Object-Oriented Programming – Sample Exam

Problem 1 – Document System

A **document system** holds a list of **documents**. Documents can be **binary** or **text** and have **name** (mandatory) and can have **content**. Documents are of two **types**: **text** and **binary**. Text documents can have **charset** (e.g. utf-8 or windows-1251). Binary documents can have **size** (in bytes). Binary documents can be of type **PDF**, **Word**, **Excel**, **Audio** or **Video**. PDF documents can hold the **number of pages** they consist of. Word documents can hold the **number of characters** they consist of. Excel documents can hold the **number of rows** and the **number of columns** in the table they hold. Word and Excel documents are both **office documents**. Office document can have **version** (e.g. "2007" or "Office97"). A special kind of binary documents are the **multimedia documents**. All multimedia documents can have **length** (in seconds). **Audio** documents and **video** documents are both multimedia documents. Audio documents can have **sample rate** (in Hz). Video documents can have **frame rate** (in fps). PDF, Word and Excel documents are **encryptable** (can be encrypted and decrypted). Word and text documents are **editable** (their content could be changed). All document characteristics except their name are **non-mandatory**.

Design the Class Hierarchy

Your **first task** is to **design an object-oriented class hierarchy** to model the document system and the documents it can hold using the best practices for object-oriented design (OOD) and object-oriented programming (OOP). Additionally you are given few C# **interfaces** that you should obligatory use:

```
using System.Collections.Generic;

public interface IDocument
{
    string Name { get; }
    string Content { get; }
    void LoadProperty(string key, string value);
    void SaveAllProperties(IList<KeyValuePair<string, object>> output);
    string ToString();
}

public interface IEditable
{
    void ChangeContent(string newContent);
}

public interface IEncryptable
{
    bool IsEncrypted { get; }
    void Encrypt();
    void Decrypt();
}
```

All your documents should implement **`IDocument`**. It specifies that documents have immutable **`Name`** and **`Content`**, can load their properties from key-value pairs through the **`LoadProperty(key, value)`** method and save their properties in a list of key-value pairs through the **`SaveAllProperties(…)`** method as well as be printed on the console through the **`ToString()`** method.

All editable documents should implement the **`IEditable`** interface. All changes of the document content should pass through this interface (direct content changes are not allowed).

All encryptable documents should implement the **`IEncryptable`** interface. You do not need to implement encryption algorithm (like AES and Blowfish), just to keep whether a document is encrypted or not in its internal state. You are allowed to encrypt / decrypt a document only though this interface.

Write a Program to Execute Commands

Your **second task** is to write a program that executes a sequence of up to 1000 commands. Each command is given in the following format:

```
Command[key1=value1;key2=value2;…]
```

Commands consist of command **name** and **attributes** given in square brackets **[ ]**. Each attribute is given in the form **key=value**. **Keys** consist of small English letters. **Values** consist single line English text and cannot contain the following characters: **[**, **]**, **=**, **;** and **\n**. The sequence of commands ends with an empty line.

The valid commands that your program should be able to execute are the following:

- **AddTextDocument[name=…;charset=…;content=…]** – adds a text document to the document system. The **name** is obligatory, but all other attributes are optional. The order of the attributes can be arbitrary. All attributes will be valid for the type of the document we create. As a result the command prints on the console "**Document added: <name>**" is case of success or "**Document has no name**" in case of missing document name. Multiple documents having the same name are allowed to be added.
- **AddPDFDocument[name=…;pages=…;size=…;content=…]** – works like **AddTextDocument**.
- **AddWordDocument[chars=…;name=…;version=…;size=…;content=…]** – works like **AddTextDocument**.
- **AddExcelDocument[rows=…;cols=…;version=…;size=…;name=…;content=…]** – works like **AddTextDocument**.
- **AddAudioDocument[name=…;content=…;samplerate=…;length=…;size=…]** – works like **AddTextDocument**.
- **AddVideoDocument[name=…;content=…;framerate=…;length=…;size=…]** – works like **AddTextDocument**.
- **EncryptDocument[name]** – changes the state of all documents matching the specified name to "encrypted". Documents that are already encrypted remain in "encrypted" state. For each document matching the specified name, the command prints as a result "**Document encrypted: <name>**" on the console or prints "**Document does not support encryption: <name>**" if the document is not encryptable. In case of no document is matching the specified name, the message "**Document not found: <name>**".
- **DecryptDocument[name]** – works similarly like **EncryptDocument**, but changes the state of all matched documents to "not encrypted" and prints as result one of the following messages: "**Document decrypted: <name>**", "**Document does not support decryption: <name>**" or "**Document not found: <name>**".
- **EncryptAllDocuments** – changes the state of all documents that support encryption to "encrypted". As result, if at least one document supports encryption, prints on the console "**All documents encrypted**", otherwise prints "**No encryptable documents found**".
- **ListDocuments[]** – prints on the console all the documents in the document system in their order of their addition. Each document should be printed alone on a single line in the following form:

  ```
  XXXDocument[key1=value1;key2=value2;…]
  ```

