Chain Processing Framework

What is it?

It’s a small library, responsible for managing order of processes execution. It’s a mix for “Chain of responsibility“ and “Composite command” design patterns implemented already in built in classes.

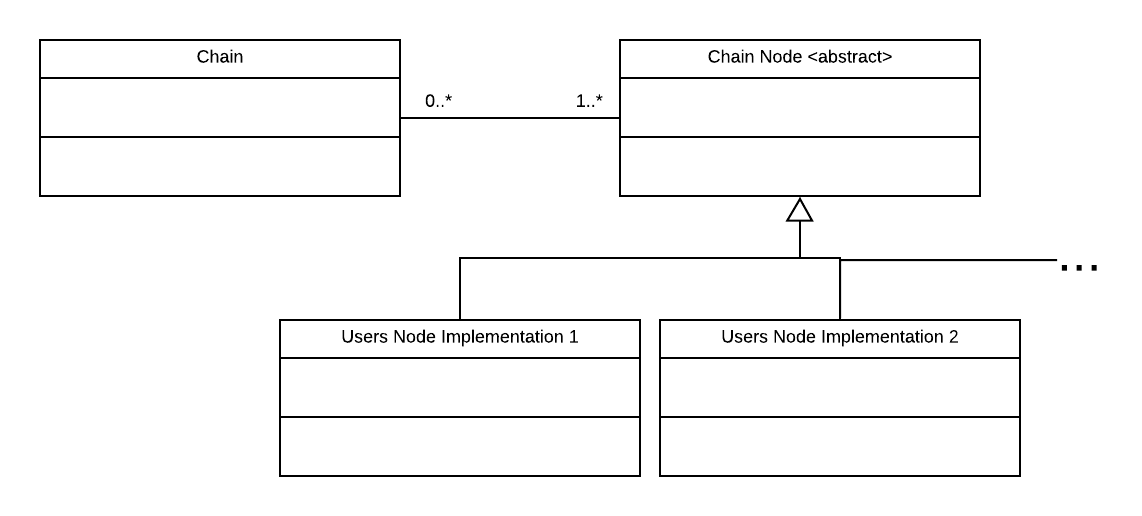
When is it useful?

It becomes useful in situations, when programmer has to implement:

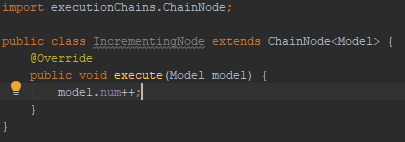
1. long, complex procedure, that consists of many steps which should be divided into smaller classes by their responsibilities
2. algorithm, consisting of many steps, executed in complicated order

How is it useful?

It provides programmer with set of classes, that help managing order of methods execution, doing all of the dirty work. User can create so called “Chain Nodes” and put them into “Chain”, in order to use “Chains” methods and organize the processing. Here is the structure of solution, using framework:

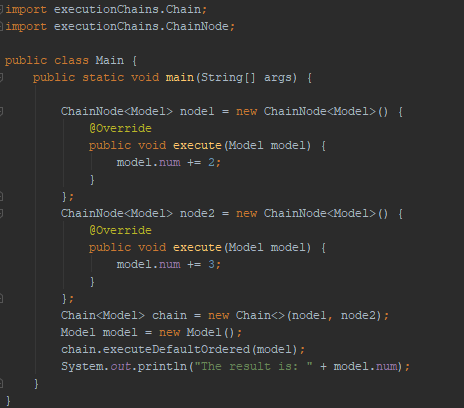


We implement “Chain Node” abstract class in our customized way, by implementing void method “execute(Model model)”. “Chain Node” is a generic class with Type parameter “Model”, so method “execute” can be called with any type of parameter we want. Example implementation of “Chain Node” with it’s model:



This example is very simple, and obviously using this classes just to increment number would be abusing this framework, but you can imagine, that such nodes could be much more complicated. For example, they could contain complex database query calls or some complicated business logic.

