Arv, Grafiska användaregränssnitt och Inre klasser

Förelasning 12

TDA540 – Objektorienterad Programmering

CHALMERS

Mer om Arv

Har tidigare konstaterat att alla klasser ärver (inherit) klassen Object om inget annat anges

- Men om vi vill kan vi ange en annan förälder (superklass)
- Vid klassdeklarationen anger vi att vår klass **ärver (extends)** en annan, befintlig klass (ej Object)
- Klassen som ärver (vår klass) kallas **subklass**
- I subklassen kan vi använda attribut och metoder från superklassen som inte har modifieraren private (dock inte konstruktorn)
- Attributen och metoderna syns inte i subklassen, men de finns och kan användas.
- I subklassen kan man använda **super** för att referera till superklass-(del)objektet (bara i ett steg inte super.super….)
- Finns en speciell modifierare **protected** som innebär att bara subklasser kan använda metoder/instansvariabler från superklassen

Varför Arv?

Genom arv får vi tillgång till den existerande koden i superklassen

- Kan röra sig om tusentals rader! Vi sparar massor med tid!
- Slipper lära oss detaljer, vi kan jobba på en högre nivå (med färdiga metoder)
- Eventuellt finns samma (liknande kod) i flera av våra egna klasser. Genom att flytta denna till en (egen) gemensam superklass (ett enda ställe) blir det lättare att ha kontroll (vi slipper samma kod på två ställen, ingen redundans)

Subklasser och Initiering

Då vi instansierar en klass anropas automatiskt konstruktorn för att ge oss en möjlighet att initiera objektet

- Om vi instansierar en subklass anropas alltid superklassens konstruktor automatiskt (osynligt) först i subklassens konstruktor
- Ger oss möjlighet att initiera superklassens instansvariabler
- Vid explicit anropa av superklassens konstruktor använder vi **super(..),** måste ske först i subklasskonstruktorn
- Om superklassen inte har en default-konstruktor (parameterlös) så måste vi anropa super-klassens konstruktor med något argument

Arv och Typsystemet

Klasser och gränssnitt introducerar nya typer. Samma sak gäller för subklasser!

- En subklass introducerar en typ (en subtyp till supertypen)
- Subklassobjekten kan alltid minst lika mycket som superklassobjekten (den ärver ju allt, ... har samma beteende som super eller “bättre”)
- Ett objekt av superklasstypen kan alltid bytas ut mot ett objekt av subklasstypen de är typkompatibla
- Tvärtom kan leda till problem och godkänns därför inte av typsystemet

Arv och Typkompabilitet

Givet: Alla klasser ärver Object (är subtyper till Object)
Givet: Alla typer är kompatibla med sina supertyper

Slutsats: Alla typer är kompatibla med Object

```
// Sub/superclass compatibility
Object o = new Tile();  // Ok, Object supertype

// Can only call Object's methods for o (that's
// what the type system can guarantee, though
// in reality o can do more)
```

Arv och Typomvandling

Typomvandling mellan sub och super-klasser

- Sub till super inga problem (sker automatiskt), de är typkompatibla
  Objektet (sub) kan minst lika mycket som typen (super) anger
- Super till sub godkänns inte av typsystemet. Objektet (super) kan sakna metoder som finns i typen (sub)
- Vi kan tvinga typsystemet att godkänna super till sub genom att använda explicit typomvandling (casting)
- Genom detta tar vi över ansvaret för att det skall fungera

```
    // Casting
Super sup = get(); // If sup references a sub object
Sub sub = (Sub) sup; //… then this will work …
                                        // … else exception!!
```

Arv och Gränssnitt

Vet sedan tidigare att ArrayList och LinkedList båda implementerar gräsnsnittet List

- Att implementera ett gränssnitt räknas också som arv d.v.s. …
- … klassen som implementerar gränssnittet räknas som en subklass till gränssnittet
- ArrayList och LinkedList är subklasser till List
- Därför är de också typkompatibla med List

Exekveringstyp

Vi kan ha Super s = new Sub();

