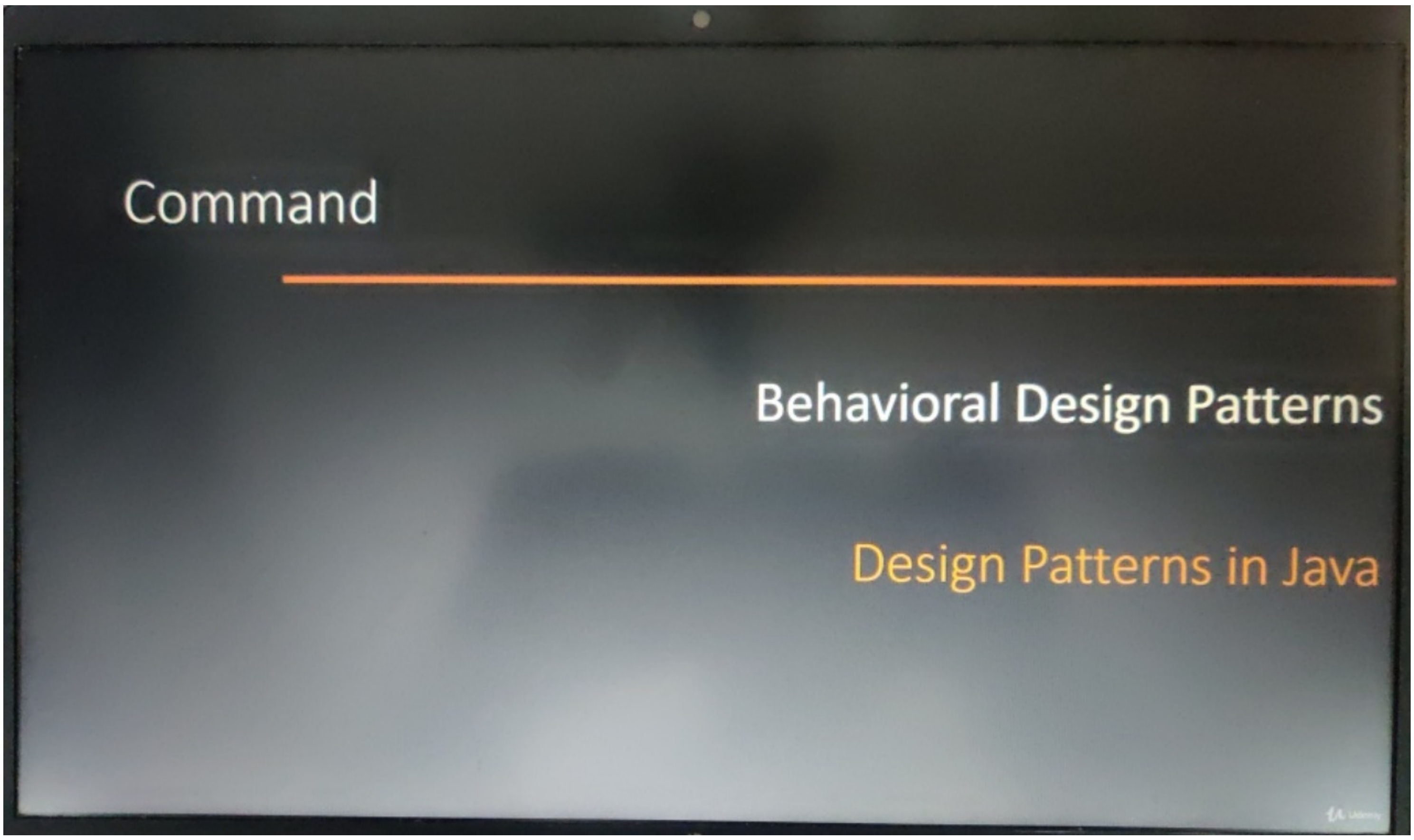
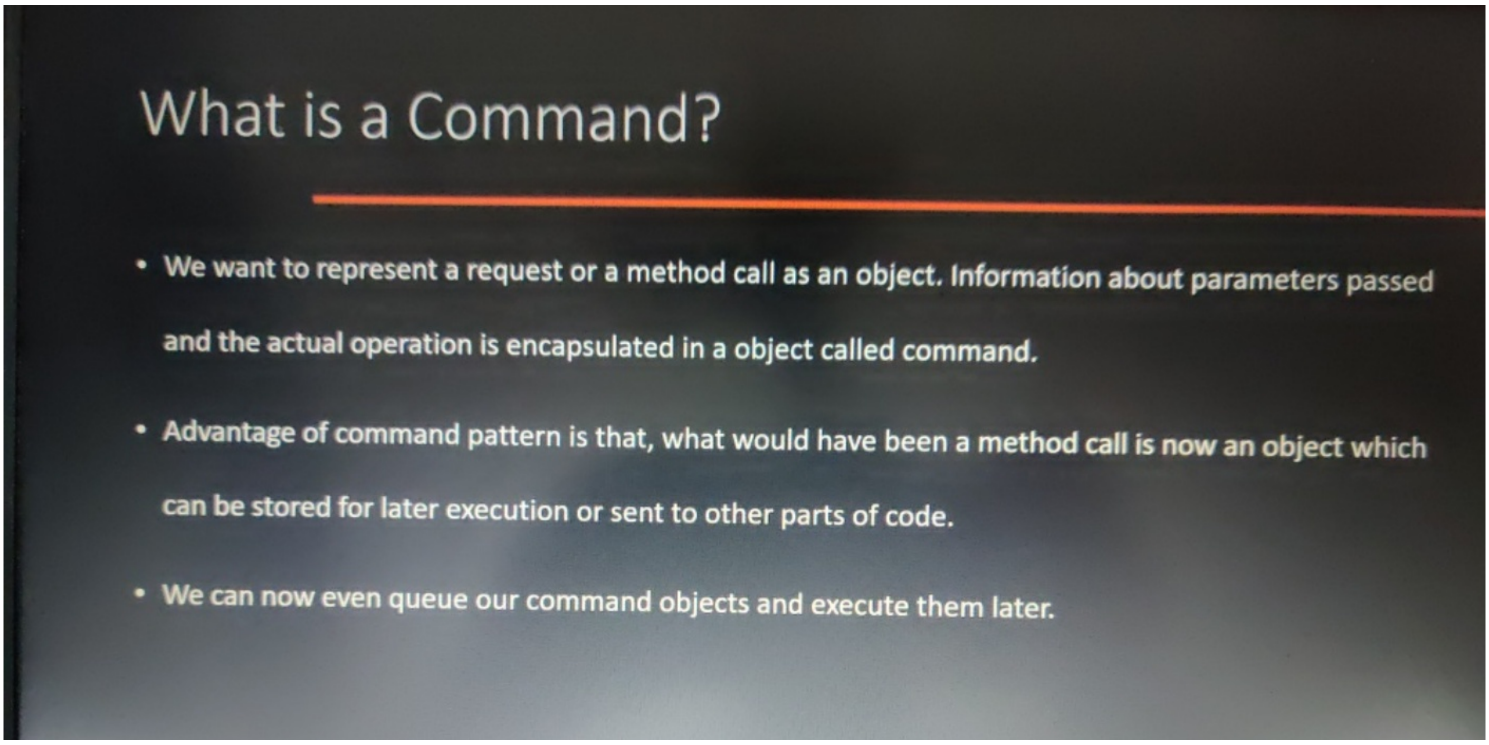


