Läsanvisningar

Börja med att läsa kapitlen i kursboken, därefter resurserna på Internet och till sist själva lektionsmaterialet.

Kursboken (5:e upplagan): Kapitel 1.16, 6.14, 4.1, 4.5, Figur 10.4 - 10.8 med tillhörande text
Kursboken (6:e upplagan): Kapitel 1.16, 6.16, 4.1, 4.5, Figur 10.4 - 10.8 med tillhörande text
Kursboken (7:e upplagan): Kapitel 1.15, 6.17, 4.1, 4.5, Figur 10.4 - 10.8 med tillhörande text
Kursboken (8:e upplagan): Kapitel 1.15, 6.17, 4, Figur 10.4 - 10.8 med tillhörande text

Internet: The Java Tutorial, Trail: Deployment: - Java Applets
(http://docs.oracle.com/javase/tutorial/deployment/applet/index.html)

Allt under Getting started with Applets bör du läsa. När det gäller avsnittet Deploying an Applet behöver du inte fokusera på JNLP utan vi nöjer oss med att använda "Manually Coding Applet Tag, Launching Without JNLP".

Klassdiagram (ett provkapitel ur boken UML for Java Programmers, ISBN-13: 9780131428485)
(http://www.pearsonhighered.com/assets/hip/us/hip_us_pearsonhighered/samplechapter/0131428489.pdf)

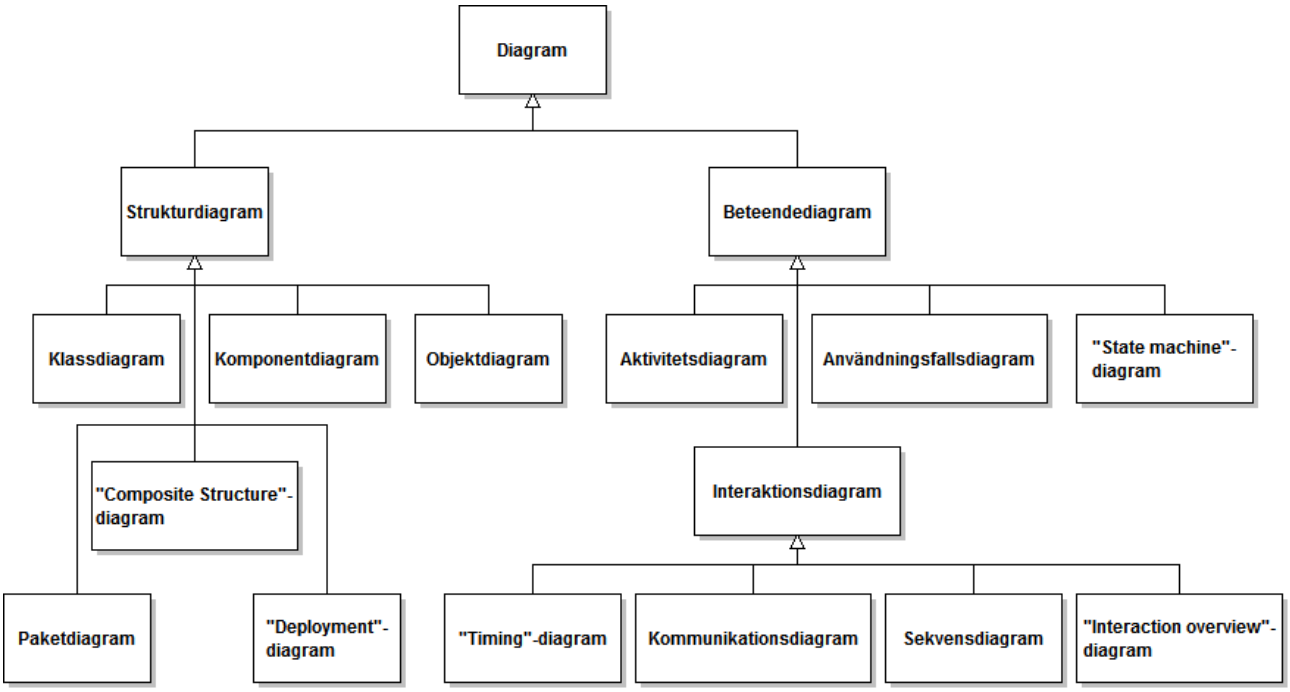
UML & Applet

I denna lektion tittar vi på vad UML är och mer specifikt hur vi skapar så kallade klassdiagram. Vi tar även en kort titt på javaapplikationer som körs i en webbläsare (applets).

Unified Modelling Language

Unified Modelling Language (UML) är ett språk (eller notation) där vi med olika typer av diagram kan visualisera hur ett objektorienterat system är uppbyggt. Med UML kan vi på ett relativt enkelt och snabbt sätt bygga en modell över vårt system och till exempel använda detta i vår kommunikation med beställaren av systemet (kunden). Ett annat exempel är att använda UML för att tydliggöra mellan olika projektgrupper, som jobbar med systemet, vad som ska göras och hur systemet ska byggas. 1997 släpptes UML version 1.0 och den senaste version är 2.4.1 (augusti 2011). Den formella specifikationen av UML är väldigt omfattande och finns att läsa på http://www.omg.org/spec/UML. Ytterligare information om UML finns på uml.org.

Det finns totalt 13 olika typer av diagram som kan användas i UML och dessa delas grovt in i struktur-, beteende, och interaktionsdiagram enligt figuren nedan (jag använder engelska namn där ingen bra svensk översättning finns).



Ett stort antal olika UML-element (grafiska figurer) används för att rita de olika diagrammen. Vissa UML-element är unika för ett specifikt diagram medan andra används av flera olika diagram. Att rita de olika diagrammen kan vi givetvis göra för hand på ett papper, men mer praktiskt är att använda något UML-modelleringsverktyg. Det finns mer eller mindre avancerade sådana program. Vissa erbjuder möjlighet att skapa diagram (klassdiagram) direkt från befintlig källkod (så kallad kodvisualisering). Tvärtom är även möjligt, det vill säga utifrån ett skapat klassdiagram så kan programmet generera källkoden (så kallad kodgenerering).

Att i detalj gå igenom vad UML är och hur de olika diagramtyperna skapas och används kan vi ägna en hel kurs åt och inget vi har tid för i denna kurs. En av de mest vanligt använda diagramtyperna är klassdiagram och det är denna diagramtyp vi ska ägna lektionen åt.

Klassdiagram

Ett klassdiagram är tänkt att beskriva den statiska strukturen ett programvarusystem har. Med detta menas att beskriva vilka klasser som ingår i systemet, vilka instansvariabler och metoder klasserna innehåller och hur de olika klasserna är relaterade tillvarandra. Nedan går vi igenom de viktigaste delarna i ett klassdiagram, som är relevanta när det gäller Java (vi tar inte upp allt som är möjligt enligt specifikationen).

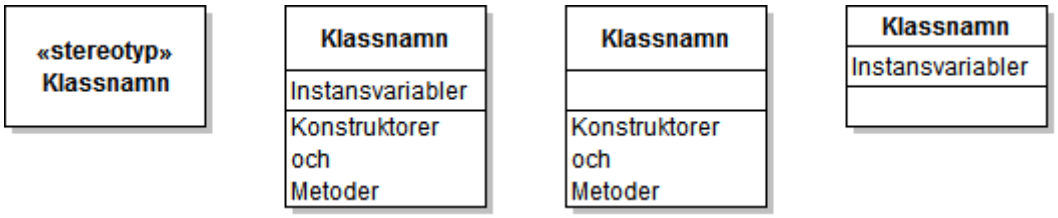
Klass

UML-elementet för att beskriva en klass i ett klassdiagram är en rektangel med tre områden separerade med linjer. Det översta området är obligatoriskt och innehåller namnet på klassen samt eventuellt en så kallad stereotyp. En stereotyp används för att förtydliga användningen av en klass (eller association) och skrivs inom dubbla vinkelcitationstecken « ». I specifikationen finns ett antal stereotyper definierade, men för Java används i stort sett enbart stereotyperna «interface» och «enumeration». Vi kan lägga till helt egna stereotyper om vi vill, men i så fall är det viktigt att de som ska använda ditt klassdiagram vet vad stereotypen betyder (om den inte är självförklarande). Stereotypen skrivs ovanför klassnamnet.

