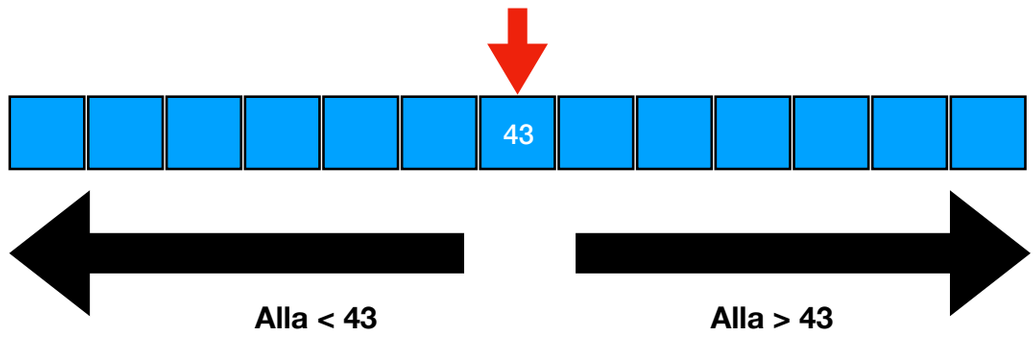
Objektorienterad Programmering DAT043

Föreläsning 10
13/2 -18
Moa Johansson
(delvis baserat på Fredrik Lindblads material)

Sökning och Sortering: Binärsökning

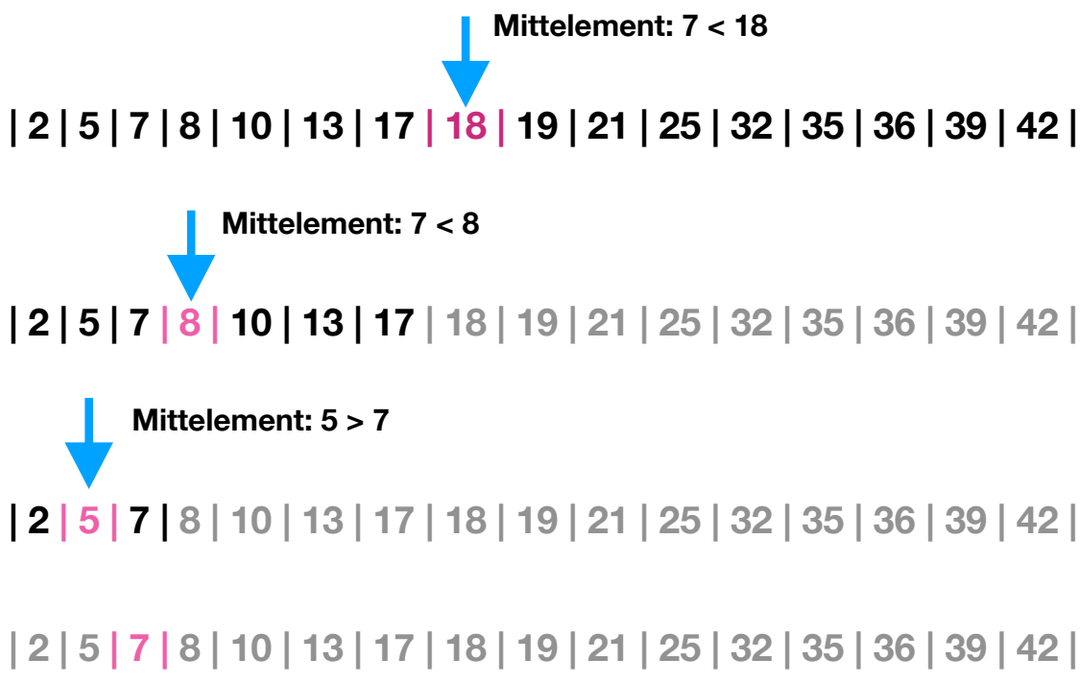
- Problem: Hitta ett element i en godtycklig lista/array.
  - I värsta fall: måste söka igenom hela.
- För sorterade listor kan vi använda mer effektiv s.k. binärsökning.
- Se metoden binarySearch i klassen Arrays i Javas API



•Ska man leta efter element i en godtycklig lista eller array så måste man i värsta fall leta igenom hela.
•Om listan är sorterad så kan man hitta elementet snabbare m.h.a. binärsökning.
•Binärsökning går ut på att titta på elementet i mitten av listan. Om det element vi söker efter är mindre än mittelementet vet vi att vi kan avgränsa sökningen till vänstra halvan. Om det är större så högra halvan.
•I nästa steg tittar vi på mittelementet i den halvan som är kvar och gör likadant.
•Bland koden som hör till föreläsningen finns en implementering.
•Antalet element vi måste titta på är ungefär 2log(n) där n är listans längd.
•Sökning på detta sätt är vanligt. Metoden binarySearch finns i Javas API, i klassen Arrays.