Command

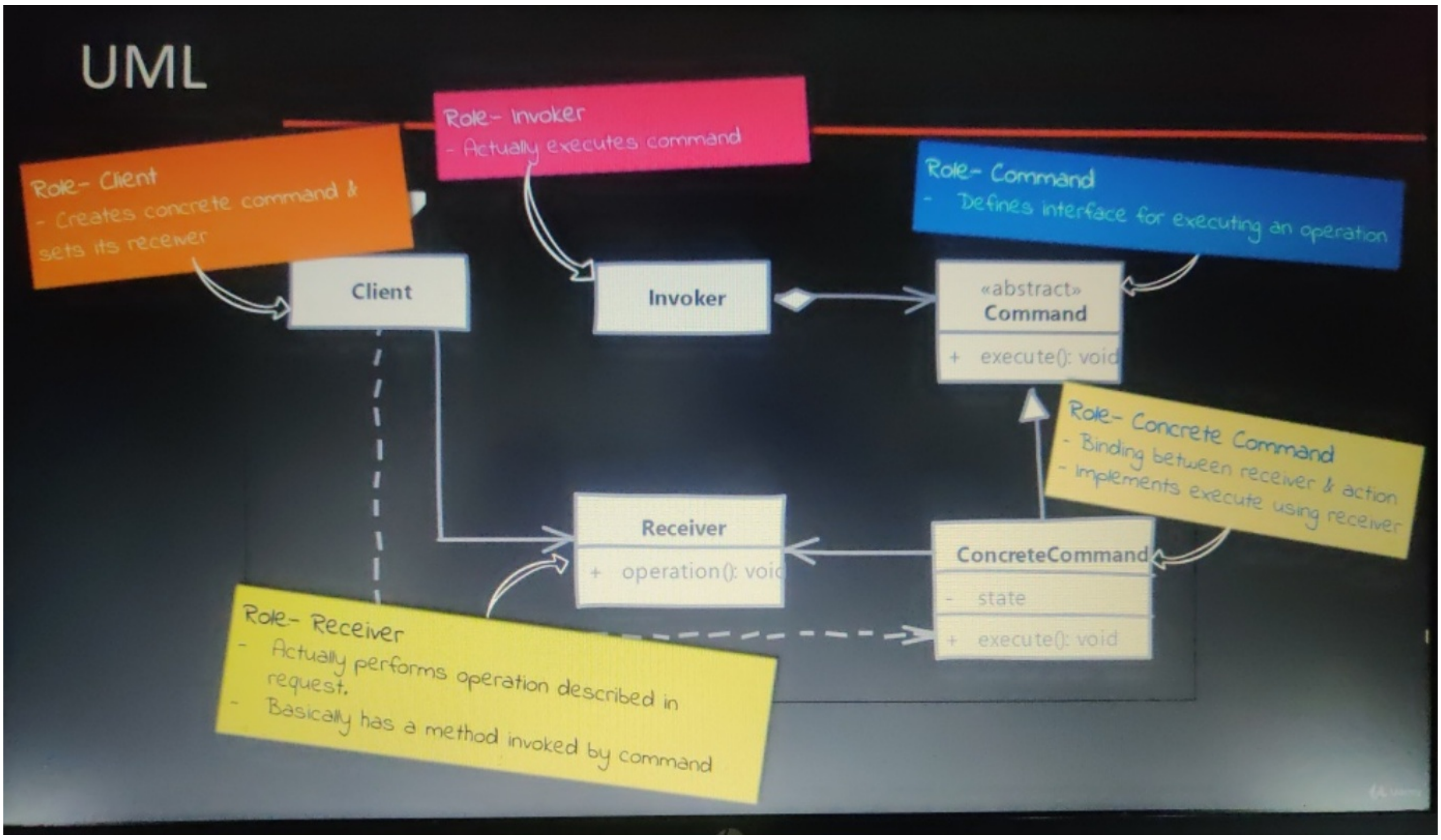
Behavioral Design Patterns

Design Patterns in Java



What is a Command?

- We want to represent a request or a method call as an object. Information about parameters passed and the actual operation is encapsulated in a object called command.
- Advantage of command pattern is that, what would have been a method call is now an object which can be stored for later execution or sent to other parts of code.
- We can now even queue our command objects and execute them later.



