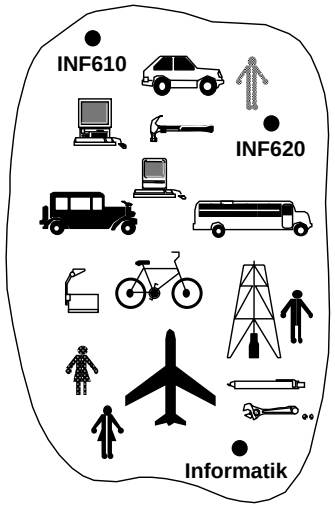
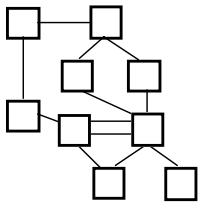
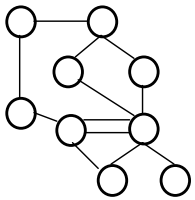
OO-ett sätt att modellera

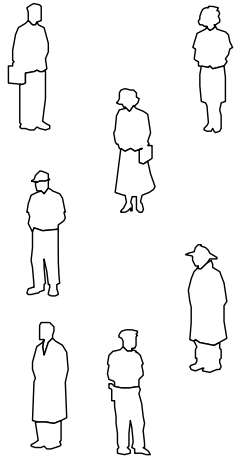






Verklighet **Konceptuell modell** **Program**

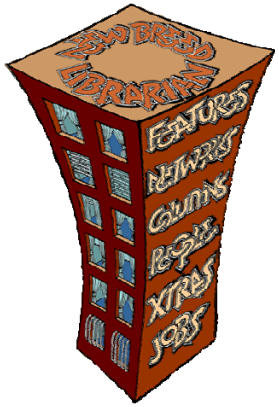
Lönesystemet, ett exempel

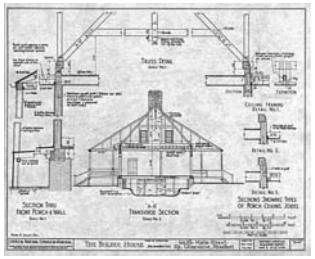


Två anställningsformer:
Timanställd
Fast anställd

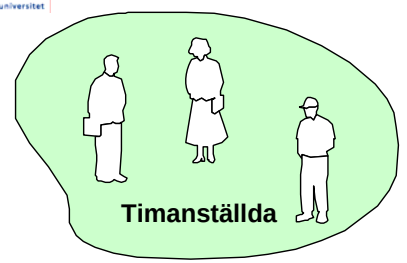
- **Varje anställd har ett unikt namn samt adress och telefon**
- **Fast anställda har en fast månadslön**
- **Timanställdas lön beräknas utifrån de timmar som de har arbetat och deras timlön**

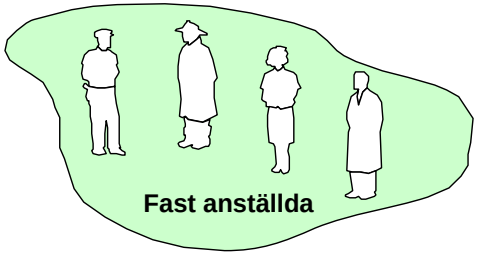
Varför modeller?

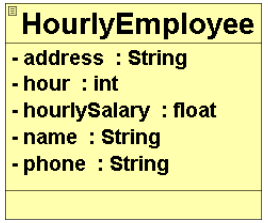


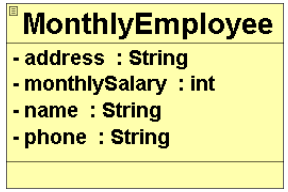


Klassificering

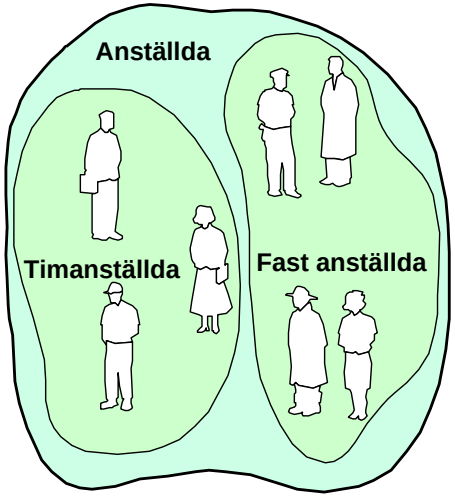


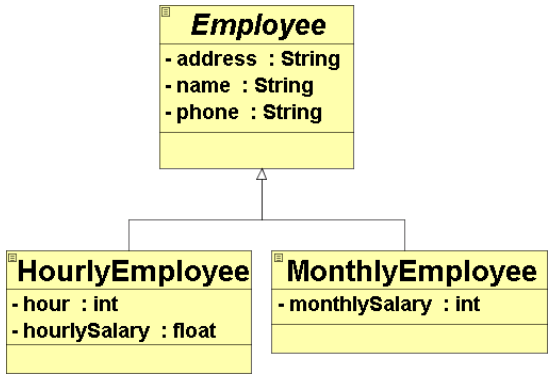




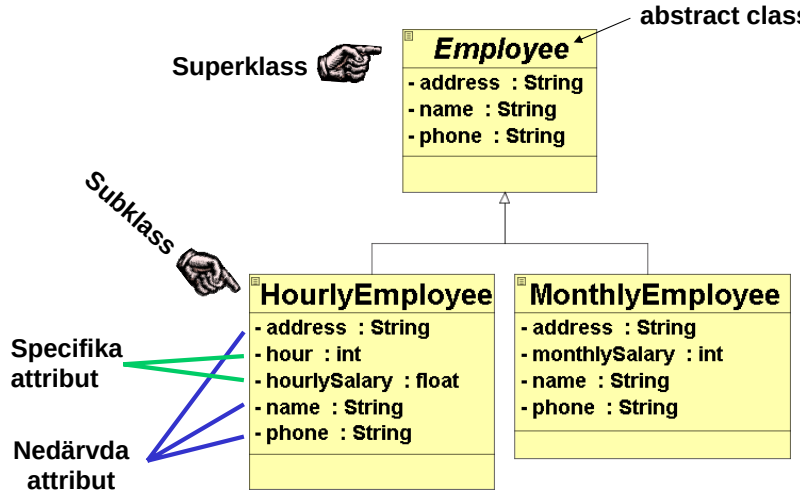


Generalisering / Specialicering



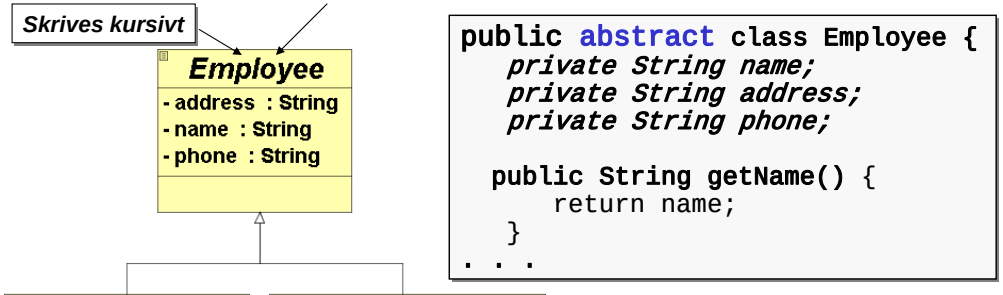


Arv i klasshierakier



Abstrakta klasser

En abstrakt klass kan inte instansieras

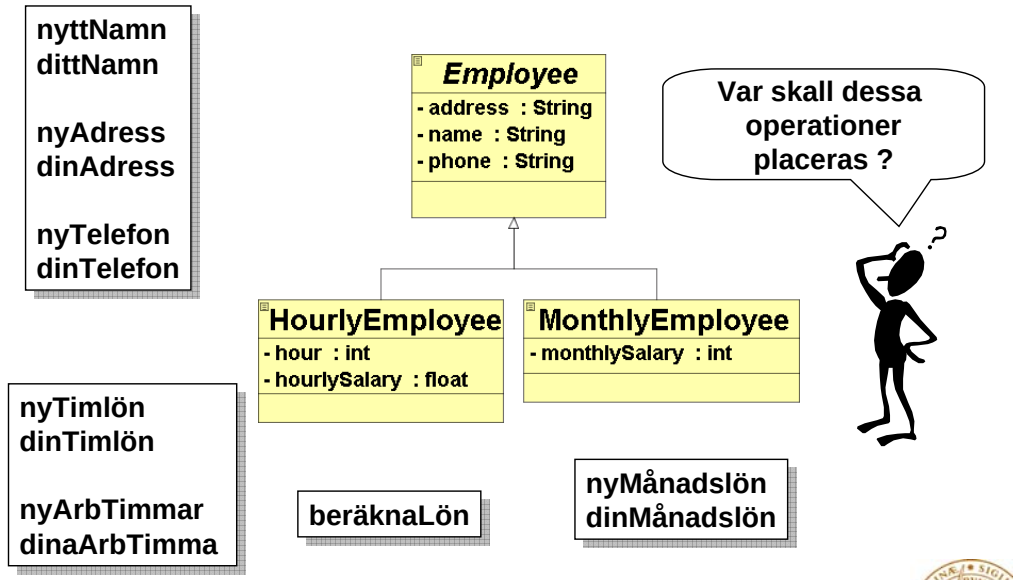



```
public abstract class Employee {
   private String name;
   private String address;
   private String phone;

  public String getName() {
      return name;
   }
. . .
```

```
public class IntroJavaApp {

  public static void main(String[] args) {
     Employee A = new Employee();
     A.setName("Mats");
      . . .
```

Compile Error . . .

Operationer



Implementering-Anstalld

```
public abstract class Employee {
  private String name;
  private String address;
  private String phone;

  public String getName() {
    return name;
  }
  public void setName(String newName) {
    name = newName;
  }
  public String getAddress() {
    return address;
  }
  public void setAddress(String newAddress) {
    address = newAddress;
  }
  public String getPhone() {
    return phone;
  }
  public void setPhone(String newPhone) {
    phone = newPhone;
  }

  public abstract float salary();
}
```


Abstrakta metoder

Om en metod deklareras som abstrakt måste klassen också deklareras som abstrakt. Alla subklasser måste då implementera dessa metoder för att bli konkret.

Alla subklasser som vill bli konkreta måste implementera metoden *salary()*

```
public abstract class Employee {
   private String name;
   private String address;
   private String phone;

   public String getName() {
      return name;
   }

   . . .

   public abstract float salary();
}
```

No body !!

