Grafiske brukergrensesnitt med Swing og AWT

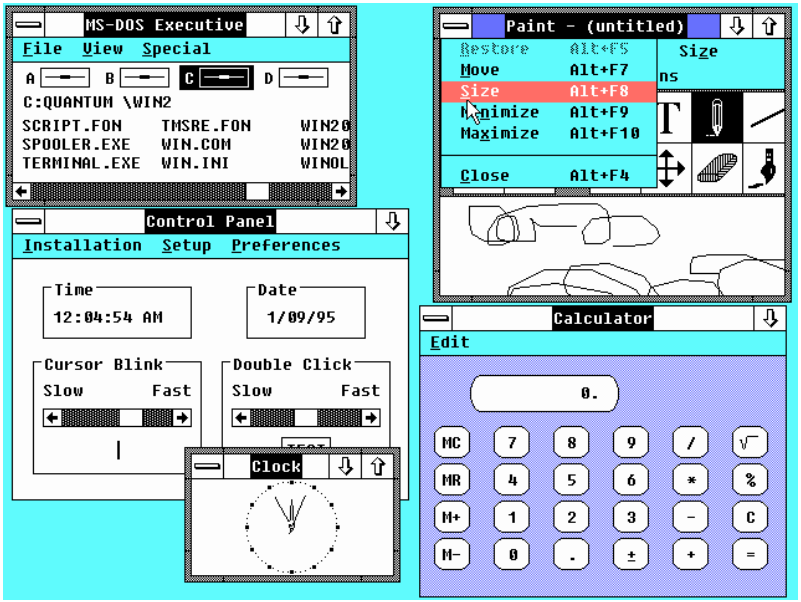
–

INF1010

INNHOLD

Introduksjon

Da Windows versjon 1.0 kom i 1985 som det første operativsystemet som i hovedsak var basert på brukerinteraksjon med et grafisk, vindusbasert brukergrensesnitt (i motsetning til en terminal med tekstinteraksjon), skapte det en revolusjon. Tidligere var bruk av datamaskin forbeholdt profesjonelle med opplæring innen IT og databehandling; nå ble datamaskinen, eller «PC» — Personal Computer — noe alle kunne bruke. Fra Vim og Emacs på terminallinjen, til MS Office med Word, PowerPoint og Excel, og fra kryptiske terminal-kalkulatorer med polsk notasjon, til kalkulatorvindu med knapper, mus og tastatur.



Siden den gang har utviklingen gått langt. Selv om Windows 1.0 i sin tid var revolusjonerende, vil nok de fleste se på det som et primitivt operativsystem i dag. Arven fra Windows 1.0 har vi likevel fortsatt. Den dag i dag, finnes det ingen operativsystemer beregnet på gjennomsnittsbrukeren, som ikke har et grafisk, vindusbasert brukergrensesnitt. Nettopp derfor er det viktig at vi, som programmerere, har kunnskap om utvikling av grafiske brukergrensesnitt. Selv om Javas innebygde GUI-biblioteker, Swing og AWT, ikke brukes i særlig stor grad i større prosjekter og applikasjoner i dag, er det nyttig å ha vært borti bruk av grensesnitt for GUI. Nesten alle operativsystemer har sine egne GUI-biblioteker. Windows har .NET, OS X har Cocoa, Android har UI, iPhone har CocoaTouch, og Linux har flere tusen forskjellige. Til felles har de aller fleste tankegangen og måten vinduene bygges opp på. Lærer man seg å bruke ett bibliotek, er det mye lettere å sette seg inn i bruk av et annet.

