

Workshops – Rammeverk og .NET

Emil Berglund

Høgskolen i Østfold

Emnekode: ITF20123

Innlevering: 03.03.2025

Til info:

Alle workshops hadde ulike oppgaver og formål, og strukturen på oppgavene var derfor ulike. Man vil derfor se at strukturen i min innlevering ikke er lik på hver oppgave, men at de heller følger den individuelle workshop-strukturen.

I hver workshop har jeg og gruppemedlemmene jobbet på samme arbeidsområde hvor oppgaven har krevd dokumentasjon av arbeid/fremgangsmåte. Det vil derfor si at noe innhold i dette dokumentet, ikke vil være mitt, men skrevet av de jeg har vært på gruppe med. På den andre siden, kan også noe av innholdet avvike fra det de andre har skrevet. Enten det er omformulering, eller at perspektivet er endret på.

Jeg er derimot ikke uenig i det som er skrevet, selv om det ikke er jeg som har skrevet det personlig.

Innhold

Workshop 1 – 20.01.2025.....	4
Workshop 2 – 27.01.2025.....	11
Workshop 3 – 10.02.2025.....	19
Workshop 4 – 17.02.2025.....	23

Workshop 1 – 20.01.2025

Gruppemedlemmer:

Første del:

Emil Berglund

Abdisalam Elmi

John Olav Rabbevåg

Andre del:

Emil Berglund

Abdisalam Elmi

John Olav Rabbevåg

Ida Tollaksen

Max Oliver Solvang

Første del:

Litt_kode.cs

Hva gjør koden:

Public class file:

Klassen representerer en fil som består av flere deler (parts), hvor hver del er en streng. Man kan lage en tom instans, eller initialisere den med en liste av strenger. Klassens override returnerer hele listens innehold separert med komma mellom hver streng.

Konstruktørene: Parts (hoved), IEnumerable som parameter for å opprette liste.

Override: returnerer strenger med komma, imellom.

Public class library:

- Klassen lager et bibliotek med filer som legges i en liste.
- _stuff lagrer File-objekter i en liste. Listen er privat.
- IReadOnlyList lines leser _stuff og legger til i listen lines.
- IncludeFile(file F) legger til File-objekter i _stuff.
- Metoden IncludeFile legger til en fil i biblioteket.

Navngivning:

Slik filnavnet er skrevet er feil. Ifølge retningslinjene skal klassenavn ha PascalCase.

Litt_kode.cs skal heller skrives som *LittKode.cs*.

Public class file:

Ifølge retningslinjer skal klassenavn ha PascalCase, og dermed være *File* fremfor *file*. Det samme gjelder *parts*.

Public class library

Klassens navn *library* skal ha stor forbokstav i følge retningslinjer. Feltet *stuff* er lite beskrivende, og bør i denne sammenhengen heller være *_files*. Videre antyder *lines* at det er linjer som går gjennom, når det egentlig er filer. *lines* burde derfor endres til *Files*. Metoden *IncludeFile* har riktig navngivning, men kan til fordel endres til *AddFile* for å bedre kommunisere hva den gjør.

YFY!.cs

Hva gjør koden:

Public class app: Klassen representerer en app for å kunne lese filer fra et bibliotek.

Public library readstuff

Koden tar imot en filsti (place) som argument. Den sjekker deretter om filstien er tom eller null, og kaster passende exceptions deretter. Videre leser filen linje for linje med StreamReader, før den senere deler opp hver linje i mindre deler med *splitstuff*. Til slutt puttes de oppdelte linjene i en *file*-instans og legger dem til i en *library*. Returnerer et *library*-objekt som inneholder alle de oppdelte linjene fra filen.

Private List<string> splitstuff

Splitstuff har som oppgave å splitte strenger i mindre deler, med komma-separering.

Splitstuff(string chunk): Splitter strenger på komma i en liste.

En tom liste result for å lagre delene, looking som er en boolean variabel som bruker for å passe på splitting.

Bygger opp delene med *StringBuilder* i slice.

Item = "", hvis looking er sann og neste felt er også tom, blir den lagt til tomrom og skipper indeksen.

Hvis looking er usann, (item = ",") blir den lagt til i result, tømmer slice.

Siste slice blir lagt til i result.

Navngivning:

Filnavnet *YEY!.cs* bryter med retningslinjene på flere områder. Først og fremst skal filnavn ha PascalCase, og burde i såfall vært *Yey!.cs*, for det andre er *YEY!* Et lite beskrivende filnavn som ikke sier noe om hva filens innhold og formål er, for det tredje inneholder det spesialtegnet «!», som generelt burde unngås i filnavn.