Let’s stick to the simple things first though, and see, how these nodes work in a “Chain”. For this purpose, we will implement “Chain Nodes” as anonymus classes.

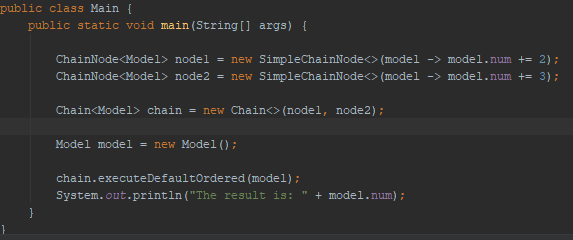


Program prints “The result is: 5”. What happened here, step by step, is:

1. Two nodes have been created.
2. They have been passed to the constructor of “Chain” when creating new “Chain”
3. Chain has been executed in a default order. It means, that nodes are being executed in the same order, as they were added. So in this case, node1 has been executed before node2.

Faster ways of creating “Chain Nodes”

Unfortunately, we can’t create chain nodes using lambdas, because they aren’t functional interfaces. They provide different, useful methods that will be discussed later. There is, however, a way to create them faster. If we don’t want to use any of “Chain Node” built in methods, we can create “Simple Chain Node”, which extends “Chain Node”. “Simple Chain Node” can take a lambda as a constructor parameter (which is implementation of „Chain Function“ functional interface). Might sound complicated, but it’s actually very simple. Here is the program shown above, but with “Chain Nodes” implemented as “Simple Chain Nodes”:



In this way we save a bit of work, but we can’t use full potential of “Chain Nodes”. In general, when using “Simple Chain Nodes”, it’s probably easier not to use this framework, but they might be useful when complexity of nodes are different and order of execution is not that simple.

“Chain Nodes” Functions

Now, let’s discuss these functions implemented in “Chain Nodes”. They all communicate with “Chain” (maybe not exactly with chain, but for now we can assume it’s chain) they are linked with, so if we will just execute this functions outside of “Chains” process, they will throw some Nullpointers!

That’s why we should invoke these functions only inside of “execute” implementation of our “Chain Node”.

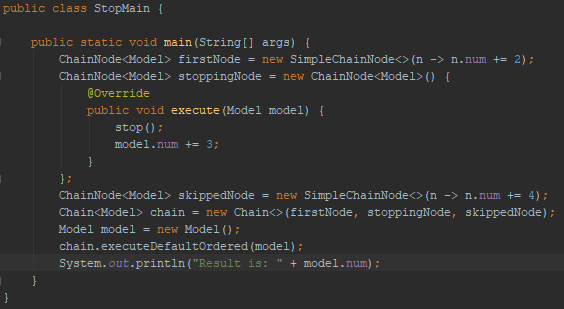
Now, let’s describe, what they do:

1. void stop() – makes the chain stop right after execution of node in which it was invoked. Tip: Adding „return;” after it will cause the chain to stop immediately.
2. void goTo(int index) – makes chain proceed to the node with selected index after execution of node in which it was invoked. It throws “ArrayIndexOutOfBoundsException” if the index does not exist in the chain. Indexing if based on order of nodes pushed to the chain, so first node added will have index “0”, second will have “1” and so on…
3. void goTo(ChainNode node) – makes chain proceed to selected node after execution of node on which it was invoked
4. void restart() – makes chain move to the first node after finishing execution of node on which it was invoked
5. void skipToEnd() – makes chain move to end after finishing execution of node on which it was invoked. It means, that in default execution mode the chain will simply end. The difference between stop() is that for example in loop() mode it will simply skip all the remaining nodes and try to execute chain again
6. void skipNode(ChainNode node) – makes chain skip given node when the chain will be trying to invoke it. The node will be skipped every time it’s tried to invoke from now.
7. void skipNodes(List<ChainNode> chainNodes) – invokes skipNode() on all nodes from the list

These functions are giving user a lot of freedom in designing the chain. User must be aware though, that they will do a bit different things, depending on how we execute chain. This will be explained in more details later. On some execution modes, several functions will do exactly the same thing, like for example in default ordered mode both “skipToEnd()” and “stop()” will have the same effect.

