Introduktion till objektorientering, grundläggande Java

F22

Vad är objekt-orientering?

- Allt är objekt
- Inkapsling (tillstånd och beteende)
- Polymorfism
- Dynamisk bindning
- I någon bemärkelse även arv (men vi väntar med det)

Java:

- Statiskt typat, klass-baserat OO-språk
- Automatisk minneshantering
- Inte rent objekt-orienterat: primitiva datatyper
- Enkelt implementationsarv
- Multipelt gränssnittsarv

Objekt kontra struct

- En strukt är en “död” samling data
- Alla operationer på datat definieras externt i procedurer och funktioner – *man gör saker med datat*
- Betrakta följande C-strukt – varför kan man säga att den är ”passiv”?

```
struct person {
  char* firstName;
  char middleInitial;
  char* lastName;
  char* ssn;
  int age;
};
```

Objekt kontra struct (forts.)

- Hur kan man se till att `age` alltid är ett vettigt tal?
- Hur kan man se till att `ssn` alltid följer 2-1-metoden?
- Hur tar man fram en persons hela namn?
- Hur hanteras aliasering?

Objekt kontra struct (forts.)

- Objekt är ”aktiva” – och tar ansvar för sitt eget data

```
class Person {
  private String firstName, middleInitial, lastName, ssn;
  private int age;

  String getFullName() { return firstName + " " +
                              middleInitial + " " + lastName; }

  void setAge(int age) {
    if (age >= 0 && age < 120) {
      this.age = age;
    } else {
      // do nothing for now, in future signal an error
    }
  }

  void setSSN(String ssn) { ... }
}
```

Ännu bättre

```
class Person {
  private String firstName, middleInitial, lastName;
  private Personnummer ssn;
  private int age;

  String getFullName() { return firstName + " " +
                              middleInitial + " " + lastName; }

  void setAge(int age) {
    if (age >= 0 && age < 120) {
      this.age = age;
    } else {
      // do nothing for now, in future signal an error
    }
  }

  void setSSN(Personnummer ssn) { ... }
}
```

Gör personunnummret intelligent” (OBS fulkod)

```
class Personnummer {
  private int[] numbers;
  Personnummer(int[] numbers) {
    int[] copy = new int[10];
    if (numbers.length != 10) ... // error, too few numbers
    int sum = 0;
    for (int i=0; i<9; ++i) {
      sum += (numbers[i]*(2-i%2) / 10 + numbers[i]*(2-i%2) % 10);
      copy[i] = numbers[i];
    }
    if (numbers[9] != 10 - (sum % 10)) ... // error, bad checksum
    copy[9] = numbers[9];
    this.numbers = copy;
  }

  String toString() {
    String result = "";
    for (int i=0; i<10; ++i) result += numbers[i];
    return result;
  }
}
```

Metoder kontra funktioner

- En metod opererar alltid på ett objekt (**`this`**)
- Objektet måste finnas för att man skall kunna anropa en metod på det
- Publika metoder, privata data – inkapsling
- Specialfall: konstruktorer
  - Kan bara anropas en gång, när objektet skapas
  - Svåra att få till korrekt, mer om det senare
  - Om det finns en konstruktor måste den anropas vid instansiering
  - På så sätt kan man se till att objekt alltid har korrekta värden (jmf. `Personunnummer`)
- Polymorfism: olika objekt kan ha olika implementationer för en metod med samma namn

Polymorfism

```
class Cowboy {
  void draw() { ... }
}

class Circle {
  void draw() { ... }
}

Cowboy c1 = new Cowboy();
Circle c2 = new Circle();

c1.draw();
c2.draw();
```

Polymorfism (forts.)

```
interface Drawable {
  void draw();
}

class Cowboy implements Drawable {
  void draw() { ... }
}

class Circle implements Drawable {
  void draw() { ... }
}

void someMethod(Drawable d) {
  d.draw();
}
```

Sammanfattning: OO

- Tankesättet kretsar kring objekt som inkapslar tillstånd och beteende
- Ett objekt slår vakt om sitt datas integritet
- Istället för göra något med datat (procedurellt) ber man datat att utföra någonting – vad som händer är upp till objektet (OO)
- Resultatet blir separation och abstraktion, vilket underlättar konstruktion och underhåll av system
- Objekt specificeras normalt genom klasser som beskriver alla objekt av en viss ”typ”
- Ett objekt *instantieras* genom att man ber klassen om att skapa en instans av sig själv
- Ovanstående är fullt möjligt även i C, bara inte lika enkelt

Vad är Java?

Man kan mena två olika saker:

- *Programmeringsspråket*
- *Plattformen*. Definierar en omgivning i vilken programmen exekverar.

När man laddar ner Java *software development kit* (SDK) så får man

- en kompilator (`javac`)
- en virtuell maskin (`java`)
- ett *klassbibliotek* eller *API* (application programming interface)

Karaktäristik av språket Java

- Objektorienterat
- Statiskt typat (som C)
- Syntaktiskt likt C (och C++)
- Väldefinierat
- Automatisk skräpsamling
- Säkert (kontroll av arraygränser, typer, odefinierade värden . . .)

Hello world i Java

```
public class HelloWorld {

    public static void main(String [] args) {
       System.out.println("Hello world!");
    }
}
```

Observationer:

- Funktionen (”metoden”) `main` är förpackad i en *klass*
- Ordet `public`
- `main` returnerar ingenting (typ `void`)
- `static` annan betydelse än i C
- `main` har *ett* argument som är en array av strängar (lite annan syntax)
- Utskrift på standard output med `System.out.println`

Vad kan vi ta med oss från C?

- De grundläggande datatyper: `int`, `float`, `double`, ... men med exakta definitioner av talområden och precision
- variabeldeklarationer
- operatorerna `+ - * / ++ - = += == < <=` ...
- Satser: **`if`**, **`for`**, **`while`**, **`do`**, **`switch`**
- Syntaxen för "funktioner" som här för det mesta kallas för *metoder*

Vad kan vi *inte* ta med?

- preprocessorn
- programstruktur med funktioner på filer
- deklarationsfiler
- pekare – motsvaras av *referenser* som är mycket mer begränsade
- `struct` – ersätts av *klass* som är ett *mycket* kraftfullare begrepp

Exempel

```
public class Factorial {
  public static long fact(int n) {
    long result = 1;
    for ( int i= 1; i<=n; i++ )
        result *= i;
    return result;
  }

  public static void main(String [] args) {
    for ( int i = 0; i < 10; i++ ) {
      System.out.println( i + "! = " + fact(i) );
    }
  }
}
```

- Två funktioner (”metoder”) i klassen
- Argumentet till `System.out.println` – strängkonkatenering

Primitiva datatyper

Typnamn	**datatyp**	**minnesutrymme**	**exempel**
`byte`	heltal	1 byte	-127, 47
`short`	heltal	2 byte	4711
`int`	heltal	4 byte	-748471
`long`	heltal	8 byte	434112345L
`float`	flyttal	4 byte	-4.57e10f
`double`	flyttal	8 byte	3.123e-128
`boolean`	logisk	1 byte	**true**, **false**
`char`	tecken (Unicode)	2 byte	'x', '4', '+', '\n'

Klassen `String`

Förutom de primitiva datatyperna kan programmen hantera *objekt*. Ett objekt tillhör alltid en viss klass.

Exempel: Den inbyggda klassen `String`

```
String s;
String t = "sträng";
System.out.println("Konkatenering av " + t + "ar");
s = "Denna strängs längd: ";
System.out.println(s + s.length());
s = "sträng";
if (s==t)
    System.out.println("Detta kommer INTE att skrivas");
if (s.equals(t))
    System.out.println("Detta kommer att skrivas");
```

Klassen `String` forts

Alltså:

- Typen `String` avser ett objekt av klassen `String`
- `String v` deklarerar en variabel `v` som antingen refererar till en sträng eller `null`
- Vid tilldelning till en `String`-variabel behöver vi inte tänka på att frigöra minnet för den gamla strängen
- `String`-värden kan *konkateneras* med additionsoperatorn (+), skapar ett nytt objekt
- Automatisk typkonvertering vid konkatenering
- Operationer ("metoder") (`length`, `equals`, ...) definierade för strängobjekt. *Punktnotation* (jfr -> i C).
- Relationsoperatorerna == och != testar *referenslikhet* – inte om objekten innehåller samma data (jfr string-pekare i C)

Utmatning i terminalfönstret

```
System.out.print(String s)
System.out.println(String s)
System.out.println()
```

Enbart radbyte

Från och med Java 5 finns en metod med namnet `printf`. Exempel:

```
System.out.printf("Längden av strängen '%s' är %d\n", s, s.l
```

Den inbyggda klassen `NumberFormat` kan användas för mer avancerade formateringar i enlighet med olika nationella konventioner.

Scanner-klassen

Kan användas för att läsa ord, tal mm (s.k. ”tokens”) från tangentbordet.

Koppla ett Scanner-objekt till inströmmen:

```
Scanner sc = new Scanner(System.in);
```

Några metoder:

`sc.hasNext()`	boolean
`sc.next()`	String
`sc.nextLine()`	String
`sc.hasNextInt()`	boolean
`sc.nextInt()`	int
`sc.hasNextDouble()`	boolean
`sc.nextDouble()`	double

Exempel: Tabell med funktionsvärden

```
// TableIO.java - Demonstrerar användning av Scanner
import java.util.Scanner; // <<< import

class TableIO {
  public static void main(String[] args) {
    double x, xlow, xhigh;
    int number;
    Scanner sc = new Scanner(System.in);
    System.out.print("Undre gräns: ");
    xlow = sc.nextDouble();
    System.out.print("Övre gräns: ");
    xhigh = sc.nextDouble();
    System.out.print("Antal värden: ");
    number = sc.nextInt();
    double step = (xhigh - xlow) / (number-1);
    for (int i = 1; i<=number; i++) {
        x = xlow + (i-1)*step;
        System.out.print(x);
        System.out.println("\t" + Math.log(x)); // <<< Math.log
    }
  }
}
```

Exempel: Tabell med funktionsvärden forts

Körresultat:

```
vega$ javac TableIO.java
vega$ java TableIO
Undre gräns: 0
Övre gräns: 10
Antal värden: 11
0.0 -Infinity
1.0 0.0
2.0 0.6931471805599453
3.0 1.0986122886681096
4.0 1.3862943611198906
5.0 1.6094379124341003
6.0 1.791759469228055
7.0 1.9459101490553132
8.0 2.0794415416798357
9.0 2.1972245773362196
10.0 2.302585092994046
```

Exempel: Tabell med funktionsvärden forts

Vad händer om man matar in felaktiga indata?

```
vega$ java TableIO
Undre gräns: 0
Övre gräns: 10
Antal värden: 12.3
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at TableIO.main(TableIO.java:16)
vega$
```

Vi skall titta mer på felhantering senare.

Klasser och objekt

Klasser används ofta för att beskriva någon typisk enhet i programmet. t ex något konkret fysiskt objekt (*bil*, *kund*, *kassa*, *kö*) eller något mer abstrakt begrepp (*teckenström*, *skärmfönster*) eller kanske något ännu mer abstrakt (*fyrdimensionellt klot*, *funktion* . . .)

En klass är alltså en abstrakt beskrivning av en viss typ av objekt.

Objekten karakteriseras av

- **egenskaper** eller **attribut** t ex *färg*, *form*, *bränslemängd*, *kölängd*, *expedieringstid*, . . . och
- **operationer** eller **metoder** dvs vad man kan göra med dem (*fylla på bensin* i ett bilobjekt, *ta ut första värdet* ur ett köobjekt, *ändra storleken* på ett fönsterobjekt . . .)

Exempel: En Kund-klass

```
public class Kund {
  private int ankomstTid;
  private int expTid;

  public Kund(int aTid, int eTid) { // En konstruktor
     ankomstTid = aTid;
     expTid = eTid;
  }

  public int getAnkomstTid() { // selektor
    return ankomstTid;
  }

  public void setExpedieringsTid(int expTid) { // mutator
    this.expTid = expTid;
  }
}
```

Observationer på Kund-klassen

- Två privata attribut
- En konstruktor som ger värden till attributen
- Två publika metoder som returnerar värdet på respektive attribut

Operatorn **new** används för att skapa objekt. Exempel:

```
Kund k1 = new Kund(10, 5);
Kund k2 = new Kund(11, 20);
System.out.println("Sammanlagd etid: " + (k1.getEtid() + k2.getEtid()));
```

Observera parenteserna i anropet till `println`!

Exempel: En tärningsklass

Antag att man vill representera en eller flera tärningar.

Vilka egenskaper (attribut) och vilka operationer (metoder) skall vi ge tärningar?

Om vi skall använda klassen för att simulera ”slå tärning(ar) och titta på resultatet” så kan vi ignorera flera av de egenskaper som verkliga tärningar har: färg, storlek, material . . .

Egenskaper vi behöver: *antal sidor* och *aktuellt värde*.

Operationer vi behöver: *skapa tärning*, *slå tärning* och *avläs värde*.

Exempel: En tärningsklass (forts.)

```
public class Die {
  private int numberOfSides;
  private int value;

  public Die() { // Konstruktor
     numberOfSides = 6;
  }

  public Die(int nS) { // Konstruktor
     numberOfSides = nS;
  }

  public int roll() { // Mutator
     return value = (int) (Math.random()*numberOfSides) + 1;
  }

  public int get() { return value; } // Selektor
}
```

(Klassen är inte perfekt men vi kan inte få allt på en gång . . .)

Observationer på Die-klassen

- Två konstruktorer – s.k. *överlagring*
- Klassen `Math` med metoden `random`
- Privata attribut, publika metoder
- Tilldelning har värde
- Inget `static`-deklarerat
- "Typecast" som t ex `(int)`

Die-klassen forts

Hur skapar man en tärning?

```
Die t1 = new Die(); // Tärning med 6 sidor
Die t2 = new Die(42); // Tärning med 42 sidor
Die t0 = null; // Variabel som (ännu) inte refererar en tärning
t1.roll(); // Slår den ena tärningen. (Behöver inte ta emot värdet)
t0.roll(); // Nonsens, kraschar under körning
```

Fråga: *Var* kan man göra detta?

Svar: I (och endast i) andra metoder.

Insikt: Jag måste alltså ha skapat ett objekt innan jag kan anropa dess metoder . . .

Förundrad fråga: Hur skapas då det första objektet?

Svar: Klasser är objekt under körning och skapas av Javas VM.

Die-klassen (forts.)

```
public class Die {
  private int numberOfSides;
  private int value;

  public Die() { numberOfSides = 6; }

  public Die(int nS) { numberOfSides = nS; }

  public int roll() {
    return value = (int) (Math.random()*numberOfSides) + 1;
  }

  public int get() { return value; }
}

public class Program {
  public static void main(String [] args) {
    Die t = new Die();
    for ( int i = 1; i<20; i++ )
        System.out.println(t.roll());
  }
}
```

Klasser, programstruktur och konventioner

- Ett program består av en eller flera *klasser* samlade i ett eller flera paket
- Varje klass lagras på en *fil* med *samma namn*
- Klassnamn *skall* börja på stor bokstav men attribut och metoder på liten.
- Attributen görs vanligen `private` medan metoderna oftast är `public` (default åtkomstmodifikator är `package`)
- En klass `main`-metod kan anropas vid programmets start och blir på så vis ett sätt en väg in i ett program (metoden måste vara deklarerad exakt som i exemplen ovan)

Exempel: Slå tärningar till par

Användning av tärningsklassen från en annan klass.

```
public class RollUntilEqual {

    public static void main(String [] args) {
        Die t1 = new Die();
        Die t2 = new Die();
        int n = 1;
        while (t1.roll()!=t2.roll())
            n++;
        System.out.println("Antal kast till par: " + n);
    }
}
```

```
bellatrix$ ls -l
-rw-r--r--+  1 tom      it          1009 Sep 23 20:47 Die.class
-rw-r--r--+  1 tom      it           510 Aug 19 14:00 Die.java
-rw-r--r--+  1 tom      it           251 Sep 23 21:18 RollUntilEqual.java
bellatrix$ javac RollUntilEqual.java
bellatrix$ ls -l
-rw-r--r--+  1 tom      it          1009 Sep 23 20:47 Die.class
-rw-r--r--+  1 tom      it           510 Aug 19 14:00 Die.java
-rw-r--r--+  1 tom      it           776 Sep 23 21:24 RollUntilEqual.class
-rw-r--r--+  1 tom      it           251 Sep 23 21:18 RollUntilEqual.java
bellatrix$ java RollUntilEqual
Antal kast till par: 13
bellatrix$ java RollUntilEqual
Antal kast till par: 1
bellatrix$ java RollUntilEqual
Antal kast till par: 4
bellatrix$ java RollUntilEqual
Antal kast till par: 4
bellatrix$
```

Exempel: array

```
import java.util.Scanner;

public class CheckDie {

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Number of sides: ");
    int nSides = sc.nextInt();
    int[] freq = new int[nSides];
    Die d = new Die(nSides);

    for (int i= 1; i<=1000; i++ ) {
      freq[d.roll()-1]++;
    }

    for (int i= 0; i<nSides; i++)
      System.out.println( (i+1) + "\t" + freq[i] );
  }

}
```

Observationer på `CheckDie`-klassen

- Flera huvudprogram (`main()`) - bara ett används
- Användning av en *array*
- Syntaxen i array-deklarationen (placeringen av `[]`)
- Arrayer skapas dynamiskt med **`new`**
- Arrayer hanteras med referenser
- Minsta index 0 i array

Övningar

1. Skriv ett tidigare program du skrivit i C i Java. Välj någonting enkelt, t.ex. en tidig övning från kursens C-del.
2. I samband med övningen ovan, fundera över skillnaderna mellan C och Java. Syntaxen är ofta snarlik, men är semantiken det också?
3. Lämpligen i samband med den första övningen, jämför Java-kompilatorns felmeddelanden med C-kompilatorns. Vilka är skillnaderna? Vilken föredrar du?

Åtkomstmodifikatorer, instantiering, referenser, identitet och ekvivalens, samt klassvariabler

F23

Inkapsling – tumregler

1. Man skall inte behöva veta detaljer i implementationen av en klass för att kunna använda den.
2. En klass bör kunna garantera att dess status (dvs dess värden på attributen) är konsistenta. För att detta skall vara möjligt så måste alla ändringar av status kontrolleras.

För att uppnå detta måste man förhindra okontrollerad access till attributen utifrån klassen själv.

Det anses vara en god stil att göra *alla* attribut privata och använda *selektorer* (“get-metoder”) för att avläsa dem och *mutatorer* (“set-metoder”) för att sätta dem men

- ingen regel utan undantag och
- man måste se till att mutatorerna sätter rimliga värden.

`public` och `private`

- En metod som är **`public`** får anropas från alla metoder i alla klasser.
- Ett attribut som är **`public`** får avläsas och ändras från alla metoder i alla andra klasser.
- En metod som är **`private`** får bara anropas av metoder i den egna klassen.
- Ett attribut som är **`private`** får bara avläsas och ändras från metoder i den egna klassen.

Obs: **`private`** *inte* skyddar mot access från andra objekt i samma klass.

Anm: Det finns två skyddnivåer till: **protected** och *package* (som inte har något nyckelord) – mer om dessa senare.

Skapande av objekt

Objekt skapas från klassen med operatorn **`new`**.

När objektet skapas sker följande:

1. Alla attribut får *defaultvärden* (0, 0.0, '\0', `false`, `null`)
2. Eventuella tilldelningar i deklarationen av attributen utförs i deklarationsordning
3. En konstruktor körs

Om det *inte* finns någon konstruktor i en klass tillhandahåller java en *parameterlös default-konstruktor*.