Den mellersta delen innehåller en lista över klassens instansvariabler (samt klassvariabler, konstanter m.m.) och den nedre delen innehåller en lista över klassens konstruktorer och metoder. I specifikationen talar man om attribut och operationer, men eftersom det är Java vi håller på med benämner jag det med instansvariabler och metoder i stället.

Vi får själv välja om vi vill ta med mellersta eller nedre delen när vi beskriver en klass. Om vi väljer att ta med någon av dem behöver vi inte heller lista alla klassens instansvariabler eller metoder. Vi kan välja ut endast en del av dem som vi anser vara viktiga för modellen.

När vi beskriver en klass i ett klassdiagram och vi endast tar med namnet på klassen räcker det att rita en rektangel i vilken klassnamnet skrivs ut. Om vi utöver klassnamnet väljer att ta med antingen listan över klassens instansvariabler eller konstruktorer/metoder måste vi även rita den andra delen (fast tom).



Klassens instansvariabler beskrivs enligt följande:

```
[synlighet] variabelnamn [: typ] [= defaultvärde]
```

De delar som omges av hakparenteserna [ ] är valfria att ta med. Vanligtvis väljer vi att ta med synlighet och typ för alla variabler samt även defaultvärden för konstanter.

Klassens konstruktorer och metoder beskrivs enligt följande:

```
[synlighet] metodnamn ([parameterlista]) [: returvärde]
```

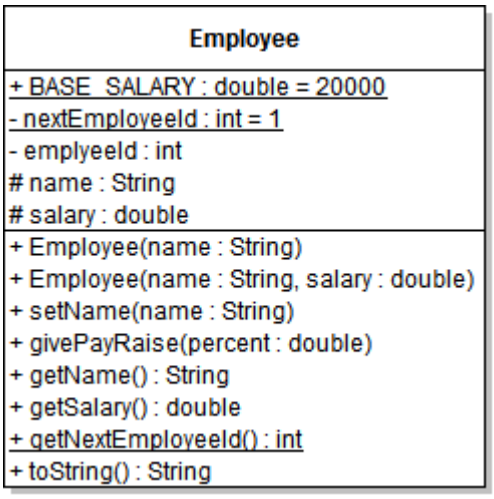
Även här är de delar som omges av hakparenteserna [ ] är valfria att ta med. Vanligt är dock att ta med alla dessa delar. Parameterlistan skrivs inom parenteser () enligt formatet parameternamn : parametertyp och separeras med ett kommatecken om flera parametrar används.

Synlighet anges med någon av följande symboler:

Symbol	Synlighet
+	Publik (public)
-	Privat (private)
#	Skyddad (protected)
~	Paket (package)

För metoder som saknar returvärde kan void användas alternativt att vi utesluter returvärde helt (vilket enligt specifikationen innebär just void).

Statiska variabler och metoder markeras genom understrykning. Namnet på konstanter skrivs med VERSALER.



Utifrån ovanstående klassdiagram får vi snabbt en klar bild över klassens ”struktur”. Det vill säga vilka instansvariabler som ska finnas, namnen på dessa och dess datatyp. Vilka konstruktorer och metoder som ska finnas och deras namn, eventuella parametrar och returvärden. Vi ser även vilken synlighet (eller åtkomst) variabler och metoder ska ha och vilka som ska deklareras som statiska. Det vi däremot inte ser är hur de olika metoderna ska implementeras (vilken kod de ska innehålla).

En översättning av ovanstående klassdiagram till javakod skulle kunna se ut så här:

```
public class Employee {
  public static final double BASE_SALARY = 20000;
  private static int nextEmployeeId = 1;
  private int employeeId;
  protected String name;
  protected double salary;

  public Employee(String name) {
    this(name, BASE_SALARY);
  }

  public Employee(String name, double salary) {
    this.name = name;
    this.salary = salary;
    employeeId = getNextEmployeeId();
  }

  public void setName(String name) {
    this.name = name;
  }

  public void givePayRaise(double percent) {
    salary = salary * (1 + percent / 100);
  }

  public String getName() {
    return name;
  }

  public double getSalary() {
    return salary;
  }
```

```
  public static int getNextEmployeeId() {
    return nextEmployeeId++;
  }

  @Override
  public String toString() {
    return employeeId + ", " + name + " (" + salary + ")";
  }
}
```

Relationer

Relationer används i klassdiagrammet för att visa att två eller flera klasser på något sätt har med varandra att göra. Totalt finns det fem olika typer av relationer mellan klasser.

- Association – En relation mellan två klasser där en klass har en instansvariabel av en annan klass och/eller tvärtom.
- Realisering – En relation mellan två klasser där en klass implementerar ett gränssnitt.
- Generalisering – En relation mellan två klasser där den ena klassen ärver den andra klassen.
- Inkapsling – En relation mellan två klasser på så sätt att en klass är deklarerad inuti en annan klass.
- Beroende – En relation mellan två klasser där en klass är beroende av en annan klass genom att första klassen använder den andra klassen vid något tillfälle utan att den är en instansvariabel. Ett beroende finns när en klass använder den andra klassen som en parameter eller lokal variabel i någon av dess metoder.

Av dessa relationer ska vi titta närmare på associationer, realisering och generalisering.

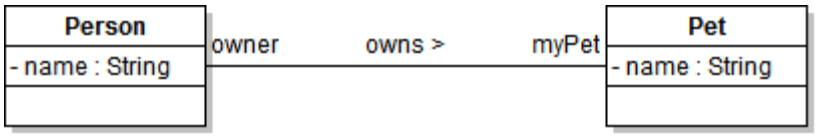
Association

En association beskriver som sagt en relation där en klass har en instansvariabel vars typ är en annan klass och på så sätt ”känner till” den andra klassen och kan anropa dess metoder. Associationer kan vara någon av dessa fyra typerna:

- Enkelriktad
- Dubbelriktad
- Aggregerad
- Reflexiv

I klassdiagrammet visar vi en association genom att rita en linje mellan de två klasserna som ingår i associationen. Beroende på vilken typ av association det är frågan om dekorerar vi linjens ändpunkter på olika sätt.

En association ges normalt ett namn för att förtydliga vad relationen innebär. Namnet skrivs ovanför linjen och vi kan lägga till en pil (fylld triangel) vid namnet som pekar i den riktning namnet ska utläsas. Vid linjens ändpunkter kan vi skriva ut ”roller” som tydligare visar vilken roll den ena klassen har i den andra. I klassdiagram kan vi ofta se att rollnamnen är det samma som namnet på instansvariabeln i klassen.



I klassdiagrammet ovan finns en association mellan klassen Person och Pet. Associationens namn är owns och pilen (som egentligen ska vara en fylld triangel) innebär att associationen ska utläsas som att en person äger ett husdjur. Rollnamnen visar att klassen Person har en instansvariabel myPet som är av typen Pet. På samma sätt har klassen Pet en instansvariabel owner av typen Person.

Som du ser ovan återfinns inte de instansvariabler en association ger i klassens lista över instansvariabler. Där listar vi enbart de instansvariabler vars typ inte finns som en klass i klassdiagrammet. När vi översätter klassdiagrammet till källkod finns de givetvis med:

```
public class Person {
  private String name;
  private Pet myPet;
}

public class Pet {
  private String name;
  private Person owner;
}
```

Multiplicitet

Utöver ett rollnamn kan vi även ange en multiplicitet vid en associations ändar. Multipliciteten anger hur många objekt/instanser den ena klassen har av den andra. Multipliciteten anges som ett heltal, ett intervall av heltal, symbolen * eller en kombination av dessa. I klassdiagrammet ovan har ingen multiplicitet angetts vilket har samma innebörd som att skriva 1. I tabellen nedan ges några olika exempel på multipliciteter och deras betydelse.

