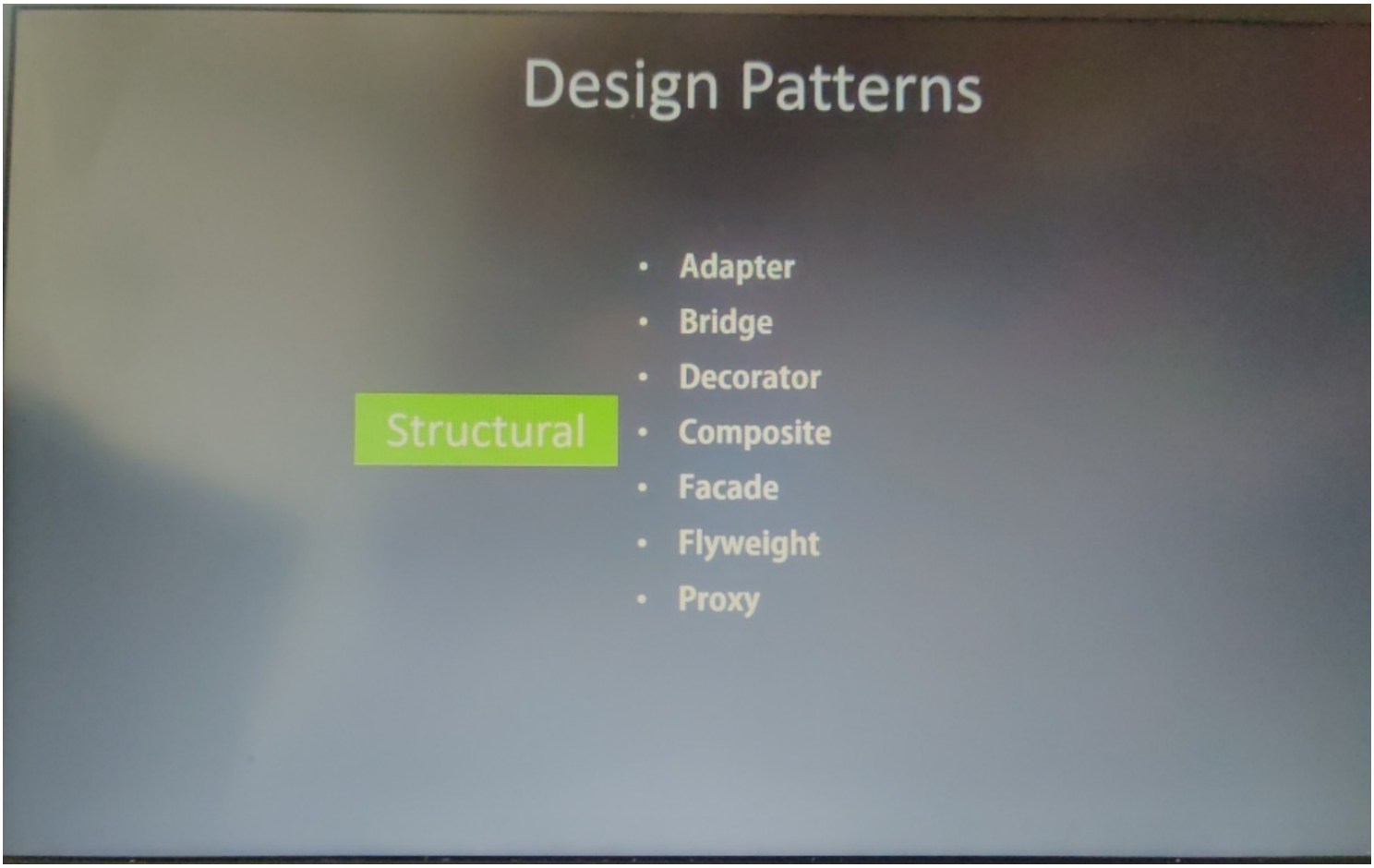
