**-D—Object-oriented programming** ***Notes by Dave Mulkey, 2015, Germany***

**D.1 Objects as a programming concept (6 hours)**

The paradigm of object-oriented programming should be introduced through discussion and example.

| | **Assessment statement** | **lvl** | **Teacher's notes** | **Explanations** |
|---|---|---|---|---|
| D.1.1 | Outline the general nature of an object. | 2 | An object as an abstract entity and its components—data and actions.<br>Familiar examples from different domains might be people, cars, fractions, dates and music tracks. | An OBJECT contains VARIABLES = PROPERTIES<br>and METHODS = BEHAVIORS. |
| D.1.2 | Distinguish between an object (definition, template or class) and instantiation. | 2 | Students must understand the difference in terms of code definitions, memory use and the potential creation of multiple instantiated objects. | In Java, an OBJECT is an INSTANCE of a CLASS.<br>We can make lots of OBJECTS from the same CLASS.<br>Each OBJECT requires some memory to store it's details.<br>Each OBJECT has it's own copy of its VARIABLES<br>and its own copy of the class METHODS.<br>But, if the METHODS or VARIABLES are STATIC,<br>then there is only one copy of the method or variable,<br>and all the OBJECTS share the same copy . |
| D.1.3 | Construct unified modelling language (UML) diagrams to represent object designs. | 3 | **LINK** Connecting computational thinking and program design. | UML diagrams show:<br>+-------------------------+<br>\| CLASS NAME \|<br>\| ------------------------ \|<br>\| Variable names \|<br>\| -------------------------\|<br>\| Method signatures \|<br>\| -------------------------\|<br>\| Dependencies \|<br>+-------------------------+<br>Example about Vehicles |

| | | | | |
|---|---|---|---|---|
| D.1.4 | Interpret UML diagrams. | 3 | **LINK** Connecting computational thinking and program design. | **Library Diagram**<br>**Shopping**<br>**Placing Orders** |
| D.1.5 | Describe the process of decomposition into several related objects. | 2 | A simple example with 3–5 objects is suggested. Examples related to D.1.1 could be employers, traffic simulation models, calculators, calendars, media collections.<br>**LINK** Thinking abstractly.<br>**AIM 4** Applying thinking skills critically to decompose scenarios. | **OOP Design Notes** |
| D.1.6 | Describe the relationships between objects for a given problem. | 2 | The relationships that should be known are dependency (“uses”), aggregation (“has a”) and inheritance (“is a”).<br>**LINK** Thinking abstractly.<br>**AIM 4** Applying thinking skills critically to decompose scenarios. | An example using Busses<br>- dependency = a Bus USES roads (depends on)<br>- aggregation = a Bus HAS seats (contains)<br>- inheritance = a Bus IS A vehicle (extends)<br>and hence has an engine, like all vehicles<br><br>An example using Students<br>- dependency = a Student USES a school<br>- aggregation = a Student HAS hands<br>- inheritance = a Student IS A person, and hence has a name |
| D.1.7 | Outline the need to reduce dependencies between objects in a given problem. | 2 | Students should understand that dependencies increase maintenance overheads. | It is desirable for Objects (classes) to be "self-contained".<br>This is partially achieved through ENCAPSULATION -<br>that means putting all the DATA and METHODS inside the Object (class). If we DEPEND on another class to provide useful methods, then we must be sure to have the class/Object available whenever we use our Object.<br>For example, if we have an Employee Object containing name, phone, etc about the employee, the program will need to |

