מדריך כללי למבחן

לפני כתיבת קוד:

1. קראו את המבחן בזהירות - ודאו שאתם מבינים כל הנחיה וכל דרישה. לנחש למה המבחן מתכוון זו לא דרישה שלנו מכם! שאלו אם משהו לא ברור.
2. צרו תרשים זרימה ברור לאפליקציה. ציינו לעצמכם איפה יש תנאים ואיפה יש לולאות
3. חפשו ישויות מרכזיות באפליקציה ורשמו לעצמכם אותן - בקוד תוודאו שיש להן ייצוג
4. חפשו פעולות מרכזיות באפליקציה ורשמו לעצמכם אותן - בקוד ודאו שיש פונקציות שמייצגות כל פעולה. הקפידו על עקרון SINGLE RESP - אם צריך לפרק פונקציות לתת פונקציות - עשו את זה.
5. עבור כל פעולה, כתבו אלגוריתם מדויק ומפורק במילים **לפני** שאתם כותבים קוד. רק אחרי שיש לכם אלגוריתם מלא, שממלא את כל הדרישות - תרגמו אותו לקוד.
6. חלקו לעצמכם זמנים. הקצו לכל פעולה/פונקציה כמה זמן אתם חושבים שתעבדו עליה - **ועקבו אחרי הזמן הזה**. ככה לא תגיעו לסוף המבחן כשאין לכם פי'צרים משמעותיים.
7. תתחילו לעבוד קודם על הפיצ'רים שנראים לכם קלים יותר.

במהלך כתיבת קוד:

8. צרו ענף נפרד בגיט לכל פיצ'ר, שלא תהרסו בטעות קוד קיים שכבר עובד ושתוכלו לחזור אחורה אם אתם מתחרטים על שינוי שעשיתם. אל תשכחו לאחד עם MAIN בסיום העבודה.
9. ודאו את עצמכם - על כל כמה שורות קוד עצרו את העבודה, ודאו עם DEBUGGING שאתם באמת מקבלים את התוצאה שאתם חושבים, ורק אז המשיכו. אל תכתבו הרבה שורות קוד ללא וידוא ובדיקה! אתם רק מסדרים לעצמכם שגיאות שיהיה קשה מאוד לאתר אחר כך.
10. תקנו כל שגיאת סינטקס או שגיאה לוגית כשאתם נתקלים בה. אל תגישו קוד שיש בו שגיאות, במיוחד אם אלו שגיאות סינטקס! אנחנו נוריד על זה נקודות, והרבה.
11. ודאו את עצמכם גם אל מול דרישות המבחן - אל תשקיעו שעות בלממש פעולה מסוימת רק כדי לגלות בסוף שזה בכלל לא מה שהמבחן דרש.

12. אם אתם נתקעים במהלך כתיבת הקוד ולא ברור לכם איך להמשיך - עצרו את כתיבת הקוד וחזרו לשלב התכנון: הגדירו בבירור מה הבעיה שאתם מנסים לפתור וכתבו אלגוריתם מפורט שפותר אותה. רק אחרי - חזרו לכתוב קוד.

אחרי כתיבת הקוד:

13. ודאו את הקוד שלכם שוב - תריצו, תראו שאין שגיאות, ודאו מול המבחן שעמדתם בדרישות.
14. נקו את הקוד - הסירות הדפסות מיותרות, הסירו קוד בהערות, הסירו שורות ריקות ורווחים לא נצרכים. אנחנו נוריד נקודות על מי שיגיש קוד מבולגן.

Stock Trading App – Test Description

General

You will build an application that works with a given stock market object and allows the user to perform several stock operations.

A stock has the following fields:

- name (unique)
- id (unique)
- current price
- available stocks (how many units can be bought)
- previous prices (an array that stores all past prices)
- category

The stock market itself is an object containing:

- last updated (a date or timestamp for when any stock price was last changed)
- stocks: an array of all stock objects

see given data.