Implementering-FastAnstalld

```
public class MonthlyEmployee extends Employee
{

  private int monthlySalary;

  public int getMonthlySalary()
  {
    return monthlySalary;
  }

  public void setMonthlySalary(int newMonthlySalary)
  {
    monthlySalary = newMonthlySalary;
  }

  public float salary()
  {
    return this.getMonthlySalary();
  }
}
```


Implementering-Timanstalld

```
public class HourlyEmployee extends Employee {
  private int hour;
  private float hourlySalary;

  public int getHour() {
    return hour;
  }
  public void setHour(int newHour) {
    hour = newHour;
  }

  public float getHourlySalary() {
    return hourlySalary;
  }
  public void setHourlySalary(float newHourlySalary) {
    hourlySalary = newHourlySalary;
  }

  public float salary() {
    return this.getHour() * this.getHourlySalary();
  }
}
```

Begrepp

- Arv, Generalisering-specialisering
- Abstrakt (klass)
- Abstrakta (metoder)
- Super klass
- Subklass

Övning

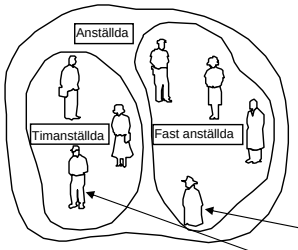
För en viss verksamhet gäller följande:

- Varje vara har namn, inköpspris och ett unikt varunummer
- Det finns två varugrupper A och B
- Försäljningspriset för varor räknas som 2-gånger inköpspriset för grupp A-varor och 3 gånger inköpspriset för varor i grupp B.

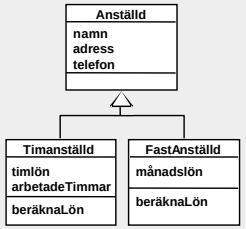
1. Rita en objektorienterad modell för denna verksamhet.
2. Implementera modellen i Java.

Modellering & programmering

Verklighet

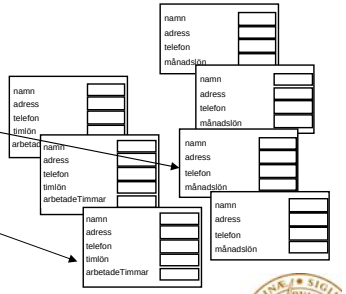


Modell



Program

```
class Anstalld {
   private String namn;
   private String adress;
   private String telefon;

public class FastAnstalld extends Anstalld {
   private int manadslon;

public class Timanstalld extends Anstalld {
   private int arbetadeTimmar;
   private float timlon;
```

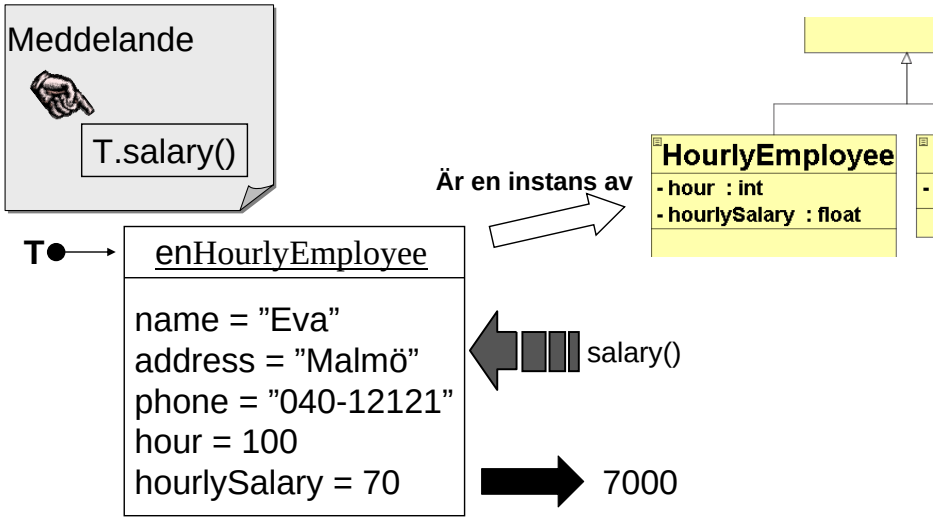


Testning av Anstalld

```
public class IntroJavaApp
{
  public static void main(String[] args)
  {
     MonthlyEmployee F = new MonthlyEmployee();
     F.setName("Peter");
     F.setAddress("Norrviddinge");
     F.setPhone("0418-12345");
     F.setMonthlySalary(18000);
     System.out.println(F.getName()+" "+F.getAddress());
     System.out.println(F.salary());

     HourlyEmployee T = new HourlyEmployee();
     T.setName("Eva");
     T.setAddress("Malmö");
     T.setPhone("040-12121");
     T.setHour(100);
     T.setHourlySalary(70);
     System.out.println(T.getName()+" "+T.getAddress());
     System.out.println(T.salary());
  } //main
} // IntroJavaApp
```

Meddelanden

All kommunikation med eller mellan objekt sker genom meddelanden



Default Constuctor

En konstruktor skiljer sig ifrån metoder på flera sätt.

- Har ingen returtyp (inte ens void)
- Heter samma som klassen.

Default Constructor tar inga parametrar.

- Har ingen returtyp (inte ens void)
- Heter samma som klassen.

```
public abstract class Employee {
  private String name;
  private String address;
  private String phone;

  public  Employee() //Default construtor
  {
    System.out.println("Hej från Employee()");
  }

  . . .
```


Default Constuctor

Alla klasser har en konstruktor som kallas *default constructor*.

Man behöver inte deklarera konstruktorer för varje klass (bara när man behöver tex. vid initiering). *Default constructor* körs alltid vid instansiering och finns det ingen implementering, så används default.

```
public abstract class Employee {
  private String name;
  private String address;
  private String phone;

  public  Employee()
  {
  }

  . . .
```


Andra konstruktorer

```
public abstract class Employee {
   private String name;
   private String address;
   private String phone;

   public Employee() { //Default contructor
      System.out.println("Hej från Employee()");
   }

   public Employee(String name,String addr,String phone) {
       this.name = name;
       address = addr;
       this.phone = phone;
   }

   public String getName() {
      return name;
   }

   . . .

   public abstract float salary();
}
```

Andra konstruktorer forts.

```
public class HourlyEmployee extends Employee
{
  private int hour;
  private float hourlySalary;

  public HourlyEmployee() { //Default contruc

    System.out.println("Hej från HourlyEmployee()");
  }

  public HourlyEmployee(String name,String addr,String phone) {

     System.out.println("Hej från HourlyEmployee(Str,Str,Str)");
  }

  . . .
}
```

Anropar (implicit) default constructor i Employee

```
HourlyEmployee T1 = new HourlyEmployee();
```

Hej från Employee()
Hej från HourlyEmployee()

```
HourlyEmployee T1 = new HourlyEmployee("Mats","Lund","12345");
```

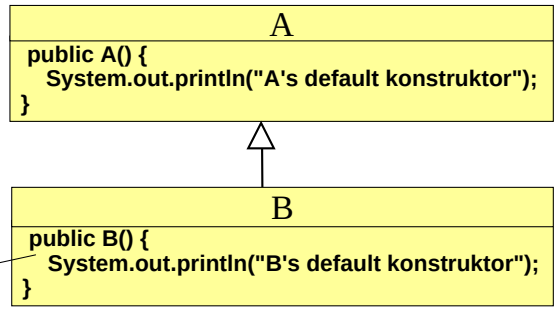
Hej från Employee()
Hej från HourlyEmployee(Str,Str,Str)

this & super

this & super är reserverade ord i Java.
this -> self (letar i mottagarklassen först)
super -> (letar i mottagarklassens superklass)

Constructor chaining

```
B b1 = new B( );
```



Andra konstruktorer forts.

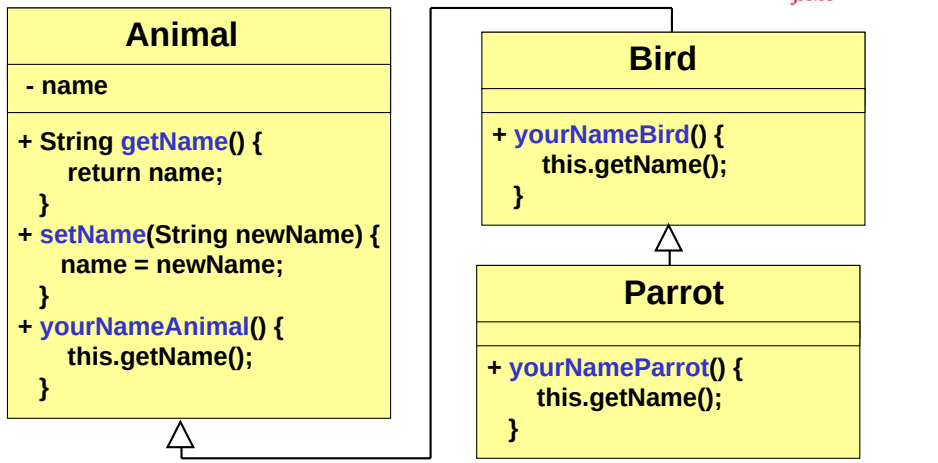
```
public class HourlyEmployee extends Employee
{
  private int hour;
  private float hourlySalary;

  public HourlyEmployee() { //Default contructor

     System.out.println("Hej från HourlyEmployee()");
  }

  public HourlyEmployee(String name,String addr,String phone) {

    super(name, addr, phone);
     System.out.println("Hej från HourlyEmployee(Str,Str,Str)");
  }

  . . .
}
```

```
HourlyEmployee T1 = new HourlyEmployee("Mats","Lund","12345");
```

Hej från HourlyEmployee(Str,Str,Str)