- s deklarerad som typ Super men objektet är av typ Sub (s kan även vara SubSub, SubSubSub, o.s.v. alla är typkompatibla)
- Hur ta reda på objektets “verkliga” typ, **exekveringtypen (runtime type)**?
- Finns färdig metod i alla objekt: getClass()

```
// Find runtime type
A a = new .... // a may be type A (the declared type) or any subtype

 … a.getClass();      // Will give the runtime type

o1.getClass() == o2.getClass()    // Possible to compare runtime types!
```

Likhet ... Igen

Ofta viktigt att objekt har egen equals()-metod

- Åstadkoms genom att överskugga equals()-metoden
- Vi måste definiera vad vi menar med lika (det är upp till oss)
- Förvånansvärt komplicerat (eftersom arv är är inblandat)
- Vi tillåter inte likhet mellan super och subklasser, … detta kontrolleras i vår equals()-metod m.h.a. getClass()

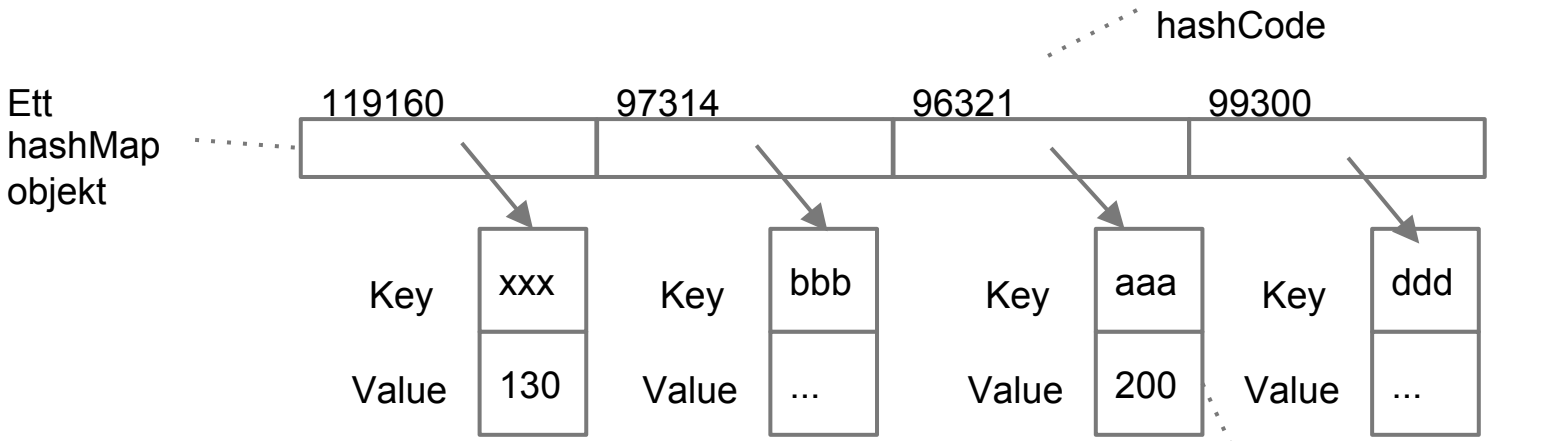
hashCode

Om man skapar en egen equals()-metod skall man alltid skapa en egen hashCode() metod

- hashCode ärvs från Object, metoden ger ett heltal skapat utifrån objekts minnesadress inte garanterat unikt!
- **Krav:** Om o1.equals(o2) sant så skall gälla att o1.hashCode() == o2.hashCode()
- Om inte detta uppfylls blir det problem då objekt sparas i tabeller (Map), olika hashCode gör att lika (equals) objekt inte kommer att hittas … nästa bild…
- Många standarklasser överskuggar hashCode, t.ex. String*)

*) hashCode för String $h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$

Map




```
Map<String, Integer>  m = new HashMap<>();
m.put("aaa", 200);    // Sparas vid 96321
int i = m.get("aaa") // Använd hashCode av "aaa"
                     // för att hitta startpunkt därefter
                     // equals
```

Använder equals för att hitta i kedjan av objekt

Implements och Extends

En Java klass kan implementera flera gränssnitt (implements) men bara (direkt) ärva en klass (extends)

- D.v.s. en klass kan uppfylla många kontrakt men bara återanvända kod från en klass (superklassen, som iof kan innehålla saker från sin superklass …)
- Vissa av våra klasser kommer att både ärva andra klasser och implementera vissa gränssnitt
- Både gränssnitt och klasser introducerar typer (är typer).