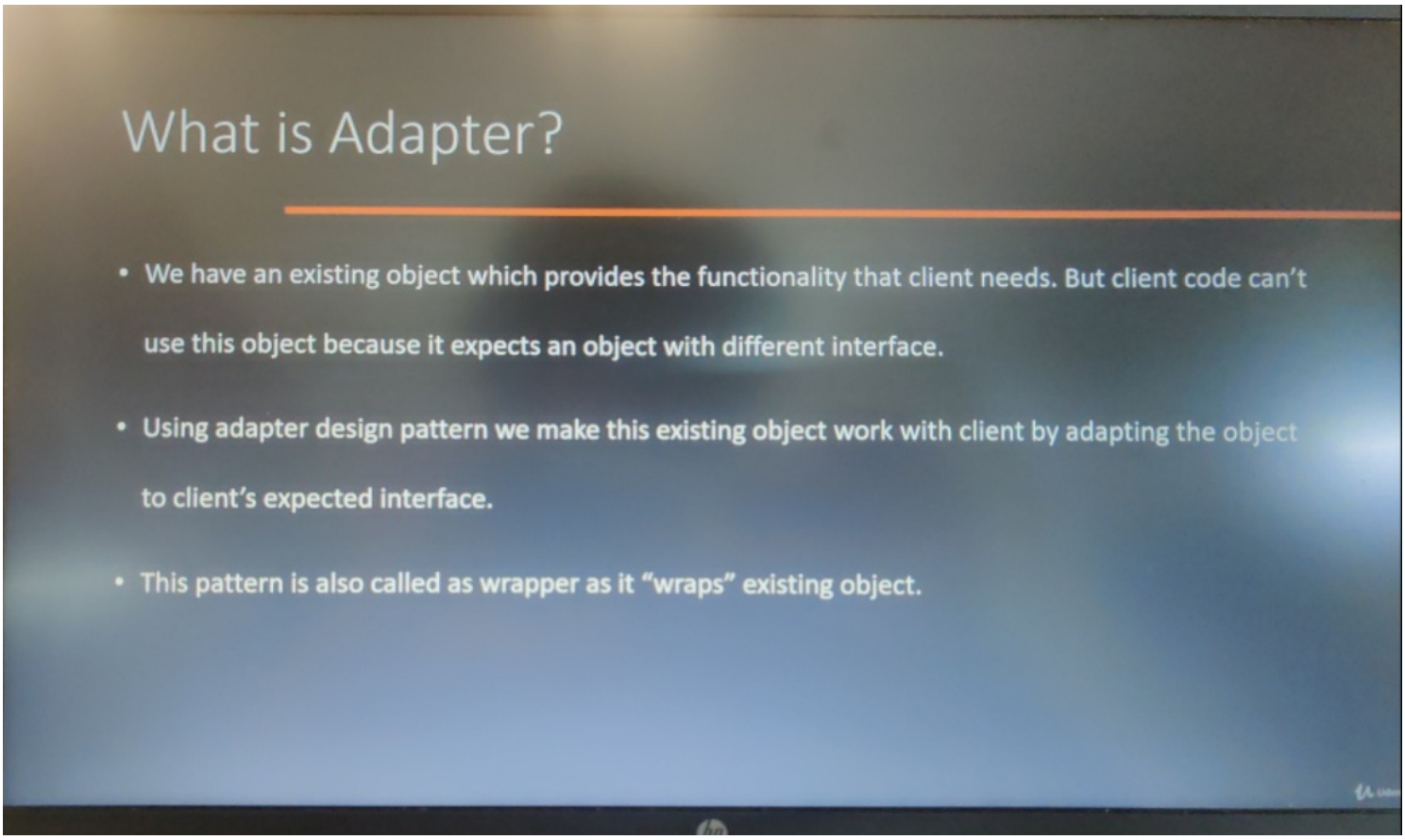
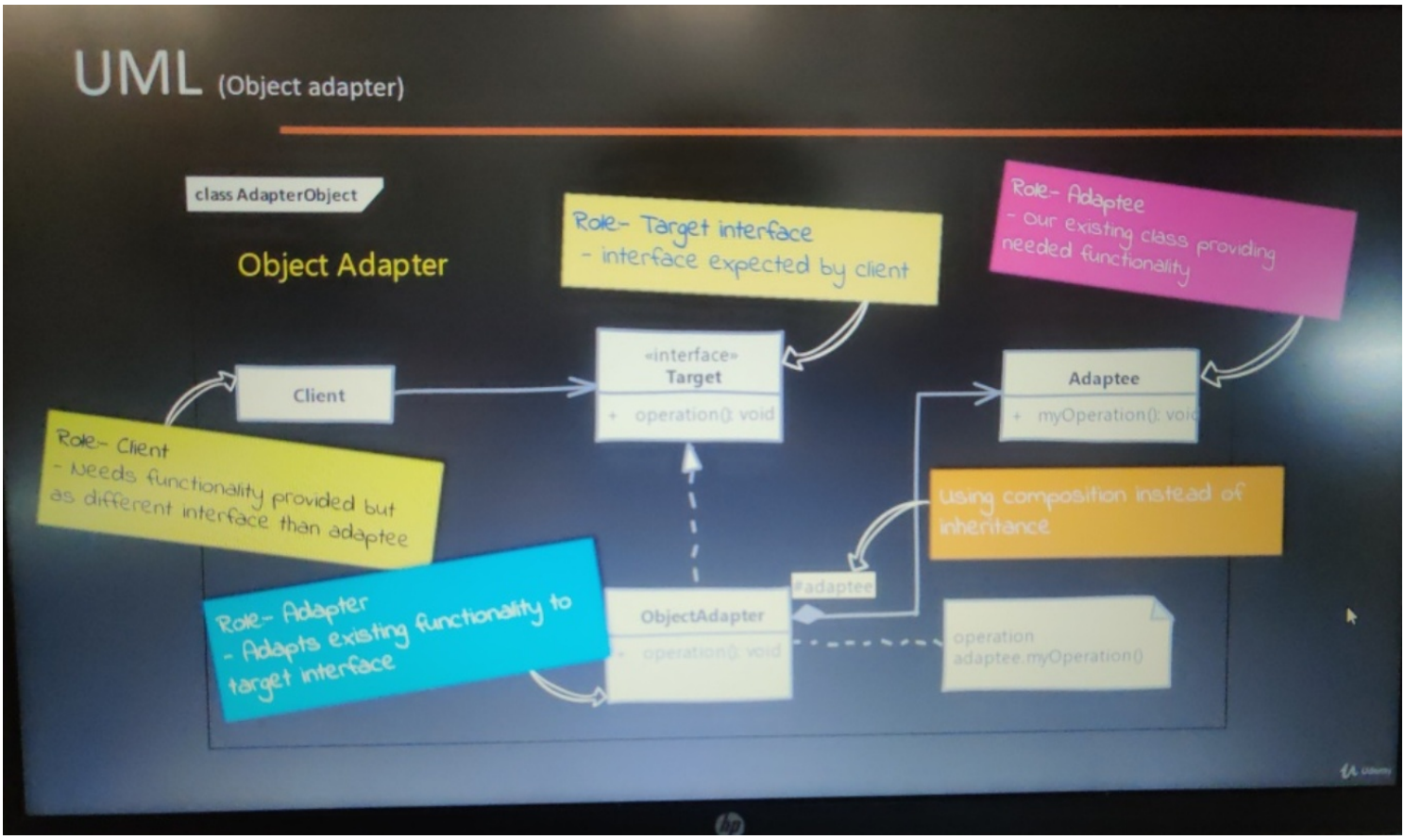
