Question 1

a) An abstract class called **CarRental** is defined in **Figure 1**. It stores information about the number of days that somebody would like to rent the car for, as well as the price per day. Answer the questions below.

```
public abstract class CarRental {
  protected int days;        // number of days booked
  protected double price;    // price per day
  public int getDays() { return days; }
  public double getPrice() { return price; }
  public void setDays(int n) { days = n; }
  public void setPrice(double p) { price = p; }
  /* Calculate and return the cost */
  public abstract double calCost();
}
```

Figure 1

[10 marks]

i) Using the abstract class **CarRental**, write a concrete sub-class called **SimpleCarRental**. Add a new attribute **tax** , as well as the accessor (getter) and mutator (setter) for the **tax** variable. This represents the tax rate applied when renting a car; for example, a 20% tax rate should be stored as **0.2**. The **calCost()** method in **SimpleCarRental** class simply calculates the cost using the formula **days*price*(1+tax)**.

(6 marks)

ii) Write a test program with a **main()** method for class **SimpleCarRental**, that calls the **calCost()** method and displays the cost for variable values **days=3**, **price=79.5** and **tax=0.2**.

(4 marks)

	Do not write in this column
i)	**6/10**
`public class SimpleCarRental extends CarRental {` [2 marks]	
`  private double tax;` [1 mark]	
`  public double getTax() {` [1 mark]	
`  return tax;`	
`  }`	
`  public void setTax(double t) {` [1 mark]	
`  tax = t;`	
`  }`	

	Do not write in this column
`  public double calCost() {` [1 mark]	
`  return days*price*(1+tax);`	
`  }`	
`}`	
ii)	**4/10**
`public class CarRentalCostTest {`	
`  public static void main(String[] args) {`	
`  SimpleCarRental myRental = new SimpleCarRental();` [1 mark]	
`  myRental.setDays(3);` [2 marks: THREE correct values]	
`  myRental.setPrice(79.5);`	
`  myRental.setTax(0.2);`	
`  System.out.print(myRental.calCost());` [1 mark]	
`  }`	
`  }`	
	10 marks

b) Write a Java program called **StringReverser** that takes a **String** from the command line argument and prints out the **String** in reversed order without any white spaces. Note that the method **charAt(int i)** can be used to retrieve the character at a particular location in a **String** object.

For example, executing the following command on the command line:

java StringReverser "EBU4201 Java Programming"

would result in the output **gnimmargorPavaJ1024UBE**.

[7 marks]

	Do not write in this column
`public class StringReverser {`	
`  public static void main (String[] args) {`	
`  String s = args[0];` [1 mark]	
`  for (int i=s.length()-1; i>=0; i--) {` [3 marks: correct loop and **`length()`** method]	
`  char c = s.charAt(i);`	
`  if (c != ' ')` [3 marks: removing spaces and reversing characters]	
`  System.out.print(c);`	
`  }`	
`  }`	
`}`	
Note: Other solutions for removing spaces and reverse printing are also accepted.	
	7 marks

c) Answer the following questions related to *method overriding*:

[8 marks]

i) What is *method overriding*?

(2 marks)

ii) When would you use the keyword **`super`** in relation to *method overriding*?

(2 marks)

iii) Explain why you might wish to override the following methods defined in the class **`java.lang.Object`**:

- `public boolean equals(Object obj)`
- `public String toString()`

(4 marks)

	Do not write in this column
i)	**2/8**
Method overriding is the practice of re-defining the implementation of a method that	
was defined in the superclass [1 mark]. The method signature must be exactly the	
same as that of the superclass [1 mark].	

	Do not write in this column
ii)	**2/8**
Any calls to an overridden method from within the subclass will by default invoke	
the version implemented in the subclass [1 mark]. The method name can be preceded	
with **`'super.'`**, in order to explicitly invoke the superclass version of the method.	
[1 mark]	
iii)	**4/8**
`equals()` – The **`Object`** class implementation compares the identities of the	
calling object and of the object passed in as the argument, i.e. it would only return	
`true` if both references are referring to the exact same object [1 mark]. However, in	
practice it is more useful to be able to return **`true`** when two objects have equal state	
– this is why we normally override the method **`equals()`** [1 mark].	
`toString()` – the **`Object`** class implementation returns a **`String`** consisting of the	
name of the class of which the object is an instance, the **`'@'`** character and the	
hashcode [1 mark]. However, in practice is more useful to return a string containing	
the state of the object, i.e. the values of the instance variables – this is why we	
normally override the method **`toString()`** [1 mark].	
	8 marks

Question marking: $\frac{\quad}{10} + \frac{\quad}{7} + \frac{\quad}{8} = \frac{\quad}{25}$

Question 2

a) Answer the following questions:

[7 marks]

i) Java has TWO types of memory space. Which memory space is used to store objects?