Om det finns en eller flera konstruktorer väljes den som matchar argumenten i antal och typ. I detta fall tillhandahåller *inte* systemet någon parameterlös konstruktor.

Exempel: klassen Point

```
// Punkt i planet
public class Point {
    private double x;
    private double y;

    public Point(double x) { // Konstruktor
        this.x = x;
    }

    public Point(double x, double y) { // Konstruktor
        this.x = x;
        this.y = y;
    }

    public double getx() { return x; } // Selektor

    public double gety() { return y; } // Selektor
}
```

Frågor om klassen `Point`

- Vad händer om vi skriver `Point p = new Point(1.0)`?
- Vad händer om vi skriver `Point p = new Point()`?
- Skriv en mutator `void moveTo(double x, double y)` som flyttar punkten till en ny position. Vad är konsekvensen av att göra detta tillägg?
- Vad skulle vi förlora på att göra attributen publika?
- Hur skulle vi göra om vi vill ha polära koordinater?

Exempel: klassen `Circle`

```
public class Circle {
   private Point center;
   private double radius;

   public Circle(Point center, double radius) {
      this.center = center;
      this.radius = radius;
   }

   public Circle(double radius) {
      center = new Point(0.0, 0.0);
      this.radius = radius;
   }

   public void scale(double sf) { radius *= sf; }

   public void moveTo(double x, double y) { center.moveTo(x,y); }

   // Samt diverse andra metoder ...
}
```

Synpunkter på klassen `Circle`

- Kontrollerar inte att det är vettiga värden på attributen `radius`
- Vad händer vid följande kod?

```
Point p = new Point(0.5, -3.5);
Circle c1 = new Circle(p, 1.0);
Circle c2 = new Circle(p, 2.0);
c1.moveTo(-1.0, -2.0);
```

Gör så att `Point` äger sitt eget data

Skriv om konstruktorn för `Circle`-klassen:

```
public Circle(Point center, double radius) {
   this.center = new Point(center.getx(), center.gety());
   this.radius = radius;
}
```

eller

```
public Circle(Point center, double radius) {
   this.center = center.copy();
   this.radius = radius;
}
```

med metoden `copy` i klassen `Point`;

```
public Point copy() { return new Point(x,y); }
```

Sammanfattning om referenser

- När ett objekt skapas med **`new`** returneras en *referens* (dvs ett *handtag* – logiskt sett motsvarande objektets *adress*) till det skapade objektet.
- En variabeldeklaration, t.ex.

  ```
  Die t;
  ```

  skapar inte en ny `Die`, utan bara en variabel `t` som kan hålla en *referens* till en `Die`
- Referenser kan tilldelas till referensvariabler (av rätt typ) och jämföras med == och `!=`
- Instansvariabler av referenstyp initieras till `null`
- **`this`** är en referens till det ”egna” objektet

Sammanfattning om referenser forts

- Flera referenser kan peka till samma objekt (aliasering). T.ex. blir resultatet av tilldelningen

  ```
  t1 = t2;
  ```

  alltid att `t1` och `t2` är alias för samma objekt (el. `null`).

- Observera att relationsoperatorerna == och != jämför om två referenser avser samma objekt, inte om objekten "ser likadana ut"

- Ett objekt som ingen refererar till är skräp som städas undan automatiskt av *skräpsamlaren* (GC)

  ```
  Die d = new Die(12);
  d = null;
  ```

  I Java är minneshantering = att komma ihåg att sätta variabler till null

Sammanfattning om referenser forts

- Variabler kan hålla referenser eller primitiva datatyper – aldrig hela objekt
- Objekt skickas som argument till metoder med “referenssemantik” (pass by reference)
- Om ett argumentobjekt förändras av en metod är det synligt för alla so har en referens till metoden (sidoeffekter)
- Man kan säga att objekten existerar globalt och är åtkomliga överallt där man har en referens till dem
- En metod kan returnera referenser som returvärde

Jämförelser av objekt – igen

- Igen: Relationsoperatorerna `==` och `!=` jämför bara med avseende på identitet
- *Alla* objekt har en metod `equals()` som jämför objekt för *strukturell* likhet
- Varje klass bör definiera en egen `equals()`-metod
- Referenser kan inte jämföras storleksmässigt men man kan naturligtvis definiera egna metoder för att jämföra objekt

Exempel på tänkbara jämförelsemetoder i `Circle`

```
public class Circle {

  private Point center;
  private double radius;

  // ... en massa metoder

  public boolean equals(Circle c) {
    return Math.abs(radius-c.radius) < 1.e-10;
  }


  public boolean equals2(Circle c) {
      return (radius==c.radius) && center.equals(c.center);
  }
}
```

Exempel på jämförelsemetoder forts

```
public int compareTo(Circle c) {
  if (radius == c.radius)
    return 0;
  else if (radius < c.radius)
    return -1;
  else
    return 1;
}
```

```
public boolean lessThan(Circle c) {
  return radius < c.radius;
}
```

Klassvariabler och klassmetoder

- Hittills har alla dataattribut varit *instans*-variabler dvs varje objekt har sin egen upplaga av dessa
- Man kan också ha *klass*-variabler som ligger i klass-objektet och därigenom är gemensamma för alla objekt i klassen (Anges i Java med `static`)
- En klassvariabel kan referas på samma sätt som vanliga attribut men vanligen genom *klassnamn.variabelnamn*
- Det går också att ha *klass*-metoder som alltså kan användas frikopplat från objekten (ex `Math.sin()` och `main()`)
- Ett objekts metoder har automatisk åtkomst till dess klass' metoder och attribut, men inte det omvända

```
public class Circle {
   private Point center;
   private double radius;
   private int id; // "cirkelnummer"
   private static int nCircles = 0;

   public Circle( Point center, double radius ) {
      this.center = center;
      this.radius = radius;
      id = ++nCircles;
   }

   public static void report() {
      System.out.println( "Antal skapade cirklar: " + nCircles );
   }

   public static void main(String [] args) {
      Circle c;
      for ( int i = 1; i<=10; i++ )
         c = new Circle( new Point( 10*i, 15*i ), 5*i );
      report();
   }
}
```

Övningar

1. Skriv en klass `Person` med en konstruktor som tar namn och personnummer som indata, och som håller reda på hur många personer som har instantierats sedan programmet startades. Det sistnämnda görs lämpligen med en *privat klassvariabel*. När skall den räknas upp? Hur kan man garantera att den alltid räknas upp? Skriv även en instansmetod (vanlig metod) **`int`** `getCount()` som returnerar klassvariabelns värde.
2. Skriv personklassen så att utomstående inte har direkt åtkomst till ett personobjekts namn och personnummer. Namn skall gå att byta, men inte personnummer.
3. Utöka personklassen ovan så att *klassen* person har en lista över samtliga personer som skapats i systemet. Modifiera `getCount()` till att returnera denna listas längd istället för att ha en räknare. Finns det några problem med denna typ av design? Vad får det för effekt på minneshantering?

Övningar (forts)

4. Utöka personklassen med en **`boolean`** `equals(Object)`-metod som returnerar **`true`** vid jämförelse av två personobjekt med samma personnummer, annars **`false`**.
5. Utöka personklassen så att det inte går att skapa två personer med samma personnummer. Med denna garanti i systemet blir implementationen av `equals`-metoden nu trivial. Vilken är den minsta möjliga implementationen av `equals` man behöver i personklassen och varför?

Sammanfattning, arrayer, inre och nästlade klasser, undantagshantering, wrapperklasser mm

F25

Sammanfattning

```
public class TheClassName {
  variable declaration (instances and classes)
  method declarations (instances and classes)
}
```

- Ingen speciell ordning krävs men ...
- Instansvariabler skall i regel inte vara åtkomliga av utomstående (**`private`**)
- Instansvariabler har *defaultvärden* beroende på typ (typiskt 0 eller `null` för referensvariabler)
- Instansvariabler kan initieras i deklarationen till godtyckliga uttryck (variabler i sådana uttryck måste vara definierade)
- Forward-referenser i initiering är inte tillåtna

```
public class InstvarOrder {
    int i = j;
    int j = 7;
}

dhcp-11-194:Temp tobias> javac InstvarOrder.java
InstvarOrder.java:2: illegal forward reference
    int i = j;
            ^
```

```
public class InstvarOrder {
    int j = 7;
    int i = j;
}

dhcp-11-194:Temp tobias> javac InstvarOrder.java
dhcp-11-194:Temp tobias>
```

Metodefinition

```
åtkomstmodifierare ReturTyp metodNamn(Typ1 param1, ..., TypN paramN) {
  lokala variabler, satser och eventuell värderetur
}
```

- Modifierarna anger t.ex. synlighet, klass/instansmetod
- Lokala variabler har inga defaultvärden utan måste tilldelas före användning
- I metoder som har en returtyp som inte är `void` måste alla möjliga vägar genom metoden sluta med `return` *exp* där *exp* är ett uttryck av den aktuella returtypen
- Metoder kan ha samma namn om de har olika *signatur* dvs skiljer sig i parameterantal och/eller typ (*överlagring*) – undvik detta i möjligaste mån, förutom för konstruktorer

Metodanrop

exp`.metodNamn(arg1, ..., argN);`

- där *exp* är ett uttryck som returnerar en referens (ett värde av referenstyp)
- Exempel:

```
this.setName(first + last)
(first + last).length()
first.trim().substring(0,7).length()
someMethod()
```

- I det sista fallet kommer `someMethod()` att *bindas* till antingen en instansmetod (=`this.someMethod()`) eller en klassmetod (=`MyClass.someMethod()`), beroende på vilken som existerar (båda är inte tillåtet i Java)
- Uttryck i parameterlistan evalueras från vänster till höger

Konstruktorer

- En konstruktor är en metod som körs automatiskt så fort minne allokerats för ett objekt, och instansvariabler tilldelats sina "startvärden"
- En konstruktor har alltid samma namn som klassen den tillhör och saknar returtyp
- Flera konstruktorer i en klass är vanligt för att kunna skapa objekt på flera sätt (t.ex. ström från fil, ström från sökväg)
- En klass utan konstruktor får en *defaultkonstruktor* automatiskt vid kompilering av Java-kompilatorn

Parameter och resultatöverföring

Java har *referenssemantik* för objekt och *värdesemantik* för primitiva datatyper.

- Referenssemantik = referensen till objektet kopieras, kopian pekar ut samma objekt som originalet; ger på så sätt upphov till aliasering
- Värdesemantik = värdet i variabeln kopieras i sin helhet
- Aliasering = två variabler som pekar ut samma objekt
- Minns att pekare inte finns! En variabel i en metod kan inte peka ut ett värde i en variabel i en annan metod. (Motsvarande beteende uppnås enklast med objekt.)

Arrayer

Arrayer är objekt med attribut och metoder. Eftersom en array är ett objekt hanteras den med referenssemantik vid parameteröverföring och returnering.

Deklaration:

typ`[] namn`

typ`[] namn =` *exp*

typ`[] namn =` $\{v_0, v_1, \ldots, v_n\}$

där v_i godtyckligt beräkningsbart uttryck av rätt typ.

Instantiering:

```
new RefTyp[47]; // skapar en array av 47 referenser
new int[4]; // skapar en array av 4 intar
new boolean[2] {true, false};
```

Exempel

```
public class Circles {
   private Circle[] circles;
   private int numberOfCircles = 0;

   public Circles() { circles = new Circle[5]; }

   public void add(Circle c) { circles[numberOfCircles++] = c; }

   public static void main(String[] args) {
      Circles circles = new Circles();
      Point p = new Point(0,0);
      for (int i = 1; i<=10; i++)
          circles.add(new Circle(p, 10*i));
  }
}
kursa$ java Circles
java.lang.ArrayIndexOutOfBoundsException: 5
       at Circles.add(Compiled Code)
       at Circles.main(Compiled Code)
kursa$
```

En bättre `add`-metod, och några observationer

```
public class Circles {
   private Circle [] circles;
   private int numberOfCircles;

   public void add(Circle c) {
      if (circles.length == numberOfCircles) {
         Circle[] _ = new Circle[2*circles.length];
         System.arraycopy(circles, 0, _, 0, circles.length);
         circles = _;
      }
      circles[numberOfCircles++] = c;
   }
   ...
```

- *Attributet* `length`
- Metoden `System.arraycopy`
- Grund kopiering (shallow copy)

`final`-specifikation

Klasser, metoder och variabler kan deklareras som **`final`**. Här talar vi bara om final-variabler.

En variabel som är **`final`** måste tilldelas sitt värde vid initiering eller konstruktorn; detta värde kan sedan inte förändras

```
public class Point {
   public final double x;
   public final double y;

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

   public Point() {
      this(0.0, 0.0); // Anropar andra konstruktorn
   }
}
```

Typkonverteringar

- Java konverterar automatiskt när det kan ske “riskfritt”
- Explicit med s.k. *typecasts*. Ex: (`short`)
- Vid aritmetik omvandlas `byte`, `short` till `int`
- Om en operand är `long` så omvandlas den andra till `long` (om den är av heltalstyp)
- Funktioner i Math returnerar `double` för att undvika precisionsförlust

Omslagsklasser

- Primitiva datatyper kan hanteras effektivt
- Ibland vill man dock behandla dem som objekt – då finns en motsvarande klass för varje primitiv datatyp (t.ex. `Integer`)
- Klasserna har oftast samma namn som de primitiva typerna men följer namnprinciperna för klasser. Wrapper-klasserna för **`int`** och **`char`** heter dock `Integer` och `Character`.
- Dessa klasser innehåller
  - Ett antal *klass*-attribut med konstanter (t.ex. max och minvärde)
  - Ett antal *klass*-metoder för t.ex. konverteringar
  - Ett antal instansmetoder motsvarande för ett objekt

Exempel: Räkna olika typer av tecken

```
import java.io.*;

public class Tecken {
   public static void main(String[] args ) throws IOException {
      int nUpper=0, nLower=0, nOthers=0;
      char c;
      BufferedReader inf =
          new BufferedReader(new InputStreamReader(System.in));
      System.out.print("> ");

      while((c=(char)inf.read()) != '\n') {
         if (Character.isUpperCase(c))
            nUpper++;
         else if (Character.isLowerCase(c))
            nLower++;
         else
            nOthers++;
      }
      System.out.println("U: " + nUpper + ", L: " + nLower + ", O: " + nOthers );
  }
}
```

Observerationer på klassen `Tecken`

- `Scanner`-klassen passar inte för att läsa tecken (ingen "`nextChar`")
- Använder en `BufferedReader` som via en `InputStreamReader` kopplas till `System.in`. Ett standardsätt.
- Import av `java.io.*`
- `code` returnerar en **`int`** i intervallet 0—65535 eller -1 vid filslut (strömslut)
- Raden **`throws`** `IOException` i början av `main`

Gå till javadokumentationen och titta vad som finns i klasserna omslagsklasserna! Lägg speciellt märke till metoderna `parseInt` och `valueOf` i `Integer` och motsvarande i `Double`.

Klassen String

- Ett objekt ur klassen `String` motsvarar en följd av tecken
- Strängkonstanter omges med citationstecken

```
String foo = "Omge strängar med \"-tecken";
```

- Objekt av typen String är *oföränderliga* (immutable). En strängoperation som har `String` som returtyp returnerar ett helt nytt strängobjekt.
- Operatorn + konkatenerar strängar
- Tecknen i strängen numreras från 0, liksom i en **`char`**-array

Några metoder i klassen `String`

```
int length()
int compareTo(String otherString)
boolean equals(Object otherObject)
char charAt(int index)
String substring(int start)
String substring(int start, int end)
int indexOf(char char)
int indexOf(String string)
String replace(char oldChar, char newChar)
String toUpperCase()
String toLowerCase()
```

Läs själva i Java-dokumentationen!

Exempel: klassen `Palindrom`

```
// Läser en följd av "ord" och avgör vilka av dessa som är palindrom
import java.util.Scanner;

public class Palindrom {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (sc.hasNext()) {
      String w = sc.next().toUpperCase();
      if (w.equals("STOP") || w.equals("QUIT"))
        break;
      final int n = w.length();
      boolean pal = true;
      for (int i=0; i < n/2 && pal; i++) {
        pal &= w.charAt(i)!=w.charAt(n-1-i);
      }
      System.out.println(w + "är" (pal ? "" : " inte") + " ett palindrom");
    }
  }
}
```

Arrayer av strängar

Parametern till `main` är ett exempel på en array av strängar. När programmet startas kommer orden på kommandoraden att lagras som element i denna array. Exempel:

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

Med körresultat:

```
kursa$ java Echo    hej du    glade!
hej du glade!
kursa$
```

Konvertering av objekt till String

Man kan definiera omvandling av objekt till typen `String` för t.ex. utskriftsändamål genom att definiera en metod `toString()`:

```
public class Circle {
   private Point center;
   private double radius;

   public String toString() {
      return "Circle(" + center + ", " + radius + ") ";
   }

   public static void main(String [] args) {
      Circle c = new Circle(new Point(0.0 , 0.0), 1.0);
      System.out.println(c);
   }
}
```

```
kursa$ java Circle
Circle(Point@1fa4d404, 1.0)
kursa$
```

Dynamiska strukturer (listor, träd . . .) i Java

```
// En Stack-mekanism som lagrar heltal
public class Stack {

    private StackNode top = null;

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

    public void push(int element) {
        top = new StackNode(element, top);
    }

    public int pop(){
        int _ = top.data;
        top = top.next;
        return _;
    }
}
```

Stackexempel forts

(Anm: Java har inbyggda klasser som kan hantera stackar, köer, etc.)

```
public class StackNode {
   private int data;
   private StackNode next;

   public StackNode(int data, StackNode node) {
      this.data = data;
      this.next = node;
   }
}
```

Men: hur skall `StackNode` se ut?

- Accessmetoder?
- Skall `StackNode` vara **`public`**?

Stackexempel forts, två (av flera) möjliga lösningar:

- Låt `Stack` och `StackNode` ingå i ett *paket* och ta bort **`public`** från definitionen av `StackNode` eller
- Gör `StackNode` till en *inre* klass till `Stack`

```
// fil Stack.java
package stack;

public class Stack { ... }
```

```
// fil StackNode.java
package stack;

class StackNode { ... }
```

Stackexempel forts, två (av flera) möjliga lösningar:

- Låt `Stack` och `StackNode` ingå i ett *paket* och ta bort **`public`** från definitionen av `StackNode` eller
- Gör `StackNode` till en *inre* klass till `Stack`

```
public class Stack {
  private class StackNode {
    int data;
    StackNode next;

    StackNode(int data, StackNode node) {
      this.data = data;
      this.next = node;
    }
  }

  private StackNode top = null;

  ...
```

Något om felhantering

Olika typer av fel: statisk/dynamisk och syntax/semantik

- Syntaxfel upptäcks av javac (motsv. stafvel)
- Semantiska statiska fel upptäcks av kompilatorn (typsystemet, dead code analysis, etc.)
- Semantiska dynamiska fel upptäcks
  - av den virtuella maskinen,
  - av den egna programkoden (felkontroll, defensiv programmering),
  - av testkod,
  - eller inte alls! – felaktiga resultat

Liksom många andra språk använder Java en generell mekanism för att hantera dynamiska, semantiska fel: *undantag* (eng. *exceptions*)

Undantag (*exceptions*)

- När programmet/javamaskinen upptäcker att något är fel så avbryts exekveringen och ett undantag genereras
- Ett undantag är ett objekt som beskriver felet och systemets tillstånd
- Därefter signalerar systemet att ett fel har inträffat – undantaget "kastas"
- Exekveringen fortsätter sedan i den del av programmet som förklarat sig beredd att hantera denna typ av undantag. Man säger att en *hanterare fångar* undantaget.
- Om ingen hanterare finns används en standardhanterare som avbryter programmet med en felutskrift.