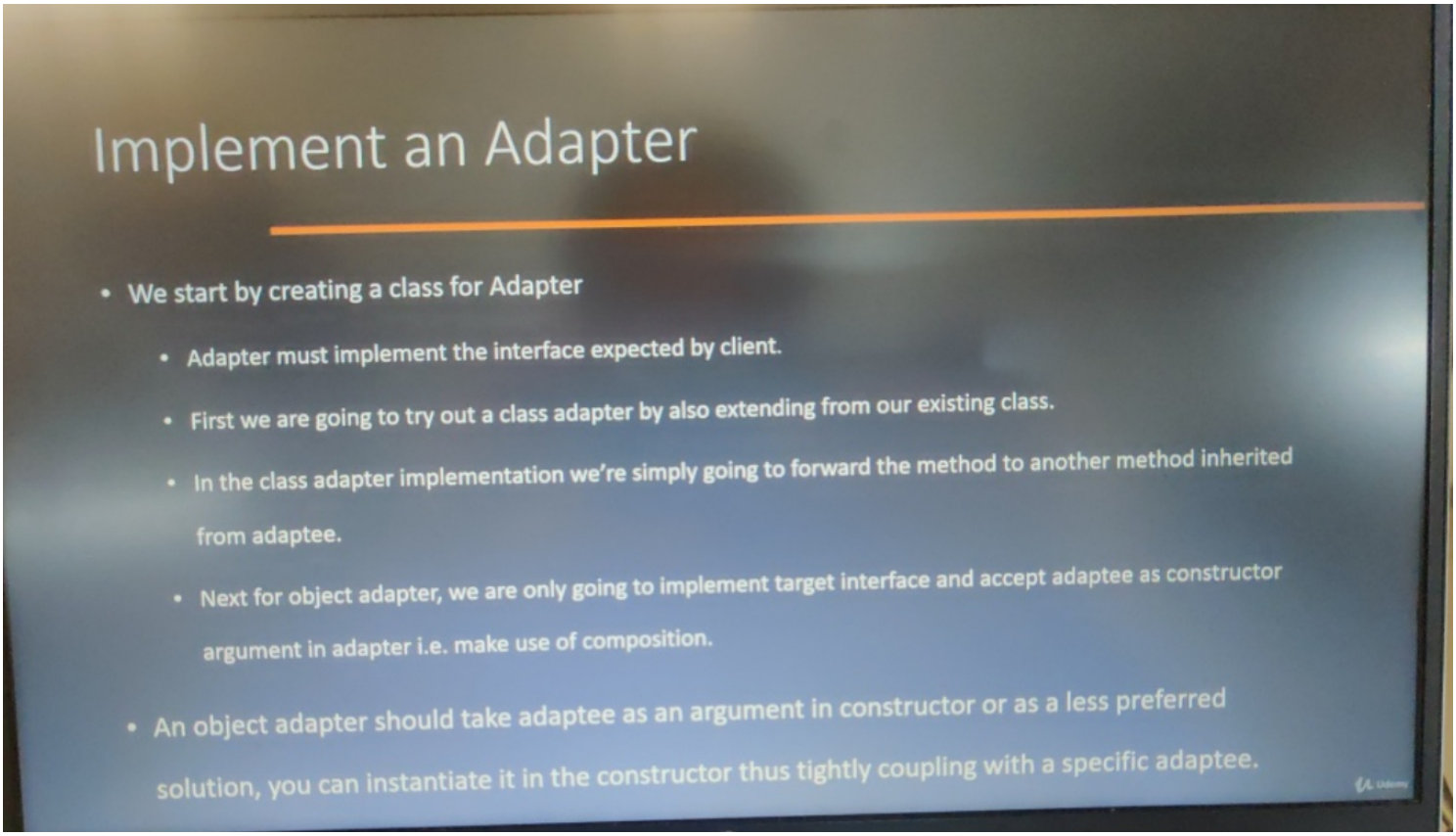

