# Appendix 1: Assessment information for candidates

This assessment applies to the following Unit, Outcomes and Assessment Standards:

### Software Design and Development (Advanced Higher)

### Outcome 1

**1 Explain how well-structured, complex modular programs work, drawing on understanding of programming constructs, algorithms and data integration by:**

1.1 Describing the purpose of a range of structured data types

1.2 Describing how a range of complex standard algorithms work

1.3 Describing the purpose of a range of programming constructs and how they work

### Outcome 2

**2 Develop well-structured, complex modular programs by:**

2.1 Selecting and using combinations of programming constructs and standard algorithms

2.2 Selecting and using appropriate structured data types

2.3 Interfacing programs with stored data

### Outcome 3

**3 Investigate some contemporary programming paradigms by:**

3.1 Investigating simple object-oriented programs

To pass this assessment you will have to show that you have met these Outcomes and Assessment Standards.

Your assessor will let you know how the assessment will be carried out and any required conditions for doing it.

## Software Design and Development (Advanced Higher)

To pass these assessments, you will have to complete all the parts of each task, handing in all the hard-copy evidence of work you have completed. Your evidence should be appropriately labelled with your name, date(s) of completion.

**Task 1**

**Assessment Standard 1.1**

**a)** Describe how the data is structured in a 2-D array

|  |
| --- |
| Data is stored in an array of arrays |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
| **b)** Describe when it would be appropriate to use a 2-D array.  When making a graphic appear on specific co-ordinates |
|  |
|  |
|  |
|  |
|  |
|  |

**Task 2 — Grand Prix race program**

**Assessment Standards 1.2, 2.1, 2.2 and 2.3**

Your task is to implement and test a program to enable a user to:

* extract and store data to a file
* sort the data
* assign points depending on the race time of the driver
* display the data

At this level, you must show that you can use, subprograms, parameter passing, structured data types, interface programs with stored data and implement a sort algorithm.

The data is stored in a file called “Racing” which your assessor will give you.

|  |  |  |  |
| --- | --- | --- | --- |
| **Driver** | **Team** | **Race Time** | **Points** |
| Sebastian Williams | Red Drink | 0 | 0 |
| Tom Hamilton | Mercidas | 0 | 0 |
| Danny Ricardo | Red Drink | 0 | 0 |
| Walter Borras | Lewis | 0 | 0 |
| Fernando Sonal | Farrori | 0 | 0 |
| Jenson Smith | McMillan | 0 | 0 |

The program should ask the race time for each driver, with the driver names retrieved from the “Racing” file, eg

Race time for Sebastian Williams is…

The race times that require inputting are:

|  |  |
| --- | --- |
| Sebastian Williams | 141.567 |
| Tom Hamilton | 140.342 |
| Danny Ricardo | 141.721 |
| Walter Borras | 140.982 |
| Fernando Sonal | 141.442 |
| Jenson Smith | 141.210 |

When all race times have been entered, the program should sort the drivers by the shortest race times and then assign the following points to the appropriate drivers:

* 25 points for 1st place
* 18 points for 2nd place
* 15 points for 3rd place

The updated file should be stored as “RacingResults”

Internal commentary should be used to explain how the sort subprogram works.

Test your program to ensure it works.

Give your assessor a printout of your program and evidence of the final output.

**Task 3a) (Java)**

**Assessment Standard 3.1**

The following code defines the **Balloon** class:

|  |  |
| --- | --- |
| 1 | public class **Balloon** { |
| 2 | private double radius; |
| 3 | private String colour; |
| 4 | public Balloon(double r, String c) { |
| 5 | radius = r; |
| 6 | colour = c; |
| 7 | } |
| 8 | public double getRadius() { |
| 9 | return radius; |
| 10 | } |
| 11 | public String getColour () { |
| 12 | return colour; |
| 13 | } |
| 14 | } |