Public class app:

Klassenavnet *app* er først og fremst i lower-case, noe det ikke skal være. *App* ville vært mer riktig her. Videre er *app* et lite beskrivende navn, som ofte betyr at klassen er en main-klasse som kjører et CLI e.l. Dette er ikke tilfelle her, og burde heller vært *CsvReader*, *FileProcessor*, eller noe i den duren.

Metoden *readstuff* er upresist og ikke i PascalCase. Navnet burde heller vært *ReadLibraryFromFile* for lettere forståelse og riktig navngiving.

Variabelnavnet *place* er upresist og vanskelig å forstå. Et bedre alternativ ville vært *filePath*.

Metoden *splitstuff* følger ikke retningslinjene for PascalCase, og burde heller vært *SplitStuff*. Videre er navnet upresist og gir lite innsikt i hva som splittes. Et bedre navn ville vært, *SplitLine*.

I metoden finner man også parameteret *chunk*, Dette navnet sier lite om hva objektet er. I stedet for *chunk*, burde det heller vært *line*.

Boolean verdien *looking* er også litt misvisende, og *insideQuotes* ville vært bedre her, for lesbarhet og forståelse.

Prog.cs

Hva gjør koden:

Klassen representerer et program for å lese data fra en fil (i dette tilfellet en .csv-fil).

Path lagrer stien til filen med data.

Det instaniseres et nytt app-objekt. Deretter brukes metoden `readstuff` fra app for å lese inn dataene. Hver linje leses inn, lagres som filobjekter og skrives ut i konsollen.

Navngivning:

Filnavnet `prog.cs` har flere feil som bryter med retningslinjene. Først og fremst er det lower-case. I tillegg er det forkortet. I dette tilfelle har defineringen av klassen det riktige navnet, mens filnavnet er forkortet og feil. Riktig navngivning er her, `Program.cs`.

Videre er det tidligere nevnte problemer som navngivning av `readstuff`. Det burde heller vært `CsvReaderProgram` eller liknende. I tillegg burde variabelnavnet `a` vært noe mer beskrivende.

Andre del:

Vi møtte gruppe 16.

Begge gruppene var ikke 100% ferdige da vi møttes 11:15, men vi fikk gått gjennom alt hvor vi gikk gjennom «prog.cs» sammen, da begge ikke rakk denne.

Totalt sett, var vi ganske samkjørte på det vi hadde funnet, og det vi mente programmene gjorde. Det samme gjaldt eventuelle forbedringer.

Samlet sett mente vi også at klassene til sammen fungerte som en CSV leser, og skriver, som i Main, skriver ut innholdet i det som er blitt skrevet til CSV filen.

Workshop 2 – 27.01.2025

Gruppemedlemmer:

Bibliotek 1:

Emil Berglund

Obaida Taha

Bibliotek 2:

Sebastian Thomsen

Hedda Ørnelund Nilsen

Bibliotek 1: Emil og Obaida

Feil/Brytning av retningslinjer

- Get og Set er skrevet vertikalt, ikke horisontalt.
- Alle List deklarasjoner har mellomrom, noe den ikke skal ha.
 - List < ShopOrder > (feil)
 - List <ShopOrder> (riktig)
- “public static decimal CalculateShippingCost” returnerer 0, men bør returnere den totale kostnaden for frakt.
- “public static bool IsWithinBudget” returnerer 0, men bør sjekke om totalprisen er innenfor budsjettet, og returnerer en bool verdi deretter.
- “public OptimizationResult OptimizeOrders” returnerer ingenting, men må dette, ettersom den ikke er markert med void. Eksempelvis “return new OptimizationResult”
- Generell feilhåndtering. Metoder med setttere, mangler feilhåndtering. Eksempelvis “MaxAmount”. Her kan man egentlig skrive et negativt tall, som ikke er heldig i programmets sammenheng.

Mangler i koden:

Koden er ufullstendig, og mangler blant annet en funksjon som søker opp de komponenetene som er tilgjengelige. Denne kunne eksempelvis sett slik ut:

```
public class Store {
    public string Name { get; set; }
    public List<Component> Components { get; set; } = new List<Component>();

    public List<Component> SearchComponent(string componentName) {
        return Components.Where(c => c.Name.Contains(componentName, StringComparison.OrdinalIgnoreCase));
    }
}
```

Pseudo-code

Scenario 1: Søk etter produkt, og finn billigst pris.

