Välkomna till kursen

OOP (5p)

Objektorienterad Programmering

Kursledare: Kjell Lindqvist

Rum 1531 Lindstedtsvägen 3 plan 5, Tfn: 790 62 76

Mail: kjellq@nada.kth.se

Mottagningstid: Vid behov.

Assistent: Alexander Baltatzis alba@nada.kth.se
Per Sedholm sedholm@nada.kth.se

Kursupplägg

Egen aktivitet är en förutsättning för att klara kursen!

Uppgifter att göra till varje gång, ger akademiska poäng!

Muntlig tentamen i slutet av kursen

Kurslitteratur:

Winder, Roberts: Developing Java software

Kompendier om OO, XP, …

Bredvidläsning:

Kent Beck Extreme Programming Explained

Jeffries, Anderson, Hendrickson Extreme Programming Installed

Booch, Rumbaugh, Jacobson The Unified Modelling Language user guide

Java in a nutshell (koncentrerad referensbok)

Kursbunt (föreläsningsanteckningar, kompendier, …)

WWW-baserat: Begrepp i OOP, …

Bakgrund

Viktiga begrepp från grundkurser som vi kommer att repetera och fördjupa.

–Typer Vad är en typ, hur görs typning, varför görs typning, när görs typning

–Objekt Tillstånd, komplexitet

–Polymorfi Varför behovs polymorfi, vad är polymorfi

–Scope Vad är tillgängligt var och varför

–Parametrar och parameteröverföringsmekanismer

– Syntax, semantik, syntaxdiagram, BNF, EBNF

–Rekursion När behövs den, när och hur skall vi eliminera den.

–Listor Homogena, inhomogena, kontinuerliga, diskontinuerliga

–Algoritmer Konstruktion och analys

–Modularisering

Kursinnehåll

Kursen kommer att handla om allt detta samt naturligtvis:

Paradigm i datalogin

Vi har i tidigare kurser använt oss av två olika sätt att programmera:

1. Funktionell programmering eller OOP?.
2. Imperativ programmering. Baseras på kommandon som uppdaterar minnet.

Vi skall nu låta programmen baseras på klasser.

Tidigare var problemet att algoritmiskt bryta ned dåtidens beräkningsintensiva program så att den algoritmiska komplexiteten kunde behärskas.

Idag är problemet snarast att programmen översvämmas av data. Nedbrytningen bör ta hänsyn till detta.

Objektorienterat synsätt:

Dela upp så att allt (data och algoritm) som hör till en viss typ av objekt utgör en enhet (klass).

Kursinnehåll, forts

Enskilda enheter (objekt) är instanser av klassen.

Kalle Andersson är en instans av person.

Objekten kan sända och ta emot meddelanden.

Ett program utgörs (statiskt) av ett antal klasser och en algoritmdel (huvudprogrammet)

Dynamiskt: ett antal objekt utbyter meddelanden.

T ex ett program som adderar två heltal:

Någon ber objekt 1 att addera sig med objekt 2.

Objekt 1 frågar objekt 2 om dess typ och får svaret heltal.

Objekt 1 frågar objekt 2 om dess värde och får svaret 2

Objekt 1 skapar ett nytt heltal med värdet lika med sitt eget värde plus 2.

Det nya objektet utgör svaret på det initiala meddelandet.

Kursinnehåll, forts

Fördelar med OO

Uppdelning efter objekt alla data som hör ihop samlas i objektet.

Den del av algoritmen som hör ihop med objektklassen finns i denna.

Generella objektklasser kan skapas och återanvändas.

Listor, träd, fönsterhantering, …

Objektklasser kan specialiseras till den aktuella applikationen

Lista av heltal

Lista av bilar

OOP

Bibliotek av fördefinierade klasser.

Modellen stämmer bättre med verkligheten.

Naturligare modellering.

Dela upp efter objekt.

Undersök vilka meddelanden objekten skall förstå och vilken respons som de kall ge.

Enkelt att modifiera och bygga ut.

Vi kan skapa generella återanvändbara klasser som är robusta och korrekta.

Klasserna skall ha minimala, ortogonala protokoll

Korrekthet: rätt värde om indata i domänen.

Robusthet : om indata utanför domänen upptäcks detta.

Vi skall alltså eftersträva moduler som har ett väldefinierat, litet gränssnitt (protokoll) och som inte använder globala variabler.

Testning

För att åstadkomma detta måste vi testa programmen.

Enhetstest:

Skrivs av programmerarna innan modulerna programmeras

Varje enhet testas avseende varje aspekt.

Alla tester måste ge korrekt resultat till 100%

Testerna skall vara automatiska

Funktionella test:

Användarna skriver dessa test

Testen avser (del)system

Allt som måste fungera skall motsvaras av ett test.

Gör en första version med minimal funktionalitet (ta bara hänsyn till det absolut viktigaste).

Fortsätt sedan att bygga ut programmet.

eXtreme Programminng (XP)

12 grundpelare

- **Planeringsspel Planera snabbt, prioritera, teknikkrav**
- **Små releaser Släpp ofta nya versioner**
- **Metafor**
- **Enkel design Inga listiga lösningar**
- **Testa Skriv testerna först, testa efter varje förändring**
- **Omstrukturera Refactoring= Ta bort onödig kod, förenkla**
- **Parprogrammering Alltid två som samarbetar**
- **Kollektivt ägande av kod Alla äger och kan ändra i koden**
- **Kontinuerlig integration Integrera och bygg systemet flera ggr per dag**
- **40-timmarsvecka Efter det inför vi buggar**
- **Inkludera en kund i teamet**
- **Följ kodstandard Förenklar kommunikation**

Risk

Grundproblemet: Risk

- **Tidsramar brister.**
- **Projekt avbryts Mängden fel gör att man måste avbryta.**
- **System surnar. För dyrt att underhålla.**
- **Många defekter Produktionsprogramvara används inte pga. för många fel.**

Problemet

- **Fel problem löses.**
- **Problemdomänen förändras.**
- **Onödiga "features".**
- **Utvecklare byts ut.**

Risk, forts

Risk	Lösning
• Tidsramar brister	Korta cykler, täta releaser.
• Projekt avbryts	Minsta funktionalitet med mesta värde, fokusera på det viktigaste.
• System surnar	Test efter varje förändring.
• Många defekter	Både utvecklare och kunder skriver tester.

Problemet

Problemet	Lösning
• Problemdomänen förändras	Korta releascykler.
• Onödiga "features"	Fokusera på det viktigaste.
• Utvecklare byts ut	Utvecklarna behandlas som intelligenta varelser.

Arbetssätt

Par programmerar tillsammans.

Utvecklingen drivs av tester.

Refactoring.

Java

Objektorienterat programspråk

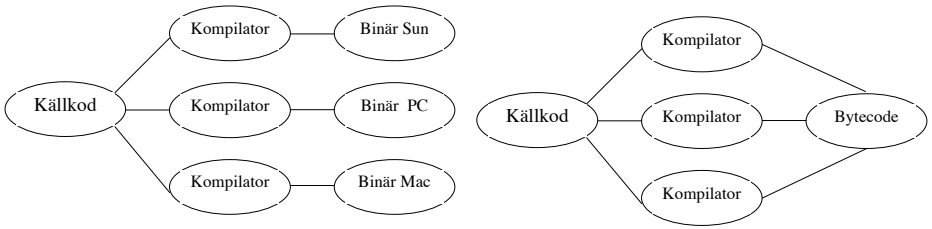
Virtuell maskinarkitektur

Plattformsoberoende och säkert

Klassbibliotek

Utvecklingsverktyg

Traditionella programspråk **Java**



Java, grunderna

Möjligheter till egendefinierade typer.

Fullt objektorienterat. Parallellitet och distribution.

Fördefinierade klasser:

- **Boolean**
- **Character**
- **Byte**
- **Short**
- **Integer**
- **Long**
- **Float**
- **Double**

De fördefinierade klasserna har motsvarigheter i enkla datatyper boolean, character, …

Operationer +, -, … kan endast utföras på enkla datatyper.

Standardbibliotek

Biblioteken är organiserade i "paket"

- **java.lang Basklasser: Objekt, String, System, Math**
- **java.applet klasser från Applets**
- **java.awt Plattformsoberoende fönsterhantering**
- **java.io In- och utmatning**
- **java.net nätverkskoppling**
- **java.util Hashtabeller, vektorer, …**

java.lang.String .substring()

paketnamn.klassnamn.metodnamn

Program

Ett javaprogram är en klass.

```
class namn {            /* Obs! Filen måste heta namn.java*/
public static void main (String[] args) {
deklarationer och satser;
  }
}
```

Ex

```
class HelloWorld{                /*Definitionen finns i filen HelloWorld.java*/
public static void main (String args[]) {
   System.out.println("Hello world"); /Obs! skillnad mellan versaler och
gemener!*/
  }                              /*Kompilera med javac HelloWorld.java*/
}
```

Interpretera med `java HelloWorld`

Metoder är
[åtkomstmodifierare] (typ|void) namn (argumentLista){
deklarationer och satser;
<om typen inte är void så skall return finnas med>
}

Program, forts

argumentlistan består av par med typ och namn

Ex public Float length(Integer x, Integer y) {…}

Metoder, konstanter och variabler kan definieras en per klass eller en per objekt.

static betyder att vi får en per klass och de som inte definieras som static blir automatiskt en per objekt.

En variabel som inte är statisk kallas instansvariabel och en statisk (en per klass) för klassvariabel.

En statisk metod kan inte manipulera instansvariabler eller anropa ickestatiska metoder i klassen.

Metoder kan vara:
Konstruktor metoden konstruerar ett nytt objekt
Destruktor destruerar objekt (i java finns Garbage collector)
Selektor Läser av någon egenskap i objektet
Mutator Muterar någon egenskap i objektet
Predikat Beroende på objektets tillstånd blir resultatet true eller false

In- och utmatning

Relativt enkelt men mycket att hålla reda på!

Interaktion med grafiska arbetsstationer *är* komplicerat men Java erbjuder relativt enkel hantering.

För att slippa hålla reda på allt som behövs för att sköta grafiska gränssnitt kan vi använda klassen System.out med procedurer för utmatning
print(arg) och println(arg) där arg kan vara numerisk typ, char, string, boolean och inmatning görs med hjälp av Scanner som finns i java.util

```
import java.util.*;
public class ex1 {
   public static void main (String[] args)
   {
      Scanner stdin = new Scanner (System.in);

      while (stdin.hasNext()) System.out.println(stdin.next());
   }
}
```


Klasser

Pascals RECORD motsvaras av CLASS men CLASS är mer generell.

```
class person {
    Integer pnr;
    String name;
}
```

Det skall finnas en konstruktor:

```
class person {
public person() {
}
Integer pnr;
String name;
}
```

Vi kan skriva en egen klass för att hantera tal:

```
class number {
    public number() {    /* konstruktor */
    }
    public int value;
}
```


Referenser

```
number n, m;          /* referenser till objekt av typen number */
n = new number();     /* m sätts att referera till ett nytt */
m = new number();     /* dynamiskt objekt av typen number */
n.value =m.value;     /* attributen kommer vi åt med punktnotation */
n = m;                /* Tilldelning av referens */
io.printLn(n.value);
```

Nu är det inte meningen att vi skall komma åt attributen med punktnotation utan vi skall ha metoder som sätter och läser av attributen.

```
class number {
    public number() {
    }
    private int value;

    public int value() {
        return value;
    }
    public void setValue(int newValue) {
        value = newValue;
    }
    public void print() {
        System.out.println(value);
    }
}
```

Referenser, forts

```
klassnamn ref1, ref 2;
```

Deklarerar av två referensvariabler.

