Diploma Programme

IB Java Examination Tool Subset (JETS)

Introduction

The object-oriented programming (OOP) option of the IB computer science syllabus allows students to learn to program in Java, and this is a choice that most teachers would reasonably be expected to make—although the IB does not insist that this is so. It would be unreasonable to expect students to learn all of Java—with the many libraries and classes and the constantly changing nature of the language that would be impractical. The intention is not for students to become Java "experts". The intention is to provide an unambiguous representation of essential OOP concepts for the examination.

Teachers should note that sample algorithms can be found in the teacher support material for this course.

Only the commands, symbols and constructs specified in JETS will be used in examination questions relating to the OOP option. Students will not be required to read or write answers involving other libraries. Students may have learned some other languages, constructs and classes, and may choose to use these in their examination answers. However, students are not permitted to use library classes and methods from any language, including Java, that would perform automated tasks, such as sorting and searching arrays and other collections.

JETS also specifies a **naming convention** and **style** for examination questions. Teachers should familiarize their students with JETS, including the naming and style conventions. These conventions are intended to make examination questions clear and easily readable. Students are not required to follow these conventions in their answers. However, they should write answers in a clear, consistent and readable style, and must not use non-standard libraries that trivialize the solution.

Students will **not** be expected to write **perfect** syntax in their answers (for example, a mistake in capitalization or a missing semi-colon would not normally be penalized) but errors that substantially change the **meaning** of the algorithm will be penalized (for example, "forgetting" an exclamation point is a real error).

Students and examiners alike are expected to use the clearest, most readable style in their answers. Students should be especially careful to avoid careless writing and syntax that is difficult to read, such as double minus signs (--) or compound assignment operators like (-=). For example:

```
x = x + 1 is clearer than x++ or x += 1

x = x - 1 is clearer than x-- or x -= 1
```

The presentation of JETS

Style conventions

The style conventions for JETS to be used in all examination papers will be as follows.

`JETS-code will be printed in Courier (fixed spacing) font 10.5 point.`

All reserved words will be written in **`lower case bold`**.

Class names will always start with an upper case letter.

Object, variable and method names will always start with a lower case letter.

`multiWordIdentifiers` will use embedded capitals to separate the words (not underscores).

Identifiers will generally use whole words, not abbreviations or acronyms.

Proper indentation will always be used.

The order of modules is irrelevant, but the **main** and**/**or **constructor** method will always be placed at the top.

Some examination questions may include statements such as, "recall that …". The intention is to remind students of any uncommon commands and functions:

Recall that `String.indexOf(String)` can be used to find the position of one string inside another, like this:

```
   String email = "exams@ibo.org";
int atSign = email.indexOf("@"); //result is 5
```

For French and Spanish versions of examination papers:

reserved words will remain in English

string constants will be translated

user-defined identifiers (class, variable and method names) will be translated as appropriate.

The syntax of JETS

Operators

Arithmetic: `+ , - , * , / , %` (students must understand the polymorphic behaviour of the division operator, for example int / int ==> int)

Relational: `== , > , < , >= , <= , !=`

Boolean: `! , && , ||` (bitwise Boolean operators `& , |` are **not** required)

Operator precedence

The standards for operator precedence in Java are assumed knowledge. Examination questions may use extra parentheses for clarity and candidates should be encouraged to do the same in their solutions.

Notation for literals (values)

string : `"in quotation marks"`

char : `'s' // in single quotes`

integer : `123456 or -312`

double : `124.75 (fixed point) or 1.2475E+02 (floating point)`

boolean : `true , false`

Constant identifiers will be written in `ALL_CAPS`, using an underscore to separate words. They will be defined using **`final static`**, as:

```
final static double NATURAL_LOG_BASE = 2.1782818;
```

Primitive data types

byte int long double char boolean

(**short** and **float** are not included)

Structured data types

String class

StringBuffer class

Linear Arrays : `int[ ] numbers = new int[100];`

`(array of 100 integers, index 0..99)`

2-D arrays: `int[ ][ ] checkers = new int[8][8];`

Text files `(sequential files)`

RandomAccessFile `(fields as primitive types)`

`LinkedList class` (including the use of the Collection interface)

Note: The numeric wrapper classes Integer, Double, and so on, will only be used to provide the functionality of static methods for doing type conversions, as demonstrated in the IBIO methods (below).

Parameter passing

Parameter passing follows the standard specification in Java, for example, primitive types are automatically passed by value, and structured types (arrays and objects) have their references passed by value (usually equivalent to pass by reference of other languages).

Symbols

```
/* multi-line

comments */

// single line

// comments
```

`( )` round brackets for parameters

`[ ]` square brackets for subscripts in arrays

`.` dot notation for dereferencing object methods and data members

`{ }` for blocks of code

`{ 1 , 2 , 3 }` for initializing an array

`< class >` to permit the use of structured classes with the LinkedList class

The assumed set of IBIO commands is listed below.

