

# EDA016 Programmeringsteknik för D

## Läsvecka 4: Aritmetik, Logik & Datastrukturer

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2015

## 4 Aritmetik, Logik & Datastrukturer

- Att göra denna vecka
- Aritmetik
- Summering
- Logik
- Datastrukturer
- Specifikationer och implementationer
- Oföränderlighet (immutability)
- Hjälpmedel på tenta: "snabbreferens"

# Att göra i Vecka 4: Förstå aritmetiska och logiska uttryck, använda klasser mha klass-specifikationer

- 1 Läs följande kapitel i kursboken:  
3.1–3.9, 5, 6.1–6.4, 7.2, 7.5  
Begrepp: heltalsdivision med rest, implicit och explicit typkonvertering, aritmetiska och logiska uttryck, De Morgans lagar, oföränderlighet
- 2 Gör övning 4: Aritmetik, Logik
- 3 Träffas i samarbetsgrupper och hjälp varandra förstå
- 4 Gör Lab 3: använda färdigskrivna klasser, kvadrat

# Primitiva datatyper i Java

Typ	Betydelse
<b>byte, short, int, long</b>	heltal
<b>float, double</b>	reellt tal (tal med decimaldel)
<b>boolean</b>	logiskt värde (sant eller falskt)
<b>char</b>	tecken, till exempel bokstav, siffra, specialtecken

String liknar vanliga klasser men har **speciellt stöd i språket**.  
Mer om detta nästa vecka.

# Numeriska typernas storlek samt min- och max-värden

Typ	Bitar	Min	Max
<b>byte</b>	8	-128	+127
<b>short</b>	16	-32 768	+32 767
<b>char</b>	16	0	65 535
<b>int</b>	32	-2 147 483 648	+2 147 483 647
<b>long</b>	64	$\approx -9 \cdot 10^{18}$	$\approx +9 \cdot 10^{18}$
<b>float</b>	32	$\approx -3.4 \cdot 10^{38}$	$\approx +3.4 \cdot 10^{38}$
<b>double</b>	64	$\approx -1.8 \cdot 10^{308}$	$\approx + - 1.8 \cdot 10^{308}$

För detaljer se [Javas språkspecifikation](#) och [IEEE-standarden 754](#)

# Konstanter för min- och max-värden

Några exempel:

```
short smin = Short.MIN_VALUE;    // smin = -32768  
int imax = Integer.MAX_VALUE;    // imax = 2147483647  
double dmin = Double.MIN_VALUE; // dmin = 4.9E-324  
                                   // OBS! pyttelitet men positivt  
double dmax = Double.MAX_VALUE; // dmax = 1.8E308
```

Se vidare subklasser till **Number**-klassen.

# Math

Exempel på statiska bra-att-ha-metoder i klassen **Math**

```
long round(double x);           // avrundning, även float till int
int abs(int x);                 // |x|, även double ...
double hypot(double x, double y); //  $\sqrt{x^2 + y^2}$ 
double sin(double x);           //  $\sin x$  även: cos, tan, asin, atan etc.
double exp(double x);           //  $e^x$ 
double pow(double x, double y);  //  $x^y$ 
double log(double x);           //  $\ln x$ 
double sqrt(double x);          //  $\sqrt{x}$ 
double toRadians(double deg);   //  $deg \cdot \pi/180$ 
```

# Precisionsproblem

Vad skriver nedan program ut?

```
1 package week04;
2
3 public class PrecisionProblems {
4     public static void main(String[] args) {
5         double x = 0.9999999999999999E6;
6         double d = 1E-12;
7         System.out.println("x == " + x);
8         System.out.println("d == " + d);
9
10        // dangerous to check floating point number equality with ==
11        boolean isSame = x + d == x;
12        System.out.println("(x + d == x) == " + isSame);
13
14        double a = 0.99999999998;
15        double b = 0.99999999999;
16
17        if (Math.abs(a - b) < 1e-4) { // a better way to check double equality
18            System.out.println("a and b are approximately the same");
19            System.out.println("Math.abs(a - b) == " + (Math.abs(a - b)));
20        }
21    }
22 }
```

Se mer detaljer [här](#).