Referensvariabelns innehåll är en referens till ett objekt.

Klassnamnet anger vilken klasstillhörighet objektet skall ha.

Referensvariabeln initieras till null.

```
new klassnamn();
```

Ger en referens till ett nyskapat objekt ur klassen "klassnamn".

Referensen överförs med tilldelningsoperatorn "=".

Kontroll av likhet och olikhet i referens:

```
ref1 == ref2; /* ref1 och ref2 refererar till samma obj? */

ref1 != ref2; /* ref1 och ref2 refererar till olika obj? */
```


Testning

Innan vi skriver klassen skall testen skrivas! Testprogrammet ser ut på följande sätt:

```
import junit.framework.*;
public class numberTest extends TestCase {
    private number n1, n2;
    public numberTest(String tc) {
        super (tc);
    }
    protected void setUp() {
        n1 = new number(5);
        n2 = new number(6);
    }
    public void testNumber() {
        assertEquals(5, n1.value());
    }
}
```

Och vår klass kommer då att se ut på följande sätt!

```
class number {
    private int value;
    public number(int initValue) {
        value = initValue;
    public int value() {
        return value;
    }
    public void print() {
        System.out.println(value);}
}
```


Huvudprogram

Och ett program som använder klassen:

```
public class huvudprog {
    number n;
    public huvudprog() {
    }
    public static void main(String args[]){
        huvudprog m = new huvudprog();
        m.n = new number(6);
        m.n.print();
    }
}
```


Styrstrukturer

Iteration och selektion

```
for (initiering ; test för fortsättning ; uppdatering av variabler) sats;

while (test för fortsättning) sats;

do sats while (test för fortsättning);

if (villkor) sats;

if (villkor) sats;
else sats;

if (villkor) {
if (villkor) sats; }
else sats;
```

Flerval:

```
switch (i) {
    case 1,2:
        sats;
        break;
case 3, 4, 5:
    sats;
    break;
default:
    sats;
}
```

Styrstrukturer, forts

Om inte break finns med testas alla alternativ och eventuellt utförs flera av satserna (om flera alternativ stämmer).

Använd if istället för switch!

Vid if direkt efter if sätt ut satsparentes!

Även om det går att skriva kompakt kod à la C eller C++ så kommer vi inte att acceptera detta i kursen!

Ex:

```
for (i = 1, tmpFac = 1 ; i <= value; tmpFac = tmpFac * i, i++) {}
```

kan skrivas

```
for (i = 1, tmpFac = 1 ; i <= value; tmpFac *= i++) {}
```

men …

Enkla datatyper

Typ	Antal byte	Område
byte	1	[-128, 127]
short	2	[-32768, 32767]
int	4	[-2147483648, 2147483647]
long	8	[-9223372036854775808, 9223372036854775807]
char	2	[0, 65535]
float	4	$\pm[10^{-38}, 10^{38}]$ ca 7 signifikanta siffror
double	8	$\pm[10^{-308}, 10^{308}]$ ca 15 signifikanta siffror
boolean		[false, true]

Operatorer:
+, -, *, /, %, =, ==, !=, <, >, <=, >=

Vad händer då vi kommer utanför talområdet?
– Vi får värdet Infinity (oändligheten) eller 0 som svar (i motsats till heltal där det blir fel).
• Division med 0.0 ger värdet Infinity. (I motsats till heltal där det blir programavbrott).
• Värdet till uttrycket 0.0/0.0 är NaN (Not-a-Number).
• Det meningslöst att fortsätta en beräkning med NaN.

Typkonvertering

Då ett värde av annan typ eller med större område skall användas tillsammans med ett annat värde måste man explicit konvertera det. T.ex. (int)1.9

Polymorfi

En operator som kan operera på fler typer kallas polymorf till skillnad mot monomorfa operatorer som endast kan operera på en typ.

Ofta behövs t ex listor för att lagra objekt.
Om vi har listor som kan hantera godtyckliga objekt så skulle vi inte behöva skriva listan varje gång.

Vi behöver kunna hantera externa moduler och ha möjlighet att skriva sådana moduler.
Vi har redan kommit i kontakt med polymorfa metoder:

Vissa fördefinierade procedurer är polymorfa t.ex. +, -, /, …
Ibland kan samma kod användas:

(define (id x)
x)
I Pascal har vi bara möjlighet att använda polymorfi i fördefinierade procedurer. Alla egendefinierade procedurer är monomorfa
cos(x) tar både reella tal och heltal som argument.

Polymorfi forts

I samband med polymorfi är typkonvertering viktig.
Ex: 2+3, 2+3.0, 2.0+3, 2.0+3.0
kan alla beräknas eftersom argumenten i första och sista fallet har samma typ och i de två övriga konverteras en integer till en real.

I Java kan vi definiera överladdade metoder under förutsättning att indatatypen inte är densamma för två metoder.
Vi kan således definiera metoder med samma namn men med olika parameterlistor.
metoderna kan inte skilja sig enbart med avseende på returtyp utan måste skilja sig med avseende på antal parametrar, parametrarnas typ eller parametrarnas ordning (eller kombinationer).
Exempel:

```
public class complex {
    protected int re;
    protected int im;
    public complex(int r) {
        re = r;
        im = 0;}
    public complex (int r, int i) {
        re =r;
        im = i;}
    …
}
```

Enkla datatyper forts

Logiska värden

Domänen omfattar bara två värden: true och false.
boolean färdigMedAlltSomSkallGöras = false;
Operationsmängd: =, !, &, &&, |, ||, ^

Char

Anonyma konstanter omges av apostrofer: 'O', '7', '#'

Variabler:

char bokstav = 'A';
char siffa = '4';

• Varje tecken har sin egen kod.

– De flesta programmeringsspråk använder ASCII-systemet, som brukar en byte per tecken.
Detta ger maximalt 256 värden, men endast värdena 0-127 är standardiserade.
Java använder Unicode-kodsystem, som använder två byte per tecken. Detta ger maximalt 65536 tecken.
Tecken nr 0-127 sammanfaller med ASCII, å, ä och ö är standardiserade.
Vissa tecken skrivs med "\": '\b' backspace, '\f' formfeed, '\n' new line, '\r' carriage return, '\t' tab, '\\' backslach, '\'' single quote men alla tecken kan skrivas på formen \123
Operatorer: =, ==, !=, <, >, <=, >=

Enkla datatyper forts

Strängar

• Anonyma konstanter omges med """:
"Detta är en sträng " "A" ""
• Ex:
String stad = "Stockholm";
String text = "Detta är fyra ord.";

```
System.out.println("Vi har " + 2 + " grupper");
String städer = stad + " Malmö";
System.out.println(städer);
```

Strängar kan inte muteras!

Alla objektklasser har en metod toString.

toString anropas automatiskt då en operation förväntar sig en sträng som argument.
t ex då + används och det ena argumentet är en sträng.

Obs det är skillnad på "A" och 'A'

Prioritet

Operatorer och dess prioritet

Unärt +, -
++, -- (increment, decrement)
! (NOT)
(type) Typkonvertering
*, /, % (multiplikation, division, rest)
+, - (addition, subtraktion)
+ (konkatenering)
<, <=, =, >=, > ==, !=
& (AND)
^ (XOR)
| (OR)
&& (AND (kortslutande))
|| (OR (kortslutande))

Villkorliga uttryck: villkor ? värdeOmTrue : värdeOmFalse

```
a +(a>b ? a : b);

while ( i>0 ? a[i] != key : false) { …
}
Kan även skrivas:
while (i>0 && a[i] != key) { … }
```


String

Den fördefinierade klassen String. Några av de ca 50 metoderna i String:

```
String(String value) // Konstruktor
charAt(int index) //Tecken i position index. Obs startar på 0 så första tecknet är tecken 0!
int compareTo(Object o) // Jfr en sträng och ett objekt av godtycklig typ
String concat(String str)// konkatenerar strängar
boolean endsWith(String suffix)//slutar strängen med det givna sufixet?
boolean startsWith(String prefix)//börjar strängen med det givna prefixet?
boolean equals(Object anObject)// lika?
boolean equalsIgnoreCase(String anotherString)// lika sånärsom på gemena och versala tecken?
int indexOf(int ch) //index till ch
int lastIndexOf(int ch)//sök bakifrån
int length()//strängens längd
String replace(char oldChar, char newChar)//skapa en ny sträng med alla old utbytta mot new
String substring(int beginIndex, int endIndex)/skapa en delsträng
String toLowerCase()//gör om till gemener
String toUpperCase()//gör om till versaler
String toString()
String trim()//ta bort all blank info
String valueOf(double d)//tolka d som en sträng
String valueOf(float f)//tolka f som en sträng
String valueOf(int i)//tolka i som en sträng
String valueOf(long l)//tolka l som en sträng
String valueOf(Object obj) //tolka obj som en sträng
```

UML

Unified Modelling Language

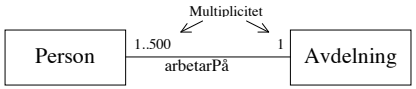
Klasser illustreras med klassdiagram
En klass beskriver en (abstrakt) datatyp **Objekt (instanser av klassen)**

bil	**Klassnamn**	**abc123:bil**
regNr **märke** **årsmodell** **hastighet**	**attribut (instansvariabler)**	**:bil**
start() **hastighet()** **öka(dv:Integer)** **bromsa(dv:Integer)** **stanna()**	**Metoder (Konstruktorer, mutatorer, selektorer och predikat)**	

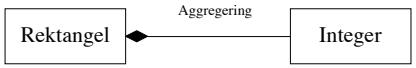
Association

En association är ett samband mellan klasser

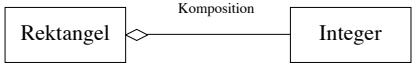
Pilen visar associationens riktning



Multipliciteten kan vara
3, 2..4, *



Aggregering kan tolkas som
”innehåller”



Komposition kan tolkas som
”sammansatt av”

XP igen

Vi försöker att kontrollera fyra variabler

- **Kostnad**
  Mer pengar kan lösa vissa problem men för mycket skapar nya
- **Tid**
  Mer tid gör att vi kan leverera mer men för mycket kan vara till skada
- **Kvalitet**
  Genom att tumma på kvaliteten kan vi göra kortsiktiga vinster men på sikt kostar det för mycket
- **Omfattning**
  Mindre omfattning gör att vi kan leverera tidigare med bättre kvalitet och till lägre kostnad

Det finns inget enkelt förhållande mellan dessa variabler

De fyra variablerna är inte oberoende:

- **Man kan inte producera koden fortare genom att spendera mer pengar**
- **Genom att öka budgeten kan man öka omfattningen eller kvaliteten**
- **Högre kostnad kan vara orsaken till att ett projekt misslyckas**
- **Högre kvalitet kan minska tiden**

Förändringskostnad

Vanligtvis ökar kostnaden, för förändring, exponentiellt över tiden.

