Practical Exercises 1

Example 1

Practice Brief: Vehicle Management Console App

1 Overview

Build a simple **Vehicle Management System** as a C# console application.
The system should allow users to create and display different types of vehicles (e.g., Car, Motorcycle, Truck), demonstrating **inheritance**, **polymorphism**, and **interface usage**.

---

2 Learning Goals

By completing this task, learners will:

- Understand **class inheritance** and **method overriding**.
- Apply **interfaces** to define common behavior across unrelated types.
- Practice **runtime polymorphism** using base class and interface references.
- Use **collections** (e.g., `List<T>`) to manage multiple objects.
- Improve code organization using **abstraction and encapsulation**.

---

3 Requirements

1. Create a **base class** `Vehicle` with:
    - Properties: `Brand`, `Model`, `Year`.
    - Method: `StartEngine()` (virtual) prints `"The vehicle engine starts."`.
2. Create at least **three derived classes**:
    - `Car`: adds `NumberOfDoors` and overrides `StartEngine()` `"The car engine starts with a key."`.
    - `Motorcycle`: adds `HasSidecar` and overrides `StartEngine()` `"The motorcycle engine starts with a button."`.
    - `Truck`: adds `CargoCapacity` and overrides `StartEngine()` `"The truck engine rumbles to life."`.
3. Define an **interface** `IDriveable` with:
    - Method `Drive()` void, no parameters.
4. Make all vehicle types implement `IDriveable`:
    - Each class should provide a specific `Drive()` implementation (e.g., `"The car is driving on the road."`).
5. In `Program.cs`:
    - Create a `List<Vehicle>` and add different vehicle types.
    - Loop through the list and:
        - Call `StartEngine()` (shows polymorphism).
        - Check if the vehicle implements `IDriveable` and call `Drive()`.
6. Allow user input:
    - A simple menu to add new vehicles (Car, Motorcycle, Truck).
    - Ask for details via `Console.ReadLine()`.
    - Add to the list dynamically.
7. When the user chooses “List Vehicles”, show:
    - Vehicle type
    - Brand, Model, Year
    - Any specific properties (e.g., Doors, Sidecar, Capacity)
8. The program exits when user selects “Exit”.

---

4 Architecture & Patterns

- **Pattern:**
    - *Inheritance* for shared attributes and methods.
    - *Interface* for cross-cutting behavior (`IDriveable`).
    - *Polymorphism* to execute derived behavior from base references.
- **Structure:**

```
VehicleManagement/
 Models/
    Vehicle.cs
    Car.cs
    Motorcycle.cs
    Truck.cs
 Interfaces/
    IDriveable.cs
 Program.cs
```

---

6 Acceptance Criteria

#	Given	When	Then
1	The app runs	User selects “Add Car”	Car object is created and added to the list
2	Vehicle list is non-empty	User selects “List Vehicles”	Each vehicle type shows type-specific StartEngine() and Drive() messages
3	All vehicle types implement `IDriveable`	User loops through vehicles	Drive() method outputs correct message
4	User selects “Exit”	—	Program terminates gracefully

---

7 Evaluation Rubric (Total 10 points)

Category	Description	Points
Correctness	Classes and interface implemented correctly	3
OOP Concepts	Proper inheritance, overriding, and polymorphism	3
Code Quality	Readable structure, naming conventions, no duplication	2
User Interaction	Menu and input handling work properly	1
Console Output	Clear and meaningful output	1

---

8 Extensions (Optional Stretch Goals)

- Add `ElectricCar` with battery range and override `StartEngine()` differently.
- Implement an `IRefuelable` interface for `Truck`.
- Serialize and deserialize vehicles to JSON using `System.Text.Json`.
- Add filtering by vehicle type before listing.

---

9 Deliverables & Constraints

- **Runtime:** .NET 8 or 9 Console App
- **Namespace:** `VehicleManagement`
- **No external packages** (only `System` namespaces)
- **Nullable reference types:** Enabled
- **Timebox:** ~1 day

What the Reviewer Should Look For

- Inheritance chain (`Vehicle` derived classes) is correct.
- Method overriding demonstrates polymorphism.
- Interface (`IDriveable`) is used correctly and implemented by all vehicles.
- Code compiles, runs, and handles user input without crashes.
- Output clearly differentiates vehicle behaviors.

Example 2

Practice Brief: Pet Shelter Console App

1) Overview

Build a console app for a small **Pet Shelter**. Staff can register animals (Dog, Cat, Bird), list them, feed them, and mark them as adopted. Use **inheritance** for shared behavior, **interfaces** for cross-cutting actions, and **polymorphism** to vary behavior per animal type.

---

2) Learning Goals

- Create an **abstract base class** and **derived classes**.
- Implement **interfaces** and use **runtime polymorphism** via base/interface references.
- Use **override** and **virtual/abstract** methods properly.
- Manage objects with collections (`List<T>`), basic input parsing, and simple menu loops.