Søk opp komponent CPU

```
cpuComponent = SearchComponent("CPU")
```

Hent ulike alternativer

```
equivalentComponents = cpuComponent.GetEquivalentComponents()
```

Initialiser minimum pris og butikk

```
minPrice = Infinity
```

```
bestStore = null
```

Gå gjennom butikkene og finn den billigste prisen

For each store in allStores:

For each component in equivalentComponents:

```
price = store.GetComponentPrice(component)
```

If price < minPrice:

```
minPrice = price
```

```
bestStore = store
```

Returner den billigste butikken og prisen

```
Print "Best store for CPU replacement: " + bestStore.ShopName
```

```
Print "Price: " + minPrice
```

Scenario 2: Bestilling av komponeneter innenfor et budsjett

Start med å søke etter komponenten "CPU"

```
cpuComponent = SearchComponent("CPU")
```

Hent de alternative komponentene

```
equivalentComponents = cpuComponent.GetEquivalentComponents()
```

Initialiser minimum pris og butikk

```
minPrice = Infinity
```

```
bestStore = null
```

Gå gjennom butikkene og finn den billigste prisen

For each store in allStores:

For each component in equivalentComponents:

```
price = store.GetComponentPrice(component)
```

```
If price < minPrice:
```

```
    minPrice = price
```

```
    bestStore = store
```

Returner den billigste butikken og prisen

```
Print "Best store for CPU replacement: " + bestStore.ShopName
```

```
Print "Price: " + minPrice
```

Scenario 3: Optimalisering av bestillinger fra flere butikker:

Start med en liste over nødvendige komponenter

```
components = [SearchComponent("CPU"), SearchComponent("GPU"),  
SearchComponent("RAM")]
```

Samle alle butikker

```
allStores = [store1, store2, store3]
```

Optimaliser bestillinger

```
result = OptimizeOrders(components, allStores)
```

Regn ut den totale prisen for bestillingen inkludert frakt

totalOrderCost = 0

For each order in result.Orders:

totalOrderCost += order.TotalCost + order.Shop.ShippingCost

Sjekk om den totale kostnaden er innenfor budsjettet

If totalOrderCost <= budget.MaxAmount:

Print "Optimal bestilling er innenfor budsjettet."

Print "Total cost including shipping:" + totalOrderCost

Else:

Print "Optimal bestilling overskriver budsjettet."

Bibliotek 2: Hedda og Sebastian

Hva programmet gjør:

Library 2 er et rammeverk for å optimalisere innkjøp og sammensetning av PC-komponenter. Det er organisert i navnerommet *PCOrderOptimizer* og gir en strukturert måte å håndtere PC-komponenter, produkter og butikker på. Komponenter kategoriseres med enumen *ComponentCategory*, mens produkter og butikker modelleres gjennom henholdsvis strukturen *Product* og klassen *Store*.

Brukere kan definere PC-konfigurasjoner med klassen *PCBuildConfig*, som inkluderer budsjett og nødvendige komponenter. Grensesnittet *IOrderOptimizer* lar brukeren implementere optimaliseringssstrategier via metoden *OptimizeOrders*. Resultatet av en optimalisering modelleres med strukturen *OrderStrategy*, som viser ordrer og totale kostnader.

Enkelte metoder, som *OptimizeOrders* og *GroupByEquivalence*, mangler implementasjon, og biblioteket kunne dratt nytte av validering og feilhåndtering. Til tross for dette gir Library 2 en fleksibel og utvidbar grunnstruktur for å planlegge og optimalisere PC-bygging.

Pseudokode

Scenario 1:

Opprett en instans av klassen “Store” og sett egenskaper:

- Navn på butikken
- Gratis frakt-grense
- Fraktkostnad
- Opprett flere butikker

Opprett klassen: “Component category”:

- Navn på de ulike kategoriene

Scenario 2:

Opprette flere produkter ved bruk av “Product” strukturen:

- Sette navnet på produktet
- Sette prisen på produktet
- Kategorisere produktet
- Gruppere med tilsvarende produkter

Legg produktene i listen til "Products" i "Store"

Scenario 3:

Opprett en instans av *PCBuildConfig*

- Sett budsjett
- Legg til nødvendige komponentkategorier (f.eks. RAM, CPU)

Implementer *OptimizeOrders`* fra *IOrderOptimizer*:

- Beregn den totale kostnaden for produkter som oppfyller kravene i *PCBuildConfig*.
- Velg den billigste strategien for innkjøp.
- Returnerer optimalisert ordrestrategi.

