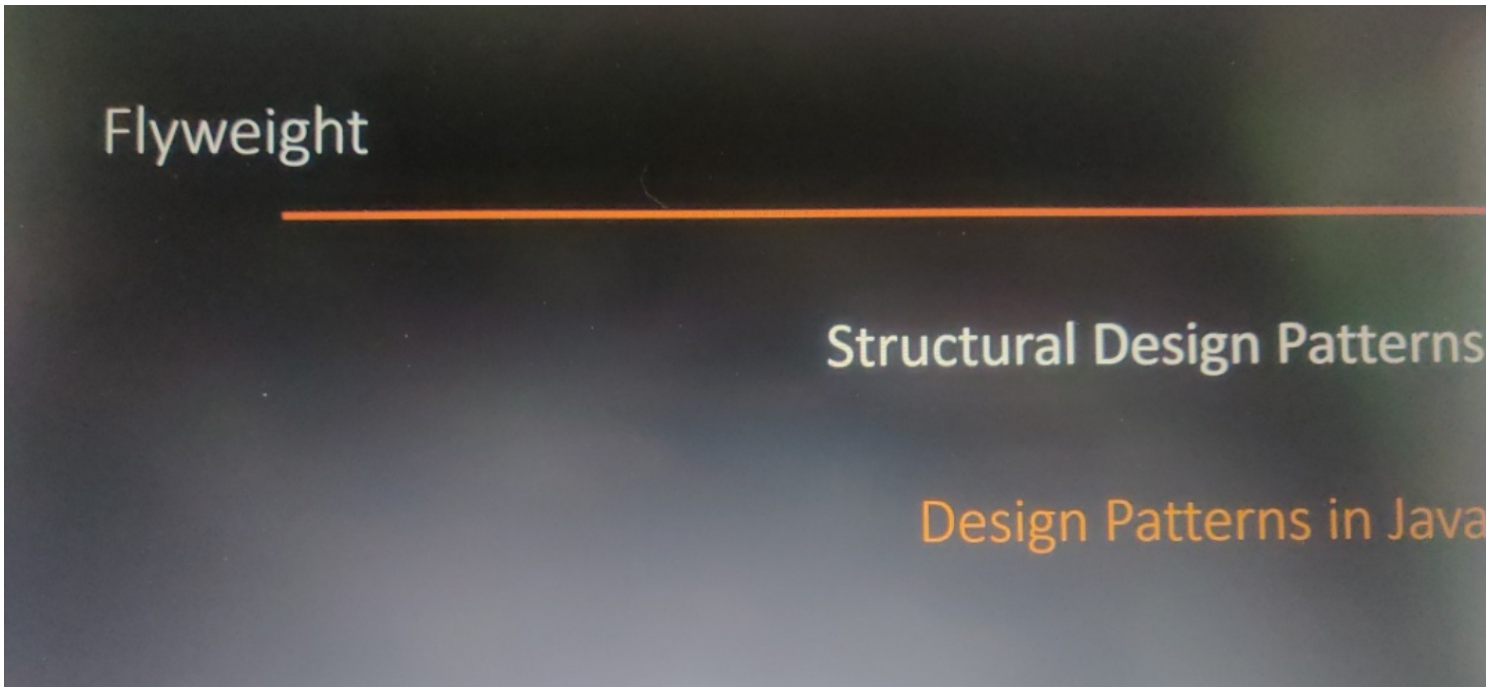