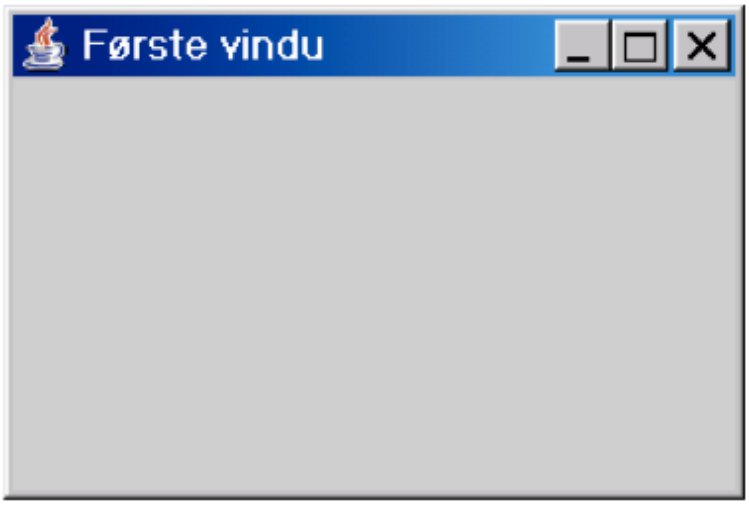
I dette notatet skal vi se på bruk av Javas Swing og AWT-pakker, og hvordan enkle grafiske brukergrensesnitt kan bygges opp ved hjelp av disse. Swing og AWT, som Java ellers, kan kjøres på de fleste operativsystemer og datamaskiner, gjennom Java-maskinen (JVM). Dette har den opplagte fordelen at koden kan skrives og kompileres ett sted, og kjøres omtrent over alt. Med tanke på GUI, har det også en ulempe; fordi programmet må være så portabelt som mulig, støtter det i liten grad bruk av et operativsystems eget vindussystem. Vinduselementer som knapper og bokser i Swing, ser ofte litt annerledes ut enn operativsystemets innebygde vinduselementer, og avanserte OS-spesifike vinduselementer støttes i liten grad. Siden vi i INF1010 ikke skal lage veldig store vindusprogrammer med avansert funksjonalitet, gjør ikke det noen ting for oss.

Vinduer, rammer og paneler

Alle standardbiblioteksklasser som nevnes i denne delen hentes fra **javax.swing.***, **java.awt.***, og **java.awt.event.***, dersom ikke annet er nevnt. For å importere dette i ditt Javaprogram, bruk følgende kode før alle klassedeklarasjoner:

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

Alle Swing-programmer trenger en ytre ramme. Uten denne vil ikke programmet kunne vise noe som helst. Rammen er et objekt av klassen **JFrame** (eller en subklasse av denne). En **JFrame** er i utgangspunktet kun et tomt vindu, og for å vise noe som helst, er vi nødt



til å lage noe vi kan legge inn i rammen.
Selv om det er mulig å legge vinduselementer «rett på» rammen, er det nesten alltid ønskelig å først opprette et eller flere **JPanel**, fylle dem med alle elementer man måtte ønske, og så legge dem på rammen til slutt. Dette gjør programmeringen en god del enklere for oss, fordi vi ikke i like stor grad trenger å tenke over hva som skal settes som «visible» når (mer om «visible» senere), og ikke minst fordi det i større programmer gjør det lettere å lage et pent og ryddig layout. Å legge til elementer i en **JFrame** og en **JPanel** gjøres på samme måte; begge klasser har en metode ***add(Component comp)***, som tar en GUI-komponent som argument.

I noen tilfeller vil det være en god idé å la en GUI-klasse være en subklasse av enten **JPanel** eller **JFrame**. Da kan man legge komponenter rett på «seg selv», uten å gå veien om å først opprette et objekt, for så å bruke pekerreferansen til denne hver gang man skal legge noe til. Vær likevel varsom — det er lett at noe havner på feil sted, og i større programmer fører dette ofte mye mer rot med seg, enn å la én klasse holde styr på

rammen og alle separate paneler og deres komponenter. For små programmer, som dem vi lager i INF1010, kan det likevel være greit, for å spare noen få linjer med kode.