DEMO: BinarySearch.java

BinarySearch: hitta 7



Jämför med koden i BinarySearch.java

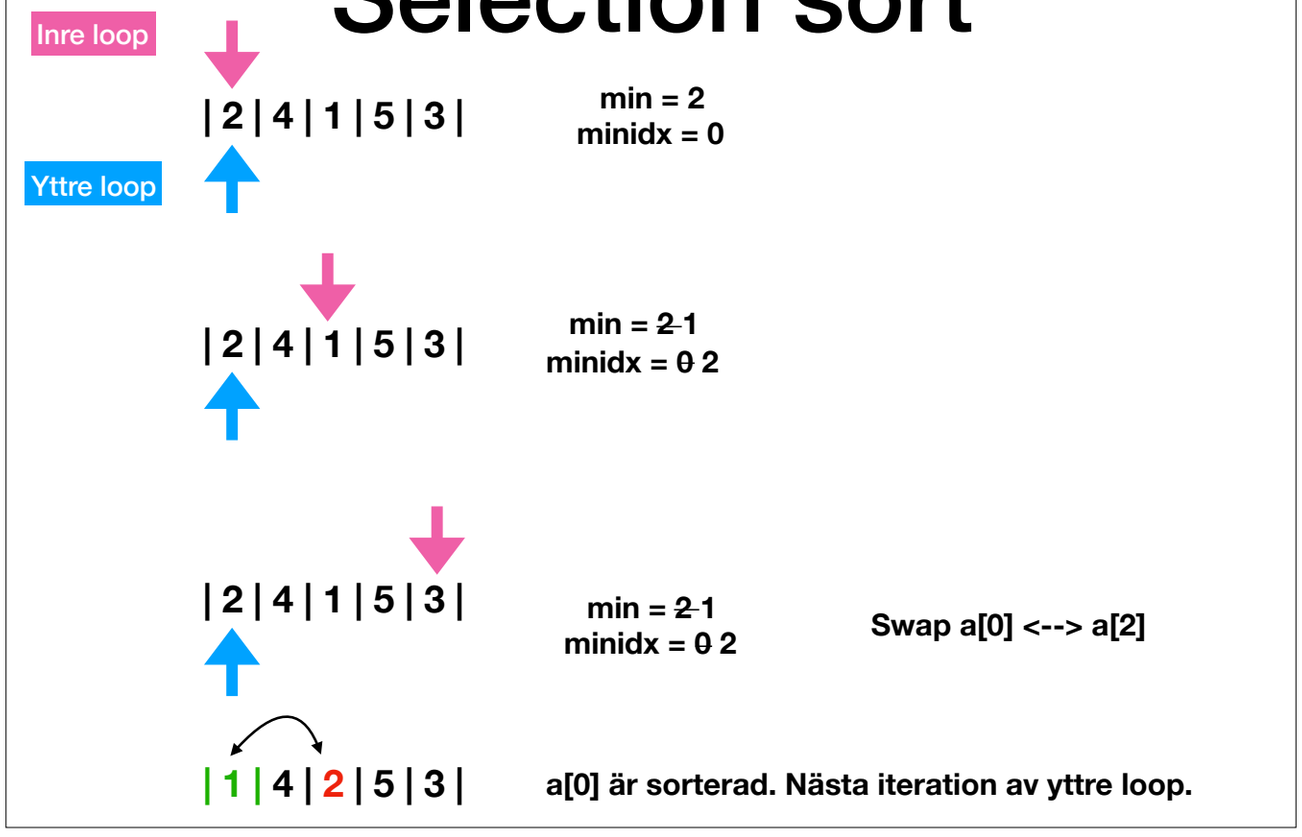
Sortering: enkla algoritmer

- *Selection sort* är en enkel algoritm för sortering:
  - De k första elementen i arrayen hålls hela tiden korrekt sorterade.
  - I varje steg: öka k, och välj det minsta av de kvarvarande elementen.
- *Insertion sort* är en annan enkel algoritm:
  - De k först elementen hålls sorterade.
  - I varje steg: öka k, sätt in nästa element på rätt plats i den sorterade delen.