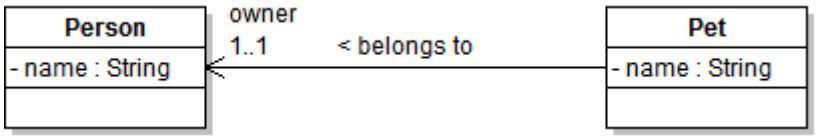
Multiplicitet	Betydelse
1 1..1	En och endast en. Alla personer måste ha ett husdjur. En person kan ha varken mer eller mindre än ett husdjur.
0..1	Noll eller en. En person behöver inte ha något husdjur och kan max ha ett husdjur.
1..*	En till många. En person måste ha minst ett husdjur, men kan ha hur många husdjur som helst.
* 0..*	Noll till många. En person behöver inte ha något husdjur, men kan ha ett eller hur många husdjur som helst.
2	Exakt 2. En person måste ha exakt två husdjur, varken fler eller färre.
n	Exakt n. En person måste ha exakt n stycken husdjur (där n är ett heltal större än 0).
3..9	Tre till nio. En person måste ha minst tre husdjur och kan ha upp till nio husdjur.
0..3, 5, 9..*	En person kan ha noll, ett, två, tre, fem, nio eller fler husdjur.

När multipliciteten innehåller siffran 0 (noll) kan objektet referera till null. Annars inte.

Enkelriktad association

En enkelriktad association innebär att det endast är den ena klassen, som ingår i associationen, som har en instansvariabel vars typ är av den andra klassen. För att i klassdiagrammet visa vilken klass det är som har instansvariabeln ritar vi ut en navigeringspil i linjens ena ändpunkt. Vi ritar pilen

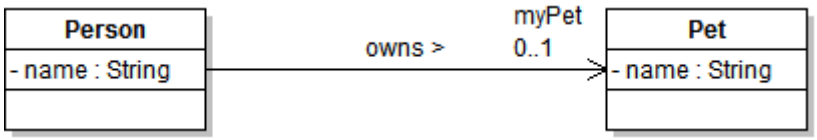
bredvid den klass som blir en instansvariabel i den andra klassen. Vid enkelriktade associationer skriver vi endast ut rollnamn och multiplicitet vid den klass navigeringspilen pekar på. Även namnet på associationen brukar vi låta ”peka” i samma riktning som navigeringspilen, men det är inget krav.



I klassdiagrammet ovan ser vi ett exempel på en enkelriktad association som säger oss att ett husdjur tillhör en och endast en person (ägaren). Med andra ord om vi har ett objekt av typen Pet måste den innehålla ett objekt av typen Person. En person har själv ingen vetskap om sitt husdjur. Ett sätt att implementera detta på är att låta Pet ha en konstruktor som tar en Person som parameter.

```
public class Person {
  private String name;
}

public class Pet {
  private String name;
  private Person owner;

  public Pet(String name, Person owner) {
    this.name = name;
    this.owner = owner;
  }
}
```

Ytterligare ett exempel på en enkelriktad association kommer här:



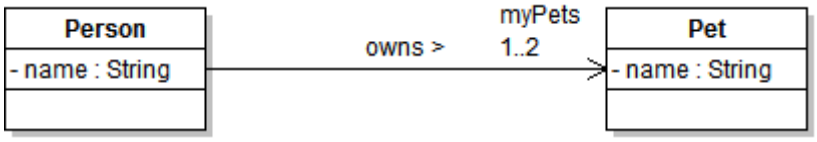
I exemplet ovan kan en person äga inget eller högst ett husdjur. Här är det personen som känner till sitt husdjur medan husdjuret inte har någon kännedom om sin ägare. Ett sätt att implementera detta på är:

```
public class Person {
  private String name;
  private Pet myPet;

  public Person(String name) {
    this(name, null);
  }

  public Person(String name, Pet pet) {
    this.name = name;
    this.myPet = pet;
  }
}

public class Pet {
  private String name;
```

```
}
```

Här är det möjligt att skapa ett objekt av klassen Person utan att samtidigt ange ett husdjur.

Hur gör vi när multipliciteten är högre än ett? Det vill säga om en person till exempel kan ha ett till två husdjur?



Här finns inga direkta regler på vilken datatyp vi ska använda. Vi kan till exempel välja att använda två instansvariabler av typen Pet,

```
public class Person {
  private String name;
  private Pet myPets1;
  private Pet myPets2;
}
```

en array,

```
public class Person {
  private final int MAX_PETS = 2;
  private String name;
  private Pet[] myPets = new Pet[MAX_PETS];
}
```

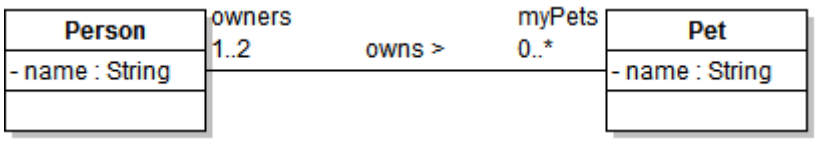
eller en lista av något slag

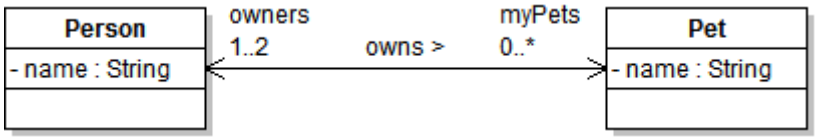
```
public class Person {
  private final int MAX_PETS = 2;
  private String name;
  private ArrayList<Pet> myPets = new ArrayList<Pet>();
}
```

Konstanten MAX_PETS ovan kan vi använda för att kontrollera så att inga fler husdjur än två läggs in i listan.

Dubbelriktad association

En dubbelriktad association innebär att båda klasserna som ingår i associationen har en instansvariabel vars typ är av den andra klassen. En dubbelriktad association ritas normalt med navigeringspilar i båda ändarna, men det förekommer även att inga navigeringspilar alls ritas ut.





Båda klassdiagrammen ovan säger samma sak. Nämligen att en person kan äga noll till många husdjur och att varje husdjur har en eller två ägare (matte och husse). Återigen finns det flera sätt vi kan översätta klassdiagrammet till kod och säkerställa att multipliciteterna följs. Här är ett sätt:

```
public class Person {
  private String name;
  private ArrayList<Pet> myPets = new ArrayList<Pet>();

  public Person(String name) {
    this.name = name;
  }

  public void addPet(Pet pet) {
    myPets.add(pet);
  }
}

public class Pet {
  private final int MAX_OWNERS = 2;
  private String name;
  private ArrayList<Person> owners = new ArrayList<Person>();

  public Pet(String name, Person owner) {
    this.name = name;
    addOwner(owner);
  }

  public boolean addOwner(Person owner) {
    if (owners.size() >= MAX_OWNERS)) {
      return false;
    }

    owners.add(person);
    return true;
  }
}
```

Eftersom en person inte behöver ha något husdjur behöver konstruktorn i Person inte ta något Pet som parameter. Det omvända gäller däremot. Ett husdjur har minst en ägare och därför låter vi konstruktorn i Pet ta ett Person-objekt som parameter.

Aggregerade associationer

Aggregering är en form av association där relationen mellan två klasser kan beskrivas som att ena klassen utgör en del av den andra klassen. Aggregering ritar vi genom att dekorera linjens ena ändpunkt med en ofylld romb. Romben placerar vi vid den klass som utgör ägaren (eller helheten). Vid den andra ändan ritar vi en navigeringspil.

Komposition är en variant av aggregation, men där ägarskapet är starkare. Ägaren sägs vara ansvarig för dess delar och utan dess delar finns egentligen ingen mening att ha kvar ägaren. Komposition ritas på samma sätt som aggregering med den skillnaden att en fylld romb används.

