

Referansegruppe

- To har meldt seg så langt
 - Datateknologi
 - Lektor matematikk & informatikk
- Helst en til fra et annet studieprogram

Agenda for idag

- Fler eksempler på innkapsling, med flere samhandlende objekter og objekt-typer
 - Validering, synlighetsmodifikatorer, tilgangsmetoder
- Innkapsling i forhold til kobling/avhengigheter mellom objekter.
- Noen viktige metoder alle objekter har:
 - toString, equals, hashCode
- Mer om unntak, hvordan de håndteres og noen forskjellige typer unntak
- Litt generell koding med collections.

Fra sist:

- OO - sikring av gyldig tilstand med innkapsling
 - valideringsmetoder og unntak
 - Synlighets-modifikatorer
 - tilgangsmetoder
 - innkapsling og implementasjon
- Java
 - valideringsmetoder og unntak
- VS Code
 - generering av tilgangsmetoder og get/set



Mer om Innkapsling

- Et objektorientert program vil bestå av flere typer objekter. Mange av disse vil samhandle på en eller annen måte.
- Et type objekt kan avhenge av andre objekter for å utføre sine oppgaver, og der derfor avhengig av disse.
- Innkapsling er også å unngå unødvendige avhengigheter, slik at systemet ikke blir unødig komplekst.
- Det er forskjellige måter avhengigheter oppstår.

Eksempel: BokSamling

Skal holde på en samling bøker med bokmerke.

- Trenger en Bok å forholde seg til en BokSamling den tilhører? Vi prøver oss først uten det.
- BokSamling må i alle fall ha en oversikt over bøkene i den.
- Den aller enkleste løsningen er at BokSamling kun er en liste, men da har vi ikke bidratt med noe...

Eksempel: BokSamling

Hva ønsker vi å gjøre mer enn å ha en liste?

- Typiske liste-metoder: Legge til og fjerne bøker
- Skrive ut eller returnere (kopi) av liste av bøker
 - Sortering etter antall sider ulest, antall sider, ...?
 - Kun de som er påbegynt?
- Finne en bok? Eksempler kan være
 - Finne en bok med en gitt tittel, eller del av tittel?
 - Finne en bok med færrest uleste sider, så vi kan lese den ferdig
 - BokSamling må i alle fall ha en oversikt over bøkene i den.
- La også legge til at **vi kan ikke ha mer enn et visst antall bøker i samlingen, angitt i konstruktøren til samlingen.**
- La oss også gi samlingene et navne-felt.

Metoder som alle objekter har

Det ligger en underliggende mal for alle objekter. Den er definert av klassen `Object`. Alle andre klasser arver metodene fra denne, selv om vi ikke definerer dem eksplisitt. Vi skal se på noen av dem her

- `String toString()`: En streng-representasjon av objektet. Default er klasse-navn (inkludert package) etterfulgt av `@` og hashcode til objektet.
- `Int hashCode()`: Default returnerer et unikt heltall for objektet, typisk minneadressen, men dette er implementasjonsavhengig.
- `boolean equals(Object other)`:

Det finnes noen fler, som vi ikke skal snakke om her.

toString()

Er men å returnere en streng-representasjon. Standard-verdien viser ikke noe av innholdet, kun klassenavn og et tall.

Vi kan redefinere (override) toString f.eks. slik for Bok:

```
@Override
public String toString() {
    return "Bok [tittel=" + tittel + ", antallSider=" + antallSider +
        ", bokmerke=" + bokmerke + "]";
}
```

VSCoDe + Java Extension kan gjør det for oss:

Høyreklikk i editoren, velg Source actions... og velg Generate toString()

Likhet av innebygde typer og objekter

En variabel deklarerert som en innebygd type (int, float, double, boolean, char, byte, short, long) lagrer selve verdien

- $B = A$ kopierer verdien i A og lagrer i B
- $A == B$ er sann når A og B lagrer lik verdi

En variabel deklarerert som en objekt type (alt annet enn de innebygde), lagrer en referanse til objektet

- $B = A$ kopierer referansen i A og lagrer i B. A og B refererer til samme objekt
- $A == B$ er sann nøyaktig når A og B lagrer like referanser, dvs. at de peker til samme objekt, eller at begge er null.

hashCode()

En del datastrukturer og algoritmer bruker hash-koder for å gjøre en del operasjoner mer effektive, f.eks. HashMap for å effektivt hente verdier basert på nøkler (tilsvarende dictionary i Python)

Standard hashCode er kun basert på objekt-identiteten. Hvis vi redefinerer equals(), så må vi passe på at **like objekter gir samme hashCode**.

