DM575

Forelæsning 06: Nedarvning og undtagelser

Casper Bach
Institut for Matematik og Datalogi

SDU

Baseret på originalslides af Robin Kaarsgaard Sales

Tætte afhængigheder

- Det hænder, at koncepter i programdomænet naturligt har meget tætte afhængigheder.
- **Eksempler:** `Ansat` → `Person`. `Rektangel` → `Form`. `Hest` → `Dyr`.
- Hvad har alle disse afhængigheder til fælles?

Tætte afhængigheder

- Det hænder, at koncepter i programdomænet naturligt har meget tætte afhængigheder.
- **Eksempler:** `Ansat` er en `Person`. `Rektangel` er en `Form`. `Hest` er et `Dyr`.
- Hvad har alle disse afhængigheder til fælles?

OO Princip #3: Nedarvning

- **Nedarvningsprincippet:** Klasseafhængigheden “er en” udtrykkes ved hjælp af nedarvning.
- Ved at udtrykke disse afhængigheder igennem nedarvning kan
  - underklasser automatisk arve egenskaber og metoder fra deres superklasser, og
  - objekter fra (mere specialiserede) underklasser bruges i stedet for objekter fra (mere generelle) superklasser.

Nedarvning i Java

- I Java udtrykkes nedarvning via nøgleordet `extends`.
- At en klasse `A` nedarver fra `B` betyder konkret at
  - alle metoder der tager et parameter af typen `B` kan tage et af typen `A` på dens plads i stedet, og
  - offentlige metoder og attributter fra `B` er også defineret på alle objekter af typen `A` (de “går i arv”).
- Private metoder og attributter er *ikke* tilgængelige for underklasser.
  - Brug i stedet nøgleordet `protected` hvis du vil gøre en metode eller attribut tilgængelig for underklasser (og andre klasser i samme pakke).

SOLID: **Liskov Substitution Principle**

- **Liskov Substitution Principle:** Programkomponenter der bruger objekter af en given klasse skal være i stand til at bruge objekter af en underklasse uden at vide det.
- **Mere formelt:** Alle de egenskaber som en programkomponent har når den agerer på objekter af superklassen skal den have når den agerer på objekter af underklassen. Det skal med andre ord ikke være muligt at kende forskel.

Tilsidesættelse

- Underklasser kan “overskrive” metoder der allerede er defineret på deres superklasse (medmindre denne er defineret som `final`). Dette kaldes *tilsidesættelse* (*overriding).*
- I Java kan en instans af en underklasse tilgå instansen af superklassen ved hjælp af nøgleordet `super`.
  - Dette tillader, at en tilsidesættende metode kan udvide superklassens opførsel ved at kalde superklassens metode som en del af sin erklæring.
  - Brug af annoteringen `@override` fortæller Java-oversætteren, at du har til hensigt at tilsidesætte en eksisterende metode: hvis en sådan metode ikke findes på superklassen genereres en fejl.
- For konstruktører kan superklassens konstruktør kaldes som `super(…)`.

SOLID: Open-Closed Principle

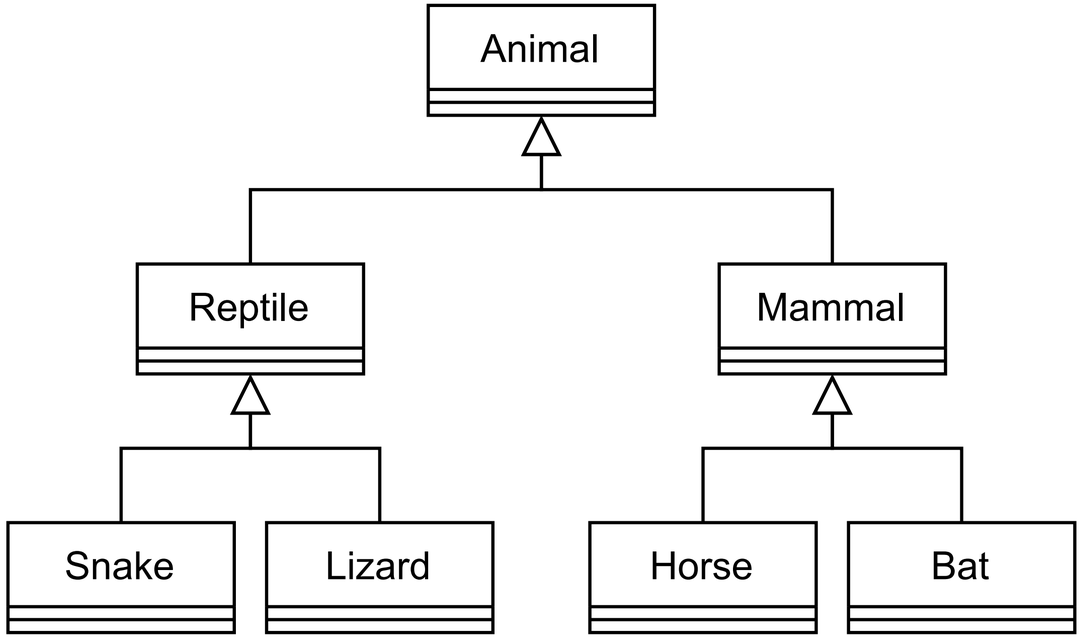
- **Open-Closed-princippet:** Programkomponenter skal være åbne for udvidelse men lukkede for ændringer.
- Med andre ord skal det være muligt at udvide programkomponenter med nye egenskaber og opførsel, men ikke at ændre den allerede definerede opførsel.
- I Java kan vi udvide klasser via nedarvning, og udvide metoder via tilsidesættelse.