Advantages of using these functions are that they are very intuitive and they help implementing chain fast. However, they might be problematic, when it comes to flexibility and independence of nodes. There is an alternative to these functions – navigations, that will be used later.

Example “stop()” usage:



Program prints “The result is: 5”. What’s happening is:

1. “firstNode” is being executed – number in model increases and becomes “2”
2. “stoppingNode” is being executed and so
3. “stop()” is being invoked – it blocks any other remaining nodes from being executed
4. number in model is being increased anyway and it becomes “5”
5. “skippedNode” isn’t executed anymore, because it was stopped by the “stoppingNode”. Number in model is not increased by 4, so it remains “5” until the and

“Chain” building

Building chain means adding and removing elements like “Chain Nodes” to it, and “Chain” provides user with several constructors and methods that help with it:

Constructors:

1. Chain(ChainNode<Model>… nodes) – builds chain from nodes, in the order given. Chain may be later modified.
2. Chain (List<ChainNode<Model>> nodes) – builds chain from list of nodes, in the order given. Chain may be later modified.
3. Chain() – builds chain and creates inside empty list of nodes, that may be later modified

Chain modifying methods

1. pushNode(ChainNode<Model> node) – adds new node to the chains node list
2. insertNode(ChainNode<Model> node, int index) – adds new node to the chains list on specified position
3. removeNode(ChainNode<Model> node) – removes node from the chain

“Chain” execution modes

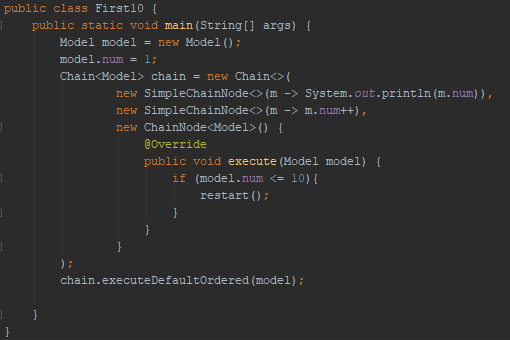
Chains can be executed in a several ways, by calling different methods on it. These are:

1. executeDefaultOrdered(Model model) – simply makes chain running from beginning to end
2. executeWhile(Model model, Predicate<Model> condition) – does exactly the same thing as above, but before execution of each node, the condition is being checked. If it’s not satisfied, chain stops
3. loop(Model model) – makes chain running over and over – from beginning to end. May be stopped only by invoking “stop()”
4. loopWhile(Model model, Predicate<Model> condition) – does exactly the same as above, but at every loop beginning the condition is checked. If it’s not satisfied, chain stops
5. loopNTimes(Model model, int times) – executes all nodes in order n times

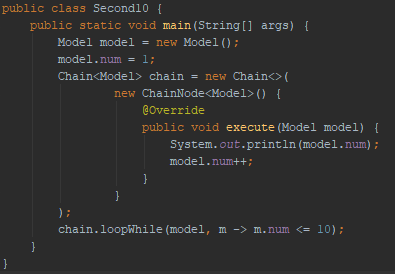
Combining execution methods with node methods

Node methods and chain methods cope with each other in a different, but predictable ways. Here are 3 methods (and not the only ones) of implementing counting from 1 to 10.

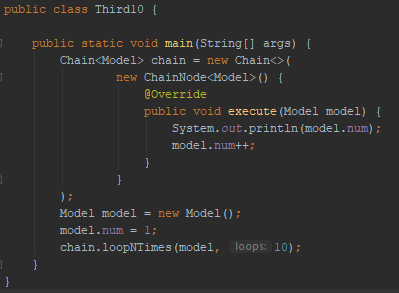
Fist approach is probably the least appropriate. Instead of loosing loop, we are restarting the chain manually from the last node, until it has printed “10”. It works, but in this framework we have better (more readable) ways to do it.