Her er et eksempel på en klasse som oppretter et panel, legger det på en ramme, og viser rammen for bruker.

```
public class MittGUI() {

	JFrame ramme;
	JPanel panel;

	public MittGUI {
		ramme = new JFrame("Første vindu");
		panel = new JPanel();

		ramme.add(panel);

		ramme.setSize(100, 100);
		ramme.setVisible(true);
		ramme.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}
}
```

Det er flere ting å merke seg her. Vi ser først at ramme- og panel-pekerne opprettes. I konstruktøren settes disse til å peke på objekter av henholdsvis **JFrame** og **JPanel**. Vi ser også at konstruktøren til **JFrame** tar en **String** som parameter. **JFrame**, **JPanel**, og stort sett de fleste innebygde GUI-klassene, har utallige konstruktører. For å finne ut hvilken du vil bruke når du programmerer, er du nødt til å bla opp i Oracle sitt bibliotek på internett (lenke i appendix). Den vi benytter oss av her tar, som sagt, en **String**. Denne tekststrengen avgjør hva som står i programmvinduets topp-linje, dersom operativsystemet har en slik. Videre kan vi merke oss at vi gjør kall til rammens metoder ***setSize(int x, int y)***, ***setVisible(boolean value)***, samt ***setDefaultCloseOperation(int op)***. Førstnevnte metode forteller Java-maskinen hvor stort vi ønsker at vinduet skal være når det vises på skjermen. Merk at dette er antall piksler*antall piksler. En typisk skjerm har omkring 1920*1080 piksler, men dette varierer fra maskin til maskin. ***setVisible(boolean value)*** forteller programmet om vi ønsker at vinduet skal være synlig for bruker. I utgangspunktet er vinduet IKKE visible, det er derfor veldig viktig å huske å gjøre den visible — ellers vises ingenting. ***setDefaultCloseOperation(int op)*** sier noe om hva som skal skje dersom bruker trykker på X-merket i GUI-vinduets topplinje. Denne tar en *int* som argument, men forventer en verdi som «gir mening» for metoden (du kan altså ikke sende med 1337 og håpe at det skal skje noe gøy). Alle de forventede verdiene finnes definert som *final int* i **JFrame**. Den vi i hovedsak benytter oss av i INF1010, og som sørger for at programmet avsluttes dersom X'en trykkes, heter **JFrame.EXIT_ON_CLOSE**. Programmet vi har skrevet her, vil vise et vindu som ser ut som det på forrige side (selv om det vil variere litt fra OS til OS).

Komponenter og vinduselementer

Nesten alle vinduselementene vi ser i Swing, er subklasser av JComponent. JComponent er igjen en subklasse av Component. Vi har tidligere sett på metoden ***add(Component comp)***, som vi finner igjen i både **JFrame** og **JPanel**. Vi ser at den tar en **Component** som argument. Det betyr at en subklasse av **Component**, for eksempel **JComponent**, vil fungere utmerket. Det finnes veldig mange komponenter i Swing, og i INF1010 lærer vi å bruke noen få av dem.

JTextField

Komponenten **JTextField** viser en liten ramme med tekstmarkør. I denne rammen kan bruker taste inn sin egen tekst. Programmet kan også sette teksten i tekstfeltet selv, ved bruk av funksjonen ***setText(String s)***. Det vil i mange tilfeller være ønskelig å hente ut teksten fra tekstfeltet, for eksempel for å lagre den til fil, eller vise den et annet sted. For å hente ut teksten som er tastet inn i tekstfeltet, brukes ***getText()***.

JLabel

En **JLabel** er et statisk tekstfelt. Teksten i feltet kan ikke endres direkte av bruker, og med mindre man spesifiserer det selv, vises ingen hvit/farget bakgrunn og tekstmarkør, slik det gjør i **JTextField**. **JLabel** er nyttig for visning av tekst til bruker, og benyttes ofte der det ikke er meningen at bruker skal endre teksten. Teksten endres av programmet ved kall til ***setText(String s)***.

