❓ Q1. What is `WorkflowOptions` in Temporal?

Answer: `WorkflowOptions` is a configuration object that tells the Temporal server *how* to run the workflow. It includes things like:

- `workflowId`: A unique identifier for the workflow execution.
- `taskQueue`: Where the worker picks up the job from.
- Retry settings, timeout, etc.

You're not executing code here — you're configuring how the Temporal engine should run it.

❓ Q2. What is `WorkflowClient`?

Answer: `WorkflowClient` is the primary entry point to communicate with the Temporal Server from your Java app. You use it to:

- Start a new workflow execution.
- Get a stub (proxy) to call workflows.
- Send signals/queries to running workflows.

You asked why we use `WorkflowClient.start(...)` instead of `client.start(...)`. That's because `WorkflowClient.start()` is a **static utility method**, used to start a workflow execution on a given stub.

`client.newWorkflowStub(...)` gives you a proxy object. Then you use `WorkflowClient.start(...)` to actually tell Temporal to run it.

❓ Q3. What is a proxy in Temporal?

Answer: When you write:

```
OrderWorkflow workflow = client.newWorkflowStub(OrderWorkflow.class, options);
```

You don't get the real object. You get a **proxy** — a dummy object that represents the real workflow. It doesn't execute logic itself.

When you call `workflow.startOrder(...)`, Temporal intercepts this and sends a **gRPC** request to the Temporal server to start the workflow remotely.

You asked: *How can an interface have an object?* — Temporal uses Java dynamic proxy to create a fake object that *implements* the interface.

❓ Q4. What is gRPC and how does Temporal use it?

Answer:

- gRPC is a fast, binary-based communication protocol (like REST, but faster).
- When you call a method on the proxy (like `startOrder(...)`), Temporal uses gRPC to send that call to the Temporal Server.

It's like saying: "Please start this workflow remotely and store the result."

❓ Q5. Why does Temporal require interface for workflows?

Answer:

1. **gRPC compatibility**: Java dynamic proxies only work with interfaces.
2. **Determinism & Replay**: Temporal replays workflow history. Having a strict interface helps guarantee deterministic logic.
3. **Contract separation**: The engine only needs the *what*, not the *how*. Interfaces define the *what*.

You asked about terms like "replay", "yielding control":

- **Replay**: Re-executing past events to get back to current state.
- **Yielding control**: Temporarily stopping the workflow execution without blocking threads.
- **Why efficient?** Because workflows don't hold memory/threads. State is stored in DB and picked back up only when needed.

❓ Q6. What happens when you call `Workflow.await(...)`?

Answer: This suspends the workflow execution until a condition becomes true. Internally:

- Temporal saves the state in history.
- Frees up the worker thread.
- When a signal arrives (via controller), it logs the signal.
- The next time the worker picks this up, it **replays** all steps and re-checks the condition.

You asked how it knows a signal has arrived: ➡️Because it's **logged as an event** in the history. Temporal uses this event to resume.

❓ Q7. How is this controller code working without `WorkflowOptions`?

```
WorkflowStub stub = client.newUntypedWorkflowStub(orderId);
stub.signal("approveOrder", approverId);
```

Answer: You're not creating a workflow — you're **signaling** an existing one. Since the workflow already exists and has a known `workflowId` (orderId), Temporal fetches that workflow and sends a signal to it.

You asked: *Does `` need to match anything?* Yes — this string must match the `@SignalMethod(name = "approveOrder")` or the method name if not explicitly named.

❓ Q8. Why do we use `Workflow.getLogger(...)`?

Answer: This logger integrates with Temporal's logging and observability tools. It is workflow-safe and respects Temporal's replay behavior (doesn't log again on replay).

❓ Q9. Why is Temporal more efficient than a cron job?

Answer:

- Cron jobs run on threads, keep resources busy, and are hard to scale.
- Temporal persists state, suspends execution, uses zero CPU when idle.
- It restarts only when needed (e.g., when a signal or timer occurs).

That's why workflows can wait hours/days/weeks and still be cost-efficient.

❓ Q10. What happens when workflow is approved in code?

You had:

```
if (isApproved) {
    PaymentActivities payment = Workflow.newActivityStub(...);
    payment.processPayment(orderId);
    shipping.prepareShipment(orderId);
    ...
}
```

Answer: Once `isApproved == true`, the workflow resumes, creates **activity stubs**, and calls `processPayment()` and `prepareShipment()`.

Each of those activities are picked up by a worker registered to the same task queue.

Let me know if you'd like this formatted as a PDF or diagram as well!