Paus

15 min

Grafiska Användargränssnitt

Grafiska användargränssnitt (graphical user interface, GUI) gör det möjligt för användaren att, på ett enkelt sätt, interagera med vår modell

- Användargränssnittet består av fönster, knappar, menyer, o.s.v. (istället för en kommandorad)
- Ett GUI anses överlägset från ett användarperspektiv
- Detta är alltså något helt annat än de grässnitt vi tidigare talat om (specifikation/kontrakt)

Gränssnitt: Där två”saker” möts (olika klasser möts eller människa-dator)

GUI Programmering

Att programmera GUI:n är icke-trivialt

- En subdisciplin till programmering, en specialisering
- Kräver detaljkunskaper specifika för området
- Fokus för denna kurs är inte GUI:programmering men
- … vissa delar är generella och kan leda till ökad förståelse (händelsestyrda program)
- … det är ganska rolig när man får till det

SUMMA: Vi försöker minimera detta

Java Plattformen

Java är inte bara ett språk utan en s.k. **plattform**

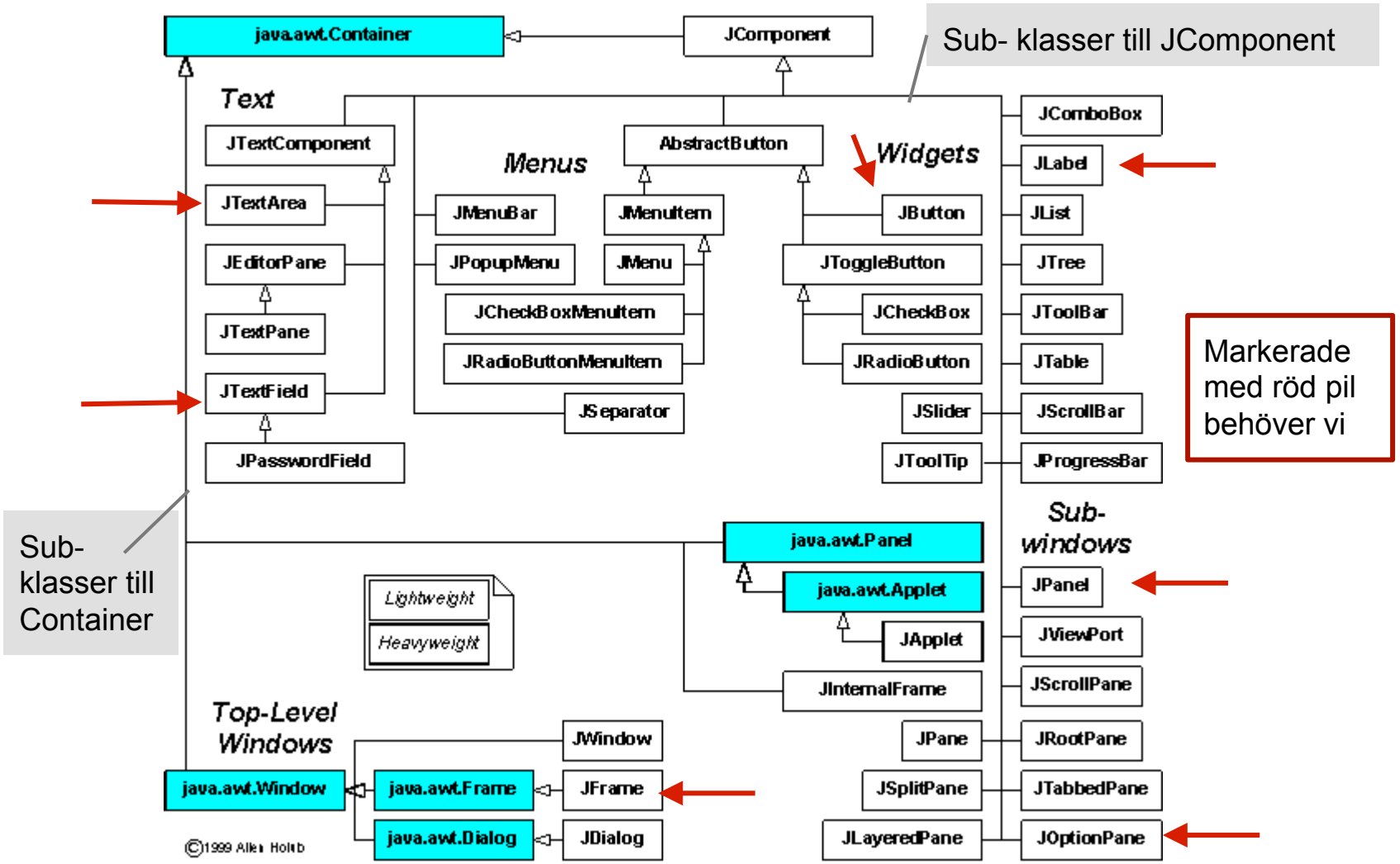
- Förutom språket finns en otrolig mängd färdiga klasser samlade i **bibliotek** (kallas också API:n, application programming interface)
- Vi kan använda färdiga klasser direkt i våra program t.ex. Scanner, olika samlingar, ...
- Genom att använda färdiga klasser sparar vi tid och får en högre kvalitet (koden är testad, buggar fixade)
- Finns även klasser för nätverkskommunikation, databaser, ….”allt”.. !
- Vi skall titta på Java’s klasser för grafiska användargränssnitt

