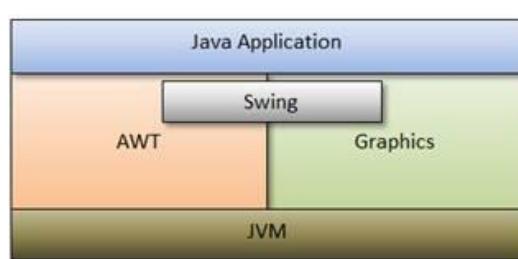
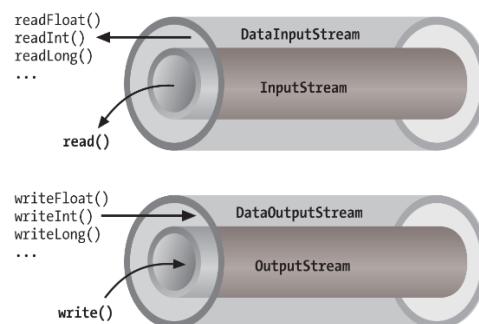
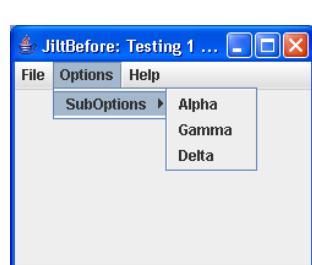


Programmering - Java III (6270)

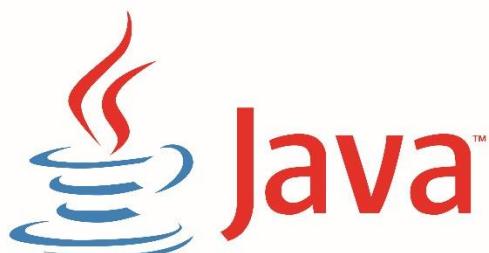
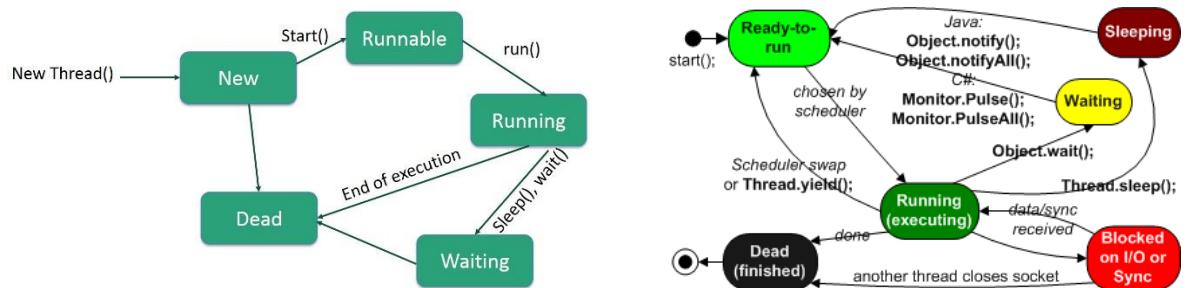
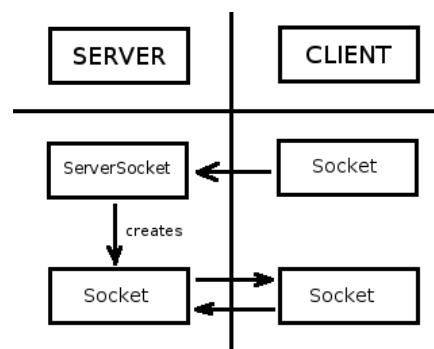
Jan Stern Johansen

```
ic class Threads_01_02 {  
    public static void main(String[] args) {  
        (new Thread(new HelloRunnable())).start();  
        (new HelloThread()).start();  
        Runnable r1 = new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("Hello from " + Thread.currentThread().getName());  
            }  
        };  
        (new Thread(r1)).start();  
    }  
}
```



Java Applets vs. Applications

Feature	Application	Applet
main() method	Present	Not present
Execution	Requires JRE	Requires a browser like Chrome, Firefox, IE, Safari, Opera, etc.
Nature	Called as stand-alone application as application can be executed from command prompt	Requires some third party tool help like a browser to execute
Restrictions	Can access any data or software available on the system	cannot access anything on the system except browser's services
Security	Does not require any security	Requires highest security for the system as they are untrusted



Indhold

Indhold	3
1: Eleven kan anvende sproget til at udvikle programmer, der anvender de objekt-orienterede egenskaber ved Java sproget, såsom nedarvning og polymorfi.	7
Opgave	7
2: Eleven kan anvende fil I/O klasse biblioteker til at læse og skrive til og fra data- og tekstfiler.	8
File	8
File og Scanner	8
Læse fra tastatur	8
Læse fra fil	8
Opret text-fil med PrintWriter	10
Opret binær fil med FileOutputStream	10
Files-klassen	10
Opret og skriv i tekstfil med Files	10
Opret og skriv i binær fil med Files	11
Og læs igen med Files	11
FileReader og BufferedReader	11
Try with resources	12
Listing af filstruktur med klassen File	14
Opgave	15
3: Eleven kan skabe og anvende Java teknologi GUI komponenter: paneler, knapper, labels, tekstmeldinger og tekstrområder.	16

WindowBuilder	16
Problem ☺.....	18
Problem Slut ☺	19
Vi ønsker at oprette et Swing-projekt med en JFrame i	20
Opgave	23
Dialogs.....	24
Indbygget Dialog	24
Custom Dialog	24
Frames.....	26
Jframe.....	26
En JFrame med JTextArea og JButton med eventhandler	26
Avanceret evenhåndtering	28
Klassen EventListenerList	28
Klassen MyFrame	29
Klassen DetailsPanel.....	29
Klassen DetailEvent.....	31
Klassen EventObject.....	31
Constructor	31
Interface DetailListener.....	32
Opgaver	32
JavaFX.....	33
JavaFX Layoutmanagers	34
JavaFX med IntelliJ	40
Eclipse med JavaFX.....	42
Hello World i JavaFX uden FXML.....	44
FXML Button med event	47
Styling af FXML med CSS	55
4: Eleven kan anvende events programmering og Applets, der er baseret på Java-teknologien.	57
Applets	57
Sikkerhed.....	57
Life Cycle of an Applet:	59
En Applet der viser kaldte metoder samt mouse-events	59
En Applet der tegnes på og modtager parametre fra HTML	61
Applet med Button, RadioButton, JTextField og grafik.....	62
5: Eleven kan programmere Standalone Java programmer og bruge Frame og Menuklasser til at tilføje grafik til Java programmer.....	65

6: Eleven kan fremstille programmer, der anvender Multithreading.....	66
En klasse der arver fra Thread	66
En klasse der implementerer Runnable.....	67
Oprettelse af en tråd med anonym klasse der implementerer Runnable.....	68
Volatile keyword	69
Data der manipuleres af flere tråde – synchronized (og join()).....	70
Opgaver	74
Thread pools	75
Eks.	75
CountDownLatch.....	77
BlockingQueue	78
Wait(), notify() og notifyAll()	79
Wait() og notify(), et arbejdende eksempel.....	81
Reentrant Locks	83
ReentrantLock await(), signal() og signalAll().....	85
Dead Lock og tryLock()	88
Semaphore	90
Callable og Future	93
Interrupt af tråde	96
Thread pool.....	99
Opdatering af GUI fra en anden tråd	100
Andre måder at tilgå GUI fra tråde	102
Join().....	103
7: Eleven kan fremstille programmer, der anvender simpel TCP/IP-klient til kommunikation gennem Sockets....	104
Simpel klient/server	104
Server	105
Client	105
Multitrådet server med flere klienter.....	105
Client	107
Client-Server Socket Opgave 1.....	108
8: Eleven kan anvende sproget til udvikling af klient/server system i Java.....	109
9: Eleven kan oprette fjernobjekter ved at bruge Java RMI.	109
Noter	112

- 1:** Eleven kan anvende sproget til at udvikle programmer, der anvender de objekt-orienterede egenskaber ved Java sproget, såsom nedarvning og polymorfi.
- 2:** Eleven kan anvende fil I/O klasse biblioteker til at læse og skrive til og fra data- og tekstfiler.
- 3:** Eleven kan skabe og anvende Java teknologi GUI komponenter: paneler, knapper, labels, tekstfelter og tekstområder.
- 4:** Eleven kan anvende events programmering og Applets, der er baseret på Java-teknologien.
- 5:** Eleven kan programmere Standalone Java programmer og bruge Frame og Menuklasser til at tilføje grafik til Java programmer.
- 6:** Eleven kan fremstille programmer, der anvender Multithreading.
- 7:** Eleven kan fremstille programmer, der anvender simpel TCP/IP-klient til kommunikation gennem Sockets.
- 8:** Eleven kan anvende sproget til udvikling af klient/server system i Java.
- 9:** Eleven kan oprette fjernobjekter ved at bruge Java RMI.

1: Eleven kan anvende sproget til at udvikle programmer, der anvender de objekt-orienterede egenskaber ved Java sproget, såsom nedarvning og polymorfi.

Opgave

Vis indkapsling, nedarvning og polymorfi.

Klassen Animal skal implementere interface Comparable og indeholde

- name, color
- constructor der modtager name og color
- set/getColor()
- doMove()
- toString()
- skal ikke indeholde metoden compareTo() ☺

Klassen Cat skal arve fra Animal og indeholde

- sound()
- compareTo()
- toString()

Klassen Fish skal arve fra Animal og indeholde

- numberOfGills
- og hvad mere ?

2: Eleven kan anvende fil I/O klasse biblioteker til at læse og skrive til og fra data- og tekstfiler.

File

Et File-objekt er en abstrakt representation af filer, mapper og stinavne.

Der oprettes et File-objekt på den angivne sti, hvor Temp er en mappe.

```
File f1 = new File("C:\\Temp");
```

Hvis mappen ikke eksisterer, kan den oprettes med

```
f1.mkdir();
```

Ved hjælp af f1 oprettes nu et File-objekt på stien "C:\\Temp\\JavaTest"

```
File f2 = new File(f1, "JavaTest");
```

Er det et dir eller en fil?

```
f1.isDirectory();
```

```
f1.isFile();
```

Hvis det File-objekterne ikke eksisterer fysisk, vil både isDirectory() og isFile() returnere false.

Returnerer en String-liste af child-navne.

```
f1.list();
```

Returnerer hele stien på File-objektet

```
f1.toString();
```

File og Scanner

Læse fra tastatur

Med Scanner er det muligt på en nem måde at læse fra tastaturet, hvilket ikke er så enkelt med System.in.

```
Scanner scanner = new Scanner(System.in);
String str = scanner.nextLine();
System.out.println(str);
```

Scanner har metoder til forskellige datatyper f.eks.

```
scanner.nextInt();
scanner.nextDouble();
```

Læse fra fil

Med klasserne File og Scanner kan der udføres en simpel læsning på en fil.

Der oprettes først en fil af placering og navn C:/TestFile/TextFile.txt med følgende indhold

3

First line

Second line

Third line

Fourth line

Derefter oprettes følgende Java-projekt.

På grundlag af filens navn og sti oprettes et File-objekt, og på grundlag af denne oprettes et Scanner-objekt.

Fil-stien skal være med for-slash eller dobbelte back-slash.

Med Scanner-objektet kan der spørges om filen har flere linjer, og der kan hentes næste linje.

```
public class FileSimpleReadTest
{
    public static void main(String[] args) throws FileNotFoundException
    {
        String fileName = "C:/TestFile/TextFile.txt";

        File textFile = new File(fileName);

        Scanner in = new Scanner(textFile);

        while(in.hasNextLine())
        {
            String line = in.nextLine();
            System.out.println(line);
        }

        in.close();
    }
}
```

Output:

```
3
First line
Second line
Third line
Fourth line
```

Scanner-klassen har uover nextLine(), der returnerer en String, metoder til at læse alle simple typer.

I det følgende er koden ændret, sådan at der først læses et heltal og derefter læses der et antal linier svarende til det første heltal i filen.

Grunden til, at der læses en linje efter læsnigen af tallet, er, at der i filen både er en CR-karakter og en LF-karakter, og det kun er den første der læses sammen med tallet, så der kommer en tom linje.

Desuden er filen kopieret ind i projektets rod-mappe, og kan derfor refereres til kun med filnavnet. Der kan selvfølgelig også oprettes en mappe til filer i projektets rodmappe og så refere relativt til den.

```
public class FileSimpleReadTest
{
    public static void main(String[] args) throws FileNotFoundException
```

```

{
    String fileName = "TextFile.txt";
    File textFile = new File(fileName);
    Scanner in = new Scanner(textFile);

    int count = in.nextInt();
    in.nextLine();

    System.out.println("Lines to be read: " + count);

    for(int i = 0; i < count; i++)
    {
        if(in.hasNext())
        {
            String line = in.nextLine();
            System.out.println(line);
        }
    }
    in.close();
}
}

```

Output:

```

Lines to be read: 3
First line
Second line
Third line

```

Opret text-fil med PrintWriter

```

try{
    PrintWriter writer = new PrintWriter("C:\\\\TestFile\\\\the-file-name.txt", "UTF-8");
    writer.println("The first line");
    writer.println("The second line");
    writer.close();
} catch (Exception e) {
}
}

```

Opret binær fil med FileOutputStream

```

byte data[] = {12,23,34,45,56,67,78,89,89};
FileOutputStream out = new FileOutputStream("C:\\\\TestFile\\\\the-file-name.byt");
try {
    out.write(data);
    out.close();
} catch (IOException e) {
    e.printStackTrace();
}
}

```

Files-klassen

Opret og skriv i tekstfil med Files

Filen oprettes, overskrives med nyt indhold eller der tilføjes.

(Pas på at det ikke er java.awt.List, der bruges)

```
java.util.List<String> lines = Arrays.asList("The first line",

```

```

        "The second line",
        "And the third line");
Path file = Paths.get("C:\\TestFile\\the-file-name.txt");
Files.write(file, lines, Charset.forName("UTF-8"));
Files.write(file, lines, Charset.forName("UTF-8"), StandardOpenOption.APPEND);

```

Opret og skriv i binær fil med Files

Filen oprettes, overskrives med nyt indhold eller der tilføjes.

```

byte data[] = {12,23,34,45,56,67};
Path file = Paths.get("C:\\TestFile\\the-file-name.bit");
Files.write(file, data);
Files.write(file, data, StandardOpenOption.APPEND);

```

Og læs igen med Files

```

byte data2[] = Files.readAllBytes(file);
for(byte b : data2)
    System.out.println(b);

```

FileReader og BufferedReader

FileReader bruges til at læse en stream af karakterer. Den kan returnere enkelte karakterer og arrays af karakterer. Constructor kan tage både File-objekt og filnavn som string.

BufferedReader bruges til at læse fra FileReader. Den kan læse linier/strings og den gør, at der ikke bruges for mange ressourcer på at tilgå den fysiske fil mange gange.

Der er en del exceptions. Alle på nær NullPointerException er checked, og den kommer hvis filen ikke findes.

```

public class FileReaderTest
{
    public static void main(String[] args)
    {
        File file = new File("Test.txt");

        FileReader fr = null;
        BufferedReader br = null;

        try {
            fr = new FileReader(file);
            br = new BufferedReader(fr);

            String line;

            while((line = br.readLine()) != null)
            {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found:" + file.getAbsolutePath());
        } catch (IOException e) {
            System.out.println("Unable to read file!");
        }
        finally{
            try {
                br.close();
            } catch (IOException e) {
                System.out.println("Unable to close file!");
            }catch(NullPointerException e){
                System.out.println("NullPointerException - The file was never opened? ");
            }
        }
    }
}

```

```
        }
    }
}
```

Output:

```
First line
Second line
Third line
Fourth line
```

Try with ressources

Antallet af exceptions der skal håndteres i forrige eksempel er lidt overvældende. Den del der har med `br.close()` at gøre, kan der gøres noget ved.

Fra Java 1.7 er der kommet begrebet 'try with ressources'- `try(...)`, som svarer til `using{...}` i C#. Det betyder at en ressource der oprettes i parenteser efter keyword `try`, og som implementerer interfacet `AutoClosable` med metoden `close()`, vil blive lukket og nedlagt automatisk, så man ikke skal tænke på ting der kan gå galt.

I det følgende er forrige kode ændret til dette. (Det er desuden undladt at lægge `FileReader`-objekt i en variabel.)
Så undgås alle `try/catch` omkring lukningen af `BufferedReader`, som sker automatisk.

```
public class FileReaderTest
{
    public static void main(String[] args)
    {
        File file = new File("Test.txt");

        try (BufferedReader br = new BufferedReader(new FileReader(file)))
        {
            String line;
            while((line = br.readLine()) != null)
            {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found:" + file.getAbsolutePath());
        } catch (IOException e) {
            System.out.println("Unable to read file!");
        }
    }
}
```

Og den tilsvarende writer. Der kan indsættes linjeskift i filen både i teksten og ved kald af metoden `newline()`.

```
public class FileWriterTest
{
    public static void main(String[] args)
    {
        try (BufferedWriter br = new BufferedWriter(new FileWriter("Test.txt")))
        {
            br.write("This is line one\n");
            br.write("This is line two");
            br.newLine();
            br.write("Last line.");
        } catch (IOException e) {
            System.out.println("Unable to write file " + e.getMessage());
        }
    }
}
```

```
}
```

Filen ligger i projektets rod-mappe. Der skal refreshes i Eclipse Package Explorer for at se den.

```
This is line one  
This is line two  
Last line.
```

Hvis der angives en absolut sti, som ikke findes kommer exception

```
Unable to write file c:\abc\Test.txt (Den angivne sti blev ikke fundet)
```

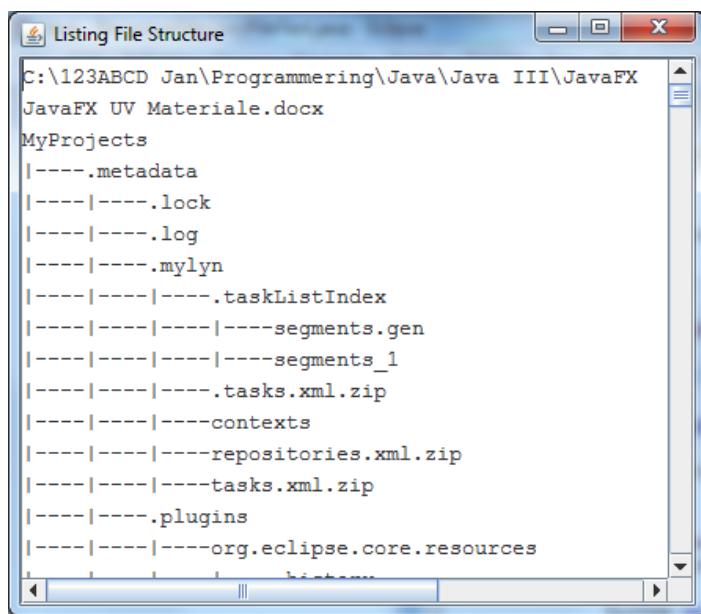
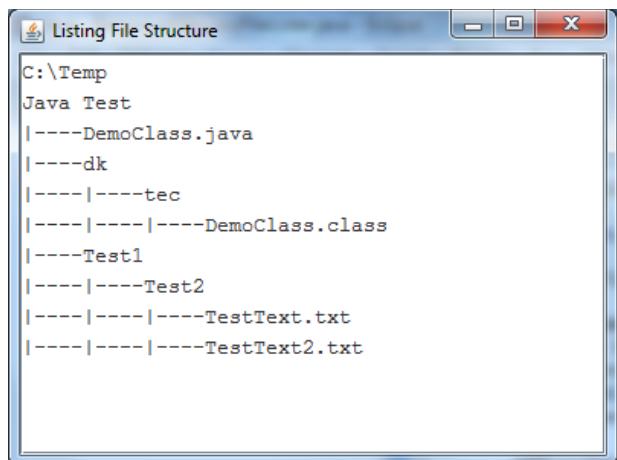
Listing af filstruktur med klassen File

Følgende program viser en listning af filstrukturen fra den angivne mappe.

Main() opretter et objekt af klassen FileLister som arver JFrame og viser denne på skærmen.

I constructoren sendes stien som et File-objekt på den mappe der er roden i den viste struktur.

Constructoren i FileLister kalder en rekursiv metode recurse(), som gennemløber hele strukturen og opbygger teksten i vinduet.



```
public class FileListerTest
{
    public static void main(String[] args)
    {
        String path = "C:\\\\Temp";

        if(args.length >0)
        {
            path = args[0];
        }

        File f = new File(path);
        if (!f.isDirectory())
        {
            System.out.println(path + " doesn't exist or not dir");
        }
    }
}
```

```

        System.exit(0);
    }

    FileLister lister = new FileLister(f);
    lister.setTitle("Listing File Structure");
    lister.setVisible(true);
    lister.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}

```

```

public class FileLister extends JFrame
{
    JTextArea ta;

    FileLister(File f)
    {
        setSize(400, 300);
        ta = new JTextArea();
        ta.setFont(new Font("Monospaced", Font.PLAIN, 14));
        JScrollPane scrPane = new JScrollPane(ta);

        add(scrPane, BorderLayout.CENTER);
        recurse(f);    }

    void recurse(File dirfile, int depth = 0)
    {
        String contents[] = dirfile.list();           // Alle childs
        for(int i=0; i<contents.length; i++)
        {                                              // For alle childs
            for (int spaces=0; spaces<depth; spaces++)
            {
                ta.append("|---");
            }
            ta.append(contents[i] + "\n");             // Udskriv child
            File child = new File(dirfile, contents[i]);
            if(child.isDirectory())                   // Hvis child er dir
            {
                recurse(child, depth+1);            // så gå dybere
            }
        }
    }
}

```

Opgave

Opret en fil i f.eks. notepad, hvor der står et regneudtryk som f.eks. 77+31. Det kan stå på en linie eller på flere linier.

Skriv et program i Java som læser filen, tyder udtrykket, beregner resultatet og skriver resultatet i filen efter regneudtrykket, så der kommer til at stå

77+31=108

Eller

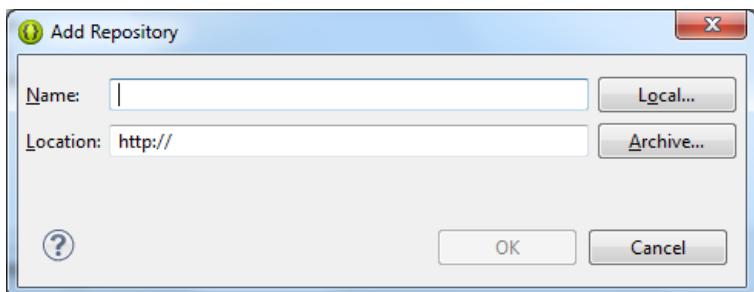
77
+
31

= 108

3: Eleven kan skabe og anvende Java teknologi GUI komponenter: paneler, knapper, labels, tekstfelter og tekstområder.

WindowBuilder

Hvis der i Eclipse ønskes en design-mulighed for Java (Swing) brugerflader, kan WindowBuilder Pro installeres ved i menuen Help at vælge Install New Software. Tryk Add



Og indtast følgende

Name: WindowBuilder Pro Eclipse Update Site

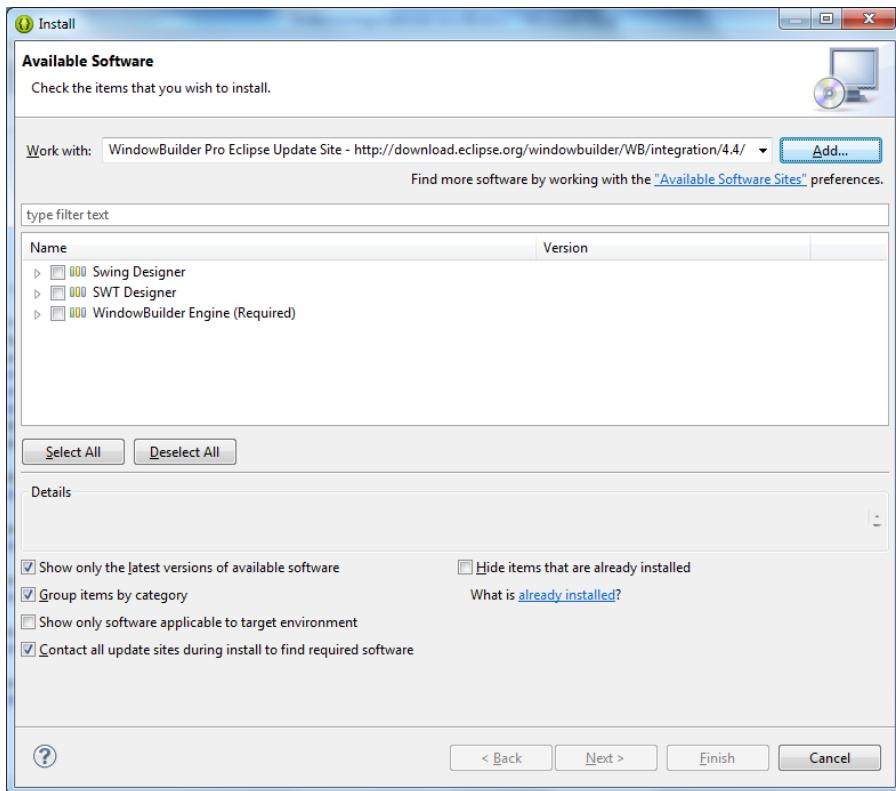
Location: <http://download.eclipse.org/windowbuilder/WB/release/R201506241200-1/4.5/>

Hvis den er forældet, så gå ind på <http://www.eclipse.org/windowbuilder/download.php>

The screenshot shows the Eclipse website's download page. The URL in the browser is <http://www.eclipse.org/windowbuilder/download.php>. The page title is "Installing WindowBuilder Pro". On the left, there's a sidebar with "MyProject" and links for "Download", "Documentation", and "Support". The main content area has a heading "Installing WindowBuilder Pro" and a note about download terms. Below that is a table titled "Update Sites" showing links for different Eclipse versions. At the bottom, there's a note about installing the update site or zip editions.