Exempel på undantagshantering

```
try {
  operation som kan leda till undantag
} catch (ExceptionTypA e1) {
  kod som körs om ExceptionTypA kastades
} catch (ExceptionTypB e2) {
  kod som körs om ExceptionTypB kastades
}
```

Exempel på undantagshantering

```
try {
  x = 1/0;
} catch (ArithmeticException e) {
  System.err.println("Det blev ett fel: " + e.getMessage());
}
```

Exempel på att kasta ett undandag

```
 1  public class Die {
 2    int sides = -1;
 3    public Die(int sides) {
 4      if (sides < 2) {
 5        throw new IllegalArgumentException("A die must have 2+ sides");
 6      }
 7
 8      this.sides = sides;
 9    }
10  }
11
12  try { new Die(0); } catch (IllegalArgumentException e) { ... }
```

Under vilka omständigheter exekveras rad 7–8?

Undantagshantering forts

Som programmerare måste man i princip alltid tala om vad skall hända om ett fel inträffar. Tre möjligheter:

- Ta hand om felet ("fånga det"): `catch`
- Skicka det vidare: `throws`
- Kombination: fånga och skicka vidare
- Det finns undantag (!) från den obligatoriska felhanteringen – s.k. *okontrollerade* undantag.
- **Rekommendation:** använd alltid okontrollerade undantag (unchecked exceptions)
- Designproblem i Java: checked exceptions leder till dålig kod

Exempel på att kasta ett undandag vidare

```
1  public Die makeMeANewDie(int sides) {
2    try {
3      return new Die(sides);
4    } catch (IllegalArgumentException e) {
5      // Cannot recover from this error, so pass it on
6      throw e;
7    }
8  }
```

```
1  public Die makeMeANewDie(int sides) {
2    return new Die(sides);
3  }
```

Vad är skillnaden mellan kodexemplen?

Javas undantag är indelade i olika kategorier:

- `Error` allvarliga fel som kan inte hanteras av programmeraren
- `Exception`
  - `RuntimeException` (*okontrollerade*)
    - `IndexOutOfBoundsException`
    - `NegativeArraySizeException`
    - `NullPointerException`
    - `ArithmeticException`
    - `...`
  - `IOException`
    - `EOFException`
    - `FileNotFoundException`
    - `...`
  - `...`

Exempel: För många pop

```
public class StackTest {
  public static void main(String[] args) {
    Stack s = new Stack();
    s.push(1);
    s.push(2);
    s.push(3);
    while (!s.isEmpty()) {
      System.out.println(s.pop());
    }
    s.pop();
  }
}
```

```
bellatrix$ java StackTest
3
2
1
Exception in thread "main" java.lang.NullPointerException
        at Stack.pop(Stack.java:36)
        at Stack.main(Stack.java:48)
bellatrix$
```

Exempel: Fånga felet från `pop`

```
...
s.push(3);
while (!s.isEmpty()) {
  System.out.println(s.pop());
}
try {
  s.pop();
} catch (NullPointerException e) {
  System.err.println("Hoppla!");
}

bellatrix> java StackTest
3
2
1
Hoppla!
...
```

Skapa en egen felklass

Gör det tydligare vilket problem som avses:

```
public class StackUnderFlowException extends RuntimeException {
  public String getMessage() {
    return "The stack was pop'd when it was empty";
  }
}
```

Modifiering av pop:

```
public int pop() {
  if (top==null) {
    throw new StackUnderFlowException();
  }
  int r = top.data;
  top = top.next;
  return r;
}
```

Egen felklass forts

```
...
s.push(3);
while (!s.isEmpty()) {
  System.out.println(s.pop());
}
try {
  s.pop();
} catch (StackUnderFlowException e) {
  System.err.println("Hoppla!" + e.getMessage());
}
```

Med körresultat:

```
bellatrix$ java StackTest
3
2
1
Hoppla! The stack was pop'd when it was empty
bellatrix$
```

Egen felklass forts

Två möjliga placering av `StackException`:

- I en egen fil i samma katalog som `Stack` (och i samma *paket* som `Stack`)
- Som en publik *nästlad* klass i *klassen* `Stack`:

```
public class Stack {
  public static class StackUnderFlowException extends RuntimeException {
    ...
  }

  private static class StackNode {
    ...
  }

  ...
}
```

Skillnad mellan nästlad klass och inre klass

En inre klass är kopplad till det omslutande *objektet*:

```
Stack stack = new Stack();
StackNode stackNode = new stack.StackNode();

// Inside the Stack class, we can simply write new StackNode(), why?
```

En nästlad klass är kopplad till den omslutande *klassen*:

```
try {
  while (true)
    System.out.println(s.pop());
} catch (Stack.StackUnderFlowException e) {
  System.out.println("Hoppla! "+ e.getMessage());
}
```

Skillnad mellan nästlad klass och inre klass

Alla kontrollerade undantag som kan kastas i en metod måste antingen fångas eller så måste metoden ange att den kan skicka kontrollerade undantag (**`throws`**):

```
public String getPath(File someFile) throws IOException {
  return someFile.getCanonicalPath();
}
```

```
public String getPath(File someFile) {
  try {
    return someFile.getCanonicalPath();
  } catch (IOException e) {
    return null;
  }
}
```

Övningar

1. I övningarna till föreläsning 24 fanns en övning ”Skriv personklassen så att utomstående inte har direkt åtkomst till ett personobjekts namn och personnummer. Namn skall gå att byta, men inte personnummer.” Implementera det sistnämnda med **`final`**.
2. Skriv en metod som tar emot två argument av typerna **`int[]`** och **`double[]`** av samma längd och returnerar en array **`Object[]`** med vartannat elemenent **`Integer`** och vartannat **`Double`**.
3. Modifiera personklassen från tidigare övningar så att ett *egendefinierat* undantag kastas om det angivna personnumret inte är korrekt enligt Luhn-algoritmen[1]. Undantaget skall ärva från `IllegalArgumentException`, dvs. **`class`** `SomeName` **`extends`** `IllegalArgumentException ...`.
4. Ändra undantaget ovan så att det istället ärver `Exception`, dvs. **`class`** `SomeName` **`extends`** `Exception ...`. Vad händer vid omkompilering? Varför? Ändra programmet så att de kompilerar!
5. Skriv ett enkelt ”driverprogram” som skapar ett par personobjekt. Gör sedan undantaget ovan till en nästlad klass i personklassen, alternativt en inre klass. Hur påverkas driverprogrammet?

[1] Se `http://sv.wikipedia.org/wiki/Personnummer_i_Sverige`.

Koddokumentation med JavaDoc

F26

Javadoc: exemplifierat med `Stack`-klassen

```
/**
 * Mekanism för stackning av heltal
 *
 * @author Arthur Dent
 * @author Ford Prefect
 * @version 42.0
 */

public class Stack {

    /**
     * Beskriver undantag som kan kastas
     */
    public static class StackException extends Exception {
        public StackException(String msg) {
           super(msg);
        }
    }
```

Javadoc forts

```
/**
 * Intern klass för representation av stacknoder
 */
private static class StackNode {
    int data;
    StackNode next;

    StackNode(int d, StackNode n) {
        data = d;
        next = n;
    }
}

/**
 * Referens till översta noden på stacken
 */
private StackNode top;
```

Javadoc forts

```
/**
 * Konstruktor som skapar en tom stack
 */
public Stack(){ top = null; }

/**
 * Testar om stacken är tom
 * @return true om stacken är tom, annars false
 */
public boolean isEmpty(){
    return top==null;
}

/**
 * Lagrar ett heltal på stacken
 * @param n Tal av typ int som skall lagras på stacken
 */
public void push(int n) {
    top = new StackNode(n, top);
}
```

Javadoc forts

```
    /**
     * Hämtar och tar bort översta talet från stacken
     * @return Det poppade talet
     * @throws StackException Vid försök att poppa tom stack
     */
    public int pop() throws StackException {
        if (top==null)
            throw new StackException("Pop from empty stack");
        int r = top.data;
        top = top.next;
        return r;
    }
}
```

Taggar

`@param` *namn beskrivning*
`@return` *beskrivning av returvärdet*
`@throws` *undantagstyp beskrivning*
`@deprecated`
`@see` *paketnamn.klassnamn*
`@author` *Författare*
`@version` *versionsinformation*

Körning av javadoc

```
$ javadoc -d dokumentation Stack.java
$
```

Flaggorna `-author`, `-private` och `-version` används för att få med respektive information.

Övningar

1. Dokumentera personklassen och personnummerklassen du skrivit om du gjort tidigare övningar med JavaDoc, generera HTML-dokumentation, och titta på den i en webbläsare.
2. Experimentera med att slå på och av synlighet för privata medlemmar i klasser.
3. Se till att typer, både standardtyper som `String` och egendefinierade typer är korrekt länkade till i den genererade dokumentationen så att det går att klicka på dem och komma till rätt sida. Använd Google för ledning!

Parsing med Recursive Descent, Avbildningsklasser

F28

Betrakta följande uttryck

$$a + (b + c) \cdot d + e \cdot (f + g \cdot h)$$

Beräkning med regler: ”multiplikation före addition”, ”parenteser först” etc.

Vi kan emellertid definiera uttryck på följande sätt:

Ett uttryck
: är en sekvens av en eller flera *termer* med + eller − mellan

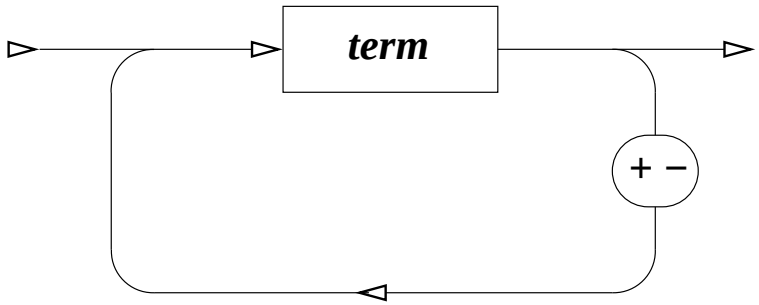
En term
: är en sekvens av en eller flera *faktorer* med · eller / mellan

En faktor
: är antingen ett tal eller ett *uttryck* omgivet av parenteser

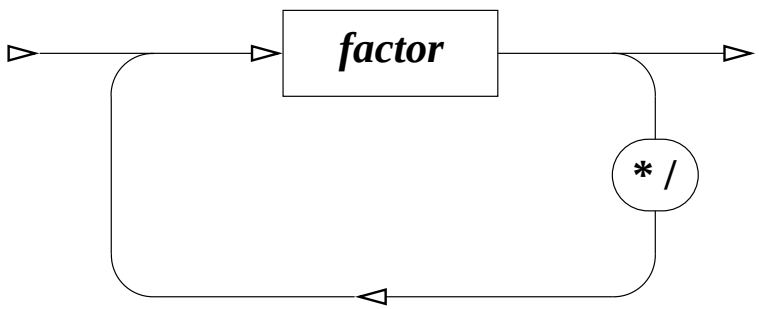
Detta kan uttryckas grafisk form i så kallade *syntaxdiagram*.

Syntaxdiagram

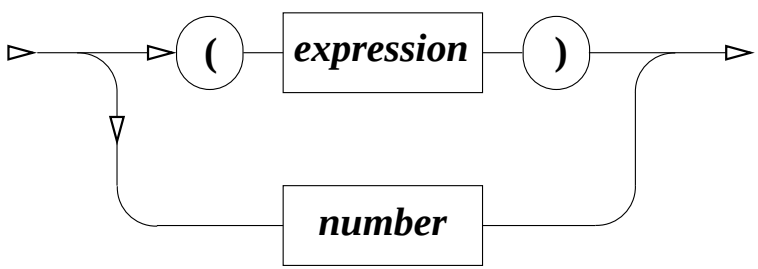
expression



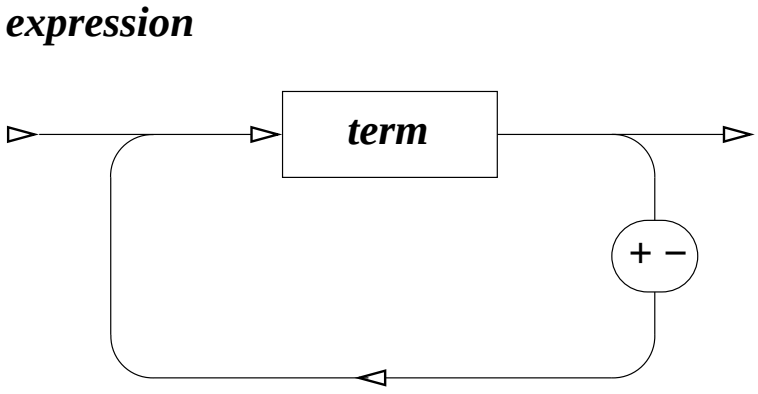
term



factor



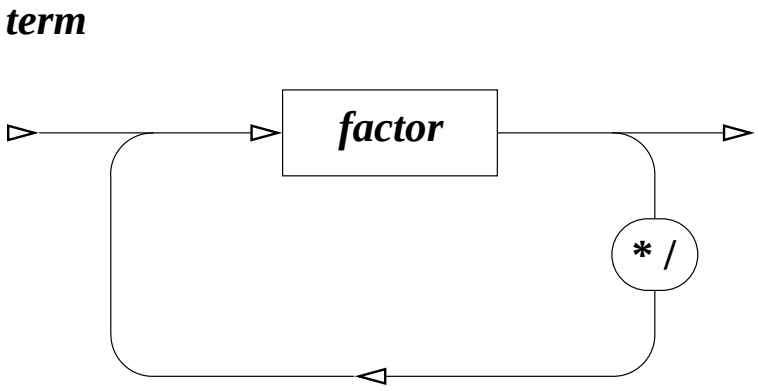
Syntaxdiagram och kodning




```
  public static double expression() {
    double sum = term();
    while (nextToRead() == '+')
    readNextChar();
    sum += term();
  }
  return sum;
}
```

(För enkelhetens skull behandlas bara addition.)

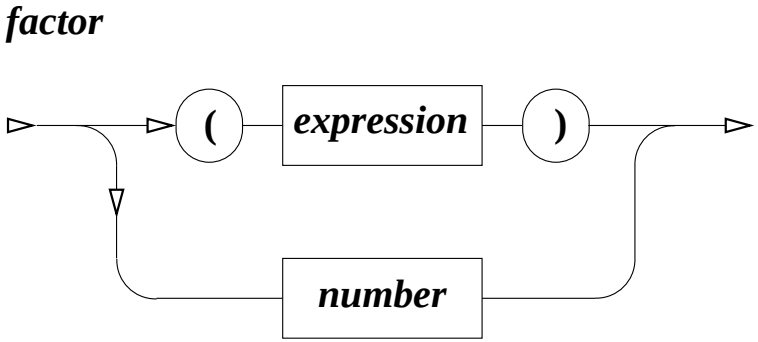
Syntaxdiagram och kodning




```
static double term() {
  double prod = faktor();
  while (nextToRead() == '*') {
    readNextChar();
    prod *= faktor();
  }
  return prod;
}
```

(För enkelhetens skull hanteras bara multiplikation)

Syntaxdiagram och kodning




```
static double factor() {
  if ( nextToRead() != '(' ) // skall vara tal
  return readNextNumber();
  else {
    readNextChar(); // läs förbi '('
    double result = expression();
    readNextChar(); // läs förbi ')'
    return result;
  }
}
```

Syntaxdiagram och kodning

Programmet bygger på tre primitiver:

- **char** `nextToRead()` som returnerar nästa tecken *utan* att ta bort det från input-strömmen,
- **char** `readNextChar()` som läser nästa tecken samt
- **double** `readNextNumber()` som läser nästa tal.

I Java kan dessa primitiver t ex uttryckas med hjälp av klassen `StreamTokenizer`.

StreamTokenizer

Skapas med

```
StreamTokenizer st = new StreamTokenizer(System.in)
```

Attribut

- **int** `ttype`: Om ett ”sammansatt token” lästs så är det en kod (`TT_WORD`, `TT_NUMBER`, `TT_EOL` . . .) annars det lästa tecknet
- `String sval`: Själva ordet om (`ttype==TT_WORD`)
- **double** `nval`: Själva talet om (`ttype==TT_NUMBER`)

StreamTokenizer forts

Metoder

- **int** `nextItem()`: Läser nästa item. Returnerar typ
- **void** `pushBack()`: Ser till att det senast lästa blir läst igen.
- **void** `eolIsSignificant(b)`: Om `b` är **true** kommer radslutstecknen (`TT_EOL`) att "synas".
- **void** `ordinaryChar(c)`: Anger att tecknet `c` inte skall kunna ingå i ord.

Exempel: `TreeMap` och `HashMap`

- Representerar en avbildning från en nyckelmängd av någon datatyp (t.ex. `String`) till en värdemängd av någon typ (t.ex. `String`, `Double` ...)

- Deklareras och skapas med

```
HashMap<__K__,__D__> map = new HashMap<__K__,__D__>()
```

  Där K är datatypen för nycklarna och D är datatypen för värdena. K och D måste vara typer (klasser hittills).

- För att lagra värden används metoden `D put(k,d)` där `k` och `d` är referenser till objekt av typen `K` respektive `D`. (`put` returnerar det gamla värdet eller **`null`** om det inte fanns något.)

