Innehåll

- OOP snabbintroduktion
- Programvaruutveckling och programmering
- Datatyper
- Uttryck
- Satser
- Arv, polymorfi och dynamisk bindning
- Att organisera Javakod
- Metodik och klassdesign
- Design (CRC)
- Abstrakt arv, polymorfism, dynamisk bindning
- Fält
- Undantag
- In-/utmatning och filer
- Grafik
- GUI:s
- Applets vs applikationer
- Rekursion
- Interfaces och sortering
- Noggrannheten i beräkningar

Abstrakta Klasser 1

- God klassdesign placerar gemensamma attribut och metoder så högt som möjligt i hierarkin
- … men ibland kan dessa egenskaper inte definieras fullständigt
- Abstrakta klasser innehåller ofta abstrakta metoder
- En abstrakt metod har ingen implementation, bara en signatur
- Klasser med abstrakta metoder blir abstrakta
- ➨ Abstrakta klasser är ofta för allmänna för att vara direkt användbara
- ➨ En abstrakt klass kan inte instantieras
- Se t ex klassen `Figure` i `shapes2` exemplet

Abstrakta Klasser 2

```
public abstract class Figure
{
   private int xPosition;
   private int yPosition;
   private String color;

   public Figure()
   {
      xPosition = 60;
      yPosition = 50;
      color = "red";
      draw();
   }// Figure()

   public int getX()
   {
      return xPosition;
   }

   public void moveVertical(int distance)
   {
      erase();
      yPosition += distance;
      draw();
   }

   public abstract void changeSize(int newSize);

   // erase();
   // change size(s) here
   // draw();
```


Abstrakta Klasser 2

```
/*
 * Draw the figure on screen.
 */
protected abstract void draw();

// Canvas canvas = Canvas.getCanvas();
// canvas.setForegroundColour(color);
// construct an appropriate Shape´ object
// and call `canvas.fill()´ with this
// `Shape´object
// canvas.wait(10);

/*
 * Erase the figure on screen.
 */
protected abstract void erase();
 // Canvas canvas = Canvas.getCanvas();
 // construct an appropriate `Shape´object and
 // call `canvas.erase()´ with this `Shape´object
}
```

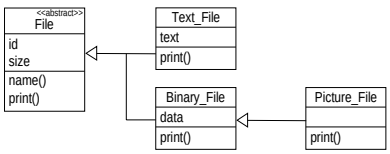
Bara metodhuvudet

Abstrakta Klasser - deklaration

```
public abstract class Classname
{
   •attribut deklareras
   •implementerade metoder
   •abstrakta metodhuvuden
}
```


Abstrakta Klasser 3

- Abstrakta klasser (och metoder) markeras med modifieraren `abstract`
- Subklasser som inte (om)definierar alla abstrakta metoder blir också abstrakta
- Exempel:



Paket 1

- Ett Java paket är en mängd klasser
- Paket användas för att gruppera liknande klasser och/eller sådana som beror på varandra
- Klasserna i ett paket behöver inte ärva från varandra
- Java `AWT` är ett paket (`java.awt`)
- Java API:n består av många paket
- `import` satsen gör innehållet i ett paket tillgängligt

```
import paketnamn.klassnamn;      <---- enstaka klass
import paketnamn.*;              <---------- alla klasser i paketet
```


Paket 2

- Programmeraren kan definiera egna paket

  `package` *`paketnamn`*`;`
- `Package` satsen måste vara första satsen i ett fil
- Alla klasser i filen läggs till det namngivna paketet
- Det får bara finnas en `package` sats i en fil
- Paketnamnet motsvarar katalognamnet där klassfilerna finns
- Det finns en s k ***environment variable*** `CLASSPATH` som lägger fast var systemet skall leta efter paket
- Vid namnkollision måste berörda klasser ***kvalificeras*** (specificeras med fullt namn)

Viktiga paket

- `java.applet`
  - javaprogram på websidor
- `java.awt`
  - grafikpaket - text, linjer, bilder
- `java.io`
  - skriva/läsa på filer, tangetbord, skärm
- `java.lang`
  - OBS! java.lang importeras automatiskt
- `java.net`
  - skicka och ta emot trafik på nätverket
- `java.util`
  - diverse användbara klasser, t.ex Date

Omdefiniera metoder 1

- Ärvda metoder kan ibland vara "olämpliga"
  - `ChangeSize()`, `draw()` och `erase()` t ex måste fungera olika för olika figurer

➨ En subklass får omdefiniera en ärvd metod

- I en omdefinition ges en metod en ***ny implementering***
- Måste dock ha ***samma signatur*** som superklassens metod

- ***signatur*** : metodens namn och parametrar (antal, typ och ordning)

Omdefiniera metoder 2

- ***Originalmetoden skuggas***, finns dock fortfarande tillgänglig genom `super` referensen

```
super.metod (parametrar);
```

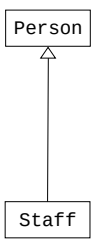
- ...på detta sätt kan originalmetoden utökas

- Se t.ex. metoden `toString` i `people2`

Omdefiniera metoder 3

```
/**
 * Return a string representation of this object.
 */
public String toString()    // redefined from "Object"
{
    return "Name: " + name + "\n" +
           "Year of birth: " + yearOfBirth + "\n";
}
```

```
/**
 * Return a string representation of this object.
 */
public String toString()   // redefined from "Person"
{
    return super.toString() +
           "Staff member\n" +
           "Room: " + room + "\n";
}
```