Nedarvning vs. implementering

- Nedarvning beskriver et særligt tæt forhold imellem klasser (“er en”-afhængigheder).
- At en klasse implementerer en grænseflade beskriver blot, at den har nogle særlige kapaciteter (altså, at den kan opføre sig på en bestemt måde).
- I Java kan klasser implementere mange forskellige grænseflader, men de kan højst nedarve fra en klasse.
  - Hvad kan gå galt hvis vi tillader nedarvning fra flere klasser?

Klassehierarkier

- En klasse kan kun have en superklasse.
- En klasse kan dog have mange underklasser, som hver især også kan have mange underklasser. Dette kan give (til tider komplicerede) *klassehierarkier*.
- I UML indikeres nedarvning med en fuldt markeret (ikke stiplet) hvid pil.



Abstrakte klasser

- Nogle klasser er så generelle, at de svarer mere til *begreber* end konkrete objekter.
- Disse kaldes *abstrakte klasser*, og indeholder en eller flere metoder, der ikke er definerede, men som alle (ikke-abstrakte) underklasser skal definere.
  - I Java erklæres abstrakte klasser og deres abstrakte metoder med `abstract`-nøgleordet.
- **Eksempel:** Alle køretøjer kan køre. Et køretøj nedarver derfor fra klassen `Vehicle` der bl.a. har en metode `public void run()`.
  - Da køretøjer kan køre på mange forskellige måder kan vi ikke allerede nu sige, hvordan ethvert køretøj vil køre, hvorfor denne metode kun kan defineres som abstrakt.
  - Når vi har et konkret køretøj (f.eks. `Car` eller `Bicycle`) kan disse nedarve fra `Vehicle` og erklære `public void run()` alt efter hvordan det virker.

Undtagelser

- Ikke alle beregninger går godt – faktisk kan beregninger tit gå galt hvis brugere giver inddata der ikke giver mening i forhold til objektet eller beregningen.
  - **Eksempel:** Et rektangel kan kun have positive sidelængder, så det giver ikke mening at lave en instans af `Rectangle` med en ikke-positiv sidelængde.
  - **Eksempel:** Man kan ikke dividere et tal med `0`.
- En beregning kan også fejle fordi der sker noget uventet (f.eks., at forbindelsen bliver lukket under en dataoverførsel).
- Java skelner imellem uventede fejl (*errors*) og fejlsituationer som programmer i rimelig grad må forventes at kunne håndtere (undtagelser; *exceptions*).
- Disse fejlsituationer kommunikeres til brugeren af programkomponenten ved *at kaste en fejl* (eller *undtagelse*).

Undtagelser

- Når en undtagelse kastes stopper al udførsel, og metoden vender tilbage til kaldstedet med en fejlmeddelelse.
- Undtagelser håndteres af en passende kontrolstruktur.
- Håndteres en undtagelse ikke bliver den sendt yderligere op igennem kaldstakken indtil nogen håndterer den.

Undtagelser i Java

- I Java kan en metode i udgangspunktet ikke kaste en undtagelse, uden at det er markeret i metodens signatur med en `throws`-erklæring.
    - **Undtagelse:** Enhver metode kan kaste såkaldt *ukontrollerede undtagelser*, som er dem, der nedarver fra klassen `RuntimeException`. Ukontrollerede undtagelser kræver *ikke* en `throws`-erklæring.
- For at håndtere undtagelser benyttes `try`-`catch`-strukturer.
    - Undtagelser følger klassehierarkiet, så hvis en blok fanger undtagelser af klassen `B`, og `A` er en underklasse af `B`, så vil den blok også fange undtagelser af klassen `A`.

Til VS Code!

- Lad os kaste nogle undtagelser og fange dem igen.