- För att se om en viss nyckel finns lagrad kan man använda
  boolean `containsKey(`*k*`)`
- För att hämta värdet används
  `D get(`*k*`)`

`HashMap` prioriterar snabb access medan `TreeMap` underhåller en ordning (den föregåendes implementation bygger på arrayer och den senare på träd, därav namnen)

Referenser till objekt av typen `HashMap` och `TreeMap` kan alla deklareras som `Map` – vi kommer till varför litet senare

Demoprogram

```
import java.util.*;
import java.io.*;

/**
 * MapDemoMedUndantag.java
 * Kommandostyrt program som knyter Sträng-värden till identifierare.
 * Såväl identifierare ('ord') som tal som string-konstanter accepteras
 * som värden.
 * Fyra kommandon finns:
 * set id val - ger 'id' värdet 'val'
 * get id - skriver värdet för 'id'
 * dump - skriver lagrade värden
 * quit - avslutar programmet
 *
 * Syftet är demonstrera
 * 1) användning av Map
 * 2) användning av StreamTokenizer,
 * 3) felhantering med undantag och
 * 4) toString-metoder i Map och Tokenizer
 */

public class MapDemoMedUndantag {
```

Demoprogram

```
// Som vanligt
private static final InputStreamReader
                        isr = new InputStreamReader(System.in);
private static final BufferedReader
                        br = new BufferedReader(isr);
private static final StreamTokenizer
                        st = new StreamTokenizer(br);

private static Map<String, String>
                        vars = new HashMap<String,String>();
  /* eller
    vars = new TreeMap<String,String>();
  */
```

Demoprogram

```
// Som vanligt
private static final InputStreamReader
                        isr = new InputStreamReader(System.in);
private static final BufferedReader
                        br = new BufferedReader(isr);
private static final StreamTokenizer
                        st = new StreamTokenizer(br);

private static Map<String, String>
                        vars = new HashMap<String,String>();
  /* eller
    vars = new TreeMap<String,String>();
  */
```

```
public static void main(String[] arg) {
  st.quoteChar('\'');
  try {
     while (true) {
       try {
           System.out.print("Command: ");
           String s = readIdent();
           if (s.equals("quit")) break;
           else if (s.equals("dump")) System.out.println(vars);
           else if (s.equals("set")) performSet();
           else if (s.equals("get")) performGet();
           else throw new MyException("Unknown command: " + s);
       }
       catch (MyException me){
           System.out.println("*** " + me.getMessage());
           System.out.println("Line skipped");
           st.eolIsSignificant(true);
           while ( st.ttype!=st.TT_EOL ) st.nextToken();
           st.eolIsSignificant(false);
       }
     }
  } catch (IOException io) {
      System.err.println("IO-error. I quit.");
  }
}
```

Demoprogram forts

```
/**
 * Utför en läser identifierare och värde och lagrar detta par i tabellen
 * @throws IOException vid IO-fel
 */
public static void performSet()
    throws IOException {
    String s = readIdent();
    String d = readString();
    vars.put(s, d);
}
```

Demoprogram forts

```
/**
 * Läser en identifierare och hämtar dess värde ur tabellen
 * @return Läst identifierare
 * @throws IOException Vid IO-fel
 * @throws MyException Om inläst värde inte finns lagrat
 */
public static void performGet()
    throws IOException {
    String s = readIdent();
    String d = null;
    if ( vars.containsKey(s) )
        d = vars.get(s);
    else
        throw new MyException("Undefined: " + s);
    System.out.println(s + " = " + d);
}
```

Demoprogram forts

```
/**
 * Läser en identifierare.
 * @return Läst identifierare
 * @throws IOException Vid IO-fel
 * @throws MyException Om nästa element inte är en identifierare
 */
public static String readIdent()
    throws IOException {
    st.nextToken();
    if (st.ttype!=st.TT_WORD)
       throw new MyException("Expected identif. Found: " + st.toString());
    else
       return st.sval;
}
```

Demoprogram forts

```
  /**
   * Läser en identifierare eller en sträng innesluten av apostofer (').
   * @return Läst identifierare eller sträng
   * @throws IOException Vid IO-fel
   * @throws MyException Om nästa element inte är en identifierare \
   * eller sträng
   */
  public static String readString()
      throws IOException {
      st.nextToken();
      if (st.ttype==st.TT_WORD || st.ttype=='\'')
          return st.sval;
      else if (st.ttype==st.TT_NUMBER)
          return "" + st.nval;
      else
          throw new MyException("Expected identifier or string. Found: " +
                                st.toString());
  }
}
```

Demoprogram forts

```
/**
 * MyException.java
 *
 * En egendefinierad klass för fel som kan uppstå i koden.
 * Klassen skrivs som en underklass till den systemdefinerade klassen
 * RuntimeException vilket bl a innebär att den inte behöver
 * deklareras med 'throws'.
 *
 * Klassen råkar också vara placerad i samma fil som 'huvudklassen'.
 */
class MyException extends RuntimeException {
    public MyException() {
        super();
    }
    public MyException(String msg) {
        super(msg);
    }
}
```

Demoprogram - testkörning

```
algol$ java MapDemoMedUndantag
Command: set x hej
Command: set y hopp
Command: dump
{y=hopp, x=hej}
Command: set 12 z
*** Expected identifier. Found: Token[n=12.0], line 4
Line skipped
Command: hej hopp!
*** Unknown command: hej
Line skipped
Command: set Tom Slöjdgatan 10
Command: *** Expected identifier. Found: Token[n=10.0], line 6
Line skipped
Command: set Tom 'Slöjdgatan 10'
Command: dump
{Tom=Slöjdgatan 10, y=hopp, x=hej}
Command: get tom
*** Undefined: tom
Line skipped
Command: get Tom
Tom = Slöjdgatan 10
Command: quit
algol$
```

Paket

Klassbiblioteken i Java är organiserade i *paket*

- För att få tillgång till en klass i ett paket använder man **import**-satsen. Exempel:

```
import java.util.Scanner;
```

- Man kan importera alla klasserna i ett paket med "wild card"-notation:

```
import java.util.*;
```

- Man kan importera `static`-metoder explicit. Exempel:

```
import static Math.*;
```

  varefter man kan referera funktionerna i klassen utan `Math.`

Paket forts

Att skapa paket:

- Välj ett *paketnamn* (inleds med liten bokstav).
- Skapa en katalog med samman namn som paketet och placera filerna som skall ingå i paketet där.
- Varje fil skall inledas med en **package**-sats. Exempel:

```
package myQueuePackage;
```

Java letar efter klasser i de kataloger som specificeras av miljövariabeln `CLASSPATH`.

Om ingen **package**-sats hör filerna klasserna till ett *anonymt* paket.

Kom ihåg: *paket*-synlighet.

Övningar

1. Skriv ett program som läser radorienterat indata från standard in på formen `key:value` och stoppar in i en `Map<String,String>`. Du kan omdirigera standard in, precis som i C, till att läsa från en fil.
2. Ändra programmet så att alla `value`:s är heltal.
3. Ändra programmet så att insertering av en dublett medför borttagning, d.v.s., om nyckeln A pekar ut värdet 27 och ytterligare en uppdatering görs med nyckeln A och värdet 27, så skall avbildningen tas bort helt ur "mappen".
4. Javas standardbibliotek har flera olika mappar, bl.a. `HashMap` och `TreeMap`. Kan du se någon förändring i programmets exekveringstid beroende på vilken slags map som används? Testa med indatafiler med 10, 1000, 100 000 och 1 000 000 rader med 25% dubbletter för olika versioner av programmet som anväder olika typer av avbildningsklasser.
5. Lägg programmet i ett paket. Hur påverkar det kompilering och körning av programmet?

Arv och klasshierarkier

F29

Aggregering

Klasser kan byggas på redan definierade klasser genom att klassobjekt används som dataattribut när en ny klass beskrivs.

Exempel:

- En klass `PairOfDice` kan konstrueras med två attribut av typen `Die`.
- En klass `CardDeck` kan byggas med hjälp av ett arrayobjekt innehållande 52 objekt av typen `Card`
- En klass `ListMap` kan konstrueras med hjälp av `ListNode`-objekt
- En klass `TrafficSystem` kan sättas ihop med objekt ur klasser som `Lane`, `Light`, `Vehicle` ...

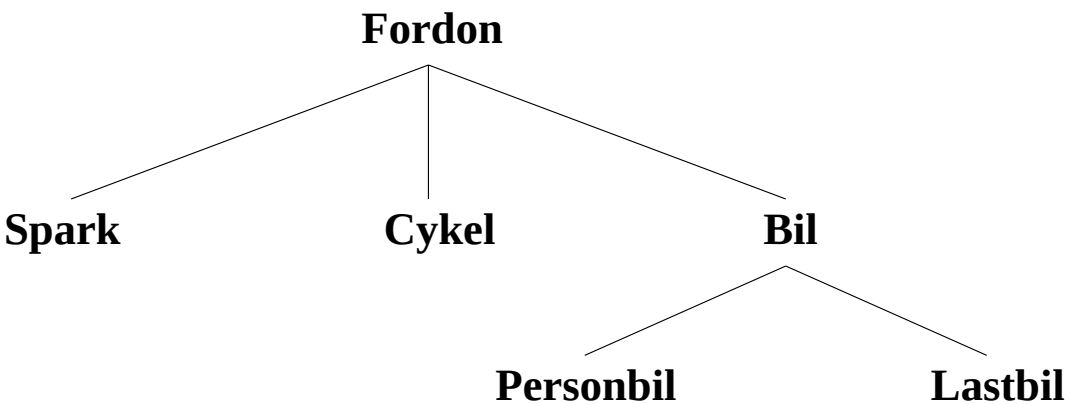
Man brukar säga att detta är en *består av*-relation (has-a)

Arv

En klass kan också byggas som en *specialisering* av en annan klass.

Exempel:

- Klassen `StackException` som används i `Stack`-klassen är skriven som en *subklass* till `Exception`
- Om vi har en klass `Fordon` så kan vi bygga nya klasser som t.ex. `Spark`, `Cykel` och `Bil`. En `Bil` kan i sin tur vara basklass för klasserna `Personbil` och `Lastbil`



Arv (forts)

I dessa fall säger man att man har en *klasshierarki*

- Klassen `Fordon` sägs vara en *basklass* (äv. superklass)
- Klasserna `Spark`, `Cykel` och `Bil` är *subklasser* eller *underklasser* till klassen `Fordon`
- En subklass (t.ex. klassen `Bil`) kan användas som *basklass* för andra klasser (t.ex. `Personbil` och `Lastbil`)

Basklass/superklass/subklass beskriver en klass *relation* till en annan klass(er). Det är *inte* en egenskap hos klassen.

Arv är en *är en*-relation (is a). Man kan pröva om arv är lämpligt genom att försöka sätta in subklassen *B* och superklassen *A* i frasen:

B är en A

ex. ”en Bil är ett Fordon”, ”en Människa är ett däggdjur”

Arv (forts)

Ett objekt ur en subklass

- "får" basklassens attribut och metoder och
- kan lägga till egna

Man säger att subklassen *ärver* basklassens egenskaper.

Exempel: Klassen `Bil` kan ha egenskapen `vikt`. En `Personbil` kan tillfoga `passagerare` medan en `Lastbil` kan tillfoga `maxlast`.

En subklass är en *specialisering* av sin superklass (jmf. Bil–Fordon)

En subklass specialiserar ofta sin superklass metoder (overriding)

Metodspecialisering (overriding)

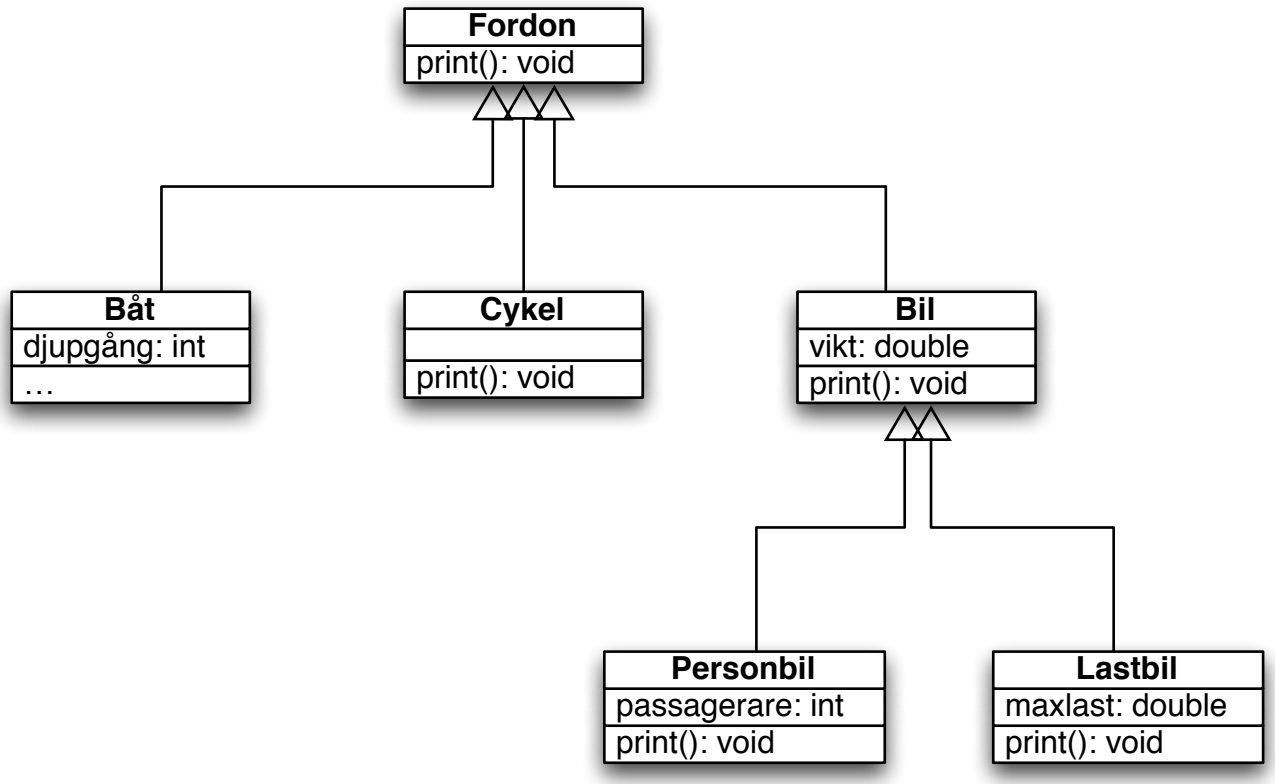
- Ponera en klass *A* och en klass *B* som är en subklass till *A*
- *A* definierar metoden `int getX()`
- *B* definierar också en metod `int getX()`
- Vi säger nu att *B*:s `getX()`-metod "override:ar" (är en specialisering av) *A*:s motsvarande metod
- En subklass kan explicit be om att anropa den *generella* versionen av en metod med nyckelordet **`super`**

Metodspecialisering (forts)

```
class A {
  int x = 27;
  int getX() { return x; }
}

class B extends A {
  // Ärver int x automagiskt
  int getX() { return super.getX() + 2; }
}

A a = new A();
a.getX(); // returnerar 27

a = new B();
a.getX(); // returnerar 29

// Vad skrivs ut nedan?
void someMethod(A a) {
  System.out.println("X-värde: " + a.getX());
}
```

Unified Modeling Language (UML)

Klasshierarkier kan ritas ut som *klassdiagram* i *UML*.



Fordonshierarkin i kod

```
public class Fordon {
   public void print() {
      System.out.print("Fordon");
   }
}

public class Spark extends Fordon {}

public class Cykel extends Fordon {
    public void print() {
       System.out.print("Cykel");
    }
}

public class Bil extends Fordon {
   double vikt;

   public Bil(double vikt) {
      this.vikt = vikt;
   }

   public void print() {
      System.out.print("Bil med vikt " + vikt);
   }
}
```

Användning

Koden

```
public static void pr(String s) {
   System.out.print(s);
}

public static void main(String [] args) {
   Fordon f = new Fordon();
   Spark s = new Spark();
   Cykel c = new Cykel();
   Bil b = new Bil(1.2);
   pr("f: "); f.print(); pr("\n");
   pr("s: "); s.print(); pr("\n");
   pr("c: "); c.print(); pr("\n");
   pr("b: "); b.print(); pr("\n");
```

ger utskriften

```
f: Fordon
s: Fordon
c: Cykel
b: Bil med vikt 1.2
```

Flera subklasser

```
public class Personbil extends Bil {
   int passagerare = 0;

   public Personbil(double vikt) {
      this(vikt, 5); // Anrop till den andra konstruktorn
   }

   public Personbil(double vikt, int passagerarAntal) {
      super(vikt); // Anrop till superklassens konstruktor gör så här
      passagerare = passagerarAntal;
   }

   public void print() {
      System.out.print("Person");
      super.print();
      System.out.print(" och platsantal " + passagerare);
   }
}
```

Fortsättning av `main`

Koden

```
...
Personbil pb1 = new Personbil(1.5);
Personbil pb2 = new Personbil(0.9,4);
pr("pb1: "); pb1.print(); pr("\n");
pr("Vikt b : " + b.vikt + "\n");
pr("Vikt pb1: " + pb1.vikt + "\n");
f = b;
pr("f: "); f.print(); pr("\n");
f = pb1;
pr("f: "); f.print(); pr("\n");
pb1 = (Personbil) f;
pr("pb1: "); pb1.print(); pr("\n");
```

ger resultatet

```
pb1: PersonBil med vikt 1.5 och platsantal 5
Vikt b: 1.2
Vikt pb1: 1.5
f: Bil med vikt 1.2
f: PersonBil med vikt 0.9 och platsantal 4
pb1: PersonBil med vikt 0.9 och platsantal 4
```

Typomvandling

Typomvandling (cast) kan göras och förändrar kompilatorns syn på ett värdes typ.

```
pb1 = f;
pb1 = (Personbil) c;
pr("Vikt: " + f.vikt);
pb1 = (Personbil) f; // om f ej refererar Personbil
```

Man kan fråga om ett värde har en viss typ med `instanceof`-operatorn.

```
if (f instanceof Personbil) {
   pb1 = (Personbil) f;
} else {
  ...
}
```

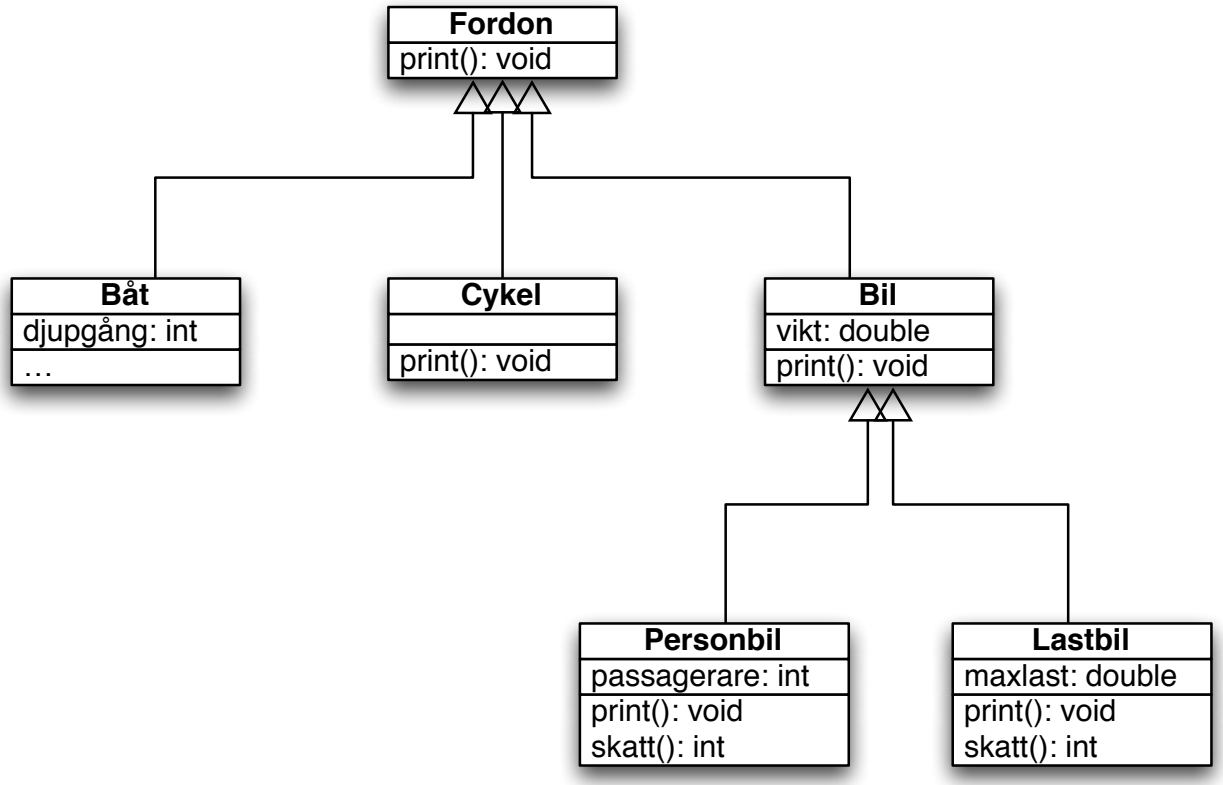
Operatorn `instanceof` bör sällan användas! (Varför?)

Sammanfattning

Om klassen `S` är en subklass till klassen `B` och om `s` är deklarerad som en referens till `S` och `b` till `B` så gäller

- Ett objekt ur klassen `S` har alla attribut och metoder som klassen `B` har
- Om `S` deklarerar ett attribut som finns i `B` *överskuggas* `B`:s attribut.
- En referens får referera objekt ur deklarerad klass *och dess subklasser*.
- För att uttrycket `b.x` (eller `b.x()`) skall vara tillåten måste attributet (eller metoden) `x` finnas deklarerad i `B` eller någon superklass till `B`.
- När man anger ett attribut `a` via punktnotation `p.a` så är det hur `p` är *deklarerad* som avgör vilket attribut som väljs.
- När man anropar en metod `m` via `p.m()` så är det typen på det element som `p` verkligen refererar som avgör vilken metod som väljs (så kallad *dynamisk* eller *sen* bindning).