| | | | | |
|---|---|---|---|---|
| | | | | SAVE that data in a disk file. If we include the SAVE method inside the Employee Object, then it is simpler to use than expecting some external methods to take care of SAVING data. And if some other programmer changes the SAVE method that is somewhere else, we cannot be sure that it will work well.<br>This is related to choosing where RESPONSIBILITIES are placed in a computer system. |
| D.1.8 | Construct related objects for a given problem. | 3 | In examinations problems will require the students to construct definitions for no more than three objects and to explain their relationships to each other and to any additional classes defined by the examiners.<br>**LINK** Connecting computational thinking and program design.<br>**AIM 4** Applying thinking and algorithmic skills to resolve problems. | See example Object Hierarchies, like the BUSSES in the Specimen Paper 2. |
| D.1.9 | Explain the need for different data types to represent data items. | 3 | The data types will be restricted to integer, real, string and Boolean. | integer = whole numbers (Java int)<br>real = floating point, decimals (Java double or float)<br>string = text data, characters (Java String)<br>Boolean = true/false (Java boolean)<br>It is actually possible to store numeric values in String variables, but this is inefficient. Adding up 1 million numbers would then require 1 million conversions from String to double. |
| D.1.10 | Describe how data items can be passed to and from actions as parameters. | 2 | Parameters will be restricted to pass-by-value of one of the four types in D.1.6. Actions may return at most one data item. | look at many Java examples<br><br>Pass-By-Value means that the value is copied from the main program into the parameter(s). If the value of a parameter is |

| | | | | |
|---|---|---|---|---|
| | | | | changed inside a method, that change does NOT effect the original values in the main program.<br><br>A Pass-By-Value (or reference) parameter does NOT have it's value copied into the parameter, but rather the MEMORY LOCATION (reference/pointer) is copied. That means the method CAN change the values in the variables in the main program. |

**D.2 Features of OOP (4 hours)**

Students should be able to describe the features of OOP that distinguish it from other approaches to computer programming.

| | **Assessment statement** | **lvl** | **Teacher's notes** | |
|---|---|---|---|---|
| D.2.1 | Define the term encapsulation. | 1 | Data and actions are limited to the object in which they are defined. | This is accomplished by:<br>- putting all DATA (properties/variables) and all METHODS inside the Object / class<br>- using PRIVATE variables, accessed through public GET- and SET- methods |
| D.2.2 | Define the term inheritance. | 1 | A parent object holds common data and actions for a group of related child objects. Multiple inheritance is not required. | This involves<br>- making an original Object/class<br>- EXTENDING the class to create a new Object/class with more properties and methods<br>- during inheritance, the new class can OVER-RIDE the old properties and methods in the original class<br>See examples in Java programs |
| D.2.3 | Define the term polymorphism. | 1 | Actions have the same name but different parameter lists and processes. | There can be several methods (different versions) with the same name. If these are in the same class, then they must have different SIGNATURES (parameter lists), so that the compiler can tell which version to use. If the various versions |

| | | | | |
|---|---|---|---|---|
| | | | | are in different classes, then they could have the same parameter lists and will be selected by the compiler according to the class currently in use. |
| D.2.4 | Explain the advantages of encapsulation. | 3 | For example, the scope of data should be confined to the object in which it is defined as far as possible in order to limit side effects and dependencies. | Encapsulation ensures that<br>- variables (properties) are not accidentally changed by another part of the program<br>- the correct methods are used, rather than accidentally using another method with the same name that is outside the class<br><br>This is particularly important in a team-programming project, where various parts of a computer system are created by many different people. Although good communication and documentation could prevent many accidents, Encapsulation enforces restrictions at the compiler level. |
| D.2.5 | Explain the advantages of inheritance. | 3 | For example, a parent object holds common data and actions, which enhances reuse and reduces maintenance overheads. | This is especially useful in team-programming projects. It means that a parent (basic) object could be changed (improved) and those changes would immediately be availble in the child (extended) objects. For example, if a better sorting algorithm is created in the parent object, it can immediately be used by the child objects. The compiler makes the improvements automatically, without the programmers passing around copies of the new sorting method. |
| D.2.6 | Explain the advantages of polymorphism. | 3 | For example, an action in a child object may choose to override actions of a parent object. This allows an external program to use the same action on a family of objects without knowing the implementation detail. | Consider a school database containing these Objects:<br>Person (name, phone, etc)<br>- Student extends person (adding homeroom, bus route...)<br>- Teacher extends person (adding room, subject, etc)<br>Now an END_OF_YEAR method can be constructed that sends an eMail to each Student reminding them to turn in all their books. A different END_OF_YEAR method for a Teacher |

