

Objektorienteret programmering 1.HF

It & Data, Odense

Rada - August 2017

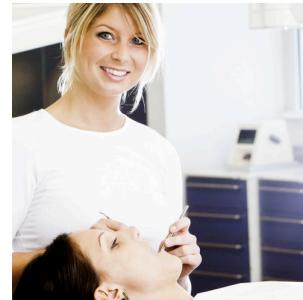
Generel information

- Faget **Objektorienteret programmering** har en varighed på 15 skoledage. 5 skoledage på **1.HF.** og de resterende 10 skoledage på **2.HF.** Faget afsluttes på 2.HF. og resulterer i en standpunktskarakter for faget.
- Faget er opdelt i to dele:
 - **Teoretisk del**, hvor læreren giver oplæg eller kommer med eksempler på projektor eller tavle.
 - **Praktisk del**, hvor eleven arbejder selvstændigt med de stillede opgaver.
- Faget afsluttes med en individuel evalueringssamtale med læreren, hvor eleven demonstrerer de færdige opgaver og læren stiller uddybende spørgsmål, denne evaluering er tidssat til ca. 10 min.
- Der kan forventes hjemmearbejde i det omfang det er nødvendigt.



Niveauer og evaluering

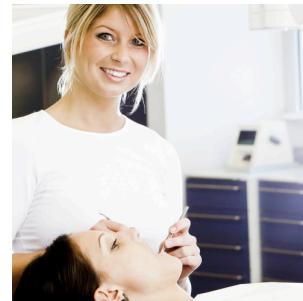
- Faget **objektorienteret programmering**, bedømmes på **3** niveauer.: **Rutineret**, **Avanceret** og **Ekspert**.
- Det er **valgfrift**, hvilket niveau man vil evalueres på, men alle starter som udgangspunkt på **Rutineret**.
- Des højere **niveau**, des højere **selvstændighedsgrad** forventes, samt **redegørelse** for valg og beslutninger skal kunne **uddybes** og **understøttes**.
- Inden evaluering, skal man tage stilling til hvilket niveau man vil bedømmes på.



Forventninger

- Jeg forventer af dig, at du er arbejdsmød, møder til tiden og **deltager aktivt** i timerne.
- Jeg forventer at du **overholder pausetiderne**.
- Jeg forventer at du lytter **opmærksomt** til oplæg og **stiller spørgsmål**, når du kommer i tvivl.
- Når du arbejder med de praktiske opgaver eller når jeg holder oplæg, er du altid **velkommen** til at **stille spørgsmål**.
- Ræk bare hånden op ☺

HUSK: Man kan ikke spørge for meget, men man kan spørge for lidt – det er din uddannelse og dit ansvar at lære noget!



Objekter og klasser

Introduktion til objektorienteret programering
(OOP)

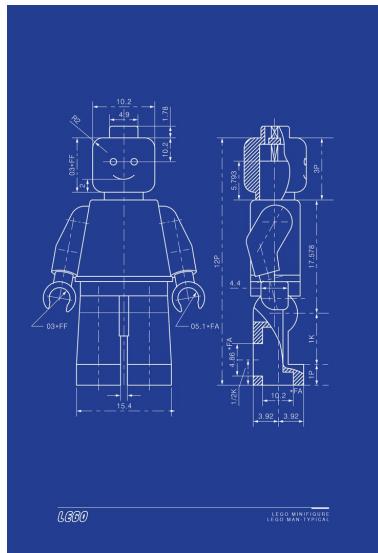
Hvad er et objekt?

