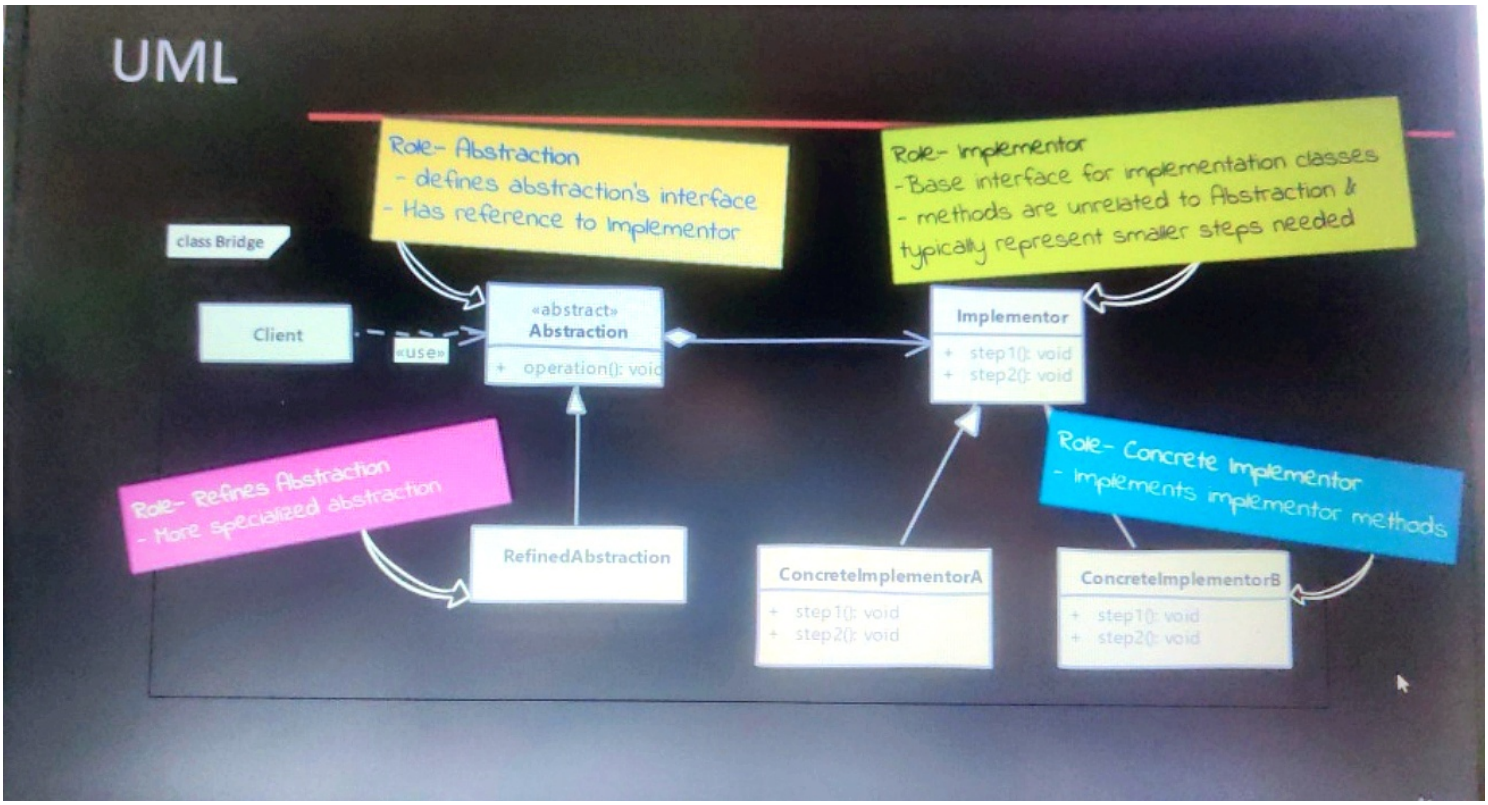

Bridge
Structural Design Patterns
Design Patterns in Java

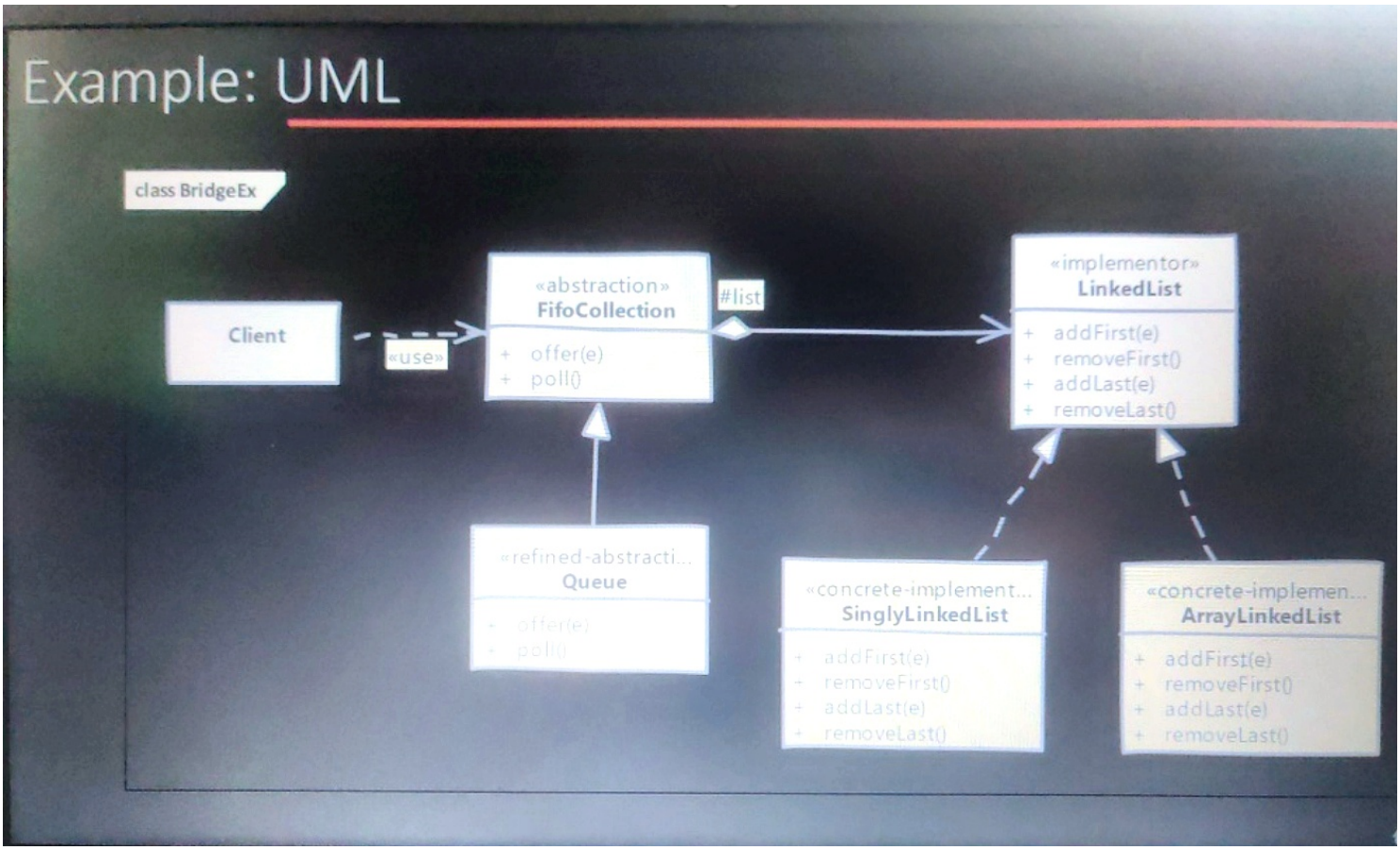
What is Bridge?

- Our implementations & abstractions are generally coupled to each other in normal inheritance.
- Using bridge pattern we can decouple them so they can both change without affecting each other.
- We achieve this feat by creating two separate inheritance hierarchies; one for implementation and another for abstraction.
- We use composition to bridge these two hierarchies.


UML
Role- Abstraction
- defines abstraction's interface
- Has reference to Implementor
Role- Implementor
- Base interface for implementation classes
- methods are unrelated to Abstraction &
typically represent smaller steps needed
class Bridge
Client
«use»
«abstract»
Abstraction
+ operation(): void
Implementor
+ step1(): void
+ step2(): void
Role- Refines Abstraction
- More specialized abstraction
RefinedAbstraction
ConcreteImplementorA
+ step1(): void
+ step2(): void
ConcreteImplementorB
+ step1(): void
+ step2(): void
Role- Concrete Implementor
- Implements implementor methods

Implement a Bridge

- We start by defining our abstraction as needed by client
  - We determine common base operations and define them in abstraction.
  - We can optionally also define a refined abstraction & provide more specialized operations.
  - Then we define our implementor next. Implementor methods do NOT have to match with abstractor. However abstraction can carry out its work by using implementor methods
  - Then we write one or more concrete implementor providing implemention
- Abstractions are created by composing them with an instance of concrete implementor which is used by methods in abstraction.


