ASSIGNMENT TITLE

Mini Project: SmartTrade – Real-Time Trading System Using Advanced C#

COURSE CONTEXT

Course: Advanced C# Programming
Module: Generics, Nullable Types, Extension Methods, Pattern Matching
Difficulty Level: Intermediate → Advanced
Assignment Type: Individual Mini Project

PROBLEM STATEMENT

A brokerage firm wants to build a **lightweight console-based trading system** for internal testing and learning purposes.
The system must simulate **equity trading**, handle **market price fluctuations**, calculate **trade value and brokerage**, and maintain **global trade analytics**.

You are required to design and implement this system using **advanced C# features**, ensuring clean architecture, type safety, and performance.

OBJECTIVES

After completing this assignment, the student should be able to:

- Apply advanced C# language features in a real-world domain
- Design extensible and maintainable systems
- Demonstrate understanding of generics, nullable types, and extension methods
- Implement clean object-oriented design

SYSTEM REQUIREMENTS

The system must:

- Support equity trades
- Handle missing market prices safely
- Store trades generically
- Track total trades at system level
- Apply brokerage and tax calculations
- Process trades dynamically

TASKS TO BE PERFORMED (VERY SPECIFIC)

TASK 1: Market Price Snapshot Using Struct

Student MUST do the following:

1. Create **ONE struct** named `PriceSnapshot`
2. The struct MUST contain:
    - Stock Symbol
    - Stock Price
3. The struct MUST be used only for **temporary market data**

Expected Implementation Proof:

- Create at least **one PriceSnapshot instance**
- Display symbol and price

TASK 2: Base Trade Abstraction

Student MUST do the following:

1. Create **ONE abstract class** named `Trade`

2. The class MUST contain:
    - Trade ID
    - Stock Symbol
    - Quantity
3. Declare **ONE abstract method**:
    - To calculate trade value
4. Override `ToString()` from `System.Object`

Expected Implementation Proof:

- Display trade details using `ToString()`

TASK 3: Equity Trade Implementation

Student MUST do the following:

1. Create **ONE concrete class** named `EquityTrade`
2. The class MUST:
    - Inherit from `Trade`
3. Add **ONE nullable property**:
    - Market Price
4. Implement trade value calculation using:
    - Null coalescing operator

Expected Implementation Proof:

- Calculate trade value with:
    - Market price present
    - Market price missing (null)

TASK 4: Generic Trade Repository

Student MUST do the following:

1. Create **ONE generic class** named `TradeRepository<T>`
2. Apply **generic constraint** so that:
    - Only Trade types are allowed
3. The repository MUST:

    - Store multiple trades
    - Add new trades
4. Increment a global counter whenever a trade is added

Expected Implementation Proof:

- Add at least **two trades**
- Show repository contents

TASK 5: Static Trade Analytics

Student MUST do the following:

1. Create **ONE static class** named `TradeAnalytics`
2. The class MUST contain:
    - Static variable to track total trades
    - Static method to display analytics

Expected Implementation Proof:

- Display total number of trades executed

TASK 6: Extension Methods for Financial Calculations

Student MUST do the following:

1. Create **ONE static class** for extensions
2. Add:
    - Brokerage calculation method
    - Tax (GST) calculation method
3. These methods MUST:
    - Extend numeric types
    - Not modify Trade class

Expected Implementation Proof:

- Apply brokerage and tax on trade value

TASK 7: Pattern Matching for Trade Processing

Student MUST do the following:

1. Create **ONE trade processing method**
2. Use **pattern matching** to:
    - Identify EquityTrade
    - Execute appropriate logic

Expected Implementation Proof:

- Display trade type during processing

TASK 8: Boxing and Unboxing

Student MUST do the following:

1. Store total trade count in an object type
2. Retrieve it back into a value type

Expected Implementation Proof:

- Print boxed and unboxed values

TASK 9: Main Program Flow

Student MUST do the following:

1. Create repository instance
2. Create at least **two EquityTrade objects**
3. Assign:
    - Trade ID
    - Symbol
    - Quantity
    - Market price
4. Add trades to repository
5. Process trades
6. Display:
    - Trade details

    - Trade value
    - Brokerage
    - Tax

7. Display global analytics

OUTPUT REQUIREMENTS (MANDATORY)

The output MUST include:

- Trade processing message
- Trade details
- Calculated trade value
- Brokerage charges
- Tax amount
- Total trades executed

CONSTRAINTS

- Do NOT use external libraries
- Do NOT skip any task
- Do NOT hardcode output values
- Follow object-oriented principles strictly

EVALUATION CRITERIA

Area	**Weightage**
Correct use of advanced C# features	40%
Clean design & structure	25%
Output correctness	20%
Code readability	15%

SUBMISSION REQUIREMENTS

- Complete source code
- Console output screenshots
- Brief explanation (1–2 lines) per task

BONUS (OPTIONAL)

- Add a new trade type
- Add risk validation logic

EXPECTED OUTCOME (TASK-WISE)

TASK 1: Market Price Snapshot Using Struct

Expected Outcome

Console Output MUST show:

```
Stock Symbol: AAPL
Stock Price: 150.50
```

What this confirms

- A `struct` is created and instantiated
- Data is stored as a value type
- Struct is used for temporary market data

TASK 2: Base Trade Abstraction

Expected Outcome

Console Output MUST show trade details using overridden method:

```
TradeId: 1
Symbol: AAPL
Quantity: 100
```

What this confirms

- Abstract class is implemented correctly
- `System.Object.ToString()` is overridden
- Base class behavior is reused

TASK 3: Equity Trade Implementation

Expected Outcome – Case 1 (Market Price Available)

```
Trade Value: 15050
```

Expected Outcome – Case 2 (Market Price Missing)

```
Trade Value: 0
```

What this confirms

- Nullable type is used correctly
- Null-coalescing operator prevents runtime error
- Trade calculation logic is safe

TASK 4: Generic Trade Repository

Expected Outcome

Console Output MUST confirm multiple trades added:

```
Trade added successfully
Trade added successfully
```

What this confirms

- Generic repository stores multiple trade objects
- Generic constraint restricts type usage
- Repository logic is reusable and type-safe

TASK 5: Static Trade Analytics

Expected Outcome

Console Output MUST show global count:

```
Total Trades Executed: 2
```

What this confirms

- Static variable tracks system-wide data
- Count persists across objects
- Static method accesses static data correctly

TASK 6: Extension Methods for Financial Calculations

Expected Outcome

Console Output MUST show calculated charges:

```
Trade Value: 15050
Brokerage: 15.05
GST: 2.709
```

What this confirms

- Extension methods are applied successfully

- Financial logic is external to core classes
- Clean separation of responsibilities

TASK 7: Pattern Matching for Trade Processing

Expected Outcome

Console Output MUST show trade classification:

```
Processing Equity Trade
```

What this confirms

- Pattern matching identifies runtime type
- Correct logic is executed based on trade type
- No casting errors occur

TASK 8: Boxing and Unboxing

Expected Outcome

Console Output MUST show both values:

```
Boxed Trade Count: 2
Unboxed Trade Count: 2
```

What this confirms

- Value type converted to object (boxing)
- Object converted back to value type (unboxing)
- Student understands performance implication

TASK 9: Main Program Flow

Expected Outcome (Complete Execution)

Console Output MUST appear in logical order:

```
Processing Equity Trade
Trade Value: 15050
Brokerage: 15.05
GST: 2.709
TradeId: 1, Symbol: AAPL, Qty: 100

Processing Equity Trade
Trade Value: 0
Brokerage: 0
GST: 0
TradeId: 2, Symbol: MSFT, Qty: 50

Total Trades Executed: 2
```

What this confirms

- All components integrate correctly
- Program flow follows real trading logic
- No runtime exceptions occur
- Analytics reflect actual operations