The **XXXDocument** is the type of the document, e.g. **PDFDocument**, **VideoDocument**, etc. The keys should be **ordered** alphabetically. Keys with no value should be skipped. In there are no documents, the command prints "**No documents found**". Encrypted documents are shown differently, as follows:

```
XXXDocument[encrypted]
```

- **ChangeContent[name;new_content]** – changes the content of all editable documents matching the specified name with the specified new content. For each document matching the specified name, the command prints as a result "**Document content changed: <name>**" on the console or prints "**Document is not editable: <name>**" if the document is not editable. In case of no document is matching the specified name, the message "**Document not found: <name>**".

The commands are guaranteed to be **valid**. Only the described above commands will be given as input. The command format will be as described above. The command parameters will also be in the described format. All attributes will be valid for their corresponding command. The commands will be no more than **1000**. Each command will be less than **500 characters** long. To simplify your work you are given a command parser that provides a skeleton for your solution (see the file **DocumentSystem-Skeleton.rar**).

Sample input:

```
AddTextDocument[name=example.txt;charset=utf-8;content=Telerik Academy Exam]
AddTextDocument[name=readme.txt]
AddTextDocument[]
EncryptAllDocuments[]
AddPDFDocument[content=6A7E889CF3A8D2;name=academy.pdf;pages=2;size=38217]
AddWordDocument[name=exam.docx;chars=12218;version=2012;size=36881]
AddWordDocument[name=exam.docx]
AddExcelDocument[name=academy.xls;rows=12;cols=3;size=9430;version=97]
AddAudioDocument[size=9834733;name=ring.mp3;samplerate=44100;length=368800]
AddVideoDocument[name=demo.mpg;framerate=24;length=87312;size=87245212]
EncryptDocument[academy.pdf]
EncryptDocument[ring.mp3]
EncryptDocument[exam.docx]
EncryptDocument[invalid.doc]
ChangeContent[example.txt;new content]
ChangeContent[demo.mpg;new content]
ChangeContent[invalid.doc;new content]
EncryptAllDocuments[]
DecryptDocument[academy.pdf]
DecryptDocument[exam.docx]
DecryptDocument[ring.mp3]
DecryptDocument[invalid.doc]
ListDocuments[]
(empty line)
```

Sample output:

```
Document added: example.txt
Document added: readme.txt
Document has no name
No encryptable documents found
Document added: academy.pdf
Document added: exam.docx
Document added: exam.docx
Document added: academy.xls
Document added: ring.mp3
Document added: demo.mpg
Document encrypted: academy.pdf
Document does not support encryption: ring.mp3
Document encrypted: exam.docx
Document encrypted: exam.docx
Document not found: invalid.doc
Document content changed: example.txt
Document is not editable: demo.mpg
Document not found: invalid.doc
All documents encrypted
Document decrypted: academy.pdf
Document decrypted: exam.docx
Document decrypted: exam.docx
Document does not support decryption: ring.mp3
Document not found: invalid.doc
TextDocument[charset=utf-8;content=new content;name=example.txt]
TextDocument[name=readme.txt]
PDFDocument[content=6A7E889CF3A8D2;name=academy.pdf;pages=2;size=38217]
WordDocument[chars=12218;name=exam.docx;size=3688
```

```
1;version=2012]
WordDocument[name=exam.docx]
ExcelDocument[encrypted]
AudioDocument[length=368800;name=ring.mp3;samplerate=44100;size=9834733]
VideoDocument[framerate=24;length=87312;name=demo.mpg;size=87245212]
```

Problem 2 – Geometry API

You are given an API containing some basic **3D geometry operations** – working with vectors, representing figures and their measures and so on. You are also given a **`FigureController`**, which executes operations coming from the standard input, over figures.

