Avancerede typer

I dette kapitel ser vi på lidt flere og lidt mere avancerede features relateret til typer. Hvis du er ny i programmering, kan du vælge at springe dette kapitel over, indtil du er blevet lidt mere erfaren.

Værdibaserede variabler med null-værdier

Som du tidligere har lært, findes der overordnet tre måder at skabe typer, der kan ligge til grund for instanser – klasser, poster og strukturer. Klasser og poster (som i virkeligheden er en autogenereret klasse – se senere i kapitlet) er referencebaserede, og strukturer er værdibaserede. Det har betydning for hvor i hukommelsen værdier placeres. Se i kapitlet om hukommelsesteori på side 207, hvis du vil have en opsummering.

At en variabeltype er referencebaseret betyder, at variabler ikke indeholder værdier, men referencer. Variablen kan dermed tildeles værdien *null* for at indikere, at den ikke indeholder nogen reference. Det kan værdibaserede variabler ikke, fordi de indeholder konkrete værdier.

Her er et eksempel:

```
int i = 10;    // det er ok – 10 er værdien
// i = null;   // fejl – en int kan ikke få værdien null
```

En int er det samme som System.Int32, og da det er en struct, kan den udelukkende indeholder værdier.

Men der kan være situationer, hvor det kunne være smart, at en værdibaseret type kan blive tildelt værdien *null*.

Det kan eksempelvis være en bool, der repræsenterer et valg, en bruger har taget på en brugerflade. Værdien kan være true eller false eller null, hvor null indikerer, at brugeren ikke har foretaget et valg.

Det kunne også være et felt i en klasse, som repræsenterer et felt i en database. I de fleste databaser kan man godt kan oprette et tal-felt, der kan tildeles en null-værdi, og det skal nogle gange kunne beskrives i kode.

Derfor har du mulighed at benytte en anden type i C#, som giver mulighed for null-værdi - den generiske System.Nullable-type.

Her er et eksempel, der benytter typen til at gøre en int *nullable*:

```
Nullable<int> i;
```

Nu kan i benyttes, som var det en int:

```
i = 10;
Console.WriteLine(i);
```

Men fordi det nu er en Nullable<int>, kan den også tildeles null:

```
i = null;
```

og den har fået et par ekstra metoder:

```
i = 10;
Console.WriteLine(i.HasValue);              // true
Console.WriteLine(i.GetValueOrDefault());   // 10
```

Egenskaben HasValue fortæller om instansen har en værdi eller er null, og metoden GetValueOrDefault returnerer en værdi, hvis den findes, og eller returnerer en default værdi (som for en int er 0).

For at gøre det nemt at benytte Nullable variabler, kan du også benytte tegnet ? ved erklæring. Således er

```
Nullable<int> i;
```

det samme som

```
int? i;
```

Det er jo noget nemmere at skrive, men giver præcis den samme type.

Her er et par andre eksempler:

```
int? a = 10;
bool? b = true;
double? c = 1000.33;
char? d = '*';

a = null;
b = null;
c = null;
d = null;

int? e = LægSammen(10, 10);
Console.WriteLine(e);       // 20
e = LægSammen(null, 10);
Console.WriteLine(e);       // null

int? LægSammen(int? a, int? b)
{
    if (!a.HasValue || !b.HasValue)
        return null;
    else
        return a + b;
}
```

int? er det samme som System.Nullable<int>

Bemærk, at typen kan benyttes på alle værdibaserede typer, i argumenter, returværdier og også i andre typer:

```
class Person
{
    public int PersonId { get; set; }
    public string Navn { get; set; }
    public bool? ErDansk { get; set; }
}
```

I klassen benyttes en nullable bool til at beskrive, om en person er dansk (true), ikke dansk (false) eller det ikke er angivet (null).