•Sortera listor är också en vanligt förekommande uppgift och ett problem som studerats flitigt. Metoden sort finns i Arrays i Javas API.
•Enkla algoritmer behöver göra ungefär n2 steg, där n är listans längd.Exempel: insättningssortering (insertion sort), urvalssortering (selection sort), bubble sort
•En implementering av urvalssortering (selection sort) finns bland föreläsningens kod.
•Urvalssortering går till så att man hela tiden har de k första (minsta) elementet korrekt sorterade och i varje steg väljer det minsta av de kvarvarande elementen.
•I insättningssortering har man hela tiden de k första (i ursprungsarrayen) element korrekt sorterade. I varje steg sätter man in nästa element på rätt plats i den sorterade delen.

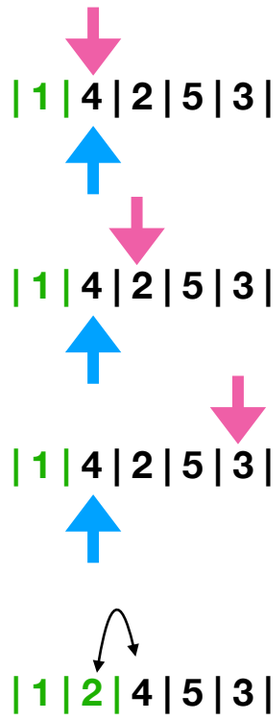
DEMO: Sort.java

Selection sort



Jämför med koden i BinarySearch.java

Selection Sort



min = 4
minidx = 1

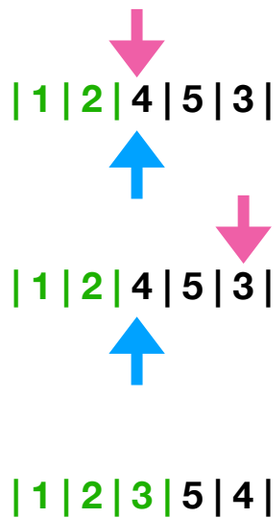
min = 2
minidx = 2

min = 2
minidx = 2

Byt a[1] <--> a[2]

a[0-1] är sorterad. Nästa iteration av yttre loop.

Selection Sort



min = 4
minidx = 2

min = 3
minidx = 4

Byt a[2] <--> a[4]

a[0-2] är sorterad. Nästa iteration av yttre loop byter plats på 5 och 4.
Därefter är listan sorterad.

Notera: Den yttre loopen går igenom arrayen från början till slut en gång. För varje iteration av den yttre loopen, går den inre loopen går igenom hela den osorterade delen av arrayen varje gång.

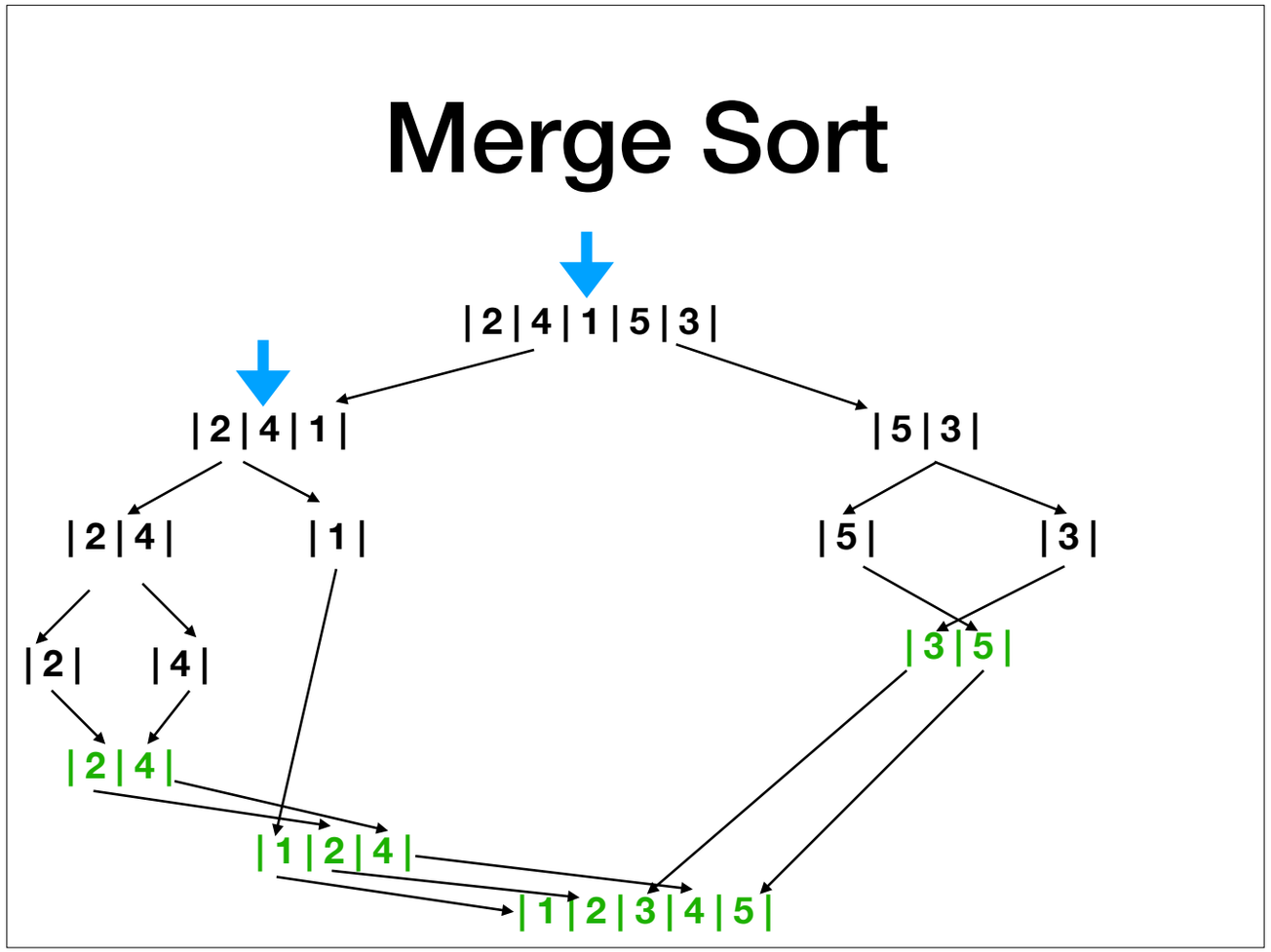
När man talar om komplexitet av en algoritm säger man därför att selection sort har tidskomplexitet O(n^2) där n är listans längd (typiskt för algoritmer med två nästlade loopar).