Eclipse Version	Release Version	Integration Version
4.5 (Mars)	link	link (MD5 Hash)
4.4 (Luna)	link	link (MD5 Hash)
4.3 (Kepler)	link	link (MD5 Hash)
4.2 (Juno)	link	link (MD5 Hash)
3.8 (Juno)	link	link (MD5 Hash)

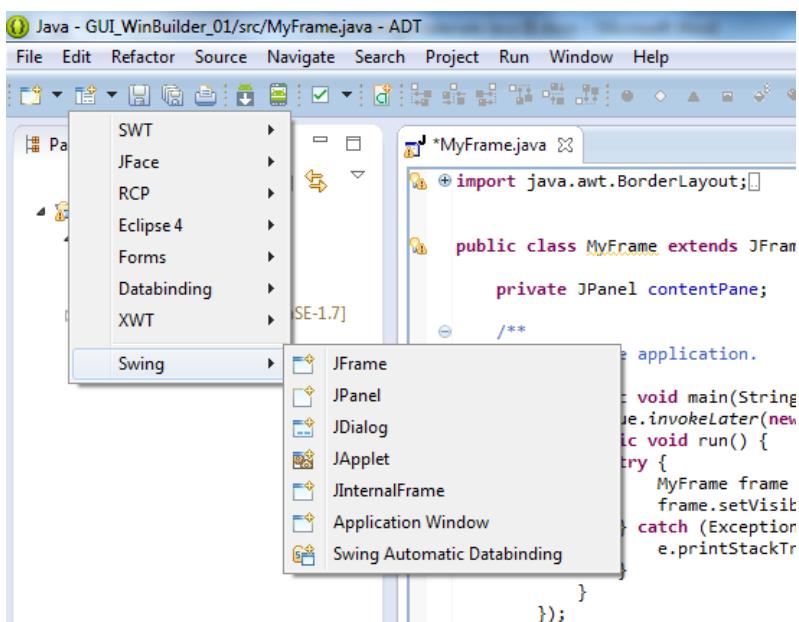
Klik på "link" og kopier URL'en. Læg den ind i Work with / location.



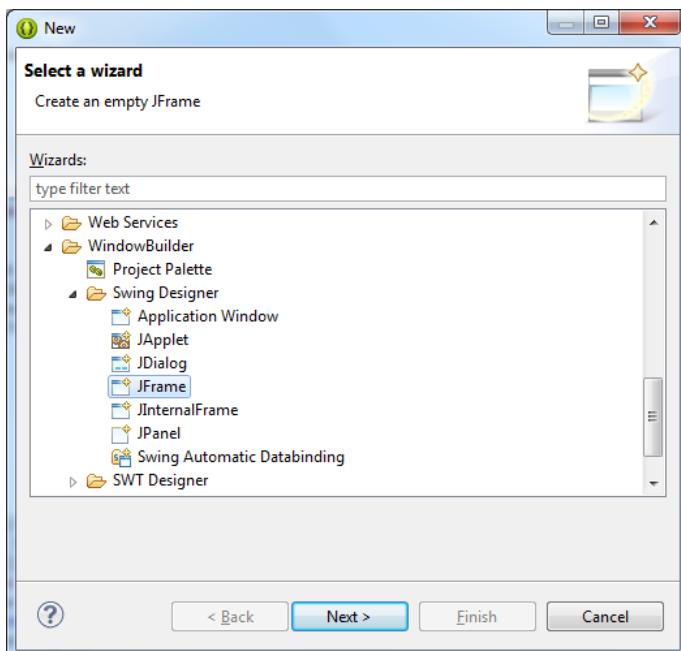
Vælg det hele og tryk Next/Finish.



Dropdown nummer 2 fra venstre er nu Create new visual classes



Eller højre-klik på package og vælg New – Other



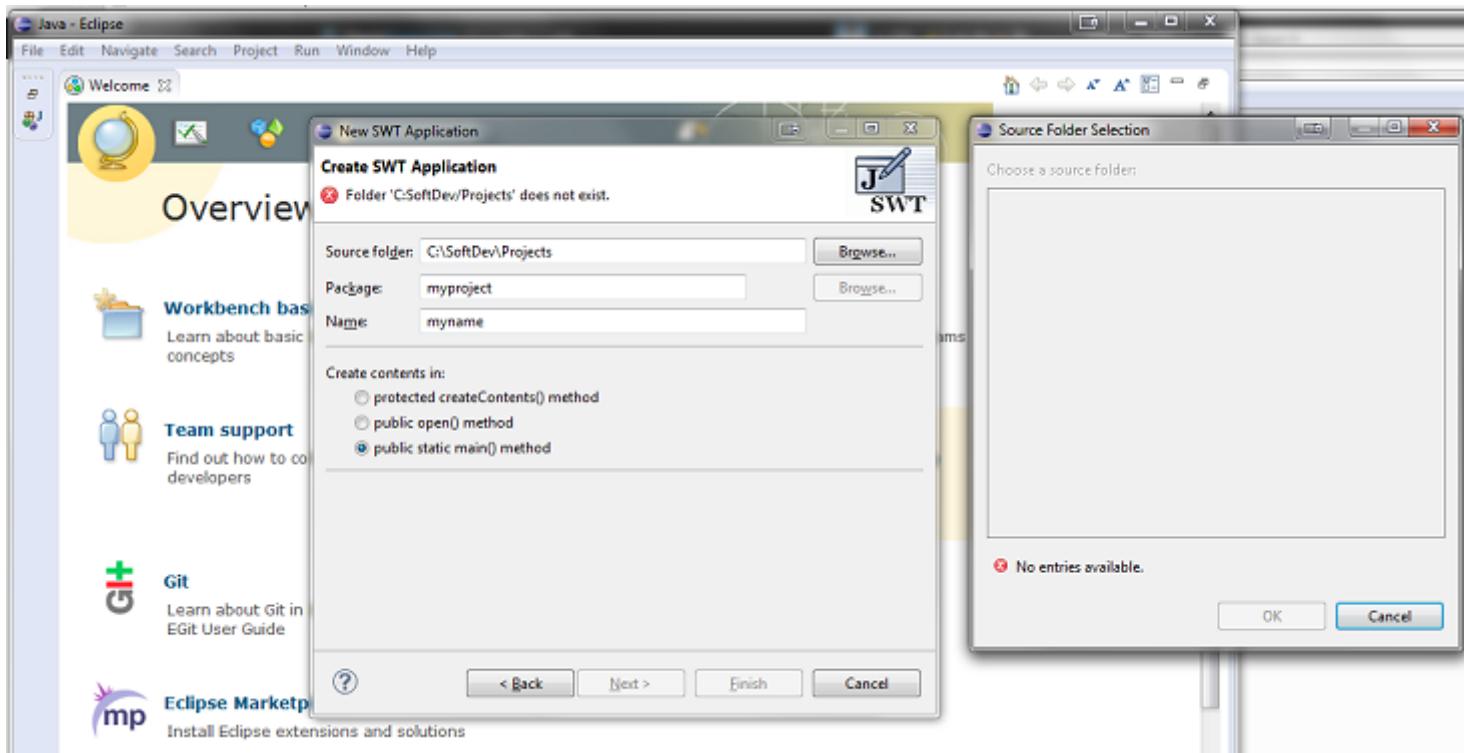
Problem ☹

Man skal ikke oprette et nyt projekt som en WindowsBuilder, men oprette et almindeligt Java-projekt, og så tilføje en WindowsBuilder, som er en klasse.

Herunder et forslag til det fra nettet, men her er det SWT. Det fungerer ikke hos mig.

SWT installed.

I then went to "Menu > File > New > Other... > WindowBuilder > SWT Designer > SWT > Application Window" and clicked on next. That yields the following image.



Follow these steps.

Go to "Menu > File > New > Project...". Create a new project. Follow the prompts.

Go to "Menu > File > New > Other..." and select "WindowBuilder > SWT Designer > SWT > Application Window", click on the next button.

The Source folder text box now comes pre-populated to the project name along with the src subfolder. The package name is also there. Enter the name and click on finish.

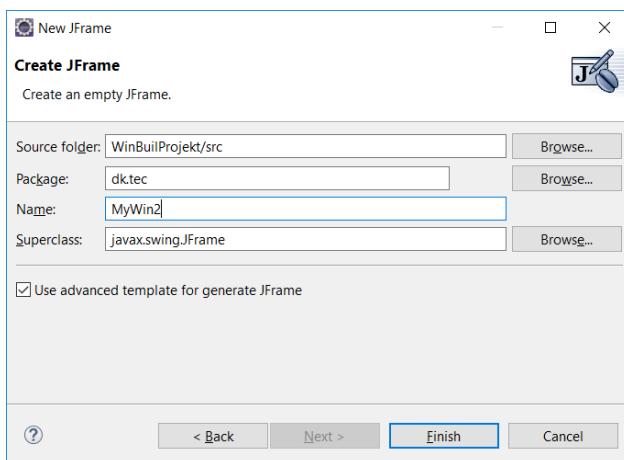
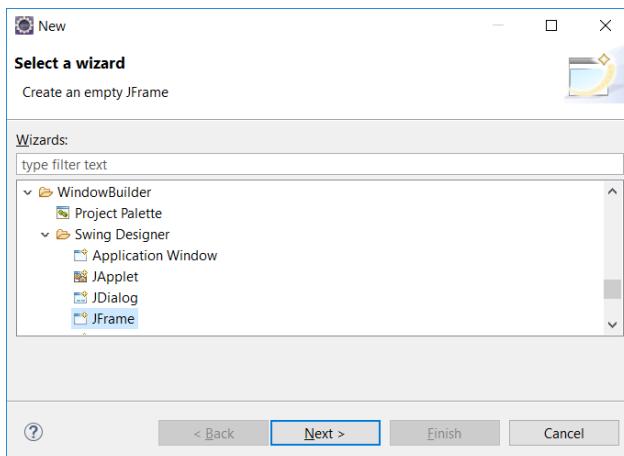
You cannot create a new SWT application window project directly. That was the problem. You must create a "Menu > File > New > Project..." project first. Sadly, the names project are identical and different meanings.

Problem Slut ☺

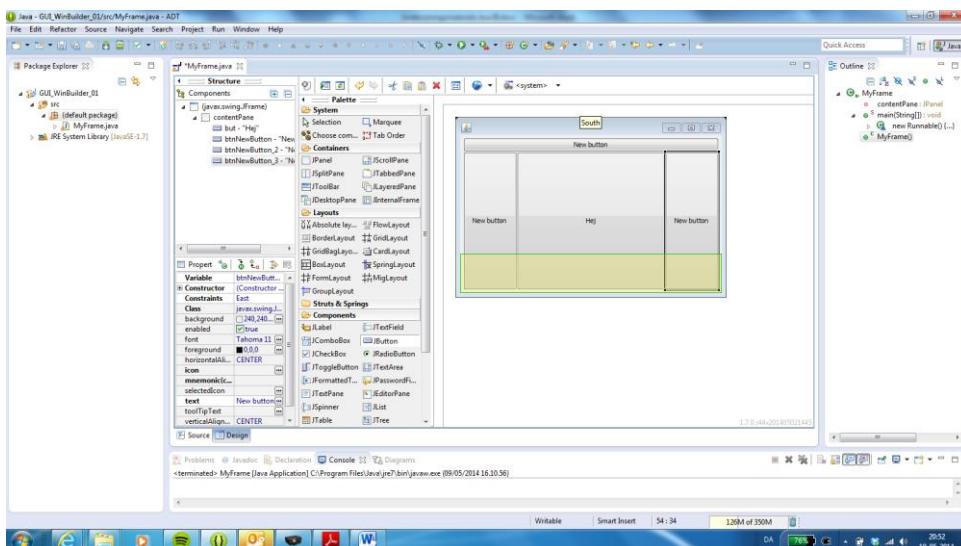
Det var så SWT. Det giver fejl hos mig, hvor den ikke kan finde de packages, der importes.

Vi ønsker at oprette et Swing-projekt med en JFrame i.

Så der oprettes først et nyt Java Project og dernæst oprettes en ny WindowBuilder-klasse med en JFrame i.



Der kommer så en klasse der arver fra JFrame. Der kommer en main() med, men den kan man slette, hvis man har en i forvejen.



Så kan man skifte imellem Source og Design.

Der trækkes nogle buttons ind en BorderLayout, og koden kommer til at se sådan ud.

```
public class MyFrame extends JFrame
```

```

private JPanel contentPane;

public static void main(String[] args)
{
    EventQueue.invokeLater(new Runnable()
    {
        public void run()
        {
            try {
                MyFrame frame = new MyFrame();
                frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

public MyFrame()
{
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 450, 300);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    contentPane.setLayout(new BorderLayout(0, 0));
    setContentPane(contentPane);

    JButton but = new JButton("Hej");
    but.setSize(40, 50);
    contentPane.add(but);

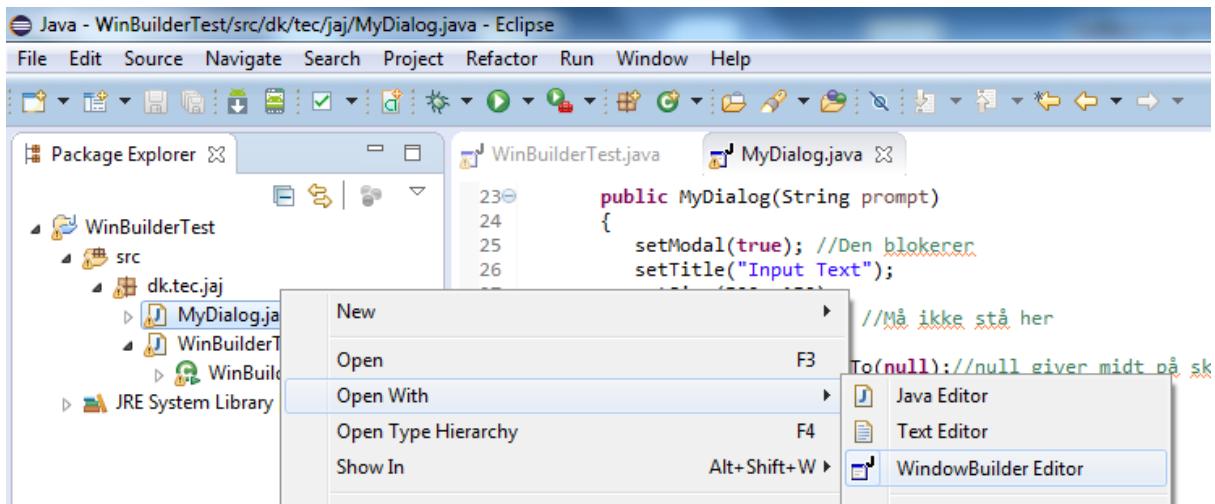
    JButton btnNewButton = new JButton("New button");
    contentPane.add(btnNewButton, BorderLayout.NORTH);

    JButton btnNewButton_2 = new JButton("New button");
    contentPane.add(btnNewButton_2, BorderLayout.WEST);

    JButton btnNewButton_3 = new JButton("New button");
    contentPane.add(btnNewButton_3, BorderLayout.EAST);
}
}

```

En eksisterende klasse, der ikke er oprettet via WindowsBuilder, kan åbnes i editoren ved at højreklikke på filen i Package Explorer og vælge Open With – WindowBuilder Editor.



A Visual Guide to Layout Managers

<http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

A Visual Guide to Swing Components (Windows Look and Feel)

<http://da2i.univ-lille1.fr/doc/tutorial-java/ui/features/components.html>

Using Swing Components

<http://docs.oracle.com/javase/tutorial/uiswing/components/index.html>

Opgave

Beskriv de forskellige typer af layout.

- BorderLayout – Oliver – kort og godt.
- BoxLayout - Mathias
- CardLayout - Martin
- JtabbedPane – Patrick -fint
- FlowLayout –Thomas – fint og kort.
- GridBagLayout – Markus, lidt svær, ok
- GridLayout - Kasper
- GroupLayout – Mark, svær at bruge
- SpringLayout – Stefan – relativ layout - fint
- AbsoluteLayout/NullLayout – Absolut Daniel – fint og kort
- FormLayout - Tor

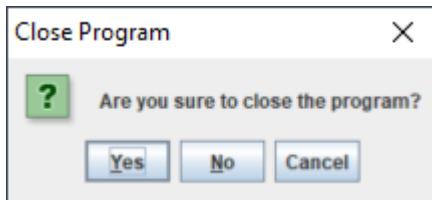
Dialogs

Indbygget Dialog

En JOptionPane.showInputDialog() er en statisk metode. Derfor kan der kun vises en ad gangen, så to tråde kan ikke vise hver deres dialog. Dette kan der rådes bod på ved at lave en custom dialog. Se senere.

```
int answer = JOptionPane.showConfirmDialog(null,
                                             "Are you sure to close the program?",
                                             "Close Program",
                                             JOptionPane.YES_NO_CANCEL_OPTION);

if(answer == JOptionPane.YES_OPTION)
{
    System.out.println("Answer Yes");
    System.exit(0);
}
if(answer == JOptionPane.NO_OPTION)
{
    System.out.println("Answer No");
}
if(answer == JOptionPane.CANCEL_OPTION)
{
    System.out.println("Answer Cancel");
}
```



Custom Dialog

En custom dialog konstrueres ved at arve fra JDialog og så fylde den med de controller, der ønskes på den.

Hvis den er modal, blokerer den tråden når den sættes visible, og kører først videre når den sættes invisible. Derfor kan der læses data ud af objektet her i linier umiddelbart efter den sættes visible. Den sættes invisible internt.



```
public class Main
{
    public static void main(String[] args)
    {
        InputTextDialog inputDialog = new InputTextDialog("Indtast venligst Deres navn");
        inputDialog.setVisible(true);
        System.out.println("Input navn: " + inputDialog.getInputText());
        System.out.println("Main ended!");
    }
}
```

```

public class InputTextDialog extends JDialog
{
    private JLabel lblPrompt;
    private JTextField txtInput;
    private JButton btnOK;

    private String text;

    public InputTextDialog(String prompt)
    {
        setModal(true); //Den blokerer
        setTitle("Input Text");
        setSize(500, 150);
        //setVisible(true); //Må ikke stå her

        setLocationRelativeTo(null); //null giver midt på skærmen

        lblPrompt = new JLabel(prompt);
        txtInput = new JTextField(40);
        btnOK = new JButton("OK");

        btnOK.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent arg0)
            {
                InputTextDialog.this.text = txtInput.getText();
                InputTextDialog.this.setVisible(false);
            }
        });
    }

    setLayout(new FlowLayout());
    // Default BorderLauout

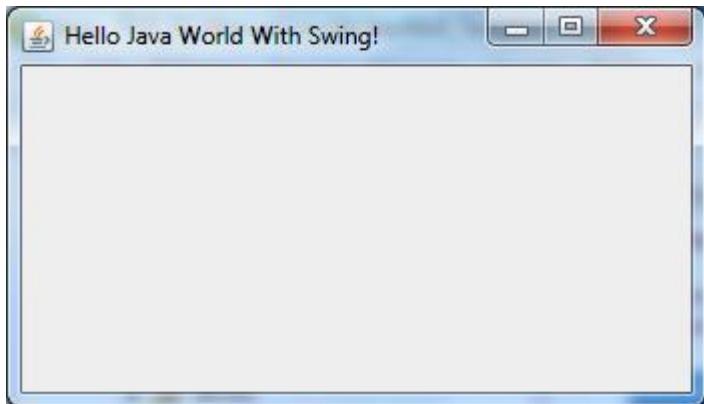
    add(lblPrompt);
    add(txtInput);
    add(btnOK);
}

public String getInputText()
{
    return this.text;
}
}

```

Frames

Jframe



```
public class GUI_Test
{
    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            @Override
            public void run()
            {
                JFrame frame = new JFrame("Hello Java World With Swing!");
                frame.setSize(350, 200);

                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}
```

En JFrame med JTextArea og JButton med eventhandler



```
public class GUI_Test
{
    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            @Override
```

```
public void run()
{
    MyFrame frame = new MyFrame("Hello Java World With Swing!");
    frame.setSize(350, 200);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
```

```
public class MyFrame extends JFrame
{
    public MyFrame(String title)
    {
        super(title);

        setLayout(new BorderLayout());

        final JTextArea textArea = new JTextArea();
        JButton button = new JButton("Click Me");

        Container c = getContentPane();

        c.add(textArea, BorderLayout.CENTER);
        c.add(button, BorderLayout.SOUTH);

        button.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                textArea.append("Hello Java World With Swing!\n");
            }
        });
    }
}
```

Avanceret eventhandling

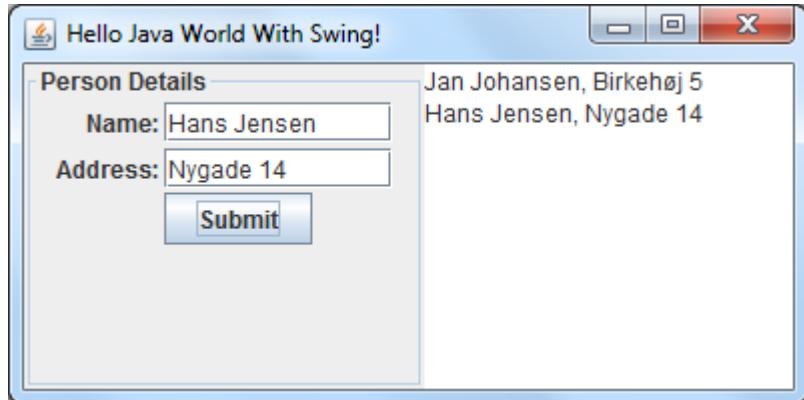
Kan også kaldes Custom Events. Tanken er at det ikke er en god ide at lægge logik ind i en knaps eventhandler. Det kan i store programmer ende med at være svært at vedligeholde. I stedet bygges det op sådan at et Panel kan udløse events og det der bliver udført er kald af en metode, der sættes udefra. Så når knappen trykkes bliver der udført en indirekte metode. Panelet skal bare aflevere informationer fra brugerfladen og ikke vide hvad der gøres med dem.

Programmet består af en JFrame/MyFrame med BorderLayout, hvor der er placeret en JTextArea i højre side og en JPanel/DetailsPanel i venstre side.

DetailsPanel har GridBagLayout. Der sættes en eventhandler på DetailsPanel ude fra MyFrame, sådan at når der trykkes på knappen Submit, kaldes denne eventhandler. Dette er for at DetailsPanel ikke skal vide noget om hvad data skal bruges til. Den skal bare aflevere data, ved at kalde den eventhandler der sættes udefra.

Det er så MyFrame der definerer hvad der skal ske med data, der smides ud fra DetailsPanel, når der trykkes på Submit-knappen i den, og skriver det i JTextArea i højre side.

For at kunne etablere denne custom-eventhandler oprettes klassen DetailEvent til at holde teksten og interfacet DetailListener med handler-metoden detailEventOccurred(), og der benyttes en EventListenerList til at holde de eventhandlere, der skal kaldes. Der kan altså tilføjes flere handlere, som udføres når knappen trykkes.



Klassen EventListenerList

I DetailsPanel er fielden listenerList af typen EventListenerList, som er en eksisterende klasse. Den indeholder de eventhandlere, der skal kaldes, når et event opstår. Lidt ligesom et event/delegate i C#, der kan inddholde flere handler-metoder.

Klassen EventListenerList indeholder følgende:

Field

- protected Object[] listenerList

Constructor

- EventListenerList()

Metoder

- public Object[] getListenerList()
- public <T extends EventListener> T[] getListeners(Class<T> t)
- public int getListenerCount()
- public int getListenerCount(Class<?> t)
- public <T extends EventListener> void add(Class<T> t, T l)

- public <T extends EventListener> void remove(Class<T> t, T l)
- public String toString()

Metoden main()

```
public class GUI_Test
{
    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            @Override
            public void run()
            {
                MyFrame frame = new MyFrame("Hello Java World With Swing!");
                frame.setSize(400, 200);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}
```

Klassen MyFrame

Her addes en listener til detailsPanel ved at implementere DetailListeners handler-metode detailEventOccured() som modtager oplysningerne i et DetailEvent-objekt, som parameter, henter teksten ud af denne, og skriver det i textArea.

```
public class MyFrame extends JFrame
{
    private DetailsPanel detailsPanel;

    public MyFrame(String title)
    {
        super(title);

        setLayout(new BorderLayout());

        final JTextArea textArea = new JTextArea();
        detailsPanel = new DetailsPanel();
        detailsPanel.addDetailListener(new DetailListener()
        {
            @Override
            public void detailEventOccurred(DetailEvent event)
            {
                String text = event.getText();
                textArea.append(text);
            }
        });
        Container c = getContentPane();

        c.add(textArea, BorderLayout.CENTER);
        c.add(detailsPanel, BorderLayout.WEST);
    }
}
```

Klassen DetailsPanel

Denne har metoderne addDetailListener(), som benyttedes ovenfor, og removeDetailListener().

