DV017A – Inledande Programmering i Java

Studieanteckningar

Jonas Sjöberg
Högskolan i Gävle

tel12jsg@student.hig.se
https://www.github.com/jonasjberg

Abstrakt

Anteckningar baserade på powerpoint-slides konverterade från ".ppt" till textbaserade format, därefter utökade och/eller modifierade med eget material. Detta är mina Egna studie-/föreläsningsanteckningar och kan således mycket väl innehålla faktafel. Dokumentet uppdateras löpande under kursens gång.

Licensiering

Föreläsning 1

2015-06-08 måndag

Om programmering och problemlösning

Användningsområden och typer av arbetsuppgifter:

- Problemlösning och beräkningar:
  - Raketbanor
  - Hållfasthet
  - Växthuseffekten
- Administration, hantering av data:
  - Löner
  - Formulär

  - Order och lager
- Simulering
  - Spice
  - Fysik
  - Matematik
- Grafik
  - Visualisering
  - Spel
- Fler områden?
  - Multimedia
  - ..

Två fundament:

- **Datastruktur** (objekt)
  - Representation av data i datorns minne.
  - Ordnad uppsättning variabler/data i en särskild datastruktur.
- **Algoritmer**
  - Det som datorn ska göra med data, någon slags beräkning.
  - Uppsättning väldefinierade instruktioner för att lösa en uppgift.
  - Från givna utgångstillstånd med säkerhet leder till ett givet sluttillstånd.
  - Systematisk procedur med ändligt antal steg löser ett problem.

Datastrukturer

Hur ska vi representera information som t.ex:

- Antal dagar?
- Lön?
- Hastighet?
- Ränta?
- Personer?
- Rätt och fel?
- Ekvationer?
- Musik?
- E-post?
- Aktier?

Särskilt välformulerade svar på den här frågan och övriga relaterat till objektorientering i Java: “The progress of abstraction” - Thinking in Java, 3rd ed. Revision 4.0

Föreläsning 2 - Numeriska typer/värden

2015-06-22 måndag

Värden och typer

I java finns två huvudtyper av värden:

1. **Primitiva värden** – Inbyggda i språket
2. **Referenser** – Referensvärden (adresser till objekt)

Alla värden har en väldefinierad typ.

Primitiva värden och typer

Inbyggda värdetyper (fördefinierade) kan delas in i:

1. **Numeriska** värden och typer:
   - `byte`
   - `short`
   - `int`
   - `long`
   - `float`
   - `double`
2. **Teckenvärde** och typ: (är dock numerisk)
   - `char`
3. **Sanningsvärde** och typ:
   - `boolean`

Typ	Storlek	Minst	Störst	Kommentar
boolean	odefinierat			sant eller falskt

Typ	Storlek	Minst	Störst	Kommentar
char	16 bitar	`\u0000`	`\uFFFF`	Unicode-tecken
byte	8 bitar	-2^7	$2^7 - 1$	$-128 - 127$
short	16 bitar	-2^{15}	$2^{15} - 1$	$-32768 - 32767$
int	32 bitar	-2^{31}	$2^{31} - 1$	$-2147483648 - 2147483647$
long	64 bitar	-2^{63}	$2^{63} - 1$	$-9.223372037 \times 10^{18} - 9.223372037 \times 10^{18}$
float	32 bitar	-3.4×10^{38}	3.4×10^{38}	*IEEE 754* floating point
double	64 bitar	-1.7×10^{308}	1.7×10^{308}	*IEEE 754* floating point

Heltalstyper

Heltalstyperna `byte`, `short`, `int` och `long` hanterar heltalsvärden och variabler.

- Typerna väljs beroende på storlek och effektivitet.
- Normalt används typen `int`.

`byte`

- Typen `byte` är ett 8-bitars signerat tvåkomplements heltal med ett minvärde på -128 och ett maxvärde på 127 (inkluderande).
- `Byte` typen kan vara användbar för att spara minnesanvändning i stora arrayer.
- Det begränsade tal som kan lagras kan också indikera tänkt användning och bidra till en "självdokumenterande" kod.

`short`

- Typen `short` är ett 16-bitars signerat tvåkomplements heltal med ett minvärde på $-32,768$ och ett maxvärde på $32,767$ (inkluderande).
- Riktlinjer lika som de för `byte`, använd för att spara minne i sammanhang där minnesanvändningen faktiskt spelar roll.

`int`

- Typen `int` är ett 32-bitars signerat tvåkomplements heltal med ett minvärde på -2^{31} och ett maxvärde på $2^{31} - 1$.

- I *Java SE 8* och senare kan `int` användas som en `unsigned` 32-bitars heltal, med ett minvärde på 0 och ett maxvärde på $2^{32} - 1$.
- Använd *Integer*-klassen för att använda `int` typen som en `unsigned int`. Klassen innehåller en del metoder för att göra t.ex. jämförelser.

`long`

- Typen `long` är ett 64-bitars tvåkomplements heltal.
- Den signerade varianten har värden mellan -2^{63} och $2^{63} - 1$.
- I *Java SE 8* och senare kan `long` användas för att representera en `unsigned` 64-itars `long`, med värden mellan 0 och $2^{64} - 1$.
- Den här typen ska användas då omfånget av typen `int` inte räcker till.
- Klassen *Long* innehåller metoder som `compareUnsigned`, `divideUnsigned`, etc, för att stödja aritmetiska operationer på `unsigned long`.

Refererens för detaljer: Primitive Data Types, Oracle Java Documentation

Flyttalstyper

Flyttalstyperna `float` och `double` hanterar flytttal (decimala tal).

- Typerna väljs beroende på storlek och effektivitet.
- Normalt används typen `double`

`float`

- Typen `float` är ett "single-precision 32-bit *IEEE 754* floating point" tal.
- För möjliga värden, se Floating-Point Types, Formats, and Values, Java Language Specification.
- Har lägre precision än `double` men är snabbare.
- Räkna med 7 siffrors noggrannhet (decimaler)

`double`

- Typen `double` är ett "double-precision 64-bit *IEEE 754* floating point" tal.
- För möjliga värden, se Floating-Point Types, Formats, and Values, Java Language Specification.
- Den här typen är generellt standardvalet för decimaltal.
- `double` har högre precision än `float`.
- Räkna med 15 siffrors noggrannhet (decimaler)

Teckenvärde/typ

- Hanterar tecken. Är av typ `char`. Exempelvärden: `'a'`, `'b'`, `'Z'`
- Alla tecken har ett numeriskt värde.
  - **ASCII-tabellen**
    ASCII står för "*American Standard Code for Information Interchange*" Det är en 7-bitars kod. Många 8-bitars teckenkoder (t.ex. *ISO 8859-1*) innehåller ASCII i den "lägre" delen. Den internationella motsvarigheten till ACSII är *ISO 646-IRV*.
  - Kompakt tabell från Release 3.74 av Linux man-pages project:

```
   2 3 4 5 6 7       30 40 50 60 70 80 90 100 110 120
 -------------     --------------------------------
0:   0 @ P ` p     0:    (  2  <  F  P  Z  d   n   x
1: ! 1 A Q a q     1:    )  3  =  G  Q  [  e   o   y
2: " 2 B R b r     2:    *  4  >  H  R  \  f   p   z
3: # 3 C S c s     3: !  +  5  ?  I  S  ]  g   q   {
4: $ 4 D T d t     4: "  ,  6  @  J  T  ^  h   r   |
5: % 5 E U e u     5: #  -  7  A  K  U  _  i   s   }
6: & 6 F V f v     6: $  .  8  B  L  V  `  j   t   ~
7: ´ 7 G W g w     7: %  /  9  C  M  W  a  k   u  DEL
8: ( 8 H X h x     8: &  0  :  D  N  X  b  l   v
9: ) 9 I Y i y     9: ´  1  ;  E  O  Y  c  m   w
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
F: / ? O _ o DEL
```

    ASCII(7) – Linux Programmer's Manual

  - Idag används oftast inte ASCII!

- Typen `char` är ett 16-bitars *Unicode*-tecken. Det har ett minvärde på `\u0000` (decimalt 0) och ett maxvärde på `\uFFFF` (65.535 inklusivt).

Sanningsvärde/typ

- Typen `boolean` hanterar sanningsvärden, två möjliga; sant eller falskt.
- Typen används för enkla "flaggor" som sparar sant/falskt- förhållanden.
- `boolean` representerar en bit av information men dess faktiska storlek är inte precist definierad i Java.
- Fås som resultat av jämförelser och logiska operationer, t.ex: `if (a > b)`

Konstanta värden - literaler

- När man i programkod skriver ett literalt värde får den en typ (default):

exempelvärden	defaulttyp
`1`, `2`, `8865`, `-45`	int
`0.5`, `.55`, `78.`	double
`4.6E23`, `1.0e-100`	double

- Man kan även uttryckligen bestämma typ:

```
123L  (long)
3.14F (float)
```

Andra talbaser för konstanter

- Ibland är det lämpligt att använda t.ex. **hexadecimal** form:

Hex	Dec
`0x0A`	10
`0xFF`	255
`0x1000`	4096

- Eller **oktal** form:

Oct	Dec
`010`	8
`011`	9
`012`	10
`0100`	64

- Anledning är "direktmappningen" till binära tal:

Hex	Binärt
`0x11`	`0001 0001`
`0xFF`	`1111 1111`
`0x1F`	`0001 1111`
`0xF1`	`1111 0001`

- Tal lämpade för "bitmasking"-tekniker:

Hex	Binärt
`0x01`	`0000 0001`
`0x02`	`0000 0010`
`0x04`	`0000 0100`
`0x08`	`0000 1000`
`0x10`	`0001 0000`
`0x20`	`0010 0000`
`0x40`	`0100 0000`
`0x80`	`1000 0000`

Variabler för primitiva värden

- För varje primitiv typ kan deklareras motsvarande variabler
- Symboler som "minns" värden, d.v.s. namngivna minnesceller.
- Variablernas värden kan givetvis förändras. Detta sker med tilldelningar.

Exempel:

```
summa = 1250;
```

Variabeln summa innehåller nu värdet 1250.

Deklaration av variabler

- Variabler har alltid en typ.
- Måste deklareras innan de används!
- Java har *statisk typning*, variabler tilldelas en datatyp vid kompilering.
  - Variabeln behåller typen under resten av dess livslängd.
  - Språk som *C*, *C++, Java* och *Pascal* är statiskt typade.

    > "Java is both a statically and strongly typed language. The term static refers to compile time or the source code, whereas dynamic refers to the runtime or the bytecode. A programming language is said to use static typing when type checking is performed during compile time as opposed to runtime. In static typing, all expressions have their types determined at compile time, prior to when the program is executed. The word strong in the context of typing means that rules about the typesystem are enforced prior to the code being run. Thus, Java is considered to be both statically and strongly typed."
    > *Java Programming Learn Advanced Skills From a Java Expert*
    > Poornachandra Sarang, Orace Press 2012

Syntax:

```
modifierare variabeltyp variabelnamnslista;
```

Variablers räckvidd (scope)

- Variablerna existerar endast i det kodblock där de har deklarerats.

```
foo
{
    int i = 7;
    // Här är 'i' åtkomlig
}
// Utanför blocket kommer man inte åt det 'i' som deklarerats inuti blocket.
```

- Olika scope – *överskuggning*

```
foo
{
    bar
    {
        int i = 100;
        /* Här kommer vi åt 'i' med värdet 100. */

        baz
        {
            int i = 7;
            /* Här inne kommer vi åt 'i' med värdet 7. */
        }

        /* Här kommer vi åt 'i' med värdet 100. */
    }

    /* Här kommer vi inte åt något 'i' alls. */
}
```

Olika typer av variabler och scope

- **Instansvariabler** (attribut)
  - Scopet gäller i hela objektet.
- **Lokala variabler**
  - Scopet gäller i en metod eller i ett inre block.
- **Parametervariabler** får sitt värde vid metodanrop.
  - Scopet gäller lokalt i metoden.

```
foo(int a)
{
    /* Här finns 'a'. */
}
```

- **Klassvariabler**
  - Gäller i hela klassen och är samma för alla objekt av den klassen.

```
Class X
{
    private int instansmedlem  = 10;
    private static klassmedlem = 20;

    public void metod1(int parameter)
```

```
    {
        int lokal = 25;
    }

    public void metod2(int parameter)
    {
        int lokal = 26;
    }
}
```

Uttryck (expressions)

- Ett program som exekverar kan beräkna värden.
- De delar av programkoden som gör beräkningar av värden kallas "uttryck".
- Ett uttryck kan innehålla konstanta (literala) värden och variabler kombinerade med operatorer; `+ - * / %`.
- Alla beräknade värden får någon av de typer vi har studerat.
- Numeriska värden kan inte blandas med t.ex. sanningsvärden.

```
10 * 9 + true   (Felaktigt!)
```

Numeriska uttryck

- Kombinationer av konstanter, variabler och aritmetiska operatorer.

```
int a = 2, b = 8;
((a + b) * 10) / 5;
```

- Beräknas enligt prioritetsregler som bestäms av de ingående operatorerna.
- Parentes-par har högst prioritet.

```
(10 * 10) / 5
100 / 5
20
```

Aritmetiska operatorer (numeriska)

Aritmetiska operatorer: `+ - * / % ++ --`

Operator	Funktion	Beskrivning
`%`	modulus	Ger rest vid heltalsdivision.
`++`	inkrement	Ökar en heltalsvariabel ett steg.
`--`	dekrement	Minskar en heltalsvariabel ett steg.
`++a`	prefix	Uppräkning sker *innan* värdet hämtas.
`a++`	postfix	Uppräkning sker *efter* att värdet hämtats.