Sortering: snabbare algoritmer

- *Merge sort:*
  - Rekursivt: Dela upp i två halvor, sortera dessa. Slå ihop resultaten så att resultatet blir sorterat.
- *Quicksort:*
  - Dela i en grupp med större och en grupp med mindre element. Sortera dessa.
  - Slå ihop den mindre och större gruppen (trivialt).

•Smartare algoritmer behöver göre ungefär n·2log(n) steg.
•Exempel: mergesort, quicksort
•Varianter av mergesort och quicksort används i Javas Arrays.sort.
•En implementering av mergesort finns bland föreläsningens kod.
•Mergesort går till så att man rekursivt delar upp problemet i två halvor. När halvorna blivit sorterade så slå man ihop dem så att resultatet är sorterat.
•I quicksort så delar man istället upp elementen i en grupp med mindre och en grupp med större element. Sedan sorteras grupperna rekursivt. Eftersom alla element i ena gruppen är mindre än alla element i andra så får man resultatlistan helt enkelt genom att lägga de två sorterade dellistorna efter varandra.

DEMO: Sort.java



På ett litet exempel kan merge sort verka komplicerad. Men på längre listor vinner man på att dela problemet i två delar rekursivt. Komplexiteten blir ungefär n·2log(n) jämfört med n^2 för selection sort.

Testning

- Köra programmet och se om det gör som väntat?
- Automatisk testning: program som kör och utvärderar tester.
- Lätt att köra tester igen.
- Lägga till nya tester vid förändringar.
- *Regression testing:* vid ändringar kör alltid tester igen.

•Testning kan vara manuell eller automatisk.
•Manuell testning innebär att en person kör ett program och ser om det gör som väntat.
•Detta är långsamt och resurskrävande.
•Man har insett i IT-branschen sedan många år tillbaka att testning bör utföras mer systematiskt.
•Automatisk testning innebär att man skriver program som utför testerna och avgör om programmet gjorde som det ska.
•Med automatisk testning kan man skriva ett antal testfall en gång för alla.
•Med jämna mellanrum kan man sedan testa systemet som utvecklas och se om någon bugg tillkommit.
•När systemet utvecklas och fler delar kommer på plats kan man fylla på listan med tester så att även de nya delarna testas.

Automatiserad Testning

- **Testdriven utveckling:**
  - Testfall skrivs först.
  - Därefter kod.
  - Fungerar även som dokumentation/specifikation. Tvingas tänka på vad programmet förväntas göra innan kodning.

- **Property based testing:**
  - *Formell specifikation* av vad programmet ska göra.
  - Testfall genereras automatiskt.
  - E.g. QuickCheck.