IBIO input methods

All input methods display a prompt String, accept keyboard input until the user presses the [enter] key, and then return a value of the specified type. It is assumed that the input routines cannot cause a runtime error. If the user types a String that cannot be converted to the correct type, the input routine returns a default value, for example, a blank String, a 0 numeric value, and so on.

```
IBIO c = new IBIO();  // Instantiate new console class
String input(String prompt)
String input() // does not print a prompt before inputting
String inputString(String prompt)
char inputChar(String prompt)
boolean inputBoolean(String prompt)
byte inputByte(String prompt)
int inputInt(String prompt)
```

```
long inputLong(String prompt)
double inputDouble(String prompt)
```

IBIO output methods

```
output( String ) --> outputs a String
output( char ) --> outputs a char value
output( boolean ) --> outputs a boolean value
output( byte ) --> outputs a byte value
output( int ) --> outputs an int value
output( long ) --> outputs a long value
output( double ) --> outputs a double value
```

JETS also uses the System console output commands:

```
System.out.print( string )
System.out.println( string )
```

`System.in.read()` is **not** included in JETS, although it is used inside IBIO.

Loops and decisions

```
if (boolean condition)
{ ... commands ... }
else if (boolean condition)
{ ... commands ... }
else
{ ... commands ... } ;
```

`switch..break..` is not included in JETS, but students may use it in their answers.

```
for ( start; limit; increment)
{ ... commands ... } ;
while (boolean condition)
{... commands ... } ;
do
{ ... commands ... }
while (boolean condition) ;
```

Files

Standard level/higher level

`BufferedReader(FileReader)`—will be used to open a sequential file for input

```
.ready
.read
.readLine
.close
```

`PrintWriter(FileWriter)`—will be used to open a sequential file for output

```
.print
.println
.close
```

```
// Serialization is not required.
```

Higher level only

`RandomAccessFile`

constructor: `randomAccessFile(String filename, String accessMode)`

```
.seek
.length
.read .... readInt, readDouble, readBytes, readUTF
.write .... writeInt, writeDouble, writeBytes, writeUTF
.close
```

Standard methods

`Math` class

```
.abs,.pow,.sin,.cos,.round,.floor
```

`String` class

+ for concatenation

```
.equals(String)
.substring(startPos, endPos)
.length()
.indexOf(String)
.compareTo(String)
.toUpperCase()
.toLowerCase()
```

`LinkedList` class

```
LinkedList<E> where E defines the type of elements held in the
list
```

Constructor `LinkedList<E>()`

```
.add(E e)
.add(int index, E element)
.addFirst(E e)
.addLast(E e)
.clear()
.element()
.get(int index)
.getFirst()
.getLast()
.remove()
.remove(int index)
.removeFirst()
.removeLast()
.size()
.isEmpty()
```

Arrays

```
.length
```

Cast

```
(int) (double) (byte) (char)

(numeric + "") // to convert a numeric value to a String
```

Static methods and variables

Where a class variable or method is declared **`static`** it does not change across instantiated objects. Students will most likely see static methods in the wrapper classes that provide utilities for each of the primitive equivalents. The wrapper classes themselves are not included in JETS although students should be aware that, for example:

```
      Integer.parseInt(String)
```

converts a valid String representation of an integer to an int primitive.

The IBIO input/output methods are also static.

Understanding the **new** construct is required. Students must be aware that **new** causes an object to be instantiated, and that this is somewhat different from declaring a primitive data type. They should thoroughly understand the rules for **scope** and **lifetime** of identifier references, and that instances may be **automatically destroyed** and **garbage collected** when they go out of scope. For example, students must understand that a value stored in a method's local variables will be lost when the method returns, and that this value cannot be retrieved by subsequent calls to the method. **Static** is a required concept, but will not be directly tested in code (it may appear, but the meaning in the code will not be directly examined).

Dynamic memory allocation

Students must also understand that an object type can be declared without instantiating, and that this reference (pointer) can be reassigned later to either a new instance or to a different existing instance.

Other syntactical issues

Java permits commands to span multiple lines. This may be done in exam questions, but only when it improves clarity and readability, for example, in a long parameter list:

```
public int sortArray( String[ ] names ,
                      int listSize ,
                      char ascendingOrDescending
                     )
```

Brackets will always be lined up, either horizontally or vertically:

```
public void printNumbers()
{  int x = 0;
    while ( x < 10 )
    { output( x ); }   //  brackets lined up horizontally
}           //  brackets for method body lined up vertically
```

Class scope

```
public, private
// implements and abstract are not included
// interface is not included
```

Overall structure of classes

Students must understand the concept of a **constructor** and a **main method**, and the difference between them. They must also understand the concept of **extends**.

Error handling

```
try{ ... commands ... }
  catch(Exception e){ ... handle the error ... };
```

Error handling in examinations will be limited to simply outputting an error message, setting a flag or returning from the method. Complex handling of specific Exception types will not be expected. Only the generic Exception and IOException errors must be trapped.

```
  methodName() throws IOException
```
//
Students must understand the idea of throwing an exception, rather than trapping it with try..catch..

Example algorithms

Algorithms to exemplify JETS, the IBIO console class and additional Java-related material may be added to CSopedia.