Exempel på användning av modulus:

- Enkel rest vid heltalsdivision:

```
10 % 7 == 3
```

- Begränsa variabel till ett visst område. Här begränsas `i` till $0 - 10$:

```
int a[10];

for (int i = 0; true; i = (i + 1) % 10) {
    /* .. använd a[i] .. */
}
```

- Omvandla tid i sekunder till timmar, minuter och sekunder:

```
timmar   =  sekunder / 3600;
minuter  = (sekunder / 60) % 60;
sekunder =  sekunder % 60;
```

- Begränsa `x` till att vara en multipel av 10:

```
temp = x - (x % 10);
```

Logiska uttryck

Logiska uttryck är kombinationer av sanningsvärden (konstanter och variabler) och *logiska operatorer*: `!`, `&&` och `||`

Exempel:

```
(hungry && dinnerTime) || candy
```

Sanningsvärden uppstår även med jämförelseoperatorer: `==`, `>`, `<`, `>=`, `<=`, `!=`

```
teenager     =  (age >= 13) && (age < = 19)
teenager     =  !((age < 13) || (age > 19))
mySalary    ==  yourSalary
myShoeSize  !=  myShoes
```

- Observera att operanderna till jämförelseoperatorerna är numeriska värden!

Logiska operatorer

Operator	Funktion	Beskrivning
`!`	icke	Logisk negation.
`&`	AND (och)	*true* om båda sidor är *true*, annars *false*.
		Verifierar båda operander.
`&&`	AND (och)	*true* om båda sidor är *true*, annars *false*.
		Avbryter utvärderingen om vänster sida är *true*.
`|`	OR (eller)	*true* om minst en sida är *true*, annars *false*.
		Verifierar båda operander.
`||`	OR (eller)	*true* om minst en sida är *true*, annars *false*.
		Avbryter utvärderingen om vänster sida är *true*.

`AND`/*och*-operatorn

`&&` (dubbelt och-tecken) är "och"-operatorn som returnerar *true* om både uttrycket till vänster och till höger om operatorn är *true*, annars *false*.

Beräkningen utförs med *lat beräkning* (lazy evaluation): Om uttrycket till vänster är *falskt*, så beräknas inte det högra uttrycket.

Detta kan vara användbart om man vill skriva en test där det högra uttrycket är giltigt endast om det vänstra uttrycket är *sant*:

```
a != null && a.full()
```

Referens: nada.kth.se: Svensk Javaterminologi

Skillnaden mellan `&` och `&&`

- **`&`**

  - boolean logical `AND`
  - **inte** "kortslutande"
  - Verifierar alltid båda operander.
  - Exempel på situation där det spelar roll:

    ```
    (x != 0) & (1/x > 1)     /* FARLIGT! Division med 0! */
    ```

- `&&`
  - logical `AND`
  - "Kortslutande"
  - Slutar utvärdera om den första operanden ger *falskt* då resultatet ändå blir *falskt*.
  - Exempel på situation där det spelar roll:

    ```
    (x != 0) && (1/x > 1)    /* Undviker division med 0. */
    ```

Referenser:

Stackoverflow: Difference between & and &&

Stackoverflow: Difference between & and && in Java?

Exempel för `&`

```
(x != 0) & (1/x > 1)
```

- Utvärdera `(x != 0)`, utvärdera sedan `(1/x > 1)` och utför till sist `&`.
- Problematiskt att ett "särfall" (exception) genereras om `x = 0`!

Exempel för `&&`

```
(x != 0) && (1/x > 1)
```

- Utvärdera `(x != 0)` och om det är *sant*
  - Gå vidare till att utvärdera `(1/x > 1)`.
- Klarar av fallet `x = 0` bättre då `(x != 0)` returnerar *falskt* och körningen avslutas uatn att `(1/x > 1)` testas.

`OR`/*eller*-operatorn

`||` (dubbelt lodstreck) är "eller"-operatorn som returnerar *true* om minst ett av uttrycken till vänster och till höger om operatorn är *true*, annars *false*.

Beräkningen utförs med lat beräkning (lazy evaluation): Om uttrycket till vänster är *sant*, så beräknas inte det högra uttrycket.

Detta kan vara användbart om man vill skriva en test där det högra uttrycket är giltigt endast om det vänstra uttrycket är falskt:

```
i == 0 || j/i > 8
```

- Referens:
  [Svensk Javaterminologi]

Skillnaden mellan `|` och `||`

- `|` does not do short-circuit evaluation in boolean expressions.
- `||` will stop evaluating if the first operand is true, but `|` won't.
- In addition, `|` can be used to perform the bitwise-`OR` operation on `byte`/`short`/`int`/`long` values. `||` cannot.

Stackoverflow: Why do we usually use `||` not `|`, what is the difference?

Exempel för `|`

```
exprA | exprB
```

- Utvärdera `exprA`, sedan utvärderas `exprB` och `|`.

Exempel för `||`

```
exprA || exprB
```

- Utvärdera `exprA` och om det ger *falskt*
  - Utvärdera `exprB` och `||`.

Tilldelningsuttryck

Syntax:

```
variabelnamn = uttryck;
```

Semantik:

- Vänstersidan måste vara en variabel.
- Högersidan evalueras först och måste vara ett uttryck (värde). Exempel:

```
a = 67 * 41 + 19  6;
```

Tilldelningen är ett uttryck i sig vilket tillåter: `a = b = c = 100 * 99;`

- Finns i flera versioner kombinerade med aritmetiska operatorer;
  `+=`, `-=`, `*=`, `/=`, `%=`
  Exempel:

Tilldelning	Ekvivalent
`a = a + 6`	`a += 6`
`a = a - 5`	`a -= 5`
`a = a * 4`	`a *= 4`
`a = a / 3`	`a /= 3`
`a = a % 2`	`a %= 2`

Tilldelningsuttryck och typomvandlingar

Typer har olika rang: t.ex. har `double` högre rang än `int`.

Ett värde med *lägre* rang kan alltid tilldelas en variabel med *högre* rang eftersom den *lägre rangens typ utgör en delmängd av den högre*, t.ex:

```
double d = 3;
```

Men inte tvärtom eftersom man förlorar precision:

```
int i = 3.14;                /* Ger kompileringsfel. */
```

Vi kan lösa det genom att göra en explicit typkonvertering (*cast*), t.ex:

```
int i = (int)3.14;           /* Görs om till 3. */
```

Uttryck med metodanrop

- I ett uttryck kan metoder anropas om de 'returnerar' ett värde (funktioner):

```
volume = circleaArea(15.0) * height;
```

- Om en metod är deklarerad som '`void`' och således inte returnerar något värde får ett sådant metodanrop **inte** ingå i ett uttryck.

  Följande är felaktigt:

  ```
  a = skrivUt("Jocke") + 100;
  ```

Konstanta "variabler"

- Modifierare *final* vid deklaration av variabler.

  ```
  final int KATTLIV = 9;
  final double PI = 3.141592;
  ```

- Dessa kan endast tilldelas ett värde EN gång.

  ```
  final int UNDERVERK;
  UNDERVERK = 7;
  ```

- Använd konstanter istället för explicita 'siffror' i programkod.
  - Om förutsättningarna förändras inför man förändringen på ett ställe istället för att göra det på många ställen.
  - Lättare att förstå programkoden.
  - Mindre risk för fel i programmet p.g.a. ofullständiga ändringar.

Föreläsning 3 – Klasser, objekt, referenser och selektioner

Klasser

Objektorienterad programkod består av klasser. Ofta en klass per textfil.

- En klass definierar objekt (som t.ex. ritningen för ett hus)
- En klass innehåller medlemmar
- Deklaration av variabler (attribut)
- Deklaration/definition av metoderEn klass

Syntax:

```
modifierare class klassnamn
kodblock
```

Ett kodblock inramas av { } och innehåller en sekvens av programsatser.

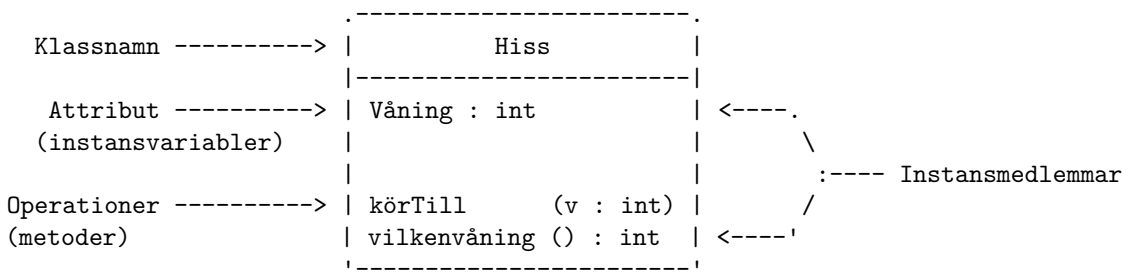
En klass

```
public class Test
{
    deklarationer av variabler (attribut)
    deklaration/definition av metoder
}
```

Variablerna och metoderna kallas instansmedlemmar.

```
public class Hiss
{
    private int våning = 1;

    public void körTill(int v) {
        våning = v;
    }

    public int vilkenVåning() {
        return våning;
    }
}
```

Klasser i UML



Klassmedlemmar

- Åtkomst av klassmedlemmar – *access control* och *modifiers*
- Begränsad åtkomst centralt koncept i Java. Åtkomst av metoder och fält bestäms med **åtkomstmodifierare**;
  - `private` anger att en medlem inte är åtkomlig från någon annan klass.
  - `protected` anger att en medlem är åtkomlig från klassens subklasser samt från övriga klasser i samma paket.
  - `public` anger att en medlem är åtkomlig från samtliga klasser.
  - Ett fält eller en metod har paketåtkomst om man inte anger någon åtkomstmodifierare. Det innebär att samtliga klasser som ligger i samma paket kan utnyttja fältet eller metoden.

Referens: nada.kth.se: Svensk Javaterminologi

Access Levels

Modifier	Class	Package	Subclass	World
`public`	yes	yes	yes	yes
`protected`	yes	yes	yes	no
no modifier	yes	yes	no	no
`private`	yes	no	no	no

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Controlling Access to Members of a Class, the Java Tutorials

Tips on Choosing an Access Level:

> If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.
>
> Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to. Avoid public fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.
>
> Controlling Access to Members of a Class, the Java Tutorials

Objekt

- Objektorienterade program "består av" objekt.
- Ett exekverande program kan beskrivas som ett antal objekt som anropar varandras metoder (operationer) och det som utför saker när programmet körs är programkod som är definierad i sådana metoder.
- Varje objekt har sitt eget minne som kan lagra objektets tillstånd/status i variabler (attribut).
- Objekten kommer åt varandra genom referenser som också lagras i objektets minne (variabler). Men objekten har ensamrätt över sitt minne. Inget annat objekt kan normalt komma åt och påverka minnescellerna.
- Det enda sättet för ett objekt att påverka ett annat är genom att anropa någon av dess metoder. På så sätt har objekten full kontroll av vad som sker.
- Metoderna och minnet är inkapslat i objekten. Detta kallas *information hiding*.