JButton

En **JButton** er en knapp som bruker kan trykke på. For at noe skal skje når bruker trykker på knappen, trengs en klasse som implementerer grensesnittet **ActionListener**. Mer om knapper og knappelytting senere.

På neste side følger et kodeeksempel på en klasse med en ramme som inneholder et panel, som igjen inneholder en **JTextField**, en **JLabel** og en **JButton** (som ikke har noen funksjon knyttet til seg enda).

```
public class MittGUI {

	JFrame ramme;
	JPanel panel;
	JTextField tekstfelt;
	JLabel tekst;
	JButton knapp;

	public MittGUI() {
		ramme = new JFrame("Første vindu");
		panel = new JPanel();
		tekstfelt = new JTextField();
		tekst = new JLabel();
		knapp = new JButton();

		tekstfelt.setText("JTextField");
		tekst.setText("JLabel");
		knapp.setText("JButton");

		panel.add(tekstfelt);
		panel.add(tekst);
		panel.add(knapp);

		ramme.add(panel);
		ramme.setSize(300, 60);
		ramme.setVisible(true);
		ramme.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}
}
```

Resultatet av dette vil se slik ut (på Mac OSX):

Knapper og knappelyttere

Vi har allerede sett på hvordan vi oppretter en knapp ved hjelp av Swing. Men hva er egentlig en knapp, dersom det ikke skjer noe når den trykkes på? Nå skal vi se på hvordan vi kan få knappen til å gjøre noe.

For å få noe til å skje når en knapp trykkes på, må vi fortelle programmet vårt *hva* som skal skje. Dette gjøres ved å kalle **JButton** sin funksjon ***addActionListener(ActionListener l)***. For å forstå hvordan denne funksjonen fungerer, er vi nødt til å vite hva en **ActionListener** er. **ActionListener** er et grensesnitt som finnes i *java.awt.event*. Grensesnittet definerer én funksjon, *public void actionPerformed(ActionEvent e)*. Idéen her, er at hver knapp skal inneholde en peker til et objekt av en klasse som implementerer denne typen. Hver gang knappen trykkes, kjøres funksjonen *actionPerformed* inne i lytterklassen.

Det er ofte en fordel å implementere lytterklassen som en indre klasse til GUI-klassen. Da har den tilgang til lokale variable direkte, og kan gjøre kall til andre funksjoner i GUI-klassen, og i objekter GUI-klassen har pekere til. Her er et eksempel på hvordan dette kan gjøres:

```
public class MittGUI {

      JButton knapp;

      public MittGUI() {
            <Gjør standard ramme- og panel-operasjoner>

            knapp = new JButton();
            knapp.setText(«Trykk meg");
            knapp.addActionListener(new Lytter());

            <Gjør flere standard ramme-og panel-operasjoner>
      }

      protected class Lytter implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                  System.out.println("Noen der?");
            }
      }
}
```

En annen måte i implementere en ActionListener på, er ved bruk av en «indre anonym klasse». Denne måten er litt mer kryptisk, og kanskje litt vanskeligere å forstå. Kort sagt er det som skjer at man oppretter en «mini-klasse» direkte i metodekallet på ***addActionListener***. Den kan spare deg for noen linjer kode, men anbefales ikke dersom du ikke forstår fullt og helt hva som skjer. Her er et eksempel:

```
public class MittGUI {

	JButton knapp;

	public MittGUI() {
		<Gjør standard ramme- og panel-operasjoner>

		knapp = new JButton();
		knapp.setText(«Trykk meg");
		knapp.addActionListener(new ActionListener(){
			public void actionPerformed(ActionEvent e) {
				System.out.println("Noen der?");
			}

		});

		<Gjør flere standard ramme-og panel-operasjoner>
	}
}
```

Som du sikkert ser, tar *actionPerformed* et argument av typen **ActionEvent**. Denne klassen behøver vi ikke forholde oss til i særlig stor grad i INF1010. Det kan likevel være verdt å nevne at den har en metode for å sjekke *hvilket objekt* som kalte *actionPerformed*. Dette kan være praktisk dersom man har én lytterklasse lenket opp mot mange knapper. Metoden som da benyttes er ***ActionEvent.getSource()*** (biblioteklenke i appendix).