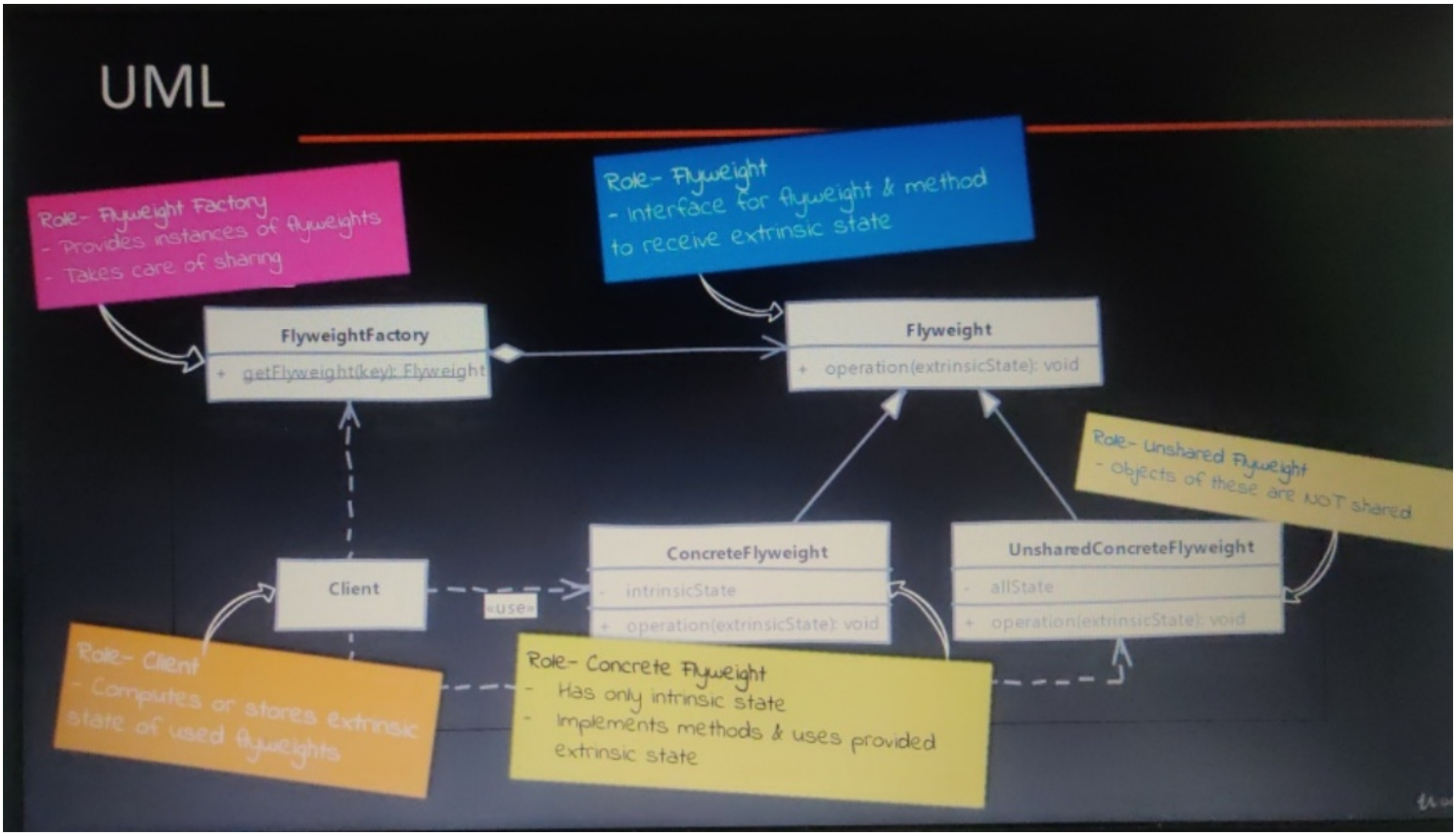

