally and approved by the Council.

The paper shall consist of three programming problems from which a candidate has to attempt any one. The practical consists of the two parts:

1. Planning Session
2. Examination Session

The total time to be spent on the Planning session and the Examination session is three hours. After completing the Planning session the candidates may begin with the Examination session. A maximum of 90 minutes is permitted for the Planning session. However, if the candidates finish earlier, they are to be permitted to begin with the Examination session.

Planning Session

The candidates will be required to prepare an algorithm and a hand written Java program to solve the problem.

Examination Session

The program handed in at the end of the Planning session shall be returned to the candidates. The candidates will be required to key-in and execute the Java program on seen and unseen inputs individually on the Computer and show execution to the Visiting Examiner. A printout of the program listing including output results should be attached to the answer script containing the algorithm and handwritten program. This should be returned to the examiner. The program should be sufficiently documented so that the algorithm, representation and development process is clear from reading the program. Large differences between the planned program and the printout will result in loss of marks.

Teachers should maintain a record of all the assignments done as part of the practical work through the year and give it due credit at the time of cumulative evaluation at the end of the year. Students are expected to do a **minimum** of twenty assignments for the year.

Marks (out of a total of 100) should be distributed as given below:

Continuous Evaluation

Candidates will be required to submit a work file containing the practical work related to programming assignments done during the year.

Programming assignments done throughout the year
(Internal evaluation) - 10 marks

Programming assignments done throughout the year
(Visiting Examiner) - 10 marks

Terminal Evaluation

Solution to programming problem on the computer
- 60 marks

(Marks should be given for choice of algorithm and implementation strategy, documentation, correct output on known inputs mentioned in the question paper, correct output for unknown inputs available only to the examiner.)

Viva-voce - 20 marks

(Viva-voce includes questions on the following aspects of the problem attempted by the student: the algorithm and implementation strategy, documentation, correctness, alternative algorithms or implementations. Questions should be confined largely to the problem the student has attempted).

NOTE:

Algorithm should be expressed clearly using any standard scheme such as a pseudo code.

EQUIPMENT

There should be enough computers to provide for a teaching schedule where at least three-fourths of the time available is used for programming.

Schools should have equipment/platforms such that all the software required for practical work runs properly, i.e. it should run at acceptable speeds.

Since hardware and software evolve and change very rapidly, the schools may have to upgrade them as required. Following are the recommended specifications as of now:

The Facilities:

- A lecture cum demonstration room with a MULTIMEDIA PROJECTOR/ an LCD and O.H.P. attached to the computer.
- A white board with white board markers should be available.
- A fully equipped Computer Laboratory that allows one computer per student.
- Internet connection for accessing the World Wide Web and email facility.
- The computers should have a minimum of 512 MB (1 GB preferred) RAM and a PIV or higher processor. The basic requirement is that it should run the operating system and Java programming system (Java compiler, Java runtime environment, Java development environment) at acceptable speeds.
- Good Quality printers.

Software:

- Any suitable Operating System can be used.
- JDK 6 or later.
- Documentation for the JDK version being used.
- A suitable text editor. A development environment with a debugger is preferred (e.g. BlueJ, Eclipse, NetBeans). BlueJ is recommended for its ease of use and simplicity.

SAMPLE TABLE FOR SENDING MARKS FOR PRACTICAL WORK

COUNCIL FOR THE INDIAN SCHOOL CERTIFICATE EXAMINATIONS

COMPUTER SCIENCE – ASSESSMENT OF PRACTICAL WORK OF CANDIDATES – CLASS XII

(To be completed by the Visiting Examiner and returned to the office of the Council along with the Practical Marksheet/s)

S. NO.	INDEX NUMBER	NAME OF THE CANDIDATE	Assessment of Practical File		Assessment of the Practical Examination (To be evaluated by the Visiting Examiner only)						TOTAL MARKS (Total Marks are to be added and entered by the Visiting Examiner)
			Internal Evaluation	Visiting Examiner	Algorithm	Java Program	Documentation	Hard Copy (printout)	Output	Viva-Voce	
			10 Marks	10 Marks	10 Marks	20 Marks	10 Marks	10 Marks	10 Marks	20 Marks	100 Marks
1.											
2.											
3.											
4.											
5.											
6.											
7.											
8.											
9.											
10.											

Name of the Visiting Examiner:_____

Signature: _____

Date:_____