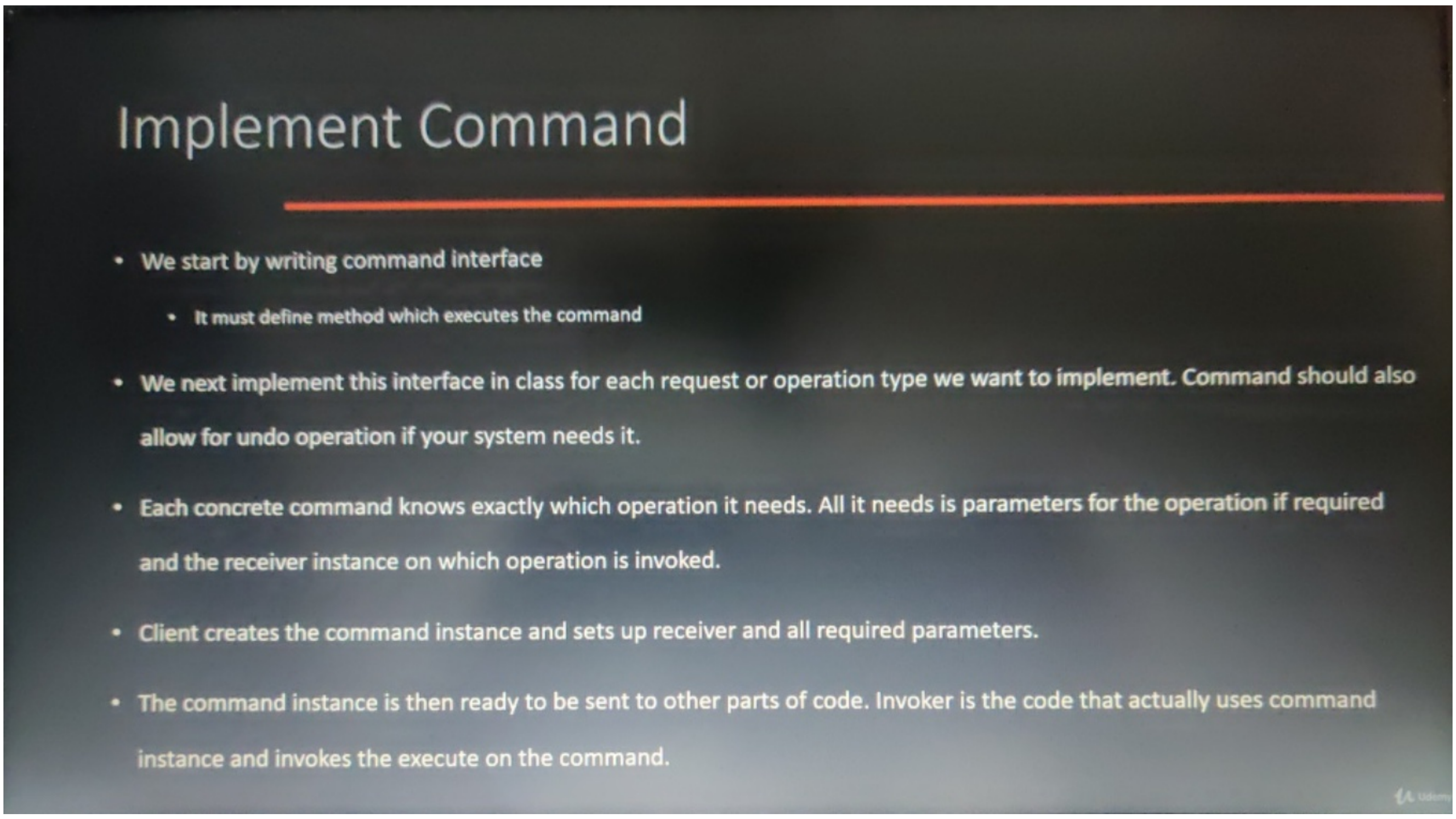
Role- Client
- Creates concrete command & sets its receiver

Role- Invoker
- Actually executes command

Role- Command
- Defines interface for executing an operation

Role- Concrete Command
- Binding between receiver & action
- Implements execute using receiver

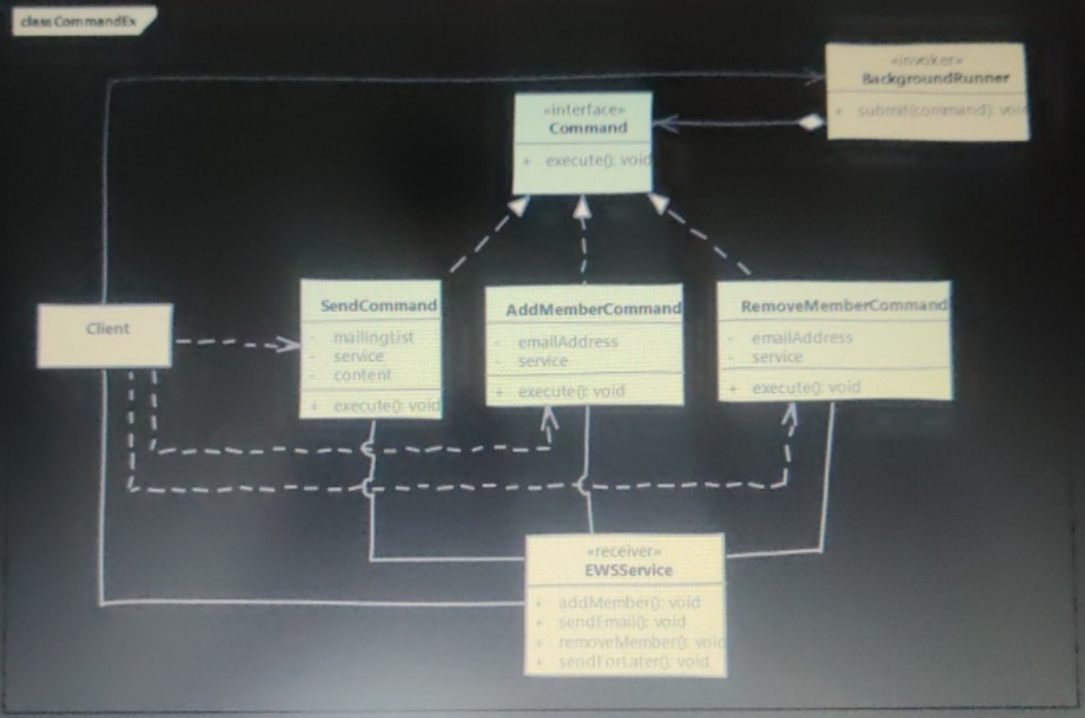
Role- Receiver
- Actually performs operation described in request.
- Basically has a method invoked by command