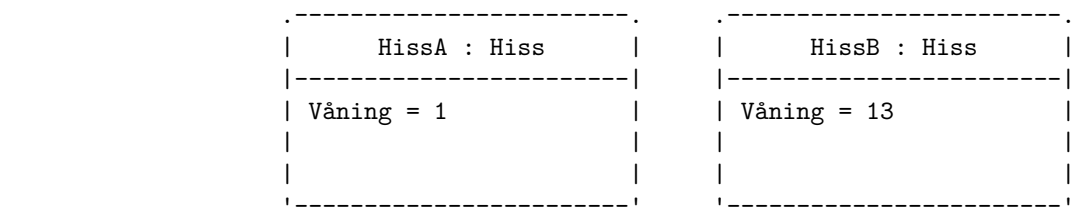
Skapande och användning av objekt

```
Hiss hissA = new Hiss();
Hiss hissB = new Hiss();

hissB.körTill(13);
```

- Operatorn `new` för att skapa objekt!
- Referensvariabler för att hålla i objekt.
- Punkt-operatorn för att komma åt medlemmar i objekt.

- Endast *public*-medlemmar kan kommas åt från andra objekt.
- *private*-medlemmar kan aldrig kommas åt utifrån.

Objekt i UML



Referenstyp/värde

- Objekt är en sorts värden, men i Java kommer man inte åt dem direkt. Man kommer åt dem via referenser.
- Referensvärden är värden som refererar till objekt. Detta kan förklaras som 'adressen' till objektet.

Ett referensvärde är antingen en referens till ett existerande objekt i run-time eller *null*.

Referensvariabler

- Referensvärden lagras i referensvariabler. Referensvariabler kan inte innehålla primitiva värden, utan endast objektreferenser.
- Syntaxen är densamma som för primitiva variabler
- Referensvariabler har en *typ* som är kompatibel med det objekt den refererar till (dess klass), referensvariabel till objektet `Bil` har en typ som är kompatabel med `Bil`.

Deklaration av referensvariabler

- Samma syntax som för primitiva variabler.

Generellt syntax:

```
modifierare  referenstyp  namnlista;
```

Exempel:

```
String a, b, c;             // Ja, String är faktiskt en klass
Automobile car1, car2;
```

- `a`, `b`, `c` kan referera till String-objekt
- `car1`, `car2` kan referera till Automobile-objekt

Att skapa objekt

- Objekt skapas med operatorn *new* enligt syntaxen:

```
new ClassName()
   (egentligen anrop av konstruktor)

new String("Jag är ett nytt objekt");

new Automobile();
```

Men hur kommer vi åt objektet?

- Objekten försvinner (‘dör’) om vi inte kan hålla fast dem med något handtag (referensvariabler)

Tilldelning av referensvariabler

Precis som för primitiva variabler.

Exempel:

```
variabelnamn = referensuttryck;

String s;                                   // Deklaration
s           = new String("Hejbaberiba!");   // Tilldelning
String t    = new String("Java");           // Initiering
String u    = "Konstant sträng";            // !
u = s;                                      // Vad händer här?
```

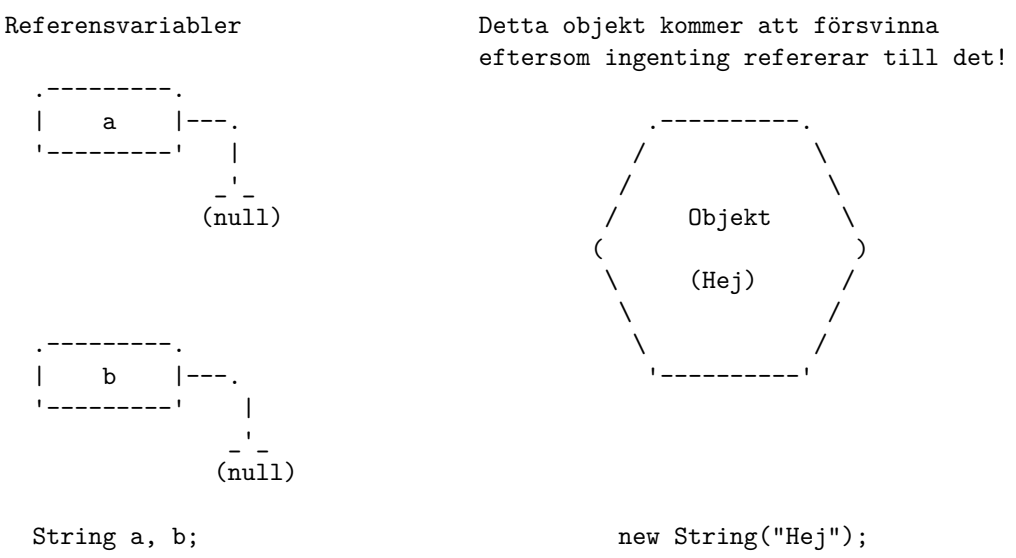
Objektens liv

- Ett objekt lever endast så länge någon referensvariabel refererar till det.
- För att "ta bort" ett objekt sätter man dess referensvariabel till null. Då 'dör' objektet och städas bort ur minnet, om ingen annan refererar till det

Exekvering av kod:

```
String a, b;
new String("Hej");
```

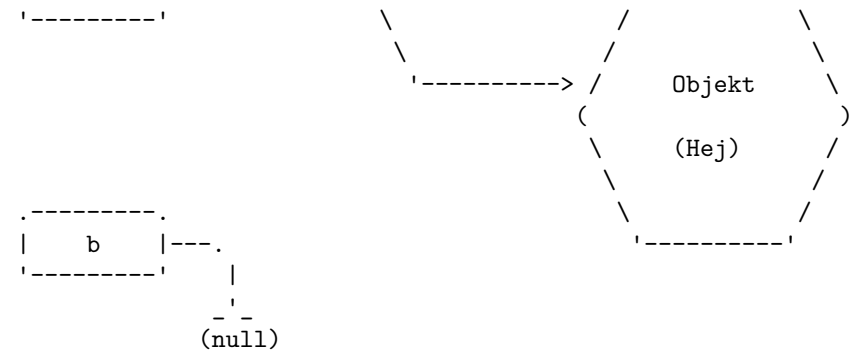
Modell:



Exekvering av kod:

```
a = new String("Hej");
```

Modell:

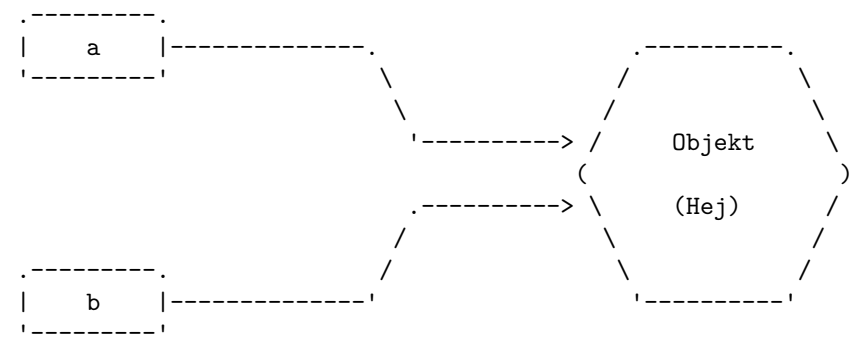


Exekvering av kod:

```
b = a;
```

Modell:

Referensvariabler

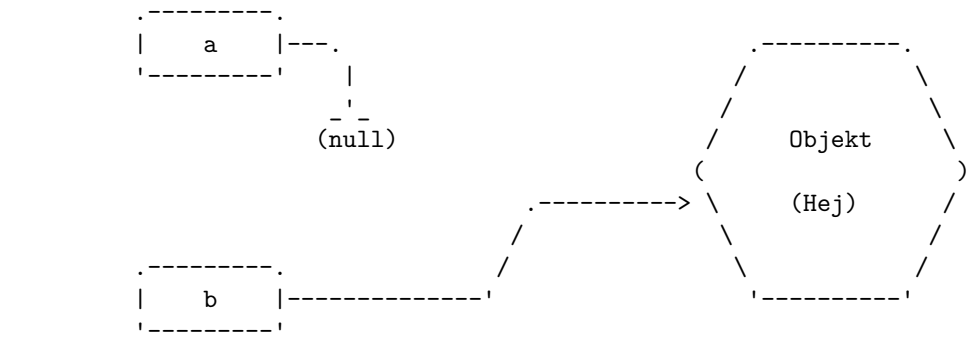


Exekvering av kod:

```
a = null;
```

Modell:

Referensvariabler



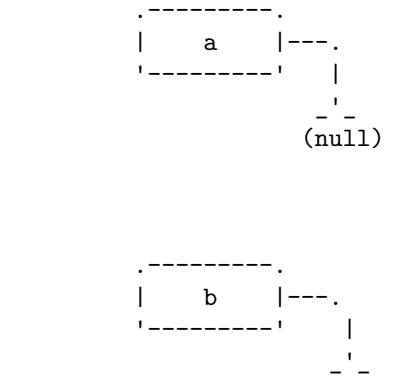

```
String a, b;
```

Exekvering av kod:

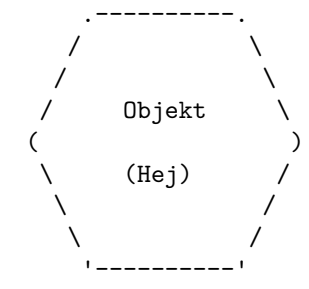
```
b = null;
```

Modell:

Referensvariabler



Detta objekt kommer att försvinna
eftersom ingenting refererar till det!



```
            (null)

String a, b;
```

Programsatser (statements)

- Semikolon används för att ange var en programsats slutar. Detta gäller tilldelningssatser, uttryckssatser och deklarationssatser.
- Semikolon används ej om sista delen i en sats är ett block.

Block {...}

- Används för att gruppera andra satser till en enhet.
- En sats kan i de flesta fall ersättas med ett helt block av satser.

```
Sats; ------> {
                  Sats1;
                  Sats2;
                  Sats3;
              }
```

- Används i deklaration/definition av klasser och metoder.
- Används när en styrande sats (villkorssats eller repetitionssats ska styra programflödet till en grupp av satser.
- Används när man vill ha ett eget *scope* (för variabler) (Exceptions och `switch`-satsen)

Villkorssatser

- Ett program styrs av olika villkor så att det kan utföra olika satser beroende av villkoret.
- För detta ändamål finns if-satsen, if-else-satsen och switch-satsen.
- `if`-satserna behöver ett logiskt uttryck för att 'välja en väg' av två möjliga.
- `switch`-satsen vill ha ett heltalsvärde för att välja en väg av flera möjliga.

if-satsen

Syntax för enkelsats:

```
if (logiskt uttryck)
    sats;
```

Syntax för block av satser:

```
if (logiskt uttryck) {
    sats1;
    sats2;
    sats3;
}
```

Semantik:

Satserna utförs endast om de logiska uttryckens värden är *true*.

if-else-satsen

Syntax för if-else:

```
if (logiskt uttryck)
    sats1;
else
    sats2;
```

Semantik:

Om det logiska uttryckets värde är *true* utförs sats1, annars utförs sats2.

Nästlad if-else-sats

Syntax för if-else:

```
if (uttryck 1)
    sats1;
else if (uttryck 2)
    sats2;
else
    sats3;
```

Semantik:

Om `uttryck1 == true` utförs sats1, annars om uttryck2 == true utförs sats2, annars utförs sats3.

Detta kan göras hur långt som helst. Men det tar tid att beräkna varje logiskt uttryck.

```
if (uttryck 1)          // tid!
    sats1;
else if (uttryck 2)     // tid!
    sats2;
else if (uttryck 3)     // tid!
    sats3;
else if (uttryck 4)     // tid!
    sats4;
else
    sats5;
```

switch-satsen

Om man har många möjliga fall och dessutom vill spara tid. Kan endast arbeta med heltalsuttryck, även tecken (‘a’, ‘b’, ‘c’, . . .) hör dit.

Syntax:

```
switch (heltalsuttryck)
{
    ...
}
```

Syntax, fortsatt:

```
switch (tal)
{
    case 1:
        System.out.println("Ett");
        break;
    case 2:
        System.out.println("Två");
        break;
    case(19):
        System.out.println("Nitton");
        break;
```

```
    default:
        System.out.println("Allt annat än ett, två och nitton");
        break;
}
```

Föreläsning 4 – Metoder

Metoder - funktioner/procedurer

Bryt ner programmet i mindre delar där varje del har en väl avgränsad och meningsfull uppgift. Sträva efter 'abstraktion' – att förenkla och hitta modeller så man slipper uppfinna hjulet på nytt varje gång.