(1 mark)

ii) When is an object marked for *garbage collection*?

(1 mark)

iii) What is a **`null`** reference?

(1 mark)

iv) Assume a **`Dog`** class has a **`sleep()`** method. What is wrong with the code fragment in line **`2`** of **Figure 2**? How can the problem be fixed?

(2 marks)

v) By the end of the code in **Figure 2**, the **`Dog`** called "Bob" is not eligible for garbage collection. Explain why that is the case. How could this object be made eligible for garbage collection?

(2 marks)

```
1.  Dog myDog;
2.  myDog.sleep();
3.  myDog = new Dog("Fido");
4.  myDog = new Dog("Bob");
```

Figure 2

	Do not write in this column
i)	**1/7**
It is the Heap. [1 mark]	
ii)	**1/7**
When there is no reference to it. [1 mark]	
iii)	**1/7**
A **`null`** reference is a reference variable that does not refer to any object. [1 mark]	
iv)	**2/7**
`myDog` cannot invoke the **`sleep()`** method [0.5 mark] because it has no reference to	
any **`Dog`** object. [0.5 mark]	
To fix the problem, we need an extra statement between lines **`1`** and **`2`**, in **Figure 2**:	
`Dog myDog = new Dog();` [1 mark]	

	Do not write in this column
Note: Other sensible instantiations are also accepted.	
v)	**2/7**
Because the **`Dog`** object called "Bob" still has a reference to it. [1 mark]	
To make "Bob" eligible for garbage collection, we could do [1 mark]:	
`myDog = null;` **OR**	
describe the same thing, i.e. make **`myDog null`**	
	7 marks

b) This question is about *interfaces*:

[13 marks]

i) Describe in detail what *interfaces* are.

(5 marks)

ii) State TWO similarities and TWO differences between *interfaces* and abstract classes.

(4 marks)

iii) Explain why multiple inheritance is not supported for abstract classes, and how *interfaces* can be used to provide a kind of multiple inheritance.

(4 marks)

	Do not write in this column
i)	**5/13**
An interface is like a certificate that says "I provide these services" [1 mark].	
You cannot make an instance of an interface [1 mark] and its entire functionality is	
left unspecified [1 mark].	
A class implements an interface by using the keyword **`implements`** [1 mark], and this	
means that it must then provide an implementation for each of the methods listed in the	
interface [1 mark].	

	Do not write in this column
ii)	**4/13**
Similarities	
Both *abstract* classes and *interfaces*, [2 marks: any TWO of the list below]	
** cannot be instantiated;	
** can contain **`abstract`** methods;	
** are supertypes to classes that extend/implement them.	
Differences [2 marks: any TWO of the list below]	
** *Abstract* classes can have concrete methods, while *interfaces* can only have	
`abstract` methods.	
** *Interfaces* support a kind of multiple inheritance, while *abstract* classes do not.	
** *Interfaces* are 'implemented', while *abstract* classes are 'extended'.	
iii)	**4/13**
Since *abstract* classes can have concrete methods, multiple inheritance of *abstract*	
classes could result in ambiguity about which superclass method to call. [2 marks]	
Interfaces provide a kind of "multiple inheritance" by allowing to extend several	
interfaces, so e.g. you could have **`interface A extends B,C`** (where both **`B`** and	
`C` are *interfaces* themselves) [1 mark]. Since no *interface* methods are concrete, this	
does not introduce ambiguity about which code to execute [1 mark].	
	13 marks

c) Consider the Java code fragment in **Figure 3** (lines numbered **`1–4`**) and determine the value of the variable **`s1`**. Write the loop to print out the value of the **`s1`** variable **`10`** times.

```
1   String s1 = new String("I love");
2   String s2 = "Java";
3   String s3 = "";
4   s1 = s1 + " " + s2 + "!" + s3;
```

Figure 3

[3 marks]