# Deklarationer av variabler

```
typ namn = startvärde;
```

- Om startvärdet utelämnas blir lokala variabler odefinierade, attribut får implicit startvärde (0, 0.0, ...).
- Om det i en klass finns flera förekomster av samma namn så gäller den "närmaste" deklARATIONEN:

```
public class A {  
    private int x; // attribut  
  
    public void p() {  
        int x = 0; // lokal variabel i metoden p  
        // alla förekomster av "x" avser här den lokala  
        // variabeln x. Om man vill komma åt attributet  
        // x skriver man this.x  
    }  
}
```

# Implicita startvärden för attribut

Om en variabel är ett attribut (field) så ges implicita startvärden när objektet konstrueras. OBS! Detta gäller **inte** för lokala variabler som måste initialiseras före första användningen.

DataType	Default Value (for fields)
<b>byte</b>	0
<b>short</b>	0
<b>int</b>	0
<b>long</b>	0L
<b>float</b>	0.0f
<b>double</b>	0.0d
<b>char</b>	'\u0000'
<b>boolean</b>	false
String or any object	<b>null</b>

# Implicit och explicit konvertering mellan numeriska värden vid tilldelning

## Tilldelningssats:

```
variabel = uttryck;
```

- Variabel och uttryck ska ha **samma eller kompatibel** typ.
- Om variabeln är "större" än uttryckets värde konverteras det nya värdet automatiskt till variabelns typ.
- Om variabeln är "mindre" måste man konvertera explicit:

```
int i = 100;  
double d = 314.61;  
short s = (short) i; // värdet av i konverteras  
                      // till short  
i = (int) d;          // värdet av d konverteras  
                      // till int, 314.61 -> 314
```

# Några aritmetriska uttryck

## Heltalsuttryck:

```
1  int a = 0;  
2  int b = 12;  
3  int c = 20;  
4  a = 2 * (b + c) + 4; // a = 68  
5  b = a / 10;          // b = 6 (6.8, decimalerna stryks)  
6  c = a % 10;          // c = 8 (68/10 = 6 + 8/10, 8 är resten)
```

## Reella uttryck:

```
1  double x = 1.4;  
2  double y = 1 + 2 * (x + 1); // y = 5.8  
3  double z = x * x + y * y;   // z = 35.60  
4  z = z / 10;                // z = 3.56  
5  int a = 5;  
6  x = 1 + (double) a / 2;    // x = 3.5
```

Explicit typkonvertering har **högre** prioritet än de aritmetiska operatorerna!

# Var n-te gång, jämt delbart med n

Vanligt trick: `if (i % n == 0) { ... }`

```
1 package week04;
2
3 public class EveryNth {
4
5     public static void main(String[] args) {
6         int max = 7*22;
7         int n = 7;
8         for (int i = 0; i <= max; i++) { // görs (max + 1) gånger
9             if (i % n == 0) {
10                 System.out.println(i + " % " + n + " == 0 JÄMT DELBART MED " + n);
11             } else {
12                 System.out.println(i + " % " + n + " == " + (i % n));
13             }
14         }
15     }
16 }
17 }
```

# Förkortade tilldelningssatser

Tilldelning med samma variabel både till höger & vänster kan förkortas:

```
x += dx;      // <=> x = x + dx
sum += term;  // <=> sum = sum + term
nbr /= 10;    // <=> nbr = nbr / 10
```

Kortformer för att öka/minska med 1 (inkrementering/dekrementering):

```
x++;          // <=> x = x + 1  <=> x += 1
x--;          // <=> x = x - 1  <=> x -= 1
```

Postfix och prefixnotation med ++ och -- (använd med försiktighet)

```
int a = 10;
int x = a++; // a blir 11, x blir 10
int b = 10;
int y = ++b; // b blir 11, y blir 11
y = --b;     // b blir 10, y blir 10
y = b--;     // b blir 9, y blir 10
```

# Algoritmexempel: summering 1

Pseudo-kod för algoritmen "Summera värden":

```
sum = 0;  
för alla termer {  
    term = "nästa term";  
    sum = sum + term;  
}
```