Övningar

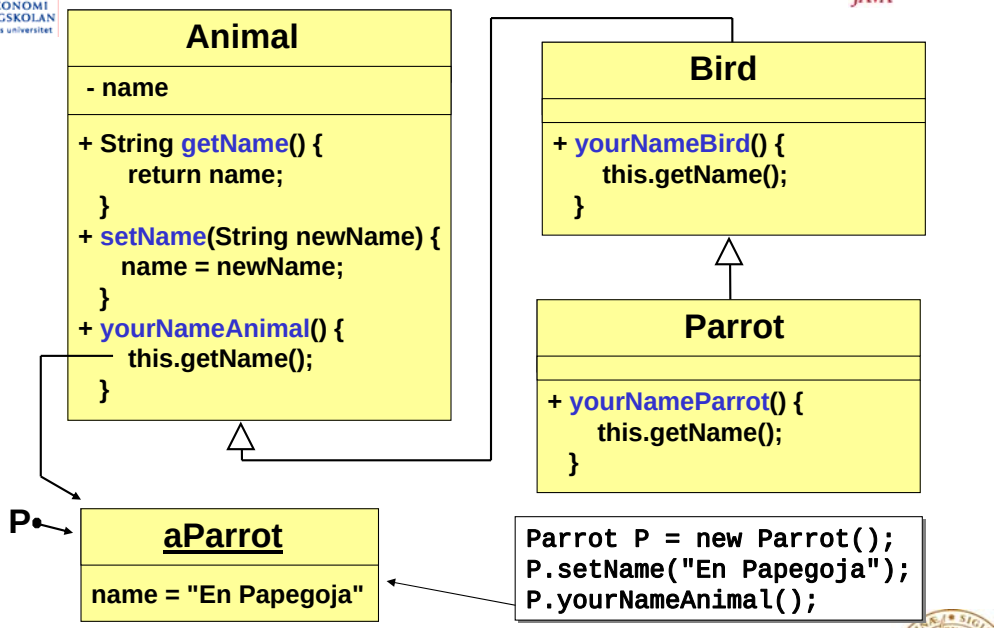
Övning



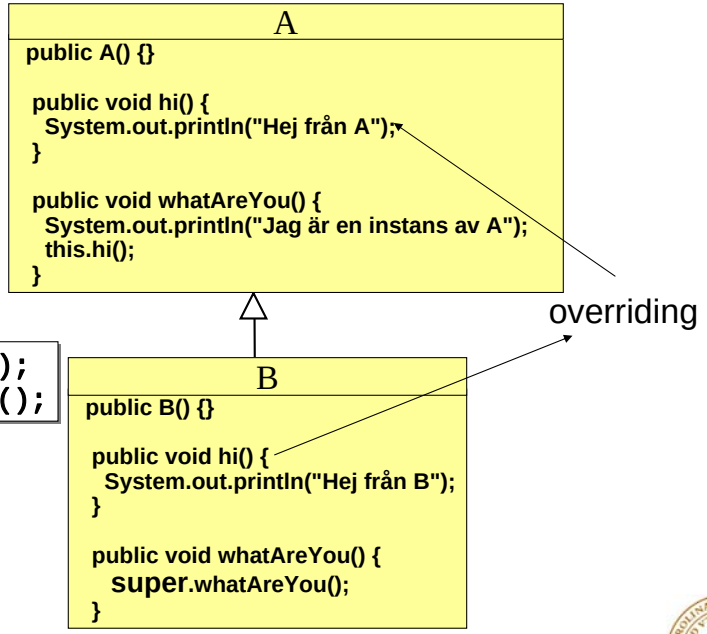

```
D = new Animal();
D.setName("Ett Djur");
F = new Bird();
F.setName("En Fågel");
P = new Parrot();
P.setName("En Papegoja"):
```

```
1. D.yourNameAnimal();
2. F.yourNameAnimal();
3. F.yourNameBird();
4. P.yourNameAnimal();
5. P.yourNameBird();
6. P.yourNameParrot();
```


this refererar alltid till mottagaren



this & super exempel




```
B b1 = new B();
b1.whatAreYou();
```


Meddelanden i sekvens

```
A = new MonthlyEmployee();
A.setName("Anna");
A.setAddress("Lund");
```

Fråga: Vad är Annas adress? Svar: Lund	`String adress = A.getAddress();`
Fråga: Hur många bokstäver finns i Lund?	`adress.length();`

Motsvarar

Fråga: Hur många bokstäver finns i Annas adress?	`A.getAddress().length();`

final

Privata metoder kan inte "*overridas*" och därför behöver det inte göras en "*dynamic method lookup*" (sker genom en dynamisk tabell) (inline).
Genom att deklarera en metod som final, kan den inte "*overridas*".
D v s. ingen lookup behövs. (Snabbare)

Ex.

```
public final String getName() {
   return name;
}
```


Package

Är en moduläriseringsmekanism som används för att gruppera klasser.
Paketet lagras i en katalogstruktur baserat på paketets namn.

java.lang.String
package name **class name**

classpath
└ java
  └ lang
    └ **String**
    ...

```
import java.lang.*;
class A {
   String s;
   . . .
}
```


Vilken klass tillhör ett objekt?

getClass()

Exempel.

```
if  (a.getClass().getName().equals("HourlyEmployee"))
{
   System.out.println(a.getHourlySalary());
}
```

Alternativt:

```
if (a.getClass() == HourlyEmployee.class)
{
   System.out.println(a.getHourlySalary());
}
```

instanceof (operator)

Exempel.

```
if (a instanceof HourlyEmployee) { . . . }
```


Klassen Object

```
public abstract class Employee extends Object {
  private String name;
  private String address;
  private String phone;

  . . .
```

Object metoder:

- clone
- **equals(Object o)** / hashCode
- finalize
- String toString()

Följande metoder kan inte overridas (de är final):

- Class getClass()
- notify
- notifyAll
- wait

Klassen Class

Under körning håller Java runtime reda på vilken klass varje objekt tillhör. Runtime typ-informationen används av den virtuella maskinen, så att rätt metod kan exekveras.
Man kan också komma åt denna information som finns i en speciell klass. Den klass som håller denna information kallas klassen Class.

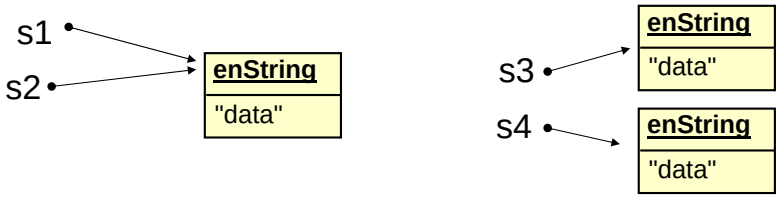
getClass() metoden i klassen Object returnerar en instans av klassen Class.

Ex.

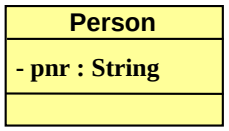
```
HourlyEmployee a;
. . .
Class c1 = a.getClass();
```


Equality & identity

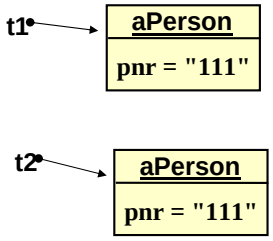
```
String s1 = "data";
String s2 = "data";
System.out.println("equals:" + s1.equals(s2)); //true
System.out.println("identical:" + (s1 == s2) ); //true


String s3 = new String("data");
String s4 = new String("data");
System.out.println("equals:" + s3.equals(s4));//true
System.out.println("identical:" + (s3 == s4) );//false
```



equals




```
class Person {
  private String pnr;
  Person(String pnr ) { this.pnr = pnr ; }
 }
...
Person t1 = new Person("111");
Person t2 = new Person("111");
System.out.println("equals:" + t1.equals(t2)); // false
```




```
class Person {
   private String pnr;
   Person(String pnr ) { this.pnr = pnr ; }

   boolean equals(Object t ) { //override Object's
      return pnr.equals( ((Person)t).pnr );
   }
 }
...
Person t1 = new Person("111");
Person t2 = new Person("111");
System.out.println("equals:" + t1.equals(t2)); // true
```


Övning

Modifiera klass definitionen, så att två instanser av Person blir likvärdiga endast om de har samma namn och adress.

Exceptions (Undantag)

Vad menas med undantag?

- Ett undantag är en onormal händelse under programexekveringen som avbryter det normala programflödet.
- En mekanism för att utföra felhantering utan att röra till i koden.
- Fel kan signaleras direkt, utan t.ex. flaggor och fält som måste kontrolleras. Programkod blir komplicerad och svårläst om den skall testa för varje tänkbart fel som kan uppstå.
- Exempel på fel som kan orsaka undantag är
  - Division med noll.
  - Indexering utanför gränserna på en array.
  - Overflow
  - Slut på minne
  - IO hantering (db kan inte kommas åt, fil finns inte etc.)

Exceptions (Undantag)

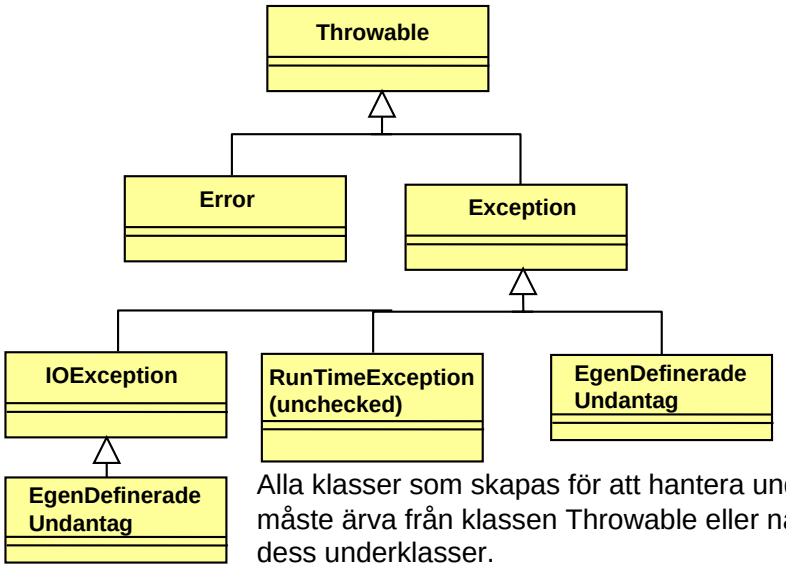
Syntax:

```
try {
   uttryck
   . . .
} catch (OneException var) {
   uttryck
   . . .
} [catch (AnotherException var1) {
   uttryck
   . . .
}]
[finally {
   uttryck
   . . .
}]
```

Två faser
1. Signalera fel
   throw an exception
2. Hantera felet
   catch an exception

Arvshierarkin-Undantagsklasser

I Java är alla undantag objekt.



Alla klasser som skapas för att hantera undantag måste ärva från klassen Throwable eller någon av dess underklasser.

Felhantering (Undantag)

try {

… Här utför man den kod som kan generera fel

} catch (Exception e)

… Om det skapas en Exception av koden som kördes fångas den upp i Exception e, och det som står i det passande Catch-blocket utförs.