•S.k. testdriven utveckling ser detta som helt centralt och låter utvecklingen drivas av att man först skriver testfallen och sedan utvecklar koden så att den klarar av testfallen.
•Automatiska testfall fungerar också som en formell dokumentation av kravspecifikationen som talar om vad systemet ska klara av.
•En typ av automatisk testning är sådan där testfallen inte skrivits manuellt utan automatiskt genererats utifrån en formell specifikation av programmets förväntade beteende.
•Detta används fortfarande inte så mycket, men Quickcheck är ett exempel.

Testfall

- **Pre-condition:** Vilka villkor måste indata uppfylla?
- **Post-condition:** Vad ska uppfyllas för att ett test ska anses lyckat?
- Hur vet man att man testat tillräckligt?
  - Alla “vägar” i koden testade?
  - Kommit ihåg “corner cases”?

•När man skriver testfall är det viktigt att tänka på vilka pre- och postconditions som gäller för metoden man testar.
•Alla tester måste anropa metoden med indata som uppfyller precondition. Implementationen av metoden behöver inte hantera indata som inte gör det.
•När man sedan avgör om ett test lyckades så är det postconditions man utgår ifrån. Om inte dessa är uppfyllda så gjorde inte metoden vad den skulle.

JUnit

- Bibliotek för att skriva unit-tests (tester av metoder).
- Integrerat i många IDE:er (t.ex. Eclipse).
- S.k. *assertions* avgör om ett test lyckas eller ej (t.ex. `assertNull`)

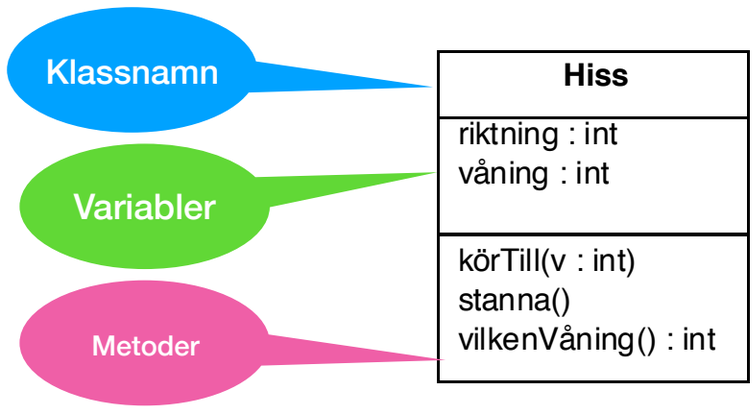
```
@Test
void test1() {
 MyLinkedList<Integer> l = new MyLinkedList<Integer>();
  assertNull(l.first);
}

@Test
void test3() {
 MyLinkedList<Integer> l = new MyLinkedList<Integer>();
 Integer i = 1;
  l.add(i);
  Integer j = l.get(0);
  assertEquals(i,j);
}
```

•För testning av Java används i stor utsträckning JUnit som är ett klassbibliotek avsett att enkelt skriva och hålla reda på enhetstester (unit tests), d.v.s. tester som testar olika delar av ett program och inte bara helheten.
•JUnit är integrerat i de flesta IDE:er.
•Man definierar klasser där varje klass innehåller metoder som gör ett test av en del av programmet.
•Metoderna som motsvarar ett test ger man attributet @Test
•I JUnit finns ett antal assertions som man använder för att rapportera om ett test lyckades eller inte, t.ex. assertTrue, assertEquals, assertNull etc...
•Sedan kan man enkelt köra alla tester och se resultatet.
•JUnit gör det också möjligt att gruppera tester i olika sviter (test suites).

(Om ni är intresserade av detta, se kursen “Testing, Debugging and Verification”.)
DEMO: I Eclipse: MyLinkedListTest, BinarySearchnSortTest.java

UML - klass

- **Unified Modelling Language** - UML
- Grafisk notation för klasser, objekt och deras relationer.
- Stora projekt - analys och designskede.
  - Vilka klasser? Vilka instansvariabler? Hur förhåller de sig till varandra?