Swing och AWT

De klasser som kan användas för att skapa GUI:n är samlade i **paketen** javax.swing.* och java.awt.*

- Vi säger Swing och AWT
- Vi använder bara Swing (förutom enstaka klasser ur AWT, AWT är föråldrat, omodernt, ...)
- Båda paketen måste importeras för att kunna användas
- Paketen innehåller klasser för fönster, paneler (delfönster), komponenter (knappar, textrutor, …), ramar (borders) och layout m.m.
- Swing och AWT-klasser innehåller ofta en otrolig mängd metoder (för utseende, fonter, färger, position, m.m.)
- Man får prova sig fram tills man hittar det man vill ha
- … ibland beter sig Swing/AWT “konstigt” (mer senare …)

Klasser för Fönster och Komponenter



JFrame

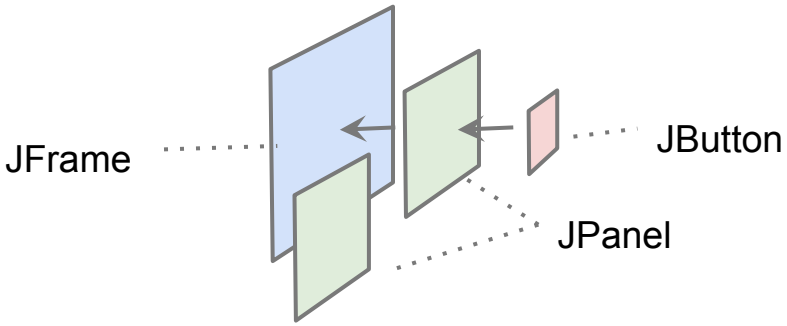
För att få ett fönster på skärmen måste vi implementera en klass som ärver **JFrame**

- För att fönstret skall visas måste vi anropa metoden setVisible(true) (som vi alltså ärver från JFrame)
- Normalt sätter vi också en del andra egenskaper i konstruktorn, storlek, position på skärmen, titel, vad som händer då fönstret stängs (closeOperation), …
- Vi placerar inte komponenter direkt i JFrame:en, istället använder vi paneler (JPanel) …

JPanel

Vi vill att GUI:et skall byggas upp av lagom stora “delar”

- Delarna byggs upp m.h.a. JPanel klassen
- En JPanel för varje syfte (t.ex. visa data för spelare, namn, poäng, …)
- Paneler skapas i metoder som anropas i JFrame:s kontruktor …
- … därefter läggs panelerna “på” JFrame:en (ibland i “lager på lager”)
- På panelerna lägger vi textrutor knappar o.s.v



LayoutManagers

JFrame och JPanel kan ha en layout manager.