Referencebaserede variabler med null-værdier

Den lidt erfarne C# udvikler vil undre sig lidt over overskriften ”Referencebaserede variabler med null-værdier”, fordi alle referencebaserede variabler (variabler skabt med udgangspunkt i en klasse og ikke en struktur) jo netop kan indeholde en null-værdi – eksempelvis:

```
class Person
{
    public string Navn { get; set; }
    public string NavnMedStort()
    {
        return Navn.ToUpper();
    }
}
```

Her er tale om en ganske almindelig klasse med en autogenereret egenskab Navn, og en metode som returnerer navnet med store bogstaver. Den kunne eksempelvis bruges som:

```
Person p = new Person();
p.Navn = "Mikkel";
Console.WriteLine(p.NavnMedStort()); // MIKKEL
```

Det fungerer jo fint i dette eksempel, men klassen indeholder en mulig, og faktisk meget udbredt, fejlmulighed. Hvad tror du, der sker, når denne kode afvikles:

```
Person p = new Person();
Console.WriteLine(p.NavnMedStort());
```

Den fejler med et hult drøn med en NullReferenceException, hvilket i virkeligheden er meget logisk. Metoden NavnMedStort forsøger at

afvikle en metode på et objekt, som ikke eksisterer. Navn er jo aldrig tildelt en værdi og har dermed værdien null.

Det kan undgås på flere måder – måske ved at ændre metoden så den returnerer null, hvis Navn er lig med null:

```
class Person
{
    public string Navn { get; set; }
    public string NavnMedStort()
    {
        return Navn?.ToUpper();
    }
}
```

Kan du huske hvad Navn?.ToUpper() betyder?
Hvis ikke, så se side 95

Eller ved at gøre Set-delen af Navn-egenskaben privat, og værdien tildeles gennem en konstruktør:

```
class Person
{
    public string Navn { get; private set; }
    public string NavnMedStort()
    {
        return Navn.ToUpper();
    }
    public Person(string navn)
    {
        Navn = navn ?? "";
    }
}
```

Problemet med mulige null værdier kan løses på mange måder, men det vigtige er, at du er bevidst om eventuelt mulige kommende fejl. I en simpel klasse, er det nemt nok at overskue, men i en kodebase på

mange tusinder linjer kan man hurtigt overse et muligt kommende problem.

Heldigvis kan man bede kompileren om at være opmærksom på eventuelle null-problemer og oplyse om dette i advarsler eller fejl. Du kan enten dekorere koden med direkte besked til kompileren (# direktiver) som følger:

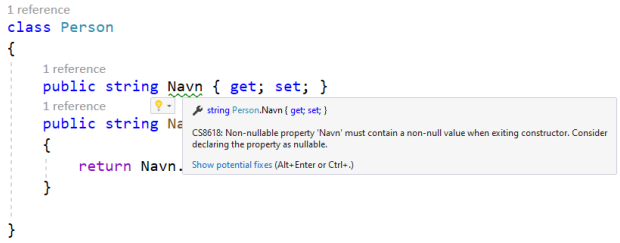
```
#nullable enable
    class Person
    {
        public string Navn { get; set; }
        public string NavnMedStort()
        {
            return Navn.ToUpper();
        }
    }
#nullable restore
```

Eller ved at fortælle kompileren i .csproj-filen (projektfilen), at du ønsker hele projektet kontrolleret for eventuelle fejl:

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net5.0</TargetFramework>
  <Nullable>enable</Nullable>
 </PropertyGroup>
</Project>
```

> Du bør altid anmode kompileren om at foretage kontrol af eventuelle fejl,, der kan opstå ved null-værdier.

Under alle omstændigheder vil kompileren nu fortælle, at der er et problem med Navn.

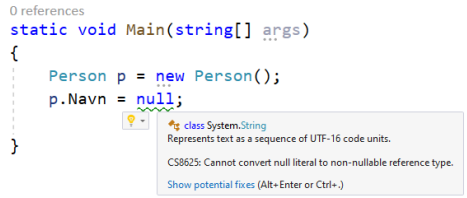


Figur 63 Advarsel om mulig null-fejl

Advarslen fortæller, at Navn bør tildeles en værdi for at undgå eventuelle fejl – eksempelvis:

```
class Person
{
    public string Navn { get; set; } = "";
    public string NavnMedStort()
    {
        return Navn.ToUpper();
    }
}
```

Det løser umiddelbart problemet, men kun indtil du opretter en instans af person og så selv tildeler null til Navn. Men den fanger kompileren også:



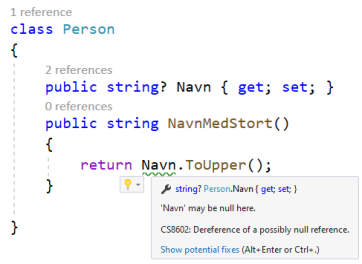
Figur 64 Ny mulighed for fejl ved Null værdi

Du kan også vælge at fortælle kompileren, at Navn gerne må få værdien null, og det gøres på samme måde, som når du erklærer en variabel af en nullable struktur:

```
class Person
{
```

```
    public string? Navn { get; set; }
    public string NavnMedStort()
    {
        return Navn.ToUpper();
    }
}
```

Nu ved kompileren, at Navn kan få en null-værdi, og det skaber en ny advarsel:



Figur 65 Det er ikke nemt at snyde kompileren ved null check

Advarslen fortæller, at ToUpper-metoden kan risikere at fejle, fordi Navn kan have en null-værdi. For at få advarslen væk, er du nødt til at være meget konkret i koden – eksempelvis:

```
class Person
{
    public string? Navn { get; set; }
    public string NavnMedStort()
    {
        return Navn == null ? "" : Navn.ToUpper();
    }
}
```

Du kan også fjerne advarslen ved at benytte udråbstegn-operatoren (som meget smukt også hedder ”null-forgiving operator”) som fortæller kompileren, at du tager ansvaret:

```
class Person
{
    public string? Navn { get; set; }
```

```
    public string NavnMedStort()
    {
        return Navn!.ToUpper(); // Eget ansvar!!
    }
}
```

Under alle omstændigheder tvinges du altså til at tage stilling til eventuelle null-relaterede problemer.

Hvis du ønsker at få deciderede kompileringsfejl (errors) og ikke blot advarsler (warnings), kan du tilføje WarningsAsErrors i .csproj-filen:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <Nullable>enable</Nullable>
    <WarningsAsErrors>nullable</WarningsAsErrors>
  </PropertyGroup>
</Project>
```

Nu kan koden slet ikke kompilere, hvis kompileren opsnuser mulige kommende problemer relateret til null-værdier.

Tuples

C# understøtter som mange andre programmeringssprog en speciel datastruktur kaldet en **tuple**. Hvis du hører ordet "tuple" udtalt, vil du sikkert høre trykket lagt på både TUple og tuPLE, men det helt korrekte er TUple (tuh·pl)[21].

Denne lidt specielle datastruktur giver en nem og effektiv mulighed for at opbevare værdier af forskellige typer, og du kan se det lidt som at slippe for at definere en klasse eller en struktur. De findes nemlig i to former – System.Tuple<> og System.ValueTuple<>. Førstnævnte er en referencebaseret type, og har været at finde i C# i flere versioner. Sidstnævnte er en værdibaseret type og blev tilføjet C# i version 7.

[21] Søg efter "how to pronounce tuple" på Google hvis du vil høre ordet udtalt

System.Tuple

Som eksempel på brug af den referencebaserede Tuple-klasse må du forestille dig en type til opbevaring af data relateret til en person – eksempelvis:

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
    public bool ErDansk { get; set; }
}
```

Klassen er, som du kan se, en almindelig DTO (Data Transfer Object), og indeholder udelukkende data. Den kan ligge til grund for opbevaring af data relateret til en enkelt person:

```
Person p = new Person()
{
    Navn = "Mathias", Alder = 14, ErDansk = true
};
```

Eller en samling af personer:

```
List<Person> personer = new List<Person> {
    new Person() { Navn = "Mikkel", Alder = 17, ErDansk = true },
    new Person() { Navn = "Mathias", Alder = 14, ErDansk = true },
};
```

Eller som argument eller returværdi:

```
List<Person> personer = new List<Person> {
    new Person() { Navn = "Mikkel", Alder= 17, ErDansk = true },
    new Person() { Navn = "Mathias", Alder= 14, ErDansk = true },
};

Console.WriteLine(FindÆldstePerson(personer).Navn); // Mikkel

Person FindÆldstePerson(List<Person> personer)
{
    return personer.OrderBy(p => p.Alder).LastOrDefault();
```

```
}
```

Alle eksempler er helt korrekte og logiske, men kræver, at du opretter klassen Person. Ideen med tuples er, at samme kode kan skrives generisk uden definition af en klasse. Der kan angives op til otte forskellige typer i en tuple, og dermed kan vores person-klasse erstattes med følgende:

```
Tuple<string, int, bool> p;
```

Nu kan variablen p indeholde referencen til en (anonym) datastruktur bestående af en string, int og en bool:

```
Tuple<string, int, bool> p1;
p1 = new Tuple<string, int, bool>("Mathias", 14, true);
// eller
Tuple<string, int, bool> p2 =
    new Tuple<string, int, bool>("Mikkel", 17, true);
```

Da det jo ikke er navngivne egenskaber, må værdier hentes ud på en anden måde:

```
Tuple<string, int, bool> p =
    new Tuple<string, int, bool>("Mikkel", 17, true);
Console.WriteLine(p.Item1); // Mikkel
Console.WriteLine(p.Item2); // 17
Console.WriteLine(p.Item3); // true
```

Egenskaber kan nu tilgås som itemX, hvor X er den rækkefølge, de er erklæret.

Førnævnte metode, der finder den ældste person, kan nu omskrives til:

```
List<Tuple<string, int, bool>> personer =
  new List<Tuple<string, int, bool>>() {
    new Tuple<string, int, bool>("Mikkel", 17, true),
    new Tuple<string, int, bool>("Mathias", 14, true)
    };

Console.WriteLine(FindÆldstePerson(personer).Item1); // Mikkel
```

```
Tuple<string, int, bool> FindÆldstePerson
    (List<Tuple<string, int, bool>> personer)
{
    return personer.OrderBy(p => p.Item2).LastOrDefault();
}
```

Og den fungerer præcis på samme måde. Forskellen er, at du ikke behøver erklære og benytte en decideret klasse. Koden er blevet generisk og kan benyttes på eksempelvis både en person, hund eller maskine.

System.ValueTuple

Endnu mere smart er den værdibaserede type (bemærk – ikke referencebaseret) ValueTuple. Den kan du sammenligne med en struktur, der består af værdier. Førnævnte Person klasse kunne omskrives til:

```
struct Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
    public bool ErDansk { get; set; }
}
```

Det betyder, at værdier er placeret et andet sted i hukommelsen (stack) med (tit) tilhørende forbedring i performance, og eksempelvis kopiering af data resulterer i kopiering af værdier og ikke referencer:

```
Person p1 = new Person()
  { Navn = "Mathias", Alder = 14, ErDansk = true };
Person p2 = p1;
p1.Navn = "Mikkel";
Console.WriteLine(p1.Navn);   // Mikkel
Console.WriteLine(p2.Navn);   // Mathias
```

Havde Person været en klasse ville p1 og p2 indeholde samme reference, og dermed udskrive 2 gange ”Mikkel”.

Hvis du ønsker en værdibaseret tuple kan du benytte System.ValueTuple<> med samme syntaks som System.Tuple<>:

```
ValueTuple<string, int, bool> p1
    = new ValueTuple<string, int, bool>("Mikkel", 17, true);
ValueTuple<string, int, bool> p2
    = new ValueTuple<string, int, bool>("Mathias", 14, true);
Console.WriteLine(p1.Item1);    // Mikkel
Console.WriteLine(p2.Item1);    // Mathias