Med XP verkar det som om kostnaden för förändring planar ut.

Detta därför att vi hela tiden testar (automatiskt), gör så lite som möjligt (inga extra finesser) gör saker på bara ett ställe, integrerar kontinuerligt och alltid omstrukturerar.

Vi blir p.g.a. detta vana vid att hela tiden förändra och modifiera designen
Grundläggande principer:

Snabb återkoppling: Kort tid mellan aktion och feedback
Antag enkelhet: Behandla problemet som om det kan lösas oerhört enkelt
Inkrementell förändring: problemet som en serie av små förändringar
Anamma förändring:
Bästa lösningen är att lösa det viktigaste först men att ha maximalt med möjligheter till variation kvar
Kvalitetsarbete:
Alla vill göra så bra jobb som möjligt

Basala element

- Kod
  Till slut är det bara koden som räknas oavsett hur många diagram du ritat och dokument du producerat
- Test
  Vi vet inte om något fungerar innan vi testat
  Testerna gör att vi kan tänka på ett problem på ett lite annorlunda sätt
- Lyssna
  Lyssna på kunden, domänexperter, …
- Design
  För att få ett bra system måste vi designa även i XP
  Om koden behöver kommenteras är detta ett tecken på att den är dålig.
  Ändra koden i stället för att kommentera den.

Vad är refaktoring?
Refactoring är en förändring av mjukvarans interna struktur som gör att den blir lättare att förstå och förändra den utan att förändra dess observerbara beteende

Ändra inte koden om du inte gör en uppenbar vinst!

Varför refactoring

- Refactoring förbättrar mjukvarans design
  Koden "ruttnar" med tiden

  Man gör kortsiktiga förändringar utan att tänka på designen
  Med kontinuerlig förändring av koden behövs vanligen mindre kod för att göra samma sak som den "fulare" koden

- Refactoring gör att det blir enklare att förstå mjukvaran
  Ett syfte med refactoring är ju att få snyggare kod …

- Refactoring hjälper dig att hitta buggar
  Man måste förstå koden för att kunna strukturerar om.

- Refactoring hjälper dig att programmera snabbare
  Kunskapen om koden ökar

När bör man göra refactoring

- 3-regeln
  Inte första gången
  Inte andra gången man gör något liknande

Men tredje gången

- När man lägger till funktionalitet
- När du måste fixa en bugg
- Vid kodinspektion

Varför fungerar refactoring?

- Program som är svåra att förstå är svåra att modifiera.
- Program med duplicerad logik är svåra att modifiera.
- Programförändringar som ger nytt beteende hos existerande kod är svåra att göra.
- Program med komplicerad villkorslogik är svåra att förändra.

Varför parprogrammering?

Alla lär sig koden.

Två programmerare som arbetar i par producerar mer och bättre kod än två ensamma Programmerare.

Två ser mer än en.

Man diskuterar koden.

Byte av partner.

Enhetstest

Fungerar som specifikation av programmet.

Skall vara automatisk.

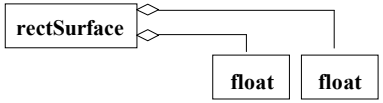
Skall köras efter varje förändring.

Skall fungera till 100%.

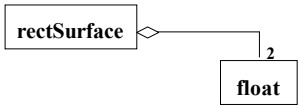
Dataintegritet

Vi måste skydda våra klasser mot felaktig användning.

Ex
Vi vill representera rektangulära ytor i ett program.



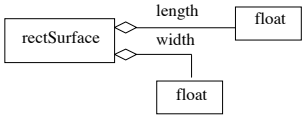
Aggregering används då ett objekt består (är sammansatt) av delar som i sin tur är objekt.
Ovanstående kan även åskådliggöras med:



Dataintegritet forts

Vilket betyder att en "rectSurface" består av exakt två "int".

Associationer och aggregat kan namnges.
Välj namn som är relevanta i sammanhanget.



Den som använder vår klass för att representera ytor i sitt program skall inte behöva känna till annat än protokollet för rectSurface.

Vi måste skriva klassen så att all normal användning blir korrekt.

Detta görs genom att inte representera sådant som kan beräknas och låta protokollet vara ortogonalt och minimalt.

Avvikelser kan tillåtas under vissa omständigheter.

Test för rectSurface

```
import junit.framework.*;
public class surfaceTest extends TestCase {
private surface s;

    public surfaceTest(String tc) {
        super (tc);
    }
    protected void setUp() {
        s = new surface(5, 7);
    }
    public void testLength() {
        assertEquals((float)5.0, s.length(), (float)0.00001);
    }
    public void testWidth() {
        assertEquals((float)7.0, s.width(), (float)0.00001);
    }
    public void testArea() {
        assertEquals((float)35.0, s.area(), (float)0.00001);
    }
}
```


Klassen rectSurface

```
public class surface {
    float length, width;

    public surface(float length, float width) {
        this.length = length;
        this.width = width;
    }
    public float length() {
    return length;
    }
    public float width() {
    return width;
    }
    public float area() {
    return length * width;
    }
}
```

Vilka problem kan inträffa om arean lagras i objekten?

Lösning: Skydda instansvariablerna (private eller protected)

Dataintegritet forts

Ortogonalitetsprincipen:
Representera inte sådant som är en kombination av övriga instansvariabler.

Detta gäller även gränssnittet: inför endast metoder för sådant som inte kan erhållas genom en sekvens av anrop av övriga metoder.

rectSurface
-length:float -width:float
+surface(length :float, width :float) +length() :float +width() :float +area() :float +setLength(newLength :float) +setWidth(newWidth :float)

+ public
- private
 # protected

Typen anges med :type

Metoder

• Kan endast förekomma i klasser (objekt).

• Inga metoder i metoder! Arv från C++ och C.

• Endast en anropsmekanism, värdeanrop. Alla värden kopieras.
För objektreferenser gäller att referensen kopieras – inte objektet.

• Endast enkla värden eller objektreferenser kan sändas som argument till metoder!
Man kan således inte sända metoder som argument eller få metoder som resultat.

• Skydd definieras "per klass", per arvskedja eller per paket – inte per objekt.
Detta får inte utnyttjas på sådant sätt att ett objekt uppdaterar ett annat objekts värden (instansvariabler) direkt. Man bör av säkerhetsskäl gå via metoder.

• Metodöverladdning är möjlig. En metod kan ta olika antal eller typ av parametrar.
Obs att det inte räcker med att uttypen är olika.

Överladdning

```
import java.lang.Math;
class exempel {
      protected int value;

      public exempel(){
      }
      public void setValue(float x) {
            value = (int)x;
      }
      public void setValue(int x) {
            value = x;
      }
      public void setValue(float x, float y) {
            value = (int)(sqrt(x*x+y*y);
      }
}
```

Följande anrop är korrekta:
x.setValue(3); x.setValue(3.3); x.setValue(3.3, 4.0); x.setValue(3, 4);

Värdeanrop

Metoden

```
public void swap(int a, int b) {
      int tmp = a;
      a = b;
      b = tmp;
}
```

är syntaktiskt korrekt men har inte avsedd effekt eftersom den enda anropsmekanismen som finns är värdeanrop.

Listor

Två möjligheter:
- **Kontinuerliga listor**
  - **array (fix storlek) och vector (storleken kan påverkas).**
- **Diskontinuerliga listor**
  - **ett antal fördefinierade varianter.**
  - **egendefinierade.**

Listorna skall konstrueras så klienten inte behöver ha kunskap om den inre strukturen (implementationsdetaljer).

Kontinuerliga listor

Arrayer

• Genereras alltid dynamiskt
```
int[] intvect = new int[100];
int[][] intmatrix = new int[20][30];
```

• Index går alltid från 0

• Övre gränsen blir ett mindre än längden.
```
int[] intvect = new int[25];
```

• Access som i Pascal:
```
intvect[23]
intmatrix[23][4]
intmatrix[3]
```

• längden kan läsas av med
```
intvect.length;
intmatrix[3].length
intmatrix.length
```


Användning av array

1. Sök efter det största elementet i en vektor med heltal

```
public int max (int[] storage) {
    int tmpMax = storage[0];
    for (int index = 1; index < storage.length; index++ )
        if ( storage[index] > tmpMax )
            tmpMax= storage[index];
    return tmpMax
}
```

2. Finns talet "key" med i vektorn?

```
public bool member(int storage[]; int key) {
    int index;
    for (index = 0; index < storage.length && storage[index] != key; index++) ;
    return index < storage.length;
}
```

Om listan är ordnad så kan villkoret index < storage.length && storage[index] != key ändras till index < storage.length && storage[index] < key

Hur påverkas prestanda?

Exempel forts

3 Skapa en enhetsmatris med n st rader.

```
public int[][] unitMatrix(int n) {
    int tmpMatrix[][] = new int[n][n];
    for (int row = 0; row < n; row++)
        for (int col= 0; col < n; col++)
            if (row == col)
                tmpMatrix[row][col] = 1;
            else
                tmpMatrix[row][col] = 0;
    return tmpMatrix;
}
```

Metodanrop sker med referensanropad array, d.v.s. referensen kopieras – inte arrayen.
Exempel: Skriv en metod som byter alla förekomster av "a" mot "b" i en array:

```
public void change(char[] txt) {
    for (int index = 0; index < txt.length; index++)
        if (txt[index] = 'a')
            txt[index] = 'b';
}
```


Array

• Arrayer betraktas som object (subklass till Object).

• Undre indexgräns är alltid 0.

• Endast integer som index.

• Arrayer skapas med "new" och är dynamiska variabler.

• Städas automatiskt bort då de inte längre refereras.

• Arraykomponenterna initieras till "0" (0, null, …).

• Skapande av arrayen och initiering kan ske med int b[] = {1, 2, 3}.

• Array är definierad som final.

• Kontroll att index uppfyller $0 \leq index \leq length-1$

• int[] a kan även skrivas int a[].

• lika många "[]" som index.