- Man undviker upprepningar av kod.
- Programmet blir lätt att förstå och lätt att felsöka.
- En sådan del kan förläggas till en funktion eller procedur som ska ha namn som talar om vad de gör.
- Metoder måste deklareras/definieras i en klass

Syntax:

```
modifierare returtyp namn (parameterlista)
{
    satser ..
    eventuellt:     return;
    eventuellt:     return uttryck;
}
```

Metoder: procedurtyp - void

- Metoder som inte 'svarar' med ett värde
- Kan inte ingå i uttryck
- Deklareras med 'returtypen' `void`
- Proceduren avslutas när programflödet kommit förbi sista satsen eller av ett explicit `return;`

Procedurer - definition

Exempel:

```
public void printName (String n)
{
    System.out.println("Namnet är: " + n);
}
```

Procedurer - definition

Exempel (tidigt återhopp):

```
public void printName (String n)
{
    if (n == null)
        return;

    System.out.println("Namnet är: " + n);
}
```

Procedurer - anrop

Anrop av metoder sker enligt syntaxen:

```
metodnamn(värde/uttryckslista)       // Alltid par av parenteser.
```

- Antalet värden (0 eller flera) och deras typer (primitiva eller referenser) måste överensstämma med metodens deklaration/definition.

Exempel på anrop inom samma objekt:

```
printName("Fredrik");
```

Exempel på anrop från annat objekt:

```
obj.printName("Fredrik");
```

Utskrift:

```
Namnet är: Fredrik
```

Metoder: funktionstyp

- Lösningen på ett isolerat enkelt problem kan ofta formuleras i en enkel algoritm (funktion) som 'svarar' med lösningen.
- Ett stort problem kan lösas genom att hitta dess 'små' problem och lösa dem med de färdiga lösningarna (funktionerna).
- Sätt samman lösningarna.

> ("Divide and conquer") In computer science, divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.
>
> *Divide and conquer algorithms*
>
> `https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms`

Funktioner - definition

- Hitta och skriv ut det största av tre värden.
  - Förenkla: hitta det största av två och formulera algoritm:

```
/* Funktion som tar två heltal som parametrar och returnerar
   det största värdet. */

int max (int a, int b) {
    if (a > b)
        return a;    // 'svara' med värdet som a innehåller
    else
        return b;    // 'svara' med värdet som b innehåller
}
```

Funktioner - anrop

Testa:

```
int largest;
int p = 123;
int q = 456;

largest = max(p, q);
```

- Innan tilldelningen skickas `p`'s och `q`'s värden till funktionen `max`.
- De kopieras till parametrarna `a` och `b`.
- Funktionen beräknar största värdet och returnerar det (456).
- Tilldelning sker till variabeln `'largest'`.

Funktioner två anrop

Åter till problemet att finna det största av tre heltal. Här med två anrop:

```
int tmp, largest;
int p = 123;
int q = 456;
int r = 789;

tmp = max(p, q);                    // 456

largest = max(tmp, r);              // max(456, 789) => 789
```

Funktioner anrop i anrop

Men det går att göra ännu elegantare med anrop i anrop:

```
int largest;
int p = 123;
int q = 456;
int r = 789;

largest = max(p, max(q, r));        // 456
```

Klassvariabler och metoder

- Modifierare: `static`
- Kan användas direkt 'i klassen' utan objekt.
- Main är deklarerad `static` vilket innebär att man lätt kan skriva små testprogram utan att skapa objekt.
- Bör undvikas, men *main()* måste vara det – varför?
- Kolla in klassen *Math*.

> The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
> Math, Java Platform, Standard Edition 7 API Specification

Skillnad mellan klassmetod och instansmetod

- En klass-metod "finns" och kan användas utan att man skapat någon instans av klassen.
- Anges genom att den är `static`.

```
JOptionPane.showMessageDialog(null, "Hej Hopp!");
```

- Metoder som ej är `static` kan ej anropas förrän man skapat ett objekt av klassen.

```
Scanner sc = new Scanner(System.in);
String input = sc.nextLine();
```

- *main()* måste därför vara `static` – den anropas ju och exekveras innan man har kunnat skapat några objekt.

```
public static void main(String[]args)
{
    ...
}
```

Konstruktorer

Som ni har sett skapas objekt enligt syntaxen:

```
new Klass ()
```

Det ser ju ut som ett metodanrop?! Det som faktiskt sker är att en speciell 'metod' i klassen som kallas konstruktor anropas.

- Det finns alltid minst en konstruktor i en klass.

Konstruktorer

En konstruktor-definition har samma form som en vanlig metod. Den saknar dock returtyp och har alltid samma namn som klassen den definierats i:

```
class A
{
    public A()
    {
        System.out.println("Nu föds jag!");
```

```
    }
}

...

new A();    ------>    Utskrift: Nu föds jag!
```

Konstruktorer används för att initiera objektet; att sätta attributvärden, sätta referenser till andra objekt och eventuellt anropa metoder.

- Om man inte själv definierat en konstruktor så skapas en default konstruktor.
- Konstruktorer kan ta parametrar.

```
class Point
{
    private int x, y;

    public Point(int x, int y)
    {
        this.x = x;         // Obs 'this' för att komma till objekt-scope.
        this.y = y;
    }
}
```

Konstruktorer – överlagring

En klass kan ha flera konstruktorer om de har olika parameterlistor. Konstruktorer kan dessutom anropa varandra internt.

```
class Point
{
    private int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public Point(Point p)   // Överlagrad konstruktor.
    {
        this(p.x, p.y);     // Med this(.,.) anropas den första konstruktorn.
    }
}
```

- Använda olika konstruktorer:

```
Point p1 = new Point( 10, 20 );
Point p2 = new Point( p1 );          // Obs att en referens skickas.
```

- Men observera att:

```
Point p3 = new Point();              // Ger kompileringsfel!
```

Det finns ingen default konstruktor `Point() {...}` eftersom man har definierat andra. Om en sådan behövs måste man explicit skriva en sådan.

Överlagring

- Överlagring gäller även för vanliga metoder.

```
int addera( int a, int b)
int addera( int a, int b, int c )
String addera( String a, String b)

addera(10, 20);                          //--->    30
addera(10, 20, 30);                      //--->    60
addera("Stu", "dent");                   //--->    "Student"
```

Programmeringsstil

- Indentering
- Namngivning:
  - klasser
  - metoder
  - variabler
- Kommentarer
- Konsekvens och att vara följdriktig
- Uppdelning i klasser
- Uppdelning i metoder

Code Conventions

> This Code Conventions for the Java Programming Language document contains the standard conventions that we at Sun follow and recommend that others follow. It covers filenames, file organization, indentation, comments, declarations, statements, white space,

naming conventions, programming practices and includes a code example.

80% of the lifetime cost of a piece of software goes to maintenance. Hardly any software is maintained for its whole life by the original author. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly. The Code Conventions for the Java Programming Language document was revised and updated on April 20, 1999.

Oracle is providing the legacy documentation here. Please use it with the Advisory Disclaimer that it is NOT up to date. The information provided is for archival purposes. It is not being actively maintained and therefore links in the documentation may be broken.

Code Conventions for the Java Programming Language

Föreläsning 5 – Strängar, iterationer och arrayer i Java

Klassen String

- En klass som ingår i standard Java.
- Finns i paketet `java.lang` och ingår i själva språket Java.
- Objekten lagrar och *hanterar* text.

```
String s = new String("Detta är textsträng");
String t = "Detta är också en textsträng";
```

String - metoder

Metod	Funktionalitet
`int length()`	Ger längden
`char charAt(int pos)`	Plockar ut tecken
`boolean equals(Object o)`	Jämför innehåll lika
`int compareTo(Object o)`	Jämför innehåll < >
`String substring(int n)`	Plocka ut från index *n*
`String substring(int m, int n)`	Från index m till *n-1*

Metod	Funktionalitet
`String toUpperCase()`	Omvandlar till stora bokstäver
`String toLowerCase()`	Omvandlar till små bokstäver

- Den lagrade texten är oförstörbar, d.v.s., det går inte att ändra ett strängobjekt.
- Ny eller ändrad text läggs automatiskt i nya objekt.

```
String s1 = new String("abcedf");          // s1 ---> abcdef

String s2 = s1.substring(0, 3);            // s2 ---> abc

String s3 = s2.toUpperCase();              // s3 ---> ABC

System.out.println(s1);                    // abcdef
System.out.println(s2);                    // abc
System.out.println(s3);                    // ABC
```

Objektfabrik

Klassen String

- Eftersom `String` är inbyggt i språket så finns det finesser.

  T.ex. konkatenering med operatorn `+`:

```
String s4 = s2 + s3;                       // abcABC
                                           // s4  --->  abcABC
```

- Konstanta strängar i programkoden är objekt!

```
String s5 = "Gissa min längd";
S5.length();                               // 15
"Gissa min längd".length();                // 15
```

Jämförelser av strängar

Om man försöker jämföra strängar med operatorn `==` så kan man lätt blir lurad. Då är det referenserna som jämförs, d.v.s är det samma objekt?

- Det är alltså inte innehållet i objekten som jämförs.

```
String s1 = new String("ABCABC");
String s2 = s1.substring(0,3);
String s3 = s1.substring(3);

/* FELAKTIGT SÄTT ATT JÄMFÖRA STRÄNGAR! */
if (s1 == s2)
    System.out.println("Lika");
else
    System.out.println("Olika");
```

- För att jämföra innehåll i objekt används metoden `boolean equals(Object o)`. Den definieras i klassen `Object` och ärvs in i alla klasser. Den som vill anpassa `equals` till sin klass skriver en egen version. Det har man gjort i klassen `String`:

```
if ( s1.equals(s2) )
    System.out.println("Lika")
else
    System.out.println("Olika");
```

- För att jämföra alfabetisk ordning används `compareTo()`.

  Den returnerar ett heltal:

Villkor	Returvärde
`s1` kommer före `s2`	< 0
`s1` är lika med `s2`	$= 0$
`s1` kommer efter `s2`	> 0

Exempel:

```
String s1 = "Karin";
String s2 = "Kerstin";

int jmf = s1.compareTo(s2);

if ( jmf < 0 )
    System.out.println(s1 + " kommer alfabetiskt före " + s2)
else if ( jmf > 0 )
    System.out.println(s2 + " kommer alfabetiskt före " + s1);
else
    System.out.println(s1 + " lika med " + s2);
```

Teckenkoder och literaler

Typ	Antal bitar	Antal tecken	Beskrivning
ASCII	7 bitar	128 tecken	(inga å,ä,ö)
LATIN_1	8 bitar	256 tecken	(lite fler tecken)
Unicode	16 bitar	65536 tecken	(kinesiska, ..)

- Java använder *Unicode*.
  - Typen `char` har således 16 bitar.
  - Bakåtkompatibelt med *LATIN_1* och *ASCII*
- Teckenkonstanter skrivs med apostrofer: `'A', 'a', 'B'`
  - Kan även uttryckas numeriskt `'\111'` (oktalt format)
  - Eller hexadecimalt `'\u1234'` (alltid fyra siffror!)

Exempel:

Tecknen `A, B, C, 1, 2, 3` kan skrivas som konstanter på följande alternativa sätt:

- Som direkta tecken:

  ```
  'A', 'B', 'C', '1', '2', '3'
  ```

- I *oktalt* format:

  ```
  '\101', '\102', '\103', '\061', '\062', '\063'
  ```

- I *hexadecimalt* format:

  ```
  '\u0041', '\u0042', '\u0043', '\u0031', '\u0032', '\u0033'
  ```

Specialkoder

Escape-sekvenser:

Kod	Betydelse
`\n`	newline

Kod	Betydelse
`\b`	backspace
`\r`	return
`\f`	formfeed
`\t`	tab
`\'`	single quote (enkel apostrof)
`\"`	double quote (dubbel apostrof)
`\\`	backslash
`\nnn`	oktal form
	(nnn är oktala siffror $0 - 7$)
`\uxxxx`	hexadecimal form
	(xxxx är hexadecimala siffror $0 - F$)

Exempel:

```
String s = "Hej\njag kallas \"Pelle\"\nmen heter Per"
System.out.println(s);
```

```
 .---------------------.
|  Hej!                |
|  Jag kallas "Pelle"  |
|  men heter Per       |
'----------------------'
```

Repetitionssatser – Loopar - iterationer

- Ett program innehåller ofta satser som upprepas.
  - Exempel; ett program som räknar från 1 till 10:

```
System.out.println(  1 );
System.out.println(  2 );
System.out.println(  3 );
System.out.println(  4 );
System.out.println(  5 );
System.out.println(  6 );
System.out.println(  7 );
System.out.println(  8 );
System.out.println(  9 );
System.out.println( 10 );
```

Ineffektivt att upprepa kod

- Stora program.
- Mycket text att skriva eller skriva om.
- Multipla fel och svårt att åtgärda alla fel.
- Frågor:
  - Hur skulle programmet se ut om det ska räkna till 20?
  - Från 5 till 150?
  - Från 20 till 10?