| | | | | |
|---|---|---|---|---|
| | | | | might send an email reminding them about important work they should do during the vacation (no work for the Students!) Although it is tricky to program in Java, it's possible to make a loop that goes through all the Person Objects, sending the appropriate EMAIL by using the END_OF_YEAR method in that Object. It's not necessary for the main program to know which EMAIL will be sent - it simply INVOKES the END_OF_YEAR method contained in each Object. |
| D.2.7 | Describe the advantages of libraries of objects. | 2 | For example, sorts and other complex algorithms and processes do not have to be “re-invented”. | A very simple example is all the GUI control Objects available from the javax.swing library. We don't want every programmer to write their own code to draw a jButton and all its associated methods (like resize). This library has been built, debugged and documented through great time and expense by the system programmers at SUN. Another example would be a 3rd-party library that draws 2D and 3D mathematical graphs. No need for every mathematics programmer to write all this code again. For example, the algorithm to draw a straight line from one point to another, choosing exactly the correct pixels, is a complex and difficult problem. |
| D.2.8 | Describe the disadvantages of OOP. | 2 | For example, increased complexity for small problems; unsuited to particular classes of problem.<br>**AIM 9** Develop an appreciation of the limitations of OOP. | If you just need a program that adds up an Arithmetic Progression, e.g. 1+3+5+...+999999 , this can be written as a simple loop that prints the answer on the console (CLI = Command Line Interface). There is no need for the complexity involved in OOP. OOP mainly creates modules (classes) that are RE-USABLE. So "one-off" programs that will be written, used and thrown away won't benefit from OOP techniques. |
| D.2.9 | Discuss the use of programming teams. | 3 | As compared to individuals working alone. Examples include speed to completion, information hiding to reduce module | "Divide and conquer" - clearly a team of 100 programmers should finish a program more quickly than one lone programmer - ESPECIALLY IF the program is a large system, |

| | | | | |
|---|---|---|---|---|
| | | | dependencies, expertise in narrow fields (eg testing, documentation), etc.<br>**INT**, **AIM 5** The need to develop a common "language" to enable collaboration across international frontiers when resolving problems. | like an email server and client. There are many thousands of lines of code needed, and a single programmer is going to spend a long time typing all those lines of code. This also requires the lone programmer to know an awful lot about email systems, and he/she may be missing some crucial knowledge. With a team of programmers, we are more likely to have a larger collection of knowledge. And if some crucial knowledge or skills are missing, we can always higher another programmer(s) to fill in the missing bits. |
| D.2.10 | Explain the advantages of modularity in program development. | 3 | Advantages include easier debugging and testing, speedier completion, etc. | OOP improves modularity by breaking the program/system into separate pieces, coded as separate Classes. This makes it possible to:<br>- test and debug each small piece, before the entire system is finished<br>- DISTRIBUTE the work to various programmers who can work in PARALLEL to complete the system more quickly<br>- RE-USE old, reliable, proven modules (libraries) in further projects<br>- make CHANGES more easily by improving individual modules (classes) and then "pluggin-in" the improvements |

**D.3 Program development (20 hours) with JETS**

| | **Assessment statement** | **lvl** | **Teacher's notes** | |
|---|---|---|---|---|
| D.3.1 | Define the terms: class, identifier, primitive, instance variable, parameter variable, local variable. | 1 | These are generally related to the object's data. See JETS. | JETS is a subset of Java, specifying the commands that IB Comp Sci students must understand. They must be able to read, trace, and write algorithms using standard Java, as specified in JETS. |

