Agenda

1. Welcome, Objectives, Quick .NET 8 refresher
2. OOP Fundamentals: Intro, Classes vs Objects, Members
3. Break
4. Properties, `this`, static, constants/readonly
5. Lunch
6. Encapsulation & Access Modifiers, Properties & Encapsulation
7. Inheritance, `base`, Method overriding, `new`, `sealed`
8. Break
9. Abstract classes vs Interfaces, explicit vs implicit implementation, multiple interfaces
10. Hands-on: guided lab, Q&A, End-of-day assignment briefing

---

OOP FUNDAMENTALS

1) Introduction to OOP

- Definition: Programming paradigm organizing code into objects that combine state and behavior.
- Theory: Encapsulation, abstraction, inheritance, polymorphism enable maintainable, extensible designs.
- Use: Model real-world domain, separate concerns, enable tests & reuse.
- Pro tip: Think of objects as nouns in your domain; methods are verbs.
- Watch out: Over-modeling (creating unnecessary classes) increases complexity.

Class from Part-of-Speech Mapping 📚🔧

> This document shows how English parts of speech map to common OOP elements, highlights a moderately complex sentence by part of speech, and provides a matching C# class using the same color-coding to reinforce the concept.

Mapping Table

Part of Speech	Maps to (OOP)	Typical Example	Color
Noun / Noun phrase	Class name, property (state)	`CourierVan`, `Package`, `Location`	Noun (Blue)
Verb	Method (behavior / action)	`Navigate()`, `Deliver()`	Verb (Green)
Adjective	Property descriptor (flags, types) or enum values	`IsFragile`, `Refrigerated`	Adjective (Orange)
Adverb	Method modifier / naming hint (how action is performed)	`NavigateQuickly()`	Adverb (Purple)
Preposition	Relationship indicator (from, to, via) used in property/method names	`FromWarehouse`, `ToCustomer`	Preposition (Teal)
Pronoun	Method parameter or local variable	`customer`, `it`	Pronoun (Brown)

Moderately Complex Sentence (color-coded)

Below is a sentence describing a delivery scenario. Each word is wrapped with a color that represents its part of speech.

The experienced courier van, equipped with a refrigerated compartment, swiftly navigated the congested downtown streets to pick up a fragile package from the central warehouse and deliver it to the customer's fourth-floor apartment before the evening rush.

> Legend: Noun • Verb • Adjective • Adverb • Preposition

Derived C# Class (concept mapping)

Below is a C# class that maps the sentence's key parts-of-speech to class elements. Use the colors to visually connect words in the sentence to elements in the class.

Colorized view (HTML, purpose: visual reinforcement)

```
public class CourierVan
{
    public string Id { get; set; }                              // Noun: unique identifier
    public bool IsRefrigerated { get; private set; }   // Adjective: refrigerated
    public string CompartmentType { get; set; }          // Noun
    public Location CurrentLocation { get; private set; }

    public void Navigate(string route)              // Verb
    {
        // perform navigation swiftly (adverb)
        PerformNavigation(route);
    }

    public void PickUp(Package package)            // Verb + Pronoun/parameter
    {
        Load(package);
    }

    public void Deliver(Location toLocation)         // Verb + Preposition-as-name
    {
        Unload(toLocation);
    }
}

public class Package { public bool IsFragile { get; set; } }
public class Location { public string Address { get; set; } }
```

Plain C# (copyable)

```
public class CourierVan
{
    public string Id { get; set; }
    public bool IsRefrigerated { get; private set; }
    public string CompartmentType { get; set; }
    public Location CurrentLocation { get; private set; }

    public void Navigate(string route)
    {
        // 'swiftly' is an adverb hint - ensure this does not change the method signature
        PerformNavigation(route);
    }

    public void PickUp(Package package)
    {
        Load(package);
    }

    public void Deliver(Location toLocation)
    {
        Unload(toLocation);
    }
}

public class Package
{
    public bool IsFragile { get; set; }
}

public class Location
{
    public string Address { get; set; }
}
```

Pro Tips 💡

- Map nouns to classes and properties—these represent things and state.
- Map verbs to methods—these represent actions or behaviors.
- Use adjectives for descriptive property names (flags, enum entries).
- Use adverbs as behavior modifiers (they usually inform method implementation or naming but rarely change signatures).

Practice Exercise ✅

1. Pick three sentences from a domain (e.g., banking, e-commerce, HR).
2. Highlight parts of speech following this color mapping.
3. Design one class per sentence, aligning nouns to class/properties and verbs to methods.

Conclusion

This file ties linguistic structure to object-oriented design: part-of-speech → OOP element helps novices model requirements more naturally. Use the color mapping as a quick mental model when turning textual requirements into classes.

---

2) Classes vs Objects

- Definition: Class = blueprint/type. Object (instance) = runtime representation with state.
- Syntax:

```
class Person { public string Name { get; set; } }
var p = new Person { Name = "Ada" }; // object
```

- Example: `class Car` describes properties and behavior; `new Car()` is an object.
- Pro tip: Keep classes focused (single responsibility).
- Pitfall: Confusing static behavior (type-level) vs instance behavior (object-level).