Flyweight
Structural Design Patterns
Design Patterns in Java

What is Flyweight?

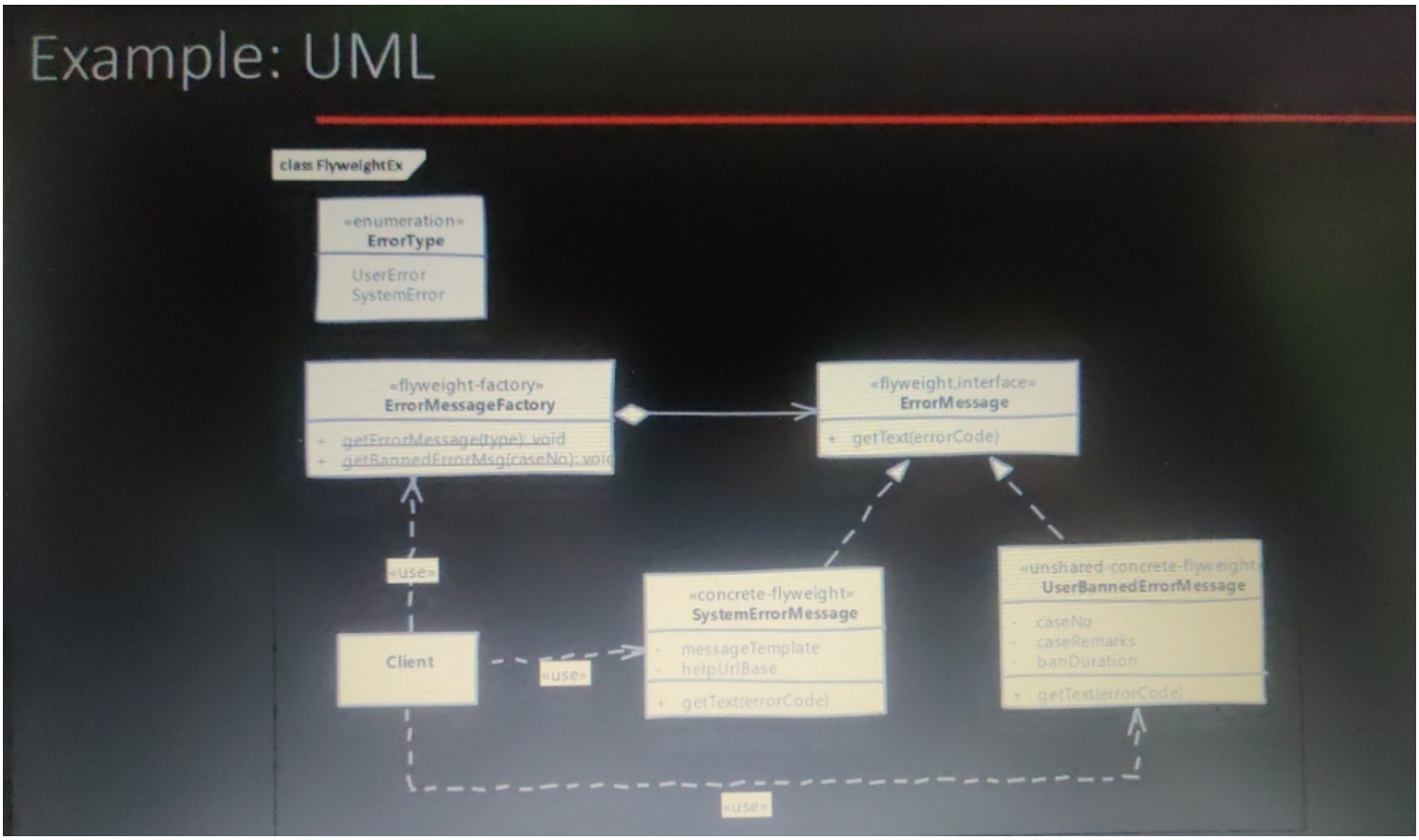
- Our system needs a large number of objects of a particular class & maintaining these instances is a performance concern.
- Flyweight allows us to share an object in multiple contexts. But instead of sharing entire object, which may not be feasible, we divide object state in two parts: intrinsic (state that is shared in every context) & extrinsic state (context specific state). We create objects with only intrinsic state and share them in multiple contexts.
- Client or user of object provides the extrinsic state to object to carry out its functionality.
- We provide a factory so client can get required flyweight objects based on some key to identify flyweight.