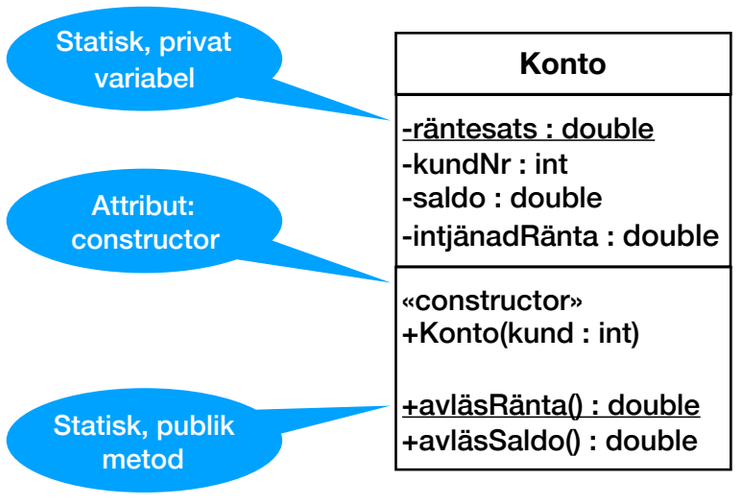


•Unified Modelling Language är en standardiserad grafisk notation för klasser och objekt och dess relationer.
•Används flitigt i objektorienterad programutveckling.
•I ett tidigt skede i ett projekt (analys och design) är UML-diagram en viktig del av det som dokumenterar systemet.
•En klass representeras av en ruta med tre sektioner, namn, variabler och operationer/metoder.
•Variabler och operationer kan ha typer och signaturer
•

•Man kan ange synlighet hos klassmedlemmar. + betyder public, - private och # protected.
•Det finns vissa attribut som man kan använda i UML. Dessa skrivs inom tecknen «».
•«constructor» anger att en operation är en konsktruktor.
•«interface» anger att en typ är ett interface och inte en klass.

UML - klass

- Kan ange synlighet för variabler och metoder.
- Kan ange statiska variabler och metoder med <u>understrykning</u>

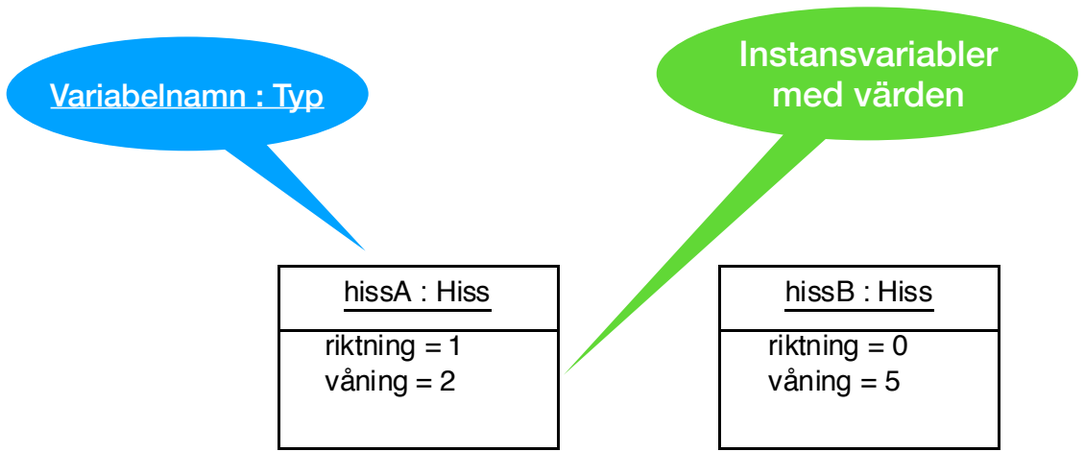


•Unified Modelling Language är en standardiserad grafisk notation för klasser och objekt och dess relationer.
•Används flitigt i objektorienterad programutveckling.
•I ett tidigt skede i ett projekt (analys och design) är UML-diagram en viktig del av det som dokumenterar systemet.
•En klass representeras av en ruta med tre sektioner, namn, variabler och operationer/metoder.
•Variabler och operationer kan ha typer och signaturer
•

•Man kan ange synlighet hos klassmedlemmar. + betyder public, - private och # protected.
•Det finns vissa attribut som man kan använda i UML. Dessa skrivs inom tecknen «».
•«constructor» anger att en operation är en konsktruktor.
•«interface» anger att en typ är ett interface och inte en klass.