List<ValueTuple<string, int, bool>> personer =
    new List<(string, int, bool)>
    {
        p1, p2
    };
```

Du kan sågar direkte sammenligne værdier:

```
ValueTuple<string, int, bool> p1
    = new ValueTuple<string, int, bool>("Mikkel", 17, true);
ValueTuple<string, int, bool> p2
    = new ValueTuple<string, int, bool>("Mathias", 14, true);
ValueTuple<string, int, bool> p3
    = new ValueTuple<string, int, bool>("Mikkel", 17, true);

Console.WriteLine(p1 == p2);  // false
Console.WriteLine(p1 == p3);  // true
```

Men ValueTuple indeholder også en del syntakssukker, som hurtigt kommer til at klistre på fingrene.

For det første kan værdier tildeles på en smart måde ved hjælp af parenteser:

```
ValueTuple<string, int, bool> p1
    = new ValueTuple<string, int, bool>("Mikkel", 17, true);
ValueTuple<string, int, bool> p2 = ("Mikkel", 17, true);
var p3  = ("Mikkel", 17, true);

Console.WriteLine(p1.Item1);    // Mikkel
Console.WriteLine(p2.Item1);    // Mikkel
```

```
Console.WriteLine(p3.Item1);    // Mikkel
```

Alle tre tildelinger giver det samme, og især

```
var p3  = ("Mikkel", 17, true);
```

er jo ret fiks, fordi kompileren er kvik nok til at udlede typer.

Yderligere kan egenskaber navngives således, at du ikke behøver tilgå egenskaber gennem ItemX:

```
var p1 = (Navn: "Mathias", Alder: 14, ErDansk: true);

Console.WriteLine(p1.Navn);    // Mathias
Console.WriteLine(p1.Alder);   // 14
Console.WriteLine(p1.ErDansk); // true

Console.WriteLine(p1.Item1);   // Mathias
Console.WriteLine(p1.Item2);   // 14
Console.WriteLine(p1.Item3);   // true
```

Som du kan se, kan egenskaber nu både tilgås gennem et logisk navn, samt gennem ItemX.

Til slut er her førnævnte metode FindÆldstePerson – nu med Value-Tuple:

```
List<(string, int, bool)> personer = new List<(string, int, bool)>()
{
    ("Mikkel", 17, true),
    ("Mathias", 14, true)
};

Console.WriteLine(FindÆldstePerson(personer).Navn);   // Mikkel
Console.WriteLine(FindÆldstePerson(personer).Item1);  // Mikkel

(string Navn, int Alder, bool ErDansk) FindÆldstePerson
    (List<(string, int, bool)> personer)
{
    return personer.OrderBy(p => p.Item2).LastOrDefault();
}
```

Bemærk, at en tuple, der benyttes som returværdi, navngives, som var det argumenter til en metode.

> Du behøver ikke benytte tuples, men det kan gøre koden både generisk og nem at vedligeholde

Nedbrydning til variabler

En anden feature som tit er relateret til tuples er nedbrydning (deconstruction). Det gør det muligt nemt at nedbryde en tuple til enkelte variabler. Se eksempelvis denne tuple:

```
var a = ("Mikkel", 17);
Console.WriteLine(a.Item1); // Mikkel
Console.WriteLine(a.Item2); // 17
```

Den kan nedbrydes manuelt til variabler som følger:

```
string navn = a.Item1;
int alder = a.Item2;
Console.WriteLine(navn);    // Mikkel
Console.WriteLine(alder);   // 17
```

Men koden kan simplificeres en hel del – her på en enkelt linje:

```
(string navn, int alder) = a;
Console.WriteLine(navn);    // Mikkel
Console.WriteLine(alder);   // 17
```

og endnu nemmere:

```
(var navn, var alder) = a;
Console.WriteLine(navn);    // Mikkel
Console.WriteLine(alder);   // 17
```

Men nedbrydning er ikke bare relateret til tuples. Samme funktionalitet kan tilføjes dine egne klasser ved hjælp af en metode kaldet Deconstruct.

Her er et eksempel med klassen Person:

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
    public bool ErDansk { get; set; }

    public void Deconstruct(out string navn,
        out int alder, out bool erDansk)
    {
        navn = Navn;
        alder = Alder;
        erDansk = ErDansk;
    }
}
```

Bemærk, at metoden Deconstruct benytter såkaldte out-parametre, som blandt andet benyttes i avancerede metoder til at tildele værdier til variabler erklæret i den kaldende metode. Men her benyttes out-parametrene til automatisk nedbrydning:

```
Person p = new Person
{
    Navn = "Mathias",
    Alder = 14,
    ErDansk = true
};
Console.WriteLine(p.Navn);      // Mathias
Console.WriteLine(p.Alder);     // 14
Console.WriteLine(p.ErDansk);   // true

// Nedbrydning
(var navn, var alder, var erDansk) = p;
Console.WriteLine(navn);        // Mathias
Console.WriteLine(alder);       // 14
Console.WriteLine(erDansk);     // true
```

Bemærk den nemme nedbrydning til enkelte variabler.

Metoden Deconstruct kan eventuelt overloades således, at nedbrydning kan ske med færre eller flere argumenter:

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
    public bool ErDansk { get; set; }

    public void Deconstruct(out string navn, out int alder,
        out bool erDansk)
    {
        navn = Navn;
        alder = Alder;
        erDansk = ErDansk;
    }

    public void Deconstruct(out string navn, out int alder)
    {
        navn = Navn;
        alder = Alder;
    }
}
```

Nu kan klassen nedbrydes til både en string, int og bool samt en string og en int.

Overskrivning af operatorer

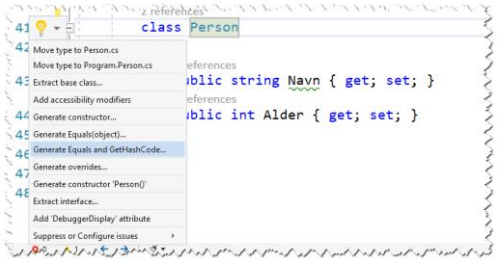
Der kan være situationer, hvor du ønsker at kunne sammenligne to objekter af klasser på forskellige måder Umiddelbart giver C# kun mulighed for at kontrollere, om referencen er det samme. Med udgangspunkt i følgende klasse:

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
}
```

kan objekter sammenlignes som følger:

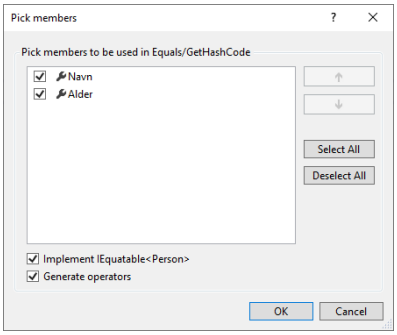
```
Person p1 = new Person
{
    Navn = "Mathias",
    Alder = 14,
};

Person p2 = new Person
{
    Navn = "Mathias",
    Alder = 14,
};

Console.WriteLine(p1 == p2);    // false
```

Selv om to objekter indeholder samme værdier, så vil en sammenligning fejle, fordi det er referencerne, der sammenlignes og ikke værdierne. Men det kan du ændre, hvis du ønsker, og du kan endda lade Visual Studio eller Visual Studio Code gøre det for dig.