Important Classes and Interfaces

The API contains these Interfaces:

- **`ILengthMeasurable`** – provides a method **`GetLength()`** for getting the length of a figure. Should be implemended by figure classes for which length is measurable (e.g. line segments).
- **`IAreaMeasurable`** – provides a method **`GetArea()`** for getting the area of a figure, similar to **`ILengthMeasurable`**.
- **`IVolumeMeasurable`** – provides a method **`GetVolume()`** for getting the volume of a figure, similar to **`IAreaMeasurable`**.
- **`IFlat`** – provides a method **`GetNormal()`**, returning a normal vector. Should be implemented by flat (plane) figures (e.g. triangle, rectangle, etc.). A normal vector is a vector perpendicular to the surface of the figure.
- **`ITransformable`** – provides method for doing linear transformations (rotate, scale, translate) on figures. All figures should implement **`ITransformable`**.

For working with vectors, the API has these classes:

- **`Vector`** – provides an abstract class for n-dimensional vectors, with a property for **`Magnitude`** (length) and a field for the components of the vector. Also has **`Normalization`** (making the vector with **`Magnitude`** = 1) and an indexer for access to the separate components.
- **`Vector3D`** – 3-dimensional vector (3 components) with sum and subtraction operators defined, as well as multiplication by number. The class also provides some static members for vector operations – cross product, dot product and angle between two vectors, as well as parsing a vector from a string and a **`ToString()`** returning the format in which a vector should be printed on the console.

For working with figures, the API has these classes:

- **`Figure`** – abstract class that provides the base functionality for all figures and implements **`ITransformable`**. It uses a list of **`Vector3D`** elements, defining the positions of vertices of the figure (or important locations in the figure). The Figure class should be inherited by any figure, which needs to be controlled by the **`FigureController`**.
- **`FigureController`** – class for reading commands from the standard input and executing them on figures. The objects of the **`FigureController`** class keep a **`currentFigure`** field, on which they execute the incomming commands.

The code you will get will have the API and an instance of the **`FigureController`** class in the **`Main()`** method. It can handle reading input and writing output in the expected format (will be explained later), but is not complete and you will have to complete it.

Study the classes to get a better understanding of the API.

Commands

Commands are strings, which are executed by the **`FigureController`**. Commands consist of several "words" which describe the command. The "words" are these types:

- Command name – a string identifying the command
  - Examples:
    - center
    - rotate
    - triangle
- Command vector argument – a string representing a 3D vector, which can be parsed by Vector3D.Parse()
  - Example: (10,-5,0.3)
- Command scalar (number) argument – a string representing a number
  - Example: -5.3

All "words" in a command are separated by whitespaces and no word can contain whitespaces in it.

There are 3 types of commands:

- Figure creation command – creates a new figure and sets it as the current figure for the FigureController.
  - Format: [figureName] [figureVector1] [figureVector2] [figureVector…]
  - Examples:
    - triangle (0,0,0) (1,1,1) (2,0,0)
    - segment (1,13,2.3) (-4,10,0)
- Figure instance command – does operations on the current figure in the FigureController
  - Format: [operationName] [arguments…]
  - Examples:
    - rotate (0,0,0) 45 – rotates the current figure by 45 degrees about (0,0,0)
    - center – prints the vector, describing the center of the current figure
- End command – signals the end of the operations on the current figure
  - Format: end

Here's a list of all commands which are currently implemented in the API:

- translate (*vectorX*,*vectorY*,*vectorZ*) – translates the current figure with the given *vector*
- rotate (*centerX*,*centerY*,*centerZ*) degrees – rotates the current figure about the *center* by the given number of degrees. Rotation is only in the XY plane.
- scale (centerX,centerY,centerZ) factor – scales all vertices of the current figure away from the *center* by the given factor