Metoder som bara finns i vissa klasser



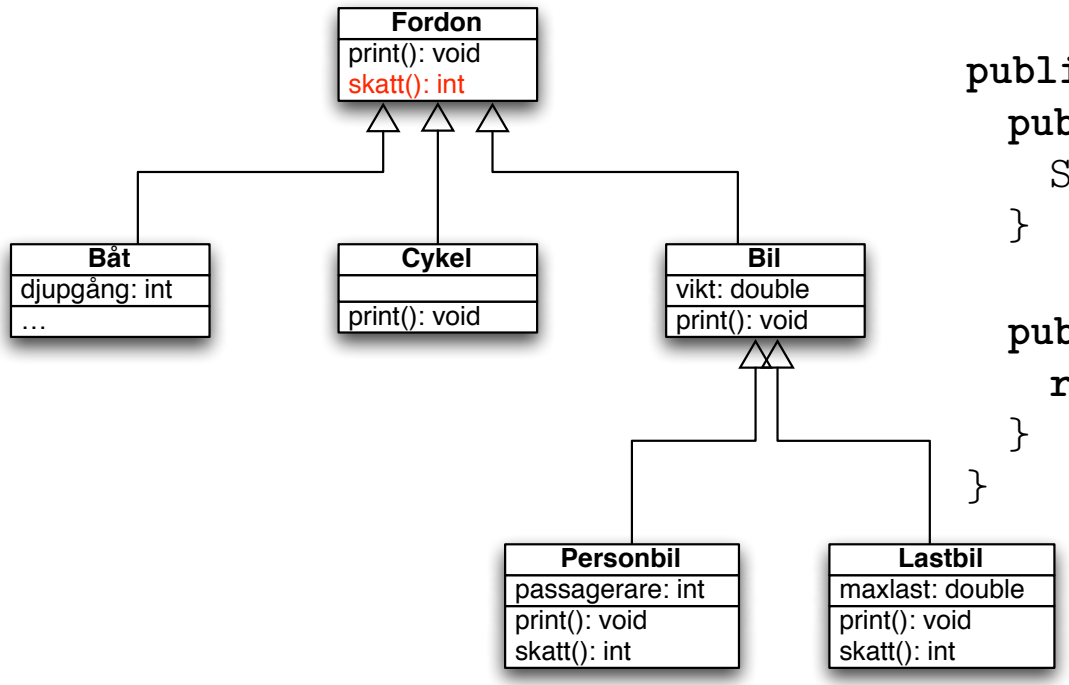
Bilar beskattas men beräkningen görs på olika sätt för person- och lastbilar.

Användning av `skatt`-metoden

```
Bil b = new Lastbil(3., 2.);
...
if (...) {
   b = new Personbil(1.2, 5);
}
...
double s;

s = b.skatt(); // Illegalt! Skatt inte finns i b

s = ((Lastbil) b).skatt(); // Riskabelt!

if (b instanceof Lastbil) // Fungerar men klumpigt och
     s = ((Lastbil) b).skatt(); // oflexibelt
else if (b instanceof Personbil)
     s = ((Personbil) b).skatt();
else
     s = 0;
```

Nyttan av att generalisera—`skatt()` åt alla

Definiera dessutom en `skatt`-metod i klassen `Bil` eller, troligen bättre, i klassen `Fordon`:




```
public class Fordon {
  public void print() {
    System.out.print("Fordon");
  }

  public double skatt() {
    return 0;
  }
}
```

Då kan metoden anropas för alla objekt av typen `Fordon` eller dess underklasser. (Test: är det vettigt att alla fordon kan beskattas?)

Generellt: Utnyttja den dynamiska bindningen istället för `instanceof`!

Åtkomstmodifikatorer

- `private` Endast åtkomligt från klassens egna metoder
- `public` Åtkomligt för alla
- `protected` Åtkomligt från egna subklasser och klasser i samma paket
- *package* (saknar nyckelord) Om ingen skyddsnivå anges så ges alla klasser i samma paket åtkomsträttighet

Som tumregel, använd alltid `private` – skydda superklassen från basklassernas beteende. Använd endast någon av de andra om det finns en vettig anledning (med minst ett bra exempel på användande).

Konstruktorer vid arv

Konstruktorer ärvs *inte* (varför?)

När ett objekt ur en subklass skapas så sker följande:

1. Instansvariablerna får sina defaultvärden
2. En konstruktor för superklassen anropas. Man använder `super` för att specificera vilken av basklassens konstruktor som skall användas. Om `super` inte används anropas basklassens parameterlösa konstruktor som då måste finnas (implicit eller explicit).
3. Eventuella initieringsuttryck evalueras och tilldelas respektive instansvariabler
4. Satserna i subklassens konstruktor exekveras

Använd `super` i stället för att upprepa kod! Notera att `super`-anropet måste stå först i konstruktorn. (Varför?)

Javas klassbibliotek

Hela Java-miljön bygger på arv och klasshierakier

Exempel:

1. Klasserna för undantag: `Throwable` med underklasserna `Error` och `Exception` med underklasserna ...
2. Grafiska komponenter: `Component` med underklasser `Button`, `Checkbox`, `Container`, ... där t.ex. `Container` har underklasserna `Panel`, `Window` ...
3. Avbildningar: `AbstractMap` med bl.a. underklassen `HashMap`
4. Samlingsklasserna: `Collection` med bl a underklassen och `List` som bl a har underklasserna `Vector` och `LinkedList`

(`Map`, `Collection` och `List` är egentligen *interface* och inte klasser)

Klassen `Object`

I Java är klassen `Object` är en s.k. rotklass – en superklass till alla klasser. En referens till `Object` får således referera vilket objekt som helst.

Kan utnyttjas för att göra generella ”samlingsklasser” (listor, tabeller . . .):

```
class ListNode {
   Object info;
   ListNode next;
   ListHead(Object i, ListNode n) {
      info = l;
      next = n;
   }
}
public class List {
   ListNode head;
   public void insertFirst(Object o) {
      head = new ListNode(o, head)
   }
      ...
}
```

Från och med Java 1.5 används dock oftast ”generics” i stället.

Klassen `Object` forts

Några metoder i klassen `Object`:

- `Object clone()`
- **`boolean`** `equals(Object o)`
- `String toString()`

Ofta finns det anledning att omdefiniera dessa i en subklass.

Exempel på användning av arv: *geometriska figurer*

Skriv ett program som kan hantera bilder bestående av cirklar, rektanglar och andra geometriska former.

Programmet ska kunna

- representera ett antal olika figurer,
- rita upp en bild av de representerade figurerna,
- beräkna den sammanlagda ytan av figurerna och
- kunna läsa in hur en bild ska vara uppbyggd.

(en rad andra operationer är naturligtvis tänkbara: flytta, rotera, skala . . .)

Forts *geometriska figurer*

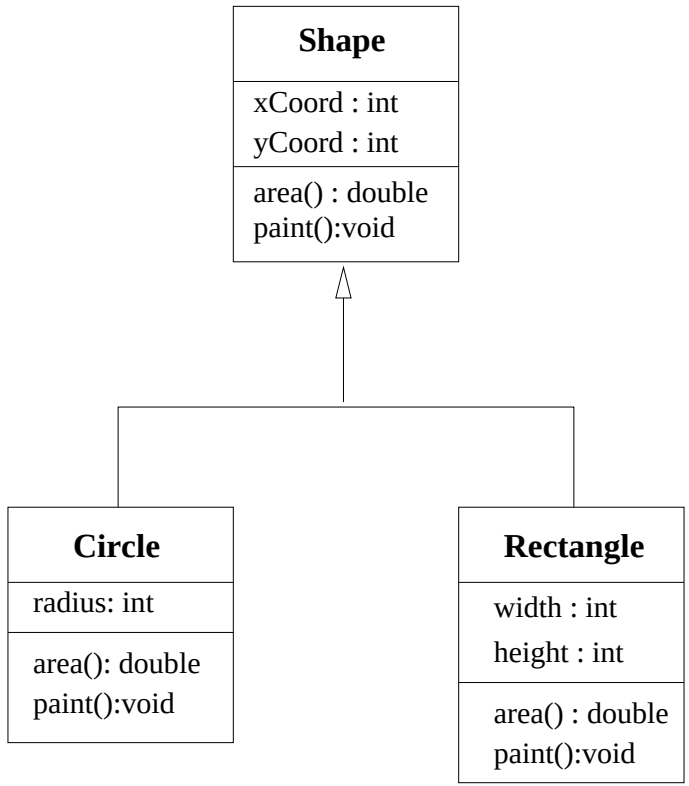
1. Vilka klasser behövs för att representera figurerna?
   Circle, *Rectangle*, ...

2. Vilka attribut och metoder ska finnas?
   - konstruktorer
   - metod för ytberäkning
   - metod för att rita

3. Hur skall man representera en mängd sådana figurer?
   - array?
   - lista?
   - annan struktur?

   Vilken typ skall det vara på elementen i strukturen?

Forts *geometriska figurer*



Forts *geometriska figurer*

Nu kan vi t ex göra en

```
Shape[] fig = new Shape[100];
fig[0] = new Recatangle(...);
fig[1] = new Circle(...);
...
```

(fast vi skall använda en mer flexibel struktur än en array)

Det finns ingen meningsfull implementation av `area()` och `paint()` i `Shape`!

Deklarera dessa metoder och klassen `Shape` som **`abstract`**.

Forts *geometriska figurer*

```
import java.awt.*;

public abstract class Shape {

    protected int xCoord, yCoord;

    public Shape(int x, int y) {
        xCoord = x;
        yCoord = y;
    }

    public abstract double area();

    public abstract void paint(Graphics g);
}
```

Forts *geometriska figurer*

```
import java.awt.*;

public class Circle extends Shape {

    protected int radius;

    public Circle(int x, int y, int r){
        super(x,y);
        radius = r;
    }

    public double area(){
        return Math.PI*radius*radius;
    }

    public void paint(Graphics g){
        g.setColor(Color.RED);
        g.fillOval(xCoord-radius, yCoord-radius, 2*radius, 2*radius);
    }

}
```

Forts *geometriska figurer*

```
import java.awt.*;

public class Rectangle extends Shape {

    protected int width, height;

    public Rectangle(int x, int y, int w, int h) {
        super(x,y);
        width = w;
        height = h;
    }

    public double area(){
        return width*height;
    }

    public void paint(Graphics g){
        g.setColor(Color.BLUE);
        g.fillRect(xCoord,yCoord,width,height);
    }

}
```

Forts *geometriska figurer*

För att samla ihop ett antal olika figurer skall vi använda en av Javas *samlingsklasser*: `LinkedList` som är en subklass (indirekt) till klassen `Collection` (egentligen ett *interface*).

För denna uppgifts skull räcker det med att kunna

- Skapa ett `LinkedList`-objekt:

```
Collection<Shape> shapes = new LinkedList<Shape>();
```

- Lägga in figurer i listan. T ex:

```
shapes.add(new Circle(x,y,r));
```

- Iterera över elementen och komma åt deras metoder. T ex:

```
for (Shape s:shapes)
    s.paint();
```

Forts *geometriska figurer*

```
import javax.swing.*;
import java.awt.*;
import java.util.*;
public class Drawing extends JPanel {

    private Collection<Shape> shapes;

    public Drawing(Collection<Shape> s, int w, int h) {
        shapes = s;
        setBackground(Color.WHITE);
        setPreferredSize(new Dimension(w,h));
    }

    public double area(){
        double a = 0;
        for (Shape s:shapes) a += s.area();
        return a;
    }

    public void paintComponent(Graphics g){
        super.paintComponent(g);
        for (Shape s:shapes) s.paint(g);
    }
}
```

Forts *geometriska figurer*

```
import javax.swing.*;
import java.awt.*;
import java.util.*;

public class DrawTest extends JFrame {

    public static void main(String[] args){
        Collection<Shape> shapes = read(new Scanner(System.in));
        Drawing d=new Drawing(shapes,400,400);
        System.out.println("Total area: " + (int)d.area());
        new DrawTest(d);
    }

    public DrawTest(Drawing d){
        getContentPane().add(d);
        pack();
        setTitle("DrawTest");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
```

Forts *geometriska figurer*

```
 private static Collection<Shape> read(Scanner sc){
     Collection<Shape> shapes = new LinkedList<Shape>();
     while (sc.hasNext()){
         String s=sc.next();
         if (s.equals("circle")){
             int x, y, r;
             x = sc.nextInt();
             y = sc.nextInt();
             r = sc.nextInt();
             shapes.add(new Circle(x,y,r));
         }
         else {
             int x, y, w, h;
             x = sc.nextInt();
             y = sc.nextInt();
             w = sc.nextInt();
             h = sc.nextInt();
             shapes.add(new Rectangle(x,y,w,h));
         }
     }
     return shapes;
 }

} // end of class DrawTest
```

Forts *geometriska figurer*

Observera hur klasserna bygger på arv från Javas grafik-klasser. Detaljerna i grafiken är överkurs.

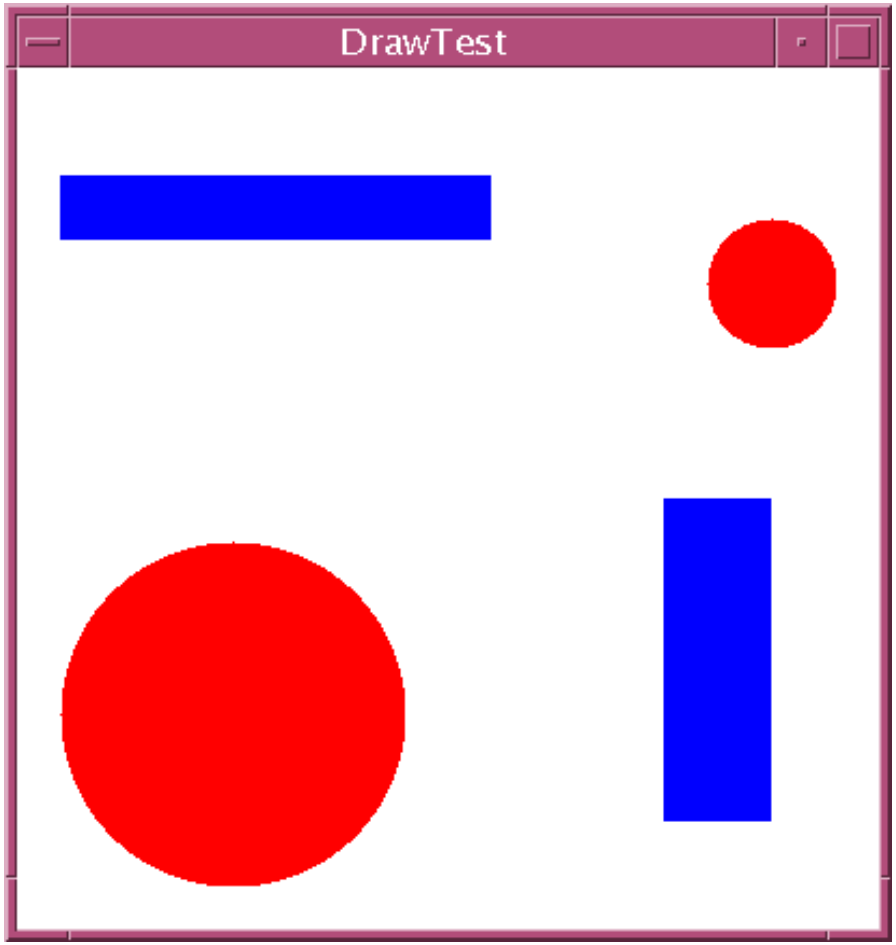
Om programmet körs med

```
java DrawTest < figure.txt
```

och filen `figure.txt` har innehållet

```
rectangle 20 50 200 30
circle 350 100 30
rectangle 300 200 50 150
circle 100 300 80
```

Forts *geometriska figurer*



Exempel: Egen tokenizer

Egen tokenizer: class Token

```
public class MyToken {

    public class MyTokenException extends RuntimeException {
        public MyTokenException(String msg) {
            super(msg);
        }
    }

    /**
     * @return the number for number tokens
     * @throws MyTokenException if not MyNumber token
     */
    public double getNumber() {
        throw new MyTokenException("getNumber called for a nonNumber");
    }

    /**
     * @return String representation (same as toString())
     */
    public String getWord() {
        return toString();
    }
```

Egen tokenizer: forts class Token

```
/**
 * @return the character for MyChar tokens
 * @throws MyTokenException if not MyChar token
 */
public int getChar() {
    throw new MyTokenException("getChar called for a nonChar");
}

/**
 * @return true if this is a MyNumber token else false
 */
public boolean isNumber() {
    return false;
}

/**
 * @return true if this is a MyWord token else false
 */
public boolean isWord() {
    return false;
}
```

Egen tokenizer: forts class Token

```
    /**
     * @return true if this is a MyChar token else false
     */
    public boolean isChar() {
        return false;
    }

    /**
     * @return true if this is a MyEOL token else false
     */
    public boolean isEOL() {
        return false;
    }

    /**
     * @return true if this is a MyEOF token else false
     */
    public boolean isEOF() {
        return false;
    }
}
```

Egen tokenizer: class MyNumber

```
package myTokenizer;

public class MyNumber extends MyToken {
    private double theNumber;

    public MyNumber(double theNumber) {
        this.theNumber = theNumber;
    }

    public boolean isNumber() {
        return true;
    }

    public double getNumber() {
        return theNumber;
    }

    public String toString() {
        return "" + theNumber;
    }

}
```

Egen tokenizer: MyWord

```
package myTokenizer;

public class MyWord extends MyToken {
    private String theWord;

    public MyWord(String theWord) {
        this.theWord = theWord;
    }

    public boolean isWord() {
        return true;
    }

    public String getWord() {
        return theWord;
    }

    public String toString() {
        return theWord;
    }
}
```

Egen tokenizer: class MyEOL

```
package myTokenizer;

public class MyEOL extends MyToken {

    public MyEOL() {
    }

    public boolean isEOL() {
        return true;
    }

    public String toString() {
        return "*EOL*";
    }
}
```

class MyTokenizer

Klasserna `MyEOF` och `MyChar` på motsvarande sätt.