De vedligeholder `listenerList` som holder de handlere der skal kaldes når eventet kommer.

Har desuden `fireDetailEvent()`, som kaldes når der trykkes på knappen. Knappens eventhandler sættes altid til at kalde denne, og der kaldes så `handler`-metoden i hver af de eventhåndlere, der er i `listenerList`.

```
public class DetailsPanel extends JPanel
{
    private EventListenerList listenerList = new EventListenerList();

    public DetailsPanel()
    {
        Dimension size = getPreferredSize();
        size.width = 200;
        setPreferredSize(size);

        setBorder(BorderFactory.createTitledBorder("Person Details"));

        JLabel lblName = new JLabel("Name: ");
        JLabel lblAddress = new JLabel("Address: ");

        final JTextField txtName = new JTextField(10);
        final JTextField txtAddress = new JTextField(10);

        JButton btnSubmit = new JButton("Submit");

        btnSubmit.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                String name = txtName.getText();
                String address = txtAddress.getText();

                String text = name + ", " + address + "\n";
                fireDetailEvent(new DetailEvent(this, text));
            }
        });
    }

    setLayout(new GridBagLayout());
    GridBagConstraints gc = new GridBagConstraints();

    gc.weightx = 0.5;
    gc.weighty = 0.5; // Lidt plads

    // First column /**
    // Højrestillet
    gc.anchor = GridBagConstraints.LINE_END;
    gc.gridx = 0;
    gc.gridy = 0;
    add(lblName, gc);

    gc.gridx = 0;
    gc.gridy = 1;
    add(lblAddress, gc);

    // Second column /**
    // Venstrestillet
    gc.anchor = GridBagConstraints.LINE_START;
    gc.gridx = 1;
    gc.gridy = 0;
    add(txtName, gc);

    gc.gridx = 1;
    gc.gridy = 1;
    add(txtAddress, gc);

    // Final row /**
    gc.weighty = 10; // Meget plads
    // I toppen af meget plads
```

```

        gc.anchor = GridBagConstraints.FIRST_LINE_START;
        gc.gridx = 1;
        gc.gridy = 2;
        add(btnSubmit, gc);
    }

    protected void fireDetailEvent(DetailEvent detailEvent)
    {
        Object[] listeners = listenerList.getListenerList();

        for(int i = 0; i < listeners.length; i++)
        {
            if(listeners[i] == DetailListener.class)
            {
                ((DetailListener)listeners[i+1]).detailEventOccurred(detailEvent);
            }
        }
    }

    public void addDetailListener(DetailListener listener)
    {
        listenerList.add(DetailListener.class, listener);
    }

    public void removeDetailListener(DetailListener listener)
    {
        listenerList.remove(DetailListener.class, listener);
    }
}

```

Klassen DetailEvent

Arver fra den foruddefinerede EventObject og danner et oplysningsobjekt, der sendes som parameter til eventhandler-metoden.

Den arver event-source og tilføjer en text-field, som kommer til at indeholde det brugeren har indtastet.

Klassen EventObject

Field

- protected transient Object source

Constructor

- public EventObject(Object source)

Metoder

- public Object getSource()
- public String toString()
- terface that all event listener interfaces must extend.

```

public class DetailEvent extends EventObject
{
    private String text;

    public DetailEvent(Object source, String text)
    {
        super(source);
        this.text = text;
    }
}

```

```
    }  
  
    public String getText() {return text;}  
}
```

Interface DetailListener

Det udvider den foruddefinerede EventListener og tilføjer metoden detailEventOccurred(DetailEvent event), som er eventhandlermetoden.

Interface EventListener

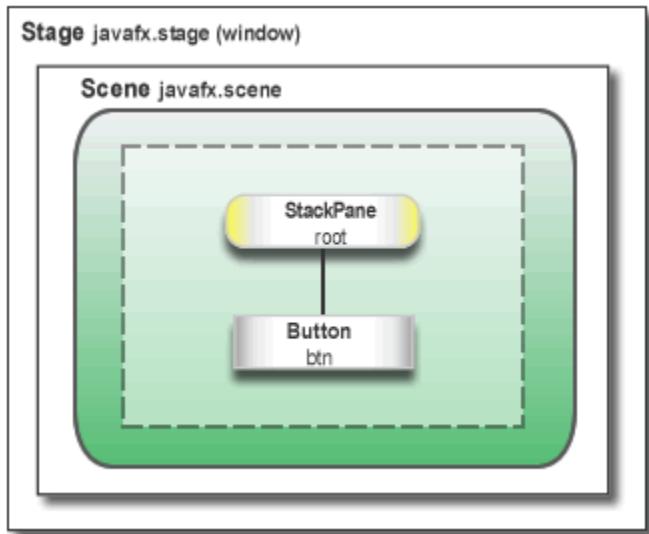
- Tom. A tagging interface that all event listener interfaces must extend.

```
public interface DetailListener extends EventListener  
{  
    public void detailEventOccurred(DetailEvent event);  
}
```

Opgaver

1. En bil der eksploderer når omdrejningerne når over 9000/pr minut.

JavaFX

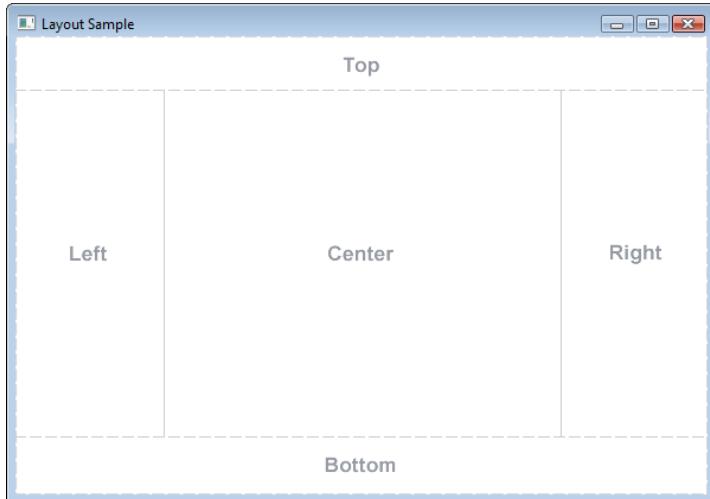


JavaFX Layoutmanagers

https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm#CHDGHCDG

BorderPane

The `BorderPane` layout pane provides five regions in which to place nodes: top, bottom, left, right, and center. The regions can be any size. If your application does not need one of the regions, you do not need to define it and no space is allocated for it.



HBox

The `HBox` layout pane provides an easy way for arranging a series of nodes in a single row.



VBox

The `VBox` layout pane is similar to the `HBox` layout pane, except that the nodes are arranged in a single column.



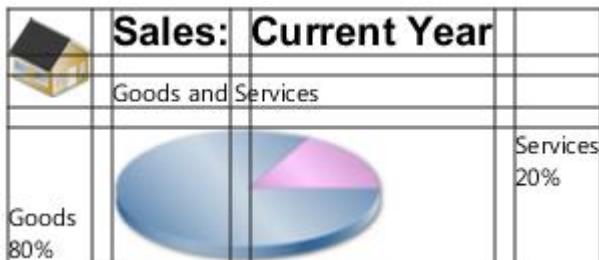
StackPane

The `StackPane` layout pane places all of the nodes within a single stack with each new node added on top of the previous node. This layout model provides an easy way to overlay text on a shape or image or to overlap common shapes to create a complex shape. The figure shows a help icon that is created by stacking a question mark on top of a rectangle with a gradient background.



GridPane

The [GridPane](#) layout pane enables you to create a flexible grid of rows and columns in which to lay out nodes. Nodes can be placed in any cell in the grid and can span cells as needed. A grid pane is useful for creating forms or any layout that is organized in rows and columns. The figure shows a grid pane that contains an icon, title, subtitle, text and a pie chart. In this figure, the `gridLinesVisible` property is set to display grid lines, which show the rows and columns and the gaps between the rows and columns. This property is useful for visually debugging your [GridPane](#) layouts.



FlowPane

The nodes within a [FlowPane](#) layout pane are laid out consecutively and wrap at the boundary set for the pane. Nodes can flow vertically (in columns) or horizontally (in rows). A vertical flow pane wraps at the height boundary for the pane. A horizontal flow pane wraps at the width boundary for the pane. The figure shows a sample horizontal flow pane using numbered icons. By contrast, in a vertical flow pane, column one would contain pages one through four and column two would contain pages five through eight.



TilePane

A tile pane is similar to a flow pane. The [TilePane](#) layout pane places all of the nodes in a grid in which each cell, or tile, is the same size. Nodes can be laid out horizontally (in rows) or vertically (in columns). Horizontal tiling wraps

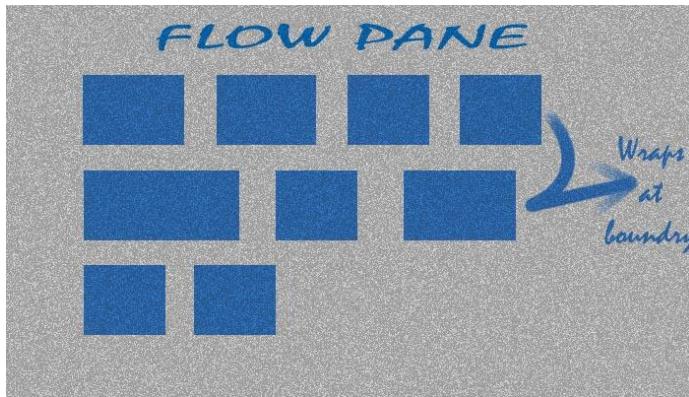
the tiles at the tile pane's width boundary and vertical tiling wraps them at the height boundary. Use the `prefColumns` and `prefRows` properties to establish the preferred size of the tile pane.

Gap properties can be set to manage the spacing between the rows and columns. The padding property can be set to manage the distance between the nodes and the edges of the pane.

Flow Vs Tile Panes

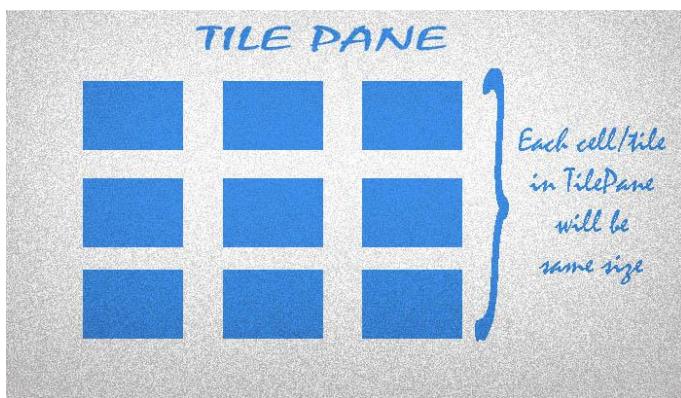
Flow pane and Tile Pane are quite similar to each other. Both layout its children consecutively one after another, either horizontally or vertically. The only major difference is, that the TilePane places all children in tiles that are the same size! So the size of the greatest children is taken for the size of each individual tile in the TilePane.

The nodes within a FlowPane layout are laid out consecutively and wrap at the boundary set for the pane. Nodes can flow vertically (in columns) or horizontally (in rows). A vertical flow pane wraps at the height boundary for the pane. A horizontal flow pane wraps at the width boundary for the pane.



FlowPane's `prefWrapLength` property establishes its preferred width (for horizontal) or preferred height (for vertical). Applications should set `prefWrapLength` if the default value (400) doesn't suffice. Note that `prefWrapLength` is used only for calculating the preferred size and may not reflect the actual wrapping dimension, which tracks the actual size of the flowpane.

TilePane works like a grid



The TilePane layout pane places all of the nodes in a grid in which each cell, or tile, is the same size. Nodes can be laid out horizontally (in rows) or vertically (in columns). Horizontal tiling wraps the tiles at the tile pane's width boundary and vertical tiling wraps them at the height boundary. Use the `prefColumns` and `prefRows` properties to establish the preferred size of the tile pane.

Flow and Tile Pane's Gap properties can be set to manage the spacing between the rows and columns. The padding property can be set to manage the distance between the nodes and the edges of the both Flow and Tile panes.

AnchorPane

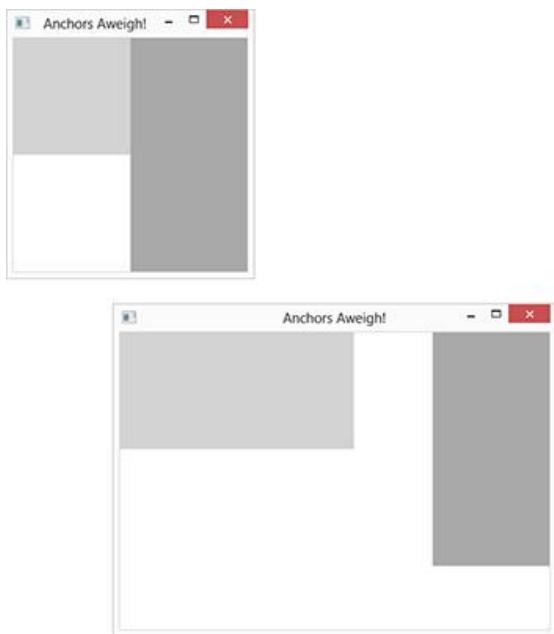
På en AnchorPane kan noder sættes fast på en af siderne eller hjørnerne, og følger så med når vinduet ændrer størrelse. Hvis de er sat fast på to modsatte sider, evt. med en afstand til disse, vil de ændre størrelse med vinduets størrelse.

You use the `setTopAnchor`, `setRightAnchor`, `setBottomAnchor`, and `setLeftAnchor` methods to anchor the nodes to the edges of the anchor pane. Each of these methods accepts two parameters: the node you want to anchor and an offset value that lets you anchor the node a certain distance from the edge.

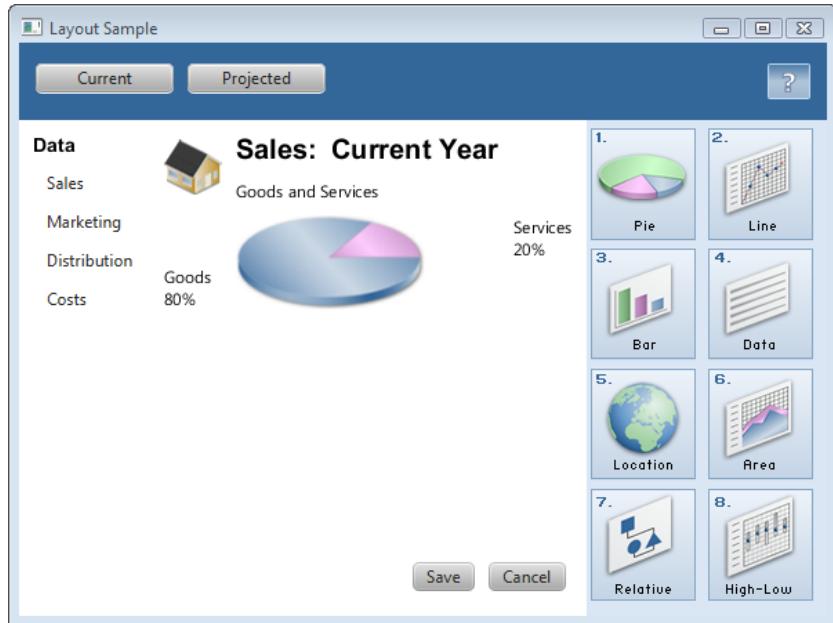
To anchor rectangle `r1` to the top and left edges and rectangle `r2` to the top and right edges, use these lines:

```
anchor.setTopAnchor(r1, 0.0);  
anchor.setLeftAnchor(r1, 0.0);  
anchor.setTopAnchor(r2, 0.0);  
anchor.setRightAnchor(r2, 0.0);
```

The following shows how this anchor pane will appear when displayed in a scene.



Kombination af flere layouts



Først en BorderPane og her på:

- Top: En Hbox med to knapper og en StackPane i højre side.
- Left: En Vbox
- Right: En FlowPane
- Center: En AnchorPane og her på:
 - Øverst en GridPane, der er bundet til alle siderne, så den stretsches.
 - Nederst: To knapper der er bundet til højre side og bund, så de følger med.



JavaFX med IntelliJ

Hent IntelliJ fra

<http://www.jetbrains.com/idea/download/>

IntelliJ er ikke gratis. Når prøvetiden udløber kan den kun bruges 30 minutter ad gangen.

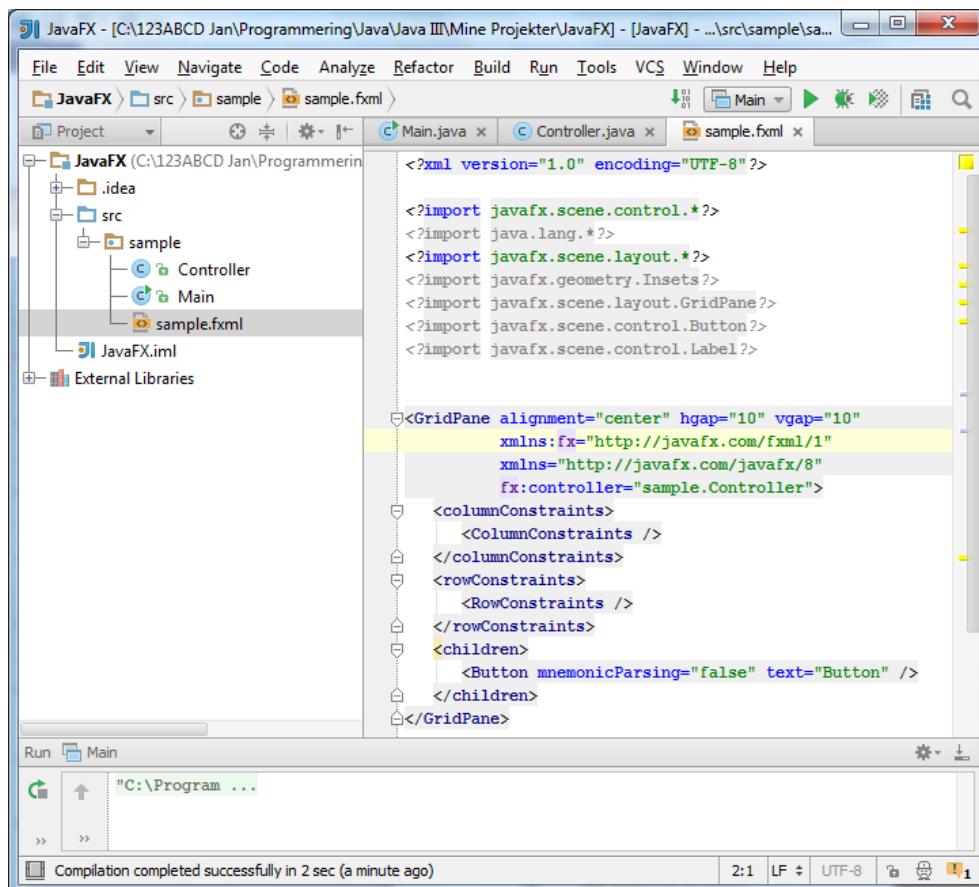
Community Edition er gratis.

Hent JavaFX Scene Builder

<http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html>

Installer begge programmer.

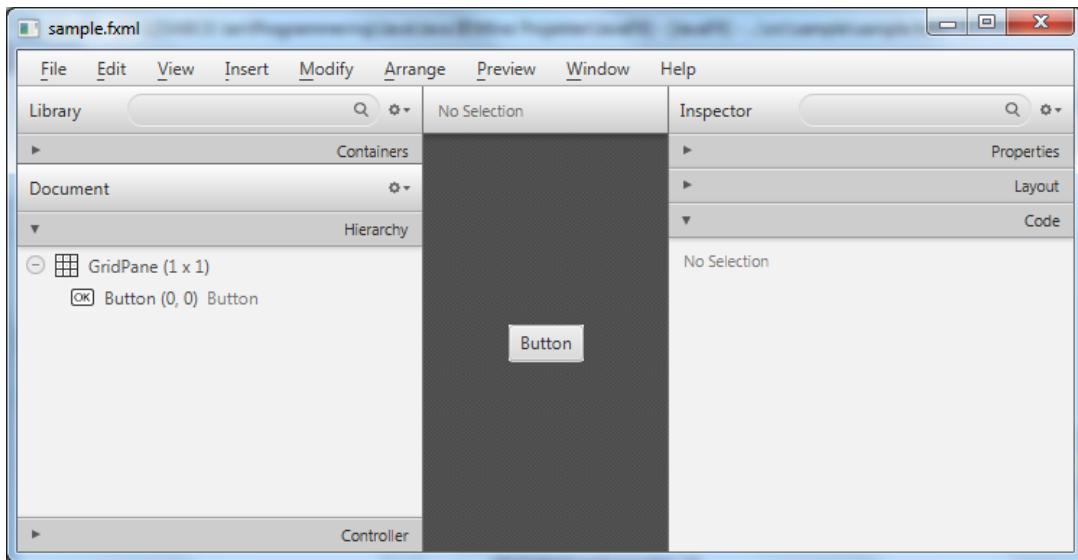
Her under ses et nyt projekt i IntelliJ



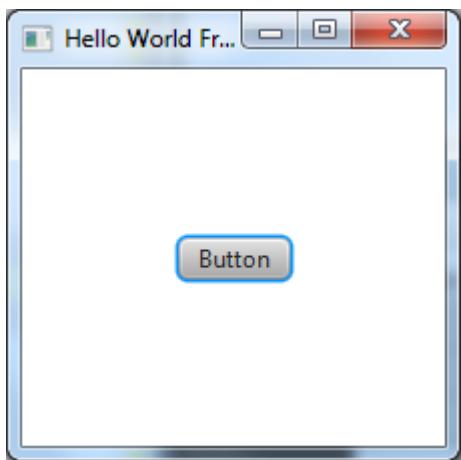
Ved at højreklikke i Projekt-træet på sample.fxml kan der vælges

Open In SceneBuilder

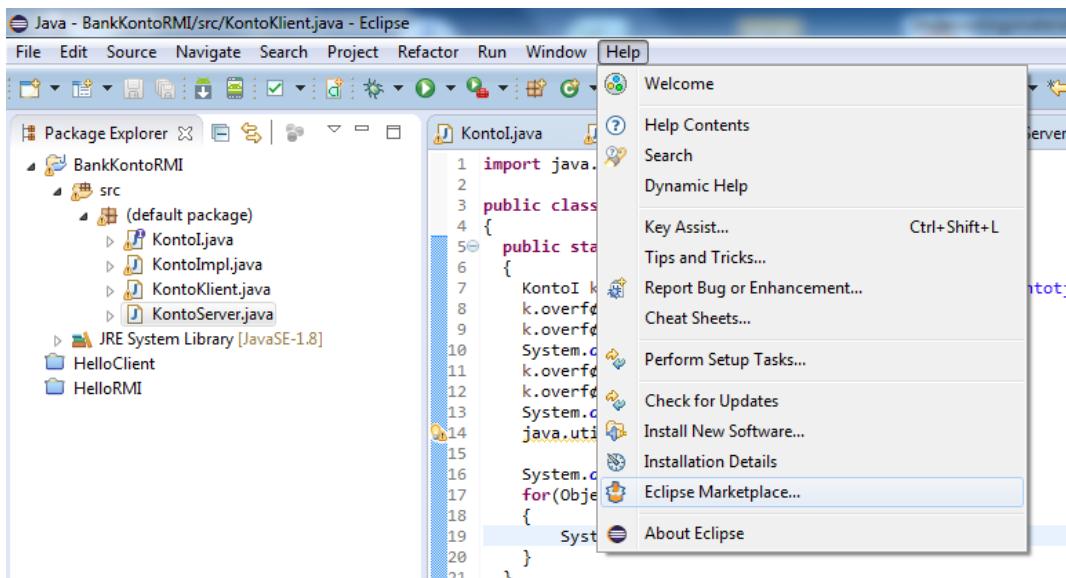
Og så vises følgende design af fxml-koden, hvor der kan ændres og gemmes.