UML
Role- Flyweight Factory
- Provides instances of flyweights
- Takes care of sharing
Role- Flyweight
- Interface for flyweight & method to receive extrinsic state
FlyweightFactory
+ getFlyweight(key): Flyweight
Flyweight
+ operation(extrinsicState): void
Role- Unshared Flyweight
- objects of these are NOT shared
Client
«use»
ConcreteFlyweight
- intrinsicState
+ operation(extrinsicState): void
UnsharedConcreteFlyweight
- allState
+ operation(extrinsicState): void
Role- Client
- Computes or stores extrinsic state of used flyweights
Role- Concrete Flyweight
- Has only intrinsic state
- Implements methods & uses provided extrinsic state

Implement a Flyweight

- We start by identifying "intrinsic" & "extrinsic" state of our object
  - We create an interface for flyweight to provide common methods that accept extrinsic state
  - In implementation of shared flyweight we add intrinsic state & also implement methods.
  - In unshared flyweight implementation we simply ignore the extrinsic state argument as we have all state within object.
- Next we implement the flyweight factory which caches flyweights & also provides method to get them
- In our client we either maintain the extrinsic state or compute it on the fly when using flyweight

Example: UML


class FlyweightEx
«enumeration»
ErrorType
UserError
SystemError
«flyweight-factory»
ErrorMessageFactory
+ getErrorMessage(type): void
+ getBannedErrorMsg(caseNo): voi
«flyweight,interface»
ErrorMessage
+ getText(errorCode)
«use»
Client
«use»
«concrete-flyweight»
SystemErrorMessage
- messageTemplate
- helpUrlBase
+ getText(errorCode)
«unshared-concrete-flyweight»
UserBannedErrorMessage
- caseNo
- caseRemarks
- banDuration
+ getText(errorCode)
«use»

Implementation Considerations

- A factory is necessary with flyweight design pattern as client code needs easy way to get hold of shared flyweight. Also number of shared instances can be large so a central place is good strategy to keep track of all of them.
- Flyweight's intrinsic state should be immutable for successful use of flyweight pattern.

Design Considerations

- Usability of flyweight is entirely dependent upon presence of sensible extrinsic state in object which can be moved out of object without any issue.
- Some other design patterns like state and strategy can make best use of flyweight pattern.

