```
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;
import java.util.*;

public class BacktestApp {

    public static class YahooFinanceAPI {
        private static final String BASE_URL =
"https://query1.finance.yahoo.com/v8/finance/chart/%s?range=%s&interval
=%s";

        // Fetch stock data (price information) from Yahoo Finance API
        public static String fetchData(String ticker, String timeFrame,
String range) throws Exception {
            String interval;
            switch (timeFrame) {
                case "daily":
                    interval = "1d";
                    break;
                case "weekly":
                    interval = "1wk";
                    break;
                case "monthly":
                    interval = "1mo";
                    break;
                default:
                    throw new Exception("Invalid time frame. Use
'daily', 'weekly', or 'monthly'.");
            }
            String url = String.format(BASE_URL, ticker, range,
interval);
            OkHttpClient client = new OkHttpClient();
            Request request = new Request.Builder().url(url).build();
            try (Response response = client.newCall(request).execute())
{
                if (!response.isSuccessful()) {
                    throw new Exception("Failed to fetch data for
ticker: " + ticker);
                }
                return response.body().string();
```

```
            }
        }

        // Parse historical prices from Yahoo Finance API's JSON
response
        public static List<Double> parseHistoricalPrices(String
jsonData) {
            List<Double> prices = new ArrayList<>();
            JsonObject jsonObject =
JsonParser.parseString(jsonData).getAsJsonObject();
            JsonObject chart = jsonObject.getAsJsonObject("chart");
            JsonObject result =
chart.getAsJsonArray("result").get(0).getAsJsonObject();
            JsonObject indicators =
result.getAsJsonObject("indicators");
            JsonObject quote =
indicators.getAsJsonArray("quote").get(0).getAsJsonObject();
            for (int i = 0; i < quote.getAsJsonArray("close").size();
i++) {

prices.add(quote.getAsJsonArray("close").get(i).getAsDouble());
            }
            return prices;
        }
    }

    public static class Backtest {
        // Calculate Simple Moving Average (SMA)
        public static List<Double> calculateSMA(List<Double> prices,
int window) {
            List<Double> sma = new ArrayList<>();
            for (int i = window - 1; i < prices.size(); i++) {
                double sum = 0;
                for (int j = i - window + 1; j <= i; j++) {
                    sum += prices.get(j);
                }
                sma.add(sum / window);
            }
            return sma;
        }

        // Calculate Exponential Weighted Average (EWA)
```

```
        public static List<Double> calculateEWA(List<Double> prices,
int window) {
            List<Double> ewa = new ArrayList<>();
            double alpha = 2.0 / (window + 1);
            ewa.add(prices.get(0)); // Initialize with the first price
            for (int i = 1; i < prices.size(); i++) {
                double newEWA = alpha * prices.get(i) + (1 - alpha) *
ewa.get(i - 1);
                ewa.add(newEWA);
            }
            return ewa;
        }

        // Calculate daily returns
        public static List<Double> calculateReturns(List<Double>
prices) {
            List<Double> returns = new ArrayList<>();
            for (int i = 1; i < prices.size(); i++) {
                double dailyReturn = (prices.get(i) - prices.get(i -
1)) / prices.get(i - 1);
                returns.add(dailyReturn);
            }
            return returns;
        }

        // Backtest strategy with SMA, EWA, or combined signals
        public static Map<String, List<Double>>
backtestMomentumStrategy(List<String> tickers, String timeFrame, String
range, String strategy, int windowSize) {
            Map<String, List<Double>> individualReturns = new
HashMap<>();
            for (String ticker : tickers) {
                try {
                    String jsonData = YahooFinanceAPI.fetchData(ticker,
timeFrame, range);
                    List<Double> prices =
YahooFinanceAPI.parseHistoricalPrices(jsonData);
                    List<Double> returns = calculateReturns(prices);

                    // Pass windowSize dynamically to SMA and EWA
calculations
                    List<Double> sma = calculateSMA(prices,
windowSize);
```

```
                    List<Double> ewa = calculateEWA(prices,
windowSize);
                    List<Double> signalReturns = new ArrayList<>();

                    // Create signals based on the selected strategy
(SMA, EWA, or combined)
                    for (int i = windowSize; i < prices.size(); i++) {
                        double signalReturn = 0;
                        if ("sma".equals(strategy) && prices.get(i) >
sma.get(i - windowSize)) {
                            signalReturn = returns.get(i - 1); // SMA
Buy signal
                        } else if ("ewa".equals(strategy) &&
prices.get(i) > ewa.get(i - windowSize)) {
                            signalReturn = returns.get(i - 1); // EWA
Buy signal
                        } else if ("combined".equals(strategy)) {