---

3) Requirements

1. **Base class `Animal` (abstract)**
    - Properties: `int Id`, `string Name`, `int Age`, `DateTime IntakeDate`.
    - Methods:
        - `public abstract void Speak();`
        - `public virtual decimal DailyCareCost() => 5m;` (base cost).
2. **Derived classes**
    - `Dog`: adds `bool IsTrained`.
        - `Speak()` prints `"Woof!"`.
        - `DailyCareCost()` base + 3.
    - `Cat`: adds `bool IsIndoor`.
        - `Speak()` `"Meow!"`.
        - `DailyCareCost()` base + 2.
    - `Bird`: adds `double WingSpanCm`.
        - `Speak()` `"Chirp!"`.
        - `DailyCareCost()` base + 1.
3. **Interfaces**
    - `IFeedable` with `void Feed();`
        - All animals implement it; each prints a type-specific message (e.g., `"Dog Buddy has been fed."`).
    - `IFlyable` with `void Fly();`
        - Only `Bird` implements it; prints a message using `WingSpanCm`.
4. **Polymorphism demo**
    - Maintain `List<Animal>` as the shelter inventory.
    - Iterate and call `Speak()` + `DailyCareCost()` on each (dynamic dispatch).
    - When `IFlyable`, call `Fly()` (interface check).
5. **Menu (loop until Exit)**
    - `1) Add Dog` prompt for `Name`, `Age`, `IsTrained (y/n)`.
    - `2) Add Cat` prompt for `Name`, `Age`, `IsIndoor (y/n)`.
    - `3) Add Bird` prompt for `Name`, `Age`, `WingSpanCm (number)`.
    - `4) List Animals` table: `Id | Type | Name | Age | Extra | DailyCost`.
    - `5) Feed All` iterate `IFeedable.Feed()` and show a summary count.
    - `6) Speak All` iterate `Speak()`.
    - `7) Adopt (by Id)` mark animal as adopted and remove from list; print confirmation.
    - `8) Exit`.
6. **Id management**
    - Auto-incrementing integer starting at 1 (local counter).
7. **Input validation**
    - Age must be `>= 0`; WingSpan must be `> 0`.
    - Re-prompt on invalid input; do not crash on empty/invalid values.

8. **Non-functional**
    - Clear, small methods (SRP).
    - No external packages.
    - Use `tryParse` patterns and guard clauses.

---

4) Architecture & Patterns

- **Inheritance:** `Animal` `Dog|Cat|Bird` for shared state/behavior.
- **Interfaces:**
    - `IFeedable` for cross-cutting feeding behavior.
    - `IFlyable` to show interface segregation (only `Bird`).
- **Polymorphism:** Call `Speak()`/`DailyCareCost()` via `Animal` references; call `Fly()` via interface check.
- **Structure**

```
PetShelter/
 Models/
   Animal.cs
   Dog.cs
   Cat.cs
   Bird.cs
 Interfaces/
   IFeedable.cs
   IFlyable.cs
 Program.cs
```

---

5) Object Model

- **Entities:** `Animal` (abstract), `Dog`, `Cat`, `Bird`.
- **Key:** `Id` (int, unique within in-memory list).
- **Relationships:** None (flat list).
- **Notes:** Keep `DailyCareCost()` as `decimal` to practice numeric formatting.

---

6) Console UI Specification

- **Routes/Actions:** Console menu items (1–8).
- **Input rules:**
    - `Name`: non-empty (re-prompt if empty).
    - `Age`: integer `>= 0`.
    - `WingSpanCm`: positive number.
- **Outputs/status:**
    - Success messages on add/adopt; friendly validation errors.
    - Listing shows `DailyCareCost()` formatted with 2 decimals.
    - Adopt (by Id): if not found `"Animal not found."`

---

7) Testing Plan (manual, no unit tests)

- Add one of each type; run **List**, **Speak All**, **Feed All** verify type-specific outputs.
- Attempt invalid inputs (negative age, bad bool) app re-prompts without crashing.
- Adopt valid Id removed from list; adopt invalid Id “not found”.

---

8) Acceptance Criteria

1. **Given** the app is running, **when** I add a Dog/Cat/Bird, **then** it appears in **List** with correct type and fields.
2. **Given** animals exist, **when** I choose **Speak All**, **then** each prints its override (`Woof/Meow/Chirp`).
3. **Given** animals exist, **when** I choose **Feed All**, **then** each prints its `IFeedable` message; Bird can also demonstrate `IFlyable` when explicitly invoked in listing or a dedicated option.
4. **Given** a valid Id, **when** I **Adopt**, **then** it’s removed and a confirmation is shown.

5. **Given** invalid inputs, **when** I enter them, **then** the app re-prompts and does not crash.

---

9) Evaluation Rubric (Total 10)

- **Correctness (3):** Classes, interface implementations, menu flow meet requirements.
- **OOP Usage (3):** Proper abstract base, overrides, interface checks, polymorphic calls.
- **Input Handling (2):** Validation and re-prompting for bad input.
- **Code Quality (1):** Naming, small methods, minimal duplication.
- **Console UX (1):** Clear outputs and formatting.