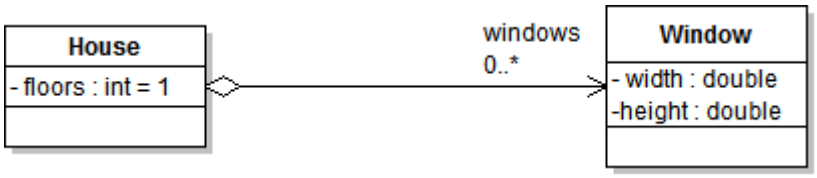
När det gäller komposition säger specifikationen att en del aldrig kan vara med i flera kompositioner utan enbart en. Detta till skillnad från aggregering där en del kan ingå i flera aggregat. Vad som egentligen är aggregering och komposition och när man ska använda vad kan man se lite olika exempel på. Dels beror det på vilken definition man använder, men även vilket sammanhang de används i. Specifikationen är väldigt otydlig på framför allt komposition och lämnar utrymme till olika tolkningar.

”Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite.”

”... composite indicates that the property is aggregated compositely, i.e., the composite object has responsibility for the existence and storage of the composed objects (parts).”

Vad innebär att delarna ”normalt” tas bort när helheten tas bort? Om ägaren har ansvar för dess delar, och delen kan tas bort, vem har då ansvaret? Som tur är kan vi alltid ersätta båda dessa typer av associationer (aggregering och komposition) med en helt vanlig association i stället.

Hur som helst. Nedan visas exempel på både aggregation och komposition och ett vanligt sätt att implementera detta i kod. Vi börjar med aggregering.



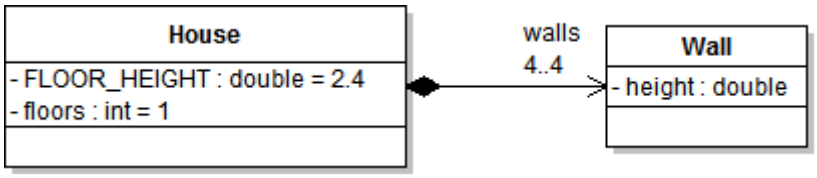
I exemplet ovan utgör fönster delar av ett hus. Ett hus kan ha noll till många fönster. Om vi tar bort ett fönster från huset kan huset fortfarande fortsätta att existera (blir bara lite kallt inomhus på vintern). En översättning till kod kan se ut så här:

```
public class Window {
  private double width;
  private double height;

  public Window(double width, double height) {
    this.width = width;
    this.height = height;
  }
}

public class House {
  private int floors = 1;
  private ArrayList<Window> windows = new ArrayList<Window>();

  public House() {
    this(1);
  }

  public House(int floors) {
    this.floors = floors;
  }
```

```
  public void addWindow(Window window) {
    windows.add(window);
  }
}

public class Test {
  public static void main(String[] args) {
    Window w1 = new Window(3, 4);
    Window w2 = new Window(4, 4);
    House myHouse = new House(2);
    myHouse.addWindow(w1);
    myHouse.addWindow(w2);
    myHouse = null;
  }
}
```

Att notera är att vi kan skapa fönstren oberoende av huset och sen lägga till våra fönster i huset allt eftersom. När vi ”river” huset (myHouse = null) fortsätter fönstren att existera.

Exemplet nedan visar komposition där en vägg utgör en del i ett hus. Ett hus har exakt fyra väggar och utan väggar kan ett hus inte existera.



En översättning till kod kan se ut så här:

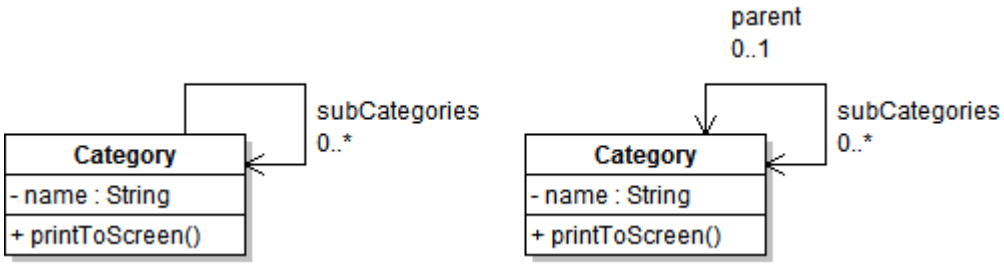
```
public class Wall {
  private double height;

  public Wall(double height) {
    this.height = height;
  }
}

public class House {
  private final double FLOOR_HEIGHT = 2.4;
  private int floors = 1;
  private Wall northWall;
  private Wall southWall;
  private Wall eastWall;
  private Wall westWall;

  public House() {
    this(1);
  }

  public House(int floors) {
    this.floors = floors;
    northWall = new Wall(FLOOR_HEIGHT * floors);
    southWall = new Wall(FLOOR_HEIGHT * floors);
    eastWall = new Wall(FLOOR_HEIGHT * floors);
    westWall = new Wall(FLOOR_HEIGHT * floors);
  }
```

```
}

public class Test {
  public static void main(String[] args) {
    House myHouse = new House(2);
    myHouse = null;
  }
}
```

Skillnaden mot aggregering syns här genom att det är huset själv som ansvarar för skapandet av dess väggar. När vi river huset (myHouse = null) kommer även husets väggar att rivas. De städas nämligen undan automatisk av Javas skräpsamlare. I andra programmeringsspråk, som till exempel C++, hade vi varit tvungna att själva se till att väggarna städas undan.

Reflexiva associationer

I alla tidigare exemplen om associationer har vi visat på relationer mellan två olika klasser. Det finns tillfällen då en klass har en relation med sig själv. Det vill säga i klassen finns det en instansvariabel vars typ är klassen själv. I klassdiagrammet ritar vi då en linje som både startar och slutar i sig själv. För reflexiva associationer kan vi använda allt vi nämnt ovan om namn på associationen, multiplicitet, enkel- och dubbelriktad association och rollnamn m.m.



I klassdiagrammet ovan ges exempel på en reflexiv association där en kategori kan ha ingen, en eller många underkategorier. Skillnaden mellan de båda varianterna är att i det högra exemplet känner en kategori till dels dess förälderkategori och dels dess underkategorier. En kategori kan ha ingen eller en förälderkategori. I det vänstra exemplet känner en kategori enbart till sina underkategorier.

En översättning av det vänstra exemplet kan se ut så här (utan metoden printToScreen):

```
public class Category {
  private String name;
  private List<Category> subCategories;

  public Category(String name) {
    this.name = name;
    subCategories = new ArrayList<Category>();
  }

  public void add(Category category) {
    subCategories.add(category);
  }
}
```

Då en kategori inte känner till sin förälderkategori utan enbart dess underkategorier räcker det här att ha en lista för att lagra underkategorierna. Vidare behöver vi, när vi skapar ett objekt av Category,

inte skicka med någon kategori eftersom en kategori inte behöver ha några underkategorier. Om multipliciteten däremot hade varit 1..* hade vi varit tvungna att använda följande konstruktor:

```
public Category(String name, Category subCategory) {
  this.name = name;
  subCategories = new ArrayList<Category>();
  add(subCategory);
}
```

En översättning av det högra exemplet kan se ut så här (utan metoden printToScreen):

```
public class Category {
  private String name;
  private Category parent;
  private List<Category> subCategories;

  public Category(String name) {
    this(name, null);
  }
  public Category(String name, Category parent) {
    this.name = name;
    this.parent = parent;
    subCategories = new ArrayList<Category>();
  }

  public void add(Category category) {
    subCategories.add(category);
  }
}
```

Nu när en kategori kan ha en förälderkategori och underkategorier måste vi ha instansvariabler dels för förälderkategorin och dels för underkategorierna. Då en kategori inte behöver ha någon förälderkategori har vi en konstruktor som enbart tar namnet på kategorin. Instansvariabeln parent sätts då att referera till null. Vi har även en konstruktor som både tar namnet på kategorin samt vilken förälderkategori den har.