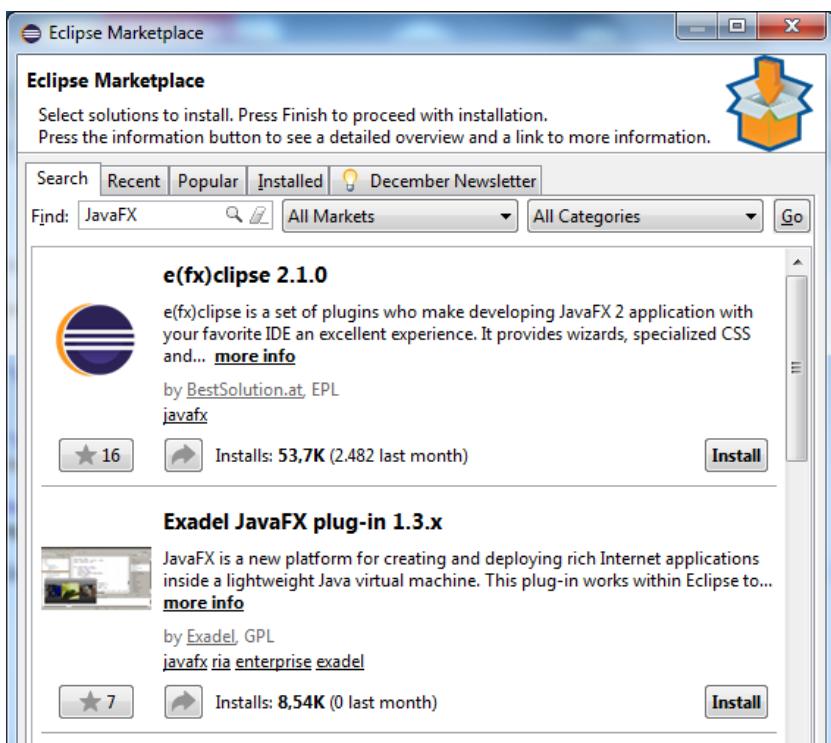
Udfør programmet i IntelliJ



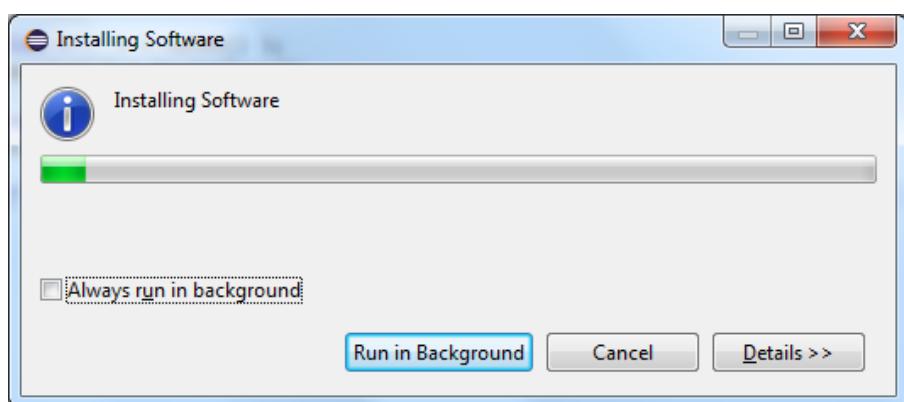
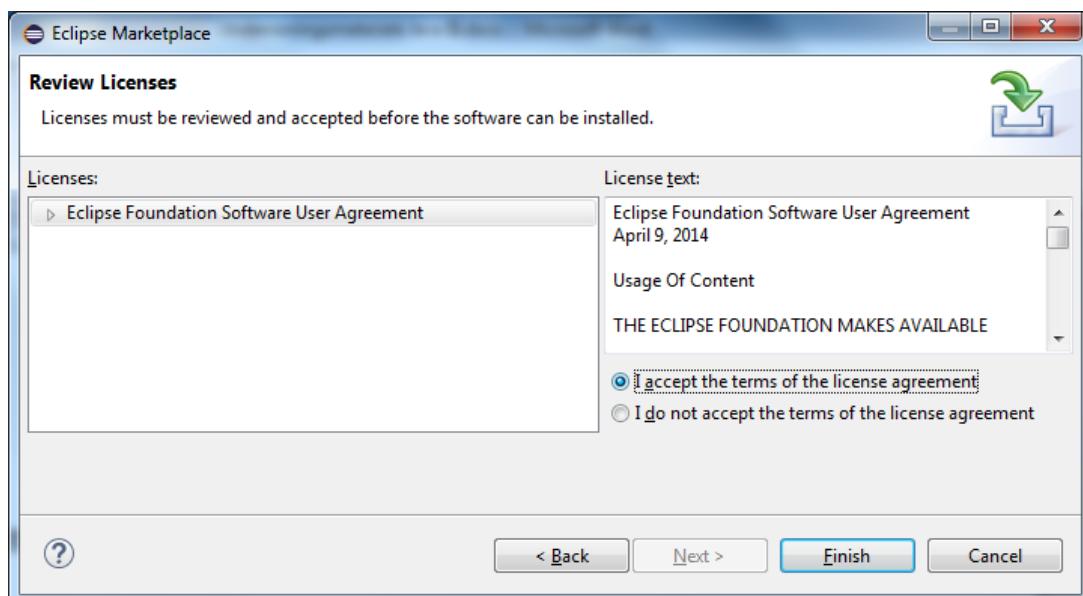
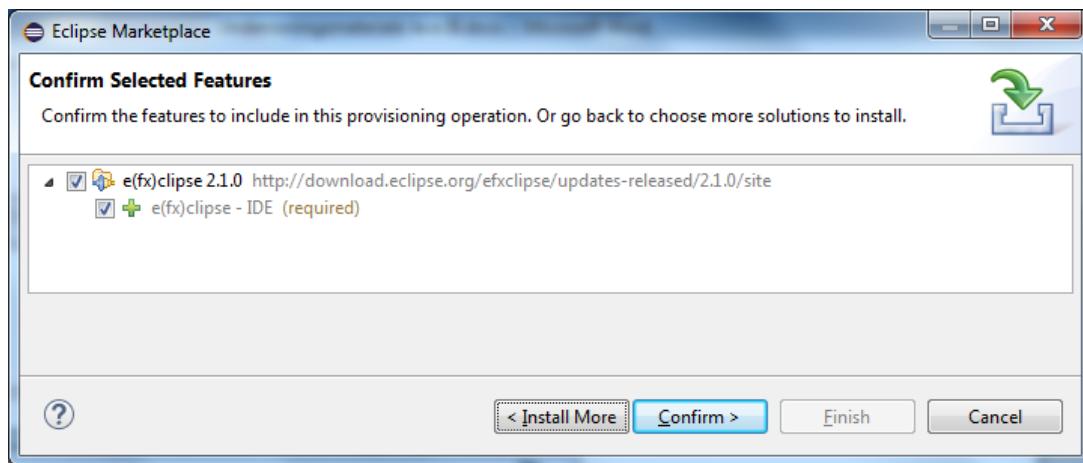
Eclipse med JavaFX



I Find: Skriv JavaFX og tryk enter.



Tryk på Install af e(fx)clipse.



Hent derefter JavaFX Scene Builder

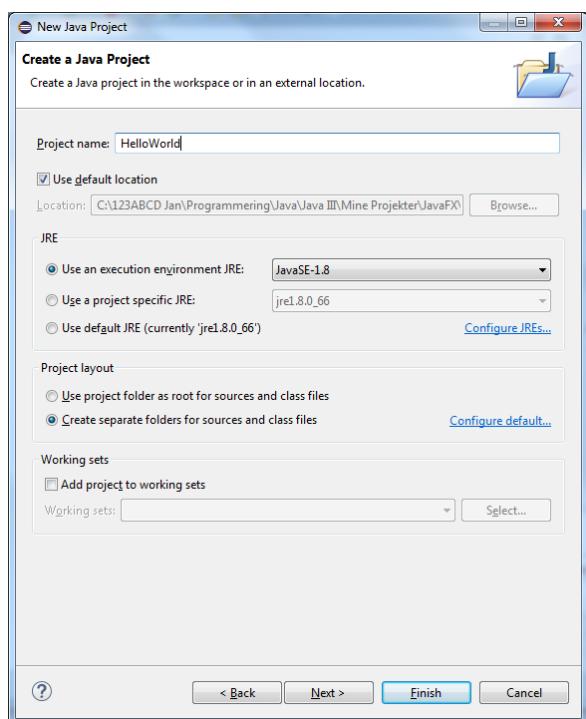
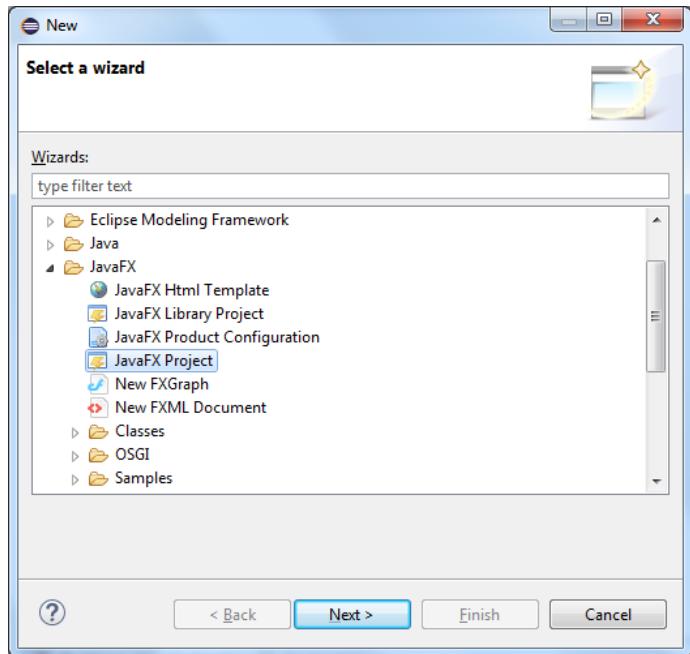
<http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html>

Restart Eclipse og vi er kørende.

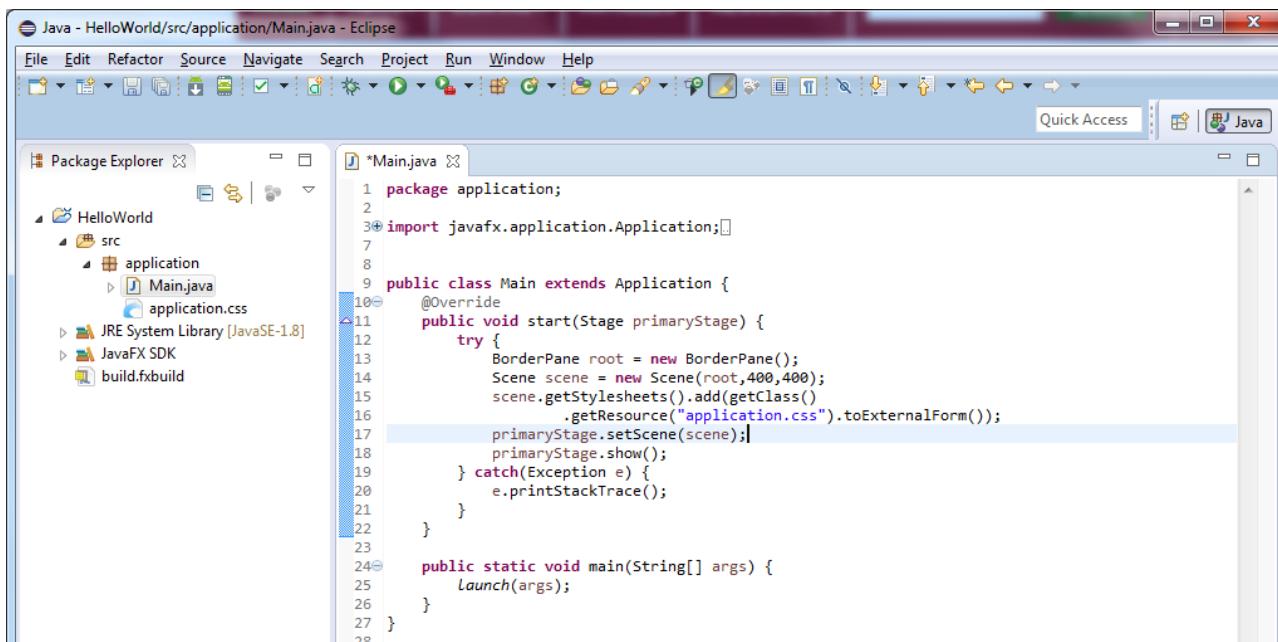
Hello World i JavaFX uden FXML

Brugerfladen oprettes i Java-koden.

Vælg File – New...



Den ser så sådan ud.



Der tilføjes så koden ser ud som nedenfor.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;

public class Main extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        try {
            BorderPane root = new BorderPane();

            Label label = new Label("Hello World");
            label.setFont(Font.font("Arial", FontWeight.BOLD, 24));

            root.setCenter(label);

            Scene scene = new Scene(root,400,400);
            scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

            primaryStage.setScene(scene);
            primaryStage.setTitle("Hello World");
            primaryStage.show();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

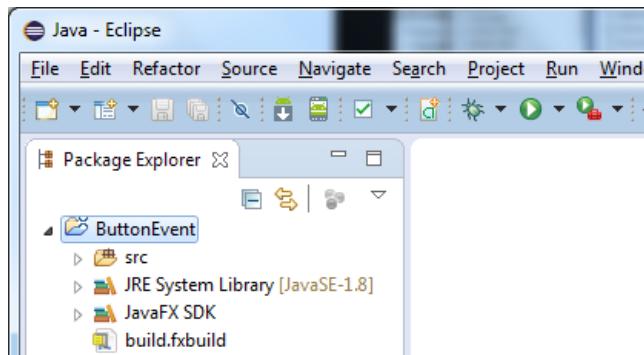
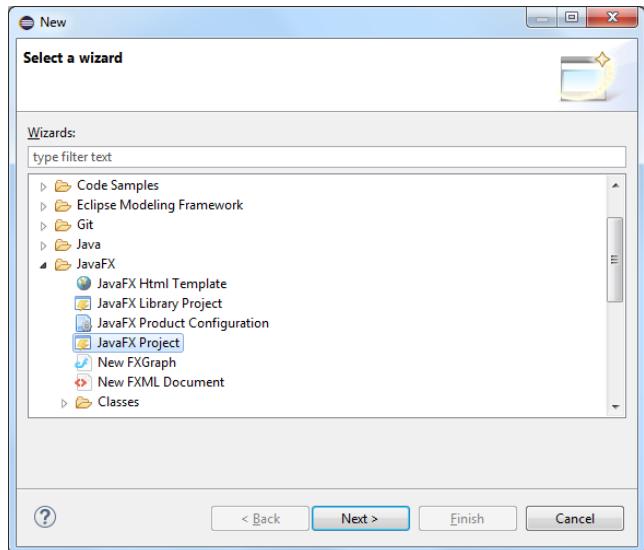
    public static void main(String[] args) {
        launch(args);
    }
}
```



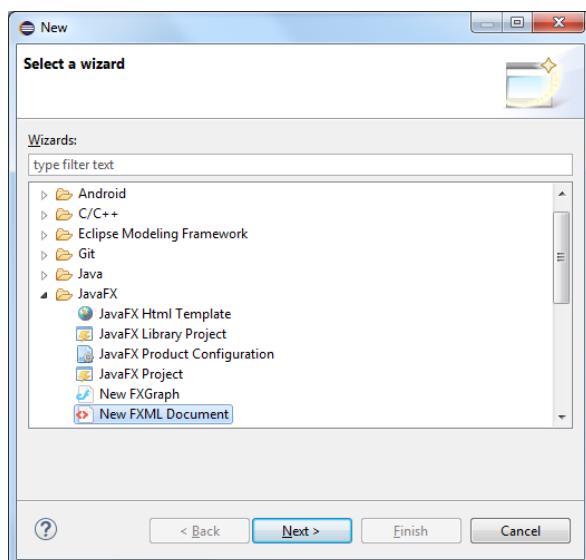
FXML Button med event

Brugerfladen oprettes i FXML.

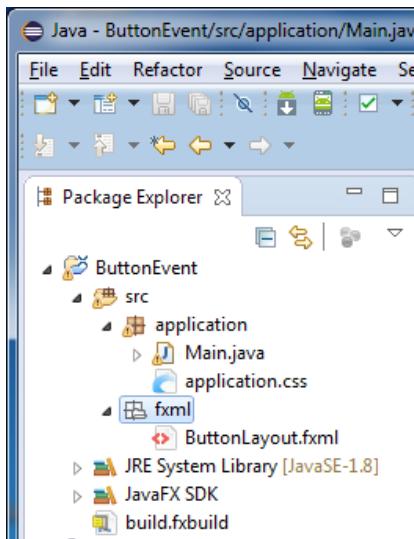
Opret et nyt JavaFX projekt af navnet ButtonEvent.



Tilføj en mappe af navnet fxml under mappen src, og tilføj her til projektet et nyt FXML Document af navnet ButtonLayout.



Så ser det sådan ud.

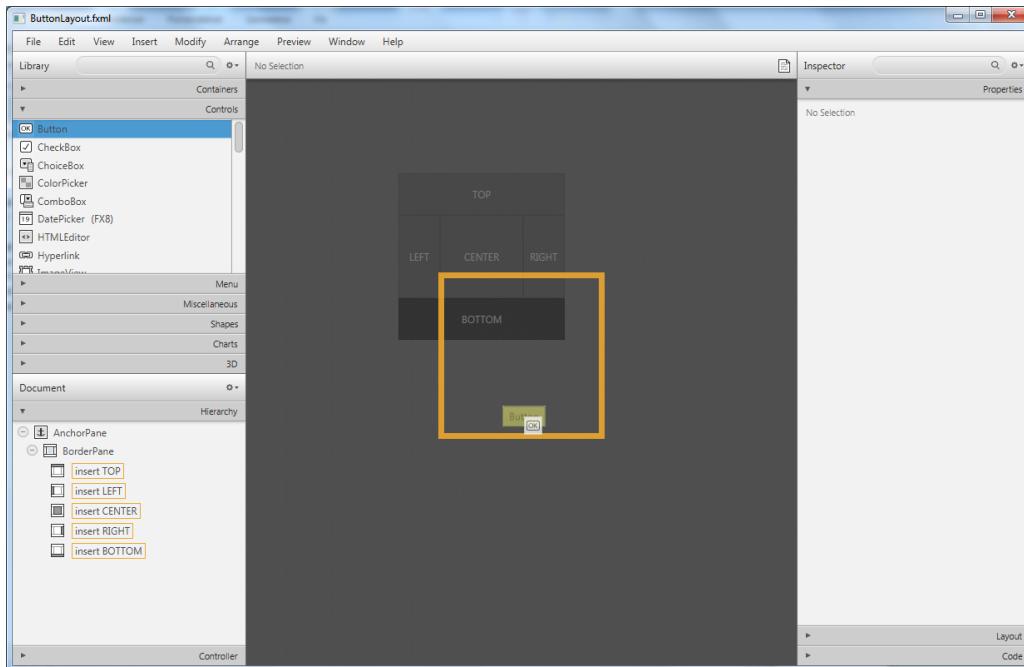


Højreklik på ButtonLayout.fxml og vælg Open with SceneBuilder.

Fjern først den automatisk indsatte AnchorPane.

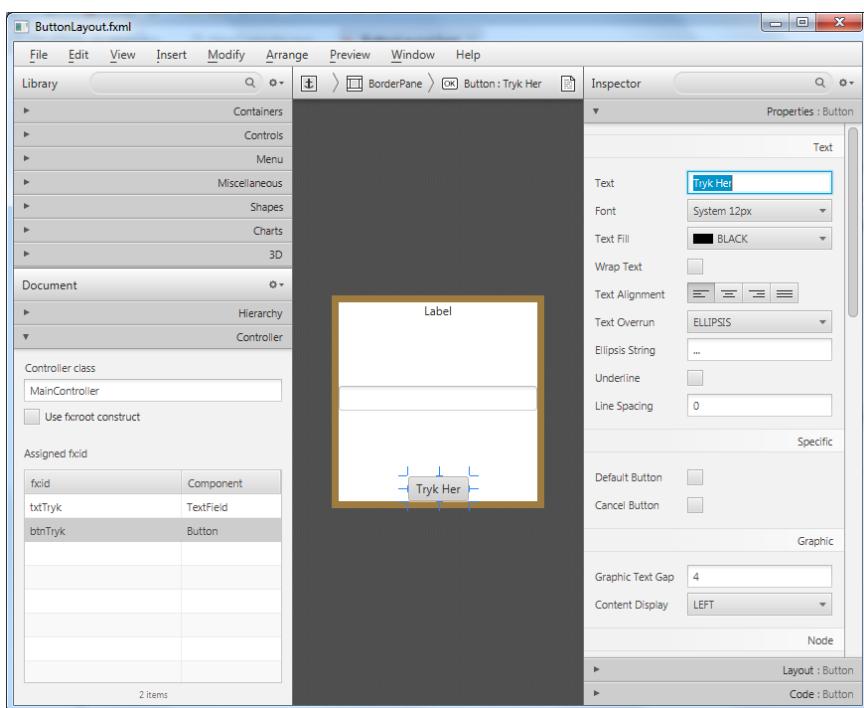
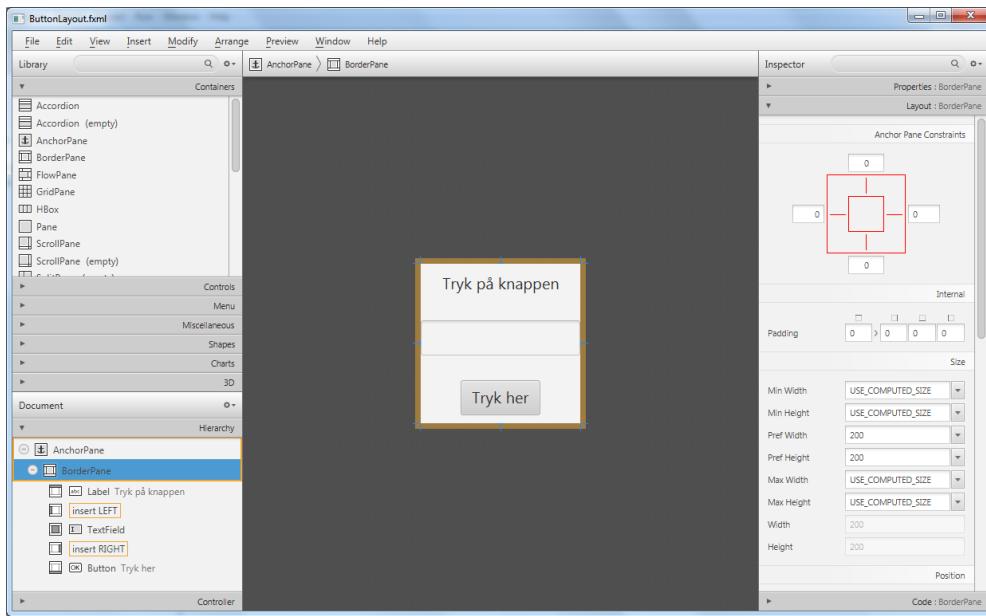
Træk her en Containers-BorderPane ind på designfladen.

Og derefter en Controls-Button ind i bunden af BorderPane.

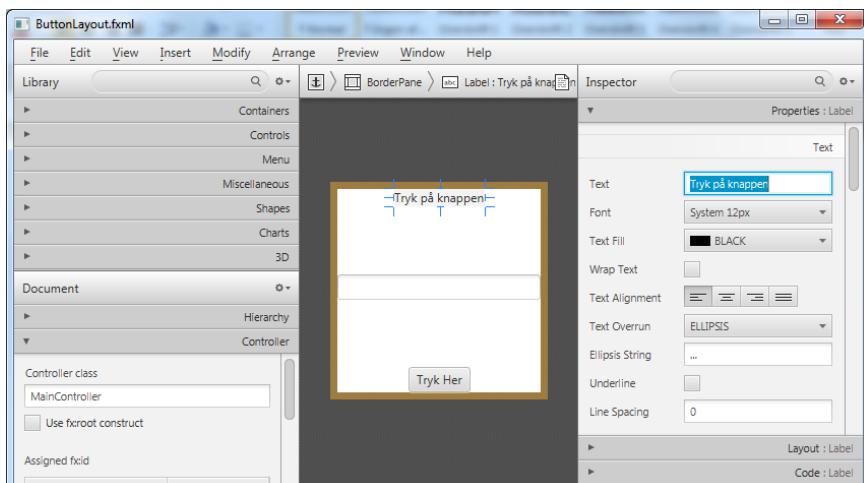


Placer en Controls-Label i toppen og en Controls-TextField i midten.

Sørg for at sætte BorderPane Anchor Pane Constraints til 0.(Se nedenfor). Ellers indeholder vinduet intet når programmet udføres.



Sæt i knappens Properties dens Text til Tryk Her.



Sæt labelens Text til Tryk på knappen.

Save og vend tilbage til Eclipse.

Åben ButtonLayout.fxml, som nu skulle se sådan ud.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.text.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>

<AnchorPane xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1">
    <children>
        <BorderPane layoutX="-148.0" layoutY="-239.0" prefHeight="200.0" prefWidth="200.0"
                    AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0"
                    AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
            <bottom>
                <Button fx:id="btnTryk" mnemonicParsing="false" text="Tryk her"
                        BorderPane.alignment="CENTER">
                    <BorderPane.margin>
                        <Insets bottom="10.0" />
                    </BorderPane.margin>
                    <font>
                        <Font size="20.0" />
                    </font>
                </Button>
            </bottom>
            <top>
                <Label text="Tryk på knappen" BorderPane.alignment="CENTER">
                    <BorderPane.margin>
                        <Insets top="10.0" />
                    </BorderPane.margin>
                    <font>
                        <Font size="20.0" />
                    </font>
                </Label>
            </top>
            <center>
                <TextField fx:id="txtTryk" BorderPane.alignment="CENTER">
                    <font>
                        <Font size="20.0" />
                    </font>
                </TextField>
            </center>
        </BorderPane>
    </children>
</AnchorPane>
```

Metoden start() i klassen Main rettes til så designet loades.

```
public class Main extends Application
{
    @Override
    public void start(Stage primaryStage) throws IOException
    {
        Parent root = FXMLLoader.load(getClass().getResource("../fxml/ButtonLayout.fxml"));

        Scene scene = new Scene(root, 200, 200);
        primaryStage.setScene(scene);
        primaryStage.setTitle("ButtonEvent");
        primaryStage.show();
    }
}
```

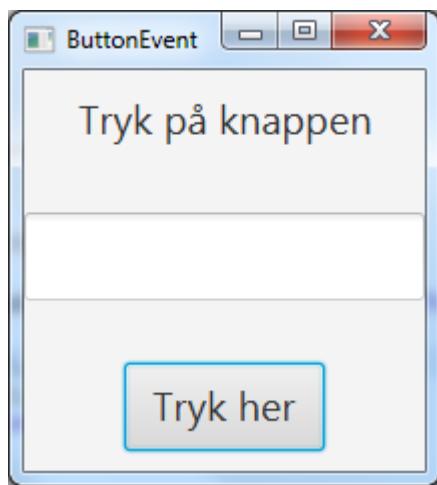
```

    }

    public static void main(String[] args)
    {
        Launch(args);
    }
}

```

Udfør programmet og der skulle komme følgende vindue.