- **Hvad er OOP?** Objekt Orienteret programmering er et **koncept**, hvormed man kan udvikle software (applikationer, programmer, websites, services, App's, IOT eller noget helt tredje). Konceptet handler om, at **afrænse** data og **funktionalitet** i **selvstændige entiteter**, også kaldet **objekter**.
- Et **objekt**, kan være alt fra:
en person, en bil, en beregning, en plante, et dyr, en terning, et produkt, en ordre osv.
Generelt, forsøger vi med objekter, at **modellere** den **fysiske** og den **abstrakte** verden ind i vores program som objekter.
- **Hvorfor?** – fordi det giver **overskuelighed, genkendelighed** og gør det nemt, at **genanvende** og **skalere**.
Det bliver nemmere, at udvikle applikationer.
- **Hvordan?** – **opdeling** af programkode i **mindre** og sammenhængende **enheder**, der kan afvikles, testes, videreudvikles og genanvendes.
- Et **objekt** er en **instans** af (et produkt af) en **klasse** og der kan **instantieres** mange objekter af én klasse.
- I en komplet OOP applikation/program, er der som oftest mange objekter af mange forskellige klasser med forskellige relationer.

Hvad er en klasse (Class)?

- En **klasse** er **grundstenen** i Objektorienteret Programmering (**OOP**).
 - En **klasse definerer et objekt** -Hvilket betyder, at en klasse er en **skabelon** for et **objekt**. Det er klassen der definerer, hvad et objekt kan (methods), hvilke egenskaber det har (properties), hvilke data objektet holder (fields) og måden hvorpå man tilgår det.

Class



Instantierung



Objekt



Klassens bestanddele

- En **klasse** i C# har altid et **Navn**. Vi navngiver klassen ud fra hvilket objekt, der kan instantieres af klassen eks. *LegoMan*. Klassenavnet begynder altid med **Stort** og anvender **CamelCase**.
- Klasser kan relatere til hinanden på flere forskellige måder: Nedarvning, interfaces, abstraktion – de forskellige relationer gennemgås senere.
- Klassen kan have en **Constructor**. En **Constructor**, er en metode, der automatisk kaldes ved instantiering af et nyt objekt af klassen.
- Klassen kan have en **Destructor**. En **Destructor**, er en metode, der automatisk kaldes ved et instantieret **objekts ophør** med at eksisterer. En destructor bruges typisk til at afbryde evt. dataforbindelser, lukke en åben fil eller lignende "oprydning", der ikke håndteres af .NET frameworkets garbage collector.
- Klassen kan have **Fields** (klasse variabler).
- Klassen kan have **Properties**
(med get og/eller set metoder).
- **Demoeksempel:** *LegoMan* klassen

Indkapsling (incapsulation)

- Et **objekt** er en **instans** af en **klasse**, og klassen er grundlæggende et lille selvstændigt og lukket program. Det skal forstås på den måde, at klassens bestanddele, som udgangspunkt, kun kan tilgås fra klassen selv.
- Det er udvikleren af klassen, der definerer, hvordan vi kan tilgå klassens bestanddele udefra. På denne måde, sikre vi vores kode fra at blive anvendt forkert og dermed brække programmet.
- **Indkapsling** er en **grundsten i OOP**.

Access modifiers

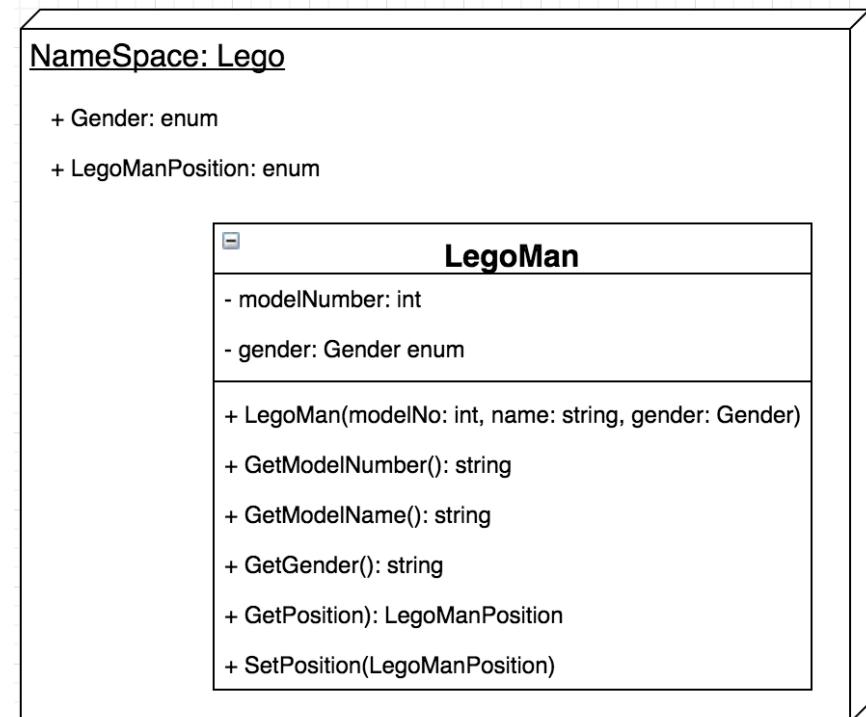
- **Adgangen** og anvendelsen af en klasses **fields**, **properties** og **metoder** kan **kontrolleres** med såkaldte **Access Modifiers**.
- I **C#** findes der **4** forskellige **Access Modifiers**: **Public**, **Private**, **Protected** og **Internal**.
- Vi angiver vores **Access Modifiers** i deklarationen af henholdsvis selve klassen og klassens: fields, properties og metoder.
- **Public** – kan tilgås direkte på et objekt af klassen, også udefra.
- **Private** – kan KUN tilgås i klassen selv.
- **Protected** – kan tilgås fra klassen og fra nedarvede klasser.
- **Internal** – kan tilgås fra samme assembly (exe, dll).