Design Patterns
Structural
Structural patterns deal with how classes and objects are arranged or composed


Design Patterns
Structural
Adapter
Bridge
Decorator
Composite
Facade
Flyweight
Proxy

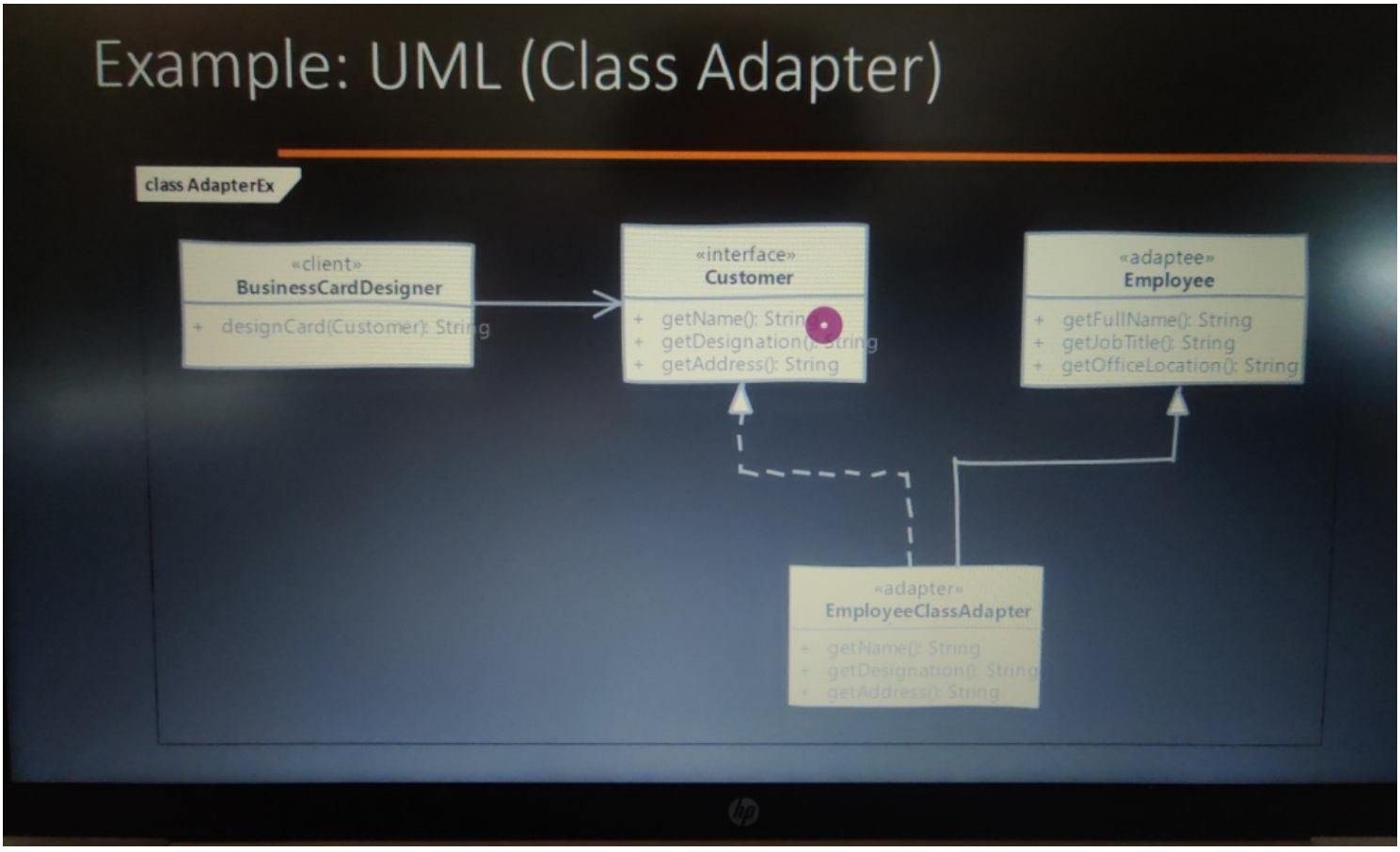

Adapter (aka Wrapper)
Structural Design Patterns
Design Patterns in Java


What is Adapter?
• We have an existing object which provides the functionality that client needs. But client code can't use this object because it expects an object with different interface.
• Using adapter design pattern we make this existing object work with client by adapting the object to client's expected interface.
• This pattern is also called as wrapper as it "wraps" existing object.