Dessutom behövs själva tokenizern:

```
/*
  A tokenizer for standard input.
  A token is one of the following subclasses to MyToken:
    MyNumber a number (digits, possibly with decimals)
    MyWord a word (a sequence of letters)
    MyEOL an eol
    MyEOF an eof
    MyChar a char i.e any nonspace character not forming the above tokens

    The tokenizer keeps track of the current and the previous token.
*/

package myTokenizer;

import java.io.*;

public class MyTokenizer {
```

class MyTokenizer

```
private StreamTokenizer st;
private MyToken current;
private MyToken previous;

public MyTokenizer() throws IOException {

    st = new StreamTokenizer(
                    new BufferedReader(
                                new InputStreamReader(System.in)));
    st.eolIsSignificant(true);
    st.ordinaryChar('+');
    st.ordinaryChar('-');
    st.ordinaryChar('/');
    st.ordinaryChar('*');
    previous = current = new MyWord("*BOF*");
}

public String toString() {
    return "" + current;
}
```

class MyTokenizer

```
/**
 * Advances to next token
 * @return next token (i e the new current token)
 */
public MyToken nextToken() throws IOException {
    previous = current;
    st.nextToken();
    if (st.ttype==StreamTokenizer.TT_WORD)
        current = new MyWord(st.sval);
    else if (st.ttype==StreamTokenizer.TT_NUMBER)
        current = new MyNumber(st.nval);
    else if (st.ttype==StreamTokenizer.TT_EOL)
        current = new MyEOL();
    else if (st.ttype==StreamTokenizer.TT_EOF)
        current = new MyEOF();
    else
        current = new MyChar(st.ttype);
    return current;
}
```

class MyTokenizer

```
/**
 * @return current token
 */
public MyToken current() { return current; }

/**
 * @return the previous current token
 */
public MyToken previous() { return previous; }

/**
 * @return the line number for the current token
 */
public int lineno() { return st.lineno(); }

/**
 * @param obj an object of any type
 * @return true if current token has same text representation as obj
 */
public boolean equals(Object obj) {
    return toString().equals(obj.toString());
}
```

class MyTokenizer

```
/* All the following methods are for convenience only */

/**
 * Precondition: current token is of type MyChar
 * @return current token as a char
 * @throws MyTokenException if the precondition is violated
 */
public int getChar() { return current.getChar(); }

/**
 * Precondition: current token is of type MyNumber
 * @return current token as a number
 * @throws TokenException if the precondition is violated
 */
public double getNumber() { return current.getNumber(); }

/**
 * @return current token as a String (same as toString()
 */
public String getWord() { return current.getWord(); }
```

class MyTokenizer

```
    /**
     * @return true if current token is end-of-line else false
     */
    public boolean isEOL() { return current.isEOL(); }

    /**
     * @return true if current token is end-of-file else false
     */
    public boolean isEOF() { return current.isEOF(); }

    /**
     * @return true if current token is a number
     */
    public boolean isNumber() { return current.isNumber(); }

    /**
     * @return true if current token is a word
     */
    public boolean isWord() { return current.isWord(); }

    /**
     * @return true if current token is a character
     */
    public boolean isChar() { return current.isChar(); }
}
```

class Parser

```
/*
  Lösning till inlämningsuppgift 4.
  Uppgiften är byggd på en egen tokenizer (MyTokenizer) som är
  implementerad med hjälp av StreamTokenizer

  För dokumentation: Se uppgiftsspecifikationen

  Förvillkor för alla parser-metoder förutom main, run och statement:
  Tokenizern positionerad till det första obehandlade token dvs till
  det token som står på tur att behandlas.

  Eftervillkor för alla parser-metoder utom main, run och statement:
  Tokenizern positionerad till det första obehandlade token.

*/

import java.util.*;
import java.io.*;
import myTokenizer.*;

public class Parser {
```

class Parser

```
private MyTokenizer mt;
private TreeMap<String, Double> map;
private HashSet<String> commands;
private HashSet<String> unaries;

public Parser() throws IOException {
    mt = new MyTokenizer();
    map = new TreeMap<String,Double>();
    unaries = new HashSet<String>();
    commands = new HashSet<String>();
    unaries.add("-");
    unaries.add("exp");
    unaries.add("log");
    unaries.add("sin");
    unaries.add("cos");
    commands.add("quit");
    commands.add("*EOF*");
    commands.add("variables");
    commands.add("clear");
}
```

class Parser

```
public class SyntaxException extends RuntimeException {
    public SyntaxException(String msg) {
        super(msg);
    }
}

public static void main(String [] args) throws IOException {
    Parser p = new Parser();
    p.run();
}
```

class Parser

```
public void run() throws IOException {
    while (true)
        try {
            System.out.print("? ");
            statement();
        } catch (SyntaxException e) {
            System.out.println("*** Syntax Error: " +
                               e.getMessage());
            System.out.println(" Occured after token " +
                               mt.previous());
            while ( !mt.isEOL() )
                mt.nextToken();
        }
}
```

class Parser

```
public void statement() throws IOException {
    mt.nextToken(); // Läs första token från raden
    if ( commands.contains(mt.toString())) {
         command();
    } else {
        double d = assignment();
        map.put("ans", d);
        System.out.println(" : " + d);
    }
    if (!mt.isEOL())
        throw new SyntaxException("Expected EOL but found: " + mt);
}
```

class Parser

```
public void command() throws IOException {
    if (mt.equals("quit") || mt.equals("*EOF*")) {
        System.out.println("Bye");
        System.exit(0);
    } else if ( mt.equals("variables") ) {
        System.out.println("Variables: " + map);
        mt.nextToken();
    } else if (mt.equals("clear")) {
        map.clear();
        mt.nextToken();
    } else
        throw new SyntaxException("Unknown command: " + mt);
}
```

class Parser

```
public double assignment() throws IOException {
    double result = expression();
    while ( mt.equals("=") ) {
        mt.nextToken();
        if (!mt.isWord())
            throw new SyntaxException("Expected identifier " +
                                      " but found: " + mt);
        else {
            map.put(mt.getWord(), result);
            mt.nextToken();
        }
    }
    return result;
}
```

class Parser

```
public double expression() throws IOException {
    double sum = term();
    while ( mt.equals("+") || mt.equals("-") ) {
        int oper = mt.getChar();
        mt.nextToken();
        if (oper=='+')
            sum += term();
        else
            sum -= term();
    }
    return sum;
}
```

class Parser

```
public double term() throws IOException {
    double prod = factor();
    while ( mt.equals("*") || mt.equals("/") ) {
        int oper = mt.getChar();
        mt.nextToken();
        if (oper=='*')
            prod *= factor();
        else
            prod /= factor();
    }
    return prod;
}

public double factor() throws IOException {
    return primary();
}
```

class Parser : primary

```
public double primary() throws IOException {
    double result = 0;
    if ( mt.equals("(") ) { // ( assignment )
        mt.nextToken();
        result = assignment();
        if (!mt.equals(")"))
            throw new SyntaxException("Expected ')' " +
                                      "but found " + mt);
        else
            mt.nextToken();
    } else if (mt.current().isNumber()) { // number
        result = mt.current().getNumber();
        mt.nextToken();
// } else ...
```

class Parser : primary forts

```
    } else if ( unaries.contains(mt.toString()) ) { // unary
        String op = mt.toString();
        mt.nextToken();
        result = unary(op);
    } else if (mt.current().isWord()) { // identifier
        if ( map.containsKey(mt.getWord()) ) {
            result = map.get(mt.getWord());
        } else {
            result = 0; // Enligt spec
        }
        mt.nextToken();
    } else { // error
        throw new SyntaxException("Unexpected: " + mt);
    }
    return result;
}
```

class Parser

```
    public double unary(String op) throws IOException {
        if (op.equals("-"))
            return -primary();
        else if (op.equals("sin"))
            return Math.sin(primary());
        else if (op.equals("cos"))
            return Math.cos(primary());
        else if (op.equals("exp"))
            return Math.exp(primary());
        else if (op.equals("log"))
            return Math.log(primary());
        else
            throw new SyntaxException("Undefined operator: " + op);
    }

} // End of class Parser
```

Övningar

1. För denna föreläsning finns en gammal labbuppgift från tidigare år på kursen som övning. Se katalogen `ioopm/ovningar/stensaxpase` i Mercurial-repositoriet.

Interface

F31

Interface

Antag att vi gör en generell listklass:

```
public class List {
  protected static class ListNode {
    public Object data;

    public ListNode next;

    public ListNode() {
      next = null;
    }

    public ListNode(Object data, ListNode node) {
      this.data = data;
      this.next = node;
    }
  }
```

Forts interface

```
    protected ListNode head;

    public List() {
       head = null;
    }

    public void prepend(Object element) {
        head = new ListNode(element,head);
    }

    public void print() {
       for (ListNode _ = head; _ != null; _ = _.next) {
           _.data.print();
       }
    }
} // end of class List
```

Forts interface

Som synes förutsätter klassen `List` att objekten har en `print`-metod.

Antag att vi försöker lägga in objekt av följande typ

```
public class Book {
  String author, title;

  public Book(String author, String title) {
    this.author = author;
    this.title  = title;
  }

  public void print() { System.out.println( author +  " : "  + title); }

  public String getAuthor() { return author; }

  public String getTitle() { return title; }
}
```

Forts interface

Med följande testklass:

```
public class BookList {

   public static void main(String [] arg) {
      List l = new List();
      l.add(new Book("Chaucer, Geoffrey", "Canterbury tales"));
      l.add(new Book("Adams, Douglas", "The Hitchhiker's Guide"));
      l.add(new Book("Falstaff, Fakir",  "Ett svårskött pastorat"));
      l.print();
  }
}
```

Forts interface

Men, när vi kompilerar:

```
bellatrix$ javac BookList.java
List.java:36: cannot resolve symbol
symbol  : method print ()
location: class java.lang.Object
          _.data.print();
                 ^
1 error
bellatrix$
```

Varför? De lagrade objekten har ju en `print`-metod!

Forts interface

`List` vet inget om de av objekt som skall lagras.

Hur kan vi garantera listan att dess element kommer att ha en `print`-metod?

Möjliga lösningar:

- Definiera en basklass med en `print`-metod och sedan kräva att användaren alltid gör subklasser till denna. Blir dock besvärligt i längden eftersom Java inte tillåter s.k. ”multipelt implementationsarv”.
- I just detta speciella exempel skulle `print`-metoden i klassen `List` anropa `_.data.toString()` i stället eftersom alla objekt har en sådan metod. Ingen generell lösning på problemet.
- Använda ett *interface*.

Forts interface

Ett *interface* är att betrakta ungefär som en abstrakt klass utan instansvariabler där alla metoder är abstrakta.

På svenska använder man ordet *gränssnitt* (eller interfejs)

Exempel:

```
public interface Printable {
   public void print();
}
```

En interfacedeklaration får bara innehålla oimplementerade (abstrakta) metoder. Konstanta klassvariabler (`static final`) får deklareras.

Man behöver inte deklarera vare sig interfacet eller metoder med ordet *abstract* — de är alltid abstrakta i ett interface.

Användning av interface

Man kan låta en klass *implementera* ett gränssnitt vilket betyder att klassen definierar de metoder gränssnittet anger.

Exempel:

```
public class Book implements Printable {
  String author, title;

  public Book(String author, String title) {
    this.author = author;
    this.title  = title;
  }

  public void print() {
    System.out.println( author + " : "  +  title );
  }

  public String getAuthor() { return author; }

  public String getTitle() { return title; }
}
```

Användning av interface

Man kan använda gränssnitt som deklarationer av objekt-referenser. Således skulle man t ex i klassen `Book` skriva:

```
public static void main(String [] args) {
  Printable p = new Book("a", "b");
  Book b = new Book("x", "y");

  p = b;                    // OK

  b = p;                    // *** Fel! Måste skrivas:
  b = (Book) p;

  Object o = b;
  p.print();                // OK
  b.print();                // OK
  o.print();                // *** Fel! Måste skrivas:
  ((Printable) o).print();
  String t = p.getTitle(); // *** Fel!
  String u = b.getTitle(); // OK
}
```

Användning i klassen `List`

Man skulle kunna ändra så att man lagrar `Printable` i stället för `Object` i `ListNode` och `add` men bättre (i detta fall) att bara typkasta där man behöver det:

```
public void print() {
  for (ListNode list = head; list != null; list = list.next)
    ((Printable)list.data).print();
}
```

På så sätt kräver man `Printable` endast om man tänker använda metoden `print` på listan.

En sorterad lista

Ibland kan det vara eftersträvansvärt att de objekt som läggs in i listan har en *ordningsrelation*.

Detta kan realiseras genom att objekten implementerar gränssnittet `Comparable` som finns definierat i `java.lang`:

```
public interface Comparable {
  public int compareTo(Object o);
}
```

Om `a` och `b` är refererar objekt från klasser som implementerar `Comparable` så skall uttrycket

```
a.compareTo(b)
```

returnera ett negativt värde om man anser att $a < b$, ett positivt värde om $a > b$ och 0 om $a = b$.

Klassen `String` implementerar detta gränssnitt.

Sorterad lista forts

Klassen byggs på klassen `List`:

```
public class SortedList extends List {

  public void add(Object o) {
    head = add(o, head);
  }

  // Övning! Skriv om denna till en loop!
  protected static ListNode add(Object o, ListNode l) {
    if (l==null) {
      return new ListNode(o, null);
    } else {
      Comparable newObject = (Comparable) o;
      if (newObject.compareTo(l.data) < 0) {
        return new ListNode(o, l);
      } else {
        l.next = add(o, l.next);
        return l;
      }
    }
  }
```

Sorterad lista forts

```
public class Book implements Printable, Comparable {

  private String author, title;

  public Book(String a, String t) {
    author = a;
    title  = t;
  }

  public void print() {
    System.out.println( author + " : " + title);
  }

  public int compareTo(Object o) {
    Book b = (Book) o;
    return author.compareTo(b.author);
  }
}
```

Sorterad lista forts

```
public class BookList {

   public static void main(String [] arg) {
       List l = new SortedList();
       l.add(new Book("Chaucer, Geoffrey", "Canterbury tales"));
       l.add(new Book("Adams, Douglas", "The Hitchhiker's Guide"));
       l.add(new Book("Falstaff, Fakir", "Ett svårskött pastorat"));
       l.print();
   }
}
```

```
bellatrix$ java BookList
Adams, Douglas : The Hitchhiker's Guide
Chaucer, Geoffrey : Canterbury tales
Falstaff, Fakir : Ett svårskött pastorat
bellatrix$
```

Iteratorer

En *iterator* är en mekanism med vars hjälp man utifrån kan gå igenom de enskilda objekten i en samling som t.ex. en lista, en hashtabell eller ett binärt sökträd.

Detta skall kunna göras *utan* kunskap om den interna representationen.

Med en iterator skulle man t ex kunna printa ut elementen i listan i föregående exempel listexempel utan att använda metoden `print()`

```
List l = new List();
....

Iterator li = l.iterator();
while (li.hasNext()) {
  System.out.println(li.next());
}
```

Iteratorer forts

Komplettera klassen *List* med till exempel en inre klass som implementerar `Iterator`-interfacet:

```
public class ListIterator implements Iterator {
  protected ListNode current = head;

  public boolean hasNext() { return current != null; }

  public Object next() {
    Object theData = current.data;
    current = current.next;
    return theData;
  }

  public void remove() {}
}

public ListIterator iterator() {
  return new ListIterator();
}
```

(Klassen behöver naturligtvis inte vara en inre klass.)

Iterator forts

`Iterator` är ett `interface` som finns deklarerat i `java.util.Iterator` som alltså måste importeras. Interfacet deklarerar de tre metoder som definierades i klassen `ListIterator` ovan.

Alla klasser som implementerar `Iterator`-interfacet kan sedan itereras över på samma sätt (generisk kod).

```
Iterator iter = list.iterator();
while (iter.hasNext() ) {
  System.out.println(iter.next());
}
```

```
for (Object data : list) {
  System.out.println(data);
}
```

Observera att `for`-loppen ovan ser annorlunda ut. Resultatet är *stabilare kod*, även än den i `while`-loopen – varför?

Sammanfattning

- Ett interface är ett kontrakt: om I deklarerar metodsignaturen m och C (D) implementerar I måste m finnas i C (D)
  - Klientkod kan då skrivas mot I som fungerar oavsett om objektet är C eller D under körning
- Eftersom interface kan implementeras av olika klasshierarkier är interface ett utmärkt sätt att separera olika klasser från varandra
- Interface är ett sätt att ge Java många av de fördelar som multipelt arv ger (främst generell typsäker kod), utan nackdelarna
- Nackdelar med multipelt implementationsarv
  - komplext (vad händer om A ärver B flera gånger, två superklasser som båda definierar m, etc.)
  - svårt att implementera effektivt
- Ex. på språk med multipelt implementationsarv: C++, Python, Eiffel
- Många standardinterface i Javas klassbibliotek, även s.k. ”marker interfaces” som är tomma

Övningar

1. Utöka den personklass som använts i tidigare övningar så att den implementerar `Comparable`-interface:t och sorterar med avseende på personnummer som också bör utökas till att implementera `Comparable`.
2. Skriv ett program som skapar tio slumpvisa personer och lägger dem i en lista som sedan skall sorteras med `Collections.sort(listan)`.
3. Skriv två klasser `A` och `B` som båda ärver från `Object` samt har en variabel `value` av typen **`int`**. `A` och `B` skall implementera `Comparable` så att när en lista med blandade `A`- och `B`-objekt sorterar sig själv kommer alla `A`-objekt först, inbördes sorterande i stigande ordning med avseende på `value`, följt av alla `B`-objekt i fallande ordning med avseende på `value`. Testa programmet genom att generera slumpmässigt data som stoppas in i en lista och skriv ut listan före och efter sortering med `Collections.sort(listan)`.
4. Skriv enhetstest med `JUnit` för att testa programmet ovan.

Generiska klasser, kombination med arv, interface etc.

F32 & F34

Generiska datatyper

- Hittills har vi sett att klasser och interface definierar nya typer som kan användas i ett program
- Vid vissa tillfällen, t.ex. för att skapa generiska datasamlingar, är det lämpligt att *binda* vissa typer i en klassdefinition vid *användningstillfället* snarare än vid *definitionstillfället*
- Man kan tänka att en *generisk klass* är en definition med ett hål, som måste fyllas innan klassen kan användas, ex.: `List< >` kan *instanseras som typen* `List<Person>` som avser en lista som kan hålla Person-objekt
- I Java fyller kompilatorn automatiskt alla icke-fyllda hål med `Object` – men sådan kod är osäker och skall undvikas
- *Men vad är egentligen en typ?*

Utvikning: Typer

- En typ är en etikett som ger en semantisk mening till ett dataobjekt
- Man kan t.ex. tolka typer med hjälp av mängdlära; i Java kan man säga att tolkningen (interpretationen) $[[T]]$ av typen T är mängden av alla objekt som kan beskrivas (helt eller delvis) av typen T
- Därav följer att $[[\texttt{Object}]]$ är alla tänkbara Javaobjekt
- Ett typsystem är en syntaktisk metod för att bevisa avsaknaden av vissa klasser av fel genom att klassificera ett programs alla uttryck utefter vilka värden de beräknar[2]
- Subtypning: $A \texttt{ extends } B \implies [[A]] \subseteq [[B]]$
- Java har nominell typning: relationer mellan typer måste deklareras explicit i koden
- Java är statiskt typat + downcasts (typomvandlingar nedåt i arvshierarkin) är osäkra = ett behov av generics

[2] Types and Programming Languages. B.C. Pierce

Generisk lista

```
/**
 * En klass för att demonstrera hur man använder
 * typparametrar -- så kallad "generisk kod"
 *
 */

public class List<T> {
  protected ListNode head;

  protected class ListNode {
    T        data;  // T används som typ
    ListNode next;

    ListNode(T data, ListNode next) {
      this.data = data;
      this.next = next;
    }
  }

  private class ListIterator implements Iterator {
    private T current;

    ...
  }
```

Generisk lista

```
public Iterator iterator() {
  return new ListIterator(head);
}

public void add(T d) {
  head = new ListNode(d, head);
}

public String toString() {
  String result = "";
  for(Object obj : this) {
    result = result + " " + obj;
  }
  return "List(type=" + data.getClass().getName() + ",values=[" + result + " ])"
}
```

Generisk lista

```
    // Testprogram för GenericList

    public static void main(String [] args) {
        GenericList<Integer> l = new GenericList<Integer>();
        for (int i = 1; i<10; i++)
            l.add(i);          // "Autoboxing"
        System.out.println(l);

        GenericList<Character> c = new GenericList<Character>();
        for (int i = 52; i>31; i--)
            c.add( (char)i ); // "Autoboxing"
        System.out.println(c);
    }

} // End of GenericList

/*** Output:

[ 9 8 7 6 5 4 3 2 1 ]
[   ! " # \$ % & ' ( ) * + , - . / 0 1 2 3 4 ]

*/
```