---

3) Classes — Members (Fields, Properties, Methods, Constructors)

- Fields: backing storage; prefer private fields.

```
private int _count;
```

- Properties: public surface for state (with validation).

```
public int Count { get; private set; }
```

- Methods: behavior; keep methods small & focused.
- Constructors: init state; use overloaded constructors and `this` chaining.

```
public class Box {
  public Box(int size) { Size = size; }
  public int Size { get; }
}
```

- Pro tip: Prefer properties over public fields for versioning and validation.
- Pitfall: Large constructors doing work — prefer factory methods or builders.

---

4) Properties

- Auto-implemented:

```
public string Name { get; set; }
```

- Calculated (computed):

```
public int Area => Width * Height;
```

- Expression-bodied:

```
public string FullName => $"{First} {Last}";
```

- `init` properties (immutable after construction):

```
public string Id { get; init; }
```

- Pro tip: Readonly computed props are great for derived state.
- Pitfall: Computed properties that do heavy work or have side effects — should be methods.

---

5) this Keyword

- Definition: Reference to current instance.
- Usage:
  - Distinguish parameter vs field: `this.name = name;`
  - Constructor chaining: `public Person(string name) : this() { ... }`
- Pro tip: Use `this` for clarity in constructors and fluent APIs.
- Pitfall: Avoid long chains that make `this` confusing; prefer explicit naming.

---

6) Static Members and Static Classes

- Static members belong to the type, not instance.

```
public static class MathUtils { public static double Pi = 3.14; }
```

- Use for stateless utilities or caches.
- Pro tip: Keep static state immutable or thread-safe.
- Pitfall: Mutable static state → concurrency bugs and test interference.

---

7) Constants and Readonly Fields

- `const` (compile-time constant): implicitly static.

```
public const int Max = 100;
```

- `readonly` (runtime immutability after ctor):

```
public readonly DateTime Created = DateTime.UtcNow;
```

- Pro tip: Use `const` for primitives and `readonly` for runtime-determined values.
- Pitfall: `const` is embedded by callers—changing library `const` is breaking for consumers.

OOP PILLARS

8) Encapsulation

- Definition: Protect object's internal state behind a well-defined interface.
- Techniques: private fields, properties with validation, internal helper methods.
- Example:

```
private int _balance;
public void Deposit(int amount) { if(amount<=0) throw ...; _balance += amount; }
```

- Pro tip: Expose behavior, not internal data.
- Pitfall: Making everything public "for convenience" breaks invariants.

---

9) Access Modifiers (in detail)

- `public`, `private`, `protected`, `internal`, `protected internal`, `private protected`
- Effects:
  - `private` only inside type
  - `protected` inside type and derived types
  - `internal` within same assembly
  - combinations for finer control
- Pro tip: Start restrictive (private) and relax as needed.
- Pitfall: Overusing `internal` for testing — prefer InternalsVisibleTo if necessary.

---

10) Properties and Encapsulation

- Use properties to validate and maintain invariants.

```
private int _age;
public int Age {
  get => _age;
  set => _age = value < 0 ? 0 : value;
}
```

- Pro tip: Favor read-only properties for immutable data models.
- Pitfall: Large property setters doing I/O or logic—use methods instead.

---

11) Inheritance

- Definition: Reuse and extend behavior (is-a relationship).
- Syntax:

```
class Animal { public virtual void Speak() { } }
class Dog : Animal { public override void Speak() => Console.WriteLine("Woof"); }
```

- Pro tip: Use inheritance for clear is-a semantics; prefer composition for reuse otherwise.
- Pitfall: Fragile base class changes can break derived classes.

---

12) Base Classes and `base` keyword

- `base` calls base constructor/method.

```
public class A { public virtual void M() {} }
public class B : A { public override void M() { base.M(); ... } }
```

- Pro tip: Call `base` when extending behavior—especially in constructors to initialize base state.

- Pitfall: Calling virtual methods in base constructors can call derived overrides unexpectedly.

---

13) Method Overriding (`virtual`, `override`, `new`)

- `virtual` marks base method overrideable.
- `override` changes behavior in derived.
- `new` hides base method (not polymorphic).

```
class Base { public virtual void X() => Console.WriteLine("B");}
class D : Base { public override void X() => Console.WriteLine("D"); }
class H : Base { public new void X() => Console.WriteLine("H"); }
```

- Pro tip: Prefer `override` for polymorphism; avoid `new` unless hiding intentionally.
- Pitfall: Using `new` may surprise consumers who hold the instance typed as base.

---

14) `sealed` Classes and Methods

- `sealed` prevents further inheritance.

```
public sealed class Utility { }
public class Base { public virtual void M() {} }
public class Derived : Base { public sealed override void M() {} }
```

- Pro tip: Seal classes/methods when you want simple, stable types and to enable compiler optimizations.
- Pitfall: Over-sealing prevents legitimate extensibility.