Exempel 1: läs n; för alla n tal: läs och summera; skriv ut.

```
Scanner scan = new Scanner(System.in);  
int n = scan.nextInt(); // läs antalet tal  
int sum = 0;  
for (int i = 0; i < n; i++) {  
    int term = scan.nextInt(); // läs nästa tal  
    sum = sum + term;  
}  
System.out.println(sum);
```

# Algoritmexempel: summering 2

Exempel 2: läs tills negativt tal påträffas, summera.

```
Scanner scan = new Scanner(System.in);  
int sum = 0;  
int nbr = scan.nextInt();  
while (nbr >= 0) {  
    sum = sum + nbr;  
    nbr = scan.nextInt();  
}
```

Exempel 3: Beräkna summan  $\sum_{i=1}^{100} \frac{1}{i * i}$

```
double sum = 0;  
for (int i = 1; i <= 100; i++) {  
    sum = sum + 1.0 / (i * i);  
}
```



# Logiska uttryck

Logiska uttryck kan kopplas samman med operatorerna && ("och"), || ("eller"), ! ("icke").

```
int a = 3;  
int b = 10;  
if (a > 1 && b > 1) ...    // true  
if (a < 0 || a > 10) ...   // false  
if (! (a > 5)) ...         // true (a <= 5)
```

Sanningstabell för logiska operatorer:

p	q	p && q	p    q	! p
true	true	true	true	false
true	false	false	true	
false	true	false	true	true
false	false	false	false	

# Negering av logiska uttryck med De Morgans lagar

$p$  och  $q$  är logiska uttryck,  $\neg$  står för "icke",  $\wedge$  för "och",  $\vee$  för "eller":

$$\neg(p \wedge q) \iff (\neg p) \vee (\neg q)$$

$$\neg(p \vee q) \iff (\neg p) \wedge (\neg q)$$

I vanligt språk:

- Om uttrycket består av deluttryck sammanbundna med `&&` eller `||`, ändra alla `&&` till `||` och omvänt.
- Negera alla ingående deluttryck. En relation negeras genom att man byter `==` mot `!=`, `<` mot `>=`, etc.

Exempel:

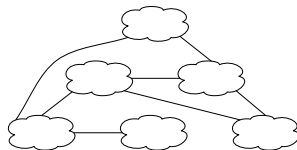
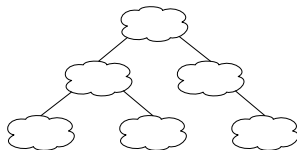
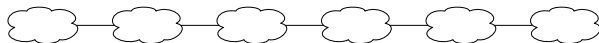
<code>! (a &lt; b    (a == 1 &amp;&amp; b == 1))</code>	$\iff$
<code>! (a &lt; b) &amp;&amp; ! (a == 1 &amp;&amp; b == 1)</code>	$\iff$
<code>! (a &lt; b) &amp;&amp; (! (a == 1)    ! (b == 1))</code>	$\iff$
<code>a &gt;= b &amp;&amp; (a != 1    b != 1)</code>	

# Vad är en datastruktur?

En datastruktur:

- kan innehålla många element,
- har *ett* namn,
- och man kan komma åt de enskilda elementen.

Har man många element av samma typ kan man organisera elementen på olika sätt, till exempel som listor, träd eller grafer:



Mer om detta i fortsättningskursen...

# Klass som datastruktur

- En enkel form av datastruktur är att kapsla in olika element som hänger i hop i en **post** (eng. *record*).  
Kallas även **tupel** (eng. *tuple*).  
Exempel: en post med persondata innehåller fälten *förnamn*, *efternamn*, *personnummer*
- I många programspråk finns speciella språkkonstruktioner för detta (record i **Pascal**, struct i **C**, Tuple i **Scala**).
- I objektorienterade språk kan man använda attribut i klasser för att skapa en sådan datastruktur. En klass kan erbjuda både lagring av data i en (intern) struktur och operationer för att läsa och bearbeta data.  
**Klasser samlar data och operationer på data.**