Lista

Vi behöver ofta arbeta med listor så vi definierar redan nu ett gränssnitt till en enkel listhanterare.
Vår lista är oordnad.
Listhanteraren har följande metoder:

insertFirst(el)	**Sätter in ett nytt element först i listan.**
insert(el)	**Sätter in ett nytt element efter markören.**
SetFirst()	**Sätter en "markör" på första elementet.**
setLast ()	**Sätter en "markör" på sista elementet.**
next()	**Stegar fram markören ett steg. Ingen effekt om markören står utanför listan.**
previous()	**Stegar bak markören ett steg. Om markören står utanför listan så har previous ingen effekt.**
delete()	**Raderar det element som markören pekar på. Om markören är utanför listan så har delete ingen effekt.**
Retrieve()	**Ger innehållet som markören pekar på. Om markören är utanför listan så ger retrieve null**
onList()	**True om markören inte är utanför listan**
full()	**True om listan är full**
empty()	**True om listan är tom**

Implementation

```
public class listHandler {
    protected int last;
    protected int current;
    protected element[] storage;
    public listHandler(int n) {
        storage = new element[n];
        last = -1; current = -1;
    }
    public void insert(element e) { }
    public void setFirst() { }
    public void setLast() { }
    public void next() { }
    public void delete() { }
    public void previous() { }
    public element retrieve() { }
    public boolean onList(){ }
    public boolean empty(){}
    public boolean full(){}
}
```


Implementation forts

```
public void insert(element e) {
    if (last < storage.length -1) {
        last++;
        storage[last] = e;
    }
}
public void setFirst() {
    if (last >= 0) current = 0;
}
public void setLast() {
    current = last;
}
public void next() {
    if (current >= 0 ) {
        current++;
        if (current > last) current = -1;
    }
}
public void delete() {
    if (current >= 0) {
        storage[current] = storage[last];
        last--;
        current = -1;
    }
}
...
```


Implementation forts

Vi kan låta lagringsutrymmet växa och krympa dynamiskt:

```
public void insert(element e) {
    if (current >= 0) {
        if (last == storage.length - 1)
            storage = copyStorage(new element[2* storage.length +1 ]);
        last++;
        storage[last] = e;
    }
}
protected element[] copyStorage (element[] newStorage) {
    for (int index = 0; index < storage.length; index++)
        newStorage[index] = storage[index];
    return newStorage;
}
public void delete() {
    if (current >= 0) {
        storage[current] = storage[last];
        last--; current = -1;
        if (last < storage.length / 2)
            storage = copyStorage(new element[storage.length / 2]);
    }
}
```

Länkad lista

**På samma sätt kan vi implementera stack, kö, …
men vi måste då hålla reda på ordningen bland elementen och inte som nu ha en oordnad lista.**

På kursbiblioteket finns en stack som är implementerad med hjälp av en vektor.

Det vore naturligare att implementera strukturer som skall vara dynamiska med hjälp av länkade listor.

**Här följer en skiss på en listhanterare som lagrar informationen i en enkellänkad lista.
Vi har samma gränssnitt som tidigare.**

```
public class listHandler {
    protected element first;
    protected element current;
    public listHandler() {
        first = null;
        current = null;
    }
    /* övriga metoder */
}
```


Listans element

**Elementen måste kompletteras med en referens till nästa element!
Men bör elementen i sig känna till andra element? Skall listan kunna manipulera elementens referenser? Hur löser vi detta?**

```
public class element {
    protected element next; int info;
    public element(int i) {
        next = null; info = i;
    }
    public void setNext(element e) {
        next = e;
    }
    public element getNext() {
        return next;
    }
    /* övriga metoder */
}
```

Men egentligen skall ju elementen inte behöva innehålla referenser som endast behövs för lagringen. Vi löser detta med s.k. kontainerelement.

Kontainerelement

Listan har ett element som bara innehåller en länk till nästa element och en länk till informationen (det element son står i listan.

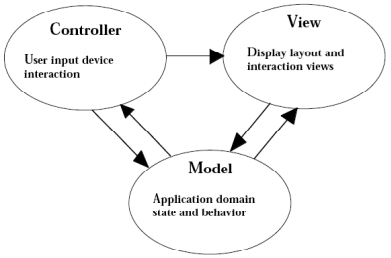
```
class container {
    element e;
    container next
}
```

Vi gör klassen lokal i listhanteraren:

```
public class listHandler {
    class container {
        element e;
        container next
    }
    protected container first;
    protected container current;
    public listHandler() {
        first = null;
        current = null;
    }
    /* övriga metoder */
}
```

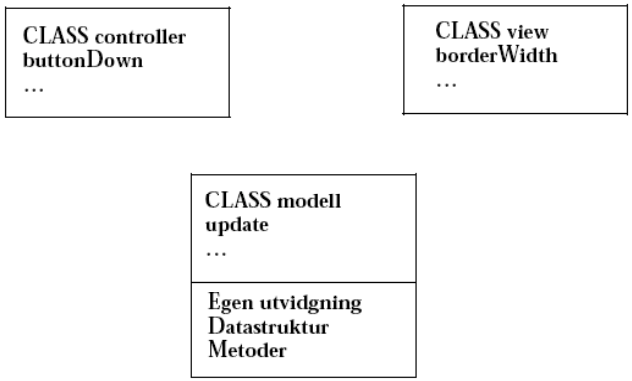

Model-View-Controler

**Ett sätt att dela upp funktionaliteten mellan olika moduler.
En modul som sköter om respons från användaren.
En modul som sköter om layout och interaktion på skärmen.
En modul som kommunicerar med det egna programmet, dvs innehåller datastruktur och accessfunktioner.
Denna modul håller reda på egna datastrukturens tillstånd och uppförande.**



MVC forts

En fördel är att mycket av funktionaliteten kan vara klar då vi bygger vårt program så att vi inte behöver starta från grunden varje gång.

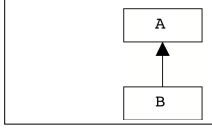


Subklasser

Vi kan låta en klass ärva från en annan klass. Man ärver då alla egenskaper och metoder. Vissa metoder och egenskaper kan vara skyddade så att klassen som ärver inte kan manipulera dem.

```
public class rat extends number {
    protected int denom;
    public rat(int t, int n) {
        super(t);
        denom = n;
    }
    public number add(rat term) {
        if (term.getClass() == getClass())
        < addera rationellt tal och rationellt tal >
        else
        <skicka vidare med detta tal konverterat till argumentets typ>
    }
    public rat convert(number n) {
        return new rat(n.getValue(), 1);
    }
}
```

A är superklass till B och
B är subklass till A.



Subklasser forts

"Rat" kommer att få alla attribut som number har samt de vi definierar i rat. Vid namkollisioner "ser" vi den metod (egenskap) som senast definierades.

getClass ger klassen för ett objekt.

Observera att allt som inte specificerats som private eller protected är tillgängligt bara man har lämplig kvalifikation hos referensvariabeln!

Vi kan bygga en hierarki av klasser genom att definiera en gemensam "superklass" och sedan lägga till egenskaper och metoder (ev modifiera metoder) i subklasserna.

Alla metoder som skall användas från en referens med kvalifikation A (referensens typ är A) skall finnas definierade i A eller en superklass till A.

Om man inte kan definiera metodens kropp i A så kan klassen definieras som abstrakt. Abstrakta metoder måste definieras i subklasser.

Exempel:

```
public abstract class a {
    public int getValue(); /* metodens typ och indata låses här */
}
```


Subklasser forts

Number tillåter att vi opererar på int och int, int och rat, rat och int samt rat och rat

```
public class number {
    protected int value;

    public number(int value) {
        this.value = value;
    }
    public void print() {
        System.out.println(value);
    }
    public int value() {
        return value;
    }
    public number add(number n) {
        return (n.getClass() == this.getClass()) ?
            new number(value+n.value) :
            n.convert(this).add(n);
    }
    public number convert(number n) {
        return n;
    }
}
```

Överladdade metoder

Vi kan definiera metoder med samma namn men med olika parameterlistor.

Metoderna kan inte skilja sig enbart med avseende på returtyp utan måste skilja sig med avseende på antal parametrar, parametrarnas typ eller parametrarnas ordning (eller kombinationer).

Då ovanstående är uppfyllt kan även uttypen skilja sig mellan metoderna.

Exempel:

```
public class complex {
    protected int re;
    protected int im;

    public complex(int r) {
        re = r;
        im = 0;
    }
    public complex (int r, int i) {
        re =r;
        im = i;
    }
…
}
```


Överskrivning

Vi kan omdefiniera metoder i subklasser.

Exempel:

```
public class A {
    int i = 1;

    public int f() {
        return i;
    }
}

public class B extends A {
    int i = 2;

    public int f() {
        return i;
    }
}
```

Observera att man I Java inte får ändra uttypen i detta fall.

Åtkomst

Värden "skuggas" medan metoder blir "överskrivna".

Instansvariabler med samma namn kan man komma åt genom typkonvertering.

(A) b.i b:s typ konverteras till A.

Metoder kan deklareras som final. Dessa metoder kan inte skrivas över.

Instansvariabler som deklarerats som final kan inte ändras.

En överskriven metod kan användas i klasskroppen genom super.metodnamn

Exempel:

```
public class A {
    protected int i = 1;
    public int f() { return i; }
}
public class B extends A {
    public B() {
    i = 2* super.f();
    }
    protected int i = 2;
    public int f() { return i; }
}
```


Stack

```
public class genericstack {
    private stackElementContainer top;

    public genericstack () { top = null;}

    protected class stackElementContainer {
        Object obj;
        stackElementContainer next;
        public stackElementContainer (Object obj, stackElementContainer next) {
            this.obj = obj;
            this.next = next;
        }
    }
    public Object top () { return top.obj;}

    public boolean empty () { return top == null;}

    public boolean full () { return false; }

    public void push(Object obj) {
        top = new stackElementContainer(obj, top);
    }
    public void pop () { top = top.next; }
}
```

Klassen kan hantera alla element som är subklasser till Object (d.v.s. alla klasser).

Stack, forts

Ett problem med stacken är att alla objekt som hämtas från stacken är av typen ”Object”.
Ex: Vi har en stack med Integer. Vi vill summera alla tal.

```
for (Object i = s.top(); !s.empty(); i = s.top()) {
    sum = sum + i.intValue(); // kompileringsfel efersom i är av typen Object
    s.pop();
}
```

Vi måste göra på följande sätt:

```
for (Object i = s.top(); !s.empty(); i = s.top()) {
    sum = sum + ((Integer)i).intValue();   s.pop();
}
```

Det vore bättre om stacken gav oss objekt av typen Integer:

```
public class genericstack {
    …
    protected class stackElementContainer {
        Integer obj; // Detta innebär att vi bara kan hantera Integer !
        stackElementContainer next;
        public stackElementContainer (Integer obj, stackElementContainer next) {
            …
        }
    }
    public Integer top () { return top.obj;}
    …
}
```


Generiska klasser

Vi kan ge klassen en typparameter för att kunna anpassa den till vårt behov:

```
public class genericstack < T > {
    private stackElementContainer top;

    public genericstack () { top = null;}

    protected class stackElementContainer {
        T obj;
        stackElementContainer next;
        public stackElementContainer (T obj, stackElementContainer next) {
            this.obj = obj;
            this.next = next;
        }
    }
    public T top () { return top.obj;}

    public boolean empty () { return top == null;}

    public boolean full () { return false; }

    public void push(T obj) {
        top = new stackElementContainer(obj, top);
    }
    public void pop () { top = top.next; }
}
```


Användning av stacken

```
stack<Integer> s = new stack<Integer> ();
…
for (Integer i = s.top(); !s.empty(); i = s.top()) {
    sum = sum + i.intValue();
    s.pop();
}
```

Men Integer kan automatiskt konverteras till int (detta gäller alla wrapperklasser) så vi kan skriva:

```
stack<Integer> s = new stack<Integer> ();
…
for (Integer i = s.top(); !s.empty(); i = s.top()) {
    sum = sum + i;
    s.pop();
}
```

Vi kommer att få problem med subklasser. Antag att B är subklass till A.

