How to run demo:

- https://github.com/KrakenMInitials/SimplifiedPAXOS
- For Windows users, demo.bat runs a configuration of a coordinator/distinguished learner + 4 proposers + 7 acceptors
- For other users, run the following while in the root of the repository (go run node.go <node_id> <1 for proposer | 2 for acceptor>) **coordinator is necessary**:
    - go run .\cordinator\coordinator.go
    - go run node.go 1 1
    - go run node.go 2 1
    - go run node.go 3 1
    - go run node.go 4 1
    - go run node.go 5 2
    - go run node.go 6 2
    - go run node.go 7 2
    - go run node.go 8 2
    - go run node.go 9 2
    - go run node.go 10 2
    - go run node.go 11 2
- OR spawn nodes as desired, but shared.go needs modifications in
    - AddressRegistry
    - Known_acceptors
    - Known_proposers

Simplified PAXOS Design Decisions

- My implementation of PAXOS is designed to be discrete, operating in rounds of consensus triggered by a coordinator.
- Overview: A Coordinator triggers a consensus among proposers and gives them an upperbound integer with consensus to be reached on a random integer below the bound. A number of proposer nodes generate their own random values below the upper-bound and begin PAXOS to ultimately produce and reach consensus on a single value within the upperbound integer.
- Node datatype & States

```
type Node struct { //stores Node state
    ID           int
    Class        shared.Class
    NodeListener net.Listener
    PeerClients  map[int]*rpc.Client //stores id to rpc.client instances for known peers

    // dynamic states
    proposedPrpslNum int
    highestPrpslNum  int
    knownVal         *int //has to be pointer to be nil-checkable
    complete         bool //current implementation only uses for acceptors to not repeat accepts for same round

    ConsensusRound int
}
```

The Nodes are state-based and can switch between classes by configuration. The following states are initialized:

    - ID: the unique identifier of the node
    - Class: acceptor | proposer | listener (listeners are not present in implemented scope; used distinguisher learner)
    - NodeListener: caches the main and only Listener for the node to receive requests
    - PeerClients: caches rpc Clients for connected peer nodes
    - DYNAMIC STATES: the states tracked for PAXOS logic and phases
        - proposedPrpslNum
            - Proposers to keep track of what proposal number they proposed with in case of new highest (could be trimmed out)
        - highestPrpslNum: highest known proposal number
            - Acceptors: uses it to reject promises
            - Proposers: uses it to generate higher proposal numbers
        - knownVal:
            - Acceptors: nil | value of highest proposal number known
            - Proposers: random | value returned by acceptors in Phase 1
        - complete: signals the Node is done and idle (should've name it 'idle')
    - Consensus Round: last known consensus round by the Node
- Nodes are shared across classes. The only other node type in the system is the Coordinator node which has less complexity and less flexibility for implemented scope. The Coordinator node also serves as the distinguished learner in question.

PROCESS | FLOW

- All nodes initialize their listeners and set up their own RPC servers, filling in their static states.
- The Coordinator triggers a consensus among proposers and gives them an upperbound integer with consensus to be reached on a random integer below the bound
    - Consensus are directly triggered as RPC calls on proposers by the coordinator
- **Proposer to Acceptor Communication | Phase 1**
    - Upon consensus round trigger, the proposer selects a random value below upper bound as its initial value and its port number as its initial proposal number making up its prepare request
    - The proposer then spawns concurrent goroutines to RPC call for a prepare response on each connected acceptor node
    - These Goroutines return through channels an error, a valid prepare response, or timeout (timeouts not simulated in implementation)
    - A prepare response is valid whether the acceptor accepts or rejects the proposal
        - The proposer keeps track off how many RPC calls succeed (acceptor accepted) and how many failed (all otherwise)
        - A prepare response that rejects a proposal still returns a prepare response with a known value and highest known proposal number from the acceptor
    - The proposer then determines majority success or majority failure
        - On success, it proceeds to Phase 2

        - On failure, it finds the next multiple of its port number higher than the known highest as its new proposal number, uses the known value, and loops back into Phase 1
            - **Problem**: Unique Proposal Numbers
            - **Solution**: Proposal Numbers generated on multiples of port number
                - Port numbers (~:8000) assigned to nodes are consecutive and unlikely to create collisions unless x8000 nodes are created

- Between Phase 1 and Phase 2, the proposer will find the majority value of prepare responses and prepare accept requests to acceptors that agreed to proposal
    - **Problem**: Needed to store which acceptors responded and continue Phase 2 with the subset.
    - **Solution:** Created a set of node_ids of acceptors that responded and accepted
        - **New Problem**: Initializing a new rpc client for the subset is inefficient, especially given multiple acceptors
        - **Solution:** Added a new Node state of type map[] (PeerClients) to cache rpc clients for connected peers. (implemented)
        - **Problem 2:** Race conditions in appending to set by concurrent goroutines
        - **Solution:** Mutex locks (implemented)
        - Alternative: Design pattern could be changed to avoid locks entirely

- **Proposer to Acceptor Communication | Phase 2**
    - In Phase 2, quite similarly to Phase 1, the proposer spawns concurrent goroutines to RPC call for an accept response from the subset of acceptors
    - These Goroutines return through channels an error, a valid accept response, or timeout (timeouts not simulated in implementation)
    - An accept response is valid whether the acceptor accepts or rejects the proposal
        - The proposer keeps track off how many RPC calls succeed (acceptor accepts accept request) and how many failed (all otherwise)
        - An accept response that rejects a proposal still returns a prepare response with a known value and highest known proposal number from the acceptor
    - The proposer then determines majority (of all acceptors) success or majority (of all acceptors) failure
        - On success, the Node sets its ‘complete’ state to true and goes into idle mode.
        - On failure, it goes back to Phase 1 with the next multiple of its port number higher than the known highest as its new proposal number and repeats.
- **Acceptors to Distinguished Learner**
    - When an acceptor accepts an accept request, it sends the value to the distinguished learner with an RPC call.