Generisk lista

```
/**
 *
 *  Allt på en gång:
 *     Arv, interface, iteratorer, generics
 */
public class Book implements Comparable {
    String author, title;
    public Book(String a, String t) {
        author = a;
        title  = t;
    }

    public int compareTo(Object o) {
        Book b = (Book) o;
        return author.compareTo(b.author);
    }

    public String toString() {
        return author  + " : " + title ;
    }
}
```

Generisk lista

```
import java.util.Iterator;

public class List <T> {

  protected ListNode head;

  public  List() {
    head = null;
  }

  class ListNode {
    T data = null;
    ListNode next;

    public ListNode() {
      next = null;
    }

    public ListNode(T data, ListNode next) {
      this.data = data;
      this.next = next;
    }

  } // end ListNode
```

Generisk lista

```
public class ListIterator implements Iterator {
    protected ListNode next = head;

    public boolean hasNext() {
        return next!=null;
    }

    public T next() {
        T theData = next.data;
          next = next.next;
          return theData;
    }

    public void remove() {
      throw
        new UnsupportedOperationException("No remove");
    }

} // end ListIterator
```

Generisk lista

```
public ListIterator iterator() {
  return new ListIterator();
}

public void insertFirst(T newObject) {
  ListNode _ = new ListNode(newObject, head);
  head = _;
}
```

Generisk lista

```
// Huvudprogram

public static void main(String [] arg) {

  List<Book> l = new List<Book>();

  l.insertFirst( new Book("Chaucer, Geoffrey", "Canterbury tales"));
  l.insertFirst( new Book("Adams, Douglas", "The Hitchhiker's Guide"));
  l.insertFirst( new Book("Falstaff, Fakir", "Ett svårskött pastorat"));
  Iterator li = l.iterator();
  while (li.hasNext() )
      System.out.println(li.next());

} // end main

} // end List

/* Output:
kursa$ java List
Falstaff, Fakir : Ett svårskött pastorat
Adams, Douglas : The Hitchhiker's Guide
Chaucer, Geoffrey : Canterbury tales
kursa$
*/
```

Sorterad generisk lista

```
import java.util.*;

public class SortedList <T> extends List<T> {

  public void add(T o) {
     head = add(o, head);
  }

  protected ListNode add(T o, ListNode l) {
    if (l==null)
        return new ListNode(o, null);
    else {
        Comparable co = (Comparable) o;
        Comparable cl = (Comparable) l.data;
        if (co.compareTo(cl)<0)
            return new ListNode(o, l);
        else {
            l.next = add(o, l.next);
            return l;
        }
    }
  }
```

Några tillkortakommanden med Javas typsystem

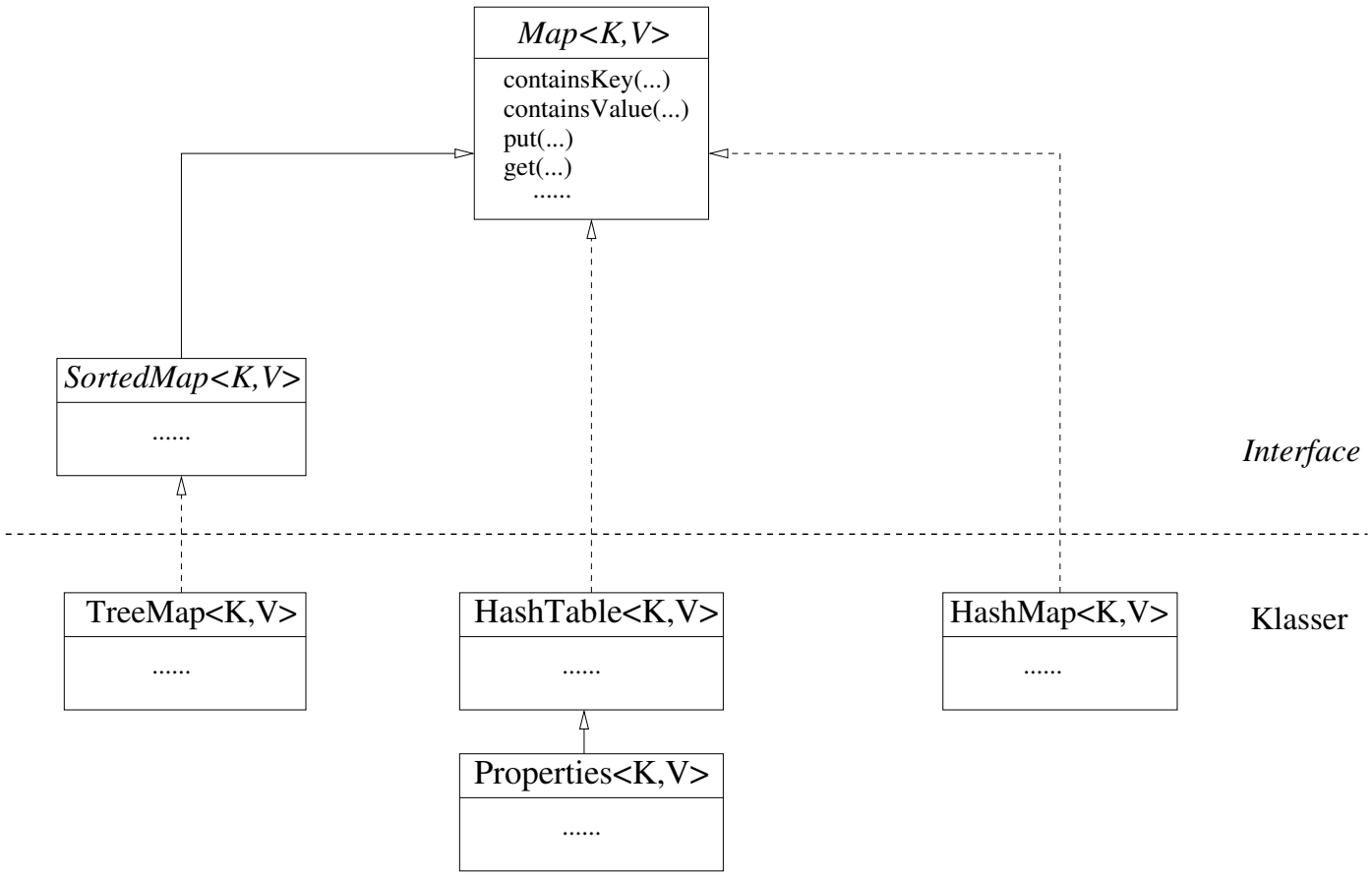
- Problem med subtypning av arrayer
- Lösning: extra typkontroll under körning

- Samma problem gäller för generiska datatyper t.ex. listor
- Lösning: wildcard-typer: `List<?>`

Sorterad generisk lista

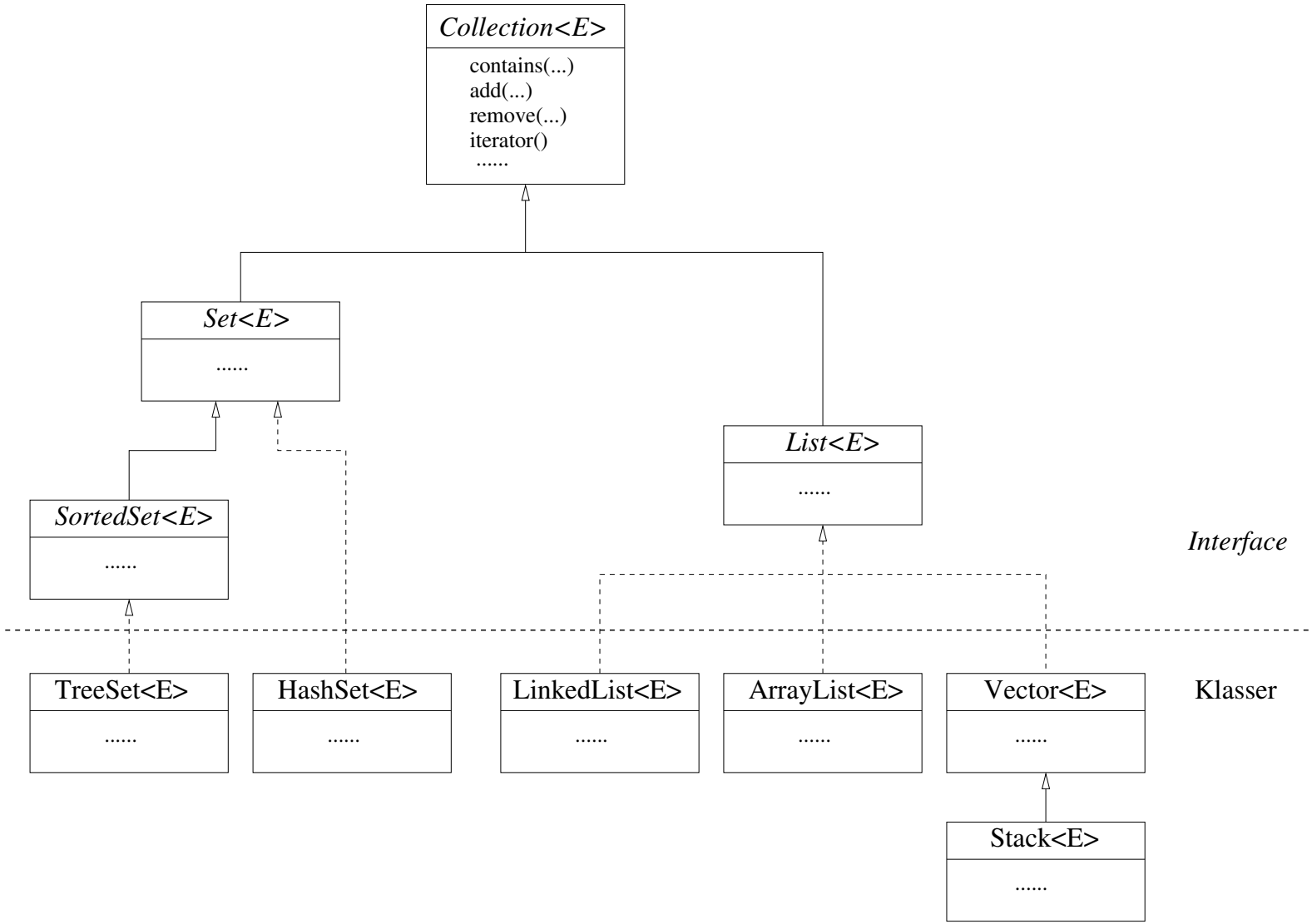
```
  // Huvudprogram

  public static void main(String [] arg) {
     SortedList<Book> l = new SortedList<Book>();
     l.add(new Book("Chaucer, Geoffrey", "Canterbury tales"));
     l.add(new Book("Adams, Douglas", "The Hitchhiker's Guide"));
     l.add(new Book("Falstaff, Fakir", "Ett svårskött pastorat"));

     Iterator li = l.iterator();
     while ( li.hasNext() )
        System.out.println( li.next() );
  }
}

/* Output:

kursa$ java SortedList
Adams, Douglas : The Hitchhiker's Guide
Chaucer, Geoffrey : Canterbury tales
Falstaff, Fakir : Ett svårskött pastorat
kursa$
*/
```

Standardklasser



(ej fullständig)

Standardklasser



(ej fullständig)

Minns ni skitkoden i `Book`?

Kan krascha när som helst om vi inte får in en `Book`!

```
public int compareTo(Object o) {
  Book b = (Book) o;
  return author.compareTo(b.author);
}
```

Bättre:

```
public int compareTo(Object o) {
  if (o instanceof Book) {
    return author.compareTo(((Book) b).author);
  } else {
    return false;
  }
}
```

Men fortfarande inte klockrent – vi vill förhindra att icke-böcker skickas in från starten.

Comparable-interfacet är generiskt!

Så här är interfacet deklarerat egentligen:

```
public interface Comparable<T>
```

Det är bara så att Java fyllt i ”hålen” hittills. Hellre skulle vi skriva:

```
public class Book implements Comparable<Book> { ...
```

vilket betyder att böcker endast kan jämföras (är *comparable to*) andra instanser av `Book`, vilket oftast är vad man vill. Konsekvensen är:

```
public int compareTo(Book otherBook) {
  return (author+title).compareTo(otherBook.author+otherBook.title);
}
```

det vill säga, argumenttypen för `compareTo` är `Book` och inte `Object`.

Användning av standardklasser

```
/** Demonstrerar användning av LinkedList och TreeSet. */

import java.util.*;

public class Book implements Comparable<Book> {

  String author, title;

  public Book(String a, String t) {
      author = a;
      title  = t;
  }

  public int compareTo(Book b) {  // Först författare, sedan titel
      if (author.compareTo(b.author)==0)
          return title.compareTo(b.title);
      else
          return author.compareTo(b.author);
  }

  public String toString() {
      return author  + " : " + title ;
  }
```

Användning av standardklasser

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

  // Skapa en LinkedList och sedan några
  // några Book-objekt som direkt läggs in i listan

  List<Book> ll = new LinkedList<Book>();
  ll.add(new Book("Chaucer, Geoffrey", "Canterbury tales"));
  ll.add(new Book("Adams, Douglas", "The Hitchhiker's Guide"));
  ll.add(new Book("Falstaff, Fakir", "Ett svårskött pastorat"));
  ll.add(new Book("Adams, Douglas", "Ajöss och tack för fisken"));

  // Skriv ut listan med hjälp av en iterator

  System.out.println("\nBoklista:\n");
  Iterator<Book> li = ll.iterator();
  while (li.hasNext())
      System.out.println("  " + li.next());
```

Användning av standardklasser

```
    // Lägg in i ett TreeSet
    li = ll.iterator();
    SortedSet<Book> ts = new TreeSet<Book>();
    while (li.hasNext()) {
        Book b = li.next();
        ts.add(b);
    }
    System.out.println("\nSorterad boklista:\n");
    li = ts.iterator();
    while (li.hasNext())
        System.out.println("  " + li.next());

    System.out.println("\nSorterad boklista:\n"); // Enklare
    for (Book b : ts) System.out.println("  " + b);

    System.out.println("\nUt från listan:\n");
    li = ll.iterator();
    while (li.hasNext()) {
        System.out.println("   " + li.next());
        li.remove(); // Tar bort den som sist returnerades av next()
    }
  }
} // end class Book
```

Användning av standardklasser - Output

```
Boklista:
  Chaucer, Geoffrey : Canterbury tales
  Adams, Douglas : The Hitchhiker's Guide
  Falstaff, Fakir : Ett svårskött pastorat
  Adams, Douglas : Ajöss och tack för fisken

Sorterad boklista:
  Adams, Douglas : Ajöss och tack för fisken
  Adams, Douglas : The Hitchhiker's Guide
  Chaucer, Geoffrey : Canterbury tales
  Falstaff, Fakir : Ett svårskött pastorat

Sorterad boklista:
  Adams, Douglas : Ajöss och tack för fisken
  Adams, Douglas : The Hitchhiker's Guide
  Chaucer, Geoffrey : Canterbury tales
  Falstaff, Fakir : Ett svårskött pastorat