Retningslinjer biblioteket bryter

- Mangler validering: Det ser ikke ut til å være innebygd validering for egenskaper som Budget eller Price, *EquivalentGroup*.
- Ingen implementasjon av metoder: Mange metoder, som *GroupByEquivalence* og *OptimizeOrders*, mangler implementasjon. Dette gjør biblioteket uferdig. Her må det implementeres manglende metoder for å gjøre biblioteket funksjonelt.
- Ingen feilhåndtering: Det finnes ingen håndtering av potensielle feil, som å legge til produkter med ugyldige verdier. Budsjetter i *PCBuildConfig* som er null eller negative. Fraktkostnader som er ugyldige eller null. Feilhåndtering bør inkluderes ved å validere input og kaste unntak der det er nødvendig.
- Det mangler return på *IOrderOptimizer*
- Store inneholder en liste av produkter (Products), men det er ingen mekanisme for å validere at produktene i butikken oppfyller en spesifikk konfigurasjon i *PCBuildConfig*.
- *PCBuildConfig* definerer krav til komponentkategorier, men det er ingen sjekk for at Products i en Store faktisk dekker alle nødvendige kategorier.

Forslag til forbedringer:

- Implementer alle manglende metoder.
- Legg til feilhåndtering for negative priser, ugyldige budsjetter og andre invalidverdier.
- Gjør avhengighetene mellom klasser tydeligere, f.eks. ved å sikre at alle nødvendige kategorier i *PCBuildConfig* kan dekkes av produkter i Store.

- Fjern nullable typer der det ikke er nødvendig, og erstatt dem med fornuftige standardverdier.

Diskusjon og Konklusjon:

I del 2 byttet vi på, hvor Emil og Obaida analyserte Bibliotek 2, mens Hedda og Sebastian analyserte Bibliotek 1. Begge gruppene identifiserte hovedsakelig de samme feilene og foreslo lignende løsninger. Hovedfokus i diskusjonen var på strukturelle forbedringer, implementering av manglende metoder, og hvordan man kan forbedre feilhåndtering og validering av input.

Enighet om forbedringer:

- Viktige forbedringer som ble enstemmig foreslått inkluderte implementering av manglende metoder og validering av input.

Workshop 3 – 10.02.2025

Gruppemedlemmer:

S1:

Emil Berglund

Andreas Isaksen

S2:

Klaus Haugnæss

Ida Tollaksen

Hedda Ørnelund Nilsen

Medlemmer fra S2 utfører oppgaven til S1

- Medlemmene i S2 fikk S1 sitt prosjekt, ferdig oppsatt med filhierarki, namespace og en tom main-funksjon.
- De fikk deretter informasjon om hva ønsket resultat var, og hvilken metode de skulle bruke.
- Andres var veileder, og Emil var observatør og skribent.

Fremgangsmåte – Oppgave A:

1. Medlemmene fant CSV-filen i downloads og flyttet denne inn i prosjekt mappen.
2. Medlemmene begynte deretter med Main-metoden.
 - o De fant først filstien (den relative)
 - o Videre fant de at filen har header, og satte deretter bool-parameteret som true i ParseCsv
 - Slet litt med tredje parameter, hvor de ble tipset om at det kunne være en char omringet av fnutter (selv om denne egentlig kunne stått tom)
 - Fikk deretter build error ved kjøring av programmet
 - o Fikk tips om at de trengte en liste med string som value, og lagde deretter en instanse av DataParser
 - Fikk deretter Error: Wrong FilePath
 - Prøvde med Absolutt path, som fungerte.
 - o Programmet kjører, men ingenting skrives ut.
 - Fant ut av at programmet returnerer noe, ikke printer
 - Prøvde derfor å Console write
 - Fikk bare objektid, og ingen tekst
 - Fikk litt hjelp ved å kalle på objektet og ikke en variabel av DataParser
 - Ved å gjøre dette, og benytte string.Join med separator, og skrive ut listeobjektet med bestemt index, fikk de ut ønsket resultat.

Fremgangsmåte – Oppgave B:

1. Medlemmene begynte på Main-metoden
 - o Lagde to objekter; FileInfo og ParserOptions
 - o Videre lagde de et objekter av ParseCsv med disse objektene som parametere
 - o Videre var det å skrive ut med console log. Likt som oppgave a.