Hopp i programmen

- Gammal idé: vi hoppar tillbaka i programmet och återanvänder samma kod om och om igen.
  - Vi definierar ett "hoppvillkor" som avgör om hoppet ska ske.
  - Programmet blir kortare och mer generellt.

I de "tidiga språken" som Assembly, Basic, Fortran, Cobol (och C) finns hoppsatser, t.ex. `Goto` eller `JMP`.

- Basic:

```
10:    print i
20:    i=i+1
30:    if i<10 then goto 10
```

- Assembly:

```
Label:  SUB R0, R0, 1
        JNZ R0, Label
```

Det kan ge program som är svåra att förstå och avlusa – man kan inte "se" vad programmet gör, utan man måste "simulera" en körning för att förstå.

```
10:    i=0
20:    i=i+1
30:    goto 50
40:    print i
50:    if i < 10 goto 20
60:    if i < 20 goto 40
```

Fritt användande av `goto` ger oöverblickbara och ohanterliga program (spagettiprogram/spagettikod).

- Motreaktioner?

Strukturerad programmering

- Artikel av Edsger Dijkstra 1968: *"Go to statement considered harmful"* blev början.
- Lösningen: använd endast *strukturerade* satser istället för hopp.
  - `if-then-else`
  - `for`
  - `while`
  - `do-while`
- Visade att det alltid går att omforma ett ostrukturerat program till ett strukturerat program.
- Numera allmänt accepterat och infört i de moderna språken.
  - `goto` finns således INTE i Java.

Repetitionssatsen while ( )

Huvudsakligen en *villkorsstyrd* repetitionssats. D.v.s. någonting ska utföras så länge som ett villkor är uppfyllt.

Syntax:

```
while (logiskt utryck)
    sats / block;
```

Semantik:

1. Beräkna värdet av det logiska uttrycket 2.
2. Om det logiska uttrycket är sant: utför upprepningssatsen/blocket.
3. Gå till punkt 1.

Exempel:

- Kör så länge som bilen har bränsle:

```
while (car.hasFuel())
    car.drive();
```

- Ge kortspelaren kort tills han är nöjd:

```
while (player.wantsCards()) {
    Card card = cardDeck.getCard();
    player.takeCard(card);
}
```

- Det är *mycket* vanligt att man vill upprepa någonting ett visst *antal* gånger.
  T.ex. vi vill skriva ut talen 1-9, d.v.s. upprepa 9 gånger:

```
int i = 1;                      // Deklaration och initiering av loopvariabel.

while (i < 10) {                // Loopvillkor
    System.out.println(i);      // Det vi vill upprepa.
    i = i + 1;                  // Uppräkning av loopvariabeln.
}
```

- Detta kan göras bekvämare och mer strukturerat med `for`-satsen.

Repetitionssatserna while ( ) och for ( )

```
int i = 1;                          // deklaration och initiering av loopvariabel

while (i < 10) {                    // loopvillkor
    System.out.println(i);          // det vi vill upprepa
    i = i + 1;                      // uppräkning av loopvariabeln
}


for (int i = 1; i < 10; i++)
    System.out.println(i);
```

Repetitionssatsen for ( )

Huvudsakligen en *antalsstyrd* repetitionssats. D.v.s. någonting ska utföras ett visst antal gånger.

Syntax:

```
for (initieringssats; logiskt villkorsutryck; ändringssats)
    upprepningssats / block;
```

Semantik:

1. Utför initieringssatsen (deklaration och initiering av loopvariabel).
2. Beräkna värdet av det logiska villkorsuttrycket.
3. Om det villkoret är sant: utför upprepningssatsen / blocket.
4. Utför ändringssats (normalt ökning av loopvariabeln).
5. Gå till punkt 2.

Exempel:

```
for (int i = 0; i < 10; i++)
    System.out.println(i);                    // Skriver ut 0 till 9

for (int i = 0; i < 10; i++)
    System.out.println(i + 1);                // Skriver ut 1 till 10

for (int i = 1; i < 10; i++) {
    System.out.println("Nästa siffra är: ");
    System.out.print(i);                      // Skriver ut 1 till 9
}
```

for ( ) – heltal inget krav

Exempel:

- Beräkna ett närmevärde till integralen av $f(x) = x^2$ i $[0, 1]$

  ```
  double sum = 0.0;
  double h   = 0.001;

  for (double x = 0.0; x <= 1.0; x += h)
      sum += x*x*h;
  ```

- 1000 varv!

for ( ) – semikolon enda kravet

Exempel 1:

```
for ( ; ; )                 /* Tolkas som true */
    System.out.println("Jag är en evighetsloop");
```

Exempel 2:

```
int i = 0;                  /* Fungerar, men det är bättre att använda while. */

for ( ; i < 10 ; ) {
    System.out.println(i);
    i++;
}
```

Exempel 3:

```
for (;;)
    ;                       /* Vad händer vid den "tomma satsen"? */
```

Repetitionssatsen do .. while ( )

Villkorsstyrd repetitionssats som alltid utför repetitionssatsen/blocket *minst en gång.* (“Vill du fortsätta?”)

Syntax:

```
do
    repetitionssats / block
while (logiskt villkorsuttryck);
```

Semantik:

1. Utför repetitionssatsen.
2. Beräkna logiska villkorsuttrycket.
3. Om sant gå till punkt 1.

Exempel:

```
// På lunchrestaurangen:

do {
    person.getFoodFrom(restaurant);
    person.eatFood();
} while (person.stillHungry());
```

Break

- Använd `Break` för att hoppa ur en loop.

```
/* Count letters until space. */

int count = 0;
String s = "hejsan svejsan";

for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == ' ')
        break;

    count++;
}

System.out.println("The number of letters until space are " + count);
```

Continue

- Använd `Continue` för att påbörja nästa "varv".

```
/* Count letters except space. */

int count = 0;
String s = "hej på dig";

for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == ' ')
        continue;

    count++;
}

System.out.println("The number of letters except spaces are " + count);
```

Nästlade loopar

Syntaxen för for ()-loopen:

```
for (sats1; logiskt utryck; sats2)
    upprepningssats;
```

En `for`-loop är en sats. Därför är följande tillåtet:

```
for (sats1a; logiskt utryck; sats2a)
    for (sats1b; logiskt utryck; sats2b)
        upprepningssats;
```

För varje repetition av den yttre utförs den inre i sin helhet.

- Viktigt med olika loopvariabler.

Nästlade loopar – multiplikationstabell

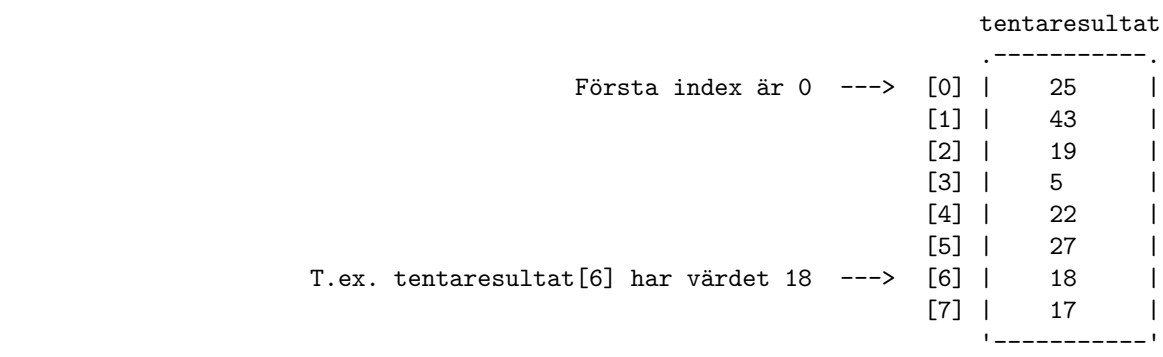
```
for (int x = 1; x < 10; x++) {
    for (int y = 1; y < 10; y++)
        System.out.print("\t" + x * y);

    System.out.println();                          // Byt rad
}
```

Nästlade loopar – testutskrift:

```
1    2    3    4    5    6    7    8    9

2    4    6    8    10   12   14   16   18

3    6    9    12   15   18   21   24   27

4    8    12   16   20   24   28   32   36

5    10   15   20   25   30   35   40   45

6    12   18   24   30   36   42   48   54

7    14   21   28   35   42   49   56   63

8    16   24   32   40   48   56   64   72

9    18   27   36   45   54   63   72   81
```

Arrayer (fält, vektorer)

- En indexerad variabel.
- Som en byrå med lådor där varje låda har ett nummer och byrån ett namn.
- Enskilda element väljs (indexeras) med `[ ]`



- I Java utgår index från 0.
- Par av hakklamrar `[ ]` används för deklaration och indexering.
- I Java kommer man åt arrayens längd med attributet `length`.
- I Java är arrayer alltid objekt som i sin tur kan innehålla:
  1. Primitiva värden
  2. Objektreferenser

Arrayer för primitiva värden

Det är alltså skillnad på ett *arrayobjekt* och en *arrayreferensvariabel*.

```
// En referensvariabel som kan referera till ett arrayobjekt för heltal:
int tentaresultat[];

// Ett arrayobjekt med 8 positioner för heltal:
new int[8];

// Komplett deklaration och initiering:
int tentaresultat [] = new int[8];
```

ARRAYEN: `[ ][ ][ ][ ][ ][ ][ ][ ]`

Att sätta in värden i arrayer

Fungerar ungefär som för vanliga variabler.

- Arrayer kan initieras *vid deklaration* och har då en speciell syntax:

```
// Det skapas ett arrayobjekt utifrån innehållet i blocket.
int tentaresultat [] = { 25,43,19,5,22,27,18,17 };
```

```
                                    ARRAYEN: [25][43][19][5][22][27][18][17]
```

- Eller *efter* deklaration:

```
// Arrayobjekt med 8 element:
int tentaresultat [] = new int[8];
```

```
                                    ARRAYEN: [  ][  ][  ][  ][  ][  ][  ][  ]
```

```
// Vanliga tilldelningar till arrayens element:
tentaresultat [0] = 25;
tentaresultat [1] = 43;
tentaresultat [2] = 19;
tentaresultat [3] = 5;
tentaresultat [4] = 22;
tentaresultat [5] = 27;
tentaresultat [6] = 18;
tentaresultat [7] = 17;
```

```
                                    ARRAYEN: [25][43][19][5][22][27][18][17]
```

- Det går även att använda block-formen senare:

```
int array [];
array = new int[] {25,43,19,5,22,27,18,17};
```

```
                                    ARRAYEN: [25][43][19][5][22][27][18][17]
```

Att komma åt värden i arrayer

Fungerar ungefär som för vanliga variabler fast med `[ ]`.

- Kom ihåg att testa för *null*!

Typiskt används loopar.

Exempel:

```
double medel = 0;

for (int index = 0; index < tentaresultat.length; index++)
    medel += tentaresultat [index];

medel /= tentaresultat.length;
```

```
                                        ----------------------------->
                                ARRAYEN: [25][43][19][5][22][27][18][17]
```

Kopiera arrayer (objekt)

- Metoden `clone` kopierar objekt (se klassen `Object`).

  OBS! *castning* till rätt typ måste göras.

  ```
  int a [] = {1,2,3};                          a  --->  [1][2][3]

  int b [] = (int[]) a.clone();                b  --->  [1][2][3]
  ```

- Nu har vi två array-objekt med samma innehåll!

  OBS! vad som händer om man gör följande:

  ```
  b = a;                                       a  ---.
                                                      \
                                                       v
                                                   [1][2][3]
                                                       ^
                                                      /
                                               b  ---'
  ```

  Det leder till att `b` och `a` refererar till samma objekt.

