# Overview:

* The command design pattern encapsulates a request as an object (command).
  + Let you parametrize clients with different requests.
    - You can queue or log these requests as well as supporting “undoing” these requests.
* The Command object encapsulates a request by binding together a set of actions on a specific receiver:
  + Does this by exposing just one method execute () that causes some actions to be invoked on the receiver.
* The Command’s execute method can store state for reversing its effects of the Command itself thus supporting “undo” operation.
  + Alternatively, you can even add an unExecute method that reverses the effects of a previous call to execute.
* The pattern can also support logging changes so that they can be reapplied in case of a system crash.
* This pattern is a data driven design pattern.
* It is widely used for undo/redo operations.
* A callback function can be designed with this pattern.
* It is very useful when we model transactions (which can be responsible for changes in data).
* Makes our code extensible as we can add new commands without changing existing code.
* This pattern allows us to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

# Examples:

* We could use the command pattern when placing orders to buy and sell stocks.
  + We would create an interface Order which acts as a command (buying and selling as concrete classes).
  + A Stock class could act as the request.
  + A class Broker could act as an invoker object.
    - Takes and places orders.
* The broker object uses command pattern to identify which object will execute which command based on the type of command.
* Another example could be a user interface toolkit.
  + Includes objects like buttons and menus.
    - Carry out requests in response to user input.
    - The toolkit cannot implement the request explicitly in the button or menu.
      * Only applications that use the toolkit know what should be done on which object.
      * No way of knowing the receiver of the request or the operations, that will carry it out.
* The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object
  + Object can be stored and passed around like other objects.

# Examples (Undo/Redo):

* We can undo and redo many operations in our daily life:
  + We can erase a pencil drawing with a rubber.
  + We can redesign our living places.
  + We can forget bad memories and start a fresh journey.
* Undo/redo operations are part of our life:
  + We are doing these operations through some commands:
    - Either externally or internally.
* The above scenario also applies with the Microsoft paint application:
  + We can do the undo/redo operations easily through some menu options or shortcut keys.

# When to use the command pattern?

* When you want to parametrize objects by an action to perform.
* When you want to specify, queue, and execute requests at different times.
  + A Command object can have a lifetime independent of the original request.
* When you want to support undo
  + The Command’s execute method can store state for reversing its effects in the command itself.
* When you want to support logging changes so that they can be reapplied in case of a system crash.
  + Can augment the Command interface with load and store operations.
    - Keep a persistent log of changes.
* When you want to implement a callback method.

# Implementation:

# Overview:

* Remember, requests are encapsulated as objects.
* Four terms are associated with the implementation of this pattern:
  + Invoker, client, command, and receiver.
* A command object is capable of calling a particular method in the receiver.
* An invoker only knows about the command interface.
  + It is totally unware about the concrete commands.
* The client object holds the invoker object and the command objects
  + The client decides which of these commands needs to execute at a particular point in time.

# Participants:

* **Command**:
  + Declares an interface for all commands.
  + A command is invoked through it’s execute method.
    - Will ask a receiver to perform an action.
  + May include an undo() method.
* **ConcreteCommand**:
  + Defines a binding between a Receiver object and an action.
  + Invoker makes a request by calling execute () and this class carries it out by calling one or more actions on the Receiver.
* **Client**:
  + Creates a ConcreteCommand object and sets its receiver.
* **Invoker**:
  + Asks the command to carry out the request by calling it’s execute () method.
* **Receiver**:
  + Knowns how to perform the work need to carry out a request.
  + Any class may serve as a Receiver.

# Workflow:

* The client creates a ConcreteCommand object and specifies its receiver.
  + Creates the receiver object as well and then attaches it to the Command (in the constructor)
* An Invoker object stores the ConcreteCommand object:
  + Client creates the invoker object and attaches the command object (in the constructor):
    - May also instead just pass the command to the execute method.
* The invoker issues a request by calling execute, using the command object (previously attached or passed in as a parameter to execute).
  + When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking execute.
* When client program executes the action (via the invoker), it is processed based on the command and receiver object.
  + The ConcreteCommand object invokes the appropriate method on its receiver to carry out the specific request.

# Implementation details:

* The Command decouples the object that invokes the operation from the one the knows how to perform it.
* **Commands** are first-class objects.
  + Can be manipulated and extended like any other object.
* **Command** objects can be:
  + **Dumb**: delegates the required action to a receiver object.
  + **Smart**: implements everything itself without delegating to a receiver object.
* Pattern is easily extendible:
  + We can add new action methods in receivers and create new Command implementations without changing the client code.
* One drawback is that the code gets huge and confusing with a high number of action methods
  + You end up with many small classes.

# Summary:

* The command object is the core of command design pattern that defines the contract for implementation.
* Receiver implementation is separate from command implementation.
* Command implementation classes choose the method to invoke on receiver object.
  + For every method in receiver, there will be a command implementation.
    - Works as a bridge between receiver and action methods.
* Invoker class just forwards the request from client to the command object.
* Client is responsible for instantiating appropriate command and receiver objects and then associating them together.
* Client is also responsible for instantiating invoker object and associating command object with it and executing the action method.

# Summary

* The Command design pattern decouples an object making a request from the one that knows how to perform it.
* A Command object is at the center of this decoupling and encapsulates a receiver with an action (set of actions).
* Commands may support undo by implementing an undo method that restores the object to its previous state.
* Commands may also be used to implement logging and transactional systems.
* Commands can be extended easily
  + While we use them, we do not need to change the classes in the system.
* In the chain of responsibility pattern, we forward a request along a chain of objects with the hope that any one of the objects along that chain will handle the request.
* In the command pattern, we will forward the request to a specific object.