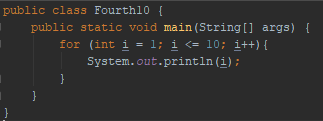
In second example, we are using “loopWhile()”. Instead of creating 3 nodes now, we created only one. We could have divided printing and incrementing into tow nodes, but there is no need. Loop is printing numbers until “10”, because of the predicate “m -> m.num <= 10”. If number exceeds “10”, predicate returns false and loop finishes. This approach is much better than previous one, but still we can do better.



In third example, we are doing something very similar, but instead of “loopWhile” we use “loopNTimes”. It works in a very similar way, but here instead of saying until when should it print, we are saying how many times it should print.

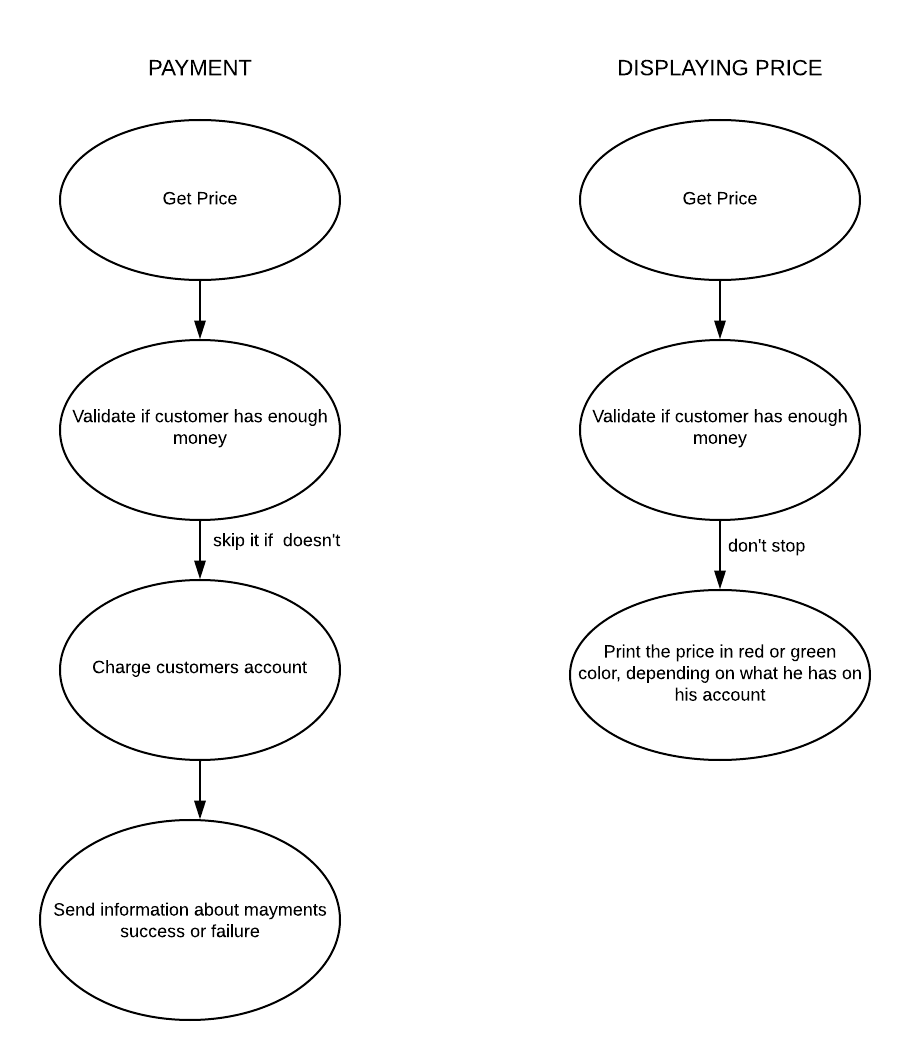


These were just examples of how the framework works. Of course, using good old for loop is much better in this case, and in examples above we are abusing chains.



