Q1. In C# there are several ways to make code run in multiple threads. To make things easier, the await keyword was introduced; what does this do?

'Await' was introduced with 'Async'. These two keywords are a kind of syntax 'sugar' for writing asynchronous code. Before their introduction, we wrote async. Code in a quite difficult style like:

```
var awaiter = <something>.GetAwaiter(); // call GetAwaiter() and get TaskAwaiter
awaiter.OnCompleted( () => // on task completed
{
 var res = awaiter.GetResult(); // get result from awaiter
 Process(res); // now we can use the result
}
);
```

With async/await we can simplify it:

```
var res = await <something>; // we're waiting for task completion and getting result
Process(res); // use result
```

The Common pattern for usage: method should have keyword async as part of its signature and at least one await keyword inside.

In two words, with async/await we can write async. code in sync. way.

Q2. If you make http requests to a remote API directly from a UI component, the UI will freeze for a while, how can you use await to avoid this and how does this work?

The answer to this question is a continuation of the previous answer. For instance, we have a method:

```
void OnClick(...) // called when we click on button
{
   var res = DoSomeHeavyJob();
   button.Content = res; // updating UI
}
```

To escape the freezes, we have to change our method to:

```
async void OnClick(...)
{
    var res = await DoSomeHeavyJobAsync();
    button.Content = res;
}
```

At first, we changed (if it was not) our DoSomeHeavyJob to the async method.

Finally, we converted our method OnClick to the async/await method and await completion of DoSomeHeavyJobAsync.

Our button's content will be changed when we get res. During that time our interface will not be frozen:

DoSomeHeavyJobAsync runs on a separate thread and the rest of OnClick on UI thread

Q3. Imagine that you have to process a large unsorted CSV file with two columns:

productId (int) and availableIn (ISO2 String, e.g. "US", "NL"). The goal is to group

the file sorted by productId together with a list where the product is available.

Example: 1, "DE" 2, "NL" 1, "US" 3, "US" Becomes: 1 -> ["DE", "US"] 2 -> ["NL"]

3 -> ["US"]

a. How would you do this using LINQ syntax (write a short example)?

b. The program crashes with an OutOfMemoryError after processing approx.

80%. What would you do to succeed?

A. Assume we have class Product with properties "int ID" and "List<string> Countries".

We read the file and put the data into collection List<Product> products.

Now we have to query collection:

```
products.OrderBy(p => p.ID).Select(x =>
        new
        {
           ID = x.ID,
           CountryList = string.Join(",", x.Countries.OrderBy(c => c))
        }
        );
```

We sorted collection by ID and put to output new entity (the country list was sorted too and joined into a string).

B. In general, we have to look at code and find the place where we probably created too many objects or did it in an endless cycle. If LINQ code cause OutOfMemory - probably it's a good idea to avoid LINQ and use suitable data structures to keep the collection sorted while we read it for example SortedDictionary plus SortedSet.

About 80%... well two approaches

- we don't save states and in case of fatal exception, we lost all our job.

- we save the current state (at least the number of read/processed lines) and update it for every processed line. In case of an exception, we can start with the last unprocessed record.

Q4. In C# there is an interface IDisposable.

a. Give an example of where and why to implement this interface.

b. We can use disposable objects in a using block. What is the purpose of

doing this?

A. In .NET we can handle as well managed objects as unmanaged resources. Managed resources are handled by .NET itself. Unmanaged resources like Windows objects or files should be handled properly by the developer. The process of releasing such objects is called disposal and we use interface IDisposable to mark the type as disposable. We have to implement this interface always when we use some unmanaged object/resource. To implement it we have to implement void Dispose() method in our class We can manually dispose of the disposable objects when we feel to do it. For instance, we open file, do some work and need to close it properly and release handles.

B. using is another syntax 'sugar'. In fact, it is translated to try/finally construction.

```
using (var stream = new StreamReader("file.txt"))
{
        // doing something
}
```

The code above is equal to the following code:

```
var stream = new StreamReader("file.txt");
try
{
        // doing something
}
finally
{
        If (stream != null)
                stream.Dispose(); // calling IDisposable.Dispose()
}
```

So, using is a way to call Dispose() method for IDisposable objects after we finish with them.

Q5. When a user logs in on our API, a JWT token is issued and our Outlook plugin uses this token for every request for authentication. Here's an example of such a token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkplcmVteSIsImFkbWluIjpmYWxzZX0.BgcLOiwBvyuisQk9yWW0q0ZScMyIHNmDctw12-meCHU

Why and when is it (or isn't it) safe to use this?

The token is granted by the server to the user and allowing him to login/manage some actions.

The server should validate the session and revoke tokens in case the session has been ended.

Another problem – tokens can be stolen and used not by the granted user. In this case, we should keep some additional information (maybe client's certificates) to be sure that our user's identity is not faked.

One more solution is to limit tokens using expiration date. We also should use secured data transfer to avoid token stealing.