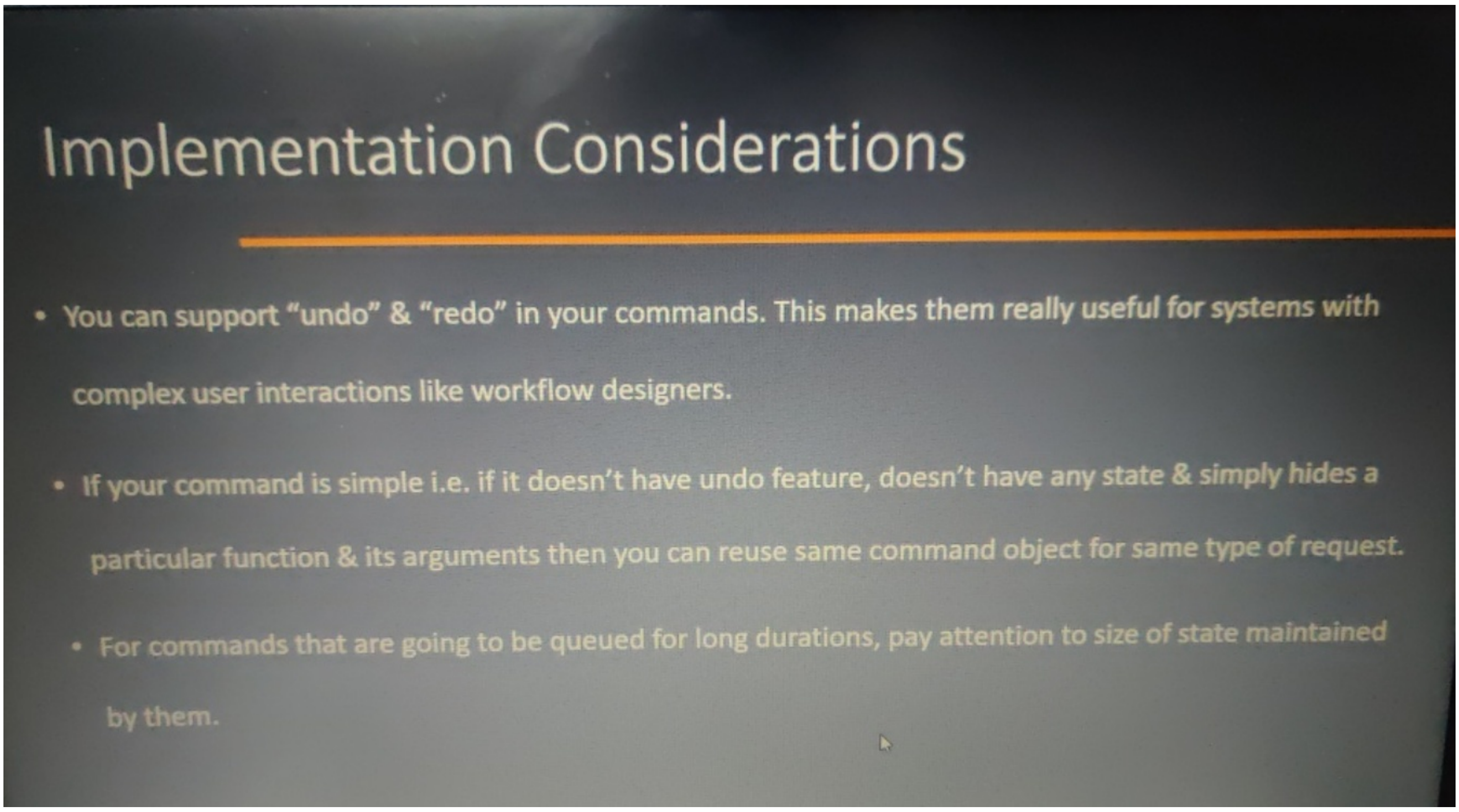
Implement Command

- We start by writing command interface
  - It must define method which executes the command
- We next implement this interface in class for each request or operation type we want to implement. Command should also allow for undo operation if your system needs it.
- Each concrete command knows exactly which operation it needs. All it needs is parameters for the operation if required and the receiver instance on which operation is invoked.
- Client creates the command instance and sets up receiver and all required parameters.
- The command instance is then ready to be sent to other parts of code. Invoker is the code that actually uses command instance and invokes the execute on the command.