The following code defines another class called **TestBalloon** which uses the **Balloon** class:

|  |  |
| --- | --- |
| 20 | public class **TestBalloon** { |
| 21 | public static void main(String[] args) { |
| 22 | Balloon b1 = new Balloon(4.0, "blue"); |
| 23 | System.out.println("Radius is " + b1.getRadius() + " Colour is " + b1.getColour()); |
| 24 | Balloon b2 = new Balloon(1.0, “purple”); |
| 25 | System.out.println("Radius is " + b2.getRadius( ) + " Colour is " + b2.getColour( )); |
| 26 | } |
| 27 | } |

|  |  |  |
| --- | --- | --- |
| **Step 1a) —** | | Identify which lines of code are used to create the **two** instances of the class **Balloon** . |
|  | Balloon b1 = new Balloon(4.0, "blue"); | |
|  | Balloon b2 = new Balloon(1.0, “purple”); | |
|  |  | |
| **Step 1b) —** | | Describe how **one** of these instances is constructed. |
|  | The line “Balloon b1 = new Balloon(4.0, "blue");” will create a new | |
|  | Instance of the “Balloon” class, with the class variables of radius at 4.0 and colour at “blue”, | |
|  | With the name “Balloon b1”. | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |
|  |  | |

**Step 2 —** Modify the **TestBalloon** code to create a new instance of the **Balloon** class that is yellow and has a radius of 3.

**Step 3** **—** Create and compile the **Balloon** class.

Create, compile and run the modified **TestBalloon** class and make sure that it works correctly.

**Step 4** **—** Produce hard-copy evidence of modified **TestBalloon** class and the output from your test run. Hand in this hard-copy evidence, labelled with your name and date of completion, to your assessor.

**Task 3a) (alternative in SQA reference language)**

**Assessment Standard 3.1**

The following code defines the **Balloon** class:

|  |  |
| --- | --- |
| 1 | CLASS Balloon IS { REAL radius, STRING colour} |
| 2 |  |
| 3 | METHODS |
| 4 |  |
| 5 | CONSTRUCTOR Balloon(REAL r, STRING c) |
| 6 | DECLARE THIS.radius INITIALLY r |
| 7 | DECLARE THIS.colour INITIALLY c |
| 8 | END CONSTRUCTOR |
| 9 |  |
| 10 | FUNCTION getRadius () RETURNS REAL |
| 11 | RETURN THIS.radius |
| 12 | END FUNCTION |
| 13 |  |
| 14 | FUNCTION getColour () RETURNS STRING |
| 15 | RETURN THIS.colour |
| 16 | END FUNCTION |
| 17 |  |
| 18 | END CLASS |
| 19 |  |
| 20 |  |
| 21 | PROCEDURE testballoon() |
| 24 | DECLARE b1 INITIALLY Balloon( 4.0, "blue") |
| 25 | DECLARE b2 INITIALLY Balloon( 1.0, "purple") |
| 26 | showBalloonDetails(b1) |
| 27 | showBalloonDetails(b2) |
| 28 | END PROCEDURE |
| 29 |  |
| 30 |  |
| 31 | PROCEDURE showBalloonDetails (Balloon b) |
| 32 | SEND "Radius is " TO DISPLAY |
| 33 | SEND b.getRadius TO DISPLAY |
| 34 | SEND "Colour is " & b.getColour() TO DISPLAY |
| 35 | END PROCEDURE |
| 36 |  |
| 37 |  |
| 38 | testBalloon() |

|  |  |  |  |
| --- | --- | --- | --- |
| **Step 1a) —** | | | Identify which lines of code are used to create the **two** instances of the class **Balloon** . |
|  | Line 24: DECLARE b1 INITIALLY Balloon( 4.0, "blue") | | |
|  | Line 25: DECLARE b2 INITIALLY Balloon( 1.0, "purple") | | |
|  |  | | |
| **Step 1b) —** | | Describe how **one** of these instances is constructed. | |
|  | Line 24 will create an instance of the “Balloon” class, which will have the name | | |
|  | “b1”, and the values of the class variables “radius” and “colour” will have the | | |
|  | Values within the brackets. | | |
|  |  | | |
|  |  | | |
|  |  | | |
|  |  | | |
|  |  | | |

**Step 2 —** Modify the program to create a new instance of the **Balloon** class that is yellow and has a radius of 3.

**Step 3 —** Create the modified program in an object-oriented language of your choice. Run the modified program and test the program to make sure it works consistently.