Figur 66 Visual Studio kan hjælpe med at overskrive operatorer

Hvis du placerer cursoren på klassenavnet, vil du få mulighed for at få hjælp til at overskrive operatorer ved at vælge ”Generate Equals and GetHashCode...”. Det vil skabe denne dialogboks:



Figur 67 Autogenerering af kode til overskrivning

Et klik på OK-knappen vil ændre koden i Person-klassen til følgende:

```
class Person : IEquatable<Person?>
{
    public string Navn { get; set; }
    public int Alder { get; set; }

    public override bool Equals(object? obj)
    {
        return Equals(obj as Person);
    }

    public bool Equals(Person? other)
    {
        return other != null &&
                Navn == other.Navn &&
                Alder == other.Alder;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(Navn, Alder);
    }

    public static bool operator ==(Person? left, Person? right)
    {
        return EqualityComparer<Person>.Default.Equals(left, right);
    }
```

```
    public static bool operator !=(Person? left, Person? right)
    {
        return !(left == right);
    }
}
```

Der tilføjes nu både en overskrivning af == og != operatorerne samt en overskrivning af metoderne Equals (som internt bruges til at sammenligne) og GetHashCode (som benyttes til at skabe en unik nøgle, der benyttes internt af runtime i eksempelvis samlinger af objekter). Det ligger uden for denne bogs rammer at komme nærmere ind på den autogenererede kode, men resultatet er meget tydeligt:

```
Person p1 = new Person
{
    Navn = "Mathias",
    Alder = 14,
};

Person p2 = new Person
{
    Navn = "Mathias",
    Alder = 14,
};

Console.WriteLine(p1 == p2);    // true
```

Nu sammenlignes på værdier og ikke på referencer, hvilket nogle gange kan være ret praktisk.

På samme måde kan du også vælge at overskrive eksempelvis < (mindre end) og > (større end) operatorerne. Det kunne være, at du ville have en forretningslogik, hvor det gav mening at sammenligne objekter på den måde. Her er eksempelvis en simpel terning:

```
class Terning
{
    public int Værdi { get; private set; }
    private Random random = new Random();
```

```
    public Terning()
    {
        Ryst();
    }
    public void Ryst()
    {
        Værdi = random.Next(1, 7);
    }
}
```

Og her er brugen af klassen med et par terninger:

```
Terning t1 = new Terning();
Console.WriteLine(t1.Værdi);    // Tilfældig værdi
Terning t2 = new Terning();
Console.WriteLine(t2.Værdi);    // Tilfældig værdi
```

Det kunne måske være smart, at kunne spørge om t1 > t2 eller omvendt, men det er jo kun dig, der ved, hvordan logikken skal være, og det kan du fortælle runtime ved at overskrive de to operatorer:

```
class Terning
{
    public int Værdi { get; private set; }
    private Random random = new Random();
    public Terning()
    {
        Ryst();
    }
    public void Ryst()
    {
        Værdi = random.Next(1, 7);
    }

    public static bool operator >(Terning? left, Terning? right)
    {
        return left.Værdi > right.Værdi;
    }

    public static bool operator <(Terning? left, Terning? right)
    {
```

```
        return left.Værdi < right.Værdi;
    }
}
```

Da klassen Terning ved hvordan instanser skal sammenlignes er følgende kode mulig:

```
Terning t1 = new Terning();
Terning t2 = new Terning();

bool mindre = t1 < t2;
bool større = t1 > t2;
```

Du kan naturligvis også vælge at overskrive andre operatorer – herunder == og != som før nævnt.

Søg på nettet efter ”operator overloading c#” for yderligere informationer.

Poster (records)

En post (record) i C# er en af de features, som du har svært ved at undvære, når du først lige har lært, hvad de kan bruges til.

I sin helt grundlæggende form kan du oprette brugen af record-kodeordet, som en hurtig måde at skabe en type til at opbevare data. Lad os antage, at du har behov for at opbevare data relateret til en person. Hvis du benytter en klasse eller en struktur, kunne du skrive kode som følger:

```
class Person
{
    public string Navn { get; init; }
    public int Alder { get; init; }
    public bool ErDansk { get; init; }

    public Person(string navn, int alder, bool erDansk)
    {
        Navn = navn;
        Alder = alder;
```

```
        ErDansk = erDansk;
    }
}
```

Det er en simpel data-klasse, som kan benyttes som eksempelvis:

```
Person p1 = new Person("Mikkel", 17, true);
```

Bemærk, at klassen er immutable. Når egenskaber først er tildelt en værdi, kan de ikke tilrettes igen.