Example: UML

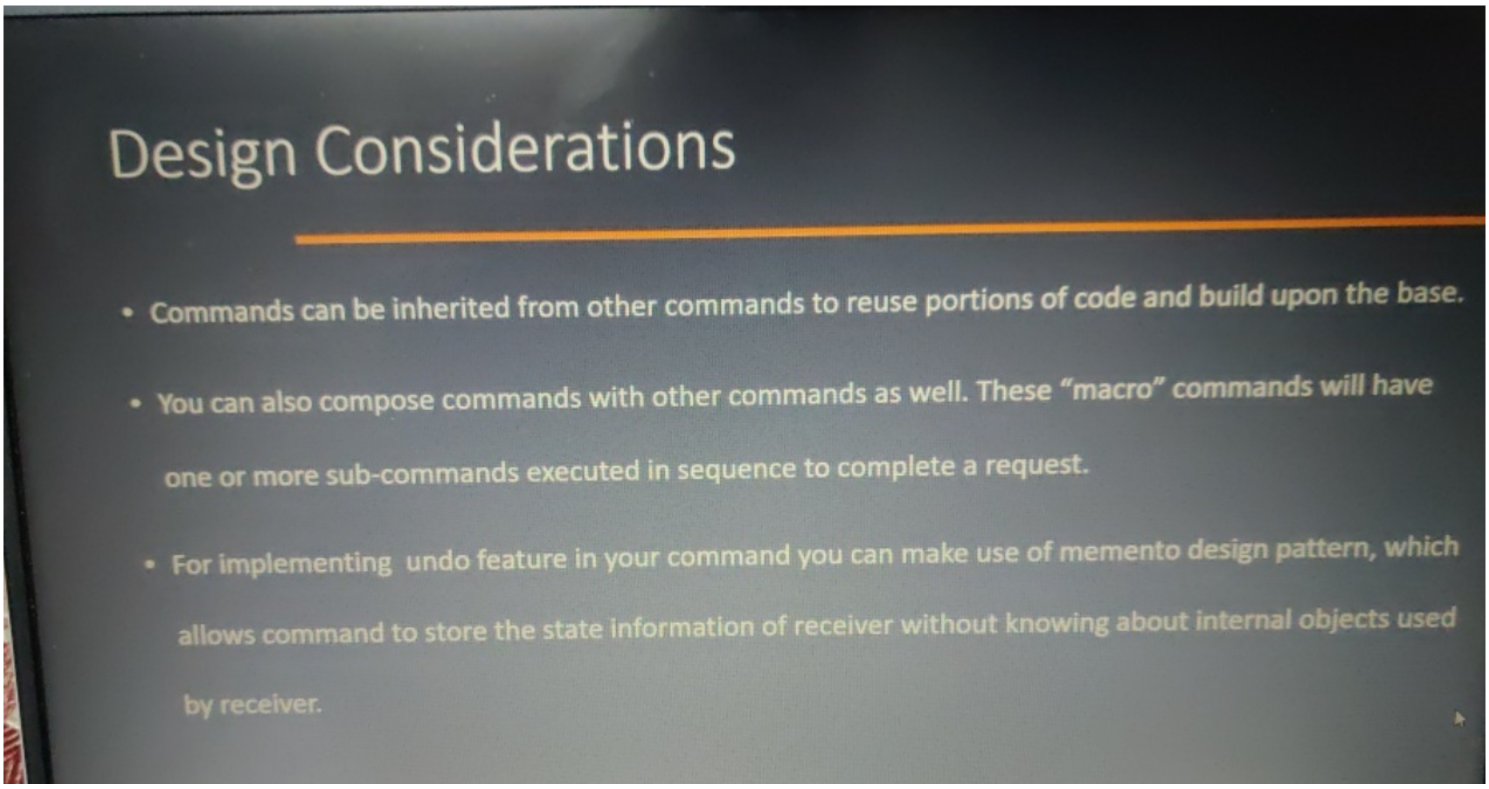


class CommandEx
«invoker»
BackgroundRunner
+ submit(command): void
«interface»
Command
+ execute(): void
Client
SendCommand
- mailingList
- service
- content
+ execute(): void
AddMemberCommand
- emailAddress
- service
+ execute(): void
RemoveMemberCommand
- emailAddress
- service
+ execute(): void
«receiver»
EWSService
+ addMember(): void
+ sendEmail(): void
+ removeMember(): void
+ sendForLater(): void