# Exempel på klass som datastruktur: Person

```
1 package week04;
2
3 public class Person {
4     private String givenNames; //separated by blank if more than one
5     private String familyName;
6     private String socialSecurityNumber; //12 digits e.g. 199901013538
7
8     public Person(String givenNames, String familyName, String socialSecurityNumber) {
9         this.givenNames = givenNames;
10        this.familyName = familyName;
11        this.socialSecurityNumber = socialSecurityNumber;
12    }
13    public String getGivenNames() {
14        return givenNames;
15    }
16    public void setGivenNames(String givenNames) {
17        this.givenNames = givenNames;
18    }
19    public String getFamilyName() {
20        return familyName;
21    }
22    public void setFamilyName(String familyName) {
23        this.familyName = familyName;
24    }
25    public String getSocialSecurityNumber() {
26        return socialSecurityNumber;
27    }
28    public void setSocialSecurityNumber(String socialSecurityNumber) {
29        this.socialSecurityNumber = socialSecurityNumber;
30    }
31 }
```

För den nyfikne: läs om [boilerplate code](#) och [denna diskussion](#)

# Specifikation av klassen BankAccount

## BankAccount

```
/** Skapar ett bankkonto med numret acctNbr och  
    saldot noll */  
BankAccount(int acctNbr);  
  
/** Tar reda på kontonumret */  
int getAcctNbr();  
  
/** Tar reda på saldot */  
int getBalance();  
  
/** Sätter in amount kronor på kontot */  
void deposit(int amount);  
  
/** Tar ut amount kronor från kontot */  
void withdraw(int amount);
```

# Implementation av klassen BankAccount, 1

```
public class BankAccount {  
    private int acctNbr; // kontonummer  
    private int balance; // saldo  
  
    /** Skapar ett bankkonto med numret acctNbr och saldot noll */  
    public BankAccount(int acctNbr) {  
        this.acctNbr = acctNbr;  
        this.balance = 0;  
    }  
  
    /** Tar reda på kontonumret */  
    public int getAcctNbr() {  
        return acctNbr;  
    }  
}
```

# Implementation av klassen BankAccount, 2

```
/** Tar reda på saldot */  
public int getBalance() {  
    return balance;  
}  
  
/** Sätter in amount kronor på kontot */  
public void deposit(int amount) {  
    balance = balance + amount;  
}  
  
/** Tar ut amount kronor från kontot */  
public void withdraw(int amount) {  
    balance = balance - amount;  
}  
}
```



# Bankkonto med ränta

**Krav** på bankkonto med ränta:

- 1 Ränta för varje dag som pengar finns på kontot skall ackumuleras.
- 2 Vid årsskiftet skall räntan sättas in på kontot.

**Design** av bankkonto-klass med ränta:

- Attributet `interestRate` anger räntesatsen i procent för kontot.
- Den ackumulerade räntan (attributet `interest`) ökas genom anrop av metoden `computeInterest()` varje gång som saldot förändras genom insättning eller uttag.
- Attributet `lastInterestDay` anger numret på den dag (med början på nummer 1 för den 1 januari) då räntan senast beräknades.
- Metoden `Date.today()` ger numret på aktuell dag.
- I slutet av ett år läggs den ackumulerade räntan till saldot när någon "utifrån" utför operationen `newYearActions()`.

Se hela lösningen **här**. (Designen är inte särskilt realistisk... Varför?)

# Implementation BankAccountWithInterest, 1

## Abstraktion:

Vi löser delproblemet att räkna ränta i en egen metod

```
public class BankAccountWithInterest {  
    private int acctNbr;           // kontonummer  
    private int balance;          // saldo  
    private double interestRate;  // räntesats i procent  
    private double interest;      // ackumulerad ränta under året  
    private int lastInterestDay;  // dagnummer för senaste ränteberäkning  
  
    //... konstruktor, getters/setters (visas ej)  
  
    public void deposit(int amount) {  
        computeInterest();  
        balance = balance + amount;  
    }  
}
```

# Implementation BankAccountWithInterest, 2

## Abstraktion:

Vi återanvänder lösningen på delproblemet att räkna ränta

```
/** Adderar årets ränta till saldot. Ska utföras vid årsskifte */  
public void newYearActions() {  
    computeInterest();  
    balance = balance + (int) Math.round(interest);  
    interest = 0;  
    lastInterestDay = 1;  
}  
  
/** Adderar räntan sedan föregående insättning eller uttag */  
private void computeInterest() {  
    interest = interest + interestRate / 100.0 *  
        (Date.today() - lastInterestDay) /  
        360 * balance;  
    lastInterestDay = Date.today();  
}  
}
```

