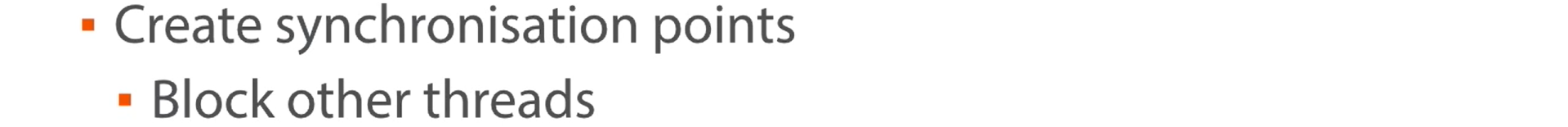
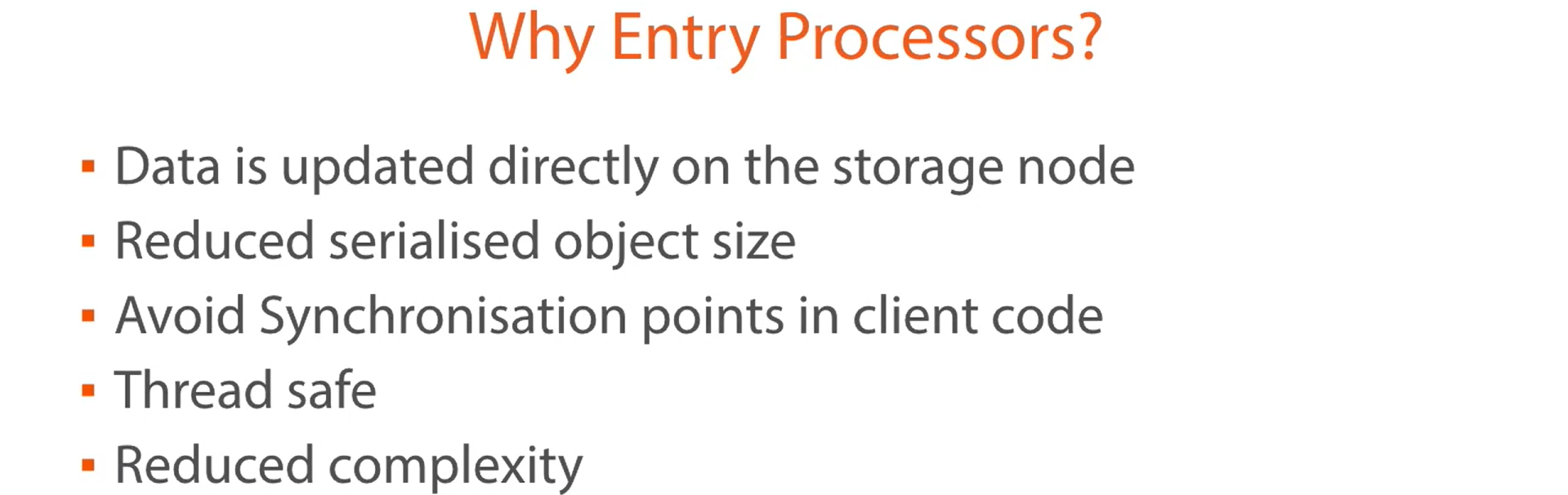
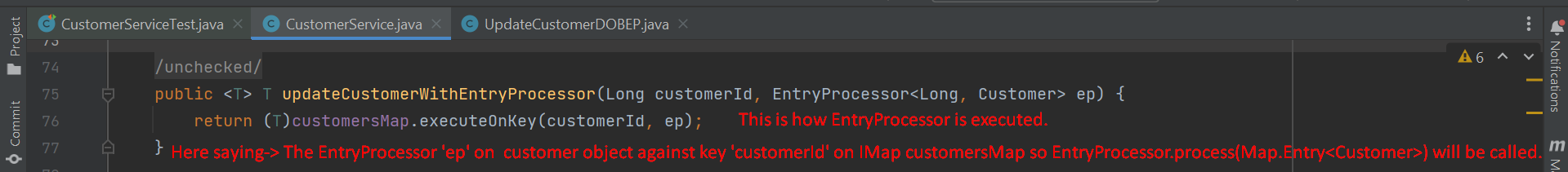
1. 2nd Approach to concurrency is to use the **Entry Processors**.
2. Why are these better than Key Locks?  
   Graphical user interface, text

   Description automatically generated  
   1. First off, by using **Key Locks**, we are creating **synchronization points** in our app and thus causing other threads to block.  
      This is generally **not good for high-performance distributed Processing**.
   2. 
      1. **Second issue with key lock** 🡺 The entire Customer object needs to be serialized over the network twice.  
         First, when we retrieve it and second when we push the updated values into the IMap.
   3. Here all we’re just trying to update is a small amount of data for example Customer’s DOB.  
      This is a lot of overhead & is not very efficient.
3. **Solution**: Better solution is to use **Entry Processors**.  
   
   1. An **Entry Processor** is an **object** that is sent to the Data & does some processing on that data in situ (in the original place) on whatever storage node that data sits on.
   2. Although this object (Entry Processor) is also serialized & sent across the network.  
      Because we’re only updating a small amount of data, the entry processors we create are generally small objects when serialized which reduces the amount of bandwidth required.
   3. We can also reduce the synchronization points in our code. Hazelcast will take care of these things for us as Entry Processors are thread safe.  
      So, we don’t have to concern ourselves with the problems associated with this.
   4. All of this will reduce the complexity as we don’t have to take care of getting and releasing locks.
4. Graphical user interface, text, application, chat or text message

   Description automatically generated
   1. Hazelcast effectively maintains .  
      There can be only one entry processor being run against the data at a particular time.  
      This means we don’t have to use locks in our client side as we don’t have to concern our code with concurrency & it results in a much more efficient way to update the data.
   2. Hazelcast monitors the queue & takes items from the queue as they become available.
   3. The Entry Point is executed, and any return object is sent back to the client.
   4. When a client invokes an Entry Processor against a key, it is added to the end of the queue & awaits its turn to be executed.  
      It is therefore important that Entry Processors are not long-running processors.  
      They should quick and efficient.
   5. **Brief**: Following both interfaces are used to process data on the storage node itself. So, Processor will be sent to the storage node having the data.
      1. **EntryProcessor** (I): To process the original data.
      2. **EntryBackupProcessor** (I): To process the copy of the original data.
5. **Let’s look at a working example of it**.
   1. **Let’s create an EntryProcessor we need**.
   2. In shared module, create a new class implementing **EntryProcessor.java** and **EntryBackupProcessor.java**
   3. **EntryProcessor**:
      1. If you’re just reading data using **EntryProcessor**, then **EntryProcessor** will not attain lock.
   4. **EntryBackupProcessor**:
      1. It is required only when you’re updating Map entry using **EntryProcessor**.  
         In our case, we’re updating DOB.
      2. If you’re reading data using **EntryProcessor**, then **EntryBackupProcessor** implementation is also no needed.
   5. EntryProcessor can be used to read data without acquiring lock but not update it.  
      As while updating, we need to ensure that the backup of that data is also updated. So, need to implement **EntryBackupProcessor**.
   6. Following defining **EntryProcessor & EntryBackupProcessor**.  
      Chart

      Description automatically generated  
        
      Following defining call to EntryProcessor.  
        
        
      Following Defining EntryProcessor.  
      Text

      Description automatically generated  
        
      **Running Test Case**:  
      