Der tilføjes nu en klasse MainController som implementerer Initializable.

Til klassen tilføjes eventhandleren btnTryk_Click().

```

package application;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.Initializable;

public class MainController implements Initializable
{

    @Override
    public void initialize(URL location, ResourceBundle resources)
    {
        // TODO Auto-generated method stub
    }

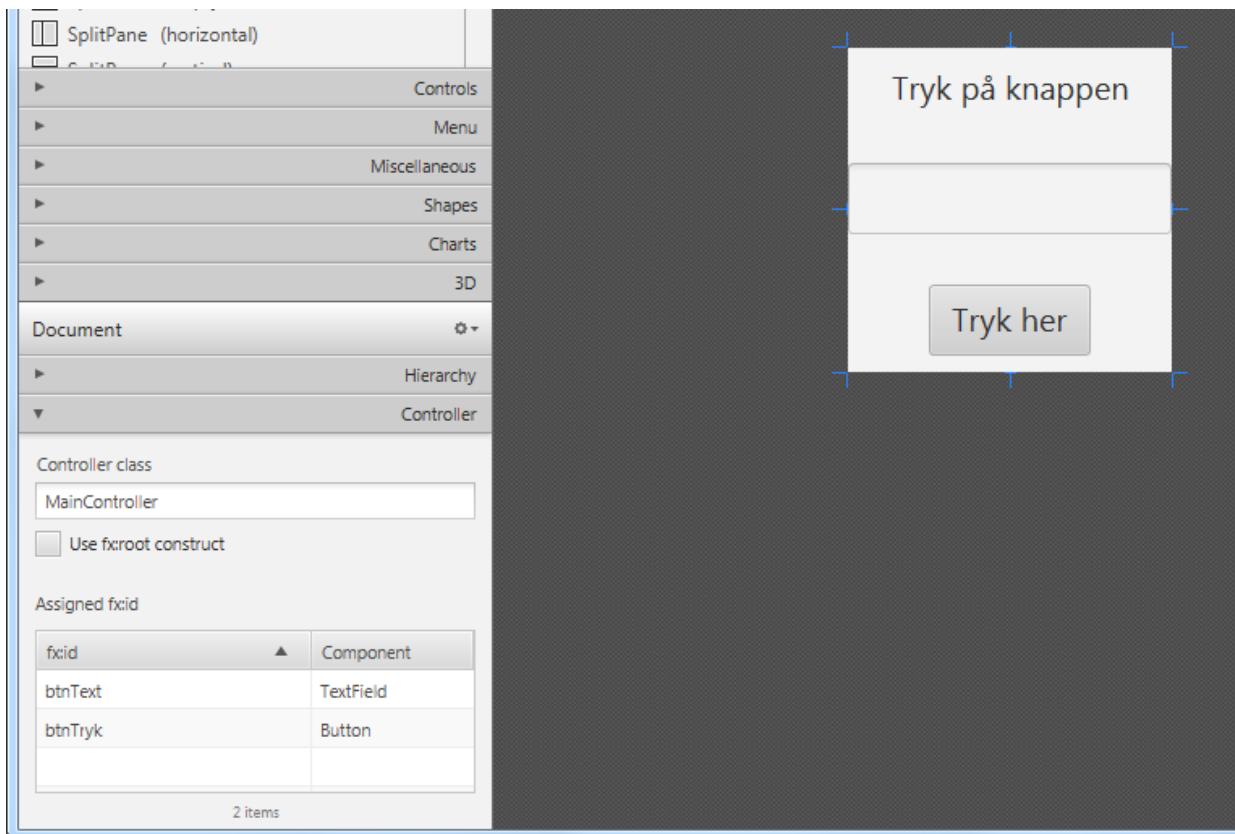
    public void btnTryk_Click(ActionEvent event)
    {
    }
}

```

Nu skal dette knyttes til knappen.

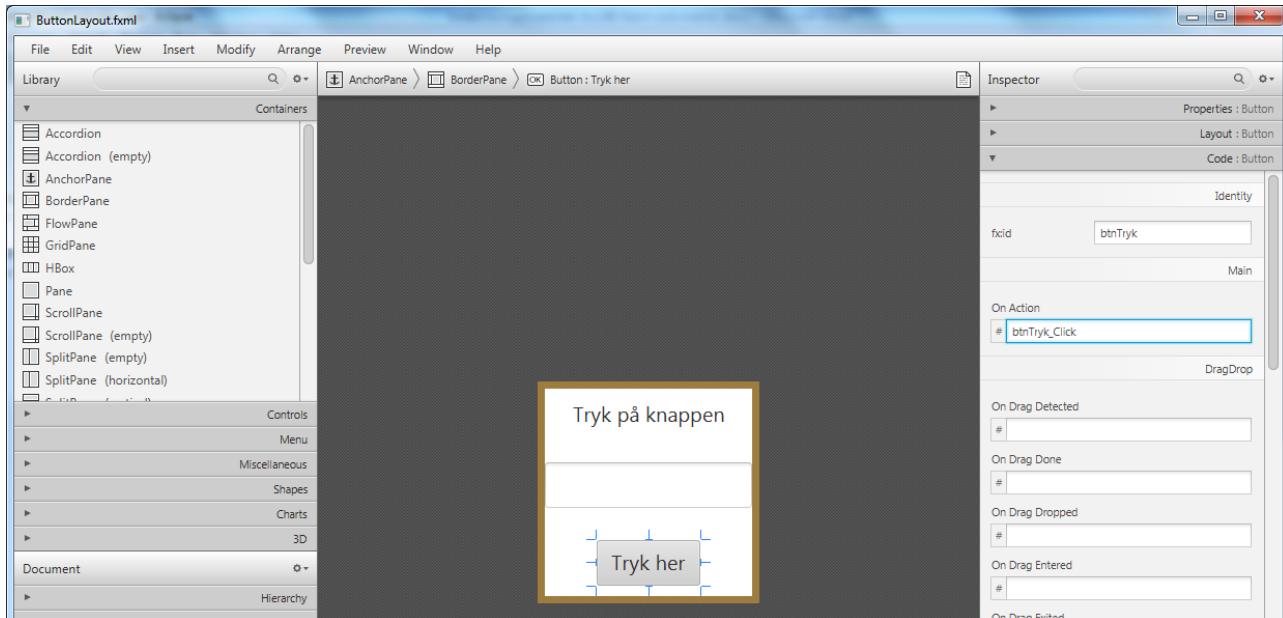
I SceneBuilder markeres hele layoutet.

I venstre side udfoldes Controller og her sættes Controller class til MainController. Dette gælder for hele layoutet.



Knappen markeres og under dens Code i højre side, sættes On Action til btnTryk_Click og knappens id til btnTryk.

Marker også TextField'en og giv den id txtTryk.



Så ser fxml-koden sådan ud.

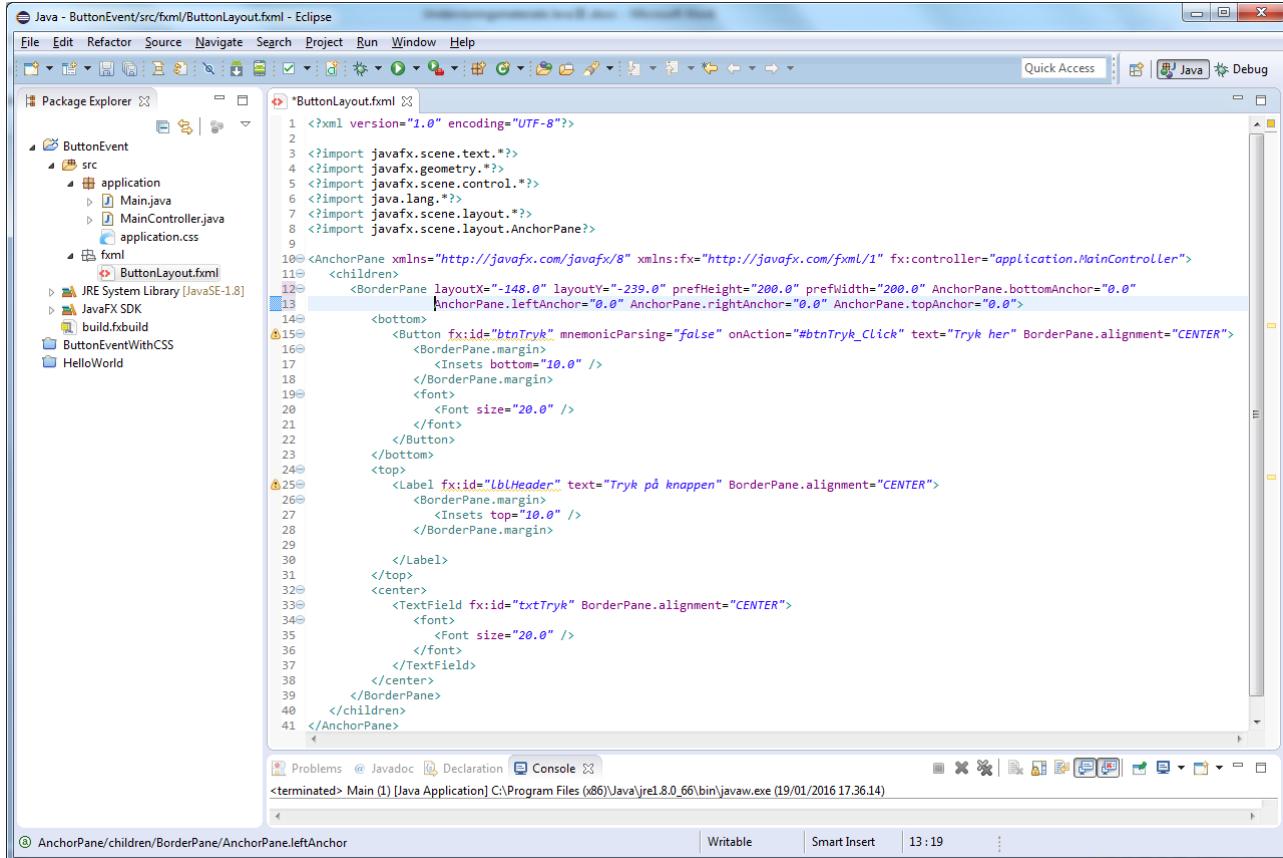
```
...
```

```
<AnchorPane xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
             fx:controller="MainController">
    <children>
```

```

<BorderPane layoutX="-148.0" layoutY="-239.0" prefHeight="200.0" prefWidth="200.0"
            AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0"
            AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
    <bottom>
        <Button fx:id="btnTryk" mnemonicParsing="false" onAction="#btnTryk_Click"
                text="Tryk her" BorderPane.alignment="CENTER">
...

```



Der tilføjes nu en field txtTryk til MainController, når den får annotation @FXML vil den blive sat til at referere til den TextField i FXML, som har samme id.

Og så kan der skrives til den i knappens eventhandler.

(Pas på at TextField er javafx.scene.control.TextField og ikke java.awt.TextField, hvilket giver en masse exceptions)

```

public class MainController implements Initializable
{
    @FXML
    private TextField txtTryk;

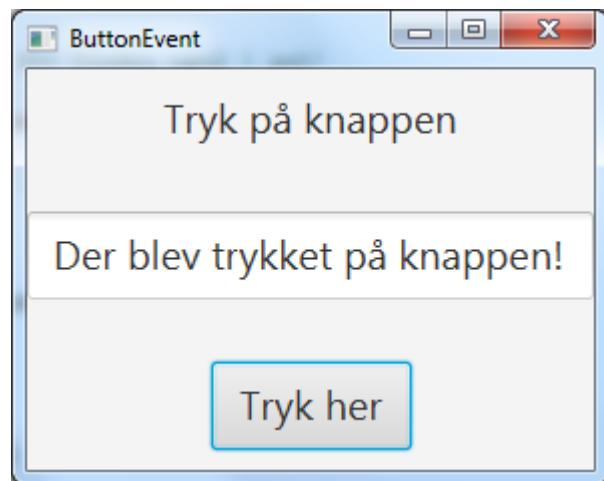
    @Override
    public void initialize(URL location, ResourceBundle resources)
    {
        // TODO Auto-generated method stub
    }

    public void btnTryk_Click(ActionEvent event)
    {
        txtTryk.setText("Der blev trykket på knappen!");
    }
}

```

}

Så virker det.



Styling af FXML med CSS

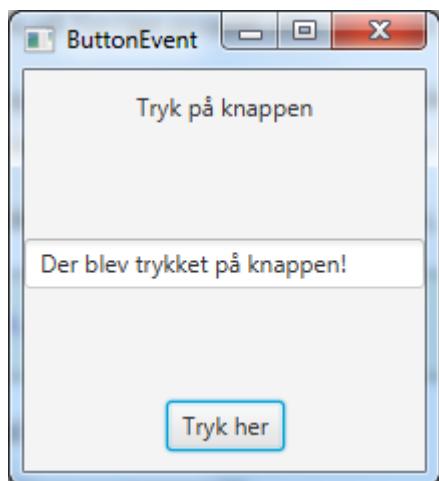
Der arbejdes videre på forrige projekt eller en kopi af denne.

Der kan laves en kopi af projektet ved at højre-klikke på projektnavnet i Package Explorer, copy, paste i tomt område, og navngive kopien af projektet, her ButtonEventWithCSS.

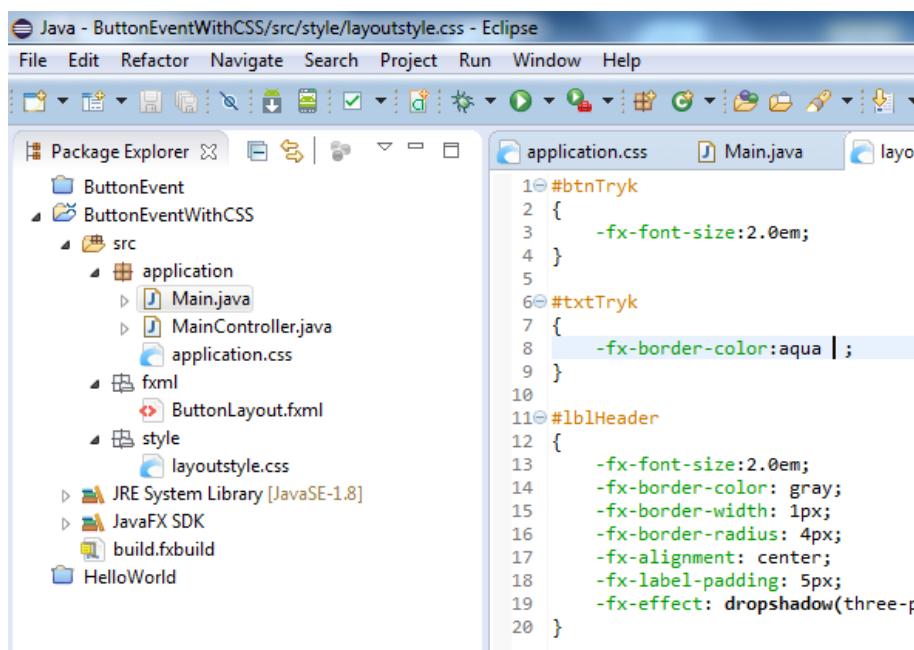
Der fjernes følgende sæt font-strørrelse fra Button, Label og TextField i brugerfladen.

```
<font>
  <Font size="20.0" />
</font>
```

Så ser det sådan ud.



Der tilføjes en package style og derunder en ny fil af navnet layoutstyle.css med følgende indhold.



```

#btnTryk
{
    -fx-font-size:2.0em;
}

#txtTryk
{
    -fx-border-color:aqua ;
}

#lblHeader
{
    -fx-font-size:2.0em;
    -fx-border-color: gray;
    -fx-border-width: 1px;
    -fx-border-radius: 4px;
    -fx-alignment: center;
    -fx-label-padding: 5px;
    -fx-effect: dropshadow(three-pass-box, rgba(0,0,0, 0.7), 1, 2, 1, 1);
}

```

I Java-koden skal style-filen loades. Så ser start() sådan ud.

```

public void start(Stage primaryStage) throws IOException
{
    Parent root = FXMLLoader.load(getClass().getResource("../fxml/ButtonLayout.fxml"));

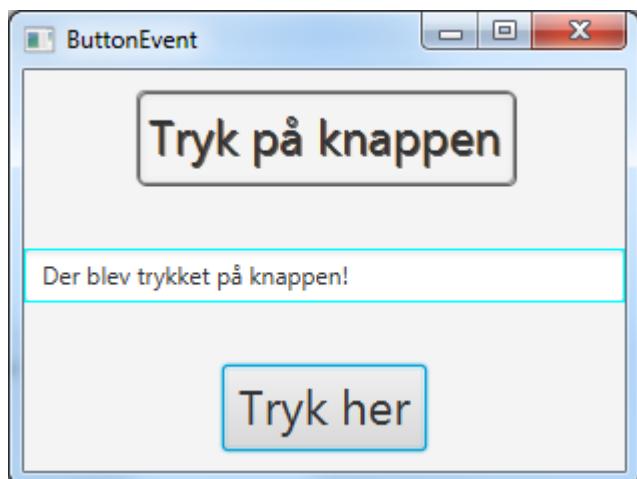
    Scene scene = new Scene(root, 300, 200);

    scene.getStylesheets().add("/style/layoutstyle.css"); // Tilføjes

    primaryStage.setScene(scene);
    primaryStage.setTitle("ButtonEvent");
    primaryStage.show();
}

```

Og programmet ser sådan ud.

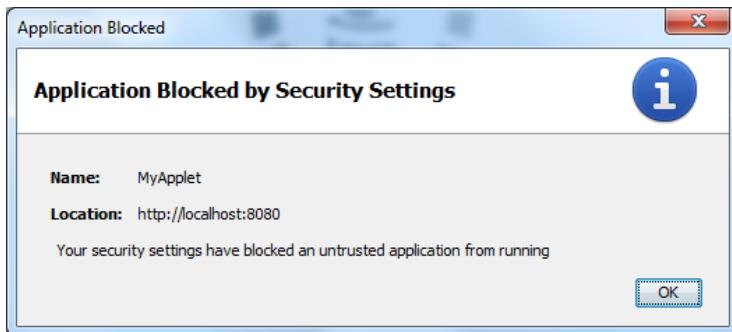


4: Eleven kan anvende events programmering og Applets, der er baseret på Java-teknologien.

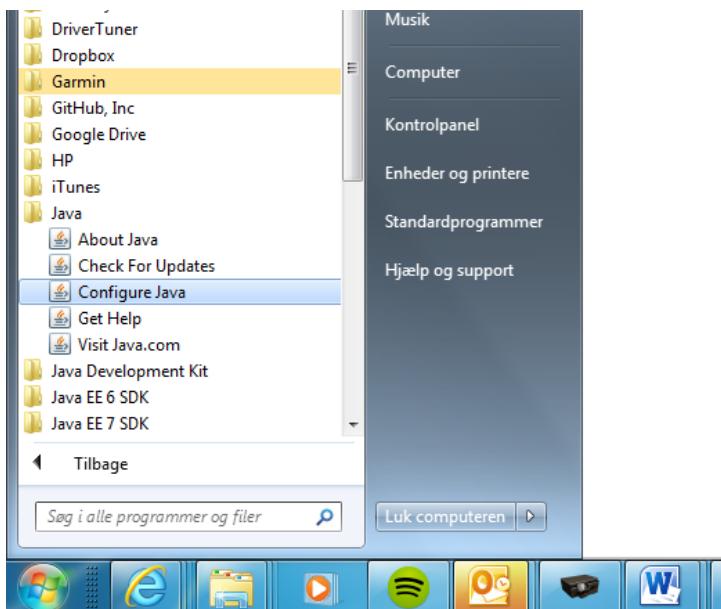
Applets

Sikkerhed

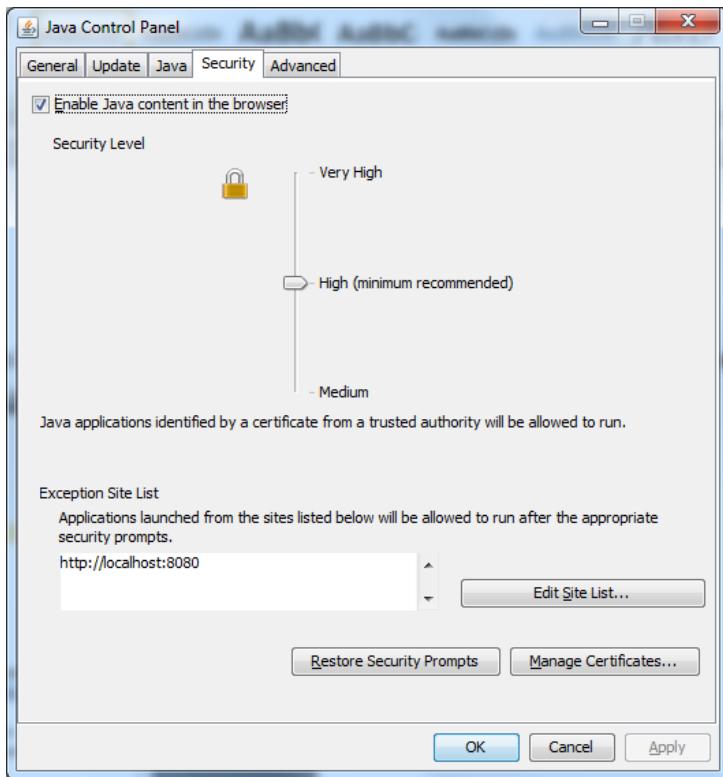
Det kan på grund af sikkerhedsrestriktioner være svært at få lov at afprøve en Applet man har udviklet.



Det kan løses ved at indsætte web-adressen i Java Control Panel.



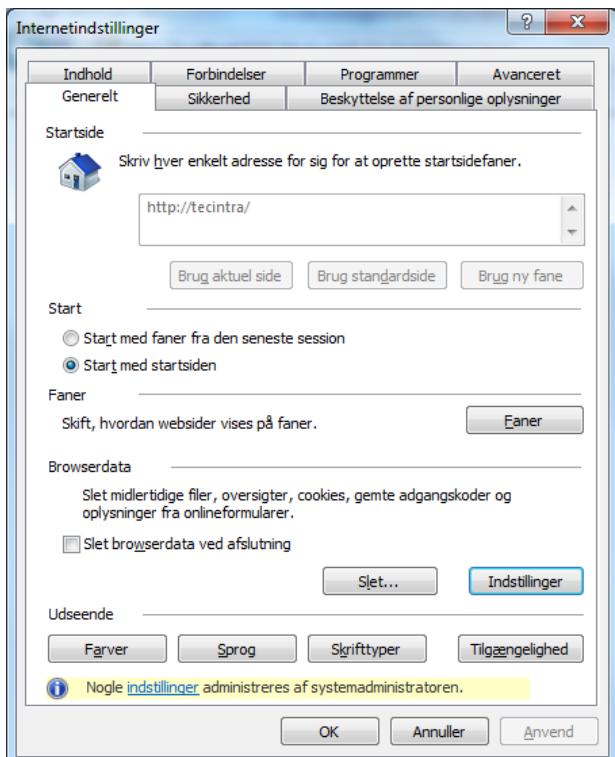
Vælg Java – Configure Java



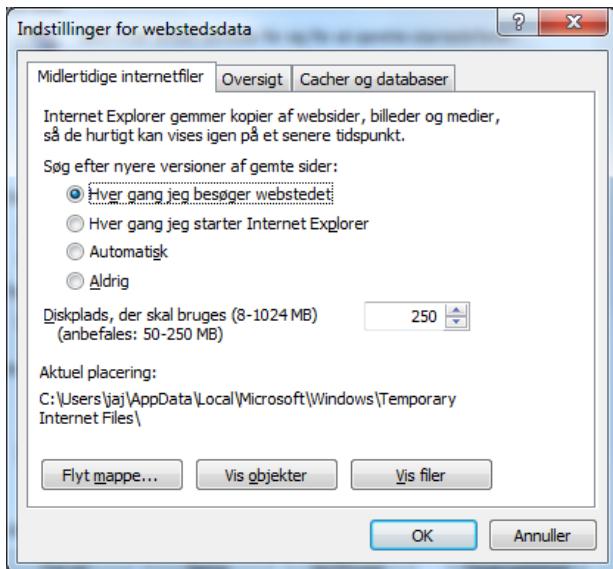
Vælg Security og tryk Edit Site List... og Add <http://localhost:8080> eller som det nu er.