- Layoutmanager:n ordnar komponenter i en viss 2d layout

BoxLayout: I en enda rad eller kolumn

FlowLayout: I rad efter rad (om inget anges gäller denna)

GridLayout: I ett rutmönster (rader och kolumner anges)

BorderLayout: nord, syd, väst, öst och center


```
myPanel.setLayout(
new BorderLayout())
```

Normalt måste man lägga till komponenter i rätt ordning (för att det skall bli som man vill)

Storlek och Omritning

Som tidigare nämnts så kan Swing ibland uppföra sig “konstigt”

- Exakt vad detta beror på kan vara mycket komplext att reda ut
- En enkel lösning vi kan använda är att tvinga fram en omritning av GUI:et, … brukar lösa problemen

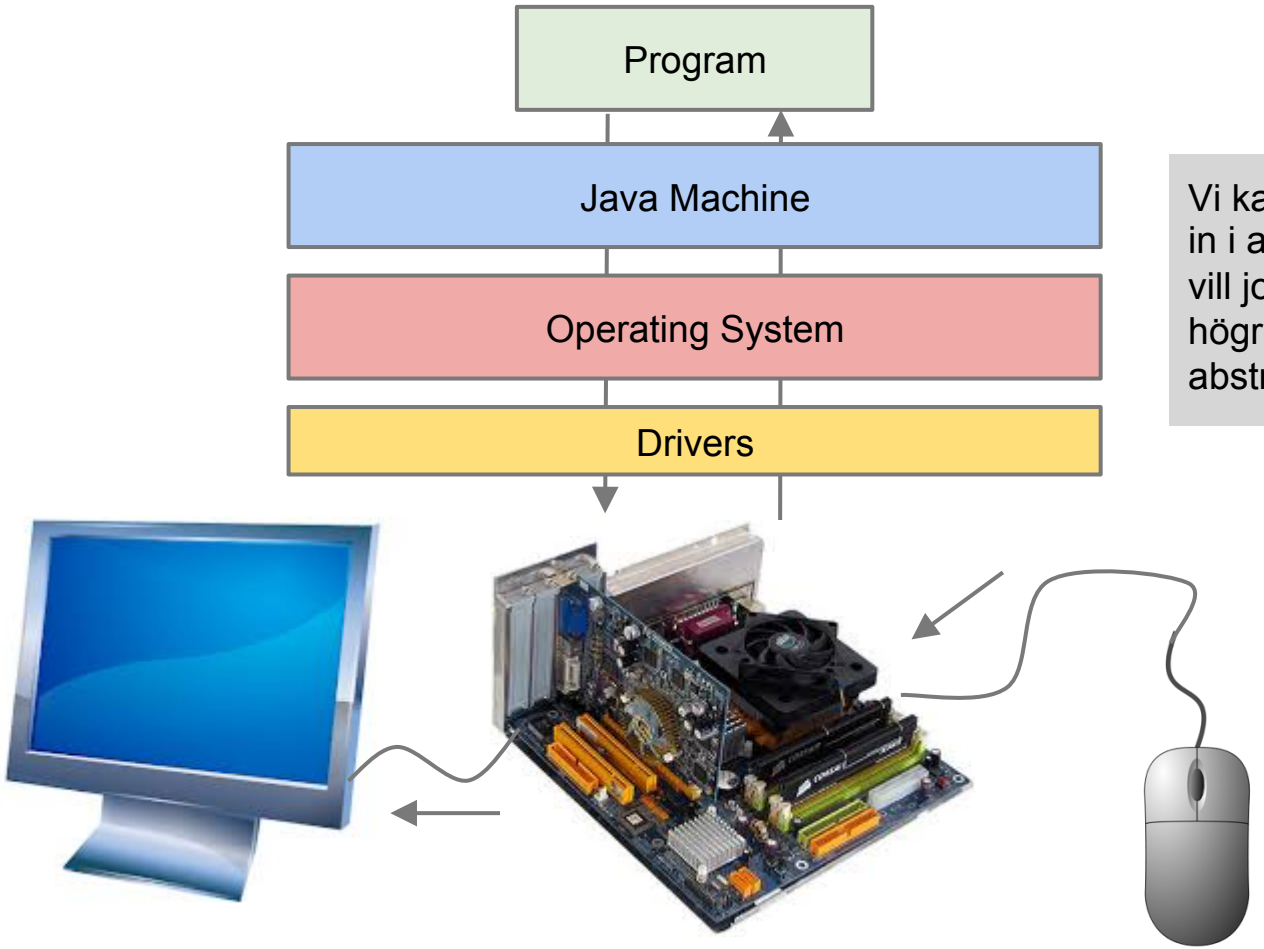
```
// Make GUI behave like we want
// Last in constructor
this.setSize(200, 300);     // width, height
// or  this.pack() will set an overall size depending
// on added components sizes

// After some manipulation of GUI, will force update of GUI
this.validate();
this.repaint();
```