UML: Klasse diagram

- Konceptuelt klasse diagram
 - Vi kan starte med at modellere vores OOP design med et konceptuelt klasse diagram.



- Detaljeret klasse diagram



UML: Klasse diagram: relationer

- **Association**

- Når et objekt, bruger et andet objekt.
- Ex. Når et objekt overføres (parses) til et andet objekt, som en parameter i en metode.

- **Aggregation**

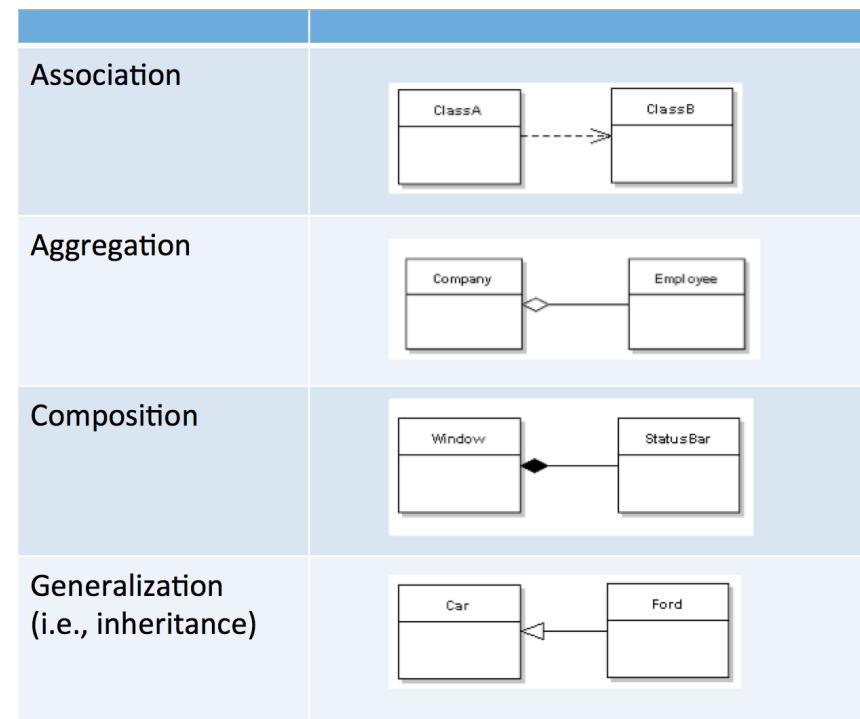
- Når et objekt tilhører et andet objekt.
- Ex. Når et objekt, der er overført (parsed) til et andet objekt, gemmes i en property.

- **Composition**

- Når et objekt er bundet til et andet objekt.
- Ex. Når et objekt instantieres i et andet objekt. Et Composite objekt, dør med dens "parent" objekt.