---

10) Extensions (Optional)

- Add `Reptile` with `IsVenomous`, custom cost.
- Add **option 9: Fly Birds** iterate `IFlyable`.
- Add **search/filter** by type or name.
- Add **total daily cost** summary for all animals.

---

11) Deliverables & Constraints

- **Runtime:** .NET 8 or 9 Console App
- **Namespace:** `PetShelter`.
- **Timebox:** ~1-2 day.
- **Coding conventions:** PascalCase for types/methods, camelCase locals, early returns for validation.

What the reviewer should look for

- Clear abstract base and derived classes; correct use of `override`/`virtual`.
- `IFeedable` implemented across all animals; `IFlyable` only on `Bird`.
- Polymorphism demonstrated via `Animal` references and interface checks.
- Robust input handling; no crashes on invalid input.
- Clean console output with correct `DailyCareCost()` calculations.

Example 3

Practice Brief: “Smart Home Console Remote”

1) Overview

Build a **console-based smart-home controller** that manages different devices (e.g., LightBulb, Thermostat, SmartPlug). Devices share common traits but have **type-specific behavior**. Emphasis: **interfaces**, **inheritance**, **polymorphism**.

---

2) Learning Goals

- Define and implement **interfaces** for shared capabilities (power, status, special features).
- Use an **abstract base class** for shared state/behavior.
- Demonstrate **runtime polymorphism** when invoking actions through base/interface references.
- Practice clean console I/O and simple state management.

---

3) Requirements

Functional

1. The app maintains an in-memory list of smart devices.
2. Supported device types at minimum: **LightBulb**, **Thermostat**, **SmartPlug**.
3. Menu operations:
    a. List devices (id, name, type, power state, key attribute).

b. Add device (choose type and properties).
c. Toggle power (On/Off) for a selected device.
d. Device actions (contextual):

- LightBulb: set brightness (0–100).
- Thermostat: set target temperature (10–30 °C).
- SmartPlug: read instantaneous load (simulated) and reset energy counter.

e. Run self-test for **all** devices (polymorphic).
f. Exit.

4. Input is validated; invalid options do not crash the app.

Non-Functional

1. Use **interfaces** for capability contracts: e.g., `IPowerSwitch`, `ISelfTest`, and optional capability interfaces (`IDimmable`, `ITemperatureControl`, `IMeasurableLoad`).
2. Use an **abstract** `SmartDevice` base class for `Id`, `Name`, `IsOn`, `GetStatus()`.
3. Use **polymorphism** by storing devices in a single `List<SmartDevice>` and invoking common actions via base/interface references.
4. Only standard .NET libraries; keep code clear.

3.1 Functional Requirements

1. The app maintains an **in-memory list of smart devices**.
2. Supported device types (at minimum):

- **LightBulb**
- **Thermostat**
- **SmartPlug**

3. Console menu operations:

a. **List devices** — display id, name, type, power state, and main attribute (e.g., brightness, temperature).
b. **Add device** — choose type and configure basic properties.
c. **Toggle power** — turn On/Off any selected device.
d. **Device actions** — context-sensitive options based on the device's capabilities:

- LightBulb set brightness (0–100).
- Thermostat set target temperature (10–30 °C).
- SmartPlug view instantaneous load and optionally reset its energy counter.

e. **Run self-test** for all devices — should iterate polymorphically across the device list.
f. **Exit** — ends the program gracefully.

4. User input should be validated; invalid input or menu options must not crash the program.
5. Each operation should display feedback in a clear and readable format.

3.2 Non-Functional Requirements

1. Use **interfaces** to define clear behavioral contracts:

- Power control
- Self-testing capability
- Device-specific features (dimming, temperature control, load measurement)

2. Use an **abstract base class `SmartDevice`** to hold shared data and implement common functionality `Id`, `Name`, `IsOn`, `GetStatus()`.
3. Use **polymorphism** — all devices should be stored as `SmartDevice` objects in a single list and interacted with through base or interface references.
4. Only standard .NET libraries; keep code clear.

3.3 Interfaces Overview (Detailed)

Interface	Purpose	Members	Implemented By	Notes
`IPowerSwitch`	Defines standard power control operations common to all devices.	`void PowerOn(); void PowerOff();`	`SmartDevice` (base class implements)	Allows turning any device on/off in a consistent way.
`ISelfTest`	Defines diagnostic behavior for checking if a device is functional.	`bool SelfTest();`	`SmartDevice` (abstract) overridden in each derived device	Demonstrates polymorphism — called on all devices via interface.
`IDimmable`	Defines adjustable brightness capability.	`int Brightness { get; } void SetBrightness (int value);`	`LightBulb`	Example of interface segregation — only lights implement dimming.
`ITemperatureControl`	Defines control for temperature-targeted devices.	`double TargetCelsius { get; } void SetTarget(double celsius);`	`Thermostat`	Used to set the target temperature range (10–30 °C).
`IMeasurableLoad`	Defines power monitoring features for energy-measuring devices.	`double CurrentWatts { get; } double TotalWh { get; } void ResetEnergy();`	`SmartPlug`	Simulates instantaneous and accumulated electrical usage.