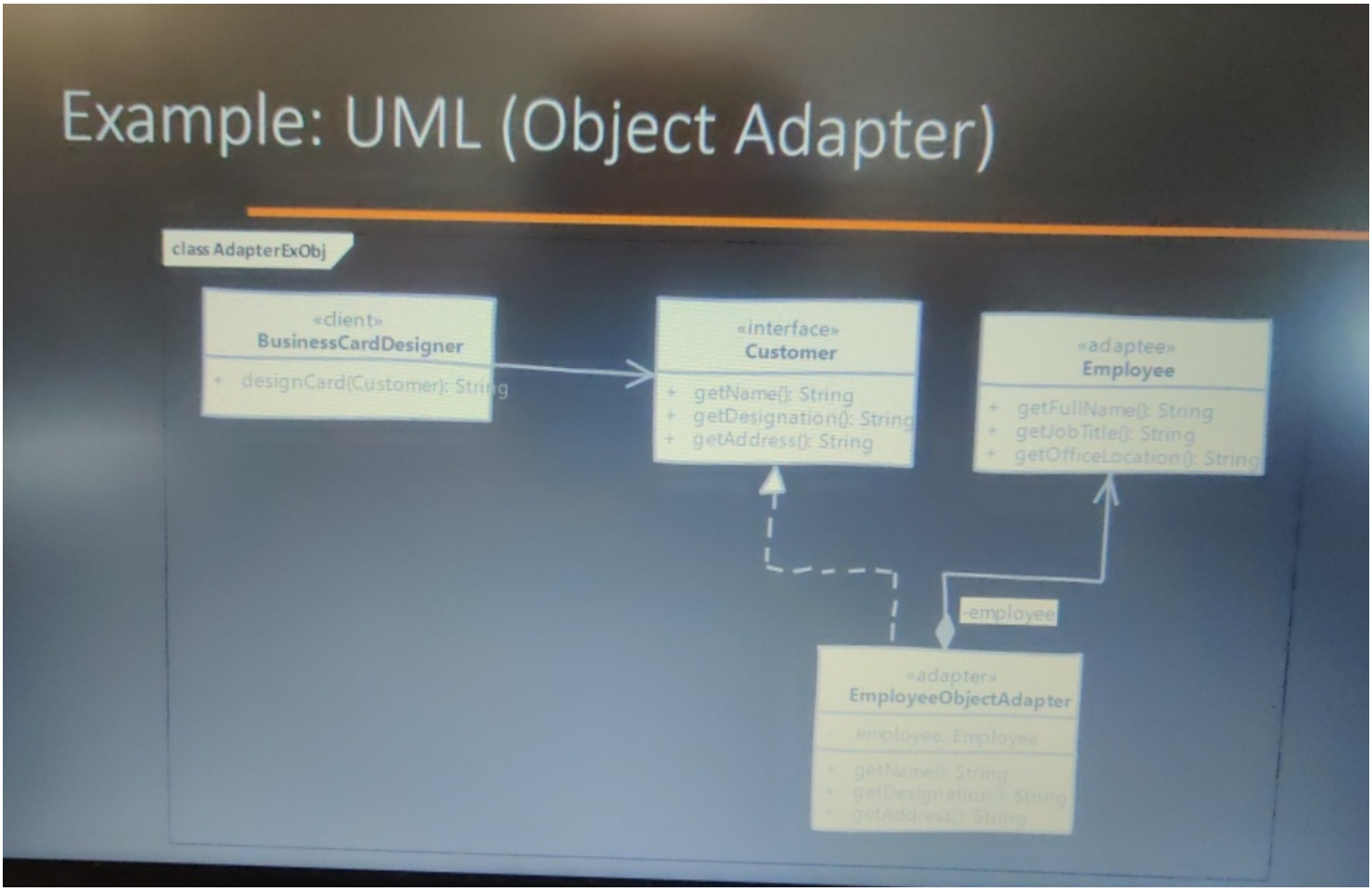

UML (Object adapter)
class AdapterObject
Object Adapter
Role- Target interface
- interface expected by client
Role- Adaptee
- our existing class providing needed functionality
«interface»
Target
+ operation(): void
Client
Adaptee
+ myOperation(): void
Role- Client
- Needs functionality provided but as different interface than adaptee
using composition instead of inheritance
#adaptee
ObjectAdapter
+ operation(): void
operation
adaptee.myOperation()
Role- Adapter
- Adapts existing functionality to target interface