Man kan ha valfritt antal catch-block och utföra olika saker beroende på vilken typ av exception som kastas.

finally

… Kod som alltid skall köras, även om ingen exception kastades.
Här stänger man tex connections till databasen.

Exception

```
try
{
  System.out.println("I try");
  int tal = 5/0;
}
catch (RuntimeException RE)
{
  System.out.println("I catch");
  System.out.println("\n"+RE.getClass());
}
finally
{
  System.out.println("I finally");
}
```

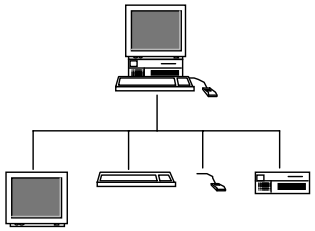
```
I try
I catch
java.lang.ArithmeticException: / by zero
I finally
```


Exception i Hierarki

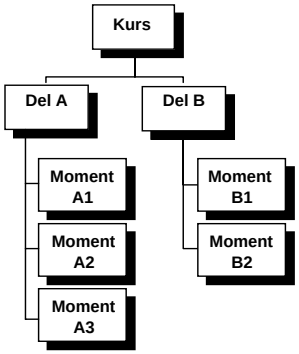
```
try {
   System.out.println("I try");
   int tal = 5/0;
}
catch (RuntimeException RE) {
   System.out.println("I catch 1");
   System.out.println(RE);
}
catch (Exception e){
   System.out.println("I catch 2");
}
finally {
   System.out.println("I finally");
}
```

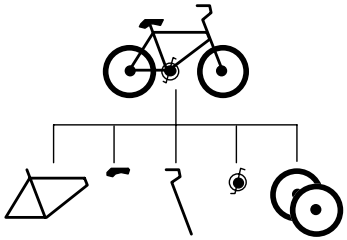
```
I try
I catch 1
java.lang.ArithmeticException: / by zero
I finally
```


Aggregering

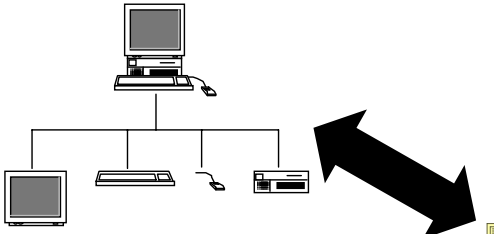


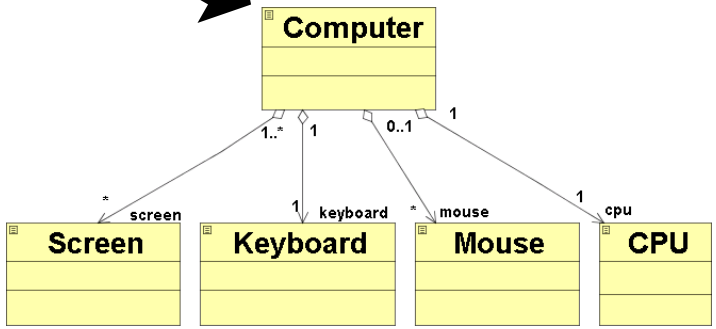
Aggregering uttrycker sambanden mellan ett sammansatt objekt och dess beståndsdelar



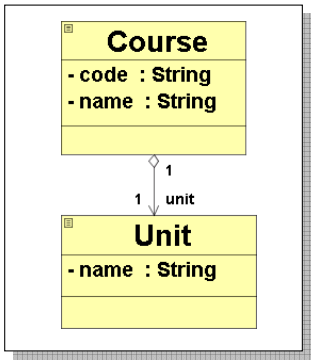


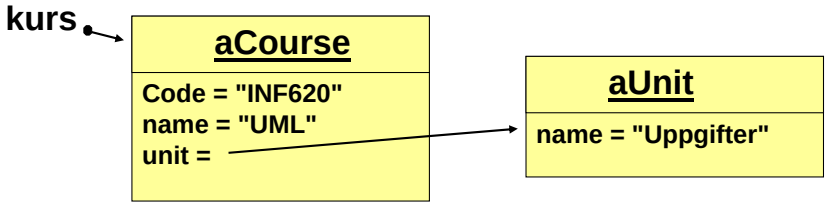
Aggregering (grafisk notation)



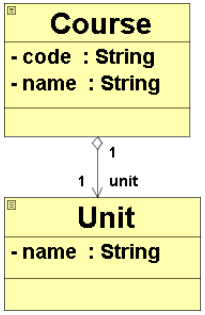


Aggregat 1 till 1 (grafisk notation)





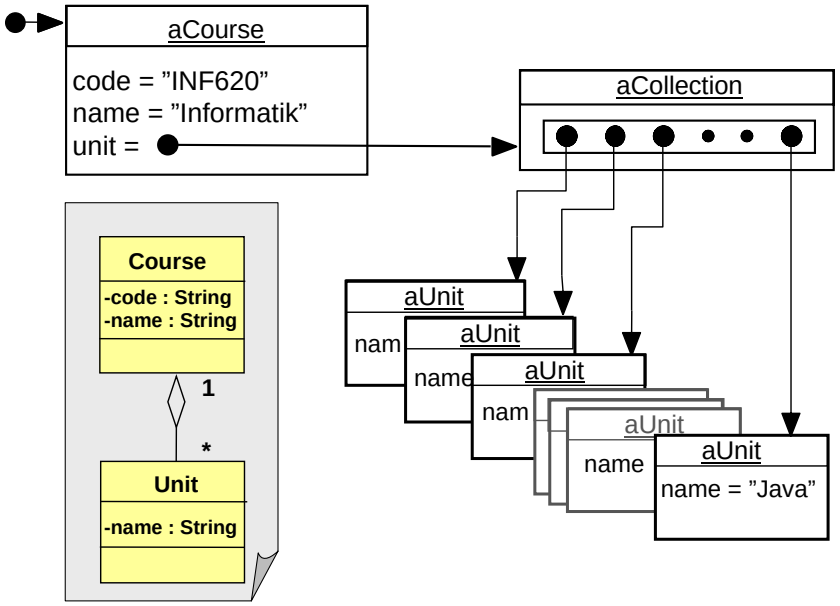
Aggregat 1 till 1 (implementering)




```
public class Course {
  private String code;
  private String name;
  private Unit unit;

  public String getCode() {
    return code;
  }

  . . .

  public Unit getUnit()
  {
    return unit;
  }

  public void setUnit(Unit newUnit)
  {
    unit = newUnit;
  }
}
```

```
public class Unit
{
  private String name;

  public String getName()
  {
    return name;
  }

  public void setName(Stri
  {
    name = newName;
  }
}
```


Aggregering (implementering)



Behållarklasser

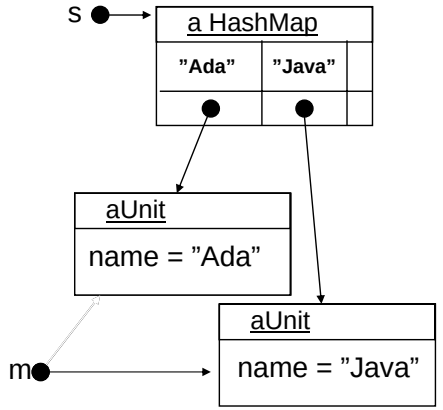
1.) **(import java.util.*;)**
HashMap s = new HashMap ();

2.) **Unit m = new Unit();**
m.setName(”Ada”);

3.) **s.put(m.getName(), m);**
//collection.put(Object key, Object value)

4.) **m = new Unit();**
m.setName(”Java”);

5.) **s.put(m.getName(), m);**

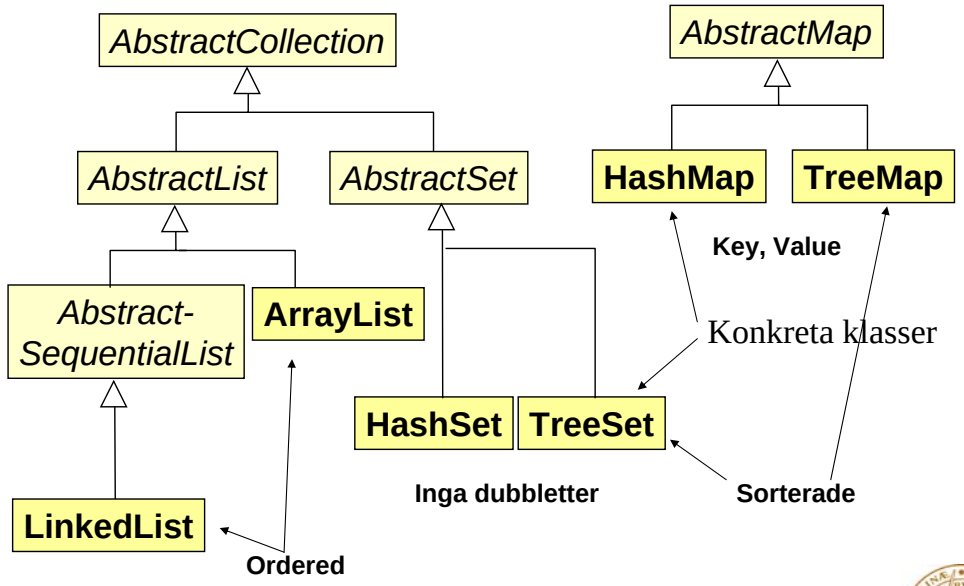


Info om behållarklasser

Hashmap är inte synchronized!!