- There needs to be a message queue on the distinguished learner node to collect values sent back from acceptors to distinguished learners.
  - Option 1: Use an unbuffered channel (implemented to Solution 2)
    - **Problem:** Blocking waits on acceptors
    - **Solution:** Acceptors use goroutines to send values to the distinguished learners asynchronously
    - **Problem 2:** Previous consensus round values can still reach the distinguished learner when the system has already started the next round
    - **Solution 2**: Acceptors send (round_id, value) back to the distinguished learners, and ignore lower round_ids based on last known highest round_id (implemented)
      - Side Adv: The distinguished learner can keep validity count and only accept the first acceptor majority number of valid values (Invalid = values from previous consensus round).
      - Side Adv: The coordinator knows when consensus is completed and doesn't require a ticker at regular intervals to trigger the next consensus round.

  - Option 2: Use buffered channels
    - Buffered channels solve both the blocking waits and follow-up problems over using unbuffered channels
    - **Problem:** the size of the channel is inflexible when scale changes
  - Option 3: Use an unbuffered channel and close + empty it every round (instead of tuples)
    - Upon majority values sent back, the distinguished learner closes the channel and empties it to prepare for next round after counting majority value
    - **Problem:** complexity

- **CONSENSUS ROUNDS & STATES**
  - **Problem**: For scope of project, the consensus rounds are made to be discrete, operating in rounds of consensus triggered by a coordinator. Acceptors are not connected to the coordinator. In new rounds of consensus, they end up producing prepare responses with known values and highest proposal numbers from previous rounds for the newer rounds.
  - **Solution** (implemented)**:** Used consensus round identifiers set by the coordinator and used in prepare requests, prepare responses, accept requests, accept responses.
    - Concurrent consensus rounds are not supported, e.g. two consented values are requested from the system simultaneously (out of scope)
    - **New Problem**: Concurrency problem when an acceptor has an RPC call amidst sending back a prepare response whilst another one resets the state, the prepare response sent back (though it has no effect on consensus value; further discussed below) can potentially modify Node states if not handled.

- Option 1: Mutex Locks
    - **Problem**: Since whether is consensus round is checked only at the start of the RPC call on an acceptor and the state needs to be consistent throughout, the lock would've enveloped the entire RPC call and be inefficient
- Option 2: Create a copy of the node state upon RPC call receipt
    - **Problem**: Inefficient and seems lazy for scope
- Option 3: Handled as edge case on RPC call responses (implemented)
    - Simple if condition that aborts the continuation of the proposing function upon receipt of a prepare response with an older round_id

■ The problem doesn't really affect consensus value, because the reset (value = 0 and highest proposal number = 0) only occurs when some newer consensus round exists and new rounds are only triggered when the previous round has already reached a consensus on some value.

○ Nodes store a 'complete' boolean state that reset each consensus round to serve as discrete ends to a Node activity (real implementations may differ)

■ Acceptors stop responding to proposals once they have fulfilled an accept request and sent the final value to distinguished learner as well as an accept response to proposer
- It returns errors to any request by proposers afterwards.

■ Proposers stop proposing *iff* they received an agreement in an accept response OR a new consensus round is triggered by the coordinator

Theory refined or polished through project

- Prepare Responses whether an acceptor rejects or accepts a proposal always includes some useful information for the proposer to use on the next new proposal
    - On failure, it includes the highest proposal number known, so the proposer can retry a new proposal with a new higher proposal number
    - On success or failure, the prepare response contains the value the acceptor has already seen from another proposer and the proposer adopts the existing value
    - Q: What's the point in acceptors rejecting lower proposal numbers in the first place then? We could have proposers sending values to acceptors and acceptors sending values (whether proposed or seen) back.
        ■ Contradiction example:
        An acceptor is allowed to only reach out to a majority subset of acceptors. Suppose
            - Proposers = {proposer A, proposer B}
            - Acceptors = {acceptor X, acceptor Y, acceptor Z}
            - and A only proposes to X,Y and B only proposes to Y,Z
        ■ If only values were sent, say A proposes to X,Y (val = 5) and B proposes to Y,Z (val = 7), there will be no tiebreakers to handle what value Y receives. A total ordering is required

        - Q: Total Ordering can be achieved in multiple ways e.g. hybrid logical clocks. Why use proposal numbers?
            - A: <unexplored>, but proposal numbers are one way to achieve it
- Even when a proposer's Accept Request is rejected, it transitions back into Phase 1 and retries with a new proposal number.
    - Proposer retries because the other proposer that created the higher proposal number with its prepare request can still fail
    - **PAXOS Problem**: The Simplified PAXOS paper suggests there exists conditions where proposal numbers indefinitely increase
        - **Solution**: Random timeout between issuing new proposals by proposers
    - Proposer only stops proposing if an acceptor accepts its accept request
- RPC is based on TCP i.e RPC is built on top of TCP such that
    - RPC can be used to call functions server-side as well as send regular messages
    - Both require standardized protocols (TCP requires message protocols, RPC requires client-server to share function signatures)
- Q: How do proposers know when to begin proposing for the next consensus round without round_id identifiers? Is this standard?
- Real implementations of PAXOs have acceptors that do not stop accepting values even once sent to learners, and learners are responsible for ignoring duplicate proposal numbers, values and settling on a final value.
    - Current implementation undergoes Learner class entirely. PAXOS seems to be a continuous process rather than discrete as implemented in consensus round
- Most problems encountered were data type based and this document does not describe them in detail due to sheer number of them