| | | | | |
|---|---|---|---|---|
| D.3.2 | Define the terms: method, accessor, mutator, constructor, signature, return value. | 1 | These are generally related to the object's actions. See JETS. | |
| D.3.3 | Define the terms: private, protected, public, extends, static. | 1 | These are generally related to the OOP features described in D.2. See JETS. | |
| D.3.4 | Describe the uses of the primitive data types and the reference class string. | 2 | In examination questions the primitive types will be limited to int, long, double, char and Boolean.<br>**MYP** Mathematics: forms of numbers. | |
| D.3.5 | Construct code to implement assessment statements D.3.1–D.3.4. | 3 | Students may be asked to trace, explain or construct algorithms using the concepts associated with the terms. | |
| D.3.6 | Construct code examples related to selection statements. | 3 | Students may be asked to trace, explain or construct algorithms using simple and compound if … else constructs. | |
| D.3.7 | Construct code examples related to repetition statements. | 3 | Students may be asked to trace, explain or construct algorithms using for, while or do … while loops. | |
| D.3.8 | Construct code | 3 | Students may be asked to trace, explain or | |

| | | | | |
|---|---|---|---|---|
| | examples related to static arrays. | | construct algorithms using static arrays. | |
| D.3.9 | Discuss the features of modern programming languages that enable internationalization. | 3 | For example, use of UNICODE character sets.<br>**INT** When organizations interact, particularly on an international basis, there may be issues of language differences. | |
| D.3.10 | Discuss the ethical and moral obligations of programmers. | 3 | For example, adequate testing of products to prevent the possibilities of commercial or other damage. Acknowledging the work of other programmers. The main aims of the Open Source movement should be known.<br>**S/E**, **AIM 8** An awareness of the ethical considerations when developing new code. | |

**HL Extension**

**D.4 Advanced program developm ent (15 hours)**

| | **Assessment statement** | **lvl** | **Teacher's notes** | |
|---|---|---|---|---|
| D.4.1 | Define the term recursion. | 1 | | A method that INVOKES (calls) itself. That means the method will contain it's own name as one of the commands. |
| D.4.2 | Describe the application of recursive algorithms. | 2 | Students should understand that recursion can be applied to a small subset of programming problems to produce elegant solutions. Students should also understand that recursive algorithms are rarely used in | A TREE is the standard example of a need for recursion.<br>At each node in the TREE, the nodes below it form a SUB-TREE. Then the algorithm starts over at the current node and recurses through the subtree. Afterward, it must RETURN to the parent node and complete the previous copy of the |

| | | | | |
|---|---|---|---|---|
| | | | practice.<br>**LINK** Thinking abstractly, thinking recursively. | algorithm.<br><br>In the real world of business programming, most tasks can be completed ITERATIVELY (using loops), and they don't require recursion. LISTS are LINEAR structures. 2D arrays are effectively LINEAR, although they don't appear that way at first. But we can TRAVERSE a 2D array using NESTED LOOPS. Most standard business problems can be treated effectively as a 2D array (as used in Excel).<br><br>IF we need to TRAVERSE an array by following NEIGHBORS (e.g. in MineSweeper to count neighbors) then a RECURSIVE method may still be required. Such problems cannot be programmed with nested loops. |
| D.4.3 | Construct algorithms that use recursion. | 3 | This is limited to a method that returns no more than one result and contains either one or two recursive calls.<br>**LINK** Connecting computational thinking and program design. | practice standard algorithms, especially Tree Traversals (in order, pre order, post order) |
| D.4.4 | Trace recursive algorithms. | 2 | All steps and calls must be shown clearly.<br>**LINK** Connecting computational thinking and program design. | The TRACE usually looks like a tree, with each new recursive call creating a new node in the tree. |
| D.4.5 | Define the term object reference. | 1 | As typified by simple classes that are self-referential. | A Linked-List or a TREE contain NODE objects that point to other NODE objects, so the pointers (next, leftChild, rightChild) will be self-referential, e.g.<br><br>class TreeNode<br>{<br>  String data; |