(map=Collections.synchronizedMap(m)

Behållarklass hierarki



Aggregering, implementering

Definiera instansvariabeln units:

```
'class Course
 private String name;
 private String code;
 private HashMap units;
```

Definiera åtkomstmetoder för units:

```
public HashMap getUnits() {
     return units;
}
public void setUnits(HashMap aHashMap) {
    units = aHashMap;
}
```

Definiera en metod som initierar instansvariabeln units till en lista: (constructor)

```
public Course() {
   units = new HashMap();
}
```

Definiera en metod som lägger till en unit i kursen:

```
public void addUnit(Unit aUnit) {
   units.put(aUnit.getName(), aUnit);
}
```

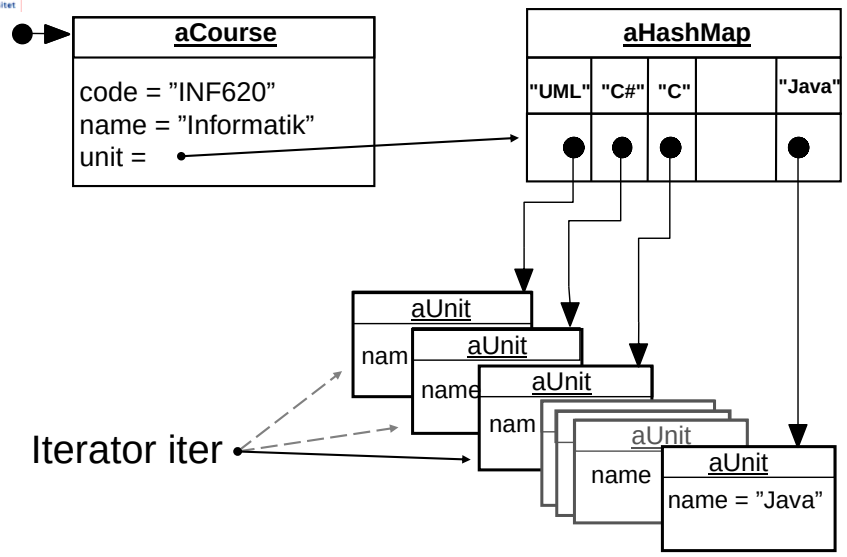

Enumeration vs. Iteration

Iteration (interface) ersätter Enumeration.

Hur får man tag i en iteration ?

- I collection (alla utom Map) :
  Iterator iter = collection.iterator();
- I Map: (return *views*)
  - ***Set* keySet()**
    Returnerar alla ***nyckelobjekten*** till ett set
    (objekt som implementerar intefacet Set)
    Iterator iter = map.keySet().iterator();
  - ***Collection* values()**
    Returnerar alla ***valueobjekten*** till en collection
    Iterator iter = map.values().iterator();
  - ***Set* entrySet()**
    Returnerar allt (**både key och value**) till ett set
    Iterator iter = map.entrySet().iterator();

Grafiskt



Iterator

Har tre fundamentala metoder:

***Object* next()**
returnerar nästa element i listan, om inget element kan nås slängs ett undantag (NoSuchElementException)

***boolean* hasNext()**
returnerar true om iteratorobjektet fortfarande har fler element i listan

***void* remove()**
Tar bort det element som senast returnerades av next ut listan

Exempel på iteration

```
import java.util.*;
public class Appl {
  public static void main(String[] args)
  {
    Course k = new Course();
    k.setCode("INF620");
    k.setName("Informatik");

    Unit m = new Unit();
    m.setName("Ada");

    k.addUnit(m);

    m = new Unit();
    m.setName("Java");
    k.addUnit(m);

    Iterator iter = k.getUnits().entrySet().iterator();
    while (iter.hasNext()) {
      Map.Entry entry = (Map.Entry)iter.next();
      Unit tmp = (Unit)entry.getValue();
      System.out.println(tmp.getName());
    }
}
```


Map.Entry

Är en inner class of map

Java.util.Map.Entry:

- *Object* getKey()
- *Object* getValue()
- *Object* setValue(Object newValue)
  returnerar det gamla värdet

Exempel på iteration

```
iter = k.getUnits().keySet().iterator();

while (iter.hasNext()) {
    String tmp = (String)iter.next();
    System.out.println(tmp);
}
```

```
iter = k.getUnits().values().iterator();

while (iter.hasNext()) {
    Unit tmp = (Unit)iter.next();
    System.out.println(tmp.getName());
}
```


Söka / Tabort i HashMap

public Object **get**(Object key) - **Returnerar null om nyckel inte finns.**

```
Unit m1 = (Unit)k.getUnits().get("Java");
if (m1 != null) {
   System.out.println("Moment är: "+m1.getName());
}
```

public Object **remove**(Object key) - **Returnerar null om nyckel inte finns.**

```
m1 = (Unit)k.getUnits().remove("Java");
if (m1 != null) {
  System.out.println("Momentet är borttaget: "+m1.getName());
}

m1 = (Unit)k.getUnits().get("Java");
if (m1 == null) {
   System.out.println("Momentet är borta");
}
```


Övning

Skriv nödvända uttryck som:

a) Skapar en lista (ArrayList) som består av tre anställda med följande namn och lön:
   Anna 15000
   Lars 14000
   Eva 16000

b) Skriver ut namnen på alla anställda som tjänar mer än 14000.

c) Skriver ut antalet anställda som tjänar mer än 14000

d) Skriver ut summan av alla anställdas löner

Employee

- **name : String**
- **salary : float**

```
public String getName() {
  return name;
}

public void setName(String newName) {
  name = newName;
}

public float getSalary() {
  return salary;
}

public void setSalary(float newSalary) {
  salary = newSalary;
}
```


Övning

Definiera klassen ***Tallista*** som förutom alla metoder för *LinkedList* även har metoder som returnerar:

- summan av all tal i listan.
- genomsnittet av talen i listan.

Du behöver inte bry dig om att kontrollera att mängden ej är tom eller att den endast innehåller tal.

Lab

ArrayList	Reultat	Returnerat objekt
ArrayList s = new ArrayList();	Skapar en ArrayList	
s.add("a");	"a"	
s.add(new Integer(20));	"a", 20	
s.add("x");	"a", 20, "x"	
s.get(0);	"a", 20, "x"	"a"
s.get(s.size()-1)	"a", 20, "x"	"x"
s.add(0, "first");		
s.add(s.size(), "last");		
s.get(0);		
s.get(s.size()-1);		
s.remove("first");		
s.remove(s.size()-1);		
s.contains("x")		
s.isEmpty();		

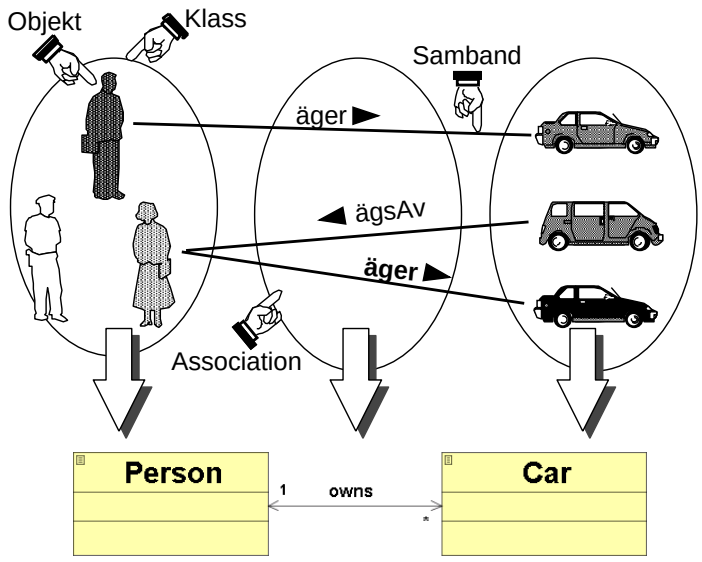
Lab

LinkedList	Reultat	Returnerat objekt
LinkedList l = new LinkedList();	Skapar en LinkedList	
l.add("a");	"a"	
l.add(new Integer(20));		
l.add("x");		
l.getFirst();		
l.getLast()		
l.add(0, "first");		
l.addLast("last");		
l.getFirst();		
l.get(l.size()-1);		
l.remove("first");		
l.remove(l.size()-1);		
l.contains("x")		
l.isEmpty();		

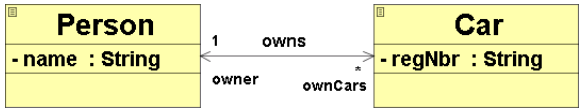
Lab

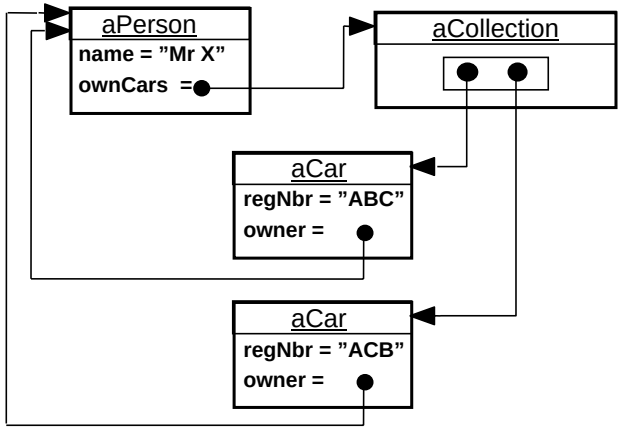
HashMap	Reultat	Returnerat objekt
HashMap h = new HashMap();	Skapar en HashMap	
h.put(new Integer(1),"one");	1 -> "one"	
h.put(new Integer(2), "two");	1 -> "one", 2->"two"	
h.put("one", "two");		
h.get(new Integer(1))		
h.get("one");		
h.put("one", "four");		
h.containsKey("one");		
h.containsValue("Two");		
h.remove(new Integer(1));		
h.size();		
h.remove("one");		
h.isEmpty();		

Association



Associationer, implementering





Associationer, Car

```
public class Car {
  private Person owner;
  private String regNbr;

  public String getRegNbr() {
    return regNbr;
  }

  public void setRegNbr(String newRegNbr) {
    regNbr = newRegNbr;
  }

  public Person getOwner() {
    return owner;
  }

  public void setOwner(Person newOwner) {
    owner = newOwner;
  }
}
```


Associationer, Person

```
import java.util.*;
public class Person {
  private List ownCars;
  private String name;

  public Person() {
    this.setOwnCars(new LinkedList());
  }
  public String getName() {
    return name;
  }
  public void setName(String newName) {
    name = newName;
  }

  public List getOwnCars() {
    return ownCars;
  }
  public void setOwnCars(List newOwnCars) {
    ownCars = newOwnCars;
  }
  public void ownCar(Car aCar) {
    this.getOwnCars().add(aCar);
  }
}
```


Associationer

```
import java.util.*;
public class Appl {
  public static void main(String[] args) {
    Car c1 = new Car();
    c1.setRegNbr("ABC123");
    Car c2 = new Car();
    c2.setRegNbr("ACB332");

    Person p = new Person();
    p.setName("Mr X");

    p.ownCar(c1);
    c1.setOwner(p);
    p.ownCar(c2);
    c2.setOwner(p);

    System.out.println(p.getName() + " äger följande bilar: ");
    Iterator iter = p.getOwnCars().listIterator();
    while (iter.hasNext()) {
      Car b =(Car)iter.next();
      System.out.print(b.getRegNbr()+" -> ");
      System.out.println(b.getOwner().getName());
    }
  }
}
```

Skapa bilar

Skapa person

Koppla Person <--> Bil

Övning

Konstruera en objektorienterad modell för en verksamhet där:

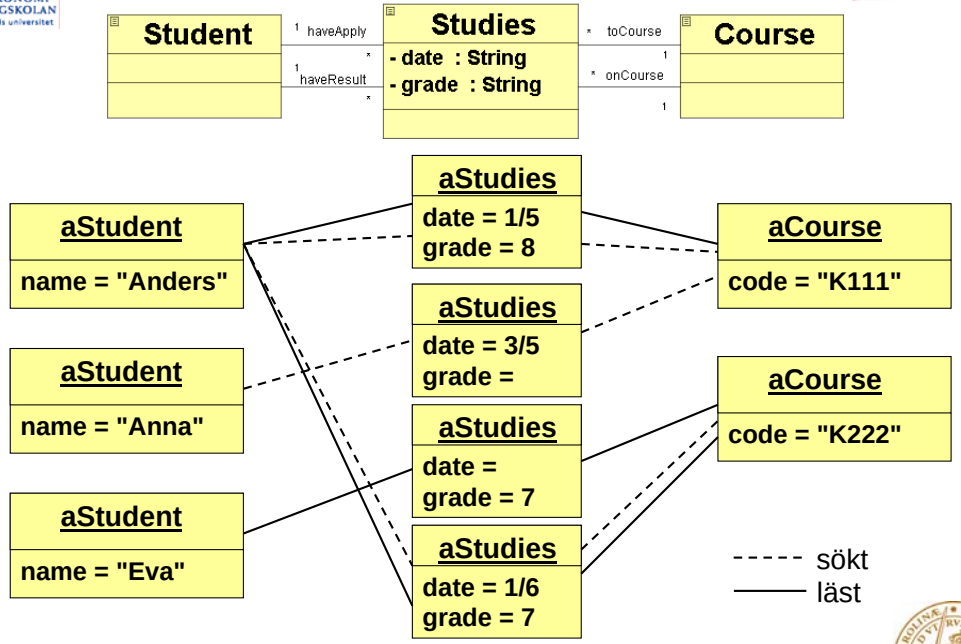
- Varje kurs kan läsas av många studenter men en student kan läsa högst en kurs.
- Det inte kan finnas studenter som inte läser någon kurs.
- En student kan söka många kurser och en kurs kan sökas av många studenter.
- Man också är intresserad av det datum som en student har sökt en viss kurs.

Övning

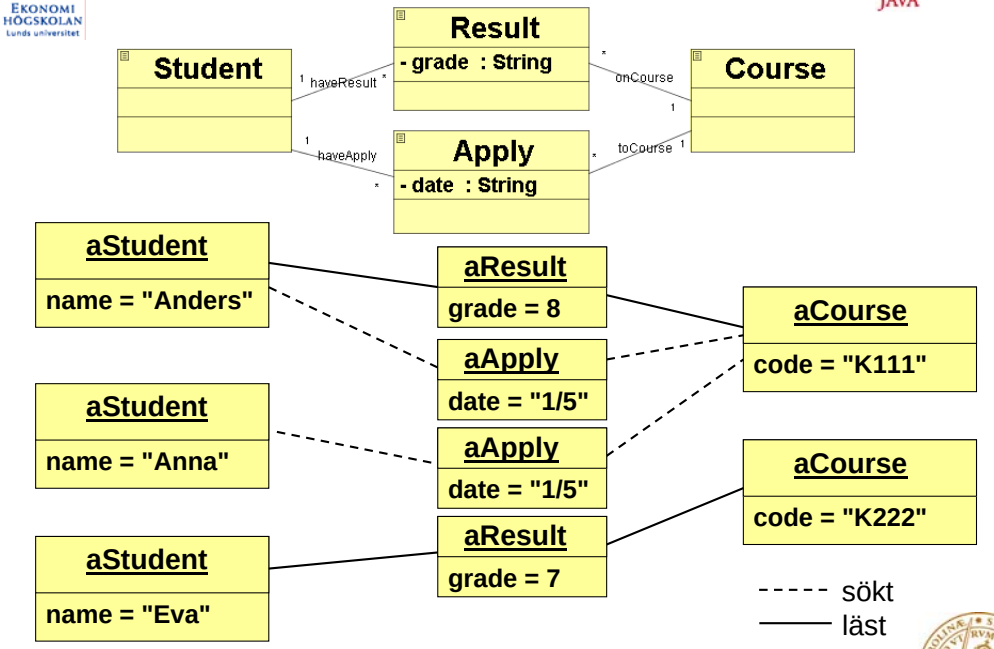
Konstruera en objektorienterad modell för en verksamhet där:

- Varje kurs kan läsas av många studenter och en student kan läsa många kurser.
- Det kan finnas studenter som inte läser någon kurs.
- Studenterna får också betyg på sina kurser.
- En student kan söka många kurser och en kurs kan sökas av många studenter.
- Man också är intresserad av det datum som en student har sökt en viss kurs.

Associationsklasser



Associationsklasser



Implementering

```
import java.util.*;

public class Student {
  private String name;
  private List results;
  private List applies;

  public Student() {
    this.setApplies(new LinkedList());
    this.setResults(new LinkedList());
  }

  public List getApplies() {return applies;}

  . . .

  public void setResult(Result r) {
    this.getResults().add(r);
  }

  public void setApply(Apply a) {
    this.getApplies().add(a);
  }
}
```


Implementering

```
import java.util.*;

public class Course {
  private String code;
  private List results;
  private List applies;

  public Course() {
    this.setApplies(new LinkedList());
    this.setResults(new LinkedList());
  }

  public List getApplies() {return applies;}

  . . .

  public void setResult(Result r) {
    this.getResults().add(r);
  }

  public void setApply(Apply a) {
    this.getApplies().add(a);
  }
}
```


Implementering

```
public class Result {
  private String grade;
  private Student aStudent;
  private Course aCourse;

  public Course getACourse() {return aCourse;}

  public void setACourse(Course newACourse) {
    aCourse = newACourse;
  }

  public Student getAStudent() {return aStudent;}

  public void setAStudent(Student newAStudent) {
    aStudent = newAStudent;
  }

  public String getGrade() {return grade;}

  public void setGrade(String newGrade) {
    grade = newGrade;
  }
}
```


Implementering

```
public class Apply {
  private String date;
  protected Student aStudent;
  private Course aCourse;

  public Course getACourse() {return aCourse;}

  public void setACourse(Course newACourse) {
    aCourse = newACourse;
  }

  public Student getAStudent() {return aStudent;}

  public void setAStudent(Student newAStudent) {
    aStudent = newAStudent;
  }

  public String getDate() {return date;}

  public void setDate(String newDate)
  {
    date = newDate;
  }
}
```

Application

```
Course k1 = new Course();
k1.setCode("K111");
Course k2 = new Course();
k2.setCode("K222");

Student s1 = new Student();
s1.setName("Anders");
Student s2 = new Student();
s2.setName("Anna");
Student s3 = new Student();
s3.setName("Eva");

Result r1 = new Result();
r1.setGrade("8");

r1.setAStudent(s1);
s1.setResult(r1);
r1.setACourse(k1);
k1.setResult(r1);

Result tmpC = (Result)s1.getResults().get(0);
System.out.println(tmpC.getACourse().getCode());
```


Application

```
Apply a1 = new Apply();
a1.setDate("1/5");

s1.setApply(a1);
a1.setAStudent(s1);
a1.setACourse(k1);
k1.setApply(a1);

a1 = new Apply();
a1.setDate("1/5");

s2.setApply(a1);
a1.setAStudent(s2);
a1.setACourse(k1);
k1.setApply(a1);


Iterator iter = k1.getApplies().iterator();
while (iter.hasNext())
{
  Apply a = (Apply)iter.next();
  System.out.println(a.getAStudent().getName());
```


Klassmetoder (static)

Statiska metoder kan inte "*be overridden*".

En statisk metod innebär att metoden har klass räckvidd (*scope)*.
Dvs. Den tillhör klassen och inte instansen.
Man behöver alltså inte skapa någon instans för att använda den.
Ex.

A.SkapadAv();

Mottagaren är en klass

```
public class A extends Object {

  static void SkapadAv() {
    System.out.println("Copyright by Informatics");
  }
}
```


Övning i klassmetoder

A

```
public void whoAreYou() {
   System.out.println("Jag är instans av A");
 }

 public static void whatAreYou() {
   System.out.println("Jag är klass A");
 }
```

B

```
public void whoAreYou() {
   super.whoAreYou();
}

public static void whatAreYou() {
   System.out.println("Jag är klass B");
}
```

C

```
public void whoAreYou() {
   super.whoAreYou();
}

public static void whatAreYou() {
   System.out.println("Jag är klass C");
}
```

```
A.whatAreYou();
B.whatAreYou();
C.whatAreYou();
```

```
new A().whoAreYou();
new A().whatAreYou();//Rekommenderas inte
new B().whoAreYou();
new C().whoAreYou();
```

Svar till övning

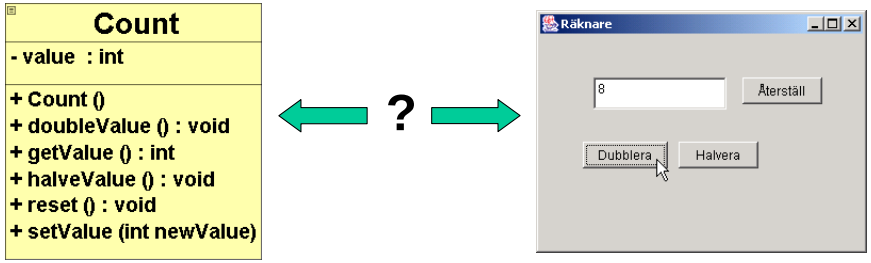
```
import java.util.*;

public class Tallista extends LinkedList {

  public int summa() {
    ListIterator l = this.listIterator();
    int sum = 0;
    Integer tal;
    while (l.hasNext()) {
      tal = (Integer)l.next();
      sum = sum + tal.intValue();
    }
    return sum;
  }

  public float genomsnitt() {
    int summa;
    int antal;
    summa = this.summa();
    antal = this.size();
    return summa/(float)antal;
  }
}
```


Svar till övning

```
Employee e = new Employee();
e.setName("Anna");
e.setSalary(15000);

ArrayList s = new ArrayList();
s.add(e);

e = new Employee();
e.setName("Lars");
e.setSalary(14000);
s.add(e);

e = new Employee();
e.setName("Eva");
e.setSalary(16000);
s.add(e);
```