Händelsestyrda program

Program med grafiska användargränssnitt är **händelsestyrda**

- De reagerar på händelser (klick, peka på, o.s.v)
- Dessa händelser genereras automatisk av operativsystemet
- Dessutom informeras Javas händelsesystem automatiskt om vad som hänt (t.ex. “klick på knappen OK”)
- Summa: Vår program kommer att ta emot automatiskt levererade GUI-händelser (i form av händelse-objekt)
- Helt annan struktur än en kommandorad

Teknik för Händelsestyrning



Vi kan inte sätta oss in i alla detaljer, vi vill jobba på en högre abstraktionsnivå

Händelsehantering

I vår program kan vi se det som att komponenter genererar händelseobjekt när de aktiveras (vi klickar, pekar o.s.v.)

- Händelseobjekten håller information om vad som hänt, vilken komponent som aktiverades, antal klick, m. m.
- Händelseobjekt kan “tas emot” i vissa metoder (sker helt automatiskt).
- Objekten levereras som inparameter till dessa. Metoderna lyssnar efter genererade händelseobjekt.
- Metoder som kan ta emot händelseobjekt specificeras av olika gränssnitt t.ex. MouseListener eller ActionListener
- Klasser som skall kunna ta emot händelseobjekt måste implementera något av dessa gränssnitt, d.v.s. garantera att det finns **lyssnar-metoder** som kan ta emot händelseobjekten
- En klass som har lyssnarmetoder kallar vi **lyssnare**

Komponent och Lyssnare

Genom att koppla en lyssnare till någon komponent kan vi i programmet ta emot “kommandon” i form av händelseobjekt

- Innebär att vi måste ha en lyssnarmetod i klassen

```
public void actionPerformed(ActionEvent e) { …}
```

- För att koppla lyssnaren till t.ex. en knapp används

```
// this objektet måste ha metoden actionPerformed
// Garanteras genom att klassen för this implementerar
// grässnittet ActionListener
button.addActionListener(this);
```

Styrning av Händelsehantering

Man kan slå på och av komponenter

- Metoden setEnabled(boolean b)
- Vi kan styra vilka möjligheter användaren har ...
- … men skall inte användas för programlogik!
- Modellen skall alltid kontrollera och hantera datan (allt logiskt sker i modellen), GUIet sköter bara in och utmatning
- Komponenten ritas “grå”:ad