`stack<B>` är inte subklass till `stack<A>` m a o vi kan inte göra på följande sätt:

`stack<Object> s = new stack<Integer>`. (Vi kan inte göra: `s.push(new Object())`.)

Okänd typ

Antag att vi vill göra en metod som räknar antalet element i en stack:

```
int cnt(Stack<Object> s) {
    Object el;
    if (s.empty()) return 0;
    else {
        el = s.top();
        s.pop();
        int n = 1 + cnt(s);
        s.push(el);
        return n;
    }
}
```

Problemet är att vi inte kan använda metoden för att räkna antalet element i `stack<Int>`.

Supertypen till alla stackar är: `stack<?>` så metoden blir:

```
int cnt(Stack<?> s) {
    Object el;
    …
}
```

Observera att inuti `cnt` kan vi fortfarande använda typen Object för elementen som ju också har typen Object.

Okänd typ, forts

Eftersom alla uttryck måste vara säkra så måste man vara försiktig:

```
Stack<?> s = new stack<Integer>;
Object el = s.top();
```

Kommer att gå bra eftersom top alltid ger ett element som är subklass till `Object` men:

```
s.push(new Integer(3));
```

Om vi, i metoden `cnt`, vill använda egenskaper i elementen så måste vi veta dess typ (eller deras supertyp). Vi kan då skriva en generisk metod:

```
public <T> int cnt(stack<T> s) {
    T el;
    // Elementens typ härleds och vi kan använda dess egenskaper
}
```

Ytterligare ett problem kvarstår: antag att stacken innehåller element som kan tillhöra en subklass till T. Vi kan lösa det på följande sätt:

```
public <E extends T> int cnt (stack<E> s) {
    E el;
    …
}
```


Okänd typ, forts

Vi kunde också ha skrivit:

```
public int cnt (stack<? extends E> s) {
    …
}
```

Vi vill kopiera elementen i en stack till en annan (i omvänd ordning):

```
public <T> void copy (stack<T> dest, stack(? extends T> src) {
    …
}
```

Vi kan nu kopiera alla element som är subtyp till T från `src` till `dest`
Det hade även gått bra att skriva:

```
public <T, S extends T> void copy (stack<T> dest, stack<S> src) {
    …
}
```


Lista igen

I en ordnad lista måste man kunna jämföra elementen med varandra:

```
private class containerElement {
    element theElement;
    containerElement next;
    public containerElement(element e, containerElement next) {
        theElement = e;
        this.next = next;
    }
    public containerElement next() {
        return next;
    }
    public containerElement setNext(containerElement next) {
        this.next = next;
        return this;
    }
    public element retrieve() {
        return theElement;
    }
    public boolean equal(containerElement e) {
        return theElement.equal(e.retrieve());
    }
    public boolean lessThan(element e) {
        return theElement.lessThan(e.retrieve());
    }
} /* containerElement */
```


Lista forts

Vi måste nu ha metoder i element som stämmer med detta gränssnitt:

```
public abstract class element {
    public element() {
    }
    public abstract boolean equal(element e);
    public abstract boolean lessThan(element e);
    public abstract boolean greaterThan(element e);
    public abstract boolean greaterOrEqual(element e);
    public abstract element retrieve();
}
```

Det enda vi nu behöver göra är att deklarera en subklass till element så kan vi arbeta med listor av sådana element.

Vi vill ha en lista av heltal:

```
public class intElement extends element {
    private int value;
    public intElement(int value) {
    super(); /* initiering av superklassen */
    this.value = value;
    }
    …
}
```

Iterator

Javas standardmetod för att iterera är att datastrukturen innehåller en klass som implementerar ett gränssnitt "Iterator".

Metoderna "hasNext()", "next()" och "remove()" skall finnas.

```
public class list {
    container first;
    public list () { first = null; }

    class container {
    …    /* som tidigare */
    }
    …  /* alla metoder I gränssnittet */
    public class iterator implements Iterator {
        container c = first;
        public boolean hasNext() {
        return c != null; }
        public Object next(){
            Object theObj = c.obj;
            c = c.next;
            return theObj;
        }
        public void remove() {}
    }
    public Iterator iterator() { return new iterator(); }
}
```


Testprogram

Ett enkelt program som testar iteratorn:

```
public class mainTest {
    public static void main(String[] a) {
        list L = new list();
        L.insert(new number(7));
        L.insert(new number(6));
        L.insert(new number(5));
        L.insert(new number(4));
        Iterator i = L.iterator();
        while (i.hasNext()) {
            System.out.println(((number)i.next()).value());
        }
        i = L.iterator();
        while (i.hasNext()) {
            L.delete((element)i.next());
        }
    }
}
```


API

Application programmer interface.

Klassbibliotek och paket.

Vi menar alla klasser och deras protokoll i en viss samling av (fördefinierade) klasser.

Det finns API för olika ändamål t ex

- **Databaskoppling (JDBC)**
- **Grafiska användargränssnitt java.awt**
- **Program som kan transporteras via weben java.applet**
- **Kommunikation via nätverk java.net**

…

Vi kan definiera egna paket:

Skapa ett bibliotek där alla klasser samlas som skall ingå i paketet.

Vi får tillgång till paketen genom:

import paket;

eller fullt ange sökvägen.

Programmeringsspråk

Uppgiften hos ett programspråk är att hjälpa programmeraren genom att förse honom med:

- **– Datatyper och operationer**
- **– Kontroll och begränsningar**
- **– Abstraktion**
- **– Beräkningsmodell**
- **–**

Syntaxen i ett programspråk beskriver den korrekta formen på vilket ett program kan skrivas.

Semantiken beskriver den mening en syntaktisk konstruktion har.

Syntaxen kan beskrivas med syntaxdiagram eller BNF (EBNF)
Syntaxdiagrammen byggs upp av två delar:
Sådant som Sådant som inte

behöver definieras vidare **def. i annan graf**

Grammatik

Definition:

En kontextfri grammatik har fyra delar:

1. **En uppsättning terminala symboler.**
2. **En uppsättning icketerminala symboler**
3. **En uppsättning produktionsregler, där varje produktion har en icketerminal på vänster sida, ett "::=" och en sträng av icke terminala och terminala symboler**
4. **En startsymbol (som tillhör mängden icketerminala symboler.**

BNF (Backus-Naur Form)
Textuell notation för att beskriva en kontextfri grammatik.

Ex:
<real-number> ::= <digit-sequence> '.' <digit-sequence>
<digit-sequence> ::= <digit> | <digit> <digit-sequence>
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

<digit> är en icketerminal
'1' … är teminala symboler.

Exempel

<expression> ::= <expression> + <term>
| <expression> - <term>
| <term>

<term> ::= <term> * <faktor>
| <term> div <faktor>
| <faktor>

<faktor> ::= (<expression>)
| <variable>
| <constant>

EBNF notation:

- **| eller**
- **( ) för gruppering**
- **{ } noll eller fler förekomster**
- **[ ] noll eller en förekomst**

Ex.

expression ::= term { ('+' | '-') term }

term ::= faktor { ('*' | 'div') faktor }

faktor ::= '(' expression ')' | variable | constant

Vi kan ha vänster- eller högerrekursiva grammatiker.

Vi beskriver våra programspråk med hjälp av EBNF eller syntaxdiagram.

Ett språks semantik beskrivs ofta med ett antal exempel.

Typerna används för att göra semantisk kontroll.

Rekursiv medåkning

Lexikal analys: Bryt ner indata i individuella "ord" lexem.

Parsing: analysera frasstrukturen (syntaxen).

Semantisk analys: Undersök meningen.

En välkänd enkel metod för parsing är rekursiv medåkning (recursiv decent).

Varje icketerminal symbol är associerad med en metod i programmet.

Ex aritmetiska uttryck:

```
public class treeBuilder {
parser p;
public treeBuilder(parser p) {this.p = p;}
public expression buildExpression() {return expr();}
```


Rekursiv medåkning forts

```
public expression expr() { // expr ::= [ '-' ] term { ( '+' | '-' ) term }
    expression e = null;
    if (p.more()) {
        if (p.lookahead("-")) {
            p.nextToken();
            e = new unarySubOperator(term());
        }
        else e = term();
        while (p.more() && (p.lookahead("+") || p.lookahead("-"))) {
            String addOp = p.nextToken();
            expression t = term();
            if (addOp.equals("+")) e = new addOperator(e, t);
            else e = new subOperator(e, t);
        }
    }
    return e;
}
private expression term() { // term ::= factor { ('*'|'/') factor}
    expression t = factor();
    while (p.more() && (p.lookahead("*") || p.lookahead("/"))) {
        String mulOp = p.nextToken();
        expression f = factor();
        if (mulOp.equals("*")) t = new mulOperator(t, f);
        else t = new divOperator(t, f);
    }
    return t;
}
```

Rekursiv medåkning forts

```
    private expression factor() {
        expression f;
        if (p.lookahead("(")) {
            p.nextToken();
            f = expr();
            p.nextToken();
        }
        else
            f = new constant(p.doubleToken());
        return f;
    }
}
```

Vi måste ha en klassstruktur för att representera de olika operatorerna och operanderna.

Interface

Vi kan specifiera ett minimalt *protokoll* för ett antal klasser.

Alla klasser som implementerar ett interface får också dess typ.

Exempel
Vi vill kunna implementera olika listor, var och en med lite olika egenskaper.
Vi vill kunna byta en lista mot en annan.
Med följande mängd funktioner kan detta realiseras:

```
insert(e)
delete(e)
retrieve(e)
doToEach(f)
```

Detta protokoll måste vara en del av protokollet för en klass som implementerar det specifierade gränssnittet.

I ett interface kan, förutom protokollet, även konstanter ingå. Dessa måste deklareras som `"static final"`.

```
interface listInterface {
    public void insert(element e);
    public void delete(element e);
    public element retrieve(element e);
    public void doToEach(method m);
}
```


Interface forts

En klass kan implementera ett eller flera gränssnitt och samtidigt ärva från en klass.