Du ser tit denne form for meget simple datatyper, som blot har til formål at transportere data.

Præcis samme type kan også oprettes som en post som følger:

```
record Person(string Navn, int Alder, bool ErDansk);
```

I stedet for 12-13 linjers kode kan du nøjes med én linje. Resten klarer kompileren for dig. Den vil nemlig sørge for at autogenerere en klasse med de nævnte egenskaber, en konstruktør og en del andre medlemmer.

Bemærk, at poster (records) er en C# 9 feature!

Oprettelse af et objekt sker på præcis samme måde som ved den førnævnte Person-klasse:

```
Person p1 = new Person("Mikkel", 17, true);
```

De autogenerede egenskaber Navn, Alder og ErDansk vil være init-egenskaber, og typen er dermed immutable:

```
Person p1 = new Person("Mikkel", 17, true);
// p1.Navn = "Mathias";  // init egenskab - kan ikke tildeles en værdi
```

Men udover at der bliver autogenereret egenskaber og konstruktør bliver der ligeledes tilføjet andre medlemmer. Eksempelvis er ToString-

metoden overskrevet, så den returnerer en string, der beskriver værdierne:

```
Person p1 = new Person("Mikkel", 17, true);
Console.WriteLine(p1);
// Udskriver:
// Person { Navn = Mikkel, Alder = 17, ErDansk = True }
```

Hvis det er din egen klasse, skal du selv tilføje den kode. Ellers vil ToString-metoden blot returnere navnet på klassen.

Yderligere er der tilføjet kode, der overskriver != og == operatorerne samt metoderne GetHashCode og Equals. Det betyder, at objekter kan sammenlignes på værdier og ikke på referencer:

```
Person p1 = new Person("Mikkel", 17, true);
Person p2 = new Person("Mikkel", 17, true);
Console.WriteLine(p1 == p2);    // true
```

Hvis det er din egen klasse, skal du selv tilføje den kode (se side 311).

Der er ligeledes autogenereret en Deconstruct-metode således, at følgende kode er mulig:

```
Person p1 = new Person("Mikkel", 17, true);
(var navn, var alder, var erDansk) = p1;
Console.WriteLine(navn);    // Mikkel
```

Slutteligt er det nemt at få en kopi (husk – en post er immutabel) af objektet ved hjælp af with-kodeordet:

```
Person p1 = new Person("Mikkel", 17, true);
Console.WriteLine(p1);
// Person { Navn = Mikkel, Alder = 17, ErDansk = True }
Person p2 = p1 with { Alder = 18 };
Console.WriteLine(p2);
// Person { Navn = Mikkel, Alder = 18, ErDansk = True }
```

Bemærk, hvor nemt det er at kopiere data fra p1 til p2 og blot ændre alder.

Hvis du er nysgerrig på, hvordan den autogenererede klasse ser ud, så prøv at erklære en post på https://sharplab.io. Der kan du se både den dekompilerede IL kode og tilhørende C# kode.

Du kan vælge at tilføje metoder og ekstra konstruktør samt egne egenskaber eller andre medlemmer, og en post kan også indgå i et arvehierarki med andre poster, men det ligger uden for denne bogs rammer at komme nærmere ind på det – bortset fra et simpelt eksempel:

```
record Person(string Navn, int Alder, bool ErDansk)
{
    public int EstimeretFødselsår()
    {
        return DateTime.Now.Year - this.Alder;
    }
}
```

Bemærk, at definitionen af typen nu benytter tuborgklammer for at gøre det muligt at tilføje en metode.

```
Person p1 = new Person("Mikkel", 17, true);
Console.WriteLine(p1);
// Person { Navn = Mikkel, Alder = 17, ErDansk = True }
Console.WriteLine(p1.EstimeretFødselsår());
// 2003
```

Helt overordnet skal du bare vide, at definition af en simpel datatype kan klares på en enkelt linje som en post, og samtidigt får du en masse yderligere funktionalitet foræret helt automatisk.