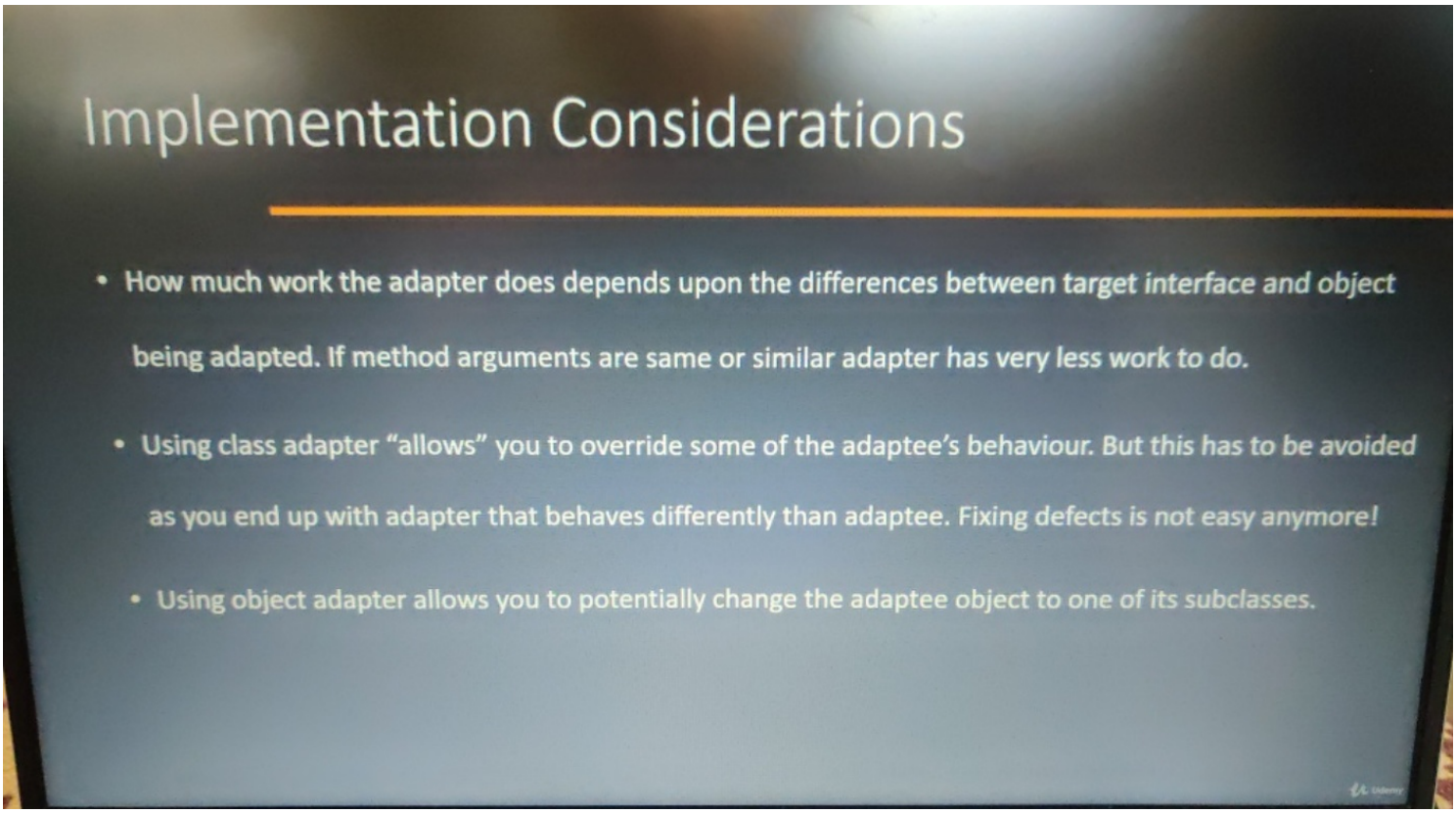

Implement an Adapter
• We start by creating a class for Adapter
• Adapter must implement the interface expected by client.
• First we are going to try out a class adapter by also extending from our existing class.
• In the class adapter implementation we're simply going to forward the method to another method inherited from adaptee.
• Next for object adapter, we are only going to implement target interface and accept adaptee as constructor argument in adapter i.e. make use of composition.
• An object adapter should take adaptee as an argument in constructor or as a less preferred solution, you can instantiate it in the constructor thus tightly coupling with a specific adaptee.

Example: UML (Class Adapter)



Example: UML (Object Adapter)




Implementation Considerations
• How much work the adapter does depends upon the differences between target interface and object being adapted. If method arguments are same or similar adapter has very less work to do.
• Using class adapter "allows" you to override some of the adaptee's behaviour. But this has to be avoided as you end up with adapter that behaves differently than adaptee. Fixing defects is not easy anymore!
• Using object adapter allows you to potentially change the adaptee object to one of its subclasses.

Design Considerations

- In java a "class adapter" may not be possible if both target and adaptee are concrete classes. In such cases the object adapter is the only solution. Also since there is no private inheritance in Java, it's better to stick with object adapter.
- A class adapter is also called as a two way adapter, since it can stand in for both the target interface and for the adaptee. That is we can use object of adapter where either target interface is expected as well as where an adaptee object is expected.