Konstruktion av GUI

GUI:et byggs “panel för panel” i huvudfönstret

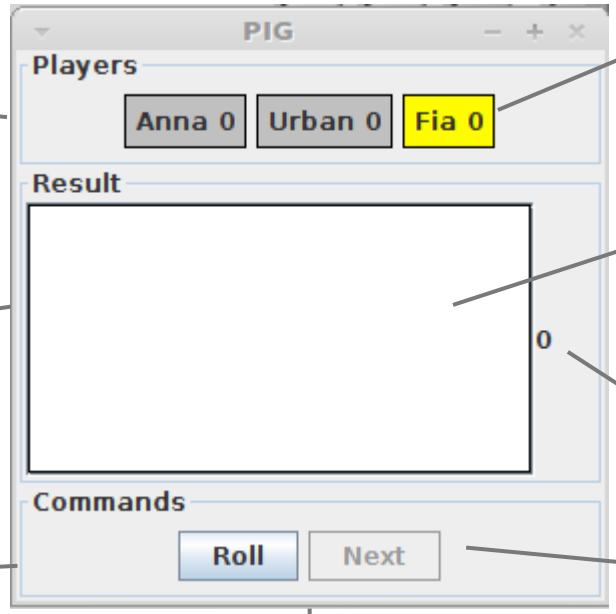
- Varje panel skapas av en metod
- Sammansättning av paneler sker oftast i en konstruktor (eller i förälderpanelen ifall en panel har barnpaneler)
- En del paneler behöver referenser till modell objekt, skapa isf get()-metoder i modellen
- Vissa delar i GUI:et hör ihop med vissa modell objekt …
- … i sådana fall fungerar en Map bra.

GUIPig

JPanel med TiteledBorder och FlowLayout. Position BorderLayout.NORTH

JPanel med TiteledBorder och BorderLayout (bara center och öst används). Position BorderLayout.CENTER

JPanel med TiteledBorder och FlowLayout. Position BorderLayout.SOUTH



PlayerPanel (sammansatt av 2 JLabel)

JTextArea (center)

JLabel (east)

JButton (disabled)

Hela fönstret är en JFrame med BorderLayout

Konstruktion av Händelsehantering

Den enda lyssnaren är vårt huvudfönster

- .. som alltså implementerar ActionListener och har metoden actionPerformed()
- Alla komponenter som skall skicka händelser kopplas till huvudfönstret (och därmed lyssnarmetoden)
- Görs i metoderna som skapar paneler (den panel där vi placerar t.ex. en knapp)

Ett Grafiskt användargränssnitt till Gris

Vi skall nu kunna skapa ett grafiskt användargränssnitt till spelet Gris

- Obs! Att vår modell är helt oförändrad
- All logik är ju den samma, det är bara visa delar i omgivningen till modellen som förändras (render-metoden och kommandoraden)
- … tanke… vi skulle kunna göra en Gris-app eller en Gris-web app (om vi lär oss hur man skapar GUI i Android/iOS eller HTML)
- Även konstruktionen av modellen är oförändrad (buildPig-metoden)
- Programmet har ett enda huvudfönster, GUIPig (som ärver JFrame)
- Huvudfönstret ersätter den tidigare omgivningen (CommanLinePig) enlig
  - Kommandoraden ersätts med händelsestyrning
  - render-metoden, sköta automatiskt av Swing

Styrning av Modell via GUI

De händelser som genereras skall leda till anrop på modellen

- På samma sätt som för kommandoradsversionen
- I lyssnarmetoden kan vi avgöra vilken knapp som klickats (e.getSource())
- Om det är Roll så anropas pig.roll(). GUI:et uppdateras med resultatet. Om resultatet blir 1 stänger vi av Roll och slår på Next
- När man klickat Next uppdateras aktuell spelars totalpoäng och ny aktuell spelare markeras. Därefter stänger vi av Next och slår på Roll
- Då man stänger fönstret avslutas programmet (EXIT_ON_CLOSE)

Inre klasser

I Java är det möjligt att deklarera inre klasser d.v.s. en klassdeklaration inuti en annan

- Om klasser på något sätt “hör ihop”

Användning

- I Gris-spelet behöver vi koppla en spelare till en spelarpanel (bl.a. för att visa spelarens poäng) …
- ... istället för en enda JLabel för poäng skapar vi en inre klass, PlayerPanel, för all data om en spelare (enklare att ha allt på ett ställe)
- PlayerPanel implementeras som en subklass till JPanel (så att den kan visas i GUI:et)
- I GUI:et kopplar vi en spelare till en PlayerPanel (i en Map) så att vi kan slå upp den aktuella spelarens panel (för att t.e.x skriva ut poäng)