Arraycopy

`System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`

En färdig funktion för att kopiera en sekvens av värden från en array till en annan.

Exempel:

Vi vill kopiera 4 element från och med index 2 ur *arr1* till *arr2* med början i index 5.

```
int array1 [] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int array2 [] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

System.arraycopy(array1, 2, array2, 5, 4);
```

Innehållet i `array2`:

```
0,0,0,0,0,2,3,4,5,0
```

Att skicka arrayer som parametrar

- Metod som tar en array som parameter:

  ```
  void printArray(int [] a)
  {
      for (int i=0; i < a.length; i++)
          System.out.println(a[i]);
  }
  ```

- Anrop av metoden:

  ```
  int array [ ] = { 1, 7, 4, 2, 7, 9, 9, 0 };

  printArray(array);
  ```

- Utskrift:

  ```
  1
  7
  4
  2
  7
  9
  9
  0
  ```

Föreläsning 6 - Iterationer & arrayer

2015-07-13 måndag

Strukturer för data

- **Data**
  - information
  - värden
  - objekt
- Vanligt att data hanteras i grupper.

Vi har sett t.ex. primitiva data i arrayer (fält). Vi kommer även att titta på hur man lagrar objekt i arrayer och i listor och att det är vanligt att hantera dem repetitivt.

Detaljerad information från Oracle: Arrays, the Java Tutorials.

Inmatning av värden från tangentbordet

Exempelkod:

```
System.out.println("Ange max 10 tal. Q avslutar.");

/* Skapa en array med plats för 10 flyttal. */
double [] resultat = new double[10];

/* Skapa ett inläsningsobjekt med klassen 'Scanner'. */
Scanner s = new Scanner(System.in);

/* Skapa en räknare som pekar ut rätt index. */
int maxIndex = 0;       // Antal inlästa element.

/* Repetera så länge det finns indata och plats. */
while (s.hasNextDouble() && maxIndex < 10) {
    resultat[maxIndex++] = s.nextDouble();
}

/* Skriv ut alla inmatade värden */
for (int i = 0 ; i < maxIndex; i++) {
    System.out.println(resultat[i]);
}
```

Körexempel:

```
C:\>java InUtArray
Ange max 10 tal. Q avslutar.
5293761084

Q
5.0
2.0
9.0
3.0
7.0
6.0
1.0
0.0
8.0
4.0

C:\>
```

Enkel sortering

- Bubbelsortering – idé:
  - Antag array med N stycken element.
  - Iterera genom alla intilliggande par, d.v.s. element $(1, 2), (2, 3), (3, 4), ..., (N-1, N)$.
  - Jämför aktuellt elementpar och byt inbördes plats på de par som är fel ordnade. Upprepa tills inga utbyten längre behövs.

Fördjupning i sorteringsmetoder: Algorithms, the Java Tutorials.

Algoritm i pseudokod

- Array med N element given
- Variabel sortera sätts till **sant**
- Upprepa så länge som sortera är **sant**
  - Sortera sätts till **falskt**
  - Upprepa för element `e = 1` till `e = N-1`
    - Om `array[e]` > `array[e+1]`
      - Byt plats på element `e` och `e+1`
      - Sortera sätts till sant.

Algoritm i Java-kod

```
Att skicka en array som parameter till en metod! --.
                                                   |
public void sortera(double [] array)  <------------'
{
    boolean sortera = true;
    final int N     = array.length;

    while (sortera) {
        sortera = false;

        for (int e = 0; e < N - 1; e++) {
            if (array[e] > array[e + 1]) {
                double tmp   = array[e];        // Byt plats.
                array[e]     = array[e + 1];
                array[e + 1] = tmp;

                sortera = true;                 // Fortsätt sortera.
            }
        }
    }
}
```

Körning

Första iterationen

```
[5] [2]  9   3   7   6   1   0   8   4

 2   5  [9] [3]  7   6   1   0   8   4

 2   5   3  [9] [7]  6   1   0   8   4

 2   5   3   7  [9] [6]  1   0   8   4

 2   5   3   7   6  [9] [1]  0   8   4

 2   5   3   7   6   1  [9] [0]  8   4

 2   5   3   7   6   1   0  [9] [8]  4

 2   5   3   7   6   1   0   8  [9] [4]

 2   5   3   7   6   1   0   8   4   9
```

Andra iterationen

```
2  [5] [3]  7   6   1   0   8   4   9
2   3   5  [7] [6]  1   0   8   4   9
2   3   5   6  [7] [1]  0   8   4   9
2   3   5   6   1  [7] [0]  8   4   9
2   3   5   6   1   0   7  [8] [4]  9
2   3   5   6   1   0   7   4   8   9
```

Tredje iterationen

```
2   3   5  [6] [1]  0   7   4   8   9
2   3   5   1  [6] [0]  7   4   8   9
2   3   5   1   0   6  [7] [4]  8   9
2   3   5   1   0   6   4   7   8   9
```

Fjärde iterationen

```
2   3  [5] [1]  0   6   4   7   8   9
2   3   1  [5] [0]  6   4   7   8   9
2   3   1   0   5  [6] [4]  7   8   9
2   3   1   0   5   4   6   7   8   9
```

Femte iterationen

```
2  [3] [1]  0   5   4   6   7   8   9
2   1  [3] [0]  5   4   6   7   8   9
2   1   0   3  [5] [4]  6   7   8   9
2   1   0   3   4   5   6   7   8   9
```

Sjätte iterationen

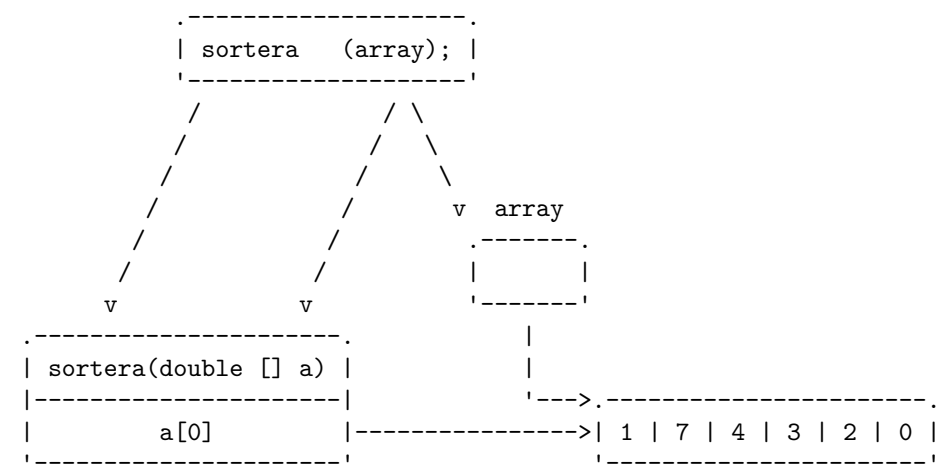
```
[2] [1]  0   3   4   5   6   7   8   9

 1  [2] [0]  3   4   5   6   7   8   9

 1   0   2   3   4   5   6   7   8   9
```

Sjunde iterationen

```
[1] [0]  2   3   4   5   6   7   8   9

 0   1   2   3   4   5   6   7   8   9
```

Insikter

- När en array skickas till en metod kan *orginalobjektet* komma att påverkas eftersom att *metoden hanterar samma objekt som "omvärlden"*.
- Det är **referensvariabelns värde** som kopieras – inte array-objektet.



Arrayer för referenser (till objekt)

- I Java hanteras objekt – hur hanteras många?
  - (T.ex. i arrayer av lämplig typ)
- Anta att vi har klassen `Person` med attributen *förnamn*, *efternamn* och *ålder*:

```
class Person
{
    private String enamn, fnamn;
    private int alder;

    public Person (String e, String f, int a) {
        enamn = e;
        fnamn = f;
        alder = a;
    }

    public String toString() {
        return enamn + " " + fnamn + ", " + alder;
    }

    public void print() {
        System.out.println(toString());
    }
}
```

Array med referenser

- En array som kan hantera 10 personer:

  ```
  Person [] grupp = new Person[10];
  ```

- Denna array innehåller nu 10 element, där varje element är en referens.
- Dessa har värdet *null*, det skapas alltså **INTE** 10 personer!

En bild av arrayen

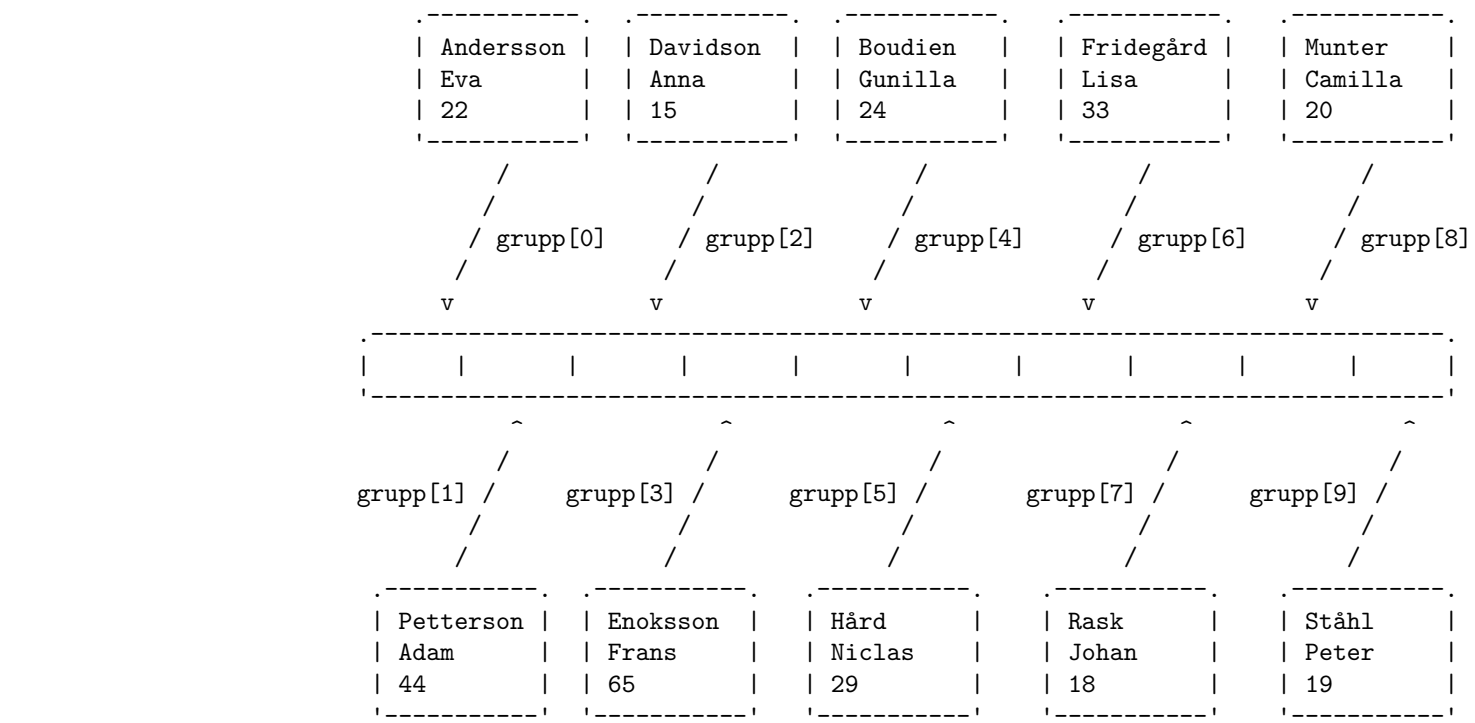
```
grupp   <--- (En referens till arrayen)
   \
    \
     V
.->  [null] [null] [null] [null] [null] [null] [null] [null] [null] [null]
|
|
'- Arrayen innehåller referenser av typen Person som är initierade till null
```

Lägg in objekt

```
grupp[0] = new Person("Andersson", "Eva",       22 );
grupp[1] = new Person("Petterson", "Adam",      44 );
```

```
grupp[2] = new Person("Davidson",  "Anna",      15 );
grupp[3] = new Person("Enoksson",  "Frans",     65 );
grupp[4] = new Person("Boudien",   "Gunilla",   24 );
grupp[5] = new Person("Hård",      "Niclas",    29 );
grupp[6] = new Person("Fridegård", "Lisa",      33 );
grupp[7] = new Person("Rask",      "Johan",     18 );
grupp[8] = new Person("Munter",    "Camilla",   20 );
grupp[9] = new Person("Ståhl",     "Peter",     19 );
```

En bild av arrayen



Alternativt – lägg till objekt vid initiering

Fördelen är bl.a. att man inte måste ange storlek.

