# Overview:

* The chain of responsibility pattern avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
  + Chain the receiving objects and pass the request along the chain until an object handles it.
* This pattern processes a series of objects one by one (in a sequential manner).
  + A source will initiate this processing.
* Let’s you send requests to an object implicitly through a chain of candidate objects
  + After one’s processing is done, if anything is still pending, it can be forwarded to the next object in the chain.
    - Each receiver contains reference to another receiver.
  + We can add new objects anytime (run-time) at the end of a chain.

# Examples:

* In an organization, there are some customer care executives who handle feedback/issues from customers:
  + They will forward those customer issues/escalations to the appropriate department in the organization.
  + Not all departments will start fixing an issue.
  + The department that seems to be responsible will look first, and if the department staff believe that the issue should be forwarded to another department, he/she will do that.
* Several class libraries use the **Chain of Responsibility** pattern to handle user events.
  + When the user clicks the mouse or presses a key, an event is generated and passed along the chain.
* Consider a context-sensitive help facility for a graphical user interface.
  + User can obtain help information on any part of the interface just by clicking on it.
  + The help that is provided depends on the part of the interface that is selected and its context.
    - A button widget in a dialog box might have different help information than similar button in the main window.

# JDK Examples:

* An Example in the JDK would be the use of the try/catch blocks:
  + Every catch block is kind of a processor to process that particular exception.
  + When any exception occurs in the try block
    - Sends to the first catch block to process.
    - If the catch block is not able to process it, it forwards the request to next object in chain (next catch block).
    - If even the last cat block is not able to process it, the exception is thrown outside of the chain to the calling program.
* Other packages/methods that use this pattern in the JDK include:
  + Java.util.logging.Logger#log ().
  + Javax.servlet.Filter#doFilter ().

# When to use this Pattern?

* When you want to decouple a request’s sender and receiver.
* When multiple objects, determined at runtime, are candidates to handle a request.
* When you do not want to specify handlers explicitly in your code.
* When you want to issue a request to one of several objects without specifying the receiver explicitly.
  + We expect any of our receivers to handle that request.
* When multiple objects can handle a request and the handler does not have to be a specific object.

# Advantages and drawbacks:

* Decouples the sender of the request and its receivers:
  + Frees an object from knowing which other object handles a request.
  + Both the receiver and the sender have no explicit knowledge of each other.
* Simplifies your object:
  + It does not have to know the chain’s structure or keep direct references to its members.
  + Keeps a single reference to their successor.
* Gives you added flexibility in distributing responsibilities among objects.
  + Allows you to add or remove responsibilities dynamically by changing the members or order the chain.
* A drawback is that the execution of the request is not guaranteed.
  + May fall of the end of the chain if no object handles it.
* Another drawback is that it can be hard to observe and debug at runtime.

# Implementation

# Participants:

* **Handler:**
  + Defines an interface for handling requests.
  + (Optional) implements the successor link.
    - Dispatches the request to chain of handlers.
* **ConcreteHandler:**
  + Handles requests it is responsible for.
  + Can access its successor.
  + If the ConcreteHandler can handle the request, it does so; otherwise, it forwards the request to its successor.
* **Client:**
  + Initiates the requests to a ConcreteHandler object on the chain.
  + When a client issues a request, the request propagates along the chain until a **ConcreteHandler** object takes responsibility for handling it.

# Important implementation considerations:

* Client does not know which part of the chain will be processing the request
  + It will send the request to the first object in the chain.
* Every object in the chain should have a reference to the next object in chain to forward the request to:
  + This is achieved by composition.
* Creating the chain carefully is very important:
  + There might be a case that the request will never be forwarded to particular processor or there are no objects in the chain who are able to handle the request.
    - Add a check to make sur it is processed fully by all the processors.
    - On the other hand, throw an exception if the request reaches the last object and there are no further objects in the chain to forward the request to.

# Summary:

* With the **Chain of Responsibility** Pattern, you create a chain of objects to examine requests.
* Each object in turn examines a request and either handles it, or passes it on to the next object in the chain.
* Each object in the chain acts as a handler and has a successor object.