---

15) Abstract Classes and Abstract Methods

- Abstract type with contract and shared behavior.

```
public abstract class Shape { public abstract double Area(); public void Log() => ... }
public class Circle : Shape { public override double Area() => Math.PI*r*r; }
```

- Pro tip: Use when you have shared code and partial contract but no complete implementation.
- Pitfall: Avoid abstract classes when multiple disparate behaviors should be implemented (use interfaces).

---

16) Interfaces

- Pure contract (can have default implementations since C# 8+, but keep minimal).

```
public interface IRepository<T> {
  void Add(T item);
  T? Get(int id);
}
```

- Pro tip: Use interfaces for behavior abstraction, especially for DI and testing.
- Pitfall: Excessively granular interfaces cause many tiny types—follow Interface Segregation.

---

17) Interface Implementation (Implicit and Explicit)

- Implicit:

```
public class Repo : IRepository<Item> {
  public void Add(Item i) { ... }
  public Item? Get(int id) => ...;
}
```

- Explicit:

```
public class Printer : IPrinter {
  void IPrinter.Print() => Console.WriteLine("explicit");
}
```

  - Explicit hides method unless the instance is cast to interface.
- Pro tip: Use explicit to avoid exposing interface members that conflict or clutter the public surface.
- Pitfall: Explicit methods invisible unless cast—can confuse callers.

---

18) Multiple Interfaces

- C# supports implementing multiple interfaces:

```
public interface IReadable { string Read(); }
public interface IWritable { void Write(string s); }
public class File : IReadable, IWritable { ... }
```

- When two interfaces have the same member signature, use explicit implementation to disambiguate.
- Pro tip: Combine small behavior interfaces to create flexible APIs.
- Pitfall: Diamond problem avoided since no multiple base class inheritance, but method conflicts require resolution.

Hands-on Labs (Guided, 15–30 min each) 🧪

Lab 1 — Classes & Objects:

- Build `Person` class with fields & properties, constructors (overloaded), and `ToString()` override.
- Validate with a small console app.

Lab 2 — Properties & Validation:

- Create `BankAccount` with `Balance` (private set) and `Deposit`, `Withdraw` methods, and computed `IsOverdrawn` property.

Lab 3 — Inheritance & Polymorphism:

- Implement `Animal` -> `Dog`, `Cat`, using virtual `Speak()` and demonstrate polymorphism via `List<Animal>`.

Lab 4 — Interfaces & DI:

- Create `IRepository<T>` and implement `InMemoryRepository<T>`. Unit test behavior with xUnit & DI.

Lab 5 — Abstract vs Interface:

- Implement `Shape` (abstract) for `Circle` & `Rectangle`. Also create `ITransformable` interface and implement on shapes.

Each lab includes tests, a README, and short reflection questions.

End-of-Day Assignment (Project) — "Fleet Manager" 🚙 📦

Build a console or minimal web (Blazor/Console) app modeling a vehicle fleet.

Requirements:

- Core domain:
  - Base `Vehicle` (abstract) with `Id`, `Make`, `Model`, `Start()`, `Stop()`, `FuelLevel`.
  - Derived classes: `Car`, `Truck` (overrides `Start`).
  - Interfaces: `IMaintainable` (ScheduleMaintenance), `ITrackable` (GetLocation).
- Use properties (auto, expression-bodied), `readonly` fields, and `static` helper for ID generation.
- Provide encapsulated validation (e.g., fuel cannot drop below 0).
- Demonstrate:
  - Polymorphism (List)
  - Interface usage for maintenance & tracking
  - Explicit interface implementation where methods conflict
- Add unit tests (xUnit or NUnit) for core behaviors.

Deliverables:

- Source code, unit tests, README with design decisions, list of known limitations.
- Optional: small demo video or gifs.

Rubric:

- Design (30%) — clean domain model, good use of OOP pillars
- Correctness (30%) — tests pass, behavior matches requirements
- Code Quality (20%) — readable, documented, small methods
- Bonus (20%) — CLI or tiny UI, DI & tests, concurrency-safe static members

---

Pro Tips & Best Practices 💡

- Prefer composition over inheritance when reusing behavior.
- Keep methods small; one responsibility each.
- Use interfaces for testability and inversion of control.
- Avoid mutable static state; prefer dependency-injected singletons when necessary.
- Immutable models (init-only props, records) simplify reasoning and concurrency.
- Write unit tests for invariants and polymorphic behavior.

Common Pitfalls & Risks ⚠️

- Exposing mutable internal state (public fields).
- Overusing inheritance (tight coupling, fragile base class).
- Using `new` to hide members and causing surprising behavior.

- Thread-unsafe static mutable fields leading to race conditions.
- Breaking binary compatibility by changing `const` values.

References & Further Reading 📚

- Microsoft Docs — C# Programming Guide (Types, Classes, Interfaces)
- “Effective C#” style guides and patterns
- Design Patterns (Gang of Four) — for applying OOP constructs
- .NET 8 release notes for language/runtime features

Quick Wrap-up ✅

- You now have: definitions, concise code patterns, pro tips, pitfalls, labs, and a solid end-of-day assignment to practice everything.
- Suggested next steps: implement the Fleet Manager, write tests, and schedule a code review session.