| | | | | |
|---|---|---|---|---|
| | | | | TreeNode leftChild;<br>TreeNode rightChild;<br>} |
| D.4.6 | Construct algorithms that use reference mechanisms. | 3 | | In exams, this is probably only going to involve single pointers, not binary trees. |
| D.4.7 | Identify the features of the abstract data type (ADT) list. | 2 | Students should understand the nature of an ADT—where no implementation details are known but the actions/methods are standard. | An ADT (Abstract Data Type) "hides" it's complexity inside the Object / class. The programmer should be able to use the ADT successfully without knowing exactly HOW it does what it does. Specific examples are : LinkedList and ArrayList (same as Vector). |
| D.4.8 | Describe applications of lists. | 2 | Students should understand that lists can be used to represent stacks and queues. | Queue - waiting lines, printer queue (server)<br>Stack - return values from methods, undo lists in applications, history in a browser<br>These can be implemented in an ArrayList, by adding and removing from the appropriate end(s) |
| D.4.9 | Construct algorithms using a static implementation of a list. | 3 | Lists will be restricted to singly linked types. Methods that should be known are add (head and tail), insert (in order), delete, list, isEmpty, isFull. | A STATIC implementation of a Linked-List means it is stored in an Array, together with appropriate "pointers" like FIRST and LAST. Then there are actually no "links", but similar features must be provided, e.g.:<br>- addAtTail<br>- addAtHead<br>- insert<br>- delete<br>- list (show all items)<br>- isEmpty (checkEmpty) and isFull (checkFull) |
| D.4.10 | Construct list algorithms using | 3 | Lists will be restricted to singly linked types. Methods that should be known are add (head | This is a DYNAMIC implementation, with a NODE class and a LinkedList ADT, containing approriate methods for |

| | | | | |
|---|---|---|---|---|
| | object references. | | and tail), insert (in order), delete, list, isEmpty, isFull. | adding, inserting and deleting NODES at specific locations. |
| D.4.11 | Construct algorithms using the standard library collections included in JETS. | 3 | The classes are ArrayList and LinkedList. Students should have a broad understanding of the operation of these lists and their interface (methods) but not of the internal structure. | e.g. use these as ADTs. Don't need to doing any programming inside of these - just use the standard features. |
| D.4.12 | Trace algorithms using the implementations described in assessment statements D.4.9–D.4.11. | 2 | In examination questions, definitions of ArrayList and LinkedList methods will be given where necessary. | Be sure to also practice use pen and paper. |
| D.4.13 | Explain the advantages of using library collections. | 3 | Students should understand that libraries provide convenient and reliable implementations of common programming tasks. | see D.2.7<br>Specifically for ArrayList and LinkedList - it is very easy to make a programming mistake when dealing with references (pointers). Libraries have been thoroughly debugged and are probably more reliable than the code that normal programmers could write. This also saves lots of time, and is sensible when implementing STANDARD algorithsm, like searching and sorting. |
| D.4.14 | Outline the features of ADT's stack, queue and binary tree. | 2 | Students should be able to provide diagrams, applications and descriptions of these ADTs. For example, they should know that a binary tree can be used to efficiently store and retrieve unique keys. | Stack = LIFO , using PUSH and POP<br><br>Queue = FIFO, using ENQUEUE and DEQUEUE<br><br>Binary Tree = using LEFTCHILD, RIGHTCHILD and traversals : in order, pre order, post order |

| | | | | |
|---|---|---|---|---|
| | | | | Store and retrieve unique "keys" - like words for a spell-check algorithm |
| D.4.15 | Explain the importance of style and naming conventions in code. | 3 | Students should understand that meaningful identifiers, proper indentation and adequate comments all improve the readability of code for humans and thus save money, time and effort in programming teams.<br>**INT**, **AIM 5** The need to develop a common “language” to enable collaboration across international frontiers when resolving problems. | Two main reasons for good style:<br>(1) enables someone else to read and understand your code more easily and successfully<br><br>(2) enables you or anyone else to make changes and improvements later<br><br>Good style includes:<br>- clear and consistent NAMING CONVENTION<br>- clear and consistent SPELLING (like camelCaps)<br>- clear and consistend INDENTATION<br>- lots of COMMENTS |