                            double smaReturn = prices.get(i) >
sma.get(i - windowSize) ? returns.get(i - 1) : -returns.get(i - 1);
                            double ewaReturn = prices.get(i) >
ewa.get(i - windowSize) ? returns.get(i - 1) : -returns.get(i - 1);
                            signalReturn = Math.max(smaReturn,
ewaReturn); // Combined Buy signal
                        }
                        signalReturns.add(signalReturn);
                    }

                    individualReturns.put(ticker, signalReturns);

                } catch (Exception e) {
                    System.out.println("Error fetching or processing
data for " + ticker + ": " + e.getMessage());
                }
            }
            return individualReturns;
        }


        // Aggregate portfolio-level returns
        public static List<Double>
aggregatePortfolioReturns(Map<String, List<Double>> individualReturns)
{
            List<Double> portfolioReturns = new ArrayList<>();
```

```
            int portfolioSize =
individualReturns.values().iterator().next().size();

            for (int i = 0; i < portfolioSize; i++) {
                double dailyPortfolioReturn = 0;
                for (List<Double> tickerReturns :
individualReturns.values()) {
                    dailyPortfolioReturn += tickerReturns.get(i);
                }
                portfolioReturns.add(dailyPortfolioReturn /
individualReturns.size());
            }
            return portfolioReturns;
        }

        // Calculate Sharpe ratio
        public static double calculateSharpeRatio(List<Double> returns)
{
            double avgReturn =
returns.stream().mapToDouble(Double::doubleValue).average().orElse(0);
            double volatility =
Math.sqrt(returns.stream().mapToDouble(r -> Math.pow(r - avgReturn,
2)).average().orElse(0));
            return avgReturn / volatility;
        }

        // Calculate maximum drawdown
        public static double calculateMaxDrawdown(List<Double> returns)
{
            double peak = 0, maxDrawdown = 0, portfolioValue = 0;
            for (double dailyReturn : returns) {
                portfolioValue += dailyReturn;
                peak = Math.max(peak, portfolioValue);
                maxDrawdown = Math.min(maxDrawdown, portfolioValue -
peak);
            }
            return maxDrawdown;
        }

        // Calculate annualized return
        // Calculate annualized return
        public static double calculateAnnualizedReturn(List<Double>
returns, String frequency) {
```

```
            double totalReturn = 1.0;
            for (Double periodReturn : returns) {
                totalReturn *= (1 + periodReturn);
            }

            double periodsPerYear = 252.0; // Default for daily data
            if ("weekly".equals(frequency)) {
                periodsPerYear = 52.0; // For weekly data, there are 52
weeks in a year
            }

            return Math.pow(totalReturn, periodsPerYear /
returns.size()) - 1;
        }

    }

    public static void main(String[] args) {
        List<String> tickers = Arrays.asList("AAPL", "MSFT", "GOOG",
"AMZN", "AACG", "NVDA", "SPY", "JPM", "V", "MA",
            "NFLX", "DIS", "BABA", "GS", "IBM", "PFE", "WMT", "UNH",
"MS", "TSLA", "PYPL");

        List<String> strategies = Arrays.asList("sma", "ewa",
"combined"); // List of strategies
        List<Integer> windowSizes = Arrays.asList(5, 50); // List of
window sizes (5, 10, 50)
        List<String> timeframes = Arrays.asList("1y", "5y"); // List of
timeframes
        List<String> frequencies = Arrays.asList("daily", "weekly"); //
List of frequencies (daily, weekly)

        // Iterate over all combinations of strategy, window size,
timeframe, and frequency
        for (String strategy : strategies) {
            for (Integer windowSize : windowSizes) {
                for (String timeframe : timeframes) {
                    for (String frequency : frequencies) {
                        System.out.println("Running strategy: " +
strategy + ", Window Size: " + windowSize + ", Timeframe: " + timeframe
+ ", Frequency: " + frequency);
```

```
                        // Backtest the momentum strategy for each
combination of strategy, window size, timeframe, and frequency
                        Map<String, List<Double>> individualReturns =
Backtest.backtestMomentumStrategy(tickers, frequency, timeframe,
strategy, windowSize);
                        List<Double> portfolioReturns =
Backtest.aggregatePortfolioReturns(individualReturns);

                        // Portfolio-level analysis for each
combination
                        System.out.println("Results for strategy: " +
strategy + ", Timeframe: " + timeframe + ", Window Size: " + windowSize
+ ", Frequency: " + frequency);
                        System.out.println("Portfolio Sharpe Ratio: " +
Backtest.calculateSharpeRatio(portfolioReturns));
                        System.out.println("Portfolio Max Drawdown: " +
Backtest.calculateMaxDrawdown(portfolioReturns));
                        System.out.println("Portfolio Annualized
Return: " + Backtest.calculateAnnualizedReturn(portfolioReturns,
frequency)); // Use frequency here

System.out.println("----------------------------------------------------
------");
                    }
                }
            }
        }
    }
}
```