Design Intent:

- Each interface represents a distinct **capability**, following the **Interface Segregation Principle (ISP)**.
- Devices **only implement** the interfaces they need — no empty methods or irrelevant members.
- The console app queries interfaces dynamically (via `is` pattern matching) to offer **context-aware menus**.
- The `ISelfTest` interface showcases **polymorphism** when invoked through a shared list of mixed device types.

Summary of Behavior Flow

- `SmartDevice` implements `IPowerSwitch` and `ISelfTest` (base-level behaviors).
- Derived classes extend functionality:
    - `LightBulb` adds `IDimmable`.
    - `Thermostat` adds `ITemperatureControl`.
    - `SmartPlug` adds `IMeasurableLoad`.
- The main loop presents menu options based on which interfaces a selected device supports — allowing polymorphic dispatch without knowing concrete types.

---

4) Architecture & Patterns

- **Base + Interfaces:**
    - `SmartDevice` (abstract) shares identity & power state.
    - Capability interfaces segregate concerns (ISP).
- **Strategy-lite via capabilities:** Behavior varies by whether a device implements a capability interface.
- **Why:** Encourages thinking in contracts, enables polymorphic operations on mixed device types.

Suggested layout

```
SmartHomeApp/
  Program.cs
  Models/
      SmartDevice.cs
      LightBulb.cs
      Thermostat.cs
      SmartPlug.cs
      Interfaces/
           IPowerSwitch.cs
           ISelfTest.cs
           IDimmable.cs
           ITemperatureControl.cs
           IMeasurableLoad.cs
  Services/
       DeviceRegistry.cs
```

---

5) UI Specification (Console Menu)

Example flow:

```
=== SMART HOME CONSOLE REMOTE ===
1. List devices
2. Add device
3. Toggle power
4. Device actions
5. Self-test all
6. Exit
Select: 2
Choose type (LightBulb/Thermostat/SmartPlug): LightBulb
Enter name: Desk Lamp
Added: 1: Desk Lamp (LightBulb) - Power: Off, Brightness: 50%
```

---

6) Testing Plan

No automated tests (per request).
Manual checks:

- Add at least one of each device type.
- Power On/Off reflects in `GetStatus()`.
- Capability menus appear only when applicable (e.g., brightness only for LightBulb).
- “Self-test all” iterates over `List<SmartDevice>` and prints pass/fail for each (polymorphism).

---

7) Acceptance Criteria

- **Given** a fresh app, **when** the user adds devices, **then** each is assigned an incrementing `Id` and appears in “List devices”.
- **Given** mixed device types, **when** “Self-test all” runs, **then** the app calls `SelfTest()` on each via base/interface references and prints results.
- **Given** a device selection, **when** the user opens “Device actions”, **then** only actions from implemented capability interfaces are offered.
- **Given** invalid input, **then** the app shows an error message and returns to the menu without crashing.

---

8) Evaluation Rubric (10 pts)

- **Correctness (4):** Menu works, actions change state, no crashes on invalid input.
- **OOP Concepts (3):** Proper interface segregation, abstract base use, runtime polymorphism.
- **Code Quality (2):** Structure, naming, comments where helpful.
- **UX (1):** Clear prompts, readable output.

---

9) Extensions (Optional)

- Add new device types: **ColorBulb** (implements `IDimmable` + `IColorControl`), **Humidifier**.
- Add scheduling (simple time slots) to auto power devices.
- Display a small dashboard summary (counts, devices On vs Off).

---

11) Deliverables & Constraints

- **Runtime:** .NET 8 or 9 Console App.
- **Structure:** single solution; folders as in “Architecture”.
- **Timebox:** ~1-2 days.
- **Conventions:** nullable enabled; PascalCase for types/members; guard input; use `int.TryParse/double.TryParse`.

What the Reviewer Should Look For

- Clear separation: **base class vs capability interfaces**.
- Polymorphic operations over `List<SmartDevice>` (no type checks where unnecessary).
- Input validation and non-crashing flow.
- Status strings reflect device-specific state (brightness, target temp, load/energy).

Example 4

Practice Brief: “MiniBank Console”

1) Overview

Build a simple **console-based banking app** that manages **accounts** (Checking, Savings, Loan).
Users can create accounts, deposit/withdraw, view balances, and run **month-end processing** that behaves differently per account type—demonstrating **interfaces**, **inheritance**, and **polymorphism**.

---

2) Learning Goals

- Model a small domain with an **abstract base class** and **derived types**.
- Define and implement **interfaces** to express optional capabilities (e.g., interest, overdraft).
- Invoke behavior **polymorphically** (e.g., month-end processing across mixed accounts).
- Practice input validation and clear console UX.