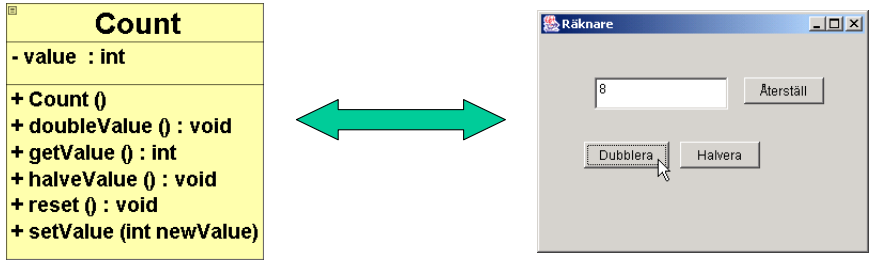
```
Iterator iter = s.iterator();
int i = 0;
float sum = 0;
while (iter.hasNext())
{
   e = (Employee)iter.next();
   sum+=e.getSalary();
   if (e.getSalary()>14000)
   {
     i++;
     System.out.println(e.getName());
   }
 }
 System.out.println(i);
 System.out.println(sum);
}
```


Koppling mellan problemområdet och användargränssnittet



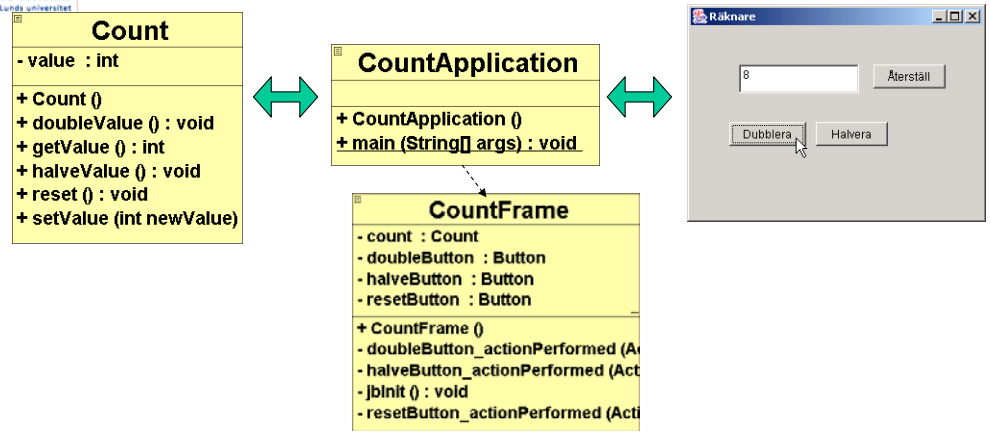
Hur skall användargränssnittet kopplas till klasserna i problemområdet?

Direkt koppling



Att anpassa klasser i problemområdeskomponenten till användargränssnittet påverkar både återanvändbarheten och underhållbarheten negativt.

Koppling genom en applikationsklass



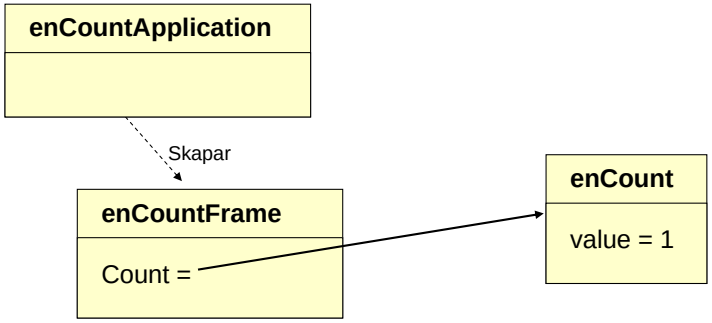
Domänmodell
Domain model

Applikationsmodell
Application model

Användargränssnitt
User interface

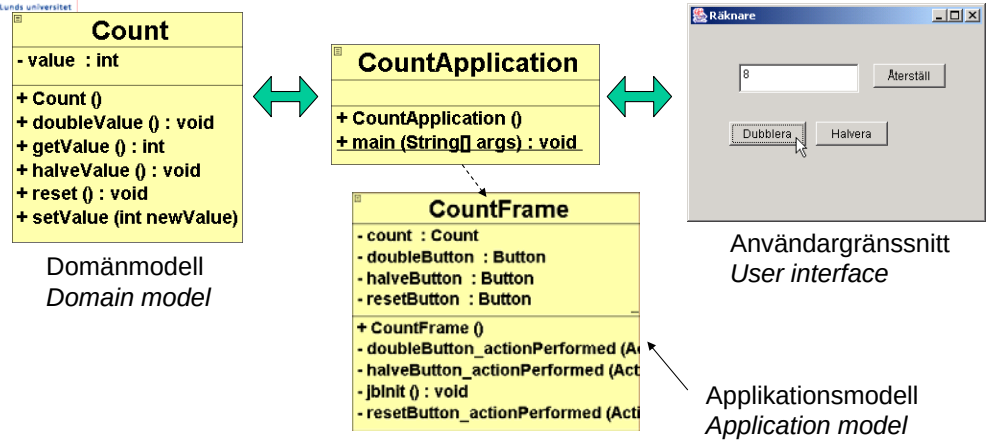
Resulterar i högre grad av återanvändbarhet och underhållbarhet

Koppling mellan applikations- och domänmodellen



Koppling mellan applikationsmodellen och domänmodellen sker genom att låta applikationsmodellen hålla en instans av domänmodellen.

Koppling mellan applikations-, domänmodellen och användargränsnittet



Domänmodell
Domain model

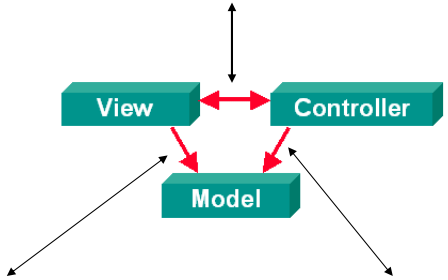
Användargränssnitt
User interface

Applikationsmodell
Application model

Koppling mellan användargränssnittet och applikationsmodellen sker genom händelser (events) och referenser.

MVC

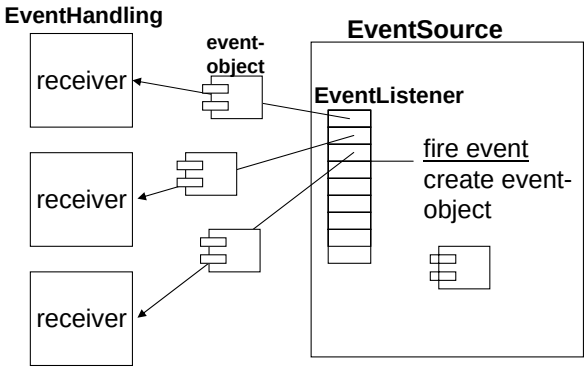
Koppling mellan användargränssnittet och applikationsmodellen sker genom händelser (events) och referenser.



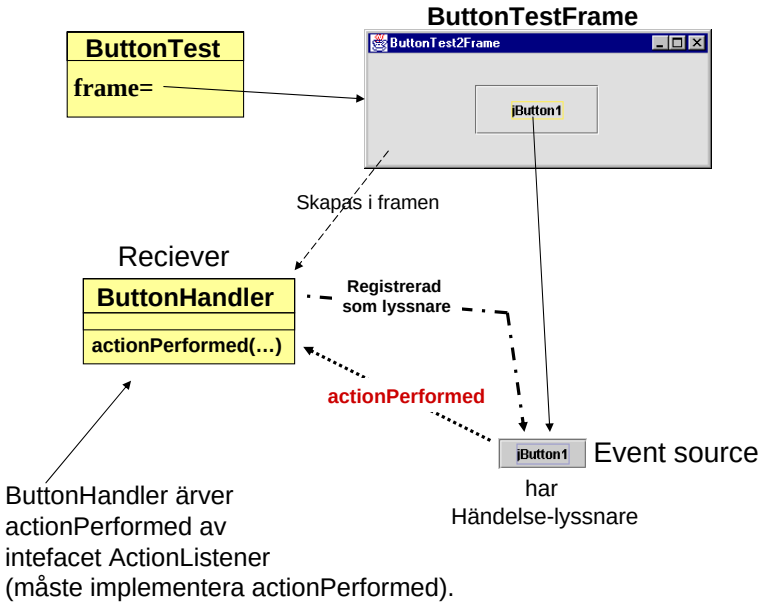
Koppling mellan applikationsmodellen och domänmodellen sker genom att låta applikationsmodellen hålla en instans av domänmodellen.

Delegation event model

- Event source (källa)
- Event objects
- Event Listeners (lyssnare)

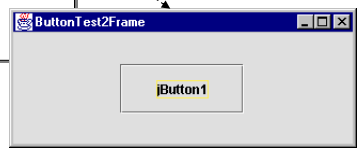


Egen klass som lyssnare



Exempel: ButtonTest (Appl)

```
import java.awt.*;

public class ButtonTest {

  public ButtonTest() {
    Frame frame = new ButtonTestFrame();
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    new ButtonTest();
  }
}
```



Exempel: Egen klass som lyssnare

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

class ButtonHandler implements ActionListener {

  public void ButtonHandler() { }

  public void actionPerformed(ActionEvent e) {
        System.out.println(e.getActionCommand());
  }
}
```

Exempel: ButtonTestFrame

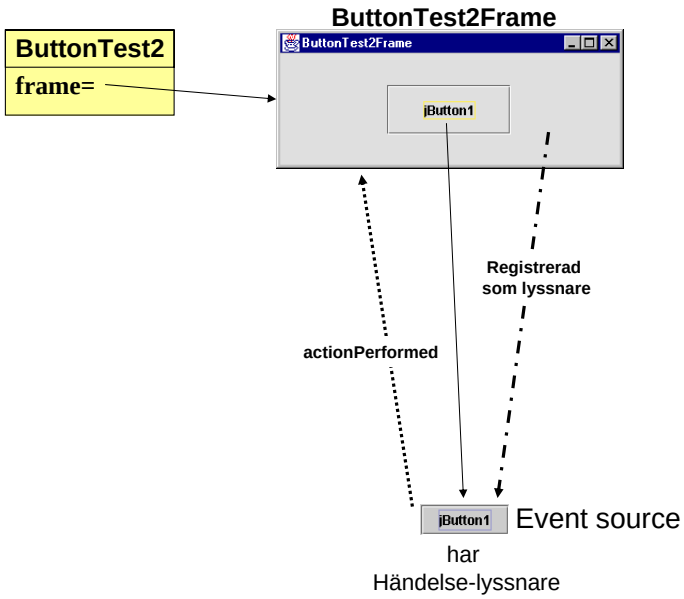
```
public class ButtonTestFrame extends JFrame {
   BorderLayout borderLayout1 = new BorderLayout();
   JPanel jPanel1 = new JPanel();