Ut från listan:
   Chaucer, Geoffrey : Canterbury tales
   Adams, Douglas : The Hitchhiker's Guide
   Falstaff, Fakir : Ett svårskött pastorat
   Adams, Douglas : Ajöss och tack för fisken
```

Övningar

1. Uppdatera övningarna från föreläsning 30 så att `Comparable`-interface:t är parameteriserat av en lämplig typ.
2. Skriv en abstrakt klass `Pair` som tar två typparametrar `T` och `V` och innehåller två privata variabler `first` och `second` av typerna `T` respektiver `V`. `Pair` skall ha metoderna `getFirst()`, `setFirst()` etc. med lämpliga typer.
3. Skapa en klass `IntStringPair` som är en subklass till `Pair` som representerar par av **`int`**:ar och strängar.
4. Utöka `Pair` till att implementera `Comparable`. Hur skall typparametern till Comparable se ut?
5. Skriv ett program som skapar olika slags par (t.ex. heltal & sträng, sträng & sträng, personnummer & person, etc.) och stoppar in dem i en lista, som sorteras med `Collections.sort(listan)`. Vilka typparametrar skall listan ha? Varför?

Avslutning Java, C, principer, etc.

F35

Imperativ programmering

Vad är det?

- Ett *programmeringsparadigm* att jämföras med *funktionell* programmering och *logikprogrammering*.
- ”Vanlig programmering”
- Programmen skrivs som en sekvens av satser (”kommandon”)
- Programmen struktureras med hjälp av *procedurer* (*subrutiner*, *underprogram*, *funktioner* . . .)
- Kommandon ändrar tillstånd och funktioner kan ha sidoeffekter
- Den äldsta programmeringsmetodiken på grund av dess nära koppling till hårdvaran (von Neuman-modellen)
- Många av de äldsta språken (Fortran, algol, C, . . .) konstruerades för den tekniken

Imperativ programmering – forts

Ett centralt begrepp i *all* programmering är *abstraktion*.

Med detta menas att man döljer tekniska detaljer, finner gemensamma drag hos problem och hittar generella lösningar.

Programmeringsspråken i sig innehåller abstraktioner i form av

- grundläggande operationer i själva språket (aritmetik på olika datatyper, selektion, iteration),
- fördefinierade *funktioner* (t ex `printf`, `getchar`, `sin` ...)

Funktioner är det äldsta sättet för programmeraren att skapa egna abstraktioner.

Andra abstraktionsmekanismer: moduler, klasser, paket

Imperativ programmering – forts

På 60-talet havererade många större programmeringsprojekt och andra drabbades av

- kraftiga förseningar
- skenande kostnader,

Dessutom:

- Programmen uppfyllde inte kravspecifikationen och
- koden gick inte att förstå och underhålla

Motmedel:

- ”Software engineering”
- Nya programmeringstekniker

Strukturerad programmering

- Edsger Dijkstra: *Go To Statement Considered Harmful*.
- Böhm och Jacopini: Alla beräkningsbara funktioner kan implementeras med kontrollstrukturerna
  - *sekvens*,
  - *selektion* typ **if** (och/eller **switch**) och
  - *iteration* typ **while**, (och/eller **for**, **do**).
- Handlar således mycket om att programmera utan **goto**.
  Dock brukar man acceptera ”strukturerade” **goto**-satser som **break**, **continue**, **return** och **throw**.
- Pascal utvecklades av Niklaus Wirth som var en av profeterna för strukturerad programmering.
  Pascal resulterar i ”monolitiska” program.

Objektorienterad programmering (OOP)

Hittills hade programmen strukturerats runt *algoritmer*.

Ole-Johan Dahl och Kristen Nygaard utvecklade Simula 67 där programmen i stället (kunde) struktureras kring de *data* som skulle behandlas vilket lade grunden för det objektorientering.

Språket innehöll väsentliga mekanismer som man förknippar med med OOP: *klasser*, *klasshierarkier*, *inkapsling*, *information hiding*, *objekt*, *polymorfi*

Objektorientering: klasser

Klassen är det mest centrala begreppet i OOP.

En klass är en *abstraktion* av något begrepp ett program hanterar. Man kan också säga att en klass är en (ritning till en) modell av något.

Exempel:

- Klassen `Car` i trafiksimlueringen. En verklig bil har många egenskaper men vi modellerar endast de egenskaper vi behöver.
- Klassen *Queue* är en abstraktion (modell) för en kö – oavsett vad som finns i kön
- Klassen *AbstractMap* är en abstraktion av en avbildning
- Klassen *Addition* är en abstraktion av alla möjliga additioner
- Klassen *SyntaxException* är en abstraktion av alla syntaxfel användaren kan göra

Klasser — forts

En klass-beskrivning innehåller en ”förteckning” över *attribut* (data) *operationer* (*metoder*) som ingår.

I Java finns *definieras* alltid metoderna i klassen (undantag: *abstrakta* klasser) men i t ex C++ ligger ofta definitionerna på annat ställe (`.h` respektive `.c`-filer)

Klasser kan *instansieras* till *objekt* dvs man skapar en modell enligt ritningen. I denna process allokeras minnesutrymme för för attributen och dessa tilldelas attributen konkreta värden.

Instansieringen initieras med operatorn `new` och själva initieringen utförs av koden i en *konstruktor*.

Inkapsling

Vi har upprepade gånger tagit upp principen *inkapsling* (se även *"information hiding"*), vilket bl.a. innebär

- att all åtkomst till objektet skall gå via ett väldefinierade *interface*-metoder och
- att användaren av klassen inte skall behöva känna till detaljer i klassens implementation.

Man vinner

- att problemet blir strukturerat — man behöver inte tänka på allt hela tiden
- att klassen själv kan kontrollera att den används på rätt sätt
- att det går att ändra i implementationen (för att rätta fel, öka effektivitet mm) utan att användarkoden behöver ändras.

Inkapsling – forts

Inkapsling åstadkommes genom att ge attribut och metoder har olika synlighet:

- `private` – förhindrar access från alla andra klasser
- *paket* – förhindra access från andra paket
- `protected` – förhindra access från andra klasser än subklasser
- `public` – inga hinder

Klasshierarkier och arv

De olika klasserna i ett program kan byggas i hierarkier (basklasser – subklasser).

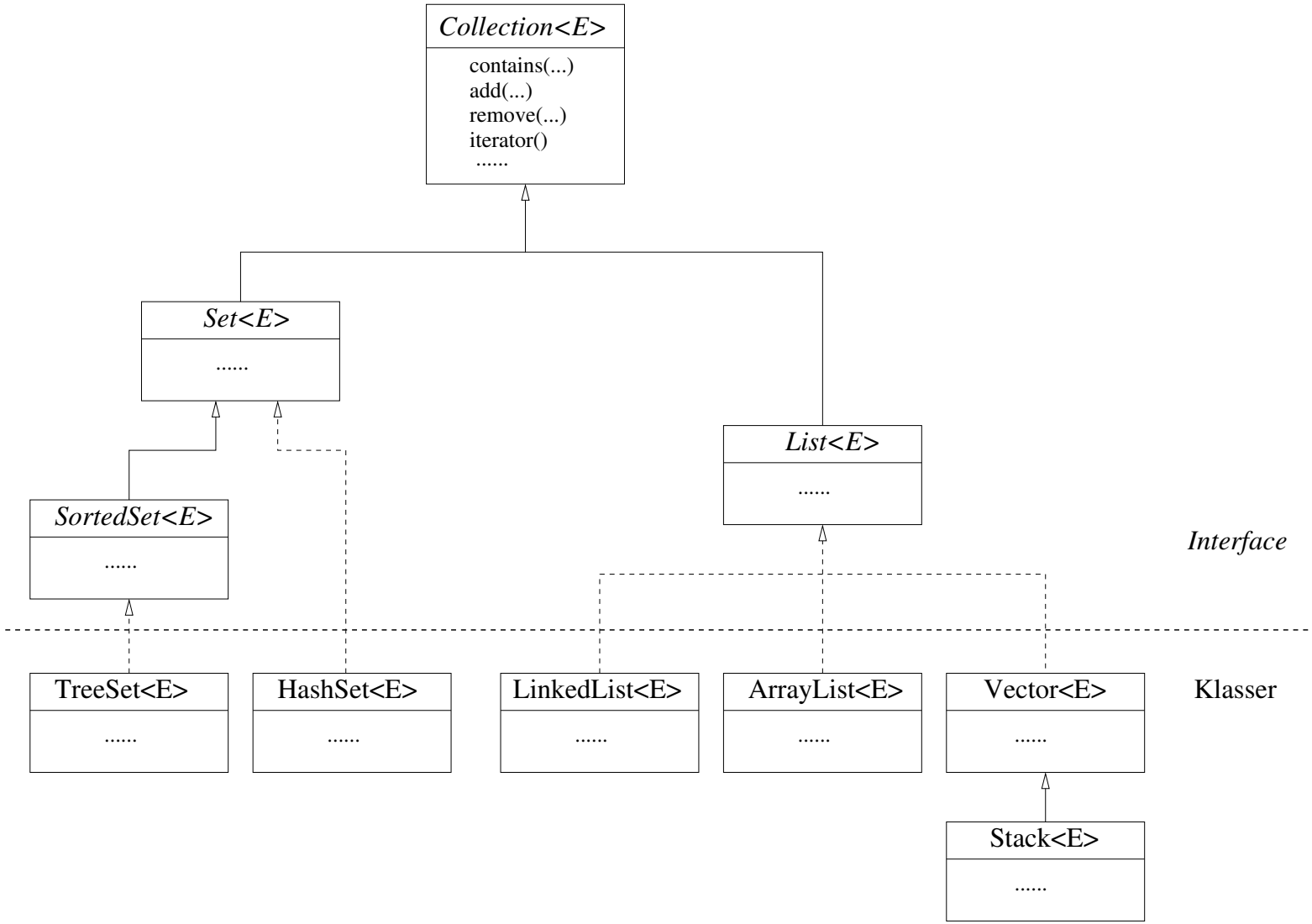
Klasser högre upp i hierarkien representerar högre *abstraktionsnivåer* medan klasser lägre ner står för ökad *specialisering*.

Exempel:

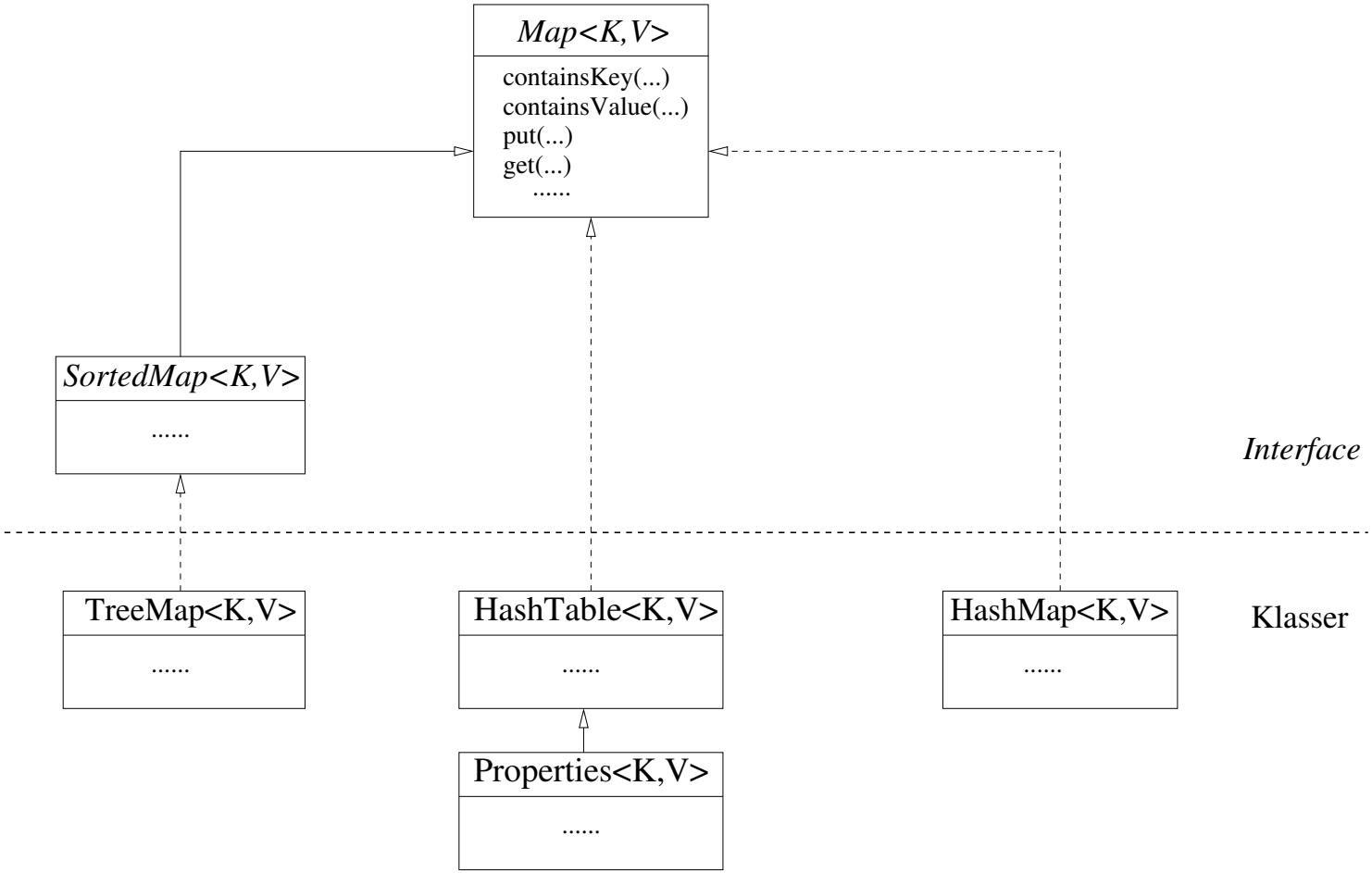
Fordon —> Motorfordon —> Lastbil

Component —> Container —> Window —> Frame —> JFrame

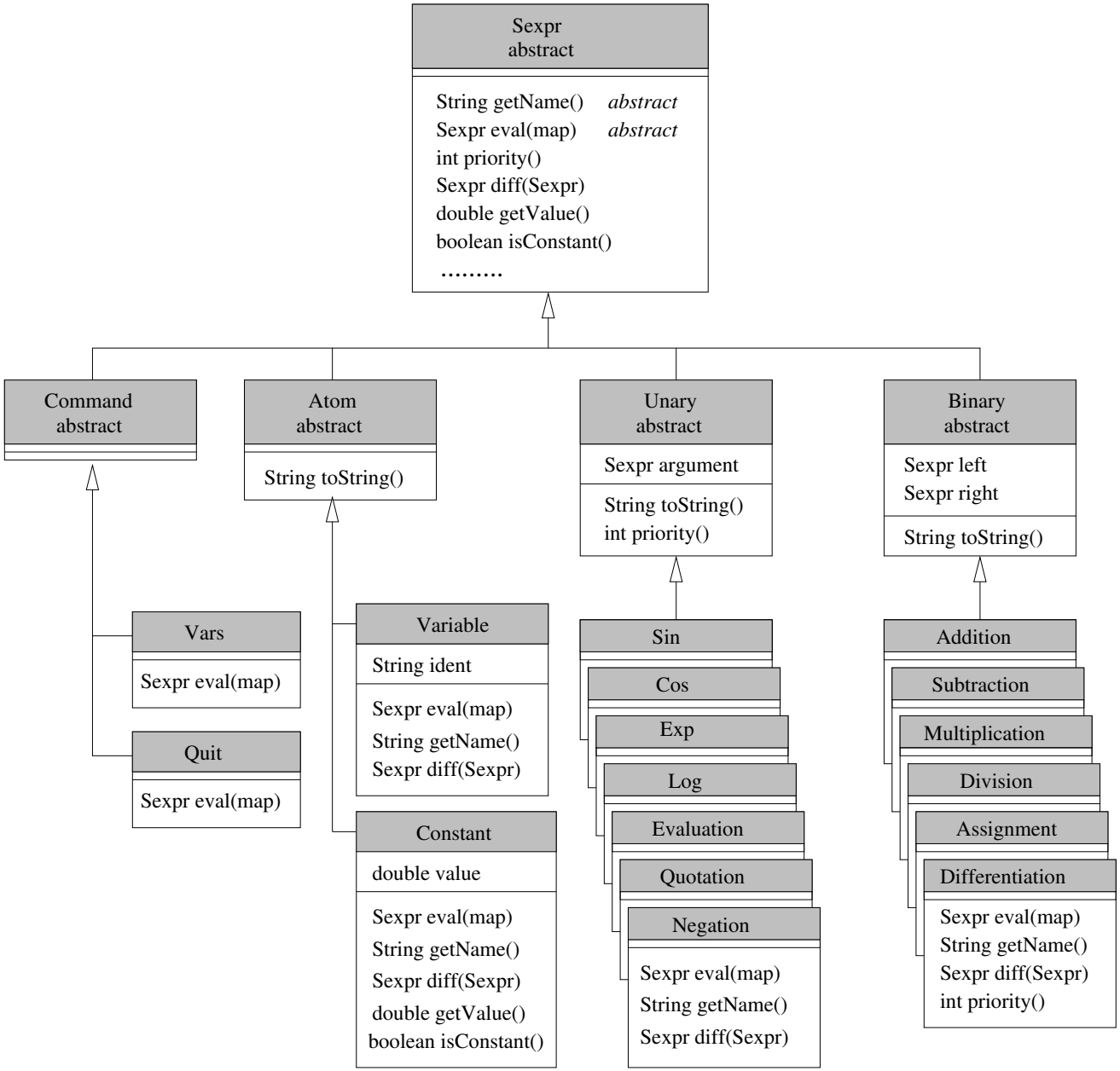
Klasshierarkier och arv — forts



Klasshierarkier och arv — forts



Klasshierarkier och arv — forts



Polymorfism

Det finns många sorters polymorfism

- De aritmetiska operatorerna är polymorfa eftersom de utför olika operationer beroende på operandernas typer (heltal, flyttal)
- I bl a C++ kan man lägga till egna definitioner för olika datatyper (klasser) för *alla* operatorer (så kallad *operatoröverlagring*)
- Samma namn på funktioner (metoder) som har olika ”signatur”
- Mest väsentligt för OOP: Man kan definiera en metod på flera ställen i en klasshierarki och det är den som passar typen (klassen) bäst som som används — dynamisk bindning.
  Exempel: `toString()`, `isConstant()`, `getValue()`
- Mallar (”generics”) kan ses som ett hjälpmedel för att åstadkomma polymorfi

Andra viktiga begrepp vi diskuterat

- Iteratorer
- Undantag
- Parameteröverföringsmetoder: värdeanrop, referensanrop
- Tekniken att bygga rekursiva strukturer. Exempel: inlägg i *mängd* organiserad som ett BST (dvs ett *TreeSet*):

```
public TreeNode insert(int key, TreeNode r) {
  if (r==null) {
    return new TreeNode(key);
  else if (key < r.key)
    r.left = insert(key, r.left);
  else if (key > r.key) {
    r.right = insert(key, r.right);
  return this;
}
```

Java-detaljer

- Struktur: en klass per fil (huvudsakligen). Paket.
- Objekthantering: alltid med referenser
- Inga pekare, ingen direkt minnesåtkomst
- Alla klasser har minst en konstruktor, anrop till **`super`**-konstruktorn
- Metoderna `toString()` och `equals(Object o)`
- **`static`**
- Arvshierarkier med **`extends`**
- *Abstrakta* klasser och metoder (**`abstract`**)
- *Interface* (**`interface`**, **`implements`**)
- Generics: klasser med *typparametrar*

C-programmering

- Struktur: deklarationer på `.h`-filer, implementationer på `.c`-filer.
- Preprocessor: `#include`, `#define`, `#ifndef`, `#endif`
- **`struct`**
- Funktioner: värdeanrop, värderetur, kan returnera **`struct`**
- Pekare och pekararitmetik. Arrayer
- Minneshantering: explicit med `malloc` och `free`. Ingen skräpsamling!

Hur programmerar man objektorienterat i C

Exempel: Klassen `Die`

- Använd **`struct`** i stället för klass:

```
typedef struct die {
        int numberOfSides;
        int value;
    } die, *Die;
```

  Inkapsling möjlig (hur?), dynamisk bindning måste implementeras för hand

Die i C

- Implementera konstruktorer som funktioner som returnerar pekare:

```
public Die newDie(int nos) {
  Die d = (Die) malloc(sizeof(die));
  if (d) {
    d->numberOfSides = nos;
    roll(d);
  } else {
    errno = ERRMEM;
  }
  return d;
}
```

Die i C

- Skriv accessfunktioner där en parameter svarar mot **this**-pekaren.

```
int getValue(Die d) {
   return d->value;
}

int roll(Die d) {
   return d->value = rand() % (d->numberOfSides) + 1;
}
```

Det går att dölja ”attributen,” men man måste vara *disciplinerad*!

Observera också att omdefiniering av t.ex. `roll()` inte är möjlig.

Användning av Die

```
int main() {
   Die t1 = newDie(42);
   Die t2 = newDie(42);
   int n = 0;
   while (roll(t1)!=roll(t2)) {
      n++;
   }
   printf("Tärningarna blev lika efter %d slag\n", n);
   printf("Värdet var %d\n", getValue(t1));
   return 0;
}

/* Körresultat:
vranx$ gcc -o die die.c
vranx$ die
Tärningarna blev lika efter 7 slag
Värdet var 16
*/
```

Exempel: En map i C

```
/* map.h Avbildning från teckensträngar till VT */

#ifndef __map__
#define __map__

#define T void * // Definierar värdetyp

typedef struct listNode {
  char *key;
  T value;
  struct listNode *next;
} listNode, *link;

typedef struct mapObj {
  link first;
  link current;
} mapObj, *Map;
```

forts map.h

```
Map newMap(); // Skapa map

void deleteMap(Map m); // Frigör allt allokerat utrymme

T put(char *key, T v, Map m); // Lagrar nyckel-värde-par

T get(char *key, Map m); // Söker värde

int containsKey(char *key, Map m); // Undersöker om värde finns

void printMap(Map m, void vprint(void *));

#endif

// end of map.h
```

Annat för objektorientering

- Variabelt antal argument till funktioner är möjligt både i C och Java, med relativt olika implementation.
- För C, se `stdarg.h` med funktionerna `va_start`, `va_arg` och `va_end`. Man måste själv hålla reda på antal och typer. Jfr t ex `printf`.
- I Java kan variabla argumentlängder enkelt abstraheras in i arrayer, men möjligheten att blanda typer försvinner (modulo `Object[]`).
- Dynamisk bindning kan implementeras med funktionspekare i poster
- Anropen tar formen `var->fpek(var, arg1, ..., argn)`
- "Magin" händer i och med uppslagningen av `fpek`

Funktionspekare: exempel

```
typedef struct person {
  void (*setAge)(Person this, int);
  int age;
} person, *Person;

Person newPerson(void (*setAge)(Person this, int)) {
   Person this = (Person) malloc(sizeof(person));
   this->setAge = setAge;
   return this;
}

void setAge(Person this, int age) { this->age = age; }
void setAgeWithCheck(Person this, int age) {
  if (age >= 0 && age <= 120) {
    this->age = age;
  } else {
    errno = EINVAL;
  }
}

int main() {
   Person p1 = newPerson(setAge); Person p2 = newPerson(setAgeWithCheck);
   p1->setAge(p1, 1000); p2->setAge(p2, 1000);
}
```

Sista ord

- ”Vem som helst” kan skriva ett fungerande program
- Vad som spelar roll är *hur programmet skrivs*
  - Kod som går att underhålla, utveckla, testa, felsöka, etc.
  - Använda existerande tekniker, fatta underbyggda beslut
- En programmerare kan hålla ca 20 000 rader kod i minnet
- Det stora flertalet program ryms inte i 20 000 rader kod
  - Abstraktion och struktur
  - Tester
- Objekt-orienterad design
  - Hur tänker man kvalitativt kring ett program uppbyggnad och icke-funktionella aspekter?
- Storskalig programmering
  - Vad är ”great code” och hur ser man till att det är sådan kod man skriver?