UML - instanser

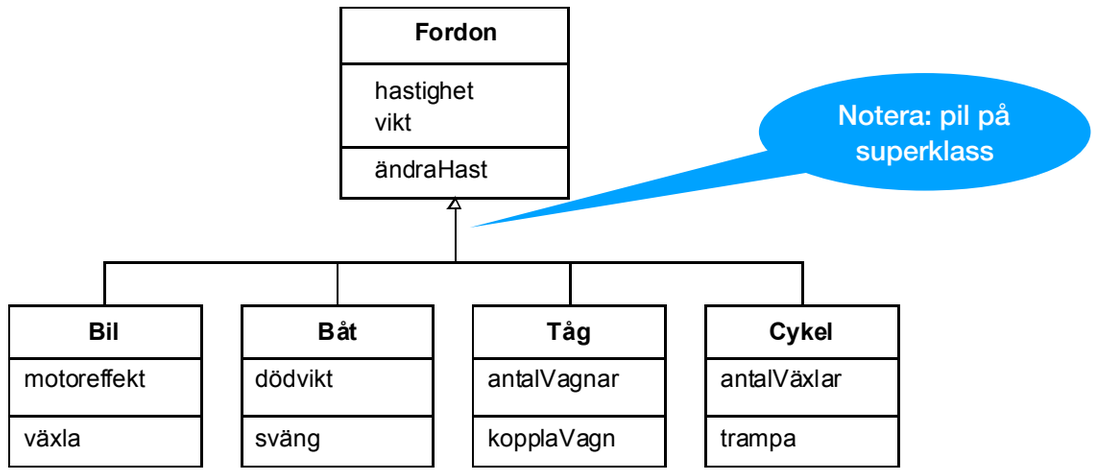
- Specifika instanser kan återges i UML.
- Beskriver ett tillstånd i systemet.



•Objekt/instanser kan också återges i UML. För dessa är namnet understruket och typat. Namnet kan utelämnas.
•Förutom detta innehåller ett objektdiagram instansvariablernas värden.
•Instanser med deras värden beskriver ett specifikt tillstånd snarare än en modell av systemet.

UML - Relationer mellan klasser

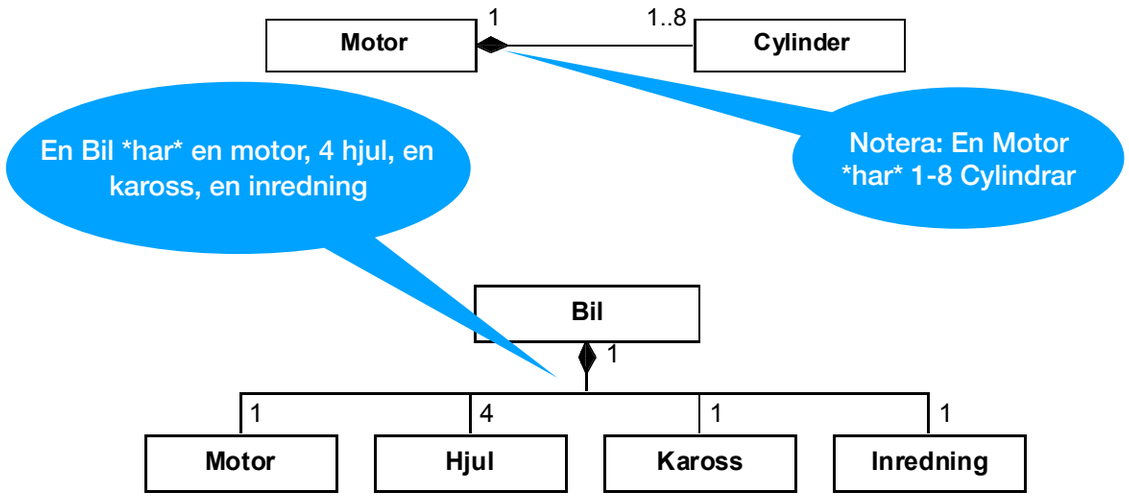
- Relationerna **är**, **har**, **känner till**.
- Subklassrelation: Bil **är** ett slags Fordon.



•Inom objektorienterad programmering brukar man prata om tre typer av relationer – är, har och känner till.
•I UML säger man att klasser är associerade om de har något med varandra att göra.
•Om B är en A så är B en slags A, d.v.s. B är en subklass till A. Det representeras av linje med trekants-pilspets vid superklassen.

Relationer mellan klasser

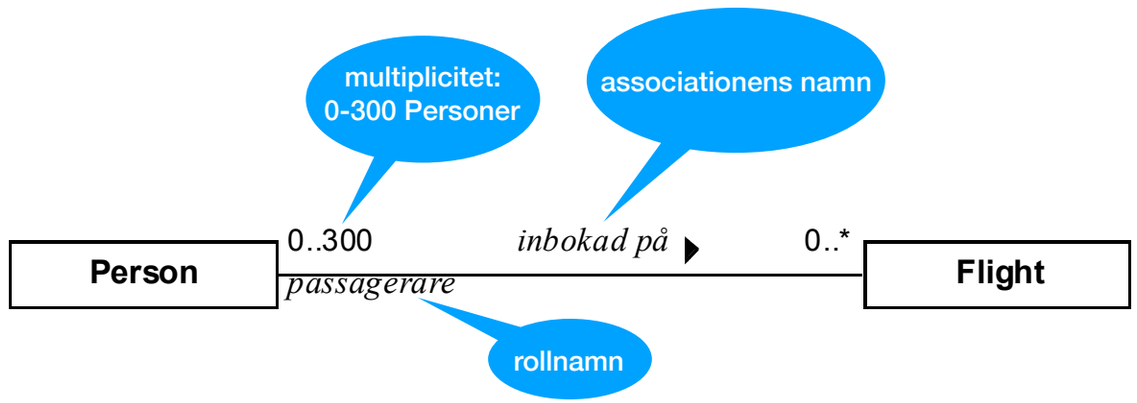
- **Relationen har:** Klass A har (en eller flera) instansvariabler av typ B.