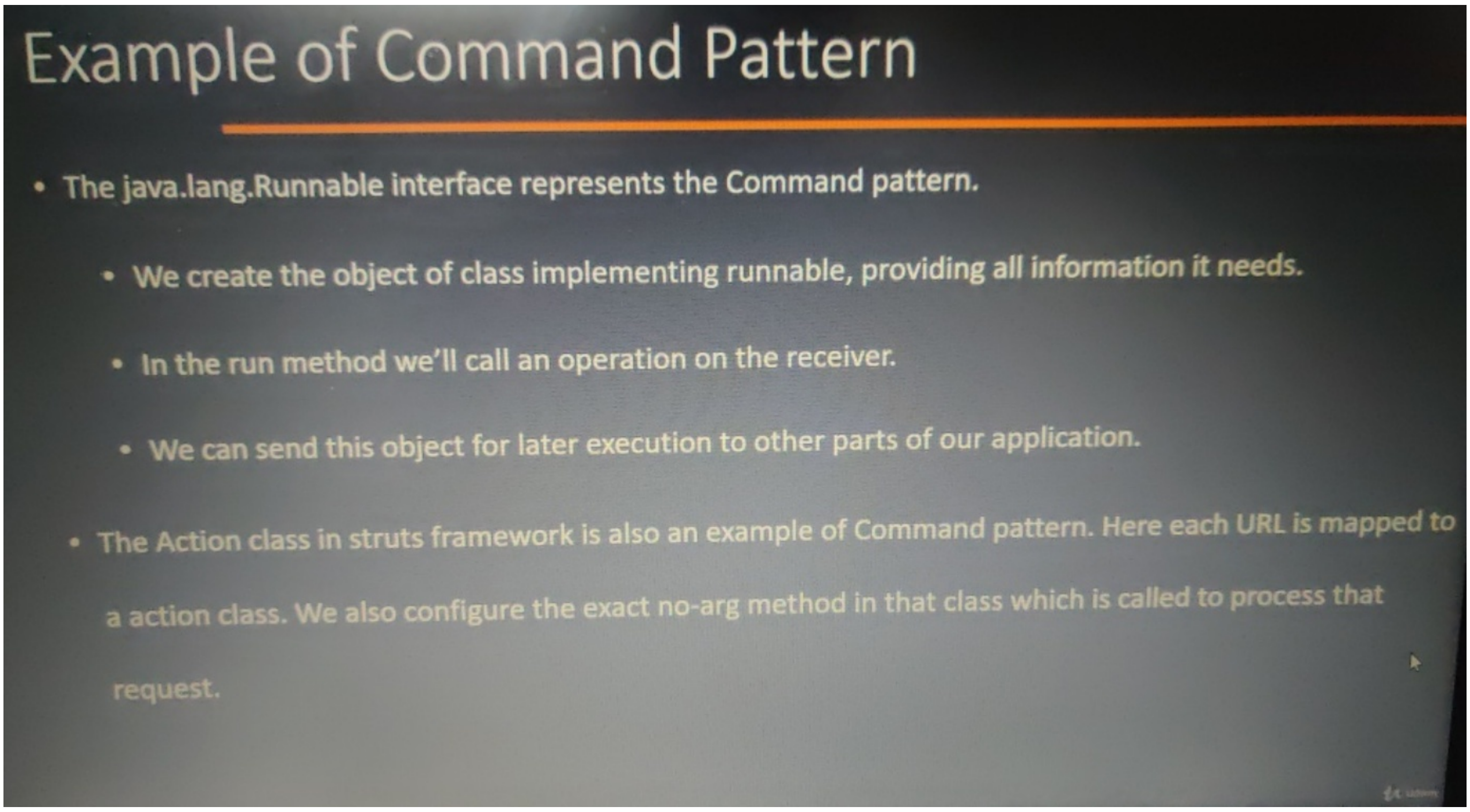
Implementation Considerations

- You can support “undo” & “redo” in your commands. This makes them really useful for systems with complex user interactions like workflow designers.
  - If your command is simple i.e. if it doesn’t have undo feature, doesn’t have any state & simply hides a particular function & its arguments then you can reuse same command object for same type of request.
    - For commands that are going to be queued for long durations, pay attention to size of state maintained by them.



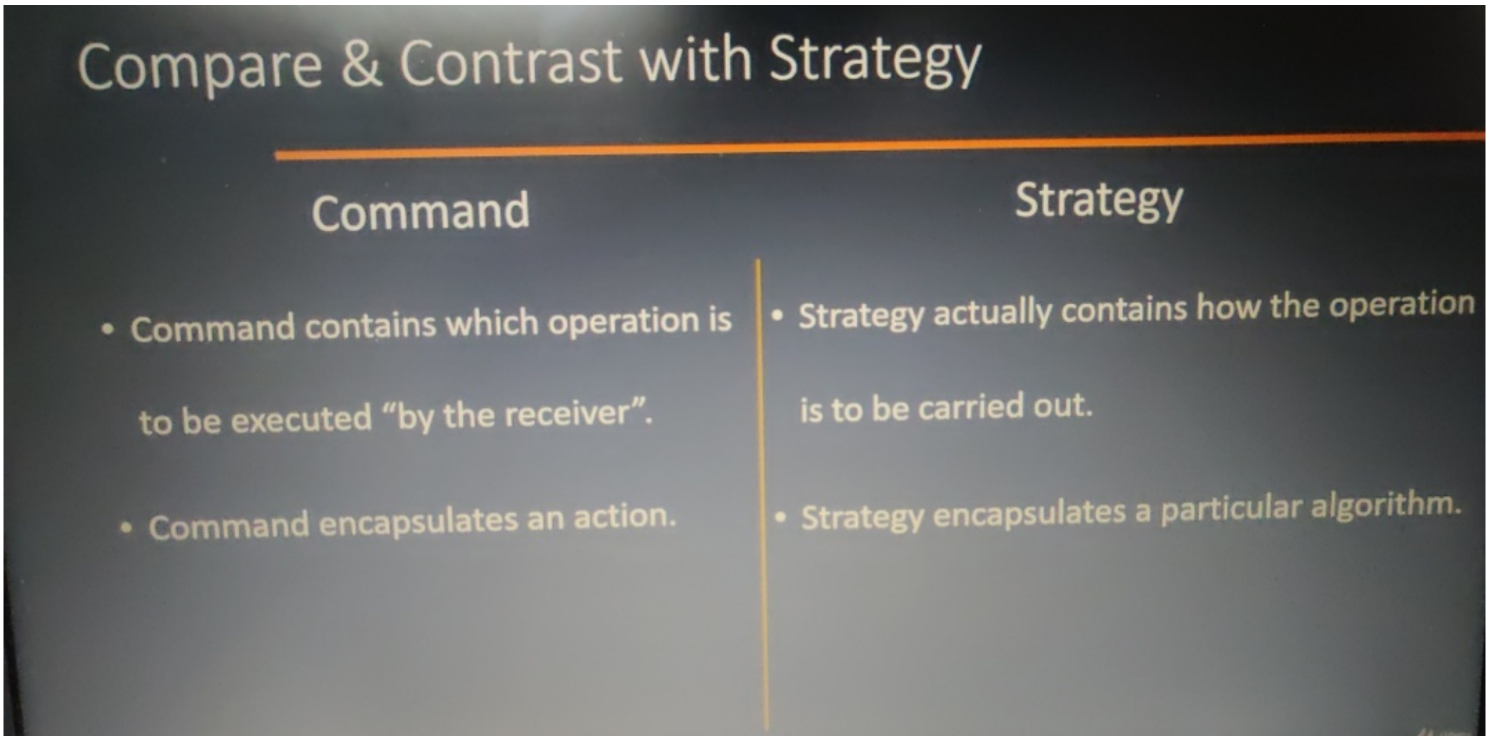
Design Considerations

- Commands can be inherited from other commands to reuse portions of code and build upon the base.
- You can also compose commands with other commands as well. These "macro" commands will have one or more sub-commands executed in sequence to complete a request.
- For implementing undo feature in your command you can make use of memento design pattern, which allows command to store the state information of receiver without knowing about internal objects used by receiver.