---

3) Requirements

3.1 Functional

1. Maintain an in-memory list of bank accounts.
2. Supported types: **CheckingAccount**, **SavingsAccount**, **LoanAccount**.
3. Menu:
    a. List accounts (id, owner, type, balance).
    b. Create account (choose type + owner name + initial deposit or loan amount).
    c. Deposit.
    d. Withdraw.
    e. View account details (recent operations).
    f. Run **Month-End Processing** for all accounts.
    g. Exit.
4. Rules (beginner-friendly):
    - **Checking**: allows overdraft up to a limit (e.g., -200).
    - **Savings**: no overdraft; earns monthly interest (e.g., 1% of balance if balance > 0).
    - **Loan**: balance is negative (money owed). **Deposit** = repayment (moves toward 0). **Withdraw** = borrow more (balance becomes more negative); monthly interest accrues on owed amount.
5. All inputs validated (amounts > 0, account ids exist, etc.).

3.2 Non-Functional

1. Use an **abstract base class** `BankAccount` for shared state/behavior.
2. Use **interfaces** for optional capabilities:
    - Interest accrual
    - Overdraft policy
    - Statement printing
3. Use **polymorphism**: run month-end across all accounts via interface/base references.
4. Standard .NET only; beginner-friendly code and messages.

3.3 Interfaces Overview

Interface	Purpose	Members	Implemented By	Notes
`ITransactable`	Common money operations	`void Deposit(decimal amount)`, `bool Withdraw(decimal amount, out string? error)`	All accounts	Unified deposit/withdraw API.
`IInterestBearing`	Monthly interest/fees	`void ApplyMonthlyInterest()`	Savings, Loan	Savings: positive interest; Loan: interest on debt.
`IOverdraftPolicy`	Overdraft rules	`decimal OverdraftLimit { get; }`	Checking	Enables controlled negatives.
`IStatement`	Print recent ops	`void PrintStatement()`	All accounts	Keeps beginner-friendly history.

---

4) Architecture & Patterns

- **Abstract Base:** `BankAccount` holds `Id`, `Owner`, `Balance`, operation log, and basic deposit/withdraw scaffolding.
- **Derived:** `CheckingAccount`, `SavingsAccount`, `LoanAccount` override rules and implement capability interfaces.
- **Patterns:**
    - **Template-ish** withdraw flow with hooks/overrides for rules.
    - **Strategy via interfaces** (`IInterestBearing`, `IOverdraftPolicy`) to vary behavior by capability.
    - **Polymorphism** for month-end and statements via interface/base refs.

Structure

```
MiniBank/
  Program.cs
  Models/
      BankAccount.cs
      CheckingAccount.cs
      SavingsAccount.cs
      LoanAccount.cs
      Interfaces/
           ITransactable.cs
           IInterestBearing.cs
           IOverdraftPolicy.cs
           IStatement.cs
  Services/
       AccountRegistry.cs
```

---

5) UI Specification (Console Menu)

Example interaction:

```
=== MINIBANK ===
1. List accounts
2. Create account
3. Deposit
4. Withdraw
5. View statement
6. Run month-end
7. Exit
Select: 2
Type (Checking/Savings/Loan): Savings
Owner: Alice
Opening deposit: 500
Created #1 Savings for Alice with BAL ¤500.00

Select: 6
Month-end applied (interest/fees)
```

Validation

- Amounts must be `> 0`.
- Withdraw rules enforced per account type (overdraft, no overdraft, borrow).
- Account id must exist (graceful errors).

---

6) Testing Plan (Manual Only)

- Create each account type; verify balances after deposits/withdraws.
- Checking: attempt to exceed overdraft expect error.
- Savings: run month-end; balance increases by 1% when positive.
- Loan: deposit reduces debt; withdraw increases debt; month-end adds interest.
- Statement lists operations in order.

---

7) Acceptance Criteria

- **Given** multiple accounts, **when** “Run month-end” is executed, **then** all `IInterestBearing` accounts apply interest via a single polymorphic pass.
- **Given** a Checking account, **when** withdrawing within overdraft limit, **then** balance may go negative but not below `OverdraftLimit`.
- **Given** a Savings account, **when** withdrawing beyond balance, **then** operation fails with a friendly message.
- **Given** a Loan account, **when** depositing, **then** the balance moves toward zero; **when** withdrawing, **then** debt increases.
- **Given** invalid input, **then** the app shows an error and returns to the menu without crashing.

8) Evaluation Rubric (10 pts)

- **Correctness (4):** Menu flows; rules enforced; statements/interest correct.
- **OOP Concepts (3):** Clear base class + capability interfaces; polymorphic month-end.
- **Code Quality (2):** Naming, structure, minimal duplication.
- **UX (1):** Clear prompts and formatted amounts.