Exempel:

```
Person [] grupp = {
    new Person("Andersson",  "Eva",      22),
```

```
    new Person("Petterson",  "Adam",     44),
    new Person("Davidson",   "Anna",     15),
    new Person("Enoksson",   "Frans",    65),
    new Person("Boudien",    "Gunilla",  24),
    new Person("Hård",       "Niclas",   29),
    new Person("Fridegård",  "Lisa",     33),
    new Person("Rask",       "Johan",    18),
    new Person("Munter",     "Camilla",  20),
    new Person("Ståhl",      "Peter",    19)
};
```

Personerna individuellt

- Plocka fram en person ur arrayen:

```
Person p = grupp[4];            // Gunilla
p.print();                      // Skriver ut info.
```

- Men eftersom referenserna finns i arrayen så kan man direkt skriva:

```
grupp[4].print();
```

Personerna som grupp

Nu kan personerna hanteras som grupp. T.ex. kan hela gruppen skickas någonstans för "behandling" (d.v.s. metodanrop).

Antag att vi vill skapa en metod som skriver ut alla persondata en godtycklig grupp av personer:

Att överföra arrayer till metoder

```
                           .----------------- OBS! syntax för array-parameter
                           |      .---------- Den lokala parameterns namn
                           |      |     .---- Arrayens längd
                           v      v     |
                                        v
public void skrivUt(Person [] personer)
{
    for (int i = 0; i < personer.length; i++) {
        System.out.println(personer[i].toString());
    }
}                                     ^        ^
                                      |        |
```

```
                        Indexering -'        |
OBS! toString() behöver inte anges ----------'
```

Att anropa metoden

```
// Antag att vi har vår gamla array med 10 personer:

Person [] grupp = new Person[10];
grupp[0] = new Person("Andersson", "Eva", 22);  // OSV...

skrivUt(grupp);                                 //Metodanropet:
```

Utskrift:

```
Andersson Eva, 22
Petterson Adam, 44
Davidson Anna, 15
Enoksson Frans, 65
Boudien Gunilla, 24
Hård Niclas, 29
Fridegård Lisa, 33
Rask Johan, 18
Munter Camilla, 20
Ståhl Peter, 19
```

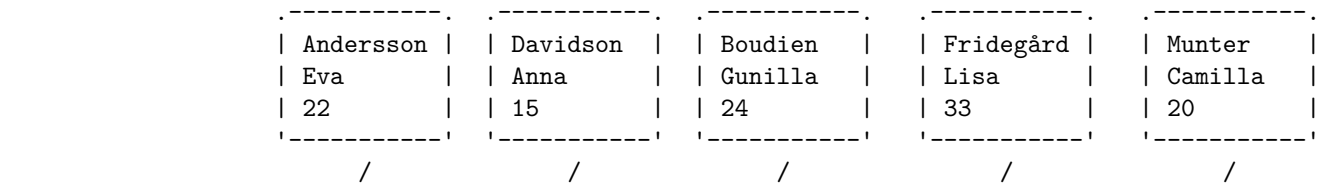
Varning för null!

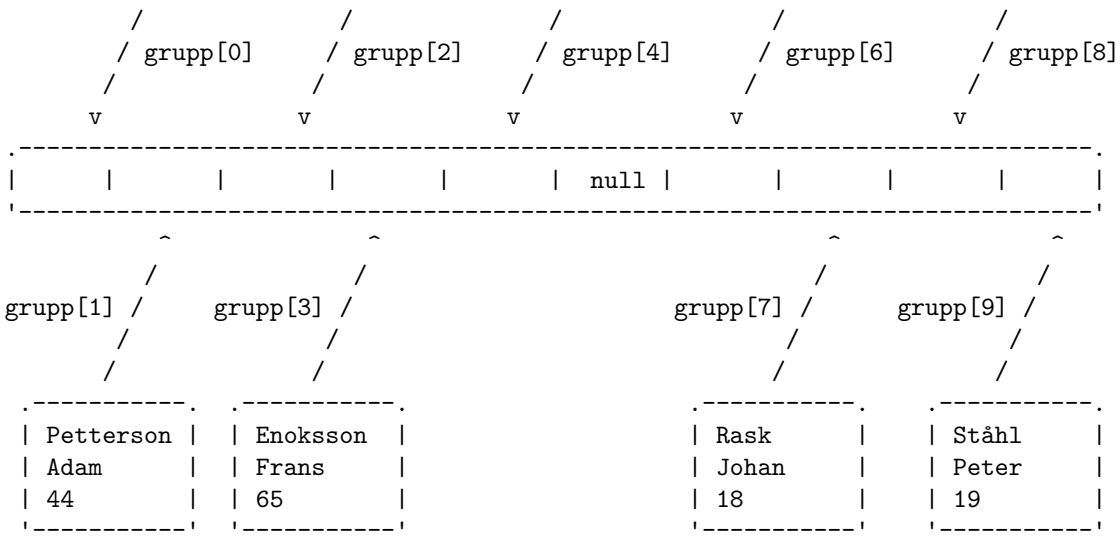
Eftersom vi här hanterar referenser så kan vi få problem.

- Antag att något element råkar vara *null*, t.ex. genom att arrayen inte är fylld eller att något element har satts till *null*:

  ```
  grupp[5] = null;
  ```

En bild av arrayen





Vad händer nu?

Det går alldeles utmärkt att skicka arrayen till metoden `skrivUt()`. Men inne i metoden anropas ju `toString()` på alla element (personer).

När loopen kommer till index 5 så blir det i princip: `null.toString()`.

Man kan ju inte anropa en metod i ett objekt som inte finns!

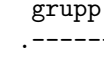
Programmet avbryts av en **null pointer exception**.

Eller ännu hemskare

- Hela array-objektet har tagits bort.
- Kom ihåg att arrayen i sig själv är ett objekt.

```
Person [] grupp = new Person[10];

...

grupp = null;
```

En bild av arrayen:



```
| null |
'------'
```

Analys

```
                                  .---------- Kan vara null
                                  |
                                  v
public void skrivUt(Person [] personer)
{
    for (int i = 0; i < personer.length; i++) {
        System.out.println(personer[i].toString());
    }
}                                  ^         ^
                                   |         |
Risk för null-pointer exception ---+---------'
i både '.length' och '.toString'
```

Åtgärd

```
                                  .---------- Kan vara null
                                  |
                                  v
public void skrivUt(Person [] personer)
{
    if (personer != null)                            <------ Kontrollera!
    for (int i=0; i < personer.length; i++) {
        if (personer[i] != null)                     <------ Kontrollera!
        System.out.println(personer[i].toString());
    }
}
```

Parametrarna till main

Programstart från kommandorad:

```
>java TestClass adam bertil 123 667
                |     |     |   |
      args[0] <--'     |     |   |
      args[1] <--------'     |   |
      args[2] <--------------'   |
      args[3] <------------------'
         |
```

```
         '------------------------.
                                  |
                                  v
```

```
public static void main(String [] args)
{
    for (int i = 0; i < args.length; i++)
        System.out.println(args[i]);
}
```

Mer om “command-line arguments”

Extra exempel på command-line arguments med “’\\0’-terminated strings”. Hämtat från egna anteckningar om C-programmering i inbyggda system.

- Programstart från kommandorad:

  ```
  $ ./progname one two three
  ```

- Argumentarrayens innehåll:

  ```
  argc        4
  argv[0]     progname
  argv[1]     one
  argv[2]     two
  argv[3]     three
  ```

- Datavisualisering, tecken för tecken:

  ```
    p  r  o  g  n  a  m  e  \0  o  n  e  \0  t  w  o  \0  t  h  r  e  e  \0
  |___.__.__.__.__.__.__.__.__|___.__.__.__|___.__.__.__|___.__.__.__.__.__|
  |   |  |  |  |  |  |  |  |##|   |  |  |##|   |  |  |##|   |  |  |  |  |##|
  |           argv[0]         |   argv[1]  |   argv[2]  |      argv[3]     |
  ```

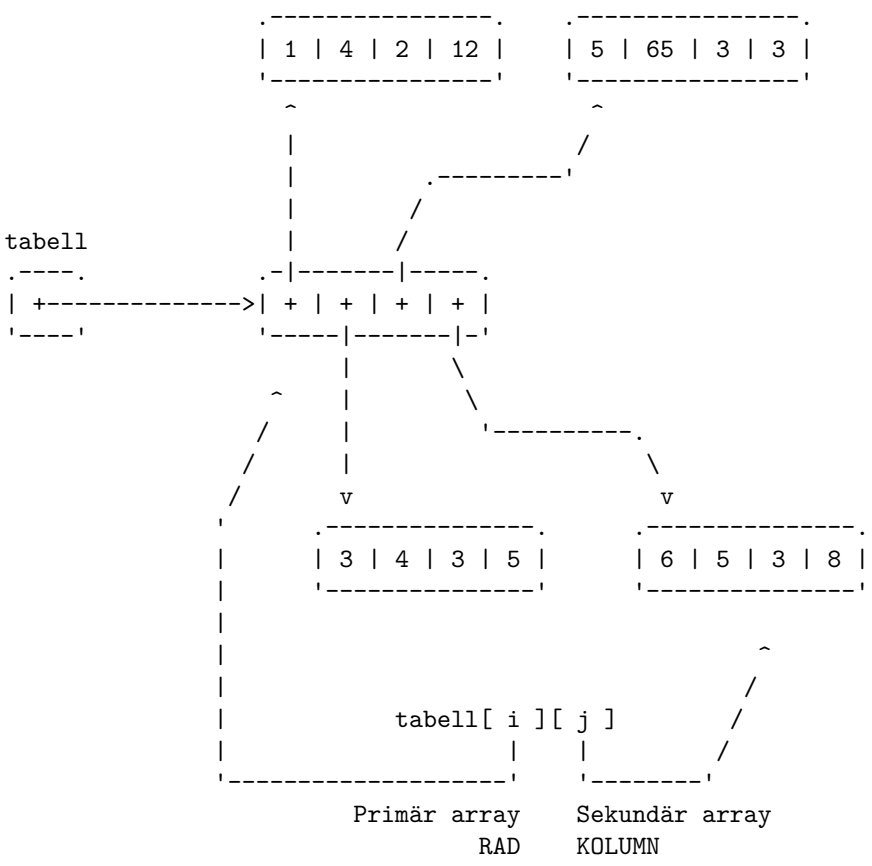
Flerdimensionella arrayer

- Eftersom arrayer är objekt, så måste de ju kunna ligga i arrayer, ...
- Vi kan skapa arrayer med *godtyckligt* många dimensioner:

  ```
  int [][] tabell = new int[4][4];
  ```

- Tabell är en referens till ett array-objekt som kan innehålla referenser till en-dimensionella arrayer.

Bild av flerdimensionell array:



Mental modell:

1	4	2	12
3	4	3	5
5	65	3	3
6	5	3	8

Exempelkod:

```
for (int i = 0; i < tabell.length; i++)
    for (int j = 0; j < tabell[i].length; j++)
        System.out.println(tabell[i][j]);
```

Utskrift:

```
1
4
2
12
3
4
3
5
5
65
3
3
6
5
3
8
```

Problem – arrayer har fix storlek

Programmet är begränsat till att hantera max t.ex. 10 resultat. Det är Svårt att "sätta in" objekt inne i arrayen och att "ta bort" element ur arrayen.

Vi vill att:

- Programmet skall kunna hantera godtyckligt många element.
- Förändringar i antal element kan ske dynamiskt så att programmet anpassar sig till det antal som är aktuellt.

Nästa föreläsning: Listor!

Föreläsning 7 - Listor i Java

2015-07-24 fredag

Problem – arrayer har fix storlek

- Vi vill att programmet skall kunna hantera godtyckligt många element.
- Att förändringar i antal element kan ske dynamiskt, så att programmet anpassar sig till det antal som är aktuellt.
- Svårt att "sätta in" objekt inne i arrayen och svårt att "ta bort" element.

Vi löser detta med listor!