# Specifikation av klassen Square

## Square

```
/** Skapar en kvadrat med övre vänstra hörnet i x,y och med sidlängden side */  
Square(int x, int y, int side);  
  
/** Ritar kvadraten i fönstret w */  
void draw(SimpleWindow w);  
  
/** Flyttar kvadraten avståndet dx i x-led, dy i y-led */  
void move(int dx, int dy);  
  
/** Tar reda på x-koordinaten för kvadratens läge */  
int getX();  
  
/** Tar reda på y-koordinaten för kvadratens läge */  
int getY();  
  
/** Tar reda på kvadratens area */  
int getArea();
```

Den interna representationen av kvadratens läge är *inte* stipulerad av specifikationen – det är upp till implementatören.

# Specifikation av SimpleWindow

Så här ser delar av specifikationen för SimpleWindow ut i ankboken Appendix C, sidan 309-312.

## SimpleWindow

```
/** Creates a window and makes it visible. */  
SimpleWindow(int width, int height, java.lang.String title);  
  
/** Moves the pen to a new position. */  
void moveTo(int x, int y)  
  
/** Moves the pen to a new position while drawing a line. */  
void lineTo(int x, int y)
```

Specifikationerna i ankboken liknar (en enklare variant av) javadoc som genereras ur dokumentationskommentarer. Jämför med [javadoc för SimpleWindow](#)

# Implementation av (delar av) Square-klassen

```
1 package week04;
2 import se.lth.cs.pt.window.SimpleWindow;
3
4 public class Square {
5     private int x;        // x- och y-koordinat för
6     private int y;        // övre vänstra hörnet
7     private int side;     // sidlängd
8
9     public Square(int x, int y, int side) {
10         this.x = x;
11         this.y = y;
12         this.side = side;
13     }
14
15     public void draw(SimpleWindow w) {
16         w.moveTo(x, y);
17         w.lineTo(x, y + side);
18         w.lineTo(x + side, y + side);
19         w.lineTo(x + side, y);
20         w.lineTo(x, y);
21     }
22
23     public void move(int dx, int dy) {
24         x = x + dx;
25         y = y + dy;
26     }
27 } // Övning: implementera getX(), getY() och getArea()
```

# Varför privata attribut?

Två viktiga anledningar till att göra attribut **private** och bara tillåta åtkomst av data via metoder:

- 1 I metoder kan vi **validera indata** och säkerställa så att manipuleringen blir rätt, till exempel att vi inte tillåter negativ ränta (eller ska vi tillåta det? – ta reda på kraven...)
- 2 Vi kan **ändra den interna representationen** av data och alltså byta datastruktur utan att behöva ändra alla metoder i koden som använder vår klass.

Nackdelen i Java är boilerplate (getters, setters)... Men en bra IDE kan hjälpa till att generera mallar för dessa med ett enkelt menykommando.

Vissa språk har explicit stöd för **properties** med getter/setter-par (t.ex. JavaScript, C#, Scala) och vissa uppfyller **uniform access principle** (t.ex. Ruby, Python, Scala) där man kan använda samma syntax oavsett om det är en metod eller ett attribut.

# Ändra den interna representationen av kvadrats läge

Läget hos en kvadrat kan representeras av ett Point-objekt i stället för av koordinaterna x och y:

```
private Point location; // övre vänstra hörnet
private int side;       // sidlängd
```

## Point

```
/** Skapar en punkt med koordinaterna x, y */
Point(int x, int y);

/** Tar reda på x-koordinaten */
int getX();

/** Tar reda på y-koordinaten */
int getY();

/** Flyttar punkten avståndet dx i x-led,
    dy i y-led */
void move(int dx, int dy);
```