Medlemmer fra S1 utfører oppgaven til S2

- Medlemmene i S1 fikk S2-sitt prosjekt, ferdig oppsatt, med en tom main-funksjon
- De fikk informasjon om hva ønsket resultat var, og hvilken metode de skulle bruke.
- Klaus var veileder, Hedda og Ida var observatør og skribent

Fremgangsmåte – Oppgave A:

1. Medlemmene gikk igjennom mappestruktur og innhold.
2. Medlemmene tittet etter hvor metoden SearchAndReplace ligger, de fant den i FileTextReplacer.cs
3. De gjør seg kjent med hvilken parameter metoden bruker. De sier høyt at det er en konstruktør, opprette søketekst, file, true false, casesensitiv, definere backup.
4. Medlemmene opprettet et FileTextReplacer objekt i main, prøver først å sette parameterene inne i oppretelsen av objektet.
5. Fikk hint om at man kanskje burde sette parameterene utenfor oppretelsen.
6. Videre prøvde de å undersøke hvordan de kunne gjøre dette, men får file not found error.
7. Forsøker å ordne filsti ved å bruke absolutt filstiden, det funker, backup ble opprettet 😊
8. Rakk ikke oppgave B grunnet tidsmangel

Fremgangsmåte – Oppgave B:

Emil og Andreas rakk å bli ferdig med Oppgave A klokken 12:05, som egentlig var etter forelesningstid. De rakk derfor ikke å se på Oppgave B i det hele tatt. Et bilde av deres program ligger under.



```
1 public static void Main(){
2     FileTextReplacer searchndreplace = new FileTextReplacer();
3
4     int options = searchndreplace.SearchAndReplace(
5         @"C:\Users\idakt\OneDrive - Østfold University College\Dokumenter\Høyskole\Var25\Net\Workshops\Workshop3\Worshop3\fil.txt",
6         "Hello",
7         "Hi",
8         caseSensitive: true,
9         true
10    );
11    Console.WriteLine($"Number of replacements: {options}");
12
13 }
14
```

Workshop 4 – 17.02.2025

Gruppemedlemmer:

Emil Berglund

Obaida Taha

Aram Naji Thalje

Herman Jacobsen

Oppgave 1:

Besvarelse oppgave 1:

Exceptions handler om å håndtere feil og forhindre at programmet krasjer. Det er viktig å være spesifikk i exception-handling ved å håndtere de situasjonen som kan skje. Eksempelvis bruk `ArgumentNullException`, ikke `ArgumentException` hvis man forventer null-error. Ved å bruke `ArgumentException`, vil den også håndtere mange andre feilmeldinger, som kan føre til at man ikke helt vet hva feilene egentlig er.

Vi som gruppe har opplevd mange feil, men veldig få som ikke har vært forståelige. Flere hadde opplevd feil i MySQL som var uforståelige, men ingen i .NET eller C#.

Konsekvensene av dårlige exceptions er forvirring for utviklere, og dårlig brukeropplevelse for bruker, ved at programmet kan krasje. Dårlige exceptions leder til mye feilsøkingstid for utviklere, som kunne vært unngått ved bedre feilhåndtering.

Oppgave 2

Scenario A:	<i>ArgumentOutOfRangeException</i> eller <i>IndexOutOfRangeException</i>
Scenario B:	<i>ArgumentOutOfRangeException</i> , <i>NullReferenceException</i> , <i>KeyNotFoundException</i> , <i>ArgumentNullException</i> , <i>ObjectNotFoundException</i> , <i>ArgumentException</i>
Scenario C:	<i>HttpRequestException</i> , <i>TimeoutException</i> , <i>System.AccessViolationException</i> , <i>InvalidOperationException</i>
Scenario D:	<i>FormatException</i> , <i>IndexOutOfRangeException</i> Bonus: Custom-exception

Begrunnelse av exceptions:

Vi har valgt ut følgende exceptions, ettersom det er disse vi har sett for hos fungerer i ulike situasjoner. Vi har også sett for oss at custom-exceptions vil være lurt her, ettersom det er noen tilfeller som kan håndteres bedre med egne feilhåndtering og sjekker. Hadde vi fått vite den spesifikke situasjonen kunne vi valgt ut enda bedre og

mer spesifikke feilhåndteringer. Eksempelvis Scenario C. Her kan det være en forespørsel til en nettside, database eller noe annet.