Det kan stadig være svært at få seneste version af Appletten at se i browseren.

Gå ind i Funktioner - Internetindstillinger



Tryk på Indstiller.



Vælg: Søg efter nyere versioner af gemte sider: Hver gang jeg besøger webstedet.

Hvis der stadig er problemer, kan man prøve at ændre web-adressen hver gang der afprøves, ved at ændre navnet på mappen med Applettens class-fil og den kaldende html-fil.

Life Cycle of an Applet:

Four methods in the Applet class give you the framework on which you build any serious applet:

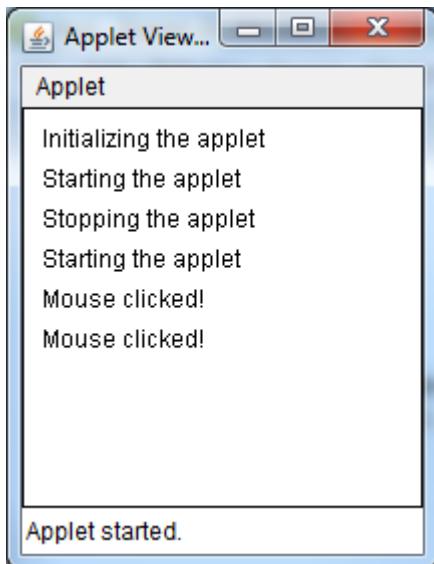
- **init:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint:** Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

En Applet der viser kaldte metoder samt mouse-events

Appletten udskriver i grafikken metodekald og mouseevents.

Alle metoder i interfacet MouseListener skal jo implementeres, også selv om der kun skal bruges nogle af dem.

Appletten er afprøvet i Eclipse AppletViewer, så der er ikke vist nogen html-fil.



```
public class MyMouseApplet extends JApplet implements MouseListener
{
    ArrayList<String> messages;

    public void init()
    {
        addMouseListener(this);
        messages = new ArrayList<String>();
        addMessage("Initializing the applet\n ");
    }

    public void start()
    {
        addMessage("Starting the applet\n ");
    }

    public void stop()
    {
        addMessage("Stopping the applet\n ");
    }

    public void destroy()
    {
        addMessage("Unloading the applet\n ");
    }

    void addMessage(String word)
    {
        System.out.println(word);
        messages.add(word);
        repaint();
    }

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

        for(int i = 0; i < messages.size(); i++)
        {
            g.drawString(messages.get(i), 10, i*20 + 20);
        }
    }

    @Override
    public void mouseEntered(MouseEvent event) {
```

```

@Override
public void mouseExited(MouseEvent event) {
}
@Override
public void mousePressed(MouseEvent event) {
}
@Override
public void mouseReleased(MouseEvent event) {
}

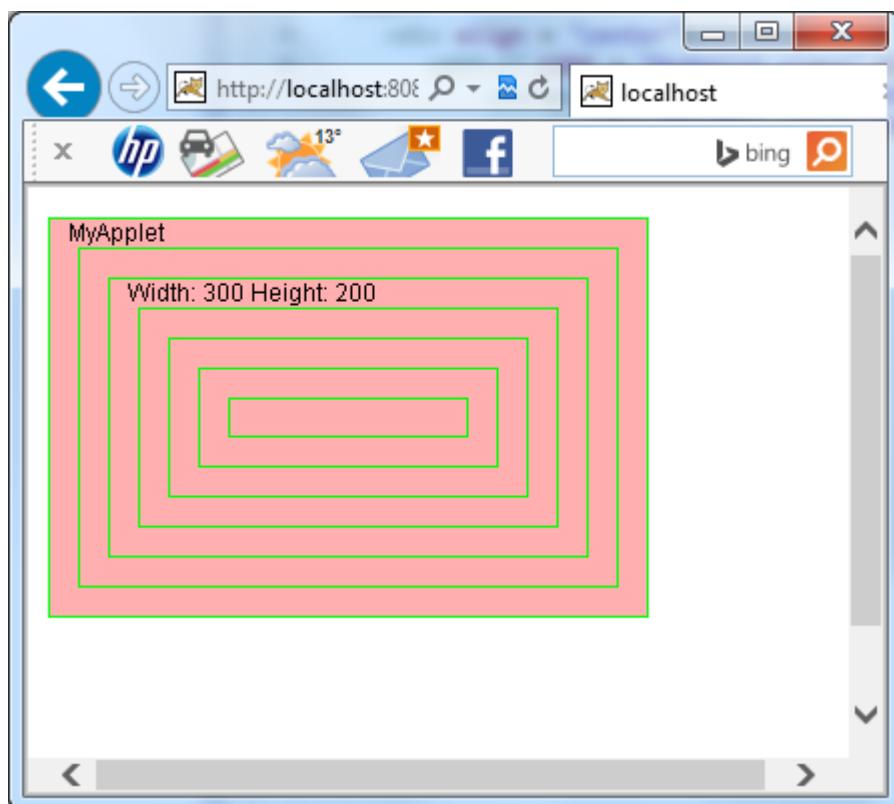
@Override
public void mouseClicked(MouseEvent event)
{
    addMessage("Mouse clicked! ");
}
}

```

En Applet der tegnes på og modtager parametre fra HTML

Parametre til Applet-klassen kan sættes i html-filens applet-tag og hentes i Applettens metode init(). Så er der mulighed for at ændre ting i Appletten uden at kode skal gen-compileres.

Applet-taggets Width- og Height-attributter bestemmer den plads der afsættes i browseren til Appletten, som er uafhængig af Applettens egentlige størrelse.



MyAppletHtml.html

```

<HTML>
<BODY>
    <div align = "center">
        <APPLET CODE = "MyApplet.class" WIDTH = "400" HEIGHT = "300">
            <param name="Width" value="300"/>
            <param name="Height" value="200"/>
        </APPLET>
    </div>
</BODY>

```

MyApplet.java

```

public class MyApplet extends JApplet
{
    private static final long serialVersionUID = 1L;
    private int width = 150;
    private int height = 100;

    public void init()
    {
        String strWidth = getParameter("Width");
        String strHeight = getParameter("Height");
        if(strWidth != null && strHeight != null)
        {
            width = Integer.parseInt(strWidth);
            height = Integer.parseInt(strHeight);
        }
    }

    public void paint(Graphics g)
    {
        int inset;
        int rectWidth, rectHeight;
        g.setColor(Color.pink);
        g.fillRect(0,0,width,height);
        g.setColor(Color.green);
        inset = 0;
        rectWidth = width - 1;
        rectHeight = height - 1;
        while (rectWidth >= 0 && rectHeight >= 0)
        {
            g.drawRect(inset, inset, rectWidth, rectHeight);
            inset += 15;
            rectWidth -= 30;
            rectHeight -= 30;
        }
        g.setColor(Color.black);
        g.drawString("MyApplet", 10, 12);
        g.drawString("Width: " + width + " Height: " + height, 40, 42);
    }
}

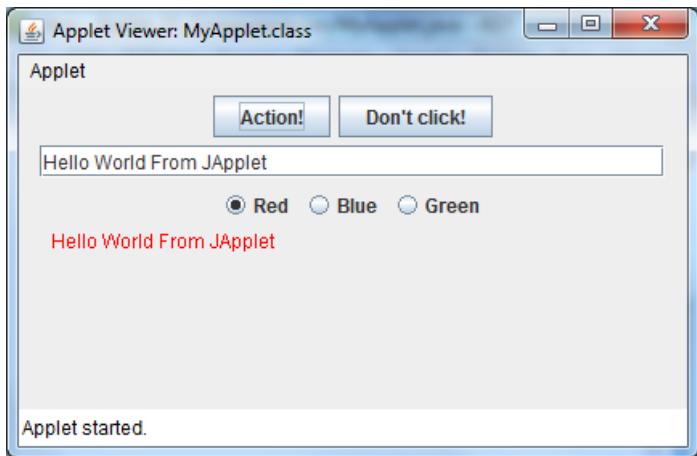
```

Applet med Button, RadioButton, TextField og grafik

For at RadioButtons kan virke sammen, skal de tilføjes til en ButtonGroup, som i øvrigt ikke har noget med med layoutet at gøre.

Når der trykkes på knappen Action, læses indholdet i tekstfeltet og tegnes i grafikken med den farve, der er valgt i radionbuttons.

Appletten er bare vist i Eclipse Applet Viewer, altså uden en html-fil der viser den.



```
public class MyApplet extends JApplet implements ActionListener
{
    JButton btnOK;
    JButton btnWrong;
    JTextField txtWrite;
    ButtonGroup radioGroup;
    JRadioButton rdbRed;
    JRadioButton rdbBlue;
    JRadioButton rdbGreen;

    public void init()
    {
        setLayout(new FlowLayout());
        btnOK = new JButton("Action!");
        btnWrong = new JButton("Don't click!");
        txtWrite = new JTextField("Type here Something",35);
        radioGroup = new ButtonGroup();
        rdbRed = new JRadioButton("Red", false);
        rdbBlue = new JRadioButton("Blue", true);
        rdbGreen = new JRadioButton("Green", false);
        add(btnOK);
        add(btnWrong);
        add(txtWrite);
        radioGroup.add(rdbRed);
        radioGroup.add(rdbBlue);
        radioGroup.add(rdbGreen);
        add(rdbRed);
        add(rdbBlue);
        add(rdbGreen);

        btnOK.addActionListener(this);
        btnWrong.addActionListener(this);
    }

    public void paint(Graphics g)
    {
        super.paint(g);

        if (rdbRed.isSelected())
        {
            g.setColor(Color.red);
        }
        else if (rdbBlue.isSelected())
        {
            g.setColor(Color.blue);
        }
        else
        {
            g.setColor(Color.green);
        }
    }
}
```

```
    g.drawString(txtWrite.getText(),20,100);  
}  
  
public void actionPerformed(ActionEvent evt)  
{  
    if (evt.getSource() == btnOK)  
    {  
        repaint();  
    }  
    else if (evt.getSource() == btnWrong)  
    {  
        btnWrong.setText("Not here!");  
        txtWrite.setText("That was the wrong button!");  
        repaint();  
    }  
}
```

5: Eleven kan programmere Standalone Java programmer og bruge Frame og Menuklasser til at tilføje grafik til Java programmer.

Menuer laver vi i JavaFX!!!

6: Eleven kan fremstille programmer, der anvender Multithreading.

(Cave of Programming – Advanced Java: Multi-threading Part 1-15)

Der er principielt to måder at oprette en ny tråd på

1. Ved at oprette et objekt af en klasse der arver fra Thread og overrider metoden run(), som hidrører fra interfacet Runnable.
2. Ved at oprette et objekt af klassen Thread og tildele denne et objekt af en klasse, der implementerer interfacet Runnable og dermed metoden run().

Når en ny tråd startes, er det metoden run() der kaldes. Så det, som en tråd udfører, er den kode der ligger i metoden run().

En klasse der arver fra Thread

Følgende eksempel er klassen MyThread, der arver fra Thread og overrider metoden run(), hvor denne gennemløber en løkke og udskriver tallene 0 - 4.

Kald af metoden sleep() lader den aktuelle tråd holde pause i de angivne millisekunder. Her er for sjov brugt en random, da der ønskes oprettet to tråde, der hver afvikler sin run() metode.

Hvis en tråd interruptes, vil dette ske i sleep() metoden (eller anden ventende metode), derfor kræves der en håndtering af InterruptedException.

For hvert gennemløb udskrives trådens navn og tæller'en i.

```
public class MyThread extends Thread
{
    Random r = new Random();

    public void run()
    {
        for(int i = 0; i < 5; i++)
        {
            System.out.println(Thread.currentThread().getName() + ": " + i);
            try {
                Thread.sleep(r.nextInt(200));
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

Der afprøves ved at oprette to objekter af MyThread og metoden start() kaldes, hvorved metoden run() udføres i den nye tråd.

Hvis metoden run() kaldes direkte bliver den udført i main-tråden.

```
public class ThreadTest
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread();
```

```

MyThread t2 = new MyThread();
t1.start();
t2.start();

System.out.println("Thread " + Thread.currentThread().getName() + " End");
}
}

```

Output:

```

Thread main End
Thread-1: 0
Thread-0: 0
Thread-1: 1
Thread-0: 1
Thread-0: 2
Thread-1: 2
Thread-1: 3
Thread-0: 3
Thread-1: 4
Thread-0: 4

```

Det ses at main-tråden slutter så snart den har igangsat de to nye tråde, og de to tråde udføres parallelt i henhold til den randomiserede sleep.

Det ses i øvrigt at main-tråden hedder 'main' og de to nye tråder hedder 'Thread-0' og 'Thread-1'. En tråds navn kan sættes i Thread-constructoren og senere med metoden setName().

En klasse der implementerer Runnable

En anden måde er at oprette en klasse der implementerer interfacet Runnable, og give et objekt af denne til et nyt generelt Thread-objekt.

Klassen er fuldstændig magen til den forrige bortset fra at den implementeres Runnable i stedet for at arve Thread.

```

public class MyRunnable implements Runnable
{
    Random r = new Random();

    public void run()
    {
        for(int i = 0; i < 5; i++)
        {
            System.out.println(Thread.currentThread().getName() + ": " + i);
            try {
                Thread.sleep(r.nextInt(200));
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}

```

Der afprøves ved at oprette to nye Thread-objekter med en constructor, der modtager et objekt af klassen MyRunnable der implementerer Runnable.

```

public class ThreadTest

```

```

{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new MyRunnable());
        Thread t2 = new Thread(new MyRunnable());

        t1.start();
        t2.start();

        System.out.println("Thread " + Thread.currentThread().getName() + " End");
    }
}

```

Output er det samme som ovenfor.

Oprettelse af en tråd med anonym klasse der implementerer Runnable

Når der til Thread-constructoren skal angives et objekt af en klasse der implementerer interfacet Runnable, kan denne klasse defineres som en anonym klasse på stedet. Syntaksen `new Runnable(){...}` betyder at der oprettes et objekt af en klasse der er defineret imellem de to braces og som implementerer Runnable.

Eksemplet viser desuden at den nye tråd gives et navn i oprettelsen,

```
Thread t3 = new Thread(new Runnable(){...}, "ThirdThread");
```

og at hovedtråden får tildelt et nyt navn med

```
Thread.currentThread().setName("MainThread");
```

Ellers er det det samme som forrige eksempel

```

public class ThreadTest
{
    public static void main(String[] args)
    {
        Thread.currentThread().setName("MainThread");

        Thread t3 = new Thread(new Runnable()
        {
            Random r = new Random();

            public void run()
            {
                for(int i = 0; i < 5; i++)
                {
                    System.out.println(Thread.currentThread().getName() + ": " + i);
                    try {
                        Thread.sleep(r.nextInt(200));
                    } catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        }, "ThirdThread");

        t3.start();
    }
}

```

```

        System.out.println("Thread " + Thread.currentThread().getName() + " End");
    }
}

```

Output:

```

Thread MainThread End
ThirdThread: 0
ThirdThread: 1
ThirdThread: 2
ThirdThread: 3
ThirdThread: 4

```

Volatile keyword

I det følgende eksempel er vist en klasse, Worker, som er en tråd der kører i en løkke indtil den udefra bliver stoppet ved at kalde metoden stopMe(), som ændrer attributten stop, der er betingelsen for while-løkkens udførelse.

I visse systemer kan en tråd finde på at cache en variabel, så den ikke opdager når den bliver ændret andetstedsfra. (Det her eksempel virker lige godt med og uden volatile i mit tilfælde på en Windows 7)

Keywordet volatile forhindrer en tråd i at cache den pågældende variabel.

```

class Worker extends Thread
{
    private volatile boolean stop = false;

    @Override
    public void run()
    {
        int count = 0;

        while(!stop)
        {
            System.out.println("Hello " + count++);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void stopMe()
    {
        stop = true;
    }
}

```

Der oprettes og startes en Worker-tråd, som kører indtil der trykkes enter.

```

public class ThreadTest
{
    public static void main(String[] args)
    {
        Worker w = new Worker();
        w.start();

        System.out.println("Press enter to stop!");
    }
}

```

```

Scanner scan = new Scanner(System.in);
scan.nextLine();

w.stopMe();

System.out.println("Stopped");
}
}

```

Output:

```

Press enter to stop!
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5

Hello 6
Stopped

```

Data der manipuleres af flere tråde – synchronized (og join())

Synchronized generelt

Når den samme variabel manipuleres af flere tråde, kan der ske fejl når manipulationen ikke er atomisk, hvilket vil sige at der indgår flere operationer i manipulationen.

I det følgende eksempel bliver attributten count forøget af en tråd og formindsket af en anden tråd. Kommandoen `count++`; læser først indholdet, lægger 1 til værdien og skriver resultatet tilbage i attributten. Det betyder at den ene tråd kan overskrive en værdi som den anden tråd lige har skrevet.

I koden har de to metoder increment() og decrement() fået modifier synchronized. Dette gør at metoden kun kan udføres af en tråd ad gangen. Den første tråd der kalder metoden får dens lås, og først når metoden er afsluttet frigives låsen, så næste tråd kan få den og gå ind.

synchronized bruger et objekt som lås, og når der som her ikke er nævnt et objekt sammen med synchronized, bliver der låst på det aktuelle objekt (this). Dette betyder at når en tråd er inde i den ene metode, der er synchronized, har den låsen, som også skal bruges til den anden metode. Derfor er der kun en af metoderne der kan udføres ad gangen. Men det er jo også det ønskede, da de arbejder på samme data.

Da værdien af attributten count skal skrives ud til sidst, bliver hovedtråden nødt til at vente på at de to tråde er afsluttede. Dette gøres ved at hovedtråden kalder metoden join() på hver af de to tråde, sådan at hovedtråden ikke går videre før t1 er afsluttet og derefter ikke går videre før t2 er afsluttet. Det gør ikke noget at t2 allerede skulle være afsluttet ved kaldet af join().

```

public class ThreadSync
{
    static int count = 0;

    static synchronized void increment()
    {
        count++;
    }
}

```

```

static synchronized void decrement()
{
    count--;
}

public static void main(String[] args)
{
    Thread t1 = new Thread(new Runnable()
    {
        @Override
        public void run()
        {
            for(int i = 0; i < 1000; i++)
            {
                increment();
            }
        }
    });
    Thread t2 = new Thread(new Runnable()
    {
        @Override
        public void run()
        {
            for(int i = 0; i < 1000; i++)
            {
                decrement();
            }
        }
    });
}

t1.start();
t2.start();

try {
    t1.join();
    t2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println("Value of count: " + count);
}
}

```

Output uden synchronized efter flere udførelser af programmet:

```

Value of count: 0
Value of count: -999
Value of count: -650
Value of count: 52
Value of count: -53

```

Output med synchronized er altid:

```

Value of count: 0

```

Synchronized med flere objekter som lås

I det følgende eksempel er der to attributter som forøges med hver sin metode incrementCount1() og incrementCount2(). De to metoder kaldes begge af to forskellige tråde. Der er lagt en lille forsinkelse ind i metoderne, sådan at det er muligt at måle hvor mange millisekunder afviklingen af trådene tager.

Programmet er gjort lidt mindre static end det forrige og der oprettes et objekt af klassen i main().

Programmet er vist herunder uden brug af synchronized (bare lige for at se at det fejler).

De to attributter lock1 og lock2 bruges ikke endnu.

```
public class ThreadSyncObject
{
    int count1 = 0;
    int count2 = 0;

    Object lock1 = new Object();
    Object lock2 = new Object();

    void incrementCount1()
    {
        count1++;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    void incrementCount2()
    {
        count2++;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args)
    {
        final ThreadSyncObject app = new ThreadSyncObject();

        Thread t1 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                for(int i = 0; i < 1000; i++)
                {
                    app.incrementCount1();
                    app.incrementCount2();
                }
            }
        });

        Thread t2 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                for(int i = 0; i < 1000; i++)
                {

```

```

        app.incrementCount1();
        app.incrementCount2();
    }
});

long start = System.currentTimeMillis();

t1.start();
t2.start();
try {
    t1.join();
    t2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

long end = System.currentTimeMillis();

System.out.println("Value of count1: " + app.count1 +
                    " Value of count2: " + app.count2 + " Time: " + (end-start));
}
}

```

Output uden synchronized. Begge variable skulle være 2000:

Value of count1: 1999 Value of count2: 1998 Time: 2018

Nu kan der tilføjes synchronized modifier til metoderne incrementCount1() og incrementCount2()

```

synchronized void incrementCount1()
{
    count1++;
    try {
        Thread.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

Men i stedet angives der en synchronized blok af kode, nemlig hele koden inde i de to metoder. Så kan der angives et objekt, der bruges som lås. Her bruges this, hvilket får den samme effekt som når synchronized står som modifier til metoderne. Når dette gøres på begge metoder bruger de samme lås, og dermed kan der kun tilgås en af metoderne ad gangen. For at bruge this, kræves at det ikke er en static kontekst.

```

void incrementCount1()
{
    synchronized(this)
    {
        count1++;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Output:

Value of count1: 2000 Value of count2: 2000 Time: 4203

Nu fejler programmet ikke mere, men det er kommet til at tage dobbelt så lang tid, da der kun kan tilgås en af metoderne ad gangen.

De to metoder modificherer hver sin variabel, så derfor er det ikke hensigtstmæssigt at kun en af dem kan udføres ad gangen. For at løse dette problem får de to metoder hver sin lås, nemlig de to attributter. Ethvert objekt i Java kan bruges som lås på en synchronized blok.

```
Object lock1 = new Object();
Object lock2 = new Object();

void incrementCount1()
{
    synchronized(lock1)
    {
        count1++;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

void incrementCount2()
{
    synchronized(lock2)
    {
        Count2++;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Value of count1: 2000  Value of count2: 2000 Time: 2160
```

Nu kan hver metode kun tilgås af en tråd ad gangen, men de to metoder kan udføres samtidig af hver sin tråd, og så går det hele hurtigere igen.

Opgaver

1. Skriv et program, der opretter tre tråde, som udfører vidt forskellige ting, som intet har med hinanden at gøre.
 - En tråd der meget langsomt udskriver tallene fra 0-10.
 - En tråd der udskriver en række navne fra et array lidt hurtigere.
 - En tråd der bruger en input-dialog og spørger brugeren om dennes navn.
2. Skriv et program der opretter to tråde som henholdsvis sætter variablen navn, og læser variablen navn.
 - Den ene tråd skal gentaget spørge om indtastning af et navn, og dernæst om det ønskes at fortsætte. Hvis ikke skal tråden afsluttes.
 - Den anden tråd skal tælle en tæller op hver gang der kommer et nyt navn, indtil navn = "Jan", hvor den afsluttes.

- Opret en klasse Data, som indeholder navn, tæller og et flag der **gør** at når en tråd afsluttes, så afsluttes den anden også.
- Udskriv til sidst hvor mange navne, der blev indtastet.

Thread pools

En ThreadPool er et objekt, der på en gang opretter et antal tråde. Disse tråde kan så tildeles Runnable-objekter. Hvis der tildeles flere Runnable-objekter end der er tråde, vil hver tråd tage et nyt objekt i behandling når den er færdig med den forrige.

Eks.

Klassen Processor har en run() som udskriver at den starter, venter 5 sekunder og udskriver at den er færdig. Den har en id, som sættes i constructor, så den kan genkendes.

I main() oprettes en pool med 5 tråde og der tildeles 5 Processor-objekter, den lukkes ned med respekt for de startede Runnables og der ventes i hovedtråden til alle er afsluttede. Der ventes maksimalt 1 dag. Det angivne skal bare være større end den tid det tager at afvikle trådene.

```
class Processor implements Runnable
{
    private int id;

    public Processor(int id)
    {
        this.id = id;
    }

    @Override
    public void run()
    {
        System.out.println("Starting: " + id);

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Completed: " + id);
    }
}

public class ThreadPoolTest
{

    public static void main(String[] args)
    {
        ExecutorService executor = Executors.newFixedThreadPool(5);

        for(int i = 0; i < 5; i++)
        {
            executor.submit(new Processor(i));
        }

        executor.shutdown();

        System.out.println("All tasks submitted.");

        try {

```

```

        executor.awaitTermination(1, TimeUnit.DAYS);

    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("All tasks completed.");
}
}

```

Output:

```

Starting: 1
Starting: 0
Starting: 2
Starting: 3
Starting: 4
All tasks submitted.
Completed: 0
Completed: 3
Completed: 2
Completed: 1
Completed: 4
All tasks completed.

```

Det ses at hver tråd får tildelt et Runnable-objekt, som bliver udført parallelt.

Hvis der f.eks. kun er to tråde

```
ExecutorService executor = Executors.newFixedThreadPool(2);
```

Vil de to tråde hver tage sig af et Processor-objekt ad gangen indtil alle er udført.

Det tager en del ressourcer at oprette en tråd, derfor kan det være bedst ikke at oprette alt for mange tråde, men at lade trådene tage sig af flere forløb.

Output:

```

Starting: 0
Starting: 1
All tasks submitted.
Completed: 1
Starting: 2
Completed: 0
Starting: 3
Completed: 2
Starting: 4
Completed: 3
Completed: 4
All tasks completed.

```

CountDownLatch

Java indeholder en række færdige klasser der er thread-safe. Dvs at man behøver ikke at bekymre sig om at lade flere tråde operere på det samme objekt.

Klassen CountDownLatch kan tælles ned fra en startværdi og den har desuden metoden wait(), hvor den kaldende tråd venter indtil latchen er talt ned til 0.

I følgende eksempel er klassen Processor en Runnable, hvor den overgivne latch tælles ned med 1.

I main oprettes en latch med værdien 3 og denne overgives til tråde, der udfører 5 Processor-objekter.