Metoden printToScreen i exemplen ovan kan implementeras på följande sätt:

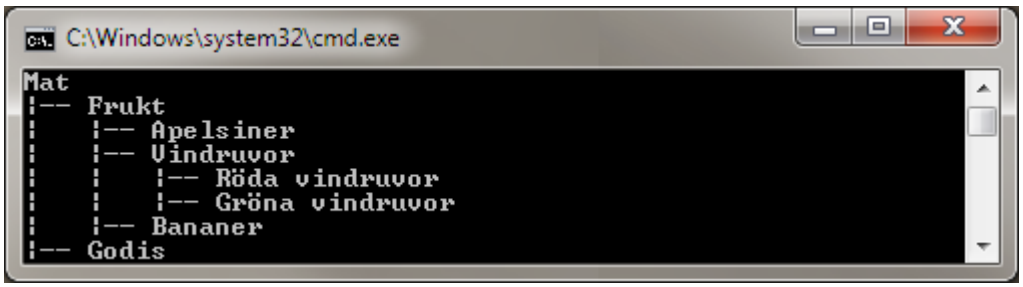
```
public void printToScreen() {
  printToScreen(0);
}

private void printToScreen(int level) {
  if (level > 0) {
    for (int i = 1; i < level; i++) {
      System.out.print("|   ");
    }
    System.out.print("|-- ");
  }

  System.out.println(name);

  if (subCategories.size() > 0) {
    level++;
    for (Category c : subCategories) {
      c.printToScreen(level);
    }
  }
```

```
}
```

I korta drag kommer metoderna ovan att set till så att varje kategoris namn skrivs ut på skärmen. Är det frågan om en underkategori skrivs detta namn ut med en indentering. Följande exempel:

```
Category food = new Category("Mat");
Category fruit = new Category("Frukt", food);
Category grapes = new Category("Vindruvor", fruit);
grapes.add(new Category("Röda vindruvor", grapes));
grapes.add(new Category("Gröna vindruvor", grapes));
fruit.add(new Category("Apelsiner", fruit));
fruit.add(grapes);
fruit.add(new Category("Bananer", fruit));
food.add(fruit);
food.add(new Category("Godis", food));
food.printToScreen();
```

skulle ge följande utskrift:

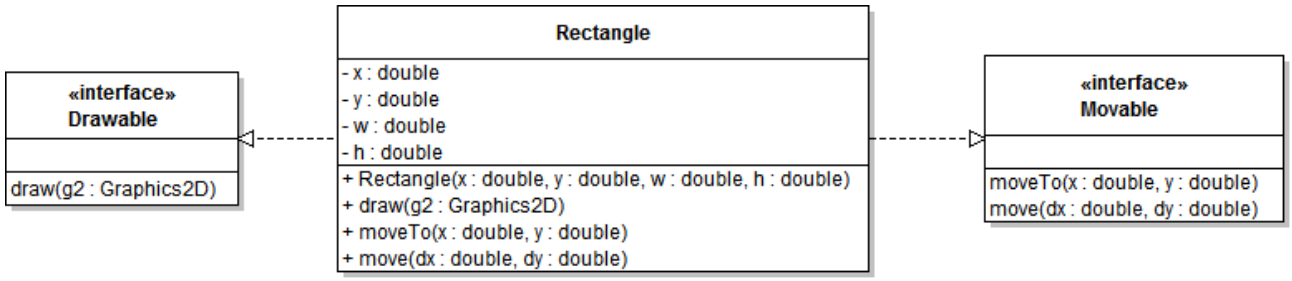


I exemplen ovan har vi gjort en väldigt förenklad översättning till kod. I en skarp situation måste vi även ta hänsyn till borttagning av förälderkategorier och underkategorier, insättning av nya kategorier och kontroll av dubbletter. Vad ska hända om vi anger en ny förälderkategori som har egna underkategorier? Vad ska hända om vi lägger till en underkategori och denna kategori redan är en underkategori i kategorin? Prova i exemplet som följer med lektionen att sist i testklassen lägga till food.add(food) och sen anropa printToScreen().

Realisering

Denna typ av relation används för att visa att en klass implementerar ett gränssnitt. I klassdiagrammet används samma UML-element för gränssnitt som för en vanlig klass. Dock lägger vi till stereotypen «interface» ovanför namnet. Detta innebär då att alla metoder i klassen (gränssnittet) ska ses som publika och abstrakta (public abstract) och att endast konstanter (public static final) kan användas.

För att visa att en klass implementerar ett gränssnitt ritar vi en streckad linje med en pil som pekar från klassen mot gränssnittet den implementerar.

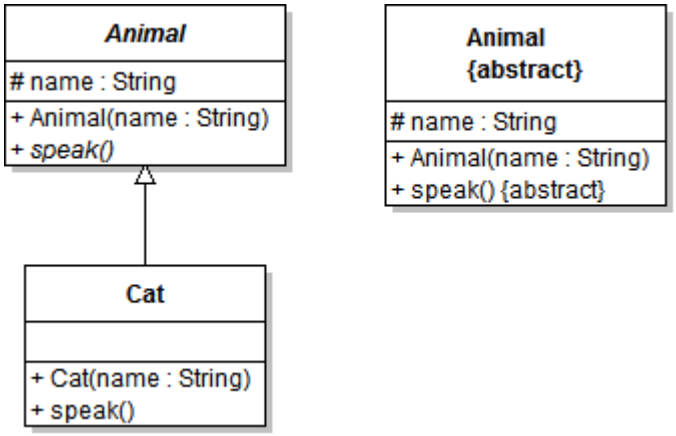


I klassdiagrammet ovan har vi de två gränssnitten Drawable och Movable som klassen Rectangle implementerar. Gränssnittet Drawable definierar att något är ”ritbart” och har en metod draw som använder ett Graphics2D-objekt för att rita på skärmen. Gränssnittet Movable definierar att något är ”flyttbart” och har metoderna moveTo och move. Den första metoden är tänkt att användas för att flytta något till positionen som anges av (x, y). Den andra metoden är tänkt att användas för att flytta från befintlig position till en ny position genom att lägga till dx och dy till den befintliga positionen. Notera att vi för gränssnitten inte behöver ange synlighet eller markera metoden som abstrakt (eftersom de är gränssnitt). En översättning till Javakod ser ut så här:

```
public interface Drawable {
  void draw(java.awt.Graphics2D g2);
}

public interface Movable {
  void moveTo(double x, double y);
  void move(double dx, double dy);
}

public class Rectangle implements Drawable, Movable {
  private double x, y, w, h;

  public Rectangle(double x, double y, double w, double h) {
    moveTo(x, y);
    this.w = w;
    this.h = h;
  }

  @Override
  public void draw(Graphics2D g2) {
    Rectangle2D r = new Rectangle2D.Double(x, y, w, h);
    g2.draw(r);
  }

  @Override
  public void moveTo(double x, double y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public void move(double dx, double dy) {
    this.x += dx;
    this.y += dy;
  }
}
```

Generalisering

Generalisering, eller arv, visar vi i klassdiagrammet genom att rita en heldragen linje med en pil som pekar från subklassen mot superklassen. Det är väldigt vanligt att man i klassdiagram ritar arv vertikalt. Det vill säga superklassen längst upp och därefter subklasser, i ett eller flera led, under superklassen. Klasser och metoder som är abstrakta markeras i ett klassdiagram genom att skriva klassnamn och metodnamn med *kursiv stil* alternativt att vi skriver {abstract} bredvid namnet. I klassdiagrammet nedan ser exempel på båda dessa sätt att ange att en klass/metod är abstrakt.



I klassdiagrammet ovan ser vi att klassen Animal är abstrakt och innehåller en abstrakt metod speak. Vidare ser vi att klassen Cat ärver klassen Animal. Cat är en subklass till Animal och Animal är i sin tur en superklass. En översättning till Javakod kan se ut så här:

```
public abstract class Animal {
  protected String name;

  public Animal(String name) {
    this.name = name;
  }

  abstract void speak();
}

public class Cat extends Animal {
  public Cat(String name) {
    super(name);
  }

  @Override
  public void speak() {
    System.out.println(name + " the cat says \"Meow\"");
  }
}
```

Vi har nu gått igenom grunderna i klassdiagram och sett exempel på hur dessa kan översättas till kod i Java. Vi har inte gått igenom allt som berör klassdiagram utan enbart sådant vi anser vara viktigt. En hel del delar och detaljer har vi utelämnat men trots det bör du ha fått tillräckligt mycket på fötterna för att själv kunna skapa helt egna nya klassdiagram och översätta befintliga till kod.