---

9) Extensions (Optional)

- Add **Transfer** between accounts (guards: no overdraft violations).
- Add **FixedDepositAccount** (locked funds, penalty on early withdrawal).
- Persist accounts to a **JSON file** (save/load).
- Add simple **authentication** (customer id) and filter lists by owner.
- Currency/locale formatting options.

---

10) Deliverables & Constraints

- **Runtime:** .NET 8 or 9 Console App.
- **Timebox:** ~1-2 days.
- **Conventions:** PascalCase for types/members; `decimal` for money; `TryParse` for input; guard clauses.

What the Reviewer Should Look For

- Interfaces accurately model **capabilities** (`IInterestBearing`, `IOverdraftPolicy`) separate from the base class.
- Polymorphic month-end processing over a mixed `List<BankAccount>`.
- Correct enforcement of overdraft and loan rules.
- Clear, readable console UX and operation logs.

Example 5

Practice Brief: “Drone Fleet Console”

1) Overview

Build a console-based **drone fleet manager** that controls different drone types: **SurveyDrone**, **DeliveryDrone**, **RacingDrone**.
Users can register drones, perform capability-specific actions (e.g., **take photos**, **load cargo**, **set waypoints**), and run a **pre-flight check** across the fleet.
Emphasis: **interfaces**, **inheritance**, **polymorphism**.

---

2) Learning Goals

- Model shared behavior via an **abstract base class** and specialized behavior via **interfaces**.
- Invoke behavior **polymorphically** over a mixed list of drones.
- Apply **interface segregation** (capabilities only where needed).
- Practice simple console I/O and input validation.

---

3) Requirements

3.1 Functional

1. Maintain an in-memory list of drones.
2. Supported types:
    - **SurveyDrone** — can take photos.
    - **DeliveryDrone** — can carry cargo (load/unload).
    - **RacingDrone** — fast, minimal features.
3. Menu:

a. List drones.
b. Add drone (choose type + name).
c. Pre-flight check (polymorphic self-test for all).
d. Take off / land a selected drone.
e. Set waypoint (lat, lon) for a selected drone.
f. Capability actions:
    - SurveyDrone take photo.
    - DeliveryDrone load/unload cargo.
g. Charge battery of a selected drone.
h. Exit.

4. Validation:
    - Battery must be 20% to take off (beginner-friendly rule).
    - Waypoints accepted for drones **implementing navigation** capability only.
    - Cargo weight non-negative and within drone capacity.
    - Input must not crash the app.

3.2 Non-Functional

1. **Interfaces** define capabilities: power/flight, navigation, photo capture, cargo handling, self-test.
2. **Abstract base** `Drone` holds `Id`, `Name`, `BatteryPercent`, `IsAirborne`, and common methods.
3. Use **polymorphism** via a unified `List<Drone>` and interface checks for capability-driven menus.
4. Standard .NET only; simple, readable code and messages.

3.3 Interfaces Overview

Interface	Purpose	Members	Implemented By
`ISelfTest`	Pre-flight diagnostics	`bool RunSelfTest()`	All drones (through base)
`IFlightControl`	Flight operations	`void TakeOff(); void Land();`	All drones (through base)
`INavigable`	Waypoint navigation	`void SetWaypoint(double lat, double lon); (double lat, double lon)? CurrentWaypoint { get; }`	SurveyDrone, DeliveryDrone
`IPhotoCapture`	Imaging	`void TakePhoto(); int PhotoCount { get; }`	SurveyDrone
`ICargoCarrier`	Cargo ops	`double CapacityKg { get; } double CurrentLoadKg { get; } bool Load(double kg); void UnloadAll();`	DeliveryDrone

---

4) Architecture & Patterns

- **Abstract Base:** `Drone` with shared identity, battery, and flight state; implements basic `IFlightControl` and `ISelfTest`.
- **Derived Types:** `SurveyDrone`, `DeliveryDrone`, `RacingDrone` add capability interfaces as needed.
- **Interface Segregation:** Capabilities are opt-in via interfaces.
- **Polymorphism:** Pre-flight check and listing operate on `Drone`; capability menus branch by interface presence (not by concrete type).
- **Simple Factory (optional):** Create drones from user input to keep `Program` tidy.

Suggested Layout

```
DroneFleet/
  Program.cs
  Models/
      Drone.cs
      SurveyDrone.cs
      DeliveryDrone.cs
      RacingDrone.cs
      Interfaces/
           ISelfTest.cs
           IFlightControl.cs
           INavigable.cs
           IPhotoCapture.cs
           ICargoCarrier.cs
  Services/
       DroneFactory.cs (optional)
```

---

5) UI Specification (Console Menu)

Example flow:

```
=== DRONE FLEET ===
1. List drones
2. Add drone
3. Pre-flight check (all)
4. Take off / Land
5. Set waypoint
6. Capability actions
7. Charge battery
8. Exit
Select: 2
Type (Survey/Delivery/Racing): Delivery
Name: Courier-01
Added #1 DeliveryDrone "Courier-01"

Select: 6
Enter id: 1
Actions: [Load, UnloadAll]  (for DeliveryDrone)
Load kg: 2.5
Loaded 2.50 kg.

Select: 4
Enter id: 1
Courier-01 took off. Battery 95%
```

Validation Rules

- Waypoint only if drone `is INavigable`.
- Photo only if drone `is IPhotoCapture` **and** airborne.
- Load/Unload only if drone `is ICargoCarrier`.
- Charge accepts `0–100` additional percent (clamped); battery max 100%.