Velg Source action, velg Generate hashCode og equals...

Likhet av innebygde typer og objekter

Vi ønsker ofte (vanligvis?) å sammenligne innholdet til to objekter. Dette tas hånd om av equals-metoden som alle objekter har. I en egendefinert klasse så har den en default implementasjon som ser slik ut:

```
Public boolean equals(Object other) {
    return this == other;
}
```

Mange klasser i Java har implementert det vi vanligvis ønsker. For eksempel for

hashCode

Hvis vi redefinerer equals, så må vi redefinere hashCode. Java krever følgende kontrakt:

Hvis `A.equals(B)`, så skal

`hashCode(A) == hashCode(B)`

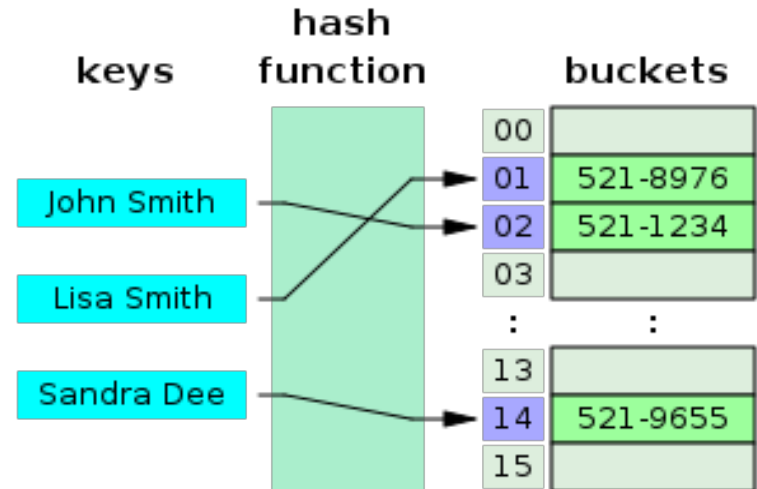
Hash-koden må beregnes på grunnlag av alle data som inngår i equals-sammenligningen.

(hashCode returnerer int, som ikke er et objekt, så denne har ikke noe equals-metode)

Litt bakgrunn: Hva er en hash?

- En effektiv måte å lagre og hente frem verdier.
- Den mest effective måten å hente en Verdi, er å vite hvor den ligger, f.eks. Ved å vite indeksen I en array.
- Hvis det er veldig mange mulige oppslags-nøkler, så kan vi ikke holde av plass til alle.
- Hash-funksjonen regner ut i stedet en *hashverdi* som gir plasseringen, både for lagring og oppslag.

Illustrasjon av hashMap



Jorge Stolfi,,

https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg#/media/File:Hash_table_3_1_1_0_1_0_0_SP.svg

Forskjellige nøkler KAN ha like hasher, som må håndteres, men hash-funksjoner designes slik at kollisjoner vanligvis er sjeldne.

Når skal vi definere egen equals og hashCode?

Hvis objektene i koden har en intrinsik identitet i programmet, dvs. representerer seg selv, så bør vi ikke redefinere.

Eksempel:

I et spill, en player er et objekt, og det er kun ett objekt som representerer den samme spilleren.

For «verdiobjekter» er det naturlig å redefinere equals basert på innholdet.

Eksempel:

String, == og equals

== er litt spesiell for objekter av type String.

Strenger som lages som «literals», lagres som objekter i en felles pool.

```
String s1 = «hei»;
```

```
String s2 = «hei»;
```

I den andre linjen vil Java sjekke om det allerede finnes en streng som er lik «hei», og så la s2 peke til denne, i stedet for å lage nytt objekt.

Derfor vil `s1 == s2` være sant her.

Derimot, så vil

```
String s3 = new String(«hei»);
```

opprette et nytt objekt, med likt innhold. Da blir

```
s1 == s3;      // false
```

```
s1.equals(s3); // true
```