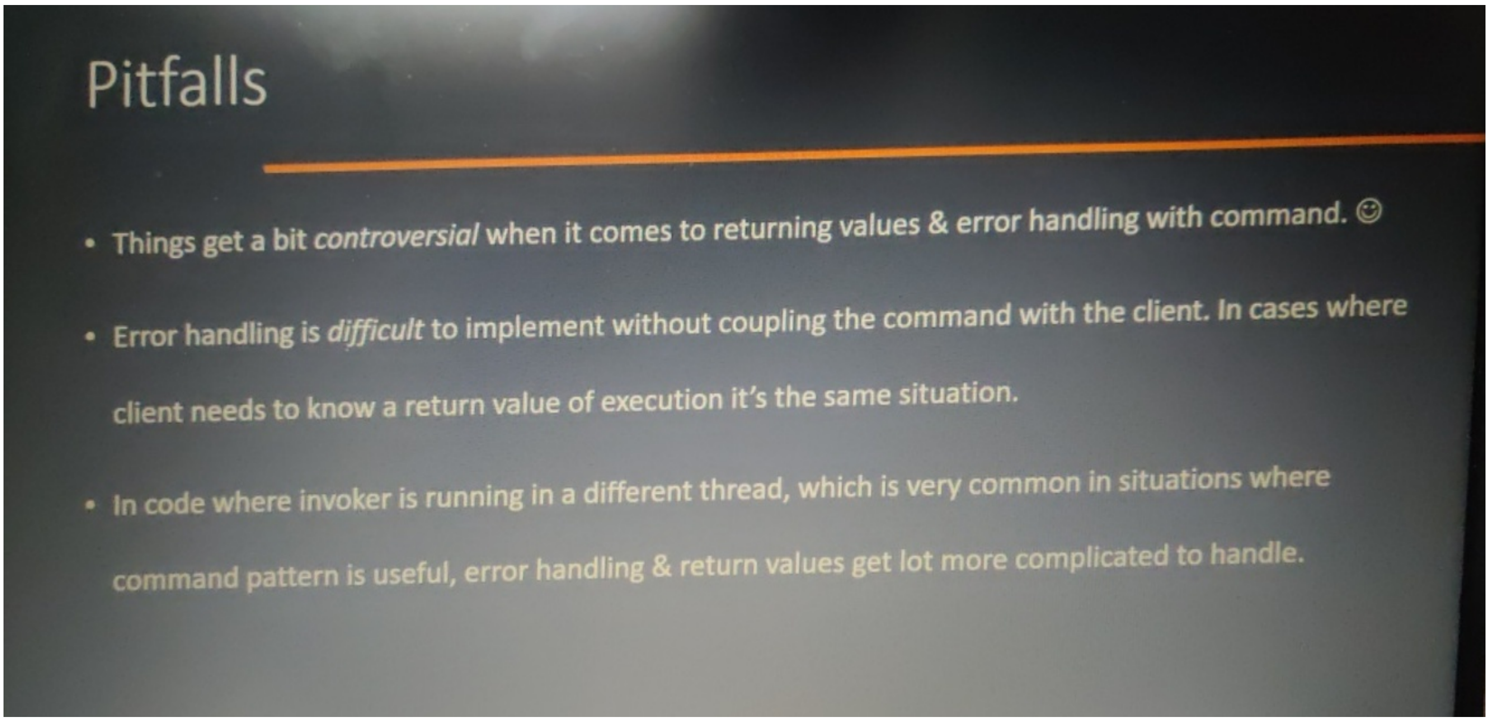
Example of Command Pattern

- The java.lang.Runnable interface represents the Command pattern.
  - We create the object of class implementing runnable, providing all information it needs.
  - In the run method we'll call an operation on the receiver.
  - We can send this object for later execution to other parts of our application.
- The Action class in struts framework is also an example of Command pattern. Here each URL is mapped to a action class. We also configure the exact no-arg method in that class which is called to process that request.