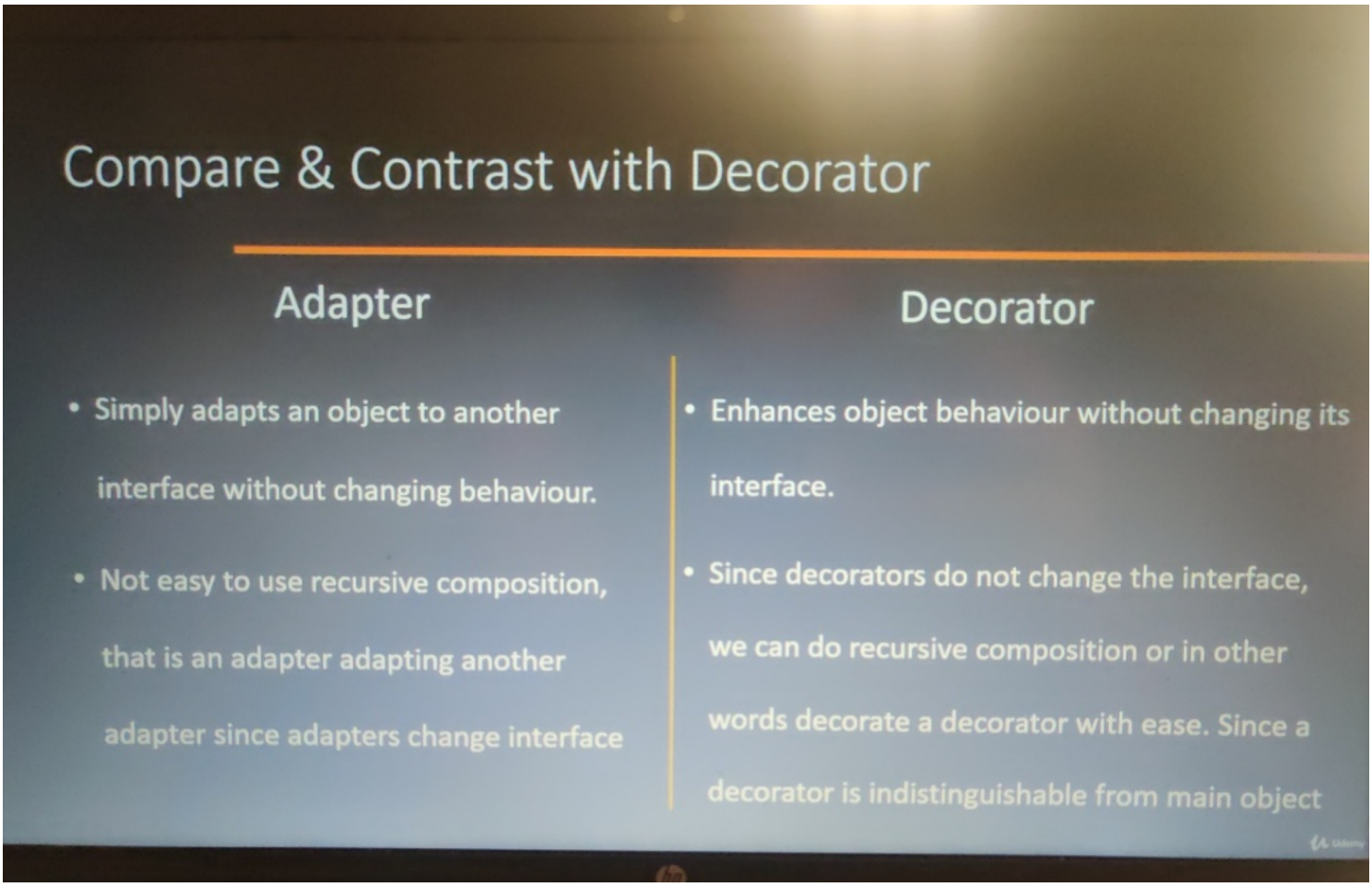
Example of an Adapter

- The java.io.InputStreamReader and java.io.OutputStreamWriter classes are examples of object adapters.
- These classes adapt existing InputStream/OutputStream object to a Reader/Writer interface.

```
public class InputStreamReader extends Reader {

    private final StreamDecoder sd;

    /**
     * Creates an InputStreamReader that uses the default charset.
     *
     * @param  in   An InputStream
     */
    public InputStreamReader(InputStream in) {
        super(in);
        try {
            sd = StreamDecoder.forInputStreamReader(in, this, (Str
        } catch (UnsupportedEncodingException e) {
            // The default encoding should always be available
            throw new Error(e);
        }
    }
    public int read(char cbuf[], int offset, int length) throws IOE
        return sd.read(cbuf, offset, length);
    }
    public int read() throws IOException {
        return sd.read();
    }
```