Applets

En applet är en ett javaprogram som exekveras eller bäddas in i ett annat program, vanligtvis en webbläsare. För att webbläsaren ska kunna exekvera en applet krävs att webbläsaren har ett Java-

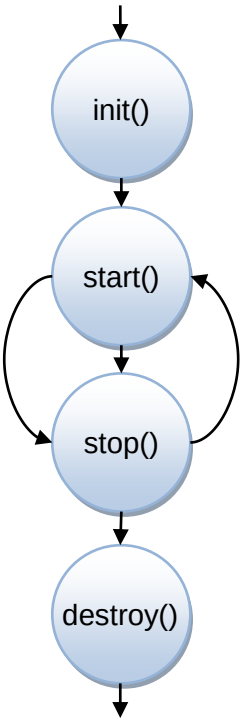
plugin installerat. När Java Runtime Enviroment (JRE) installeras på en användarens dator följer det med plugin till de mest vanliga webbläsarna.

För att skapa en Applet skriver vi en klass som ärver från Applet (i paketet java.applet) eller JApplet (i paketet javax.swing). JApplet ärver från Applet som i sin tur ärver Container. Med andra ord är en applet en behållare på samma sätt som JPanel och JFrame. Vi kan alltså ange vilken layouthanterare en applet ska använda, lägga till grafiska komponenter av olika slag och rita frihandsgrafik i en applet. Det vill säga precis på samma sätt som en vanlig applikation i Java (applikation). Vi kompilerar även en applet på samma sätt som en applikation.

En skillnad mellan en applet och en applikation är hur de startas och exekveras. En applikation startar vi genom att ge kommandot java NamnPåKlass vilket leder till att main-metoden i den klassen exekveras. En applet laddas ned, skapas och startas däremot av webbläsaren. När en användare besöker en hemsida, på vilken en applet finns, laddas den ned och därefter skapas en instans av appleten (den klass som ärver Applet/JApplet). Därefter anropas en metod med namnet init följt av ett anrop till metoden start. Allt detta görs automatiskt och är ingenting vi har någon kontroll över.

En applets livscykel

I klassen Applet finns fyra metoder som är knutna till en applets livscykel, nämligen init, start, stop och destroy. Dessa metoder anropas automatiskt i samband med att en applet ska startas, köras och avslutas. När vi utvecklar en applet har vi möjlighet att överskugga en eller flera av dessa metoder.



Metoden init anropas endast en gång under en applets livscykel. Anropet sker i samband med att en applet laddas färdigt och i denna metod lägger vi kod som vi normalt annars hade valt att lägga i en konstruktor. Det vill säga kod för att initiera instansvariabler och kod för att skapa det grafiska användargränssnittet. Om appleten använder så kallade trådar (threads, vilket inte ingår i denna kurs) är det i init vi skapar dessa. För att en applet ska laddas och visas snabbt ska vi inte lägga kod som tar lång tid att köra i denna metod. Behöver appleten ladda ner resurser som bilder, ljud och dokument med mera ska detta göras i en separat tråd.
Metoden start anropas av webbläsaren för att säga till appleten att det är dags att starta exekveringen. Det första anropet till denna metod sker direkt efter init. Metoden kan anropas flera gånger under en applets livscykel. I de allra flesta fallen behöver vi inte överskugga denna metod. I en applet som använder trådar, vilket används i många spel eller applets som spelar upp ljud eller en animation, är denna metod ett lämpligt ställa att starta trådarna.
Metoden stop anropas av webbläsaren när en applet behöver stoppas. Inte heller denna metod behöver vi normalt överskuggas, men använder vi trådar är det här vi bör stoppa trådarna (pausa spelat eller animationen). En applet som är stoppad ska inte förbruka några resurser (belasta CPU).
Metoden destroy anropas enbart en gång under en applets livscykel. Anropet sker just innan det är dags att avsluta och förstöra appleten. Här måste vi se till att släppa eventuella resurser vi förbrukar som inte släpptes i stop.

Metoderna start och stop kan som sagt anropas flera gånger under en applets livscykel, men i praktiken sker det sällan. Om vi i en webbläsare surfar till en annan sida kommer appleten att

stoppas och förstöras (metoderna stop och destroy anropas). Om vi därefter går tillbaka till sidan med appleten vi nyss lämnade kommer appleten att laddas om på nytt (metoderna inti och start anropas).

Vi har hittills endast nämnt en webbläsare som kör en applet. Ett annat program som också kan köra applets är appletviewer som följer med utvecklingsmiljön för java (JDK). I appletviewer är det möjligt att starta och stoppa en applet flera gånger utan att den förstörs. Under utvecklingsarbetet med en applet kan det därför vara en fördel att använda appletviewer.

Nedan följer nu ett första exempel på en applet.

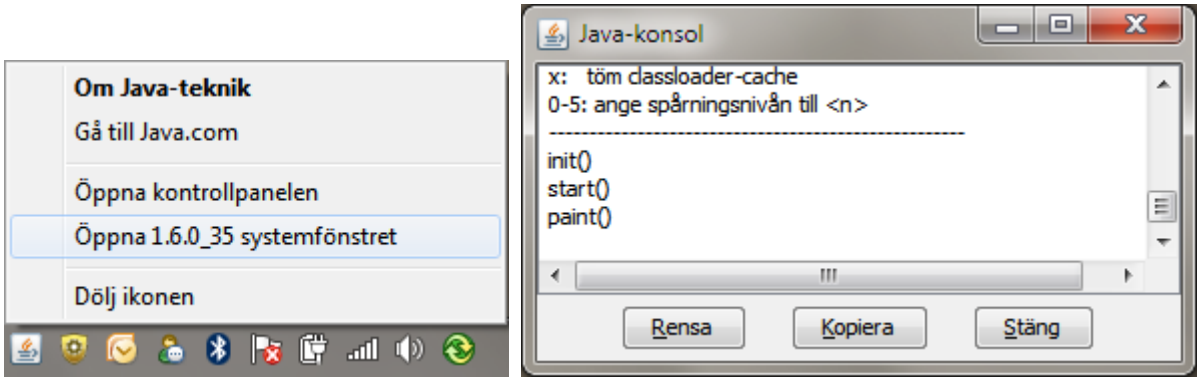
```
import javax.swing.*;
import java.awt.*;

public class LifeCycleApplet extends JApplet {
  private JTextArea output;

  public void init() {
    output = new JTextArea();
    add(new JScrollPane(output), BorderLayout.CENTER);
    showMessage("init()");
  }

  public void start() {
    showMessage("start()");
  }

  public void stop() {
    showMessage("stop()");
  }

  public void destroy() {
    showMessage("destroy()");
  }

  private void showMessage(String message) {
    output.append(message+"\n");
    System.out.println(message);
  }

  public void paint(Graphics g) {
    showMessage("paint()");
    g.drawRect(0, 0, getWidth()-1, getHeight()-1);
  }
}
```

Vi skapar en klass som ärver JApplet och i dess init-metod bygger vi ett enkelt GUI bestående av en JTextArea. Notera att en JApplet som standard använder BorderLayout som layouthanterare varför vi måste specificera i vilket område vi ska lägga till textrutan (en klass som ärver Applet använder FlowLayout som standard). Givetvis kan vi byta layouthanterare genom ett anrop till metoden setLayout.

Tanken med exemplet ovan är att demonstrera de fyra metoderna som har med livscykeln att göra. I dessa metoder anropar vi showMessage och skickar med namnet på den metod som gör anropet. I showMessage skriver vi ut meddelandet dels i textrutan men även med System.out.println. När vi i en applet använder System.out kommer utskrifterna att hamna i ett konsolfönster. Använder vi en

appletviewer hamnar utskrifterna i samma konsolfönster som startade appletviewer. I en webbläsare kommer vi åt utskrifterna genom en Java-konsol som följer med i det plugin webbläsaren använder för att köra appleten.

I Windows kan du öppna Java-konsolen genom att högerklicka på Java-ikonen i aktivitetsfältet och välja Öppna systemfönstret.