Vi kan med detta simulera multipelt arv.

```
public class list implements listInterface {
    // instansvariabler
    protected element first;
    // Konstruktor
    list() {}
    …
}
```

Klassen

```
    public abstract class method {
        public abstract void doIt(element e);
    }
```

kommer aldrig att användas direkt.

Vi kommer alltid att bilda subklasser till "method" som sedan instansieras.

En klass som innehåller abstrakta metoder måste vara abstrakt.

Interface forts

En abstrakt klass har nästan samma funktion som ett interface.

Skillnaden är att den abstrakta klassen kan innehålla instansvariabler som inte är konstanter och metoder som inte vidare behöver specifieras.
I ett interface är alla metoder abstrakta och alla instansvariabler är konstanta.

En abstrakt klass kan innehålla instansvariabler, abstrakta och precicerade metoder, implementera interface och innehålla godtyckliga definitioner.

Vi kan implementera en lista genom att använda Javas gränssnitt "Collection" och iterera över elementen med hjälp av en iterator á la Java.

Interface Collection:

```
boolean add(Object o)
boolean addAll(Collection c)
void clear()
boolean contains(Object o)
boolean equals(Object o)
int hashCode()
boolean isEmpty()
Iterator iterator()
boolean remove(Object o)
int size()
Object[] toArray()
Object[] toArray(Object[] a)
```

Åtkomst

public, protected och private talar om vem som får se data (metoder)
Detta kallas för 1:a ordningens skydd.

Det som deklareras i ett lokalt scoop och som därför inte kan ses från omgivningen kommer också att skyddas.

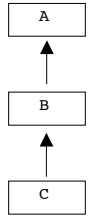
Vi har i detta fall en form av semantiskt skydd av data (metoder).
Detta kallas 2:a ordningens skydd.

Det skydd vi förut beskrivit är en form av syntaktiskt skydd.

I Java ligger skyddet på klassnivå och två objekt ur samma klass kan manipulera varandras instansvariabler utan hinder även om de deklarerats som Private.

OO begrepp

Protokoll = mängden meddelanden ett objekt förstår.
Gränssnitt mängden metoder som är synliga i ett object.
I Java menas den "minsta" profil ett objekt måste ha för att implementera ett gränssnitt. (I vårt fall så kallar vi detta för protokoll.)
Inkapsling delas in i första och andra klassens skydd och innebär att allt som inte innehålls i klassens protokoll skyddas.
Dynamisk bindning:
Betrakta följande klassstruktur:
Om alla har en metod m och ett program innehåller satsen A.m() vet vi inte vid kompileringstillfället vilken metod som skall användas.



Vi måste avgöra detta vid körningen av programmet.
Man talar då om sen eller dynamisk bindning.
Abstrakt datatyp
En av användaren definierad datatyp (en mängd värden och en mängd operationer)
Operationsmängden är ADTens protokoll och domänen är mängden tillstånd objektet kan anta.

Begrepp

Listan som tidigare har beskrivits är en ADT.
Fördelar med ADTer:
- **de kan specifieras matematiskt**
- **de kapslar in data och användaren behöver endast känna till dess protokoll**
- **vi kan definiera subtyper som ärver från supertypen. På detta sätt kan polymorfa typer definieras.**
  typkonvertering, överladdning osv kan användas.

Kontraktsprogrammering:
om alla programdelar innehåller pre- och postvillkor så vet man att respektive programdel fungerar bara previllkoret är uppfyllt.
Kontraktet är mellan te x en ADT och användaren.

Arv eller aggregering?

Om typen B är en sorts A skall arv användas.
Om A består av B skall aggregering användas.

Exempel:
Vi vill skriva en ADT stack.

Arv eller aggregering, forts

Det går bra att bilda en subklass till listan och där definiera om insert, delete och retrieve.
push(e) skall lägga ett nytt element överst på stacken

top skall ge översta elementet på stacken

pop skall ta bort översta elementet på stacken.

Om insert och definieras så att insättning sker först och delete tar bort det första elementet samt retrieve återvinner informationen i det första elementet så kommer det att fungera.

```
public class stackList extends list {
    public void insert(element e) {
        first = new containerElement(e, first);
    }
    public void delete() {
        if (first != null)
        first = first.next();
    }
    public element retrieve() {
        return (first != null) ? first.retrieve() : null;
    }
}
```

Arv eller aggregering, forts

Och stacken:

```
public class stack extends stackList {
    public void push(element e) {
        insert(e);
    }
    public void pop() {
        delete();
    }
    public element top() {
        return retrieve();
    }
    public boolean empty() {
        private int n = 0;
        doToEach(fun (element e)->void {n++});
        return n == 0;
    }
}
```

Nu kommer stacken att ha ett protokoll som är summan av klasskedjans protokoll t ex kan vi använda doToEach i stacken eller insert istället för push.

Man säger att protokollet är fett!

Vi vet från tidigare diskussion att ett protokoll bör vara ortogonalt och minimalt men här är inget av dessa krav uppfyllt!

Aggregering

Delegering:
En stack består av en lista.

```
public class stack {
    private stackList s;
    public stack(){
        s = new stackList();
    }
    public void push(element e) {
        s.insert(e);
    }
    public void pop() {
        s.delete();
    }
    public element top() {
        return s.retrieve();
    }
    public boolean empty() {
        private int n = 0;
        s.doToEach(fun (element e)->void {n++});
        return n == 0;
    }
}
```

De enda metoder som är synliga är pop, push, top och empty (och konstruktorn).
Vi kan inte använda stacken på fel sätt!

Gränssnitt

Användning av gränssnitt:

```
public interface drawable {
    public void draw(drawWindow dw);
    public void erase(drawWindow dw);
    public color setColor(color c);
}

public class circle implements drawable {
    private color c;
    private point position;
    private float radius;
    public void draw(drawWindow dw) {
        dw.drawCircle(position, radius);
    }
    public void erase(drawWindow dw) {
        dw.eraseCircle(position, radius);
    }
    public color setColor(color c) {
        this.c = c;
    }
}
```


Gränssnitt forts

```
public class rectangle implements drawable {
    private color c;
    private point position;
    private float length, width;
    public void draw(drawWindow dw) {
        dw.drawRectangle(position, length, width);
    }
    public void erase(drawWindow dw) {
        dw.eraseRectangle(position, length, width);
    }
    public color setColor(color c) {
        this.c = c;
    }
}
```

Båda dessa klasser kan sändas som argument till en metod vars formella parameter är av typen drawable

Exempel:

```
public void show(drawable d, drawWindow dw) {
    dw.draw(d);
}
```

Det får finnas ytterligare metoder i klasserna som implementerar ett interface.

Gränssnitt

En klass kan ärva och implementera ett eller flera gränssnitt!

```
public abstract class shape {
    public abstract double area();
}

class square extends shape implements drawable {
    private float width;
    public square(float w) {
        super(); width = w;
    }
    public void draw(drawWindow dw) {
        dw.drawRectangle(position, length, width);
    }
    public void erase(drawWindow dw) {
        dw.eraseRectangle(position, length, width);
    }
    public color setColor(color c) {
        this.c = c;
    }
    public double area() {
        return w * w;
    }
}
```

På detta sätt kan multipelt arv simuleras i Java.

Generiska gränssnitt

Ett gränssnitt kan göras generiskt på samma sätt som klasser.
Följande beskriver interfaceen List och Iterator som finns i java.util.

```
public interface List<E> {
    void insert(E x);
    Iterator<E> iterator();
    …
}

public interface Iterator<E> {
    E next;
    Boolean hasNext();
    …
}
```

Vi kan ha typparametrar, okända typer o s v I gränssnitten.

Package

Vi kan samla klasser i s k klassbibliotek. Dessa klasser kan vara kompilerade så att man endast behöver referera till dem.

Dels finns det standardbibliotek dels kan vi själva bygga bibliotek med användbara klasser.

Vi kallar sådana bibliotek för "packages".

```
package bin.java.storage;
public class listHandler {
    /* hela definitionen av listHandler */
}

package bin.java.storage;
public class stack {
    /* hela definitionen av stack */
}
```

kompilera med:

```
javac -d bin listHandler.java
javac -d bin stack.java
```


Package, forts

Kompilatorn placerar de kompilerade filerna i biblioteket bin/java/stoarge/
-d behövs för att kompilatorn skall kunna göra bibliotek om de inte redan finns.
I vårt program kan vi nu skriva:

```
import bin.java.storage.*;
public class user {
    stack s = new stack();
    listHandler L = new listHandler();
    …
}
```

Om vi inte specifierar skyddet för instansvariabler och metoder så får vi automatiskt "package".

Detta betyder att om ett program använder flera klasser från samma paket så kan dessa klasser "se" varandras instansvariabler och inre tillstånd!

Undantagshantering

Vid körning av program kan en mängd fel inträffa.

Exempel:
- division med 0
- användaren matar in felaktiga data
- referenser är null vid access
- avbrott i kommunikation

…

I Java kan avbrott genereras i extraordinära situationer (inte som del i ett normalt förlopp).

Ett sådant avbrott måste tas om hand och hanteras i programmet.

Avbrottet genereras i den del av programmet där felet uppstår och hanteras i den del av koden där analys av felet kan ske.

Den del av koden som kan ge upphov till avbrottsgenerering omges av:

```
try { kod som kan ge upphov till avbrott }
catch (feltyp objektReferens) {
del av koden där felet hanteras }
…
```


Undantagshantering forts

Avbrott genereras av:

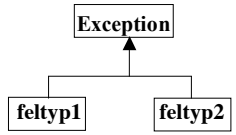
throw new felTyp();

Där "felTyp" är Exception eller subbklass till Exception.

Exception är en subklass till Throwable.

Det finns en mängd fördefinierade feltyper och vi kan själva definiera nya feltyper som subklass till de redan befintliga.

Om följande klasshierarki är definierad
så måste analysen (då fel av typ 1 och 2 kan inträffa)
ske med början från feltyp2:




```
try { kod som kan ge upphov till fel av typen 1 och 2 }
      catch (FelTyp2 ft2) { hantera felet av typ 1}
      catch (FelTyp1 ft1) { hantera felet av typ 2}
finally { satser som alltid skall exekveras }
```

Felet genereras i en metod där man deklarerat att viss typ av fel kan genereras:

```
Ex: public void delete(node aNode) throws notFoundException {
      …
```


Undantagshantering forts

Metoden där felet tas om hand måste antingen ta hand om alla feltyper som kan genereras av de metodanrop som sker då metodkroppen exekveras eller sända dem vidare.

Metoden måste då vara definierad på följande sätt:

```
accessmodifierare typ namn (parameterlista) throws
felSomSkallSändasVidareTillAnropandeMetod{
      …
}
```

Händelser som inte kan hanteras med denna typ av felhantering
- användaren trycker på mustangenten eller en tangent på tangentbordet
- meddelanden kommer in via nätet

Avbrott skall endast genereras då någon typ av fel har inträffat som skulle ha orsakat programavbrott om felet inte hade hanterats med hjälp av avbrottshanteringen.