**Step 4 —** Produce hard-copy evidence of modified program and the output from your test run. Hand in this hard-copy evidence, labelled with your name and date of completion, to your assessor.

**Task 3b) (Java)**

**Assessment Standard 1.3 and 3.1**

The following code defines the **Countdown** class:

1. class **Countdown**
2. { Countdown(){}
3. void runCountdown(int startingValue)
4. {
5. if(startingValue!=0{
6. System.out.println(startingValue);
7. runCountdown(startingValue-1);
8. }
9. else
10. { System.out.println("BLAST OFF!");
11. }
12. }
13. }

The following code defines the **TestCountdown** class:

1. class **TestCountdown**
2. { public static void main(String [] argv)
3. { Countdown launch = new Countdown();
4. launch.runCountdown(10);
5. }
6. }

This code produces the following output:

|  |
| --- |
| 10  9  8  7  6  5  4  3  2  1  BLAST OFF! |

**Step 1a) —** Identify where recursion occurs in the **Countdown** class.

|  |
| --- |
|  |
|  |
| **Step 1b) —** Explain how it achieves the desired result. |
|  |
| **Step 1c) —** What is the purpose of recursion? |
|  |

**Step 2 —** Modify the above program to produce the following output:

The dog rounded up 1 sheep

The dog rounded up 2 sheep

The dog rounded up 3 sheep

The dog rounded up 4 sheep

The dog rounded up 5 sheep

And then went home for tea!

**Step 3** **—** Create and compile the **Countdown** class. Create, compile and run the modified **Countdown** class and make sure that it works correctly.

**Step 4** **—** Produce hard-copy evidence of modified **Countdown** class and the output from your test run. Hand in this hard-copy evidence, labelled with your name and date of completion, to your assessor.

**Task 3b) (alternative in SQA reference language)**

**Assessment Standard 1.3 and 3.1**

The following program defines the **Countdown** class:

|  |  |
| --- | --- |
| 1 | CLASS **Countdown** IS { INTEGER startingValue } |
| 2 |  |
| 3 | METHODS |
| 4 | PROCEDURE keepCounting( INTEGER currentCount) |
| 5 | IF currentCount ≠ 0 THEN |
| 6 | SEND currentCount TO DISPLAY |
| 7 | keepCounting( currentCount – 1 ) |
| 8 | ELSE |
| 9 | SEND "BLAST OFF!" TO DISPLAY |
| 10 | END IF |
| 11 | END PROCEDURE |
| 12 |  |
| 13 | PROCEDURE start() |
| 14 | keepCounting(startingValue) |
| 15 | END PROCEDURE |
| 16 |  |
| 17 | END CLASS |
| 18 |  |
| 19 |  |
| 20 | DECLARE fullCountdown INITIALLY Countdown(10) |
| 21 |  |
| 22 |  |
| 23 | fullCountdown.start() |

This program produces the following output:

|  |
| --- |
| 10  9  8  7  6  5  4  3  2  1  BLAST OFF! |

**Step 1a) —** Identify where recursion occurs in the **Countdown** class.

|  |
| --- |
| Line 7: keepCounting( currentCount – 1 ) |
|  |
| **Step 1b) —** Explain how it achieves the desired result  Line 14 will run the procedure “keepCounting” with the value of “currentCount” as one unit lower than |
| before. |
| **Step 1c) —** What is the purpose of recursion?  To minimise repetitiveness of code when using or reusing similar lines |
|  |

**Step 2 —** Modify the above program to produce the following output:

The dog rounded up 1 sheep

The dog rounded up 2 sheep

The dog rounded up 3 sheep

The dog rounded up 4 sheep

The dog rounded up 5 sheep

And then went home for tea!

**Step 3** **—** Create the modified program in an object-oriented language of your choice. Run the modified program and test the program to make sure that it works correctly.

**Step 4** **—** Produce hard-copy evidence of modified program and the output from your test run. Hand in this hard-copy evidence, labelled with your name and date of completion, to your assessor.