Example: UML
class BridgeEx
Client
«use»
«abstraction»
FifoCollection
+ offer(e)
+ poll()
#list
«implementor»
LinkedList
+ addFirst(e)
+ removeFirst()
+ addLast(e)
+ removeLast()
«refined-abstracti...
Queue
+ offer(e)
+ poll()
«concrete-implement...
SinglyLinkedList
+ addFirst(e)
+ removeFirst()
+ addLast(e)
+ removeLast()
«concrete-implemen...
ArrayLinkedList
+ addFirst(e)
+ removeFirst()
+ addLast(e)
+ removeLast()

Implementation Considerations

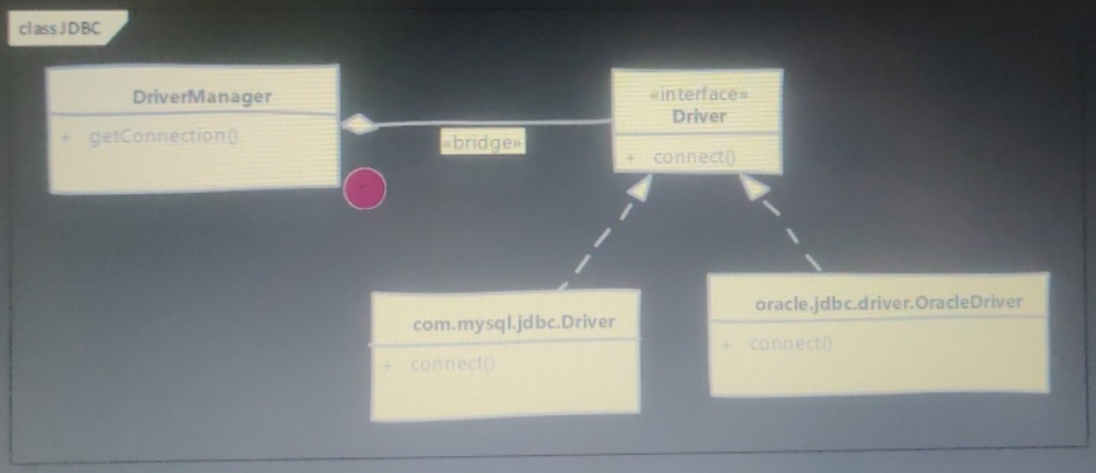
- In case we are ever going to have a single implementation then we can skip creating abstract implementor.
- Abstraction can decide on its own which concrete implementor to use in its constructor or we can delegate that decision to a third class. In last approach abstraction remains unaware of concrete implementors & provides greater de-coupling.

Design Considerations

- Bridge provides great extensibility by allowing us to change abstraction and implementor independently. You can build & package them separately to modularize overall system.
- By using abstract factory pattern to create abstraction objects with correct implementation you can de-couple concrete implementors from abstraction.

Example of a Bridge

- An example of bridge pattern often given is the JDBC API. More specifically the java.sql.DriverManager class with the java.sql.Driver interface form a bridge pattern.