Undantagshantering exempel

Exempel med den generiska stacken:

```
public class stack {
    private stackElementContainer top;
    public stack () {
        top = null;
    }
    public class emptyStackException extends Exception {
        public emptyStackException() {
        super("Stack is empty");
        }
    }
    public class fullStackException extends Exception {
        public fullStackException() {
        super("Stack is full");
        }
    }
    protected class stackElementContainer {
        Object obj;
        stackElementContainer next;
        public stackElementContainer (Object obj, stackElementContainer next) {
            this.obj = obj;
            this.next = next;
        }
    }
```

Exempel forts

```
public Object top () throws emptyStackException {
    if (top != null)
        return top.obj;
    else
        throw new emptyStackException();
}
public int cardinal () {
    int n = 0;
    for (stackElementContainer s = top; s != null; s = s.next)
    n++;
    return n;
}
public boolean empty () {
    return top == null;
}
public boolean full () {
    return false;
}
public void push(Object obj) throws fullStackException {
    if (! full())
        top = new stackElementContainer(obj, top);
    else
        throw new fullStackException();
}
```


Exempel forts

```
public void pop () throws emptyStackException {
    if (top != null)
        top = top.next;
    else
        throw new emptyStackException();
}
}
```

Och ett program som använder stacken:

```
import stack.emptyStackException;
import stack.fullStackException;
class main {
    public static void main (String[] args) {
        stack s = new stack();
        try {
            s.pop();
        }
        catch (emptyStackException ese) {
            System.out.println("tom stack");
        }
```


Exempel forts

```
        try {
            for (int i=2 ; i < 20; i++, i++)
                s.push(new number(i));
        }
        catch (fullStackException fse) {
            System.out.print(fse.toString());
        }
        boolean ready = false;
        while (! ready) {
            try {
                number n = (number)s.top();
                s.pop(); System.out.println(n.value());
            }
            catch (emptyStackException ese) {
                System.out.print("tom stack"); ready = true;
                System.out.print(ese.toString());
            }
        }
        System.out.println();
    }
}
```

Man måste ta hand om alla tänkbara fel som kan inträffa (alla subklasser till Exception)

Exempel forts

Koden kan se ut så här:

```
stack s = new stack();
try {
    s.pop();
}
catch (emptyStackException ese) {
    System.out.println("tom stack");
}
catch (Exception e) {}
try {
    for (int i=2 ; i < 20; i++, i++)
    s.push(new number(i));
}
catch (fullStackException fse) {
    System.out.print(fse.toString());
}
catch (Exception e) {}
```

Observera ordningen: ta först hand om de specifika felen sedan fel som är superklasser till de som redan hanterats.

Antag att felen inte kan hanteras i den metod som hämtar element från stacken:

Exempel forts

```
public void moveNumber(stack s1, stack s2) throws Exception {
    s2.push(s1.top());
    s1.pop();
}
```

Metoden som anropar moveNumber:

```
public void build(String expr) {
    private stack s = new stack();
    private stack help = new stack();
    …
    try { moveNumber(s, help);
    …}
    catch (emptyStackException ese) {
        här får vi fixa felet som beror på att help är tom
    }
    catch (fullStackException fse) {
        här fixar vi felet som beror av att s är full
    }
    catch (Exception e) {}
}
```

Om felet förmedlas i flera steg i en anropskedja kan det vara av intresse att veta vilka anrop som har avvecklats efter det att felet genererades.

Man kan då anropa printStackTrace() i en Exception.

Undantagshantering forts

Vissa strukturer som skapats i ett "try-block" måste städas oavsett om ett fel inträffar eller ej.

Ex
En fil öppnas i try-blocket men inläsningen misslyckas.
Try-blocket avbryts och ett catch-block fångar upp felet.

```
try {…}
catch (somException e) {…}
finally {detta block exekveras i alla lägen
}
```

Finally exekveras

1. **om try exekveras i sin helhet (inget fel inträffar)**
2. **om return break eller continue exekveras så exekveras ändå finally-blocket**
3. **om fel inte hanteras utas sänds vidare exekveras först finally-blocket**
4. **om ett lämpligt catch-block hanterar felet kommer finally-blocket att exekveras efter catch-blocket.**

Undantagshantering forts

Ex

```
try {
öppna filen
läs information från filen
}
catch ( FileNotFounException e) {felhantering för detta fall}
catch ( IOException e) {felhantering för detta fall}
…
catch ( Exception e) { … }
finally {om filen är öppen så stäng den
…
}
```

Fel genereras då man exempelvis matar in ett för stort heltal, skriver något som inte kan tolkas som ett tal då det förväntas ett sådant osv.

Använd avbrottsgenerering och hantering då fel uppstår som annars är svåra att åtgärda.

Man kan t ex upptäcka att det inte finns några siffror då man vill mata in ett tal och i en loop uppmana användaren att mata in ett tal.

Med avbrottshantering blir det betydligt enklare!

Filer

En fil är en följd av tecken, vanligtvis stor och vanligtvis lagrad på ett externt lagringsmedium.
Data i de flesta filer är partitionerade i "images" (poster, rader) där var och en består av ett antal tecken. En sådan fil kallas imagefile.
Om en fil inte organiseras i images blir det en ström av tecken (bytes) och vi kallar en sådan fil "bytefile".
Filer kan vara:
1. sekventiella dvs vi startar med en tom fil (utfil) och skriver sedan image efter image på filen. (Vid läsning startar vi från filens första image och läser sedan image efter image från filen.)
En sekventiell fil öppnas för att antingen läsas eller skrivas.
2. random access (direktfil). Vi kan både läsa och skriva på filen.
Direktfilen består av ett antal platser numrerade från 0 uppåt. Vissa platser kan vara tomma.
Posterna på filen kan skrivas eller läsas i en godtycklig ordning.

Filer, forts

En fil kan vara av typen text och kommer då att vara, förutom maskinläsbar, humanläsbar.

Skillnaden mellan bytefiler och textfiler är att informationen kan packas bättre på byteorienterade filer.

En fil kan vara "öppen" vilket innebär att filens poster (bytes) är tillgängliga för att läsas (skrivas) eller "stängd" vilket innebär att innehållet inte är tillgängligt.
En extern fil måste ha ett namn.

Strömmar

Vi behöver en koppling mellan den externa enheten och programmet. I Java används strömmar för att åstadkomma denna koppling.

En ström är en ADT som är avsedd att hantera en sekvens av data som "strömmar" mellan program eller program och olika enheter (skärm, tangentbord, skrivare, skivminne, …).

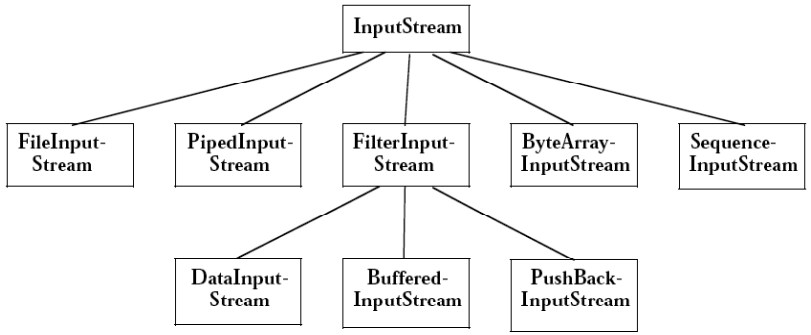
Strömmen skall tillåta programmet att:
InputStream : read() hämta en byte från filen, close() stäng strömmen (och därmed filen)
OutputStream: write(byte) skriv en byte, flush() tvinga ut data på filen, close()

I Java finns det ett stort antal strömmar. Var och en för ett visst ändamål.

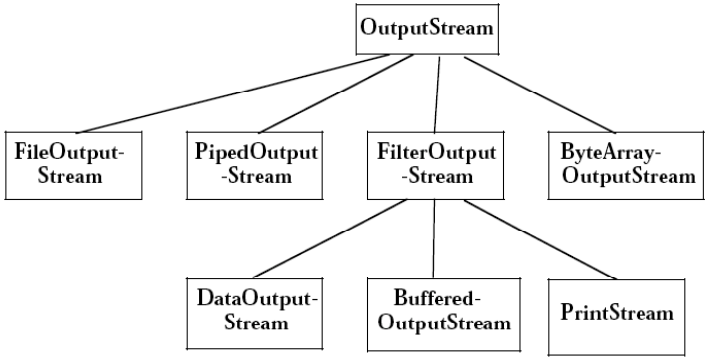
Filter

Anväds för att filtrera en ström för att låta viss typa av data att passera eller för att åstadkomma buffering, aggregering eller hålla reda på radnummer.
I Java finns det ett stort antal klasser för att hantera ut och inmatning.
Dessutom finns det två olika hierarkier för filer (och Strömmar)
en för att hantera byteorienterade filer och en för att hantera teckenorienterade filer (två bytes). Vissa klasser som betecknas som strömmar är egentligen både en ström och ett filter.
Då en ström "stängs" frigörs de systemresurser som varit låsta till strömmen.

Klasser för att hantera byteorienterade filer
Infiler



Utfiler



Byteorienterade strömmar

Ovanstående klasser hanterar bytefiler.

FileInputStream läster data från en fil

FileOutputStream skriver data på en fil

Piped in/out arbetar via en kanal (pipeline) mot en motsvarande fil (out/in)

ByteArray (in/out) skriver/läser en vektor med byte

Filter innebär att restriktioner kan avgöra vad som skall läsas/skrivas

Data (input/output) hanterar fördefinierade primitiva datatyper.

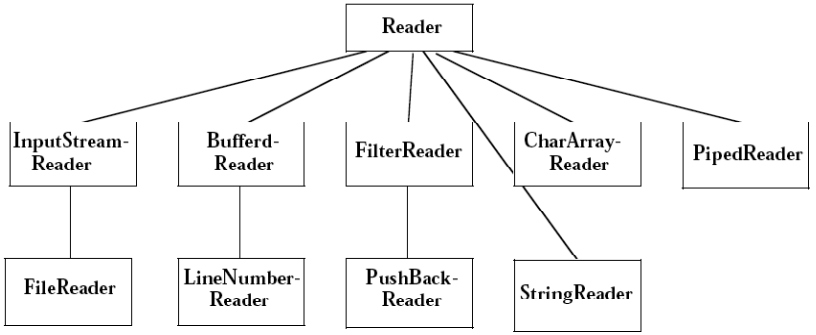
Bufferd (input/output) innebär att man arbetar med en buffert som sedan skrivs/läses till/från filen.

Pushback innebär att man kan göra en läsning ogjord.

Sequence innebär att flera indataströmmar slås samman.