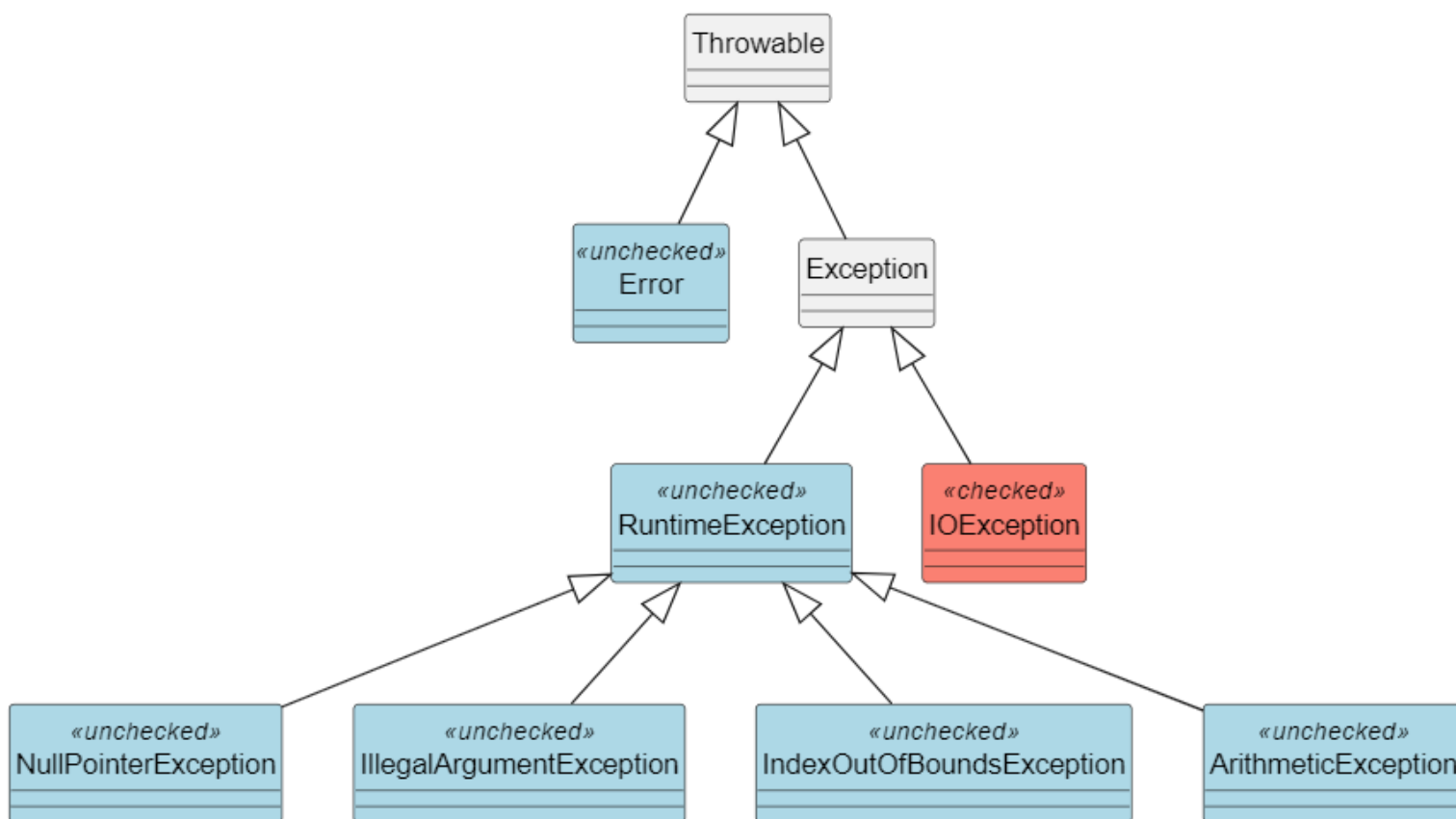
Avhengigheter mellom Klasser/Objekter

- I Bok & BokSamling var det ingen kobling fra Bok til Boksamling.
- Hvis det finnes flere samlinger en bok kunne ha tilhørt, så kan en ikke finne den fra boka selv. Da må en spørre boksamlingene.
- Hvis en får tilgang til bok via en samling, så trenger vi ikke noe kobling fra Bok.
- Hvis ikke, så vil det være litt jobb å finne samling(ene) med boka, hvis noen. Eller en kan ha en referanse til samlingen, men da er det viktig at disse synkroniseres.

Noen variasjoner

Noen typer unntak

- Unntak er objekter. De forskjellige typene er organisert i et hiarki. De lenger ned er spesialiseringer. Dette kalles «arv» i OO.



Håndtering av unntak

- Unntak håndteres ved å legge kode som kan utløse unntak i en try – catch blokk.

```
try {
    // Block of code to try
}
catch(Exception e) {
    // Block of code to handle
    errors
}
finally { //valgfritt, gjøres
    uansett utfall
}
```