This is an excellent question that gets to the core value of a container like StreamMessageListenerContainer!

You're right that at a fundamental level, both approaches involve using an Executor (a separate thread pool) for constant polling. However, the StreamMessageListenerContainer provides significant benefits and abstractions that you would have to build yourself if you were to manually implement the polling loop.

Here's the difference between "having a container" (like StreamMessageListenerContainer) and "not having a container" (i.e., manually implementing the polling loop with redis-streams-java-dist or even just plain Jedis):

1. Automated Polling Loop Management:

With Container: The container automates the polling loop. You don't write while(true) { read messages; process messages; }. The container handles:

Starting and stopping the polling threads.

Managing the frequency of polling (e.g., using XREAD BLOCK with a timeout, or a fixed delay between polls if non-blocking).

Error handling within the polling loop (e.g., reconnecting to Redis if the connection drops).

Graceful shutdown of the polling threads when your application stops.

Without Container: You are fully responsible for this loop. You would write code like:

Java

```
// Example pseudo-code for manual polling
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
   while (!Thread.currentThread().isInterrupted()) {
      try {
```

```
        // Call XREADGROUP or TopicConsumer.readNextMessage()
        List<MapRecord<String, Object, Object>> messages =
redisTemplate.opsForStream().read(Consumer.from("mygroup", "consumer1"),
StreamOffset.create("mystream", ReadOffset.lastConsumed()));

        if (messages != null && !messages.isEmpty()) {
          for (MapRecord<String, Object, Object> message : messages) {
            // Process message
            System.out.println("Received: " + message.getValue());
            // Acknowledge message
            redisTemplate.opsForStream().acknowledge("mystream", "mygroup", message.getId());
          }
        } else {
          // Short sleep to avoid busy-waiting if using non-blocking read
          Thread.sleep(100);
        }
      } catch (Exception e) {
        // Handle errors, maybe log and retry or exit
        e.printStackTrace();
      }
    }
});
// Need to manage executor lifecycle (shutdown)
```

This involves more boilerplate, careful error handling, and manual thread management.

2. Listener Abstraction & Invocation:

With Container: You simply register a listener (e.g., StreamListener). The container takes care of:

Deserializing the raw Redis stream entry into your desired Java object type.

Invoking your listener method with the strongly typed message.

Often handles automatic acknowledgement (XACK) of messages after successful processing (depending on configuration).

Without Container: You receive raw data (e.g., Map<String, String>). You are responsible for:

Manually parsing and converting the Map<String, String> into your domain object.

Manually calling your message processing logic.

Explicitly calling XACK at the correct time, ensuring it happens only after successful processing.

3. Error Handling & Retry Policies (More Sophisticated):

With Container: Containers often come with more advanced error handling capabilities:

Automatic retries for transient failures.

Configurable error handlers (e.g., ErrorHandler in Spring).

Dead-letter queue (DLQ) support for messages that consistently fail processing.

Handling of XPENDING messages and reclaiming messages from failed consumers.

Without Container: All error handling, retry logic, and dealing with XPENDING/XCLAIM must be custom-implemented by you. This is where a lot of complexity lies in building robust message consumers.

4. Spring Integration & Configuration:

With Container: Seamless integration with the Spring ecosystem. Configuration can often be done via @Configuration classes, properties files, or auto-configuration. It fits naturally into the Spring lifecycle.

Without Container: While you can still use Spring to manage your Executor and your custom polling component, you lose the higher-level, opinionated Spring Data Redis abstractions designed for message consumption.

5. Consumer Group Management (Beyond Basics):

With Container (Spring Data Redis): Handles the basic XREADGROUP and XACK logic. For more advanced distributed scenarios (like redis-streams-java-dist's sharding/topic management), it's less opinionated.

Without Container (redis-streams-java-dist): This is where libraries like redis-streams-java-dist shine if you need complex sharding, automated stream creation for topic scaling, and more sophisticated distributed consumer group coordination. However, as noted, even redis-streams-java-dist still requires you to implement the actual polling loop.

When to use which:

StreamMessageListenerContainer (Spring Data Redis):

Ideal for most Spring Boot applications consuming from single, logical Redis Streams.

Prioritizes ease of use, developer convenience, and rapid development.

You want Spring to manage the threading, polling, and listener invocation for you.

Manual Polling (using redis-streams-java-dist or basic Jedis):

When you have very specific, custom requirements for polling, error handling, or resource management that the container doesn't provide.

When you need the advanced distributed topic management and sharding capabilities of redis-streams-java-dist for extremely high-throughput scenarios that scale beyond a single Redis Stream.

When you are not in a Spring environment or prefer a lower-level, more granular control over every aspect of message consumption.

In essence, the StreamMessageListenerContainer acts as a framework that sits on top of the Executor and the raw Redis client calls. It handles all the repetitive, boilerplate logic of building a reliable message consumer, allowing you to focus purely on your business logic (what to do with the message) rather than the mechanics of how to receive it.