---

6) Testing Plan (Manual Only)

- Add each type; list shows correct status.
- Run pre-flight: drones with battery < 20 fail (message).
- Take off/land flow respects battery and state.
- Set waypoint works for Survey/Delivery; should not appear for Racing.
- Take photos only while airborne on SurveyDrone.
- DeliveryDrone loading respects capacity; unload resets to 0.
- Charging clamps to 100%.

---

7) Acceptance Criteria

- **Given** a mixed fleet, **when** “Pre-flight check” runs, **then** each drone prints a pass/fail using `ISelfTest` via a polymorphic loop.
- **Given** a selected drone, **when** attempting actions it does **not** support, **then** the menu hides those actions or prints a friendly message.
- **Given** a SurveyDrone airborne, **when** `TakePhoto()` is called, **then** `PhotoCount` increments and a message is shown.
- **Given** a DeliveryDrone with capacity 5 kg, **when** loading beyond capacity, **then** the operation is rejected with a message.
- **Given** a RacingDrone, **when** setting waypoint, **then** the app disallows it (no `INavigable`).

---

8) Evaluation Rubric (10 pts)

- **Correctness (4):** Menu works; state transitions valid; constraints enforced.
- **OOP Concepts (3):** Clear abstract base + capability interfaces; polymorphic operations.
- **Code Quality (2):** Readable, small methods, no giant `if-else` for types where interfaces suffice.
- **UX (1):** Clear prompts, formatted status, helpful errors.

---

9) Extensions (Optional)

- Add **Battery drain** while airborne per action/tick; auto-land at 5%.
- Add **Geo-fencing** validation for waypoints.
- Add **Photo gallery count** per flight session.
- Add **HeavyDeliveryDrone** with higher capacity and different takeoff battery rule.

---

10) Deliverables & Constraints

- **Runtime:** .NET 8 or 9 Console App.
- **Timebox:** ~1-2 days.
- **Conventions:** PascalCase for types; `TryParse` for numbers; guard clauses.

What the Reviewer Should Look For

- Interfaces model **capabilities** (navigation, photo, cargo) rather than types.
- Polymorphic loops over `List<Drone>` for self-test and listing.
- Clean separation of concerns (base for shared logic).
- Input validation that prevents crashes and enforces simple domain rules.

Example 6 - Book Club Quiz Challenge

Practice Brief: “Deep Thought”

1) Overview

Build a small console program that acts like ***Deep Thought*** from the book *The Hitchhiker’s Guide to the Galaxy*. Users type a “Ultimate Question,” choose a simple algorithm, and the program “computes” an answer (e.g., “42”) while showing progress. Finished jobs and answers are saved to a local JSON file so results persist across runs.

---

2) Learning Goals

- Basic console input/output and menu loops.
- Organizing code with classes, interfaces, and the **Strategy** pattern.
- Asynchronous methods with async/await and CancellationToken.
- Simple validation and error messages.
- Reading/writing JSON files for persistence.
- Writing basic unit tests with xUnit/NUnit

---

3) Requirements

3.1 Functional

1. On start, display a menu:
    - (1) Submit Question
    - (2) List Jobs
    - (3) View Result by JobId
    - (4) Cancel Running Job
    - (5) Exit
2. **Submit Question**:
    - Prompt for QuestionText (1–200 chars).
    - Prompt for Algorithm choice: Trivial, SlowCount, or RandomGuess.
    - Create a **Job** with a new JobId (GUID), set Status=Pending, and save to disk.
3. **Run Job immediately** after submission (single job at a time).
    - Show progress updates in console (e.g., Progress: 0% ... 100%).
    - Produce an **Answer** string and set Status=Completed, saving to disk.
4. **Cancellation**:
    - While a job runs, allow user to press C to cancel. Canceled jobs get Status=Canceled and Progress<100%.
5. **List Jobs**:
    - Print: JobId | Status | Algorithm | CreatedUtc | Progress.
6. **View Result**:
    - Given a JobId, show { Answer, Summary, DurationMs } if Completed.
    - If not found, show a friendly message.
7. **Persistence**:

- All jobs are stored in deepthought-jobs.json. On startup, the app loads existing jobs.

8. **Validation**:
    - Reject empty or too-long questions; reject unknown algorithms.

3.2 Non-Functional

1. Use clear method names, small classes, and comments where helpful.
2. Do not crash on invalid input—re-prompt or show a message and return to menu.
3. Keep code easy to read (regions or small files OK).