En metod vi hittills inte nämnt är paint. Denna kan vi överskugga om vi vill rita frihandsgrafik direkt i en applets yta. Som du ser i bilden till höger ovan kommer vår applet i exemplet att i tur och ordning anropa metoderna init, start och paint när den startas. Även om det går att rita direkt i appletens yta tycker jag vi ska använda en panel (JPanel eller Panel) rita på.

Visa en applet på hemsidan

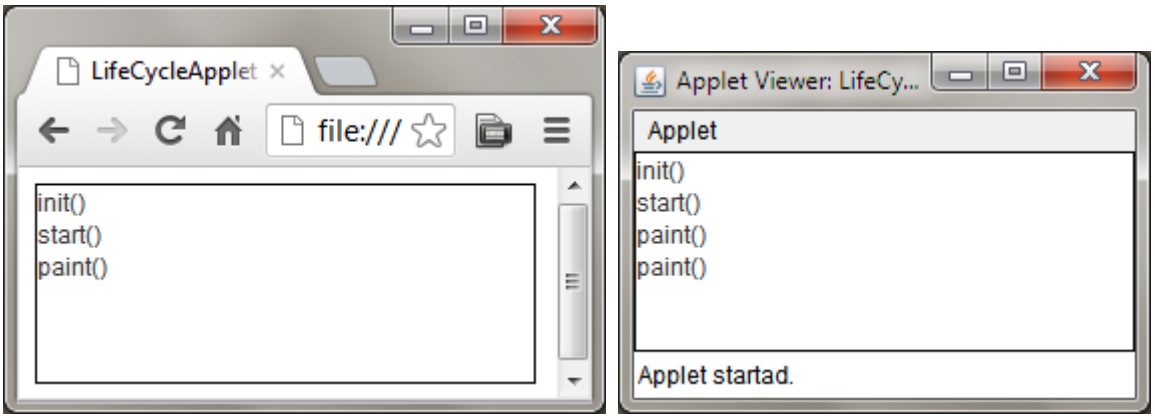
För att starta en applet behöver vi en HTML-fil i vilken vi använder taggen <applet> för att specificera vilken applet som ska visas på sidan. Förutom att ange vilken applet som ska visas kan vi med taggen även ange storleken på den yta appleten ska visas.

```
<html>
  <body>
    <applet code="LifeCycleApplet" width="250" height="100"></applet>
  </body>
</html>
```

Ovanstående kod säger åt appletviewer eller webbläsaren att ladda appleten LifeCycleApplet, som ligger i samma katalog som html-filen, och sätter storleken på appleten till 250 pixlar bred och 100 pixlar hög. Ligger både LifeCycleApplet.class och html-filen ovan (anta vi sparade den som LifeCycleApplet.html) i samma katalog på hårddisken eller på en webbserver kan vi starta appleten genom att antingen öppna html-filen i webbläsaren eller genom att i ett kommandofönster skriva:

```
appletviewer LifeCycleApplet.html
```

Bilderna nedan visar hur det ser ut i en webbläsare (vänster) och i appletviewer (höger).



I html-filen använde vi tre attribut till taggen applet, nämligen code för att specificera vilken applet som ska visas samt width och height för att specificera bredd och höjd på den yta appleten ska ha på sidan. Tabellen nedan visar alla attribut vi kan använda i taggen applet.

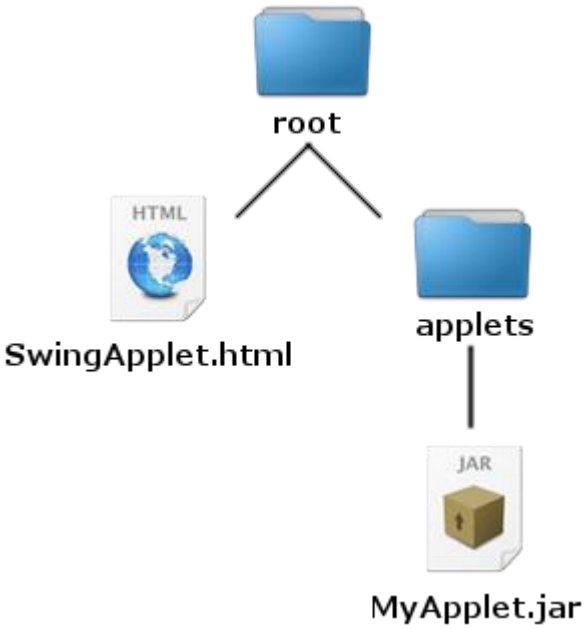
Attribut	Värde	Beskrivning
align	left, right, top, bottom, middle, baseline	Anger appletens placering i förhållande till omgivande innehåll på sidan (till exempel text och bilder med mera).
alt	Strängvärde	En alternativ text som visas i webbläsaren om den inte kan köra applets.
archive	Jar-fil	En lista över en eller flera jar-filer, separerade med ett kommatecken, som innehåller klasser eller andra resurser (som bilder). Den applet som ska köras kan finnas i en Jar-fil.
code	Class-fil	Namnet på den applet som ska köras. Ange hela namnet enligt ditt.paket.NamnPåApplet.
codebase	Strängvärde	En sökväg, relativt den katalog html-filen ligger i, till var applet- och jar-filerna finns. Anges ingen codebase används som standarvärde samma katalog som html-filen.
height	Heltal	Hur många pixlar bred appletens initiala yta ska vara (kan ändras i efterhand i appletviewer, men inte i en webbläsare).
hspace	Heltal	Hur många pixlar bred marginalen ska vara ovanför och nedanför appleten till omgivande innehåll på sidan.
name	Strängvärde	Ett namn på appleten. Kan användas för att kommunicera mellan flera olika applets på samma html-sida.
vspace	Heltal	Hur många pixlar bred marginalen ska vara till vänster och höger om appleten till omgivande innehåll på sidan.
width	Heltal	Hur många pixlar bred appletens initiala yta ska vara (se height).

När vi har en applet som består av flera klasser och andra resurser bör dessa samlas i en eller flera jar-filer. Anledningen är dels att den totala storleken på alla filer blir mindre och att jar-filer ”förladdas” i stället för att laddas allteftersom de används. Följande html-fil laddar en applet som ligger i en jar-fil. Jar-filen ligger i en underkatalog till katalogen i vilken html-filen ligger. Underkatalogen specificeras i attributet codebase.

```
<html>
  <body>
    <applet code="SwingApplet" width="300" height="250"
      codebase="applets" archive="MyApplet.jar">
```

```
    </applet>
  </body>
</html>
```

På användarens eller webbserverns hårddisk skulle filerna lagras i en hierarki enligt bilden nedan.



Applikation till applet

Vi har hittills använt JFrame för att skapa grafiska användargränssnitt (med swing). JFrame ärver i flera led klassen Container (i paketet java.awt) vilket även JApplet gör. Båda dessa klasser är behållare för andra komponenter och de GUI vi kan skapa i en JFrame kan vi med andra ord även skapa i en JApplet.

Även om vi kan skapa GUI på samma sätt i en JFrame och en JApplet finns det skillnader i vad en applikation och en applet kan göra. Det finns signerade och osignerade applets. En applet som inte har signerats är osignerad och en sådan applet körs i en skyddad miljö (sandlåda) på användarens dator och har vissa begränsningar i vad den får göra. Detta för att skydda användaren mot skadliga applets som kan laddas ned och köras på användarens dator utan att personen vet om det.