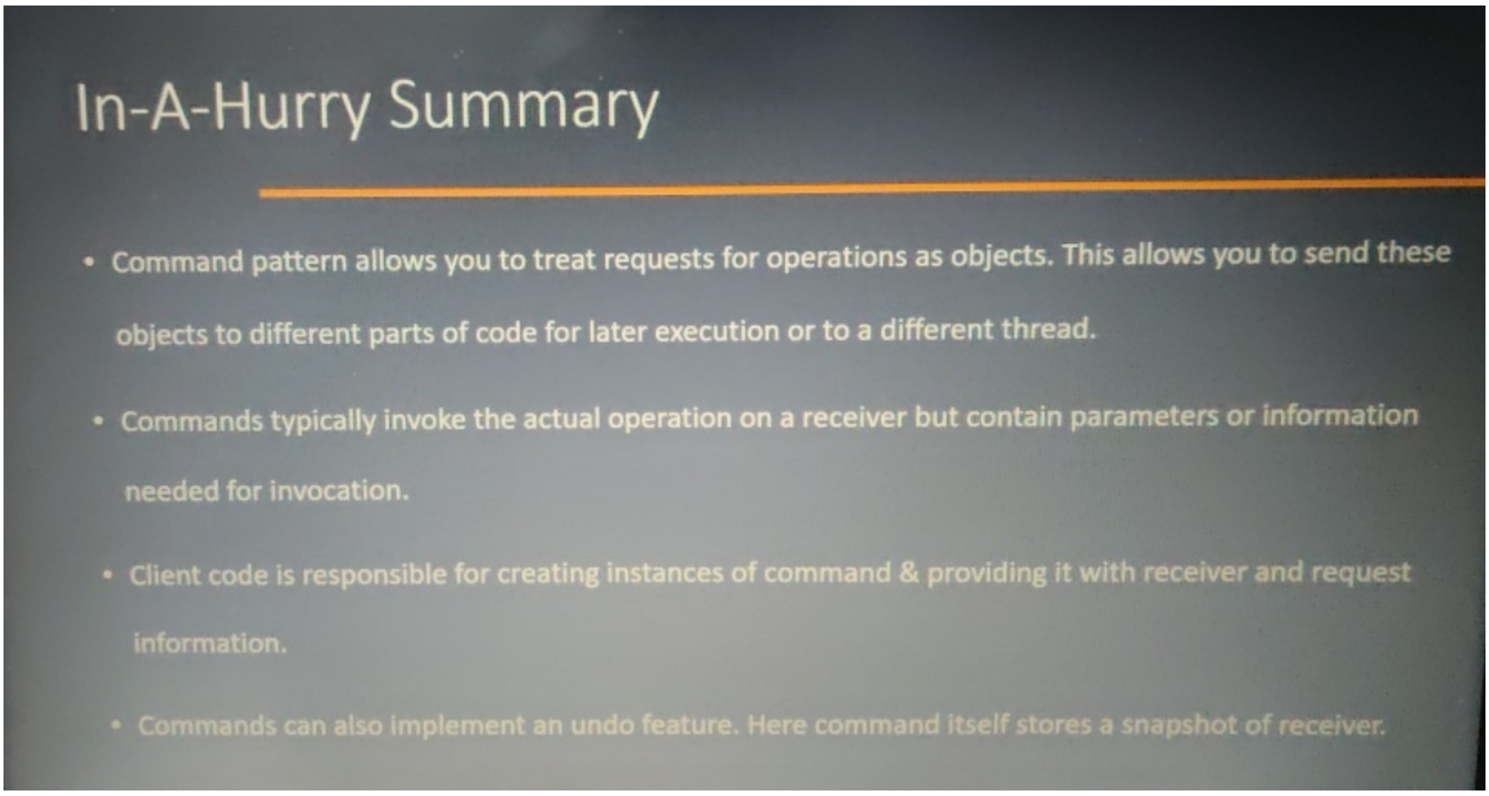
Compare & Contrast with Strategy

Command	Strategy
• Command contains which operation is to be executed "by the receiver".	• Strategy actually contains how the operation is to be carried out.
• Command encapsulates an action.	• Strategy encapsulates a particular algorithm.



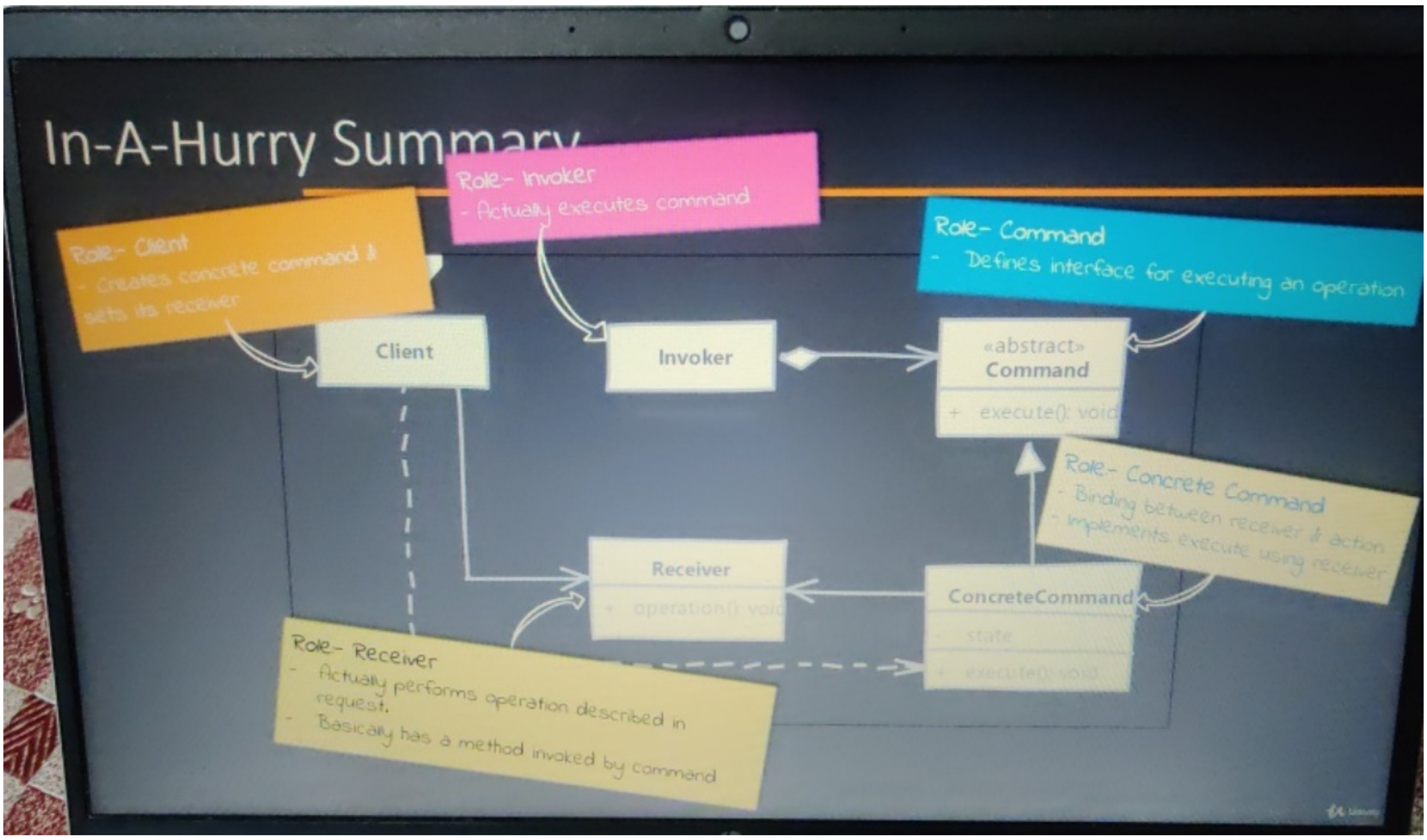
Pitfalls

- Things get a bit *controversial* when it comes to returning values & error handling with command. ☺
- Error handling is *difficult* to implement without coupling the command with the client. In cases where client needs to know a return value of execution it's the same situation.
- In code where invoker is running in a different thread, which is very common in situations where command pattern is useful, error handling & return values get lot more complicated to handle.



In-A-Hurry Summary

- Command pattern allows you to treat requests for operations as objects. This allows you to send these objects to different parts of code for later execution or to a different thread.
- Commands typically invoke the actual operation on a receiver but contain parameters or information needed for invocation.
- Client code is responsible for creating instances of command & providing it with receiver and request information.
- Commands can also implement an undo feature. Here command itself stores a snapshot of receiver.



In-A-Hurry Summary

Role- Client
- Creates concrete command & sets its receiver

Role- Invoker
- Actually executes command

Role- Command
- Defines interface for executing an operation

Role- Concrete Command
- Binding between receiver & action
- Implements execute using receiver

Role- Receiver
- Actually performs operation described in request.
- Basically has a method invoked by command