```
class Processor implements Runnable
{
    private CountDownLatch latch;

    public Processor(CountDownLatch latch)
    {
        this.latch = latch;
    }

    @Override
    public void run()
    {
        System.out.println("Thread Started.");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        latch.countDown();
        System.out.println("Thread Completed.");
    }
}

public class ThreadLatchTest
{
    public static void main(String[] args)
    {
        CountDownLatch latch = new CountDownLatch(3);

        ExecutorService executor = Executors.newFixedThreadPool(3);

        for(int i = 0; i < 5; i++)
        {
            executor.submit(new Processor(latch));
        }

        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main Completed.");
    }
}
```

Output:

```
Thread Started.
Thread Started.
Thread Started.
Thread Completed.
Thread Completed.
```

```
Thread Started.  
Thread Completed.  
Thread Started.  
Main Thread Completed.  
Thread Completed.  
Thread Completed.
```

Det ses at når der er completed 3 runnables, afslutter main, fordi den venter på at latchen tælles ned til 0.

Det kunne også benyttes som det omvendte, sådan at tråde ventede på at der udefra blev talt ned til 0.

BlockingQueue

En anden thread-safe klasse er ArrayBlockingQueue, som er en af klasser der implementerer interfacet BlockingQueue.

ArrayBlockingQueue bygger på et almindeligt array, som får størrelsen ved oprettelsen og som ikke kan ændres.

Den indeholder bl. a. metoderne put() og take(), som begge er blokerende metoder. Dvs. at hvis arrayet er fyldt ved kaldet af put(), ventes der til der bliver plads. Og hvis der ikke er noget i arrayet ved kaldet af take(), ventes der til der er blevet fyldt en værdi i arrayet.

Følgende kode viser en tråd der fylder i en ArrayBlockingQueue, og en tråd der henter ud.

```
public class ThreadSafeQueueTest  
{  
    static BlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(10);  
  
    public static void main(String[] args)  
    {  
        Thread t1 = new Thread(new Runnable(){  
            public void run(){  
                try {  
                    producer();  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
    }  
}
```

```

        Thread t2 = new Thread(new Runnable(){
            public void run(){
                try {
                    consumer();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t1.start();
        t2.start();
    }

    private static void producer() throws InterruptedException
    {
        Random random = new Random();
        while(true)
        {
            Thread.sleep(400);
            queue.put(random.nextInt(100));
        }
    }

    private static void consumer() throws InterruptedException
    {
        while(true)
        {
            Thread.sleep(1000);
            Integer value = queue.take();
            System.out.println("Taken value: " + value + "; Queue size: " + queue.size());
        }
    }
}

```

Output:

```

Taken value: 77; Queue size: 1
Taken value: 58; Queue size: 3
Taken value: 66; Queue size: 4
Taken value: 17; Queue size: 6
Taken value: 84; Queue size: 7
Taken value: 6; Queue size: 9
Taken value: 55; Queue size: 9
Taken value: 19; Queue size: 10
Taken value: 69; Queue size: 9

```

Fordi opfyldningen går hurtigere end udtagningen, vokser antallet af indsatte elementer indtil køen er fuld, og så bliver det deromkring.

Wait(), notify() og notifyAll()

Hvis der i to synchronized blokke låses på samme objekt, vil et kald af metoden wait() medføre at der ventes og låsen friges. Når der så efterfølgende i den anden blok kaldes metoden notify(), vil den når blokken er færdigudført, tilbagegive låsen til den første blok og der returneres fra wait().

I det følgende eksempel er klassen Processor som har metoderne producer() der kalder wait() i en synchronized blok, og metoden consumer() der kalder notify() i en synchronized blok. De to metoder udføres i hver sin tråd i metoden main().

Det er this der låses på, og der skal lige lægges mærke til, at det er på this der kaldes wait() og notify().

Metoder wait(), notify() og notifyAll() kan kun kaldes inde i en synchronized blok, og kaldes på det objekt, der låses med.

```
public class Processor
{
    public void producer() throws InterruptedException
    {
        synchronized(this)
        {
            System.out.println("Producer thread running ....");
            this.wait();
            System.out.println("Producer thread resumed.");
        }
    }

    public void consumer() throws InterruptedException
    {
        Scanner scanner = new Scanner(System.in);
        Thread.sleep(2000);
        synchronized(this)
        {
            System.out.println("Waiting for return key ....");
            scanner.nextLine();
            System.out.println("Return key pressed.");
            // this.notifyAll();
            this.notify();
            Thread.sleep(2000);
        }
    }
}

public class TestWaitNotify
{
    public static void main(String[] args)
    {
        final Processor processor = new Processor();

        Thread t1 = new Thread(new Runnable(){
            public void run(){
                try {
                    processor.producer();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread t1 completed.");
            }
        });

        //Thread t2 = new Thread(new Runnable(){    //helt magen til t1

        Thread t3 = new Thread(new Runnable(){
            public void run(){
                try {
                    processor.consumer();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        t1.start();
        //t2.start();
        t3.start();
    }
}
```

```
}
```

Output:

```
Producer thread running ....  
Waiting for return key ....
```

```
Return key pressed.  
Thread t3 completed.  
Producer thread resumed.  
Thread t1 completed.
```

Det kan iagttages at låsen først friges når synchronized-blokken er slut og ikke allerede ved kaldet af metoden notify(). Men sådan skal det jo også være.

Når der i metoden producer() kaldes wait() og låsen friges, kan en anden tråd jo også udføre blokken og kalde wait(). Så er der lige pludselig to tråde der venter på notify. Men notify() frigiver kun låsen til den ene af trådene. Så kan man selvfølgelig kalde notify() to gange. Men i stedet kaldes notifyAll(), som frigiver låsen til alle tråde, der står og venter i wait().

Dette kan vises ved at oprette en tråd mere, t2 som er fuldstændig magen til t1, så metoden processor() udføres i to tråde umiddelbart efter hinanden, og først derefter udføres consumer() i t3 og kalder metoden notifyAll().

Output:

```
Producer thread running ....  
Producer thread running ....  
Waiting for return key ....
```

```
Return key pressed.  
Thread t3 completed.  
Producer thread resumed.  
Thread t2 completed.  
Producer thread resumed.  
Thread t1 completed.
```

Om det er t1 eller t2 der først kommer til at resume, virker lidt tilfældigt.

Wait() og notify(), et arbejdende eksempel

Klassen Processor har to metoder, producer() og consumer(), som udføres i hver sin tråd.

De arbejder begge på samme list-objekt, som producer() fylder fortløbende tal i, og consumer() henter tal ud af.

Begge metoder låser i en synchronized blok på samme objekt af navnet lock, for bedre at illustrere at det er på denne metoderne wait() og notify() kaldes.

Metoden producer() kalder wait() hvis listen er fuld, og bliver notify() af metoden consumer(), når denne henter et tal. Når listen ikke er fuld adder metoden producer() et tal til listen, og notify() så consumer kan komme videre, hvis den har kaldt wait() fordi der ikke var noget i listen.

```
public class Processor  
{
```

```

private LinkedList<Integer> list = new LinkedList<Integer>();
private final int MAXSIZE = 10;
private Object lock = new Object();

public void producer() throws InterruptedException
{
    int value = 0;
    while(true)
    {
        synchronized(lock)
        {
            while(list.size() == MAXSIZE)
            {
                lock.wait();
            }
            list.add(value++);
            lock.notify();
            Thread.sleep(500);
        }
    }
}

public void consumer() throws InterruptedException
{
    Random random = new Random();

    while(true)
    {
        synchronized(lock)
        {
            while(list.size() == 0)
            {
                lock.wait();
            }
            System.out.print("List size is: " + list.size());
            int value = list.removeFirst();
            System.out.println("; Value is: " + value);
            lock.notify();
        }
        Thread.sleep(random.nextInt(1000));
    }
}
}

public class ThreadWaitNotifyTest
{
    public static void main(String[] args)
    {
        final Processor processor = new Processor();

        Thread t1 = new Thread(new Runnable()
        {
            public void run(){
                try {
                    processor.producer();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }

    Thread t2 = new Thread(new Runnable()
    {
        public void run(){
            try {
                processor.consumer();
            } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }
});

t1.start();
t2.start();
}
}

```

Output:

```

List size is: 1; Value is: 0
List size is: 4; Value is: 1
List size is: 5; Value is: 2
List size is: 9; Value is: 3
List size is: 10; Value is: 4
List size is: 10; Value is: 5
List size is: 10; Value is: 6

```

Det ser ud til at virke, da alle fortløbende tal er med en og kun en gang.

Reentrant Locks

Et alternativ til låsninger på synchronized blokke, er Reentrant Locks. Der oprettes i det følgende projekt i klassen Runner, et objekt af klassen ReentrantLock. Når en tråd ønsker at udføre en metode, som også bruges af andre tråde, tager den låsen og låser den. Hvis ikke låsen er åben, ventes der på at den bliver åben og så låses den. Dette gør at andre der benytter låsen må vente til den er åben.

Låsen har metoderne lock() og unlock().

I det følgende har klassen Runner metoderne firstThread() og secondThread(), som udføres i hver sin tråd. De kalder begge metoden increment(), hvilket jo som tidligere er set giver fejl, da det sker parallelt. Når trådene er slut kaldes metoden finished() som udskriver værdien af attributten count, der bliver talt op i metoden increment().

Løsningen er at i begge metoder som kalder increment(), udføres der først en lock() og når increment() er færdig udføres unlock(). Den der kommer første får så lov at udføre increment() alene, og derefter får den anden lov.

Unlock() bør ske i en try/finally for at en exception i increment() ikke kommer til at forårsage at der aldrig bliver låst op igen.

```

public class Runner
{
    private int count = 0;

    private Lock lock = new ReentrantLock();

    private void increment()
    {
        for(int i = 0; i < 10000; i++ )
        {
            count++;
        }
    }

    public void firstThread()
    {
        lock.lock();
        try{
            increment();
        }finally{
            lock.unlock();
        }
    }

    public void secondThread()
    {
        lock.lock();
        try{
            increment();
        }finally{
            lock.unlock();
        }
    }

    public void finished()
    {
        System.out.println("Count is: " + count);
    }
}

public class ReentrantLocksTest {

    public static void main(String[] args)
    {
        final Runner runner = new Runner();

        Thread t1 = new Thread(new Runnable(){
            public void run(){
                runner.firstThread();
            }
        });

        Thread t2 = new Thread(new Runnable(){
            public void run(){
                runner.secondThread();
            }
        });

        t1.start();
        t2.start();
    }
}

```

```

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        runner.finished();
    }
}

```

Output uden lock:

Count is: 13419

Output med lock:

Count is: 20000

ReentrantLock await(), signal() og signalAll()

ReentrantLock har desuden metoderne await(), signal() og signalAll(), som svarer til ethvert objekts wait(), notify() og notifyAll() og kan bruges i synchronized blokke.

En ReentrantLock som jo er et alternativ til synchronized blokke, kan således også afgive låsen med metoden await() og få den igen når låneren kalder dens signal() eller signalAll().

I det følgende er de to metoder firstThread() og secondThread() tilføjet dette.

Metoden firstThread() får låsen først, da dens tråd startes først og secondThread() begynder med en sleep(). Ved kald af await() friges låsen og der ventes her til secondThread() modtager et enter-key og kalder signal() på låsen. Så genoptager firstThread() låsen og fortsætter. Men først når secondThread() også har kaldt unLock().

Det er nødvendigt både at kalde signal() og unlock() for at metoden, der har kaldt await() både kan vågne op og få låsen igen.

```

package dk.tec;

public class ReentrantLocksTest
{
    public static void main(String[] args)
    {
        final Runner runner = new Runner();

        Thread t1 = new Thread(new Runnable()
        {
            public void run()
            {
                try {
                    runner.firstThread();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        Thread t2 = new Thread(new Runnable()
        {
            public void run()
            {
                try {
                    t1.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}

```

```

    {
        public void run()
        {
            try {
runner.secondThread();
            } catch (InterruptedException e) {
e.printStackTrace();
            }
        }
    });

t1.start();
t2.start();

try {
    t1.join();
    t2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

runner.finished();
}
}

```

```

package dk.tec;

import java.util.Scanner;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Runner
{
    private int count = 0;

    private Lock lock = new ReentrantLock();

    private Condition cond = lock.newCondition();

    private void increment()
    {
        for(int i = 0; i < 10000; i++ )
        {
            count++;
        }
    }

    public void firstThread() throws InterruptedException
    {
        lock.lock();
        cond.await();
        System.out.println("Woken up!");

        try{
            increment();
        }finally{
            lock.unlock();
        }
    }

    public void secondThread() throws InterruptedException
    {
        Thread.sleep(1000);
        lock.lock();

        System.out.println("Press return key!");
        new Scanner(System.in).nextLine();
    }
}

```

```
System.out.println("Got return key!");

cond.signal();

try{
    increment();
}finally{
    lock.unlock();
}

public void finished()
{
    System.out.println("Count is: " + count);
}
}
```

Output:

Press return key!

Got return key!

Woken up!

Count is: 20000

Dead Lock og tryLock()

For at demonstrere dead lock benyttes følgende klasse Account, hvor der kan indsættes, hæves og flyttes et beløb imellem to Account-objekter.

```
public class Account
{
    private int balance = 10000;

    public void deposit(int amount)
    {
        balance += amount;
    }

    public void withdraw(int amount)
    {
        balance -= amount;
    }

    public int getBalance()
    {
        return balance;
    }

    public static void transfer(Account acc1, Account acc2, int amount)
    {
        acc1.withdraw(amount);
        acc2.deposit(amount);
    }
}
```

Ideen er så at der arbejdes med to Account-objekter, acc1 og acc2. Når der skal gøres noget på acc1, skal der låses på lock1, og når der sker gøres noget på acc2, skal der låses på lock2.

Dette sker i følgende klasse Runner. Der er to metoder, der kører i hver sin tråd, som oprettes i main, som i forrige eksempel. (Dette er ikke vist her. Vi har set det så mange gange 😊)

Til sidst kaldes metoden finished() hvor balancen i acc1 og acc2 samt summen af dem udskrives. Summen skulle gerne give 20000, da de begynder med at have 10000 hver og der bare flyttes beløb imellem dem.

Den ene metode firstThread() kalder Accounts transfer() og overfører beløb fra acc1 til acc2 en masse gange.

Den anden metode secondThread() kalder også Accounts transfer og overfører beløb fra acc2 til acc1 en masse gange.

Hvis der ikke er nogen løsninger, går det galt som set tidligere, så summen ikke er 20000.

Hvis de begge låser først på lock1 og derefter på lock2 går det også godt. Så er det bare den første der vinder, fordi den anden hænger i lock1.lock().

Men hvis der låses i forskellige rækkefølge sker der en dead lock og hele programmet hænger.

```
public class Runner
{
    private Account acc1 = new Account();
    private Account acc2 = new Account();

    private Lock lock1 = new ReentrantLock();
    private Lock lock2 = new ReentrantLock();
```

```

public void firstThread() throws InterruptedException
{
    Random random = new Random();

    for(int i = 0; i < 10000; i++)
    {
        lock1.lock();
        lock2.lock();
        try{
            Account.transfer(acc1, acc2, random.nextInt(100));
        }finally{
            lock1.unlock();
            lock2.unlock();
        }
    }
}

public void secondThread() throws InterruptedException
{
    Random random = new Random();

    for(int i = 0; i < 10000; i++)
    {
        lock2.lock();
        lock1.lock();
        try{
            Account.transfer(acc2, acc1, random.nextInt(100));
        }finally{
            lock2.unlock();
            lock1.unlock();
        }
    }
}

public void finished()
{
    System.out.println("Account 1 balance: " + acc1.getBalance());
    System.out.println("Account 2 balance: " + acc2.getBalance());
    System.out.println("Total balance: " + (acc1.getBalance() + acc2.getBalance()));
}
}

```

Output hvis der låses i samme rækkefølge:

```

Account 1 balance: 12685
Account 2 balance: 7315
Total balance: 20000

```

Output hvis der låses i forskellig rækkefølge:

Ingenting – programmet hænger i en dead lock

For at undgå dead lock oprettes en metode acquireLocks(), som kaldes med de to låse som input, der ønskes låst på. Metoden sørger for at der først returneres når begge låse er opnået.

Dette gøres med metoden tryLock(), som ikke hænger, hvis låsen ikke er fri, og returnerer om det gik godt eller skidt. Hvis begge låse opnås returneres der fra metoden. Hvis en af låsene opnås, frigives denne igen og løkken gentages indtil begge låse opnås.

Koden

```

lock1.lock();
lock2.lock();

```

og

```
lock2.lock();
lock1.lock();
```

ændres ændres i begge metoder til

```
acquireLocks(lock1, lock2);
```

Rækkefølgen af parametrene er ligegyldig.

```
private void acquireLocks(Lock firstLock, Lock secondLock)
    throws InterruptedException
{
    while(true)
    {
        boolean gotFirstLock = false;
        boolean gotSecondLock = false;
        try{
            gotFirstLock = firstLock.tryLock();
            gotSecondLock = secondLock.tryLock();
        }finally{
            if(gotFirstLock && gotSecondLock)
            {
                return;
            }
            if(gotFirstLock)
            {
                firstLock.unlock();
            }
            if(gotSecondLock)
            {
                secondLock.unlock();
            }
        }
        Thread.sleep(1);
    }
}
```

Semaphore

En semafor er et objekt der minder om en lås. Den indeholder et antal tilladelser, permits, som bliver talt en ned når metoden acquire() bliver kaldt, og talt op når metoden release() bliver kaldt. Ved oprettelsen angives antallet af permits, og når dette er talt ned til 0 pga. et antal kald af acquire(), vil et kald af acquire() blokere og tråden tages ud af scheduling indtil der sker kald af release() så den kan få en permit.

En semafor med antal tilladelser på 1 vil virke som en lås, blot med den forskel at en lås skal låses og låses op i samme tråd. En semafors acquire() kan ske i en tråd og release() i en anden tråd. De hænger egentlig ikke sammen, da det kun er et spørgsmål til enhver tid, om det maksimale antal tilladelser er opnået.

Man kan undre sig over at det er muligt at øge antallet af tilladelser til flere end startværdien, ved at kalde release() flere gange end acquire(). Så det må man selv sørge for - at der er balance imellem disse.

```
Semaphore sem = new Semaphore(1); // 1

sem.acquire(); // 0
sem.release(); // 1
sem.release(); // 2
```

```
System.out.println("Available permits: " + sem.availablePermits());
```

For at demonstre en semafor benyttes følgende klasse, der simulerer oprettelse af connections.

Klassen er en såkaldt singleton, et af de klassiske mønstre inden for programmering. Dvs. at der altid er en og kun en instans af klassen. Dette opnås ved at have en static attribut, der holder en internt oprettet instans af klassen, lade constructoren være private, så der ikke kan oprettes instanser udefra, og oprette en metode getInstance(), der returnerer instansen.

Klassen simulerer at den kan oprette et antal connections, holde styr på antallet, og nedlægge hver connection igen efter udført arbejde.

Hvis disse oprettes i hver sin tråd, kan der være mange samtidig.

```
public class Connection
{
    private static Connection instance = new Connection();

    private int connections = 0;

    private Connection(){}

    public static Connection getInstance(){return instance;}

    public void connect()
    {
        synchronized(this)
        {
            connections++;
            System.out.println("Current connections: " + connections);
        }

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        synchronized(this)
        {
            connections--;
        }
    }
}
```

Der afprøves med følgende main(), hvor der oprettes 200 tråde, som hver udfører metoden connect().

```
public class ThreadsSemaphoreTest {  
    public static void main(String[] args) throws InterruptedException  
    {  
        ExecutorService executor = Executors.newCachedThreadPool();  
  
        for(int i = 0; i < 200; i++)  
        {  
            executor.submit(new Runnable(){  
                @Override  
                public void run()  
                {  
                    Connection.getInstance().connect();  
                }  
            });  
        }  
        executor.shutdown();  
        executor.awaitTermination(1, TimeUnit.DAYS); //max ventetid  
    }  
}
```

Output:

```
Current connections: 1  
Current connections: 2  
Current connections: 3  
...  
Current connections: 198  
Current connections: 199  
Current connections: 200
```

Hvis det nu ønskes at der skal være en begrænsning på f.eks. 10 connections ad gangen, benyttes en semafor. Klassen Connection ændres til følgende.

Der oprettes en semafor med 10 tilladelser. Argumentet true er parameteren fair, som sikrer at den, der har ventet længst på en acquire(), også er den der først får den, når den bliver ledigt.

Metoden connect() ændres til doConnect(), og der oprettes metoden connect(), som kalder doConnect(). Inden kaldet udføres semaforens acquire(), og bagefter udføres semaforens release(). Dette sker i en finally-blok, så en evt. exception i doConnect() ikke gør at release() ikke bliver udført.

```
public class Connection  
{  
    private static Connection instance = new Connection();  
  
    private int connections = 0;  
  
    private Semaphore sem = new Semaphore(10, true);  
  
    private Connection(){}
  
    public static Connection getInstance(){return instance;}
  
    public void connect()
    {
        try {
            sem.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
  
        try{
            doConnect();

```

```

        }finally{
            sem.release();
        }
    }

public void doConnect()
{
    synchronized(this)
    {
        connections++;
        System.out.println("Current connections: " + connections);
    }

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    synchronized(this)
    {
        connections--;
    }
}
}

```

Output:

```

Current connections: 1
Current connections: 2
Current connections: 3
Current connections: 4
Current connections: 5
Current connections: 6
Current connections: 7
Current connections: 8
Current connections: 9
Current connections: 10
Current connections: 9
Current connections: 10
...

```

Callable og Future

For at kunne returnere data fra kode der udføres i en tråd, benyttes interfacet Callable, som har metoden call(), i stedet for Runnable med metoden run().

Det er selvfølgelig muligt at lade koden i en tråd gemme data i en instans-variabel og så hente data fra denne.

Men med Callable er det muligt at få data som en returværdi fra tråden.

Det er desuden muligt at få exceptions ud til omgivelserne.

Først vises en hel almindelig Runnable der udskriver på skærmen, venter mellem 0 og 4 sekunder, udskriver igen og slutter.

```

public class ThreadCallableTest
{
    public static void main(String[] args)
    {
        ExecutorService executor = Executors.newCachedThreadPool();
        executor.submit(new Runnable()
        {
            @Override
            public void run()

```

```

    {
        Random random = new Random();
        int duration = random.nextInt(4000);

        System.out.println("Starting ...");

        try {
            Thread.sleep(duration);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Finished.");
    }
}); executor.shutdown();
}
}

```

Output:

```

Starting ...
Finished.

```

Så ændres koden til at returnere en værdi fra trådens kode.

Callable og Future er parameterized typer, og her ønskes der at returnere et heltal fra trådens kode.

Nu udskiftes Runnable med Callable<Integer> og void run() med Integer call().

Metoden executor.submit() returnerer typen Future. Her skal det så være Future<Integer>.

Metoden call() sættes til at returnere den random-tid der sleepes.

Når tråden er slut, vil værdien ligge i variablen future, og kan hentes med future.get().

Det er ikke nødvendigt at kalde metoden

```
executor.awaitTermination(1, TimeUnit.MINUTES);
```

da future.get() vil blokere og vente indtil tråden er afsluttet og værdien er til stede.

```

public class ThreadCallableTest
{
    public static void main(String[] args)
    {
        ExecutorService executor = Executors.newCachedThreadPool();

        Future<Integer> future = executor.submit(new Callable<Integer>()
        {
            @Override
            public Integer call()
            {
                Random random = new Random();
                int duration = random.nextInt(4000);

                System.out.println("Starting ...");

                try {
                    Thread.sleep(duration);
                } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }

    System.out.println("Finished.");

    return duration;
}
});

executor.shutdown();

try {
    System.out.println("Result is: " + future.get());
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
}
}
}

```

Output:

```

Starting ...
Finished.
Result is: 3364

```

Future giver også mulighed for at få en exception ud fra tråden via future.get().

Den kaster jo både InterruptedException og ExecutionException. Hvis der sker en exception inde i call(), vil denne uanset typen, komme ud gennem future.get() ExecutionException.

I det følgende er der tilføjet at der kastes en IOException hvis duration er over 2000. Den kommer ud som ExecutionException med en message "java.io.IOException: Sleeping for too long!", men den egentlige exception kan hentes med e.getCause() og typecastes til IOException, og så er message "Sleeping for too long!". (Jeg er i tvivl om hvornår future.get() skulle kaste en InterruptedException, da en sådan kastet også ser ud til at komme ud som ExecutionException.)