Oppgave 3

```

1  using System;
2
3  public class InvalidTransactionException : Exception
4  {
5      public string CreditCardNumber { get; }
6
7      public InvalidTransactionException(string creditCardNumber)
8          : base($"Invalid transaction for card: {MaskCardNumber(creditCardNumber)}")
9      {
10         CreditCardNumber = creditCardNumber;
11     }
12
13     public InvalidTransactionException(string creditCardNumber, string message)
14         : base(message)
15     {
16         CreditCardNumber = creditCardNumber;
17     }
18
19     public InvalidTransactionException(string creditCardNumber, string message, Exception innerException)
20         : base(message, innerException)
21     {
22         CreditCardNumber = creditCardNumber;
23     }
24
25     private static string MaskCardNumber(string cardNumber)
26     {
27         if (string.IsNullOrWhiteSpace(cardNumber) || cardNumber.Length < 4)
28             return "****"; // Minimal maskering hvis nummeret er ugyldig eller for kort
29         return new string('*', cardNumber.Length - 4) + cardNumber[^4..]; // F.eks. "*****1234"
30     }
31
32     public override string ToString()
33     {
34         return $"{base.ToString()}, CardNumber: {MaskCardNumber(CreditCardNumber)}";
35     }
36 }
```

Vi tok utgangspunktet i Scenario D hvor feilhåndteringen skulle ta hensyn til kortformat, ved at korttypen skulle identifiseres med tallene 3 og 5. Vi navnga først klassen *InvalidCreditCardException*. Dette endret vi derimot litt senere, slik at vi ikke låste oss til kredittkort, men at klassen heller kunne brukes på flere områder, og dermed være litt mer universell.

Oppgave 4

Deloppgave 1 – Ulemper ved egne exceptions:

Unødvendig kompleksitet, lager ting som allerede fins, det som finnes er basert på godt kodegrunnlag. Dersom vi lager egne exceptions kan vi utelukke små faktorer som har stor betydning for koden, som den eksisterende exceptions pakken allerede har inkludert. Koden blir også lettere å forstå for andre utviklere, siden de eksisterende pakkene er velkjent og utviklere slipper å sette seg inn i ny kode.

Deloppgave 2 – Hvordan skrive gode feilmeldinger:

Feilmeldinger må være presise, konkrete, men ikke oppgi sensitiv informasjon. En god feilmelding kan se slik ut: *Customer ID -5 is invalid. ID must be a positive integer.* Feilmeldingen skal påpeke hva som er feil, og fortelle hvordan man unngår feilen.

Deloppgave 3 – Når skal man bruke exceptions:

Exceptions bør brukes når metoder ikke kan fullføre sin oppgave på grunn av interne eller eksterne feil, for eksempel at en fil ikke finnes. For forventede scenarier, som at en bruker angir feil passord, er det ofte bedre å returnere en statuskode eller en boolsk verdi.

Deloppgave 4 – Dokumentasjon rundt exceptions:

Det er viktig å dokumentere hvilke exceptions en metode kan kaste, slik at utviklere vet hva de kan forvente og hvordan de kan håndtere dem. Dette forbedrer kodekvaliteten og gjør det lettere å vedlikeholde systemet over tid.

Deloppgave 5 – Har vårt syn på exceptions endret seg?

Ja, etter diskusjonene har definisjonen blitt tydeligere. I starten var fokuset kanskje kun på å kaste exceptions, men nå ser vi at god exception-håndtering handler om når og hvordan de brukes, samt hvordan man kan gjøre koden mer robust og lesbar.

Nye innsikter:

- Exceptions bør kun brukes på uventede feil, ikke for flytkontroll
- Gjenbruk av standard exceptions gjør koden mer forståelig og enklere å vedlikeholde.

- Klare og presise feilmeldinger hjelper både utviklere og brukere.

Best practices:

- Bruk eksisterende exceptions, fremfor å lage custom hvor det ikke er nødvendig
- Gode og tydelige navn og feilmeldinger
- Velg flere exceptions som er forventet, ikke én som tar alt.
- Ikke vis sensitiv informasjon i feilmeldinger

Kilder til Workshop 4:

Marquette, K. (2017). *All .net Exceptions List*. Powershellexplained.com.

<https://powershellexplained.com/2017-04-07-all-dotnet-exception-list/>

Gewarren (2024, May 2). *Best practices for exceptions - .NET*. Microsoft.com.

<https://learn.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions>

Jez. (2015, March 10). *What exception should be thrown for an invalid ID?* Stack Overflow. <https://stackoverflow.com/questions/28970912/what-exception-should-be-thrown-for-an-invalid-id>