Omdefinition vs. Överlagring

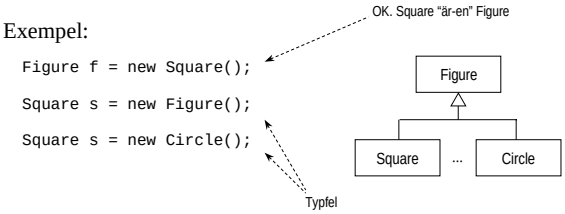
- Omdefinition
  - Två (eller fler) metoder med ***samma signatur***
  - Deklarerat i ***olika klasser***
  - "Originalversionen" deklarerat i en superklass
  - Omdefinitioner i olika subklasser
  - ➨ Metodval beror på klasstyp
- Överlagring (Overloading)
  - Två (eller fler) metoder med samma namn men ***olika signatur***
  - Får vara deklarerat i ***samma klass***
  - Konstruktorer ofta överlagrade
  - ➨ Metodval beror på parameterlistan
- Se klassen `Triangle` i `shapes2` exemplet

Arv och polymorfi

Polymorfi: Kan förekomma i många former.

- Referenser får referera till objekt av en subklass
- Exempel:

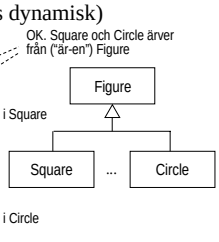
```
Figure f = new Square();
Square s = new Figure();
Square s = new Circle();
```



Dynamisk bindning

- Objektets typ (inte referensens) bestämmer vilken metod som exekveras
- Samma kod kan ge olika anrop vid olika tillfällen
- Kopplingen görs under körningen (dvs dynamisk)
- Exempel:

```
Figure f = new Square();
f.changeSize (100);
f = new Circle();
f.changeSize (100);
```



Type casting

- Gör om en typ till en annan
  - vid tilldelning

```
Figure f = new Circle();
…
Circle c = (Circle)f;
```

  - vid parameteröverföring

```
metodSomVillHaEnCircle( (Circle)f );
```

- Används bara när man är HELT säker på att referensen verkligen pekar på rätt typ
- Om f ovan pekar på en Square blir det fel !

Visibility

Variabel eller metod	Nås i samma klass	Nås i samma paket	Nås i subklass	Nås överallt
`private`	ja	nej	nej	nej
default	ja	ja	nej	nej
`protected`	ja	ja	ja	nej
`public`	ja	ja	ja	ja

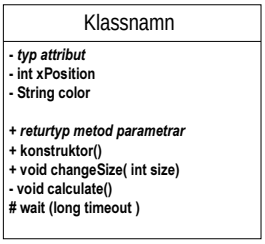
Visibility via arv

Variabel eller metod	Ärvs av subklass i samma paket	Ärvs av subklass överallt
`private`	nej	nej
default	ja	nej
`protected`	ja	nej
`public`	ja	ja

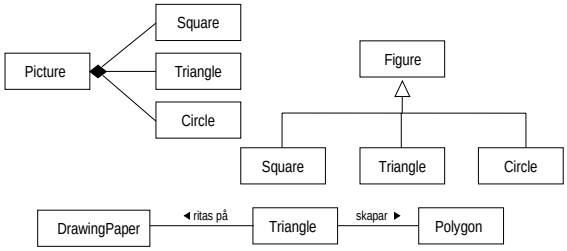
UML

- Notation för programsystem
- Nio olika diagramtyper ...
- Exempel på några olika typer av diagram
  - Klassdiagram
    - Aggregat/Komposition
    - Generalisering (Arv)
  - Sekvensdiagram
  - Statechart diagrams (state-diagram)
  - Aktivitetsdiagram
  - Use-case

Klassdiagram

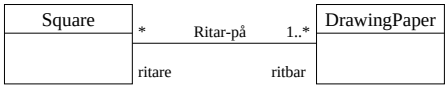


Klassdiagram - relationer



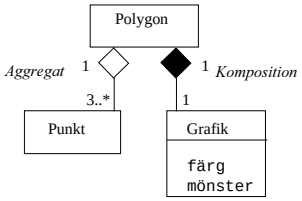
Association

- Strukturell relation
- betyder att object är 'ihopkopplade'
- Dubbelriktade som default
- Har 'cardinality'
- Klasserna kan ha roller



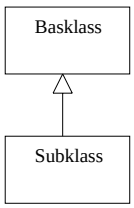
Aggregat, komposition

- *Part-of* eller *has*
- Komposition är starkare
  - Delarna delas inte med någon annan
  - Delarnas existens beror på omslutande klass:s existens



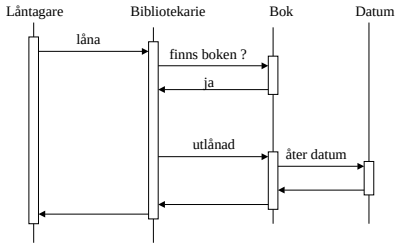
Generalisering (Arv)

- *Is-a* eller *är-ett*



Sekvensdiagram

- Beskriver scenarier
- Beskriver hur objekt samarbetar



Statechart (state-diagram)

- Beteende hos klasser
- Beskriver alla möjliga tillstånd och övergångar mellan dem