Layout

Til nå har vi sett miniprogrammer der vinduselementer har vært lagt til på et panel i vilkårlig rekkefølge, ved bruk av *add*. Selv om dette fungerer fint og elementene legger seg pent etter/over og under hverandre, er det ofte ønskelig å ha litt mer kontroll over hvordan elementene skal legge seg i forhold til hverandre. Heldigvis for oss, har Swing støtte for en del forhåndsdefinerte *layouts*.

Layouts er klasser som avgjør hva som skjer når vi sier *add* i et panel eller en ramme. Når man oppretter et panel eller en ramme, benyttes **FlowLayout** dersom vi selv ikke ber om å få noe annet. For å endre layout, gjør vi kall til funksjonen ***setLayout(LayoutManager mgr)*** som finnes i både **JFrame** og **JPanel**. Det er veldig vikgit at du legger til alle elementene dine etter at du har bestemt hvilket layout som skal brukes — legger du elementer på før du velger layout, risikerer du at ting forsvinner, eller dukker opp på helt feil plass. Som nevnt finnes det en del layouts å velge i blant, men i INF1010 bruker vi bare et knippe. Her er noen få ord om hver av dem.

FlowLayout

FlowLayout er kanskje den mest grunnleggende layout'en i Swing. Elementer du legger til, havner etter hverandre på rekke og rad, så lenge det er plass i vinduet. Er det ikke mer plass på linjen, rykker neste element ned på linjen under. Nesten som når du skriver i et tekstbehandlingsprogram! Du trenger, som nevnt, ikke sette denne layout'en når du oppretter et panel eller en ramme, siden den er standardlayouten i Swing.

GridLayout

GridLayout gir deg et «rutenett» der du kan fylle rutene med dine elementer. Når du oppretter et **GridLayout**, spesifiserer du hvor mange ruter du vil ha på x-aksen og y-aksen ved å sende med to int-verdier til konstruktøren. Når du siden sier *add*, legger elementene seg i første ledige rute (høyre->venstre, ned ved enden av raden).

BorderLayout

BorderLayout lar deg spesifisere hvor i rammen/panelet du ønsker at elementet skal legge seg. Når du kaller *add* og har spesifisert at **BorderLayout** skal brukes, må du i tillegg til elementet, også sende med en *int*. Verdien av *int*'en kan ikke være hva som helst — alle andre enn de forhåndsdefinerte verdiene i **BorderLayout**-klassen fører til at ingenting skjer. Det finnes veldig mange verdier å velge mellom, men de som kan være verdt å merke seg heter SOUTH, NORTH, WEST, EAST og CENTER, og legger elementene hhv. nederst, øverst, til venstre, til høyre og i midten av rammen/panelet. Her er et kalleksempel:
mittPanel.add(minKnapp, BorderLayout.SOUTH);

Appendix

Javas standardbibliotek
http://docs.oracle.com/javase/7/docs/api/

Swing og AWT
http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html
http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

ActionEvent (getSource)
http://docs.oracle.com/javase/7/docs/api/java/awt/event/ActionEvent.html

JFrame og JPanel
http://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html

Forelesninger om GUI
http://heim.ifi.uio.no/~inf1010/v14/lysark/GUI1.pdf
http://heim.ifi.uio.no/~inf1010/v14/lysark/GUI2.pdf

Notatet er skrevet av Henrik H. Løvold, gruppelærer i INF1010 ved Universitetet i Oslo, våren 2014. Notatet er basert på de deler av INF1010-pensumet som omhandler grafiske brukergrensesnitt, med det mål å samle alt eksamensrelevant GUI-stoff på ett og samme sted.