4) Architecture & Patterns

- **Project structure (simple)**
    - MiniDeepThought (console app)
    - MiniDeepThought.Tests (xUnit/NUnit)
- **Core parts**
    - IAnswerStrategy with three implementations:
        - TrivialStrategy: returns "42" quickly.
        - SlowCountStrategy: loops from 1..N with small delays, reports progress, returns "42" at the end.
        - RandomGuessStrategy: “thinks” for a bit, returns a random number as string and a short summary. Random numbers should be generated from the following list: [42]
    - Job entity with JobId, QuestionText, AlgorithmKey, Status, Progress, timestamps, and optional Result.
    - JobStore for JSON load/save (all in one file).
    - JobRunner that runs one job on the main thread using async/await, accepts CancellationToken, updates progress, and saves.
- **Why Strategy?** Lets beginners add/swap algorithms without changing the runner.

---

5) UI Specification (Console Menu)

- **Menu loop** (pseudo):

1) Submit 2) List 3) View 4) Cancel 5) Exit

Select: _

- **Submit flow**
    - Ask: Enter your Ultimate Question (1-200):
    - Ask: Algorithm [Trivial|SlowCount|RandomGuess]:
    - Print: Job queued: {JobId}
    - Start running it immediately; show progress like Progress: 0% … 100%.
    - Tip text: Press 'C' to cancel.
- **List flow**
    - Prints rows: JobId | Status | Algorithm | CreatedUtc | Progress%
- **View flow**
    - Ask JobId:; if Completed, show:
    - {
    - "jobId": "...",
    - "answer": "42",
    - "summary": "Because reasons.",
    - "durationMs": 5321
    - }
- **Cancel flow**
    - If a job is currently running, stop it and mark Canceled.

Validation rules

- Question trimmed length 1..200.
- AlgorithmKey in { Trivial, SlowCount, RandomGuess }.

Status messages

- Success show a short confirmation.
- Not found / invalid input show a friendly line and return to menu.

---

6) Testing Plan (Optional) XUnit/NUnit

- **Strategy tests**
    1. TrivialStrategy_Returns42_AndReportsProgress().
    2. SlowCountStrategy_AdvancesProgress_To100().
    3. RandomGuessStrategy_ProducesNumberString().
- **Cancellation test**
  4. SlowCountStrategy_HonorsCancellation_BeforeCompletion().
- **Store tests** (can use temp file)
  5. JobStore_SavesAndLoads_JobsRoundTrip().

Hints: Factor delays into a configurable IClock or TimeSpan delayPerStep parameter for faster tests (or reduce delays in test).

---

7) Acceptance Criteria

1. **Submit & Complete**
   - Given the app is running
   - When I submit a valid question with Trivial
   - Then I see a JobId, progress reaches 100%, and the job is saved as Completed with Answer="42".
2. **Cancel Running Job**
   - Given a SlowCount job is running
   - When I press C
   - Then the job stops, Status=Canceled, Progress<100%, and is saved.
3. **View Result**
   - Given a completed job
   - When I input its JobId in View Result
   - Then I see Answer, Summary, and DurationMs.
4. **List Jobs**
   - Given at least one job exists
   - When I choose List Jobs
   - Then I see a table with correct statuses and progress.

---

8) Evaluation Rubric (10 pts)

- Correctness & Requirements (4) — menu flows, progress, cancel, persistence.
- Code Organization (2) — Strategy pattern, small classes, meaningful names.
- Async & Cancellation (1) — proper async/await, CancellationToken usage.
- Testing (1) — at least 3 passing unit tests.
- Validation & UX (1) — input checks, helpful messages.
- Documentation (1) — short README with run steps.

---

9) Extensions (Optional)

1. Save a tiny **audit log** per job (messages and timestamps) in JSON.
2. Add a **"Resume last incomplete job"** menu option.
3. Show a **spinner** during computation.
4. Add a **fourth algorithm** that combines two strategies (compose answers).

---

10) Deliverables & Constraints

Runtime: .NET 8 or 9 Console App.

Timebox: ~1-2 days.

Conventions: PascalCase for types; `TryParse` for numbers; guard clauses.

Repo structure

/MiniDeepThought

/src/MiniDeepThought

Program.cs

Domain/Job.cs

Domain/JobResult.cs

Strategies/IAnswerStrategy.cs

Strategies/TrivialStrategy.cs

Strategies/SlowCountStrategy.cs

Strategies/RandomGuessStrategy.cs

Services/JobRunner.cs

Services/JobStore.cs

Util/ConsoleHelpers.cs

/tests/MiniDeepThought.Tests

TrivialStrategyTests.cs

SlowCountStrategyTests.cs

RandomGuessStrategyTests.cs

JobStoreTests.cs

README.md

What the Reviewer Should Look For

- Clear menu and user flow; sensible prompts and messages.
- Strategy pattern correctly implemented and selected by key.
- Progress updates occur during computation; cancellation works.
- Jobs persist to and load from JSON correctly.
- Basic unit tests exist and pass.
- Code is tidy, readable, and reasonably commented.