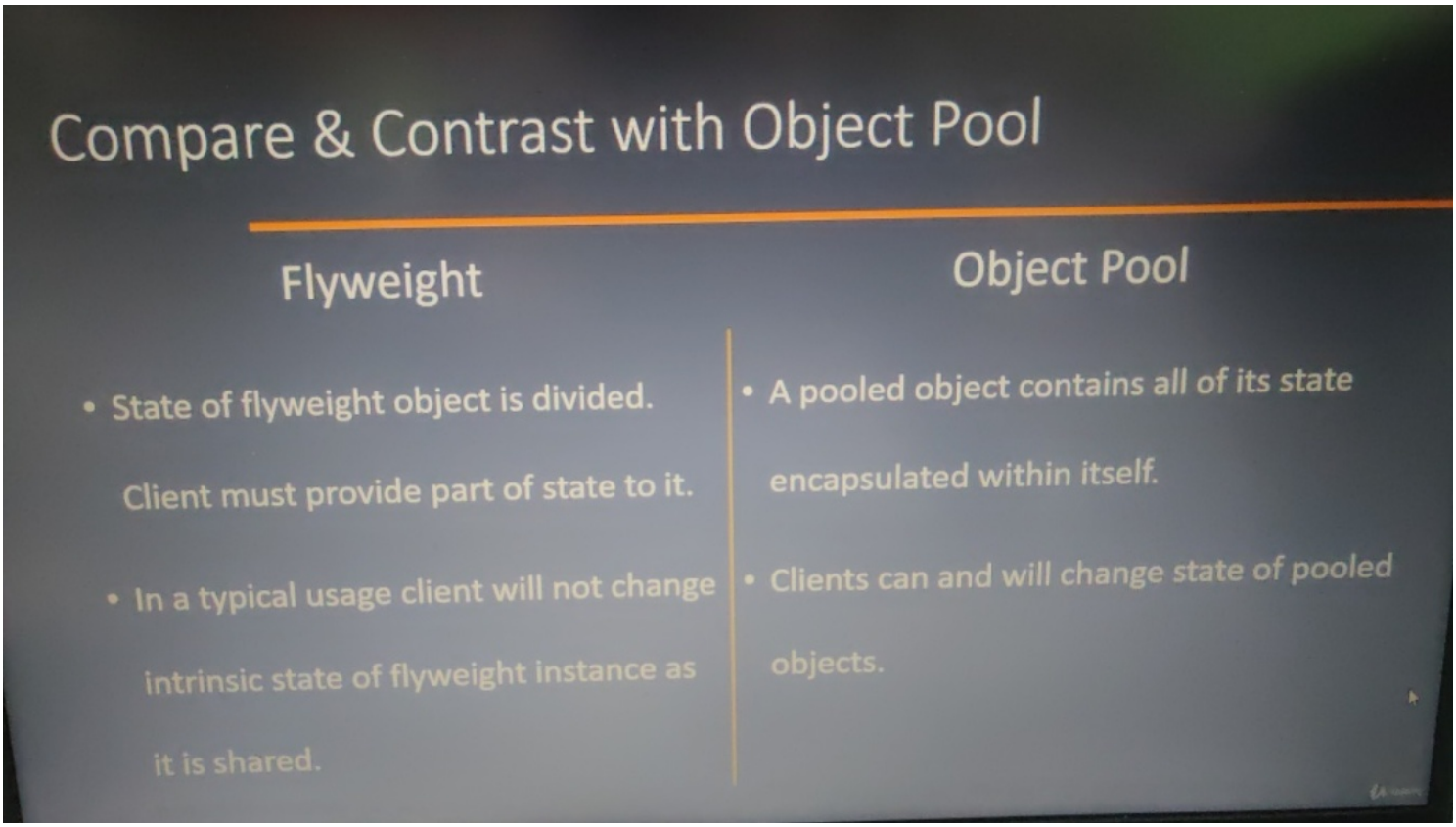
Examples of a Flyweight

- Java uses flyweight pattern for Wrapper classes like java.lang.Integer, Short, Byte etc. Here the valueOf static method serves as the factory method.

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

- String pool which is maintained by JVM is also an example of flyweight. We can call the intern() method on a String object to explicitly request this String object to be interned. This method will returned a reference to already cached object if present or else will create new String in cache if not present.

Note:- String.intern() is a native method.