•Om en klass A har en klass B så är en instans av A uppbyggd av en eller flera instanser av B.
•När man är mest intresserad av relationerna mellan klasser så kan man utlämna medlemmar och bara ha klassens namn.
•En sådan association har en ifylld romb i den ändan som har den andra.
•För har-associationer kan man ange s.k. multiplicitet vid ändarna av associationen.
•Det kan vara ett tal eller ett intervall. n..* betyder n eller fler. * betyder 0 eller fler.
•Multipliciteten hos den klassen som har är alltid 1.
•I Java innebär att A har B att det finns en variabel i A som är ett B eller en samling av B.

Relationer mellan klasser

- Relationen ***känner till***: Klass A *känner till* klass B om en instans av A refererar till/är **associerad** med någon instans av klass B.



•En klass A känner till en klass B om en instans av A refererar/är associerad till en eller flera instanser av klassen B.
•Associationen kan ha multipliciteter och vara oriktad.
•Man kan ange rollnamn för associationen på klassernas ändar.
•Man kan ange associationens namn och vilken riktning den ska läsas i.

•Skillnaden mot har-relationer är att om en A har en B så existerar inte denna B som annat än en del av A.

Relationer mellan klasser

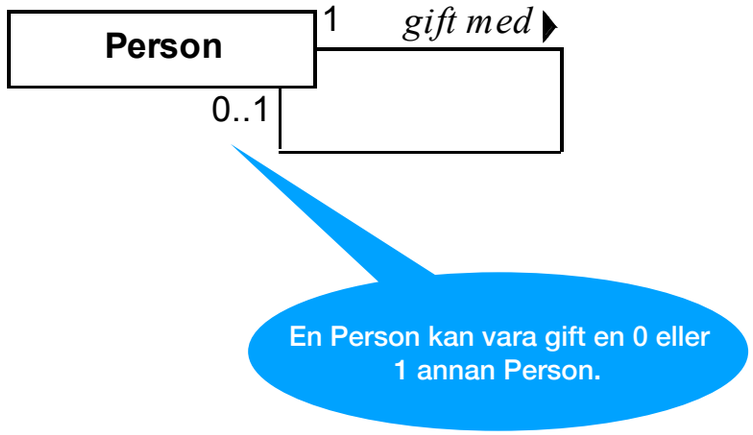
- Riktade associationer: Bara en klass känner till den andra.
- Skillnad mot *har-relation*: Om A *har* en B existerar bara detta B som en del av A.
- Om A **har** en B, så ska *inte B förändras utan att gå via metoder i A.*
- Om A **känner til**l B, så kan B *förändras oberoende* av A.

•Associationen kan också vara riktad, d.v.s. bara ena klassen känner till den andra. Man använder då en pil för att ange riktning (här känner kundregistret till ett antal personer, men inte tvärt om.
* En romb vid Kundregistret anger att den är ett s.k. “aggregat”, d.v.s. består av ett antal Personer.
* När romben är vit betyder det att aggregatet “känner-till” vilka delar som ingår i det.

•Även känner till-relationer mellan en klass A och en klass B innebär att det finns variabler av typen A i B eller tvärtom eller både och.
•Skillnaden mot har-relationer är att om en A har en B så existerar inte denna B som annat än en del av A.
•Implementationsmässigt så innebär det t.ex. att man inte exponerar objekt som en klass A har så att de kan förändras utan A:s vetskap och om man kopierar en instans av A så kopierar man också objekten som A har (man använder inte samma referenser i kopian).

Relationer mellan klasser

- En klass kan vara associerade till sig själv med känner-till relationen.



•Klasser kan vara associerade till sig själv
med känner till-relation.

•Här har vi bara tittat på de viktigaste delarna av klassdiagram och lite på objektdiagram.
•Dessa är exempel på s.k. strukturella UML-diagram. Det finns flera andra sorters strukturella diagram, t.ex. paketdiagram och komponentdiagram.
•Det finns också s.k. behavioral diagrams, t.ex. sekvensdiagram, användarfallsdiagram, tillståndsdiagram och kommunikationsdiagram.