	Do not write in this column
The value of **`s1`** is [1 mark]: `I love Java!`	
Loop:	
`for (int i=0; i<10; i++) { System.out.println(s1); }` [2 marks]	
	3 marks

d) Consider the code for the TWO Java classes in **Figure 4** and determine the output of the program.

```
public class Person {
  public String id = "c ";

  public Person() {
    System.out.print("Person ");
  }
}
-----------------------------------------------------------------
public class Student extends Person {
  public Student() { System.out.print("Student "); }

  public static void main(String[] args) { new Student().go(); }

  void go() {
    id = "x ";
    System.out.print(this.id + super.id);
  }
}
```

Figure 4

[2 marks]

	Do not write in this column
The output is [2 marks]: `Person Student x x`	
	2 marks

Question marking: $\frac{}{7} + \frac{}{13} + \frac{}{3} + \frac{}{2} = \frac{}{25}$

Question 3

a) Consider the Java class shown in **Figure 5**.

```
public class Account {
  private static int noAccounts;
  private int balance;

  void withdraw(int amount) {
    int charge;
    if (balance < amount) charge = 10;
    else charge = 0;
    balance = balance – amount – charge;
  }
}
```

Figure 5

Identify all the variables in **Figure 5**, and provide the following information for each of them:

- the name of the variable;
- the variable type, i.e. whether it is a *class* variable, *instance* variable, etc;
- the scope and default value.

[10 marks]

Do not write in this column

Variable	Type	Scope	Default value
noAccounts	class	class	0
balance	instance	class	0
charge	local	method only	no value
amount	parameter	method only	specified by method caller

[4*0.5 mark: each correctly identified variable]

[4*1 mark: each correctly identified type]

[4*1 mark: each variable's correct scope and default value]

10 marks

b) Write a Java method named **`printTable()`** that accepts one **`int`** parameter representing the number of rows and one **`char`** parameter representing the separator. This method should print a Multiplication-Table according to the number of rows (which also defines the number of entries in each row), and the separator.

For example: Calling **`printTable(3, ' ')`** should produce the output in **Figure 6** (a), whereas calling **`printTable(4, ',')`** should produce the output in **Figure 6** (b).

Important: You must use nested **`for`** loops.

```
1 2 3
2 4 6
3 6 9
```

(a)

```
1,2,3,4,
2,4,6,8,
3,6,9,12,
4,8,12,16,
```

(b)

Figure 6

[6 marks]

Do not write in this column

```
void printTable(int n, char c) {      [1.5 marks]
  for (int i = 1; i <= n; i++) {      [1 mark]
    for (int j = 1; j <= n; j++) {    [1 mark]
      System.out.print(i*j + c);      [1.5 marks]
    }
    System.out.println();             [1 mark]
  }
}
```

Note: Other sensible answers are also accepted.

6 marks

c) Answer the questions below:

[6 marks]

i) In Java, how does an *object* relate to a *class*?

(1 mark)

ii) What is the difference between the **`private`**, **`public`** and **`protected`** access modifiers?

(3 marks)

iii) Briefly describe the concepts of *pass-by-reference* and *pass-by-value*.

(2 marks)

	Do not write in this column
i)	**1/6**
An object is an instance of a class [1 mark].	
ii)	**3/6**
`private` access – it means that something is visible to (or within) the class only.	
[1 mark]	
`public` access – it means that something is visible to the world. [1 mark]	
`protected` access – it means that something is visible to (or within) the package and	
all subclasses. [1 mark]	
iii)	**2/6**
With pass by reference, the address of the variable is passed to a method. [1 mark]	
With pass by value, the actual value of the variable is passed to the method. [1 mark]	
	6 marks

d) Consider the code fragment in **Figure 7** and determine what will be printed when **`grade = 'c'`**. Justify your answer.

```
char grade;
switch (grade) {
  case 'a':
    System.out.println("Excellent");
  case 'b':
    System.out.println("Good!");
  case 'c':
    System.out.println("Average!");
  case 'd':
    System.out.println("Bad!");
  default:
    System.out.println("No such grade!");
}
```

Figure 7

[3 marks]

	Do not write in this column
Output [1 mark]:	
`Average!`	
`Bad!`	
`No such grade!`	
This is because none of the **`switch`** clauses has a **`break;`** statement. [1 mark]	
In this case, as soon as a **`switch`** clause is found to be **`true`**, the next clause will also	
be executed regardless of the variable value, and so on. [1 mark]	
	3 marks

