1. Hvorfor skal din “type” være en “value type”?

En value type (struct) er ideel i dette scenarie, fordi den er et lille dataobjekt, der gemmes direkte på stacken som en kopi.
Det betyder, at hver gang du arbejder med den, kopieres den, og ændringer i den ene kopi påvirker ikke de andre. Desuden håndteres den automatisk, når den ikke længere er i brug.

A. Vil din app virke, hvis din type er en “reference type”?

Ja, appen vil stadig fungere, men adfærden i hukommelsen ændrer sig. En reference type (klasse) gemmes på heapen, og variablerne indeholder en reference (en slags pointer) til objektet. Det betyder, at flere variabler kan pege på det samme objekt, og ændringer via en af disse vil påvirke objektet globalt.

B. Hvad er forskellen mellem brugen af value vs. reference “type”?

Value type (struct):
Gemmes direkte på stacken som en kopi.
Ændringer i en kopi påvirker ikke den oprindelige værdi.
Reference type (klasse):
Gemmes på heapen, og variablen indeholder en reference til objektets placering.
Flere referencer kan pege på det samme objekt, så en ændring via én reference påvirker objektet, uanset hvilken variabel der bruges.

2. Konstruktorer

A. Ud over default constructor, hvilket andre konstruktører findes?

Parameteriserede konstruktører: Hvor du kan give startværdier for dine felter/egenskaber.

Copy konstruktører: (Mere almindeligt i andre sprog end C#, men principielt en metode til at lave en kopi af et objekt.)

B. Vil din app virke, hvis din “type” slet ikke implementerer en konstruktør? Og hvis din “type” var en reference-type, vil din app virke, hvis den slet ikke implementerer en konstruktør?

For value types (structs) er en default parameterløs konstruktør altid til stede, selvom du ikke selv skriver den, så appen vil virke.

For reference types (klasser) skal du have en konstruktør – hvis du ikke skriver en, genereres en default parameterløs konstruktør automatisk, så længe du ikke definerer en anden. Appen vil derfor også fungere for reference types uden eksplicit konstruktør, men det er god praksis at have styr på, hvordan dine objekter bliver initialiseret.

3. Hvordan kan man tilgå din type, uden at instantiere den som et objekt?

Man kan tilgå statiske medlemmer (som konstanter eller statiske metoder) uden at lave en instans af typen. Det betyder, at du direkte kan skrive typenavnet efterfulgt af punktum og den statiske medlemsnavn, uden at skulle oprette et objekt.

4. Hvorfor er det ikke en god idé at skrive til eller læse fra konsollen i din "type" og i stedet gøre dette i din Program.cs?

Det er bedst at holde forretningslogikken (din type) adskilt fra brugergrænsefladen (konsol I/O).
Hvis du blander dem, bliver koden sværere at teste og vedligeholde.
Det gør det også sværere at genbruge din type i andre sammenhænge, hvor konsol I/O ikke er relevant.

5. Dependency Injection (DI)

Krav: Du skal anvende dependency injection til instansiering af dine "types" så vidt, det er muligt.

A. Har du oplevet problemer, når din "custom value type" skal instantieres med DI? Og hvordan løser du problemet?

Value types kan være tricky med DI, fordi de kopieres ved hver overførsel. Det betyder, at man kan ende med flere kopier i stedet for at dele én enkelt instans.
Løsningen kan være at bruge DI primært til reference types, eller hvis du bruger value types, skal du være opmærksom på, at de ikke opfører sig som delte objekter. Alternativt kan du overveje at lave din type til en klasse, hvis du har brug for, at den deles.

B. Undersøg, hvor mange instantierings-teknikker der findes, og beskriv dem i din dokumentation?

De mest almindelige teknikker er:

Constructor injection: Afhængigheder leveres via konstruktøren.

Property injection: Afhængigheder sættes via offentlige egenskaber.

Method injection: Afhængigheder sendes ind som parametre til metoder.
Constructor injection er ofte at foretrække, fordi den sikrer, at alle nødvendige afhængigheder er til stede fra starten.

6. DTO vs. Tuple

I opgaven anvendes en DTO-klasse som retur-type for din metode, fordi den skal returnerer flere værdier af forskellige typer - En anden teknik er f.eks. en Tuple. Forklar hvilken returtype er bedst til dette formål:

DTO (Data Transfer Object):
Fordelen ved en DTO er, at den har navngivne felter, som gør det klart, hvad hver værdi repræsenterer.
Det gør koden mere læsbar og vedligeholdelsesvenlig.

Tuple:
En Tuple en hurtig løsning, men dens værdier har ikke selvforklarende navne, hvilket kan gøre koden sværere at forstå.

Til dette formål er en DTO ofte foretrække, da den gør det klart, hvad der bliver returneret, og dermed øger koden læsbarhed og vedligeholdelse.