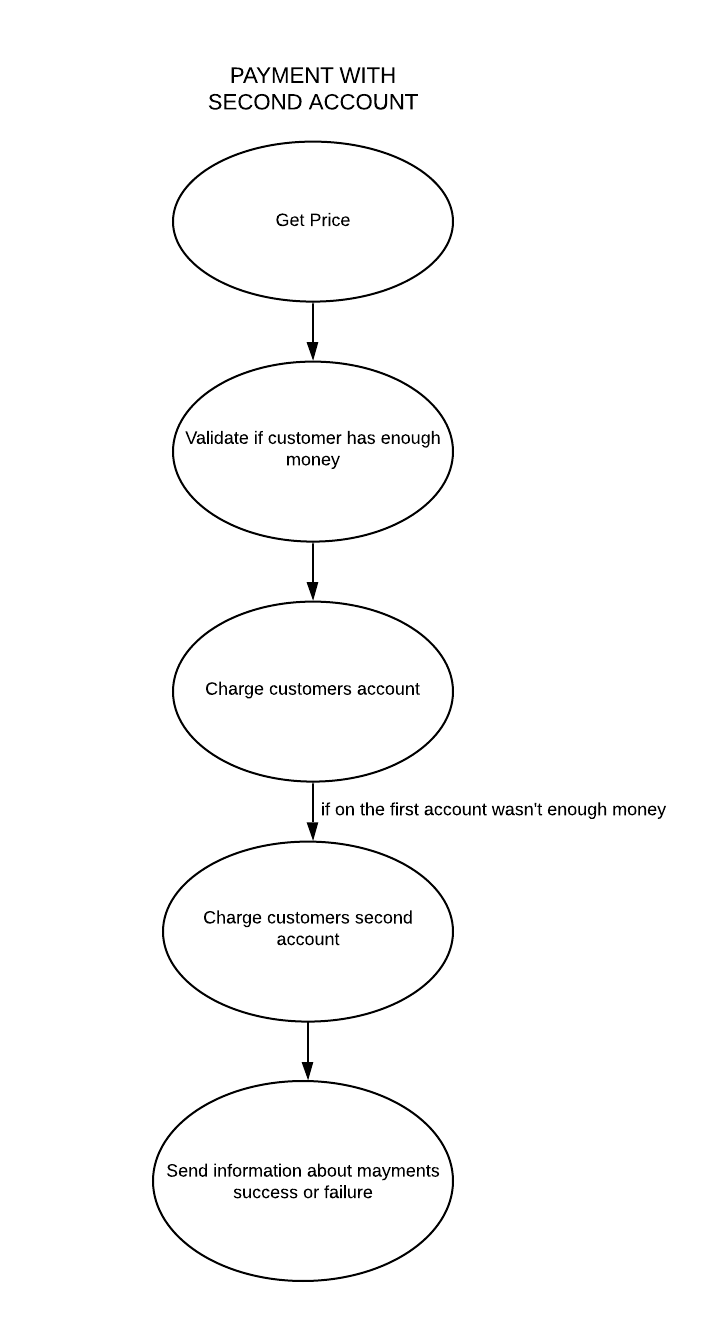
Navigations – chain nodes flexibility problem

Let’s imagine following situation:



We have 2 chains using the same nodes – “Get Price” and “Validate, if customer has enough money”. There is a problem in second one. In “PAYMENT” chain, we don’t charge customers account if customer doesn’t have enough money – we skip one node, order is being changed. In the second one, we don’t skip anything and chain goes on. What we could do in this situation is:

1. use “skipNode(Charging node)” method in validating payment node. This would work for this case, but what if we had third chain like this:



Here, we assume that customer has second account that we can always charge if he doesn’t have enough money. So in this case, we can’t skip charging account node anymore.

1. Another solution would be to implement validating node twice – once with skipping and once without it. This is bad, because we might have another chain that will need even different behaviour.
2. We can use navigations.