```

public class ThreadCallableTest
{
    public static void main(String[] args)
    {
        ExecutorService executor = Executors.newCachedThreadPool();
        Future<Integer> future = executor.submit(new Callable<Integer>()
        {
            @Override
            public Integer call() throws Exception
            {
                Random random = new Random();
                int duration = random.nextInt(4000);

                if(duration > 2000)
                {
                    throw new IOException("Sleeping for too long!");
                }

                System.out.println("Starting ...");

                try {
                    Thread.sleep(duration);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            System.out.println("Finished.");
        });
    }
}

```

```

        return duration;
    }
});

executor.shutdown();

try {
    System.out.println("Result is: " + future.get());
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    System.out.println(e.getMessage());

    IOException ex = (IOException)e.getCause();
    System.out.println(ex.getMessage());
}
}
}
}

```

Output:

```
java.io.IOException: Sleeping for too long!
Sleeping for too long!
```

Future-klassen har forskellige metoder, bl.a. cancel(), der svarer til interrupt() i Thread, som kan benyttes til at afbryde udførelsen af tråden.

Det ser ud til at Callable og Future kun kan bruges i forbindelse med thread-pools, hvor submit() returnerer et Future-objekt.

Interrupt af tråde

For at afslutte en tråd, vil man ofte benytte et flag, der sættes sådan at en uendelig løkke afslutter.

Det er også muligt at bruge Thread-klassens metode() interrupt(), som der skal ses på her i det følgende.

Først oprettes et program med en tråd der tager noget tid at udføre, her på nærværende PC 5-10 sekunder.

```

public class ThreadInterruptTest
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Starting!");

        Thread t = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                Random ran = new Random();
                for(int i = 0; i < 1E8; i++)
                {
                    Math.sin(ran.nextDouble());
                }
            }
        });
        t.start();
        t.join();
    }
}

```

```

        System.out.println("Finished!");
    }
}

```

Output:

```

Starting!
Finished!

```

Koden ændres sådan at efter tråden er startet ventes der en halv sekund og derefter kaldes trådens interrupt(). Men dette ændrer ikke noget. Det tager stadig den samme tid at udføre koden. For at udløse et interrupt inde i tråden, kræves der en blokerende metode, der kaster en InterruptedException. Dette kunne være metoden Thread.sleep().

Så inde i løkken tilføjes Thread.sleep() med tilhørende try/catch.

Selv om sleep(1) kun skulle være 1 millisekund er der så meget overhead på kald af metoden, at løkken nu tager alt for lang tid, så den sættes ned fra, i dette tilfælde 1E8 til 1E4. Så tager det nogle sekunder.

Men der sker stadig ikke andet end at der udskrives meddelelsen i catch, sådan at det kan ses, at der kommer en InterruptedException. Man bestemmer altså selv hvad der skal ske når en tråd interruptes. Så først ved tilføjelse af kommandoen break i catch, så løkken afbrydes, afslutter tråden nu hurtigt.

```

public class ThreadInterruptTest
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Starting!");

        Thread t = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                Random ran = new Random();
                for(int i = 0; i < 1E4; i++)
                {
                    try {
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                        System.out.println("Thread has been interrupted!");
                        break;
                    }
                    Math.sin(ran.nextDouble());
                }
            }
        });
        t.start();
        Thread.sleep(500);

        t.interrupt();

        t.join();

        System.out.println("Finished!");
    }
}

```

Output:

```
Starting!
Thread has been interrupted!
Finished!
```

Det der i virkligheden sker, når metoden interrupt() kaldes på en tråd, er at der sættes et interrupted-flag i tråden.

Interrupted-flaget resettes når InterruptedException kastes.

Der kan spørges på interrupted-flaget med

```
Thread.interrupted();
```

som også resetter flaget.

Eller der kan spørges på interrupted-flaget med

```
Thread.currentThread().isInterrupted()
```

som ikke resetter flaget.

Følgende er vist koden med sleep() og try udskiftet med en if-sætning. (Antal løkke-gennemløb, måtte sættes op igen efter fjernelse af sleep().)

```
public class ThreadInterruptTest
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Starting!");

        Thread t = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                Random ran = new Random();
                for(int i = 0; i < 1E8; i++)
                {
                    if(Thread.currentThread().isInterrupted())
                    {
                        System.out.println("Thread has been interrupted!");
                        break;
                    }

                    Math.sin(ran.nextDouble());
                }
            }
        });

        t.start();

        Thread.sleep(500);
        t.interrupt();
        t.join();

        System.out.println("Finished!");
    }
}
```

Output

```
Starting!
Thread has been interrupted!
```

Finished!

Thread pool

Følgende kode gør det samme som forrige eksempel, men bruger ThreadPool i stedet for en Thread. Det viser hvordan Future.cancel() kan interrupte en tråd. Hvis den kaldes med false som argument, aflyses en ikke startet tråd kun. Med true aflyses en ikke startet tråd, og interrupthes hvis den er kørende.

Hvis der er startet mange tråde i poolen, må deres Future-objekter være gemt i en liste, så kan man interrupte de enkelte tråde.

Med exec.shutdownNow();
Aflyses ikke kørende tråde og interrupthes alle tråde kørende tråde i poolen.

```
public class ThreadPoolFutureCancelTest
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Starting!");

        ExecutorService exec = Executors.newCachedThreadPool();
        Future<?> fu = exec.submit(new Callable<Void>()
        {
            @Override
            public void call()
            {
                Random ran = new Random();
                for(int i = 0; i < 1E8; i++)
                {
                    if(Thread.currentThread().isInterrupted())
                    {
                        System.out.println("Thread has been interrupted!");
                        break;
                    }
                    Math.sin(ran.nextDouble());
                }
                return null;
            }
        });
        exec.shutdown();
        Thread.sleep(500);

        fu.cancel(true);
        //Eller
        exec.shutdownNow();

        exec.awaitTermination(1, TimeUnit.DAYS);
        System.out.println("Finished!");
    }
}
```

Output:

```
Starting!
Thread has been interrupted!
Finished!
```

Opdatering af GUI fra en anden tråd

Da GUI ikke er trådsikker, bør man ikke opdatere den fra en anden tråd end GUI-tråden/hovedtråden.

Klassen SwingWorker er beregnet til at løse dette problem.

I det følgende program er der en brugerflade, hvor knappen Start opretter en ny tråd, som gennemløber en løkke der tæller, og som udskriver tællerværdien løbende i den øverste label, og når tråden slutter, returneres en værdi, som en tekst, der skrives i den anden label.

Klassen SwingWorker har metoden doInBackground(), som udføres i en ny tråd når metoden execute() kaldes.

Inde i doInBackground() kan metoden publish() kaldes, hvilket vil medføre at metoden process() kaldes og udføres i GUI-tråden.

Når doInBackground() afslutter, vil metoden done() kaldes og udføres i GUI-tråden.

Der angives to typer i oprettelsen af SwingWorker-objektet.

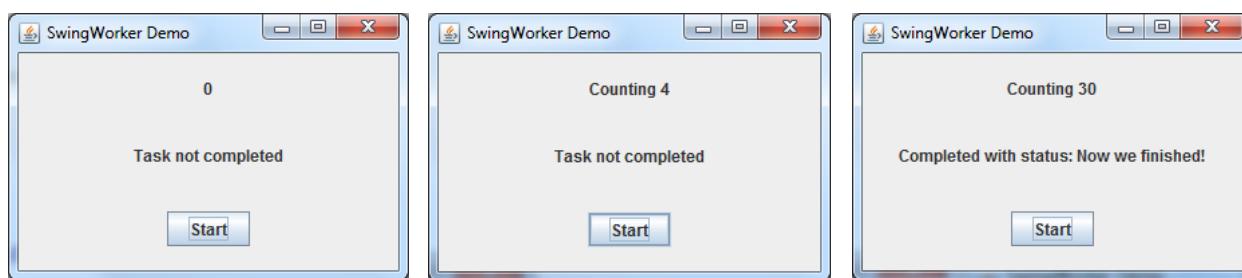
Den første type, her String, er den type doInBackground() returnerer og som kan hentes i done() ved at kalde get(). Den anden type, her Integer, er den type, der kan gives til publish() som parameter, og hentes i process() som en liste. Grunden til listen er, at der ikke kan garanteres at process() kaldes hver gang publish() kaldes.

```
SwingWorker<String, Integer> worker = new SwingWorker<String, Integer>()
{
    • doInBackground()
        ○ publish(i);
        ○ return "Now we finished!";

    • process(List<Integer> chuncks)

    • done()
        ○ get()

    • worker.execute();
}
```



```
public class Program {

    public static void main(String[] args)
    {
        MyFrame mf = new MyFrame("My Frame");
        mf.setVisible(true);
    }
}
```

```

package dk.tec.jaj;

import java.awt.EventQueue;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.List;
import java.util.concurrent.ExecutionException;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
import javax.swing.SwingWorker;

public class MyFrame extends JFrame
{
    private JLabel lblCount = new JLabel("0");
    private JLabel lblStatus = new JLabel("Task not completed");
    private JButton btnStart = new JButton("Start");

    public MyFrame(String title)
    {
        super(title);

        Font font = new Font(lblStatus.getFont().getName(),
                             Font.PLAIN, lblStatus.getFont().getSize() * 2);
        lblStatus.setFont(font);
        lblCount.setFont(font);
        btnStart.setFont(font);

        setLayout(new GridBagLayout());
        GridBagConstraints gc = new GridBagConstraints();
        gc.fill = GridBagConstraints.NONE;

        gc.gridx = 0; gc.gridy = 0;
        gc.weightx = 1; gc.weighty = 1;
        add(lblCount, gc);

        gc.gridx = 0; gc.gridy = 1;
        gc.weightx = 1; gc.weighty = 1;
        add(lblStatus, gc);

        gc.gridx = 0; gc.gridy = 2;
        gc.weightx = 1; gc.weighty = 1;
        add(btnStart, gc);

        btnStart.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                start();
            }
        });
    }

    setBounds(700, 200, 300, 200);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
}

protected void start()
{
    SwingWorker<String, Integer> worker = new SwingWorker<String, Integer>()
    {
        @Override
        protected String doInBackground() throws Exception
        {
            for(int i = 1; i <= 30; i++)

```

```

    {
        Thread.sleep(100);
        System.out.println("Hello " + i);

        publish(i);
    }
    return "Now we finished!";
}

@Override
protected void process(List<Integer> chunks)
{
    Integer value = chuncks.get(chuncks.size() - 1);
    lblCount.setText("Counting " + value);
}

@Override
protected void done()
{
    try {
        lblStatus.setText("Completed with status: " + get().toString());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
};

// Start det hele
worker.execute();
}
}

```

Andre måder at tilgå GUI fra tråde

Inde i en anden tråd end GUI-tråden, kan der også udføres noget på GUI ved at oprette en ny tråd med EventQueue.invokeLater() eller SwingUtilities.invokeLater().

```

EventQueue.invokeLater(new Runnable()
{
    @Override
    public void run()
    {
        // Udfør GUI ting.
    }
});

```

```

SwingUtilities.invokeLater(new Runnable()
{
    @Override
    public void run()
    {
        // Udfør GUI ting.
    }
});

```

Join()

Metoden join() får den tråd der kalder metoden til at vente på at den tråd, metoden kaldes på afsluttes.

Dvs at der udføres ikke videre før den anden tråd er afsluttet.

t.join() gør at der ventes på at tråden t afsluttes.

t.join(1000) gør at der ventes på at tråden t afsluttes, dog højest et sekund. Dvs. at den aktuelle tråd udføres videre når der er gået 1 sekund. Hvis t afsluttes inden der er gået 1 sekund udføres den aktuelle tråd videre allerede da.

7: Eleven kan fremstille programmer, der anvender simpel TCP/IP-klient til kommunikation gennem Sockets.

Simpel klient/server

Simpel klient/server, hvor serveren returnerer, det en klient sender og begge lukker derefter ned. Der sendes kun en gang fra klienten og returneres kun en gang fra serveren.

```
cmd Kommandoprompt - java SocketServerMain
C:\Java Test\Server>dir
Disken i drev C er OSDisk
Diskens serienummer er 3ACB-7FEC
Indhold af C:\Java Test\Server
19-05-2014 16:43    <DIR>    .
19-05-2014 16:43    <DIR>    ..
19-05-2014 16:38           1.899 SocketServerMain.class
    1 fil(er)      1.899 byte
    2 mappe(r)   195.291.926.528 byte ledig
C:\Java Test\Server>java SocketServerMain
Server Started!
```

```
cmd C:\Windows\system32\cmd.exe
C:\Java Test\Client>dir
Disken i drev C er OSDisk
Diskens serienummer er 3ACB-7FEC
Indhold af C:\Java Test\Client
19-05-2014 16:43    <DIR>    .
19-05-2014 16:43    <DIR>    ..
19-05-2014 11:18           1.613 SocketClientMain.class
    1 fil(er)      1.613 byte
    2 mappe(r)   195.291.582.464 byte ledig
C:\Java Test\Client>java SocketClientMain
Client wrote: Hej fra klienten!
C:\Java Test\Client>
```

```
cmd Kommandoprompt
C:\Java Test\Server>dir
Disken i drev C er OSDisk
Diskens serienummer er 3ACB-7FEC
Indhold af C:\Java Test\Server
19-05-2014 16:43    <DIR>    .
19-05-2014 16:43    <DIR>    ..
19-05-2014 16:38           1.899 SocketServerMain.class
    1 fil(er)      1.899 byte
    2 mappe(r)   195.291.926.528 byte ledig
C:\Java Test\Server>java SocketServerMain
Server Started!
Server Ended!
C:\Java Test\Server>-
```

Server

```
public class SocketServerMain
{
    public static void main(String[] args)
    {
        System.out.println("Server Started!");

        try (ServerSocket serverSocket = new ServerSocket(2000);
             Socket ioSock = serverSocket.accept())
        {
            BufferedReader in =
                new BufferedReader(
                    new InputStreamReader(ioSock.getInputStream()));
            PrintWriter out = new PrintWriter(ioSock.getOutputStream(), true);

            String inString = in.readLine();
            out.println("Client wrote: " + inString);

        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

        System.out.println("Server Ended!");
    }
}
```

Client

```
public class SocketClientMain
{
    public static void main(String[] args)
    {
        try (Socket sock = new Socket("localhost", 2000))
        {
            InputStream in = sock.getInputStream();
            OutputStream out = sock.getOutputStream();

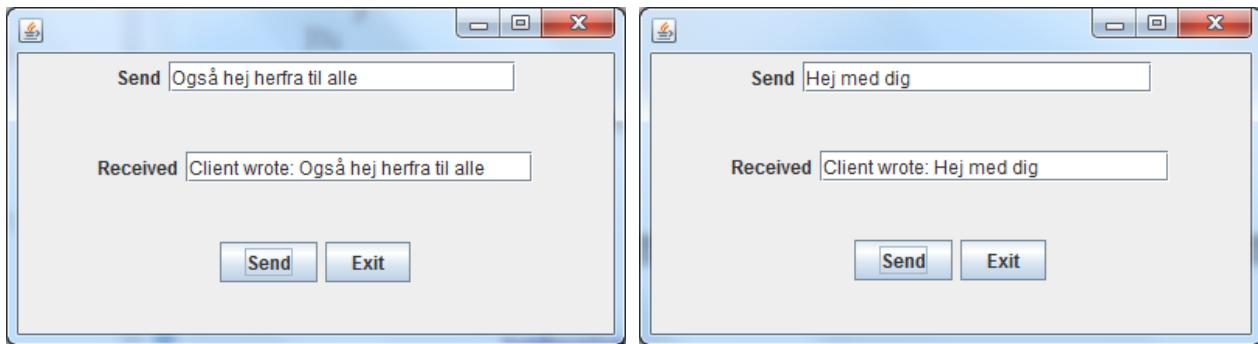
            String str = "Hej fra klienten!\n";
            byte[] bytes = str.getBytes();
            out.write(bytes);

            int ch;
            while((ch = in.read()) != -1)
            {
                System.out.print((char)ch);
            }

        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Multitrådet server med flere klienter

Serveren starter en ny tråd for hver klient, der henvender sig, og bliver ved med at returnere det, den aktuelle klient sender, tilbage til denne.



```
public class ServerMain
{
    public static void main(String[] args)
    {
        System.out.println("Server Started!");

        try (ServerSocket serverSocket = new ServerSocket(2000))
        {
            while(true)
            {
                ClientWorker w;
                w = new ClientWorker(serverSocket.accept());
                Thread t = new Thread(w);
                t.start();
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Server Ended!");
    }
}
```

```
class ClientWorker implements Runnable
{
    private Socket client;

    ClientWorker(Socket client)
    {
        this.client = client;
    }

    public void run()
    {
        String line;
        BufferedReader in = null;
        PrintWriter out = null;
        try
        {
            in = new BufferedReader(new InputStreamReader(client.getInputStream()));
            out = new PrintWriter(client.getOutputStream(), true);

        } catch (IOException e) {
            System.out.println("IO failed: " + e.getMessage());
            System.exit(-1);
        }

        while(true)
        {
            try
            {
```

```

        line = in.readLine(); // + (char)10;
        out.println("Client wrote: " + line);
    }catch (IOException e) {
        System.out.println("Read failed");
        System.exit(-1);
    }
}
}
}

```

Client

```

public class ClientMain extends JFrame
{
    JLabel lblSend, lblReceived;
    JTextField txtSend, txtReceived;
    JButton btnSend, btnExit;

    Socket sock;
    BufferedReader in;
    PrintWriter out;

    public static void main(String[] args)
    {
        ClientMain client = new ClientMain();
        client.setBounds(200, 200, 400, 300);
        client.setDefaultCloseOperation(EXIT_ON_CLOSE);
        client.setVisible(true);
    }

    public ClientMain()
    {
        lblSend = new JLabel("Send");
        lblReceived = new JLabel("Received");
        txtSend = new JTextField(20);
        txtReceived = new JTextField(20);
        btnSend = new JButton("Send");
        btnExit = new JButton("Exit");

        JPanel pnl1 = new JPanel();
        JPanel pnl2 = new JPanel();
        JPanel pnl3 = new JPanel();

        pnl1.add(lblSend);
        pnl1.add(txtSend);
        pnl2.add(lblReceived);
        pnl2.add(txtReceived);
        pnl3.add(btnSend);
        pnl3.add(btnExit);

        getContentPane().setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));
        add(pnl1);
        add(pnl2);
        add(pnl3);

        try {
            sock = new Socket("localhost", 2000);
            in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
            out = new PrintWriter(sock.getOutputStream(), true);
        } catch (IOException e) {
            e.printStackTrace();
        }

        btnSend.addActionListener(new ActionListener()
        {
            @Override
            public void actionPerformed(ActionEvent arg0)
            {

```

```
try {
    out.println(txtSend.getText());
    String strReceived = in.readLine();
    txtReceived.setText(strReceived);
} catch (IOException e) {
    System.out.println(e.getMessage());
}
});
```

Client-Server Socket Opgave 1

Lav et klient/server-system, hvor klienter kan henvende sig til serveren og skrive noget, som sendes tilbage til den selv og alle andre klienter, der har tilsluttet sig.

8: Eleven kan anvende sproget til udvikling af klient/server system i Java.

9: Eleven kan oprette fjernobjekter ved at bruge Java RMI.

(Eksemplet er fra <http://javabog.dk/OOP/kapitel19.jsp>)

For at udbyde metoder fra en fjern maskine, skal disse defineres i et interface der arver interfacet Remote. Den er tom, og giver derfor kun en type.

The Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends java.rmi.Remote are available remotely.

Interfacet Kontol arver Remote og definerer 3 metoder til at servicere en bankkonto.

```
import java.util.ArrayList;

public interface KontoI extends java.rmi.Remote
{
    public void overførsel(int kroner) throws java.rmi.RemoteException;
    public int saldo()                throws java.rmi.RemoteException;
    public ArrayList bevægelser()     throws java.rmi.RemoteException;
}
```

Klassen KontoImpl implementerer interfacet Kontol og dermed dennes metoder.

```
import java.util.ArrayList;
import java.rmi.server.UnicastRemoteObject;

public class KontoImpl extends UnicastRemoteObject implements KontoI
{
    public int saldo;
    public ArrayList bevægelser;

    public KontoImpl() throws java.rmi.RemoteException
    {
        // man starter med 100 kroner
        saldo = 100;
        bevægelser = new ArrayList();
    }

    public void overførsel(int kroner)
    {
        saldo = saldo + kroner;
        String s = "Overførsel på "+kroner+" kr. Ny saldo er "+saldo+" kr.";
        bevægelser.add(s);
        System.out.println(s);
    }

    public int saldo()
    {
        System.out.println("Der spørges om saldoen. Den er "+saldo+" kr.");
        return saldo;
    }
}
```

```

public ArrayList bevægelser()
{
    System.out.println("Der spørges på alle bevægelser.");
    return bevægelser;
}
}

```

KontoServer er programmet hvor main() opretter et objekt af KontoImpl og udbyder det ved at starte RMI på port 1099.

```

import java.rmi.Naming;

public class KontoServer
{
    public static void main(String[] arg) throws Exception
    {
        // Enten: Kør programmet 'rmiregistry' fra mappen med .class-filerne, eller:
        java.rmi.registry.LocateRegistry.createRegistry(1099); // start i server-JVM

        KontoI k = new KontoImpl();
        Naming.rebind("rmi://localhost/kontotjeneste", k);
        System.out.println("Kontotjeneste registreret.");
    }
}

```

Klientprogrammet får interfacet Konto fra udbyderen og kalder metoderne.

```

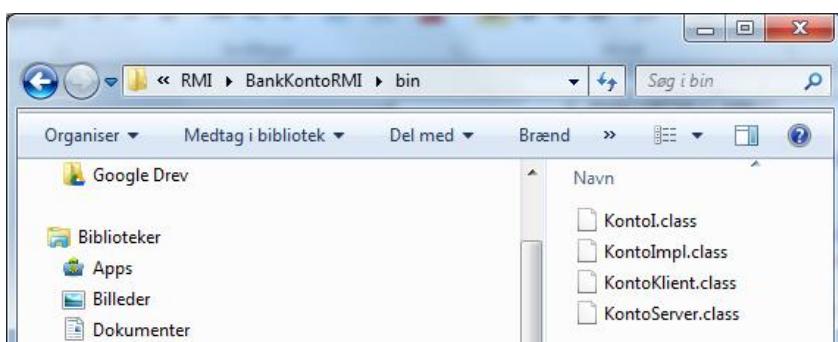
import java.rmi.Naming;

public class KontoKlient
{
    public static void main(String[] arg) throws Exception
    {
        KontoI k =(KontoI) Naming.lookup("rmi://localhost/kontotjeneste");
        k.overførsel(100);
        k.overførsel(50);
        System.out.println( "Saldo er: "+ k saldo() );
        k.overførsel(-200);
        k.overførsel(51);
        System.out.println( "Saldo er: "+ k saldo() );
        java.util.ArrayList bevægelser = k.bevægelser();

        System.out.println( "Bevægelser er: "+ bevægelser );
    }
}

```

Her ligger så de 4 compilerede class-filer.



Åben en CMD-prompt i mappen med klasserne og start hovedprogrammet KontoServer.

```
C:\BankKontoRMI\bin>java KontoServer  
Kontotjeneste registreret.
```

Hvis der kommer en fejlmeddeelse om at porten 1099 er i brug, findes processen og slettes ved at gøre følgende.

```
C:\BankKontoRMI\bin>netstat -ano
```

Aktive forbindelser

Proto	Lokal adresse	Fjernadresse	Tilstand	PID
TCP	0.0.0.0:80	0.0.0.0:0	LISTENING	4
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	468
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	4
TCP	0.0.0.0:1099	0.0.0.0:0	LISTENING	1708
TCP	0.0.0.0:1433	0.0.0.0:0	LISTENING	3152
TCP	0.0.0.0:2383	0.0.0.0:0	LISTENING	3276

```
C:\ BankKontoRMI \bin>taskkill -pid 1708 /f  
LYKKEDES: Processen med PID 1708 er afsluttet.
```

```
C:\ BankKontoRMI\bin>java KontoServer
```

Kontotjeneste registreret.

<<Bliver hængende her indtil klienten startes>>

Overførsel på 100 kr. Ny saldo er 200 kr.

Overførsel på 50 kr. Ny saldo er 250 kr.

Der spørges om saldoen. Den er 250 kr.

Overførsel på -200 kr. Ny saldo er 50 kr.

Overførsel på 51 kr. Ny saldo er 101 kr.

Der spørges om saldoen. Den er 101 kr.

Der spørges på alle bevægelser.

Klienten startes.

```
C:\ BankKontoRMI \bin>java KontoKlient
```

Saldo er: 250

Saldo er: 101

Bevægelser:

Overførsel på 100 kr. Ny saldo er 200 kr.

Overførsel på 50 kr. Ny saldo er 250 kr.

Overførsel på -200 kr. Ny saldo er 50 kr.

Overførsel på 51 kr. Ny saldo er 101 kr.

Noter