Each stock can be bought or sold.
When a stock is bought or sold, its price changes, and this also affects other stocks in the market:

- Stocks in the same category move in the same direction. When buying - price goes up, when selling - price goes down.

Example:
If you buy a stock from the **general tech** category:

- That specific stock price goes up.
- All general tech stocks go up

Selling reverses these effects.

Price Change Rules for Buying

When a buy action happens:

- The **selected stock** price increases by 5% from its previous price.
- All stocks in the same category increase by 1% of their current price - **not including the selected stock**.

Other than updating the prices after a stock price changes, you must:

1. Add the current price into the “previous prices” array.
2. Update the “current price” field with the new value.
3. Update the “last updated” field in the stock market object.

User Menu

Your program must show a menu that supports:

- Search for a stock by name or id
- Show all stocks above or below a given price
- Buy or sell a stock
- Exit

print the result of each selected operation - the user must know what happened. make the logs meaningful!

the menu should show after every operation until the user selects “exit”.

Required Functions

You must implement these functions with the **exact signatures**.
You may add helper functions if needed.

If a function becomes too large, break it into smaller, more focused functions.

in the functions you can add relevant logs that explain what is happening

---

searchStock(identifier)

Input: name or id
Output: returns an array of all stocks whose name or id matches exactly.
if no matches are found, log it, and return an empty array.

---

filterStocksByPrice(givenPrice, above)

Searches all stocks and returns an array of stocks whose price is:

- above the given price, if above is true
- below the given price, if above is false

must verify that the given price is a number.

if no stocks are found - log it and return an empty array.

OperateOnStock(operation, identifier)

Handles buying or selling a stock.

Requirements:

- operation must be “buy” or “sell”
- identifier can be name or id
- Handle invalid operations and unknown identifiers
- Ask the user how many units to buy or sell
- Update the available stock count
- Update the price of the selected stock
- Update the prices of all stocks in the same category

returns nothing

Code Quality Requirements

Your code must be:

- Organized and modular
- Clearly written, with meaningful names - variables, functions, files and folders.
- Structured logically (place functions and variables where a reader expects them to be)
- Free of debug logs - you can keep logs if describe what is happening in the flow of the app (not, for example, if they just print out some value)
- No commented-out code
- No unnecessary spacing or empty lines
- Functions generally should validate inputs and handle errors

Bonus Task – Market Trend Analysis

Create an additional feature called **Market Trend Analyzer**.
This feature must help the user understand long-term stock behavior, using the data stored in the previous prices arrays.

You must implement the following function:

analyzeMarketTrends()

This function returns a report object with the following fields:

1. **topIncreasingStocks**
   An array of the **three stocks with the largest overall growth**, where:
   growth = (current price – oldest previous price)
   If a stock has no history (empty previous prices), skip it.

2. **topDecreasingStocks**
   An array of the **three stocks with the largest overall decline** using the same formula above.

3. **mostVolatileStock**
   The single stock whose price changed the most over time.
   Volatility = maximum(previous prices including current) – minimum(previous prices including current)

4. **categoryStability**
   An object where each key is a category name and each value is the **average volatility** of all stocks in that category.
   Higher value = less stable category.

Requirements:

- The calculations must be correct and based only on actual stock data.

- You may create helper functions (for example: calculateGrowth, calculateVolatility, groupByCategory).

- The function must not print; it must **return** a structured report.

Example Output Structure (not real values):

```
{
  topIncreasingStocks: ["Google", "BP Oil", "Apple"],
  topDecreasingStocks: ["EduTech", "GreenFuture", "AI Systems"],
  mostVolatileStock: "Tesla",
  categoryStability: {
    education: 12.4,
    AI: 33.1,
    factories: 7.9,
    general tech: 15.2,
    oil: 5.0,
    green energy: 27.3
  }
}
```

General grading:

manu - 10 points
search function - 10 points
filter function - 10 points
price update logic - 50 points
clean code and modularity - 20 points
bonus - 10 points