PrintStream utström men som gör det lättare att skriva text.

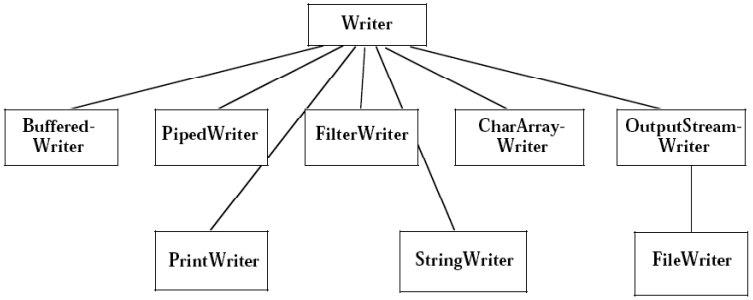
Klasser för att hantera teckenorienterade filer
Två byte används



Övriga klasser som behövs för att hantera filer:

File Klass som representerar en fil i filsystemet. Öppnas med filnamnet.
FileDescriptor handtag till filen

Utfiler



LineNumberReader läser radvis. Håller reda på radnumret.
InputStreamReader initieras med en InputStream och kan konvertera mellan de två typerna av strämmar.
OutputStreamWriter konverterar en teckenström till en byteström
StreamTokenizer delar upp strömmen med avseende på blank information (\n, \r, \t, " ").
Kan initieras med InputStream eller Reader

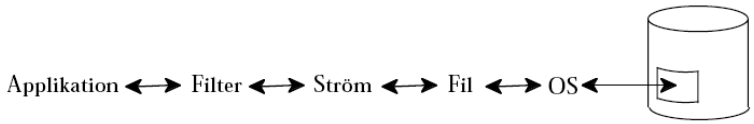
Användning av strömmar

Algoritmerna i klasserna genererar avbrott av typen IOException då man:
- **försöker läsa efter endOfFile uppnåtts**
- **Försök att skriva på en fil som inte är tillgänglig**
- **…**

För att kunna använda en fil måste man:
1. **Skapa ett filobjekt (filen öppnas automatiskt)**
2. **koppla en ström till filobjektet**
3. **Efteråt stänga filen**

Exempel:

```
File out;                  //deklarera en filreferens;
out = new File("fil.ext");//Skapa ett nytt filobjekt som kopplas ihop med en
                           //extern fil med namnet fil.ext;
FileWriter outStream = new FileWriter(out);
for (int data = 'a'; data < 'c'; data++)
outStream.write(data);
outStream.close()          //Stäng filen;
```



Användning av strömmar forts

Eller för en infil:

```
File in;                // deklarera en filreferens;
in = new File("fil.ext");
FileReader inStream = new FileReader(in);
int data;
data = inStream.read();
while (data != -1) {
    process(data);
    data = inStream.read();
}
inStream.close();
```

De fel som kan inträffa måste fångas:

```
File out;               // deklarera en filreferend;
try {
    out = new File("fil.ext");
    FileWriter outStream = new FileWriter(out);
    for (int data = 'a'; data < 'z'; data++){
        outStream.write(data);
        System.out.println(data);
    }
    outStream.close();  //Stäng filen;
}
catch (IOException e) {}
```


Användning av strömmar forts

Program för att kopiera en fil till en annan.
Filnamnen anges som argument till programmet då det startas:

```
import java.io.*;
class copy {
    public static void main(String[] args) {
        try {
            File inFile = new File(args[0]);
            File outFile = new File(args[1]);
            FileReader in = new FileReader(inFile);
            FileWriter out = new FileWriter(outFile);
            int data;
            data = in.read();
            while (data != -1) {
                out.write(data);
                data = in.read();
            }
            in.close();
            out.close();
        }
        catch (FileNotFoundException e) {System.out.println("File not found")}
        catch (IOException e) {}
        catch (ArrayIndexOutOfBoundsException e) {"usage: srcfile destfile"}
    }
}
```


Serialisering

Ett objekt kan spara sitt inre tillstånd på en fil då programmet terminerar och återladdas då programmet startar:

```
public void store() throws IOException {
  DataOutputStream out = new DataOutputStream(new
  FileOutputStream(getClass().getName() + objId));
  out.writeInt(objId);
  out.writeString(…
  …
  out.close();
}
```

Och återladdningen:

```
public Object reStore(String className, int id) throws IOException {
    DataInputStream in = new DataInputStream(new FileInputStream(className + id));
    try {
        Class c = Class.forName(className);
        Object res = c.newInstance();
        (persistentObject) res.read(in);
        return res;
    }
    catch (ClassNotFoundException e) {System.out.print(systemError));}
    catch (InstansiationException e) {System.out.print(systemError));}
    catch (IllegalAccessException e) {System.out.print(systemError));}
    catch (IOException e) {System.out.print(systemError));}
                                    }
```


Serialisering forts

Bättre:
Låt klassen, vars objekt skall lagras på en fil, implementera gränssnittet Serializable
Enkelt eftersom gränssnittet inte innehåller några metoder!

Hela objektet sparas med en enda skrivning till filen.
Filan är inte läsbar med en editor.
Filen är sekventiell och kan öppnas för läsning eller skrivning.

```
FileOutputStream ut = new FileOutputStream("FilNamn);
ObjectOutputStream objFile = new ObjectOutputStream(ut);
objFile.writeObject(obj);
objFile.close();
För att återskapa objekten:
FileInputStream in = new FileInputStream("FilNamn);
ObjectInputStream objFile = new ObjectInputStream(in);
Object obj = objFile.readObject();
objFile.close();
```

Objektets instansvariabler lagras, inte klassvariabler.

Information om klasstillhörighet lagras.

Object som refereras lagras första gången referensen påträffas.

Uppdatering av sekventiella filer

Enda sättet är att överföra innehållet till en ny fil och i samband med det uppdatera de värden som skall ändras, ta bort det som inte längre skall finnas på filen och lägga till nya poster.

Direktfiler

Ett objekt av klassen RandomAccessFile knyts till en fil som kan öppnas för både läsning och skrivning.

Vi kan flytta oss framåt och bakåt i filen genom att positionera oss med ett antal byte framåmt och bakåt.

Klassen RandomAccessFile implementerar ett gränssnitt för binär dataöverföring.

Konstruktor:
```
RandomAccessFile(String fileName, String mode)
```
där mode är "r" för read och "rw" för read/write
Metoder som alla kastar undantag
```
long getFilePointer(), void seek(long pos), long length(),
void close()
char readChar(), int readInt(), double readDouble()
void writeChar(int ch), void writeInt(int i), void writeDouble(double d)
```


StreamTokenizer

Klassen utför lexikal analys av inströmmen och bryter upp denna i lexikala enheter.

```
public int ttype
```
Talar om vad som lästs.

Om vi läst ett enstaka tecken så är det detta tecken (som en integer) annars ett av följande värden

TT_WORD indicates that the token is a word.
TT_NUMBER indicates that the token is a number.
TT_EOL indicates that the end of line has been read.
TT_EOF indicates that the end of the input stream has been reached.

Ovanstående är statiska konstanter.

```
public String sval
```
Om `ttype = TT_WORD` innehåller `sval` strängen.
Motsvarande om vi har ett numeriskt värde:
```
public double nval
```


StreamTokenizer, metoder

```
public void parseNumbers()
```
Specifierar att numeriska värden skall parsas. Syntaxtabellen omfattar
0 1 2 3 4 5 6 7 8 9 . –

```
public void eolIsSignificant(boolean flag)
```
Bestämmer om eol skall behandlas som tokens eller ej.

```
public void lowerCaseMode(boolean fl)

public int nextToken() throws IOException

public void pushBack()

public int lineno()
```
public String toString()

Ex:
```
import java.io.*;
public class Demonstrate {
    FileInputStream inputFile = new FileInputStream("input.data");
    StreamTokenizer tokens = new StreamTokenizer(inputFile);
    while (tokens.nextToken() != tokens.TT_EOF)
        System.out.println("Integer: " + (int) tokens.nval);
}
```


Exempel

```
import java.io.*;
public class Demonstrate {
    public static void main(String argv[]) throws IOException {
        FileInputStream inputFile = new FileInputStream("input.data");
        StreamTokenizer tokens = new StreamTokenizer(inputFile);
        int next = 0;
        while((next = tokens.nextToken()) != tokens.TT_EOF) {
            if (next == tokens.TT_WORD)
            System.out.print("Sträng: " + tokens.sval);
            else if (next == tokens.TT_NUMBER){
                System.out.print("Heltal: ");
                System.out.print((int)tokens.nval);
            }
        }
        inputFile.close();
    }
}
```

Direktfiler

Fungerar både som infil och utfil.

`file.seek(bytenr);` positionerar i filen.

Vi kan ta bort poster genom att blankställa dem.
Vi kan lägga till poster i slutet av filen.

File

Program som testar metoder i File

```
public class FileTest {
    File name;
    public FileTest(String fileName) { name = new File(fileName);
        if ( name.exists() ) {
            System.out.println (
            name.getName() + " exists\n" +
            (name.isFile() ? "is a file\n" : "is not a file\n" ) +
            (name.isDirectory() ? "is a directory\n" : "is not a directory\n" ) +
            (name.isAbsolute() ? "is absolute path\n": "is not absolute path\n" )+
            "Last modified: " + name.lastModified() + "\nLength: " + name.length()+
            "\nPath:" + name.getPath()+"\nAbsolute path: " +
            name.getAbsolutePath() +
            "\nParent: " + name.getParent() );
            if ( name.isFile() ) {
                try {
                    RandomAccessFile r =
                    new RandomAccessFile( name, "r" );
                    StringBuffer buf = new StringBuffer();
                    String text;
                    System.out.println ();
                    while( ( text = r.readLine() ) != null )
                    buf.append( r.readLine() + "\n" );
                    System.out.println ( buf.toString() );
                }
                catch( IOException e) {
                }
```


File, forts

```
            } else if ( name.isDirectory() ) {
                String directory[] = name.list();
                System.out.println ( "\n\nDirectory contents:\n");
                for ( int i = 0; i < directory.length; i++ )
                System.out.println ( directory[ i ] + "\n" );
            }
        }
        else {
            System.out.println ( e.getActionCommand() +
            " does not exist\n" );
        }
    }
    public static void main( String args[] ) {
        FileTest f = new FileTest(args[0]);
    }
}
```

Man kan inte läsa instanser av de enkla datatyperna direkt utan måste använda någon av klasserna StreamTokenizer eller StringTokenizer som delar upp strömmen eller strängen i instanser av strängar.
Java är inte speciellt bra på denna typ av I/O.
Förklaringen är att moderna program använder grafiska användargränssnitt och behöver därför inte ha tillgång till ett fullständigt bibliotek för textmässig in- och utmatning.
Det är dessutom så att program som inte får krascha inte kan läsa in tal på annat sätt än genom text och sedan hämta talet genom egen parsing.