En osignerad applet får till exempel inte läsa eller spara filer lagrade på användarens hårddisk, ansluta till andra servrar annat än till den server appleten finns på eller läsa vissa systemegenskaper som till exempel vilken användarens hemkatalog är. En applet som till exempel försöker göra följande anrop:

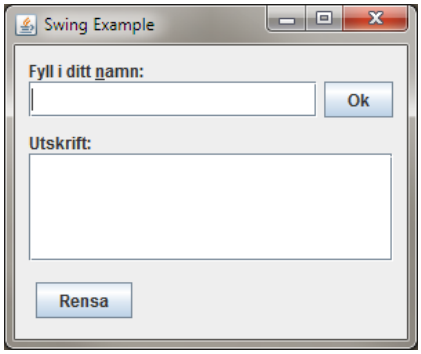
```
String userHome = System.getProperty("user.home");
```

kommer att kasta följande undantag (syns i Java-konsolen):

```
java.security.AccessControlException: access denied
(java.util.PropertyPermission user.home read)
      at java.security.AccessControlContext.checkPermission(Unknown Source)
      …
```

Bortsett från begränsningarna som nämnts ovan skiljer det som sagt inte mycket från att skapa en applikation och en applet med ett GUI. Vi kan använda samma komponenter och även lyssnare med mera fungerar på samma sätt. Skapa en klass som ärver JPanel och i denna skapar du ditt GUI. Skapa därefter en annan klass som ärver JApplet och använd din JPanel som panel i appleten. Ett relativt enkelt sätt att göra om en befintlig applikation till en applet är att ta den klass som ärver JFrame och i stället låta den ärva JPanel. Ta sen bort all kod som har att göra med initieringen av fönstret (sätta titel, storlek med mera).

I lektion 5 tittade vi på ett exempel där namn användaren skriver in i ett textfält hamnar i en textruta när användaren trycker på knappen Ok eller om användaren trycker på enter i textfältet. När användaren tryckte på knappen Rensa togs all text bort.



För att konvertera detta exempel från en applikation till en applet gör vi följande förändringar i källkoden (markerade med fetstil):

```
//public class SwingExample extends JFrame implements ActionListener {
public class SwingExamplePanel extends JPanel implements ActionListener {
  // Inastansvariabler (som förut)

  //public SwingExample(String title) {
  public SwingExamplePanel() {
    //super(title);
    //setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    //setLocationRelativeTo(null);
    // Ange vilka ikoner som fönstret ska använda
    //ArrayList<Image> iconImages = new ArrayList<Image>();
    //iconImages.add(new ImageIcon("miun16x16.png").getImage());
    //iconImages.add(new ImageIcon("miun32x32.png").getImage());
    //iconImages.add(new ImageIcon("miun64x64.png").getImage());
    //setIconImages(iconImages);
    //setSize(300, 250);
    //setVisible(true);

    super();
    setLayout(new BorderLayout());
    initComponents();
  }

  private void initComponents() {
    // som förut
  }

  public void actionPerformed(ActionEvent e) {
    // som förut
  }
```

```
  private void addNameToTextArea(boolean keepText) {
    // som förut
  }

  /*public static void main(String args[]) {
    new SwingExample("Swing Example");
  }*/
}
```

Vi börjar med att byta namn på klassen genom att lägga till Panel i slutet av namnet. Därefter ärver vi JPanel i stället för JFrame. I konstruktorn tar vi bort all kod som är specifik för JFrame. Till exempel titel på fönstret, placering på skärmen, ikoner och storlek. I stället nöjer vi oss med att anropa super, sätta layout till BorderLayout och anropa metoden som initierar våra komponenter. Eftersom en applet inte har någon main-metod tar vi bort den helt. I övrigt ändrar vi inget i klassen. Det vill säga alla instansvariabler finns kvar och metoderna initComponents, actionPerformed och addNameToTextArea är oförändrade.

Vi skapar nu en klass som ärver JApplet:

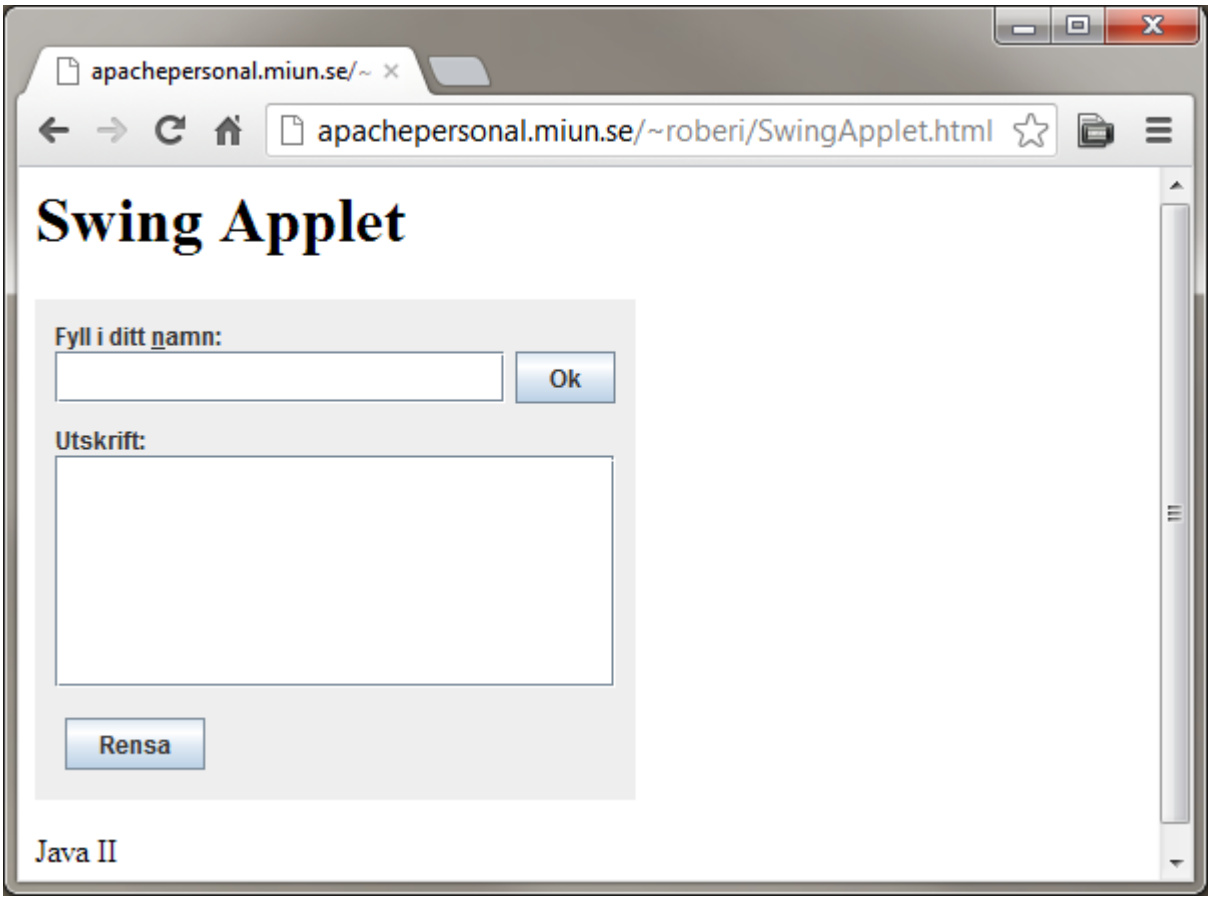
```
public class SwingApplet extends JApplet {
  private SwingExamplePanel panel;

  public void init() {
    panel = new SwingExamplePanel();
    add(panel, BorderLayout.CENTER);
  }
}
```

I denna klass har vi en instansvariabel av typen SwingExampelPanel. I init-metoden skapar vi en instans av vår panel och lägger till den i appleten genom ett anrop till add (JApplet använder BorderLayout som standard). Någon mer kod än så här behöver vi inte.

Eftersom vår applet består av fler än en klass bör vi paketera den i en jar-fil när det är dags att lägga ut den på en hemsida (under testning är det ett steg vi inte behöver göra). Den html-sida som laddar och visar vår applet kan se ut så här:

```
<html>
  <body>
    <h1>Swing Applet</h1>
    <applet code="SwingApplet" width="300" height="250"
      codebase="applets" archive="MyApplet.jar">
    </applet>
    <p>Java II</p>
  </body>
</html>
```

Resultatet i en webbläsare ser då ut så här:


apachepersonal.miun.se/~
apachepersonal.miun.se/~roberi/SwingApplet.html
Swing Applet
Fyll i ditt namn:
Ok
Utskrift:
Rensa
Java II