- center – prints on the console the center of the figure as a vector in the format (x,y,z)
- measure – prints the "primary measure" of the current figure on the console as a number (e.g. 43). The primary measure for a line segment is its length, for a plane figure – its area, and for a 3D figure (e.g. cube) – its volume.
- vertex (x,y,z) – creates a "single vertex" figure and sets it as the current figure at coordinates the given x,y,z
- segment (ax,ay,az) (bx,by,bz) – creates a "line segment" figure and sets it as the current figure, with a first vertex at the given ax,ay,az and a second vertex at the bx,by,bz
- triangle (ax,ay,az) (bx,by,bz) (cx,cy,cz) – creates a triangle and sets it as the current figure, with vertices at the given coordinates

Tasks

You are advised to use everything you can from the existing API to complete the tasks that follow. You are not allowed to edit the **`Main()`** method or any provided by the API classes.

- Implement parsing of a command to create a **circle**:
  - The circle should be able to calculate its **area**
  - The circle should be able to **return a vector perpendicular to its surface**.
  - Circles will always lay in the XY plane
  - The radius of a circle doesn't scale
  - The **area** of the circle is its "**primary measure**"
  - Format: **circle (centerX,centerY,centerZ) radius**
  - Example: circle (0,1,-2) 5 – creates a circle with radius 5 and center the coordinates (0,1,-2)
- Implement parsing of a command to create a **cylinder**:
  - The cylinder should be able to calculate its **area** (the area of its walls)
  - The cylinder should be able to calculate its **volume**
  - The **volume** of a cylinder is its **primary measure**
  - The radius of a cylinder doesn't scale
  - Format: **cylinder (bottomX,bottomY,bottomZ) (topX, topY, topZ) radius** – creates a cylinder by making a base circle with a center (bottomX, bottomY, bottomZ) and the given radius, a top circle with a center (topX, topY, topZ) and the given radius and connecting them
  - Example: cylinder (5,-5,3) (5,-5,6) 7 – creates a cylinder with a base circle at (5,-5,3) and top circle at (5,-5,6), both with a radius of 7
- All **currently defined commands must work** for the new and the old figures.
- Implement an "**area**" command, which, if the figure can calculate its area, prints on the console the area of the figure as a number, and if the figure can not calculate its area, prints "undefined".
- Implement a "**volume**" command, which, if the figure can calculate its volume, prints on the console the volume of the figure as a number, and if the figure can not calculate its volume, prints "undefined".
- Implement a "**normal**" command, which, if the figure can calculate a its normal vector, prints on the console the normal vector, using the ToString of the Vector3D class, and if the figure can not calculate its normal vector, prints "undefined". The **normal** vector is a vector **perpendicular to the surface** of the figure and has a **Magnitude of 1**.

Input

The input for the commands is read from the console.

On the first line the number of figures **N** that are going to be created is specified. N will be **between 1 and 20**.

On the next line there is a command for creating a figure, and after that on each line there is a single command related to the figure, until a line with the command "end" is reached. This is repeated for each of the N figures.

The input is currently handled by the Main method through an instance of the `FigureController` class. You should think only about parsing the new commands and using the **`FigureController`**'s methods for the old commands.

Note: **all numbers and coordinates can be floating-point numbers**. Using a 64-bit floating point is advised. All **numbers in the output must be rounded to the second digit after the decimal point**.

Output

Not all commands require output – for example translate, rotate, scale and creation of figures do not print anything on the console. Other commands require output in the form of either a **number** or a **vertex**. Commands that require printing a **number** must just print a number on a **new line**. Commands that print a **vertex** must print the vertex **through** its **`ToString()`** method on a **new line**.

Example

Input	Output
3 triangle (0,-1,0) (0,1,0) (1,0,0) area translate (1,1,1) scale (1,1,1) 2 area end triangle (0,0,0) (1,-1,0) (2,0,0) volume measure end circle (0,0.5,0) 4.5 area normal end	1.00 4.00 undefined 1.00 63.62 (0.00,0.00,1.00)