# Implementera ny Square, 1

```
public class Square {
    Point location;
    int side;

    /** Skapar en kvadrat med övre vänstra hörnet i x,y
     *  och med sidlängden side
     */
    public Square(int x, int y, int side) {
        this.location = new Point(x, y);
        this.side = side;
    }

    public void draw(SimpleWindow w) {
        w.moveTo(location.getX(), location.getY());
        w.lineTo(location.getX(), location.getY() + side);
        w.lineTo(location.getX() + side, location.getY() + side);
        w.lineTo(location.getX() + side, location.getY());
        w.lineTo(location.getX(), location.getY());
    }
}
```

# Implementering av ny Square, 2

draw-variant med lokala variabler x och y:

```
public void draw(SimpleWindow w) {  
    int x = location.getX();  
    int y = location.getY();  
    w.moveTo(x, y);  
    w.lineTo(x, y + side);  
    w.lineTo(x + side, y + side);  
    w.lineTo(x + side, y);  
    w.lineTo(x, y);  
}
```

# Implementering av ny Square, 3

Inte svåra:

```
public int getX() {  
    return location.getX();  
}  
  
public int getY() {  
    return location.getY();  
}
```

I move måste man utnyttja move-operationen i Point för att flytta kvadraten:

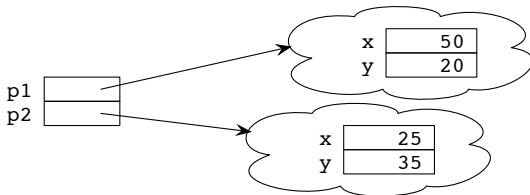
```
public void move(int dx, int dy) {  
    location.move(dx, dy);  
}
```

# Objekt som parameter

## Point

```
/** Beräknar avståndet mellan denna punkt och punkten p */  
double getDistanceTo(Point p);
```

```
Point p1 = new Point(50, 20);  
Point p2 = new Point(25, 35);  
double dist = p1.getDistanceTo(p2);  
System.out.println("Avståndet mellan punkterna är " + dist);
```



# Två olika implementationer av getDistanceTo

## 1. Använd bara metoderna i Point:

```
public double getDistanceTo(Point p) {  
    int xDist = getX() - p.getX();  
    int yDist = getY() - p.getY();  
    return Math.sqrt(xDist * xDist + yDist * yDist);  
}
```

## 2. Utnyttja kunskapen om att Point har attributen x och y:

```
public double getDistanceTo(Point p) {  
    int xDist = x - p.x;  
    int yDist = y - p.y;  
    return Math.sqrt(xDist * xDist + yDist * yDist);  
}
```

Fördelar/nackdelar med 1 vs 2?

Övning: använd Math.hypot istället

# Delade objekt

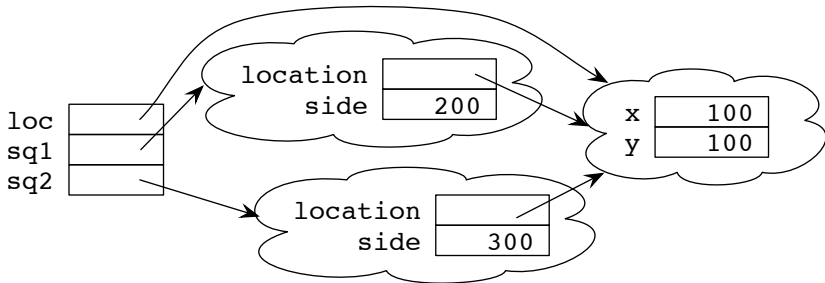
Point-objektet som definierar kvadratens läge kan skapas utanför objektet och skickas med som en parameter till en ny konstruktor:

```
public Square(Point location, int side) {  
    this.location = location;  
    this.side = side;  
}
```

Nu kan flera kvadrater dela samma läge:

```
Point loc = new Point(100, 100);  
Square sq1 = new Square(loc, 200);  
Square sq2 = new Square(loc, 300);
```

## Delade objekt — var försiktig!



Nu kan man flytta båda kvadraterna med `loc.move(25,30)`.  
Men observera att `sq1.move(25,30)` också medför att **båda** kvadraterna flyttas!