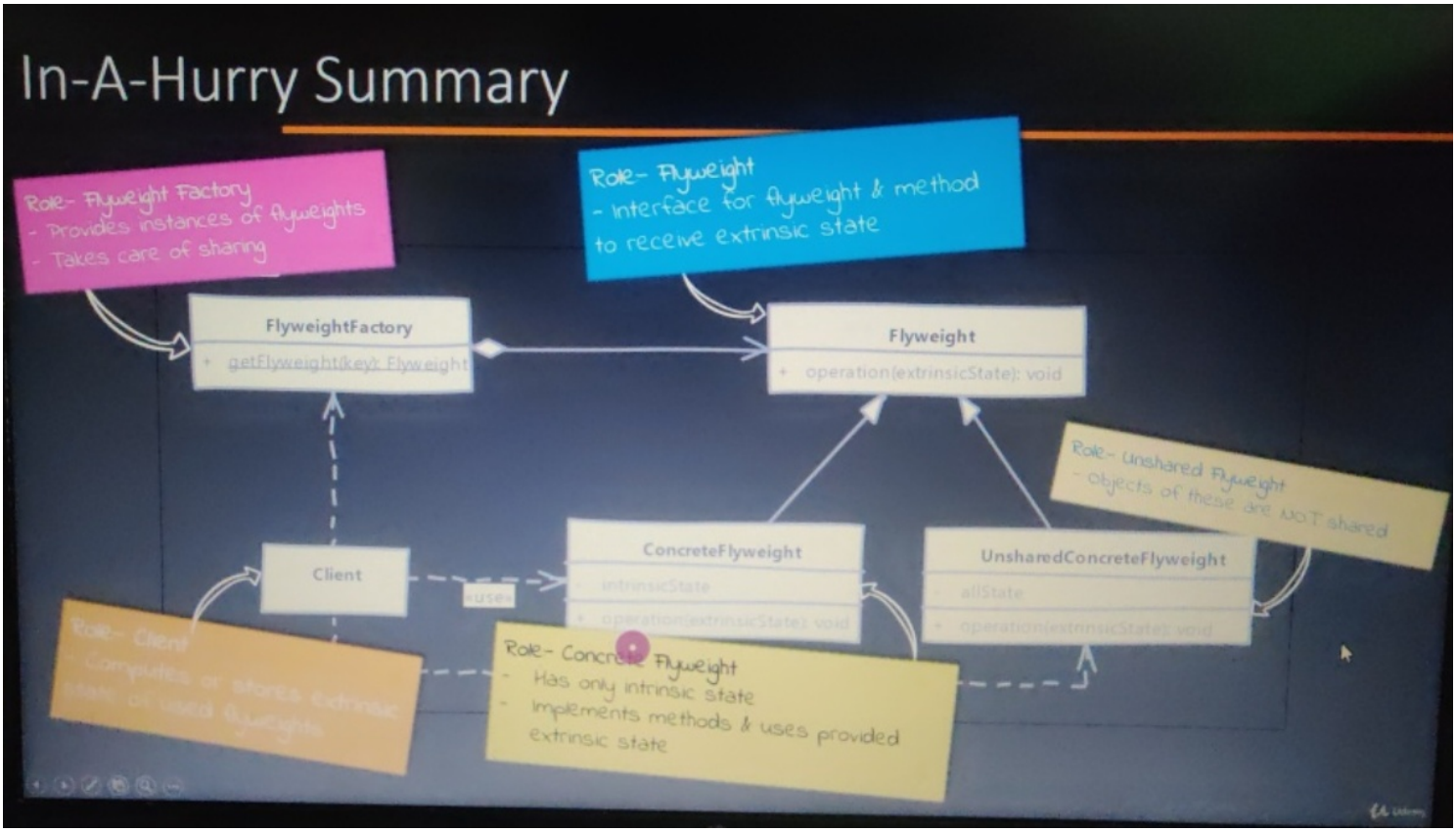

Compare & Contrast with Object Pool
Flyweight
• State of flyweight object is divided. Client must provide part of state to it.
• In a typical usage client will not change intrinsic state of flyweight instance as it is shared.
Object Pool
• A pooled object contains all of its state encapsulated within itself.
• Clients can and will change state of pooled objects.

Pitfalls

- Runtime cost may be added for maintaining extrinsic state. Client code has to either maintain it or compute it every time it needs to use flyweight.
- It is often difficult to find perfect candidate objects for flyweight. Graphical applications benefit heavily from this pattern however a typical web application may not have a lot of use for this pattern.

In-A-Hurry Summary

- We use flyweight design pattern if we need large number of objects of class where we can easily separate out state that can be shared and state that can be externalized.
- Flyweights store only "intrinsic" state or state that can be shared in any context.
- Code using flyweight instance provides the extrinsic state when calling methods on flyweight. Flyweight object then uses this state along with its inner state to carry out the work.
- Client code can store extrinsic per flyweight instance it uses or compute it on the fly.


In-A-Hurry Summary
Role- Flyweight Factory
- Provides instances of flyweights
- Takes care of sharing
Role- Flyweight
- Interface for flyweight & method to receive extrinsic state
FlyweightFactory
+ getFlyweight(key): Flyweight
Flyweight
+ operation(extrinsicState): void
Role- Unshared Flyweight
- objects of these are NOT shared
Client
uses
ConcreteFlyweight
intrinsicState
+ operation(extrinsicState): void
UnsharedConcreteFlyweight
allState
+ operation(extrinsicState): void
Role- Client
- Computes or stores extrinsic state of used flyweights
Role- Concrete Flyweight
- Has only intrinsic state
- Implements methods & uses provided extrinsic state