   Button button;
   JButton jButton1 = new JButton();

  public ButtonTestFrame() {
    super();
    try  {
      jbInit();
    } catch (Exception e) { e.printStackTrace(); }
  }
```

forts.

Forts. ButtonTestFrame

```
 private void jbInit() throws Exception {
    this.getContentPane().setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    jButton1.setText("jButton1");
    jButton1.setBounds (new Rectangle(82, 154, 172, 41));
    jPanel1.setLayout(null);
    this.setTitle("ButtonTestFrame");
    this.getContentPane().add(jPanel1, BorderLayout.CENTER);
    jPanel1.add(jButton1, null);
    jButton1.addActionListener (new ButtonHandler());
  } //jbInit
} //ButtonTestFrame
```


Frame som lyssnare



Exempel: ButtonTest2 (Appl)

```
import java.awt.*;

public class ButtonTest2 {

  public ButtonTest2() {
    Frame frame = new ButtonTest2Frame();
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    new ButtonTest2();
  }
}
```

Exempel: Frame som lyssnare

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

public class ButtonTest2Frame extends JFrame
                              implements ActionListener {

   BorderLayout borderLayout1 = new BorderLayout();
   JPanel jPanel1 = new JPanel();
   JButton jButton1 = new JButton();

   public ButtonTest2Frame() {
      super();
      try  {
        jbInit();
      }catch (Exception e) { e.printStackTrace(); }
   }
```

forts.

Frame som lyssnare (forts.)

```
   private void jbInit() throws Exception {
      this.getContentPane().setLayout(borderLayout1);
      this.setSize(new Dimension(313, 139));
      jButton1.setText("jButton1");
      jButton1.setBounds(new Rectangle(100, 29, 116, 46));
      jPanel1.setLayout(null);
      this.setTitle("ButtonTest2Frame");
      this.getContentPane().add(jPanel1, BorderLayout.CENTER);
      jPanel1.add(jButton1, null);
      jButton1.addActionListener(this);
   }

   public void actionPerformed(ActionEvent e) {
       System.out.println(e.getActionCommand( ));
   }
}//ButtonTest2Frame
```


Inre klass

```
public class InnerClassTest {
  private int data1;

  public InnerClassTest(int d) { //constructor
    data1 = d ; //10
    MyInnerClass inner = new MyInnerClass(20);
    inner.print();
  }

  public class MyInnerClass {
    private int data2; //20
    MyInnerClass( int d) { data2 = d; }

    void print() {
     System.out.println(InnerClassTest.this.data1+this.data2+data1);//40
    }
  }

  public static void main(String[] args) {
    InnerClassTest t = new InnerClassTest(10);
  }
}
```

Samma sak (ingen namnkonflikt)

Regler för inre klasser

Regler:

- OuterClassNamn**$**InnerClassNamn**.class** // för inner klasser med namn OuterClassNamn**$#.class** // för anonyma klasser

  I Exemplet:
  InnerClassTest$MyInnerClass.class
  InnerClassTest.class

- OuterClassNamn.**this.data1** // för att komma åt yttre klass -> this

- **Yttre klass är ansvarig för att skapa objekt av inre klass**

WindowEvent (Adapter)

```
import java.awt.*;
import java.awt.event.*;

public class EventAppl {

  public EventAppl() {
    Frame frame = new EventFrame();
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
      } );
    frame.setVisible(true);
  }

  public static void main(String[] a
    new EventAppl();
  }
}
```

Registrerar applikationen som lyssnare till Frame

När någon stänger fönstret, så genereras ett WindowEvent (windowClosing metoden skickas till alla registrerade lyssnare).

Det finns 6 andra också i WindowListener klassen. Tex. windowIconified, windowActivated etc.)

WindowAdapter

Syntax för anonym klass:

```
Klass obj = new Klass()
            {
              //Klass definition
            };
```

Exempel på en anonym klass:

```
new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

Varför Adapter ?
Java interface kräver att alla metoder i interfacet skall implementeras.

Adapter implementerar dessa metoder (implementerar WindowListener) att inte göra någonting.

Detta betyder att vi bara behöver definiera de vi vill använda.

Skapar en ny instans av klassen WindowAdapter samtidigt som man definierar metoden windowClosing

Exempel: Anonym klass

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

public class EventFrame extends JFrame {
    BorderLayout borderLayout1 = new BorderLayout();
    JPanel jPanel1 = new JPanel();
    JButton jButton1 = new JButton();

    public EventFrame() {
       super();
       try  {
          jbInit();
       }
       catch (Exception e) {e.printStackTrace(); }
    }
```

Ingen implements ActionListener behövs.

forts.

Exempel: Anonym klass (forts)

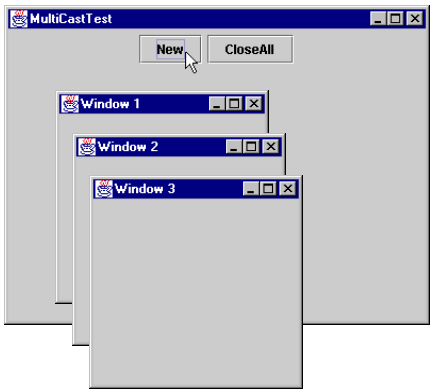
```
private void jbInit() throws Exception {
      this.getContentPane().setLayout(borderLa
      this.setSize(new Dimension(257, 155));
      jPanel1.setLayout(null);
      this.setTitle("JBuilderEvent1Frame");
      jButton1.setText("jButton1"); //Label
      jButton1.setBounds(new Rectangle(59, 32,

      //Registrera framen som lyssnare till knappen: jButton1
       jButton1.addActionListener(
            new java.awt.event.ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    jButton1_actionPerformed(e);
                }
            } );
    this.getContentPane().add(jPanel1, BorderLayout.CENTER);
     jPanel1.add(jButton1, null);
  }

  void jButton1_actionPerformed(ActionEvent e) {
      System.out.println(e.getActionCommand());
  }
}
```

Ingen implements ActionListener behövs.

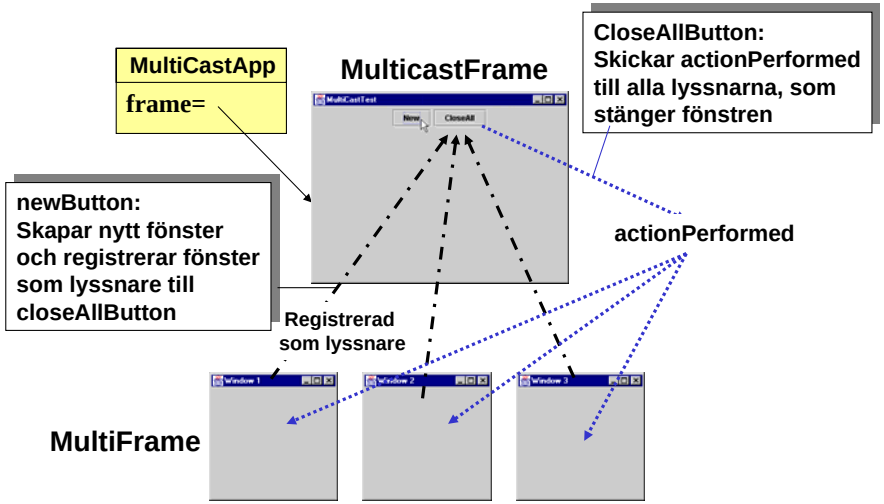
Skapar en ny ActionListener som anonym klass. (Registrerar "*framen*" *(den anonyma/lokala klassen)* som lyssnare)

Skriver ut: **jButton1** se //Label

Multicasting exempel



Multicasting exempel



MultiCastAppl

```
import java.awt.*;
import java.awt.event.*;

public class MultiCastAppl {

  public MultiCastAppl() {
    Frame frame = new MulticastFrame();
    //Registrerar applikationen som lyssnare till MulticastFrame
    //windowClosing
    frame.addWindowListener(
         new WindowAdapter() {
             public void windowClosing(WindowEvent e) {
                 System.exit(0);
             }
         } );
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    new MultiCastAppl();
  }
}
```


MulticastFrame

```
import javax.swing.*;import java.awt.*;import java.awt.event.*;
public class MulticastFrame extends JFrame {
   BorderLayout borderLayout1 = new BorderLayout();
   JPanel jPanel1 = new JPanel();
  JButton newjButton = new JButton();
  JButton CloseAlljButton = new JButton();
  int counter = 0;

  public MulticastFrame() {
    super();
    try  {jbInit();}
    catch (Exception e) {e.printStackTrace();}
  }

  private void jbInit() throws Exception {
     this.getContentPane().setLayout(borderLayout1);
     this.setSize(new Dimension(400, 300));
     this.setTitle("MultiCastTest");
    newjButton.setText("New");
    newjButton.addActionListener(
      new java.awt.event.ActionListener() {
          public void actionPerformed(ActionEvent e) {
              newjButton_actionPerformed(e);
          }
      } );
```

forts.

MulticastFrame (forts.)

```
    CloseAlljButton.setText("CloseAll");
    this.getContentPane().add(jPanel1, BorderLayout.CENTER);
    jPanel1.add(newjButton, null);
    jPanel1.add(CloseAlljButton, null);
  }

  void newjButton_actionPerformed(ActionEvent e) {
      MultiFrame f = new MultiFrame();
      counter ++;
      f.setTitle("Window " + counter);
      f.setSize(200,200);
      f.setLocation(30*counter, 30*counter);
      f.show();
      CloseAlljButton.addActionListener(f);
  }
} //Slut MulticastFrame
```


MultiFrame

```
import javax.swing.*;import java.awt.*;import java.awt.event.*;
public class MultiFrame extends JFrame
                          implements ActionListener {
  BorderLayout borderLayout1 = new BorderLayout();
  JPanel jPanel1 = new JPanel();

  public MultiFrame() {
    super();
    try  {jbInit();}
    catch (Exception e) {e.printStackTrace();}
  }

  private void jbInit() throws Exception {
    this.getContentPane().setLayout(borderLayout1);
  }

  public void actionPerformed(ActionEvent evt) {
      dispose();
  }
}
```