# Objekt som funktionsresultat

Metoder för att ta reda på kvadratens läge:

```
public int getX() {  
    return location.getX();  
}  
  
public int getY() {  
    return location.getY();  
}
```

Man kan skriva en metod som ger Point-objektet som resultat, men då ger man användaren **tillgång till den interna representationen**, så det brukar man inte göra om man vill kunna ändra denna i framtiden:

```
public Point getLocation() {  
    return location;  
}
```



# Oföränderlighet (immutability)

# Om man kan undvika förändring blir det mycket lättare

- Varje förändring av variablers värden (ändring av tillstånd) är svåra att resonera om och ger lätt upphov till buggar.
- Vi tränar mycket på att **mutera** värden i denna kurs, men i praktiken är det ofta bra att **undvika** mutering.
- Om man ska utnyttja flera trådar och parallell exekvering kräver föränderliga variabler **synkronisering** så att olika trådar inte förstör för varandra och svårhittade buggar uppkommer. Med oföränderliga värden blir jämlöpande exekvering (eng. *concurrency*) enklare.  
Mer om detta i **realtidsprogrammering**.

# Förhindra att variabler **ändras** med **final**

Attributet `latinsktNamn` nedan är en **konstant**.

Kompilatorn hjälper oss att kolla så att vi inte råkar ändra på det vi har deklarerat som **final**.

```
1  class Gurka {
2      public int vikt = 100; //gram
3
4      public final String latinsktNamn = "Cucumis sativus";    // *1
5
6      public String visa() {
7          System.out.println("Denna gurka (" + latinsktNamn + ") väger " + vikt + "g");
8      }
9  }
10
11  public class Constant {
12      public static void main(String[] args){
13          Gurka g = new Gurka();
14          g.vikt = 200;
15          g.latinsktNamn = "Tomat";    // ERROR: ger kompileringsfel! Vilket?
16          g.visa();
17      }
18  }
19
20  // *1: final deklarereras gärna även static om det bara behövs en enda
```

# Lokala variabler, konstanter

- Deklarera variabler så sent som möjligt, omedelbart innan de används första gången.
- Bara de storheter som behövs för att beskriva tillståndet hos objekt ska vara attribut.
- Deklaration av konstanter görs **final static** med namn i versaler:

```
public class CardGame {  
    private final static int MAX_PLAYERS = 10;  
    ...  
}
```

# Oföränderligt objekt

```
1  class Gurka { // exempel på oföränderligt objekt (eng. immutable objekt)
2      private int vikt;
3
4      void Gurka(int vikt) {
5          this.vikt = vikt; //endast här tilldelas attributet ett värde
6      }
7
8      public Gurka halva(){ // förändrar inte denna instans, skapa ny istället
9          return new Gurka(vikt/2);
10     }
11
12     public void visa() {
13         System.out.println("Denna gurkan instans väger för alltid " + vikt + " gram");
14     }
15 }
16
17 public class ImmutableObject {
18     public static void main(String[] args){
19         Gurka g1 = new Gurka(42);
20         Gurka g2 = g1; // g1 och g2 refererar till samma objekt
21         g1 = g1.halva(); // förändringen av g1 påverkar inte g2
22         g1.visa();
23         g2.visa();
24     }
25 }
```

# Hjälpmedel på tenta: "snabbpreferens"

# Enda hjälpmedlet på kontrollskrivning och tenta

Ta med denna snabbreferens **på papper** vid examination:

<http://cs.lth.se/eda016/javaref>

**Träna på att hitta i den och att använda den!**

*Innehåll:* satser, uttryck, deklARATIONER, klasser, standardklasser: Object, Math, System, Integer, String, StringBuilder, List, ArrayList, LinkedList, Random, Scanner), läsa/skriva till/från fil

# Träna på kontrollskrivning och tenta

- Det är svårt även för en van programmerare att ta tentan om man inte tränar!
- Jag vill att **alla** tränar mycket på att programmera utan hjälpmedel och att **alla** siktar på **högsta betyg**!
- Boka in **2h** i läsvecka 5 (nästa vecka) med din samarbetsgrupp för att **träna inför kontrollskrivning**.
- Följ instruktionerna på kurshemsidan:  
**Examination → Träna inför diagnostisk kontrollskrivning**