Question marking: $\frac{}{10} + \frac{}{6} + \frac{}{6} + \frac{}{3} = \frac{}{25}$

Question 4

a) Consider the Java class in **Figure 8** and answer the questions below.

```
import java.awt.*;
import javax.swing.*;

public class SimpleApp extends JFrame {
  private JLabel label;
  private JTextField txtField;
  private JButton button;
  private int advance = 0;

  public SimpleApp() {
    setLayout(new FlowLayout());
    label = new JLabel("Counter");
    add(label);
    txtField = new JTextField(advance + "", 10);
    txtField.setEditable(false); // The text field is read-only.
    add(txtField);
    button = new JButton("Advance");
    add(button);
    setSize(250, 100);
    setTitle("SimpleApp");
    setVisible(true);
 }

 public static void main(String[] args) {
   new SimpleApp();
 }
}
```

Figure 8

[14 marks]

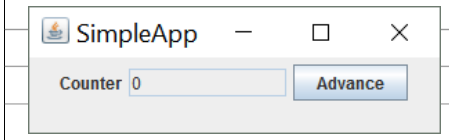
i) The code in **Figure 8** compiles. Draw a sketch of what is displayed when the code is run.

(5 marks)

ii) When you run the code in **Figure 8**, nothing will happen if you click the **`Advance`** button because there is no associated *event handling*.

- Describe the first TWO steps required to implement an *event handler* for the button.
- Write the code for the button's *event handling* method such that, every time the button **`Advance`** is clicked, it increments the value in the text field by **`2`** units.

(9 marks)

	Do not write in this column
i)	**5/14**
	
[0.5 mark: title of frame; 1 mark: GUI frame; 1 mark: labelled **`JButton`**;	
1 mark: **`JTextField`** with value **`0`** displayed; 1 mark: labelled **`JLabel`**;	
0.5 mark: general layout]	
ii)	**9/14**
First TWO steps to implement an event handler for the button:	
1. Implement the **`ActionListener`** interface [1 mark]: In the declaration for the	
event handler class, code specifies that class either implements a listener interface	
OR extends a class that implements a listener interface [1 mark].	
2. Register with the widget (i.e. the button in this case) [1 mark]: Code indicates that	
you want to listen for events, by registering an instance of the event handler class	
as a listener upon one or more components [1 mark].	
Event handling method:	
`public void actionPerformed(ActionEvent evt) {` [1.5 marks]	
`advance = advance + 2;` [1.5 marks]	
`txtField.setText(advance + "");` [2 marks]	
`}`	
	14 marks

b) You are asked to write a Java program that will display the first **`n`** lines of a text file called **`input.txt`** (where **`n`** is an integer value greater than zero). The program is called **`NLinesReader`** and both the text file name and the number of lines to read should be given as command line arguments to the program. It is not necessary to write code to check whether the program is always given the correct number and type of arguments.

Note: In order to access the text file contents, the program must use a **`FileReader`** and **`BufferedReader`** together, to create the required text file input stream. It must also handle exceptions.

[9 marks]

	Do not write in this column
`import java.io.*;` [0.5 mark]	
`public class NLinesReader {` [0.5 mark]	
`  public static void main(String[] args) {`	
`  try {` [2 marks: **try-catch** with error message]	
`  BufferedReader in = new BufferedReader(new`	
`  FileReader(args[0]));` [2 marks]	
`  int numLines = Integer.parseInt(args[1]);` [1 mark]	
`  for (int lineCt = 0; lineCt < numLines; lineCt++) {`	
`  String line = in.readLine();`	
`  System.out.println(line);`	
`  }` [2 marks: **for** loop with text output]	
`  }`	
`  catch (Exception e) { System.out.println("Error: " + e); }`	
`  }`	
`}` [1 mark: sensible program structure]	
Note: Other sensible implementations are also accepted.	
	9 marks

c) Describe the concept of *exception* in Java.

[2 marks]

	Do not write in this column
An exception is a Java object [0.5 mark] and indicates that something out of the	
ordinary can happen when a program executes [1 mark]; exceptions provide both a	
notification of errors and a way to handle them [0.5 mark].	
	2 marks

Question marking: $\frac{}{14} + \frac{}{9} + \frac{}{2} = \frac{}{25}$