Listor

- En lista är en samling data bestående av element.
- Listans längd kan vara godtyckligt lång.
- Implementeras ofta som en länkad lista, där varje element pekar ut nästa element.
- Element kan läggas till och tas bort var som helst i listan.

Listor i Java

I Java finns färdiga standardklasser som hanterar samlingar av data (objekt).

Det finns 3 färdiga klasser som implementerar listor (`java.util`):

- `Vector`
- `LinkedList`
- `ArrayList`

Utåt sett erbjuder de samma tjänster men internt löser de sin uppgift på olika sätt.

Några gemensamma metoder för `Vector`, `ArrayList` och `LinkedList`:

Metod	Funktionalitet
`add(obj)`	Lägger till sist
`remove(nr)`	Tar bort element nr
`remove(obj)`	Tar bort detta objekt
`get(nr)`	Hämtar referens nr
`clear()`	Tömmer listan

Metod	Funktionalitet
`size()`	Antal element
`isEmpty()`	Är listan tom?

Dessa metoder finns deklarerade i interfacet `List`.

List – ett interface

- **Interface** är en speciell typ som på ytan påminner om en klass
- Den talar om vilka metoder som ska finnas men den implementerar inte dessa själv.
- Det görs av `Vector`, `LinkedList` och `ArrayList`

Eller om ni vill; skriv en egen klass som implementerar `List`:

```
class MyList implements List
{
    ...
}
```

Att skapa listor

```
// Specifik hantering:
Vector v     = new Vector();
LinkedList l = new LinkedList();
ArrayList a  = new ArrayList();

//Generell hantering:
List list = new Vector();       // eller ..
List list = new LinkedList();   // eller ..
List list = new ArrayList();
          ^
          |
          '---  Fungerar eftersom ArrayList implements List
```

Att lägga in data i listor

```
Vector v = new Vector();

v.add("första strängen" );
v.add("andra strängen" );
v.add("tredje strängen" );
v.add("fjärde strängen" );
```

Objekttypen är godtycklig

```
Vector v = new Vector();

v.add(new Person("Älg"     , "Eva" , 25));
v.add(new Person("Persson" , "Ola" , 35));
v.add(new Person("Svan"    , "Mia" , 15));
```

Studie av add

- **public void add(Object o)**
- Att parametertypen är Object innebär att vilket objekt som helst kan läggas in i listan.
- Men det får inte vara primitiva typer (int, double, . . .)
- En intressant effekt är att en lista samtidigt kan innehålla olika objekttyper.
- Det kan ge problem när man ska hämta ut data ur listan. För hur kan man veta vad som finns i listan?

Studie av get & remove

- **public Object get(int index)**
- **Public Object remove(int index)**

När man ska hämta ut ett objekt ur en lista har man två val:

1. Använd en referensvariabel av typen Object. Då skjuter man upp problemet till ett senare tillfälle.

   ```
   Object o = list.get(5);
   // OK inget kompileringsfel
   ```

2. Typkonvertera (casta) referensen som kommer ut ur listan till den typ som referensvariabel har. Förutsätter att objektet verkligen har den typen.

```
Elefant e = (Elefant)list.get(3);
// om elementet "är" en elefant är det ok
```

Att skicka listor till metoder

```
// Använd generell hantering:
public void printList(List list)
{
    if (list != null) {                              // bra att kolla
        for (int i = 0; i < list.size(); i++)        // size() anger storlek
            System.out.println("Nr: " + i + list.get(i));
    }
}

// Anrop:
Vector v = new Vector();
printList(v);               // funkar eftersom Vector är en List
```

Bättre traversering i Java 1.5

```
// Ineffektivt att anropa size() och get() varje varv:
for (int i = 0; i < list.size(); i++)
    System.out.println(list.get(i));

// Java 1.5 -- helt ny for-loop för listor:
for (Object element : list)
    System.out.println(element);
```

Templates i Java 1.5

```
// Nytt i Java 1.5
// Denna lista hanterar endast personer
List<Person> plist = new Vector<Person>();

plist.add(new Person("Olsson" , "Berit" , 33);
plist.add(new Person("Agard"  , "Eva"   , 19);

// Då slipper man casta:
Person p = plist.get(1);
System.out.println(p.toString());
```

Templates i Java 1.5

```
List<Person> plist = new Vector<Person>();

plist.add(new Person("Olsson" , "Berit" , 33);
plist.add(new Person("Agard"  , "Eva"   , 19);

...

for (Person p : plist)      // alla personer
    p.print();              // i listan
```

Rekursion

- Ett alternativt sätt att utföra repetitioner.
- Sker genom att en funktion anropas sig själv.
- Viktigt att inse att flera instanser av en metod kan vara aktiva "samtidigt".
- Varje instans har en egen uppsättning av lokala variabler och parametrar.

Rekursiv räknare 1

```
// rekursiv funktion som räknar ner till 0

public void countDown(int n)
{
    System.out.println(n);

    if (n > 0)              // rekursiva fallet
        countDown(n-1);
}
```

Rekursiv räknare 2

```
// rekursiv funktion som räknar från 0 till n

public void countUp(int n)
{
    if (n > 0)              // rekursiva fallet
        countDown(n - 1);

    System.out.println(n);
}
```

Rekursion på sträng

```
// rekursivt sätt att vända en sträng - ineffektiv
// antar s != null

public String reverse(String s)
{
    if (s.length() <= 1)                // basfallet
        return s;

    String head = s.substring(0,1);     // första
    String tail = s.substring(1);       // resten
    return reverse(tail) + head;
}
```

Återanvändning av metoder

Dagens sista fråga:

Hur skriver man en metod som avgör om en sträng är ett palindrom? T.ex. "nitalarbralatin"

- `public boolean isPalindrome(String s)`

Använd det du redan har!

```
public boolean isPalindrome(String s)
{
    return s.equals(reverse(s));
}

public String reverse(String s)
{
    if (s.length() <= 1)                // basfallet
        return s;

    String head = s.substring(0,1);     // första
    String tail = s.substring(1);       // resten
    return reverse(tail) + head;
}
```

Föreläsning 8 – Wrapper-klasser, Autoboxing, listor i Java

Listor i Java hanterar endast objekt!

- Hur ska man då kunna hantera numeriska värden i listorna?
- Men kan man inte göra en klass som innehåller ett numeriskt värde?

```
class Number
{
    private int number;

    public Number(int n)
    {
        number = n;
    }

    public int getValue()
    {
        return number;
    }
}
```

Hur "stoppar man in"?

```
List<Number> lista = new Vector<Number>();    // Skapa listan.

for (int i = 1; i <= 10; i++) {               // Stoppa in 1..10 i listan.
    Number n = new Number(i);                 // Skapa ett objekt som innehåller talet.
    lista.add(n);                             // Lägg in i listan.
}
```

Hur "plockar man ut"?

```
// Samma lista som på föregående bild...

// plocka ut och skriv ut
for (int i = 0; i < lista.size(); i++) {
    Number n = lista.get(i);                // Hämta ut objektet ur listan.
    System.out.println(n.getValue());       // Skriv ut talet som finns inuti objektet.
}
```

Vi kan göra det bekvämt med toString()!

```
class Number
{
    private int number;

    public Number(int n)
    {
        number = n;
    }

    public int getValue()
    {
        return number;
    }

    public String toString()
    {
        return "" + number;          // Returnerar en strängrepresentation av talet.
    }
}
```

Enklare att skriva ut

```
// plocka ut och skriv ut
for (int i = 0; i < lista.size(); i++) {
    Number n = lista.get(i);         // Hämta ut objektet ur listan:
    System.out.println(n);           // Skriv ut talet som finns inuti objektet.
                                     // toString() anropas automatiskt av println.
}
```

... och ännu enklare...

```
// plocka ut och skriv ut
for (int i = 0; i < lista.size(); i++) {
    System.out.println(lista.get(i).getNumber());
}                                   ^
                                    |
                                    '--- Objektet hämtas ut och
                                           getNumber() anropas
```

... och ännu enklare...

```
// plocka ut och skriv ut
for (int i = 0; i < lista.size(); i++) {
    System.out.println(lista.get(i));
}                                ^
                                 |
                                 '--- Objektet hämtas ut
                                      och toString() anropas
```

... och ännu enklare...

- Använd den nya for-loopen för listor
- Då slipper vi indexera

```
// plocka ut och skriv ut
for (Number n : lista) {
    System.out.println(n);
}                     ^
                      |
                      '--- Objektet hämtas ut
                           och toString() anropas
```

MEN – måste vi skriva klassen Number?

- NEJ !
- Det finns redan färdiga klasser i Java
  - Integer
  - Double
  - Boolean
  - ...
- Dessa kallas *Wrapper-klasser*

Wrapper-klasser

```
 Integer            Double              Boolean
.------.          .--------.          .---------.
| int  |          | double |          | boolean |
'------'          '--------'          '---------'
```

Omslagsklasser är representanter för primitiva data och innehåller klassmetoder (`static`-deklarerade) som har med dessa att göra.

- Har även instansmetoder
- `byte`, `short`, `int`, `long`, `float`, `char`
- `Byte`, `Short`, `Intereger`, `Long`, `Float`, `Character`
- Även en objektrepresentation (tex för listor)

Wrapperklasser

- Klassmetoden `parseInt(String s)` i `Integer`
- Klassmetoden `parseDouble(String s)` i `Double`
- Det finns beskrivande attribut, t.ex:
  - `MIN_VALUE` och `MAX_VALUE`
  - `NEGATIVE_INFINITY` & `POSITIVE_INFINITY`
  - `SIZE` ger antalet bitar

Wrapper - Character

```
Character
.------.
| char |
'------'
```

- isDigit, isLetter,
- isLetterOrDigit, isLowerCase,
- isUpperCase, isWhitespace,
- toLowerCase, toUpperCase,
- getNumericValue

Vad wrappas in?

- En primitiv variabel av motsvarande typ.
- Referenser kan inte referera till primitiva typer.
- Men referenser kan referera till instanser av wrapperklasser.

```
Integer i = new Integer(5);
Integer j = new Integer("156");
int     a = i.intValue();           // hämta ut primitivt värde
String  s = i.toString();
```

- Detta innebär att man kan använda dem t.ex. i listor.

Auto -boxing och -unboxing

Från Java version 5.0 finns automatik för konvertering mellan primitiva värden och Wrapper-objekt.

Exempel:

```
Integer i;
int     j;

i = 5;
// automatiskt: i = new Integer(5);

j = i;
// automatiskt: j = i.intValue();

i = i + j;
// automatiskt: i = new Integer( i.intValue() + j );
```

Kom ihåg att ovanstående kod kan vara ineffektiv.

Auto -boxing och –unboxing för listor!

Detta gör det enkelt att hantera primitiva värden i listor.

```
import java.util.*;

List<Integer> lista = new Vector<Integer>();

for (int i = 0; i < 100; i++)
    lista.add(i);

for (int i : lista)
    System.out.println( i );

int ivar = l.get(7);
System.out.println(ivar);
```

Var läggs elementen in?

`add` – normalt sist i listan. Men man kan ange önskad plats med: `add(plats, objekt)`

Var hämtas objekten

- get(plats)
- remove(plats)

Finns ett visst objekt?

- boolean contains(objekt)

Ex. en kö (sist in – sist ut)

```
class Queue
{
    private List<Integer> list = new Vector<Integer>();

    public void in(Integer i)
    {
        list.add(i);                // sist
    }

    public Integer out()
    {
        return list.remove(0);      // först
    }

    public boolean isEmpty()
    {
        return list.isEmpty();
    }
}
```

Ex. en stack (sist in – först ut)

```
class Stack
{
    private List<Integer> list = new Vector<Integer>();
```

```
    public void push(Integer i)
    {
        list.add(0,i);              // först
    }

    public Integer pop()
    {
        return list.remove(0);      // först
    }

    public boolean isEmpty()
    {
        return list.isEmpty();
    }
}
```

Använda Queue & Stack

```
Stack s = new Stack();
Queue q = new Queue();

for (int i = 0; i < 5; i++) {
    q.in(i);
    s.push(i);
}

while (!q.isEmpty())
    System.out.println(q.out());

while (!s.isEmpty())
    System.out.println(s.pop());
```

```
.---.
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 4 |
| 3 |
| 2 |
| 1 |
| 0 |
'---'
```

Exempel

Vi kommer att göra ett program för att spela kort och listor kommer att användas.

Klasser:

- Card
- Deck
- Hand
- Player
- Dealer

Denna kod kommer att läggas ut...