Navigations – idea

In order to avoid the problem discussed above, we can separate ordering nodes from nodes themselves and encapsulate ordering in navigations. It means, we will leave validating node without any chain functions and add some properties to a particular chain. Navigations are the reason, why using this framework is better that implementing “Chain of responsibility”. Here, we can separate ordering process from business logic of nodes.

Navigations – implementation

Class “Navigation” is a generic type (Navigation<Model>) and has two similar constructors:

1. Navigation(ChainNode<Model> from, ChainNode<Model> to)
2. Navigation(ChainNode<Model> from, ChainNode<Model> to, Predicate<Model> condition)

Both of these have “from” and “to” node. When we apply navigation to the chain, every time after executing “from” chain, navigation will proceed to “to” node. This happens in case we use first constructor. Second constructor gives us a bit more control. We pass here also a predicate which will be checked every time, when navigation is trying to move chain to “to”. If predicate will return false, navigation will fail and default order from chain processing mode will be used.

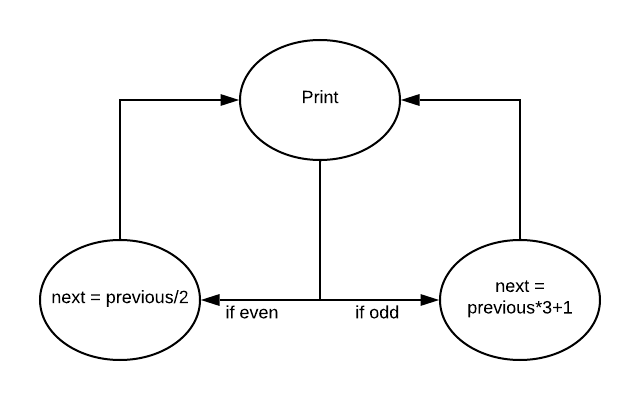
Navigations – collatz example (how it can be used in algorithms)

Let’s see, a collatz sequence example. In this case, framework will actually improve readability of the code and save programmer a bit of struggle.

Our job is to simply print over and over next terms of collatz sequence starting from number we choose (let’s say “500”). Generating next term of collatz sequence is based on the previous one. If previous is even, our next term is equal to “previous/2” and if it’s odd, next term is equal to “3\*previous + 1”. We will divide generating each term into 3 different nodes:

1. Print current term
2. Do n/2 (when printed term was even)
3. Do 3n+1 (when printed term was odd)

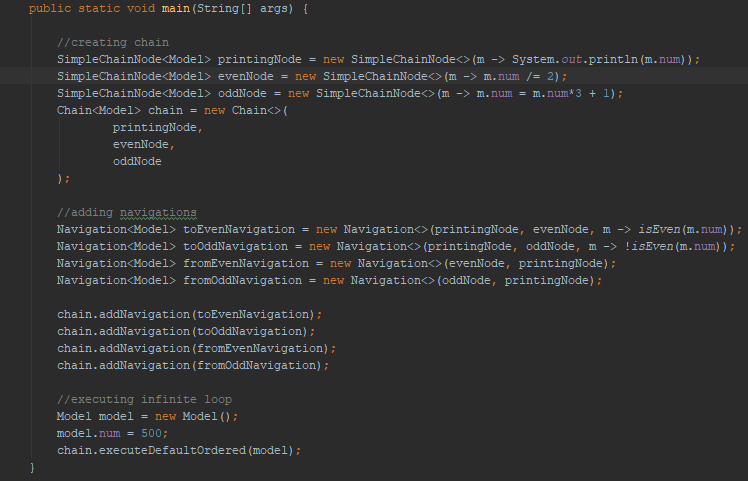
Now we won’t use loops or any chain node functions, we will use only navigations on these 3 steps. We will be printing current term, then generating next one using second and third node and then go back to first, printing node. And repeat.



Let’s see implementation. We will be using “isEven(int num)” function:



And here we go:



This is one way of implementing it. Not the shortest way, but it requires from programmer less thinking about what can go wrong, etc.