- **Generalization**

- Når et objekt er nedarvet fra et andet objekt.
- Ex. Nedarvning, implements interface



Nedarvning

- En **klasse** kan **nedarves** fra en anden klasse. På denne måde, arver en klasse al funktionalitet fra ovenstående klasser. Med nedarvning sparer vi kode og dermed tid.
- I en klasse der arver en ovenstående klasse, kan vi:
 - Genbruge arvet funktionalitet
 - Udvide arvet funktionalitet
 - Overskrive/ændre arvet funktionalitet
- Nedarvning giver dynamik til OOP og gør det nemmere at udvikle, vedligeholde og genbruge.
- Find fællesnævnerne for objekter og overvej om der er grubund for nedarvning.
- Eks. Forestil dig et bil objekt!

Polymorfi

- **Polymorfi** er et **begreb**, der findes i **OOP** og som betyder at **klasser** kan have **forskellig** funktionalitet, selvom de deler samme **interface**.
- I praksis betyder det: at to forskellige klasser, der arver fra samme ovenstående klasse, kan have forskellig funktionalitet i den samme metode (med samme navn).
- **Eks.**
Forestil dig klassen **køretøj**, der definerer en **metode** der hedder: **Brems()**. Køretøj **nedarves** til klassen: **Bil** og klassen **Cykel**.
Både klassen **Bil** og klassen **Cykel** **overskriver** den arvede metode **Brems()** og implementerer, hver sin måde at bremse på.
 - Bil -> Brems() -> "træd på bremsen"
 - Cykel -> Brems() -> "træk i håndbremsen"

Interface

- Et **Interface** er en entitet, der indeholder metoder **uden** implementeret **kode**
– altså ”tomme” metoder.
- Det er udelukkende **deklarationen** af **metoden**, der er i et interface.
- Interface entiteter, bruges til at lave en ”**kontrakt**” med klasser, der implementerer interfacet. Interfacet definerer overfor klassen, hvad der skal implementeres af kode.
- I praksis, bruger vi ofte **Interface** entiteter, som et **API** indgang til adskilte stykker af programkode.
- Eks. Forestil dig at en gruppe (GRP1) af udviklere, har lavet en logisk adskilt (encapsulated) del af et program. En anden gruppe (GRP2) har lavet en anden del af et program og de skal nu have kommunikation mellem programdelene.
GRP1 har defineret et Interface, der fortæller GRP2, hvordan de skal implementere kommunikationen, med deres kode. Dette gøres ved at GRP2 implementerer GRP1's Interface og er dermed tvungen til at implementere de angivne metoder i deres arvede klasse.

Abstrakte klasser

- En abstrakt klasse minder om et interface, men til forskel for Interfacet er det en klasse.
- Der kan **ikke** instantieres objekter af abstrakte klasser, den kan kun arves af underliggende klasser.
- En abstrakt klasse kan have både abstrakte metoder, der ikke har nogen implementeret kode og reelle metoder med færdig implementeret kode.
- En klasse, der arver en abstrakt klasse, tvinges gennem en "kontrakt" til at færdiggøre (implementere) de abstrakte metoder.
- En abstrakt klasse kan bruges til at sikre, at underliggende klasser, overholder de angivne metoder. Samtidig kan underliggende klasser gøre brug af den funktionalitet, der stilles til rådighed gennem de almindelige metoder.

Opgave



- Du skal lave et konsol-baseret Ludospil.
- Det er vigtigt, at spillet er fuldt objektorienteret og overholder præmisserne for OOP.
- Der SKAL laves et UML klassediagram.
- Man kan spille Ludo, med 2 – 4 spillere.
- Spillereglerne for Ludo, kan læses her:
<http://www.papskubber.dk/braetspil/ludo>
- **Udvidelser:**
 - Lav en variant af dit ludospil, der anvender reglerne for Globeludo. Terningen har globus og stjerne, samt visse felter på pladen.
 - Lav AI (Artificiel Intelligence) robotter, der spiller de tomme pladser om bordet. Således der altid er 4 spillere med.
 - Lav evt. Sværdhedsgrad/snyd på "robotterne"