Example of a Bridge

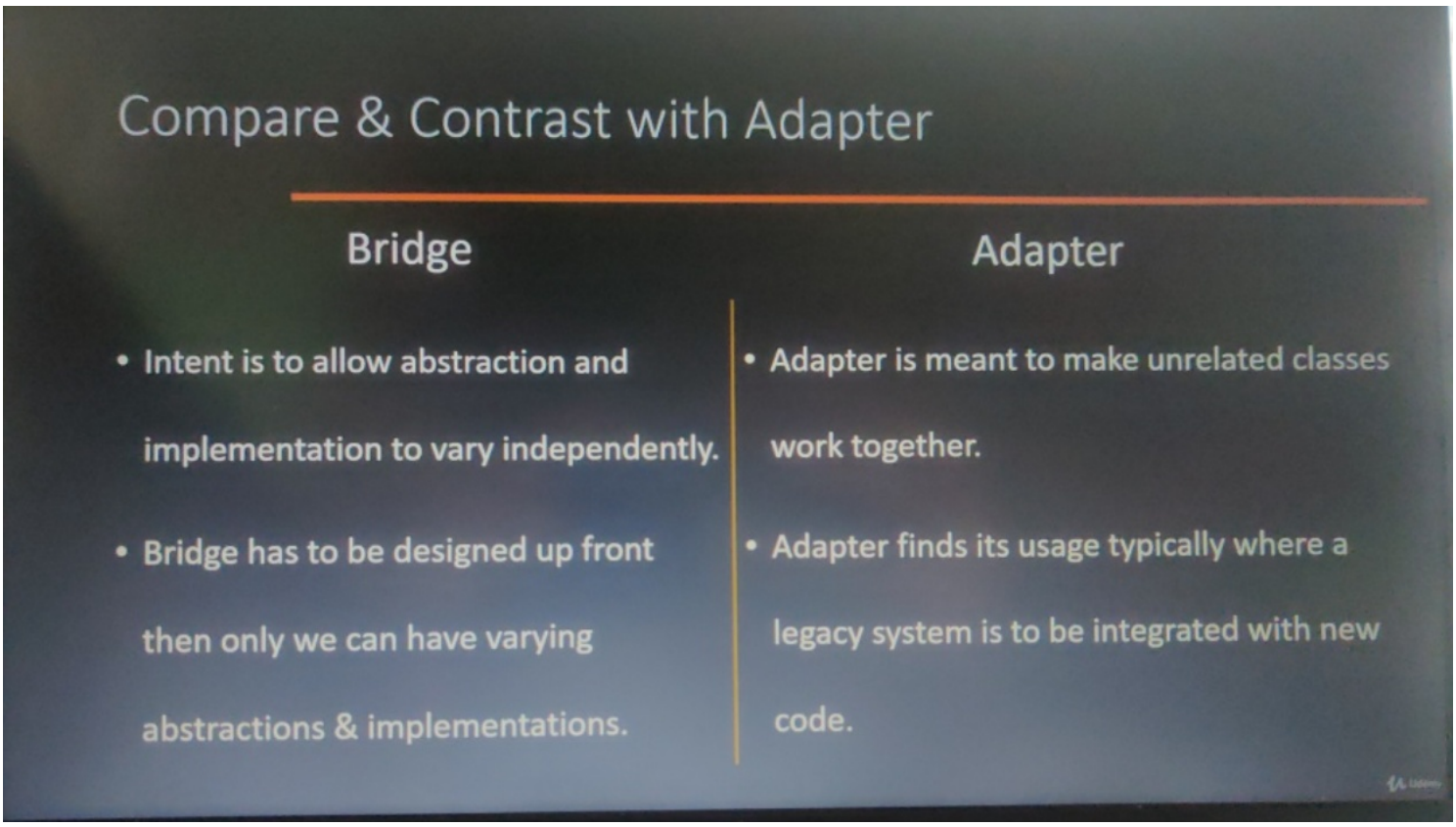
- An example of bridge pattern often given is the Collections.newSetFromMap() method. This method returns a Set which is backed by given map object.

```
private static class SetFromMap<E> extends AbstractSet<E>
    implements Set<E>, Serializable
{
    private final Map<E, Boolean> m;  // The backing map
    private transient Set<E> s;       // Its keySet

    SetFromMap(Map<E, Boolean> map) {
        if (!map.isEmpty())
            throw new IllegalArgumentException("Map is non-empty");
        m = map;
        s = map.keySet();
    }

    public void clear()               {        m.clear(); }
    public int size()                 { return m.size(); }
    public boolean isEmpty()          { return m.isEmpty(); }
    public boolean contains(Object o) { return m.containsKey(o); }
    public boolean remove(Object o)   { return m.remove(o) != null; }
    public boolean add(E e) { return m.put(e, Boolean.TRUE) == null; }
    public Iterator<E> iterator()     { return s.iterator(); }
    public Object[] toArray()         { return s.toArray(); }
    public <T> T[] toArray(T[] a)     { return s.toArray(a); }
    public String toString()          { return s.toString(); }
    public int hashCode()             { return s.hashCode(); }
    public boolean equals(Object o)   { return o == this || s.equals(o); }
    public boolean containsAll(Collection<?> c) {return s.containsAll(c);}
    public boolean removeAll(Collection<?> c)   {return s.removeAll(c);}
```

Code taken from Collections.class


Compare & Contrast with Adapter
Bridge
Adapter
• Intent is to allow abstraction and implementation to vary independently.
• Bridge has to be designed up front then only we can have varying abstractions & implementations.
• Adapter is meant to make unrelated classes work together.
• Adapter finds its usage typically where a legacy system is to be integrated with new code.

Pitfalls

- It is fairly complex to understand & implement bridge design pattern.
- You need to have a well thought out & fairly comprehensive design in front of you before you can decide on bridge pattern.
- Needs to be designed up front. Adding bridge to legacy code is difficult. Even for ongoing project adding bridge at later time in development may require fair amount of rework.

In-A-Hurry Summary

- We use bridge pattern when we want our abstractions and implementations to be decoupled.
- Bridge pattern defines separate inheritance hierarchies for abstraction & implementations and bridge these two together using composition.
- Implementations do not HAVE to define methods that match up with methods in abstraction. It is fairly common to have primitive methods; methods which do small work; in implementor. Abstraction uses these methods to provide its functionality.