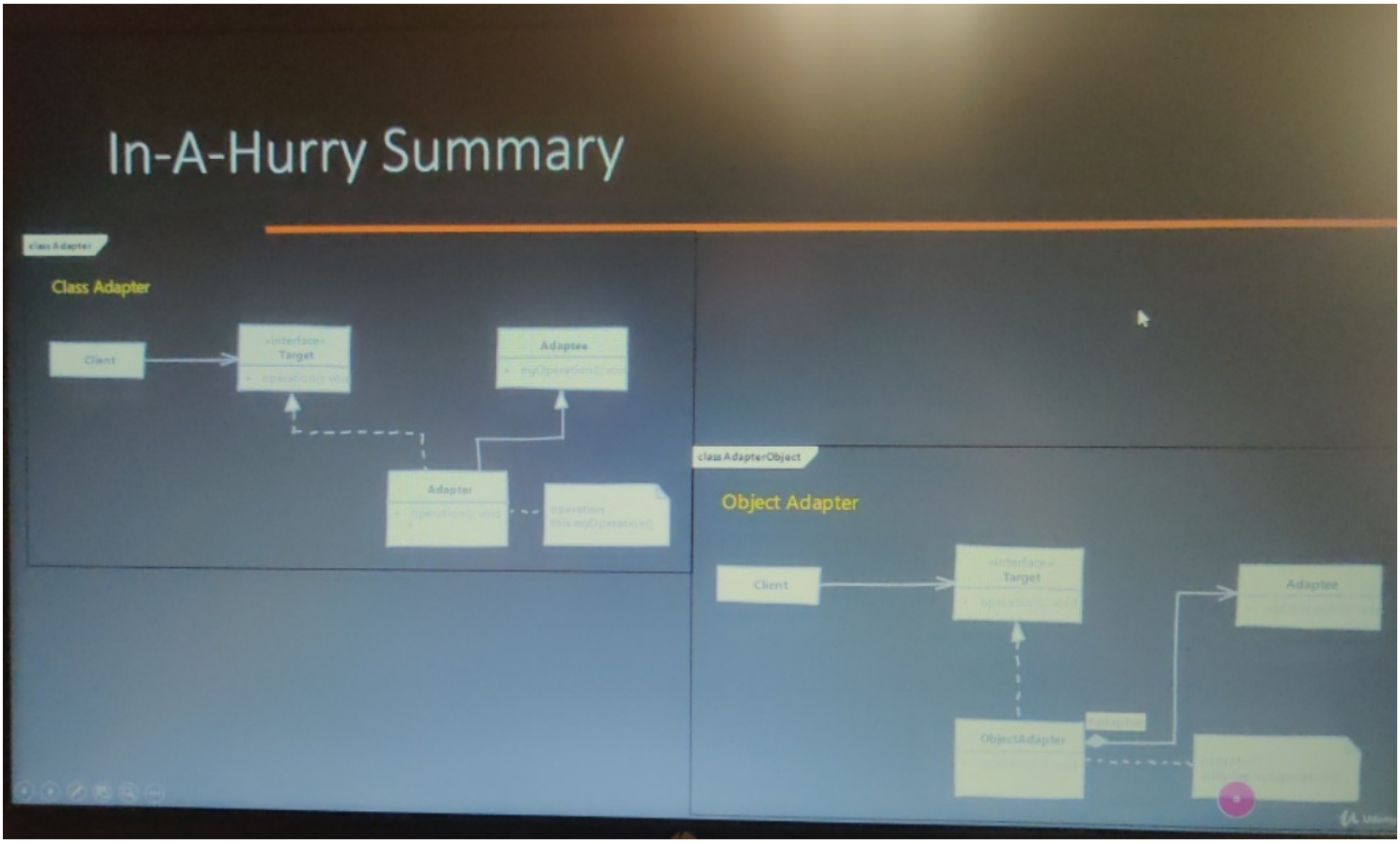

Compare & Contrast with Decorator
Adapter
Simply adapts an object to another interface without changing behaviour.
Not easy to use recursive composition, that is an adapter adapting another adapter since adapters change interface
Decorator
Enhances object behaviour without changing its interface.
Since decorators do not change the interface, we can do recursive composition or in other words decorate a decorator with ease. Since a decorator is indistinguishable from main object

Pitfalls

- Using target interface and adaptee class to extend our adapter we can create a "class adapter" in java. However it creates an object which exposes unrelated methods in parts of your code, polluting it. Avoid class adapters! It is mentioned here only for sake of completeness.
- It is tempting to do a lot of things in adapter besides simple interface translation. But this can result in an adapter showing different behaviour than the adapted object.
- Not a lot of other pitfalls! As long as we keep them true to their purpose of simple interface translation they are good.

In-A-Hurry Summary

- We have an existing object with required functionality but the client code is expecting a different interface than our object.
- A class adapter is one where adapter inherits from class of object which is to be adapted and implements the interface required by client code. This adapter type should be avoided.
- An object adapter uses composition . It'll implement the target interface and use an adaptee object composition to perform translation. This allows us to use subclasses of adaptee in adapter.


In-A-Hurry Summary
class Adapter
Class Adapter
Client
«interface»
Target
Adaptee
Adapter
operation
this.myOperation()
class AdapterObject
Object Adapter
Client
«interface»
Target
Adaptee
ObjectAdapter