

User Interfaces in Java

Mikael Svahnberg*

2024-02-20

1 Introduktion

- Användargränssnitt i Java
 - textbaserade gränssnitt
 - Grafiska gränssnitt med Java AWT/Swing
- Enkel filhantering i Java
- Felhantering, `try-catch`, felrapportering

2 Gränssnitt

- *Gränssnitt* är en något överanvänd term
 - Komponenter erbjuder *programmeringsgränssnitt* som man använder när man skriver kod
 - API** Application Programming Interface
 - Klasser erbjuder ett *publikt gränssnitt*
 - * De publika metoder och attribut som man kan använda på objekt
 - Paket erbjuder också ett *publikt gränssnitt*
 - * De publika klasser som man kan använda sig av utifrån paketet
 - Servrar erbjuder ett *gränssnitt*, ett API
 - * De anrop man kan göra mot servern
 - * Vanligt idag är ett *REST-API*, som man ofta gör över *HTTP*

3 Användargränssnitt

- *Användargränssnitt* är det sätt som en *användare* kommer åt ett system
 - UI** User Interface
 - Parametrar på kommandoraden
 - Utskrifter i ett terminalfönster

*Mikael.Svahnberg@bth.se

- In- och Ut-matning i ett terminalfönster
 - Läs och Skriv till filer (är det till för användaren?)
- TUI Text-based User Interface
- * “Fullständigt” användargränssnitt, fast bara med text, och tillgängligt i en terminal
- GUI Graphical User Interface
- * “Fönster och knappar”

4 Exempel på Textbaserade gränssnitt

- De flesta Unix-kommandon (ex `ls -al` , `curl https://www.zombo.com`) tar parametrar och skriver ut en massa text
- Några startar en interaktiv miljö (ex *GNU debugger* `gdb`)
- Några skapar ett textbaserat UI, (ex filhanteraren *Midnight Commander* `mc`, editorn `vi` , eller `emacs -nw`)

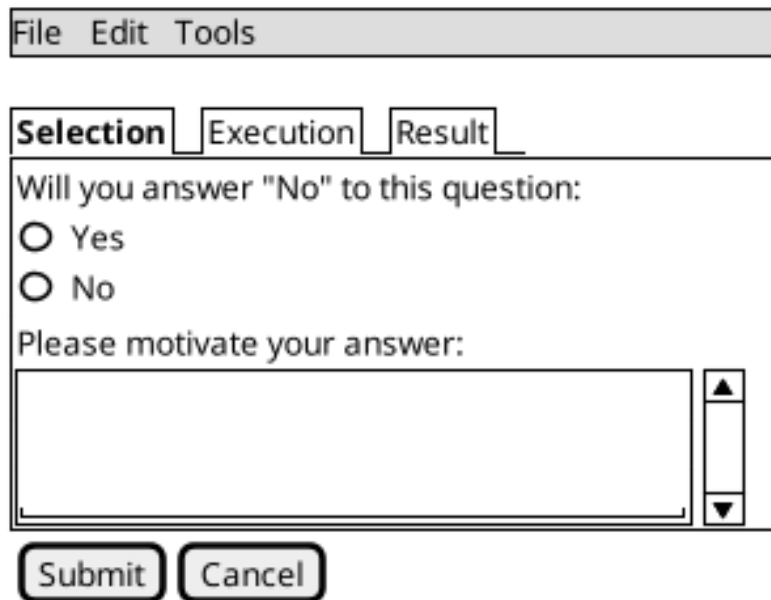
Fördelar:

- De fungerar alltid
- De fungerar även när du kör dem på en annan maskin (e.g. via `ssh`)
- De fungerar även när Windows-miljön – av olika anledningar – kanske inte kan starta.

Nackdelar:

- Svårare att “upptäcka” hur de fungerar
- Kräver lite mer tanke för att konstruera för att de skall bli enkla
- Begränsat med information som får plats på 80x25 tecken

5 Grafiska Användargränssnitt



- Skrivbordsapplikationer
 - Qt
 - Gnome
 - ...
 - Java Swing/Java AWT
- Webapplikationer

6 Att Kunna Byta Gränssnitt

- En bra grundprincip är att man skall bygga ett system så att man kan *byta gränssnitt*
- Separera *logik* från *Presentation*
- Model -- View
- I praktiken byter man sällan gränssnitt, men det ger också en tydligare design och är lättare att underhålla.
- Ibland använder sig ett “vackrare” gränssnitt av ett “enklare” under, en s.k. *porcelain*
 - (Linus Torvalds är, som så ofta, ursprunget till denna analogi: Man sitter inte direkt på avloppsröret)
 - Din utvecklingsmiljö kan mycket väl använda `gdb` internt
 - Den använder också nästan säkert kommandorads- `git` undertill

7 In- och utmatning i Java

- Klassen `System` är alltid tillgänglig
- I synnerhet `System.out.print()` och `System.out.println()`
- `System.out` är en `PrintStream` som normalt hänvisar till standard output (terminalen)
- Läs mer om `PrintStream` i java-dokumentationen:
 - <https://docs.oracle.com/en/java/javase/20/docs/api/index.html>
 - <https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/io/PrintStream.html>
- `System.in` är en `InputStream`, men den är svårare att använda
- Använd hellre `Console`, som du får tag i genom att anropa `System.console()`.

```
import java.io.*;
Console con = System.console();

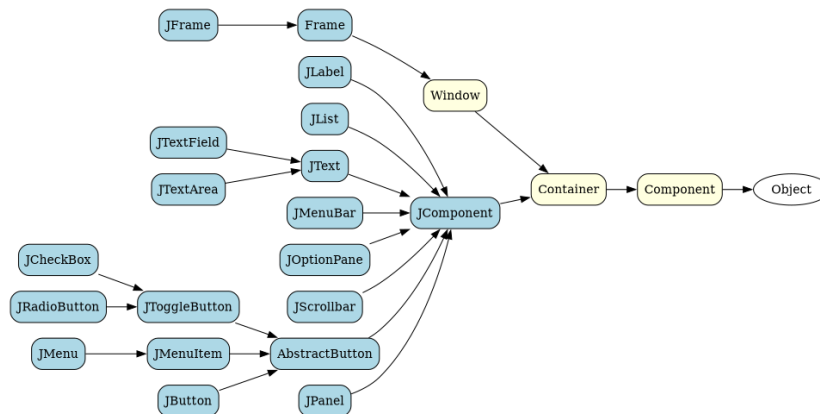
System.out.print("Enter your name: ");
String name = con.readLine();
System.out.println("Hello " + name);

// Or, shorter
name = con.readLine("Please enter your name again: ");
System.out.println("Hello again, " + name);

// We can complicate things
String lastname = con.readLine("What is your lastname, %s? ", name);
con.printf("Greetings, %s %s!\n", name, lastname);
```

8 Grafiska Gränssnitt i Java

- Två generationer: `java.awt` och `java.Swing`
 - AWT är äldre, men byggt specifikt för varje platform/operativsystem
 - Swing är skrivet i och för Java
 - De flesta klasser som används börjar med bokstaven “J”: `JFrame`, `JPanel`, `JLabel`, ...
 - De flesta finns också i flera varianter, t.ex. dekorerade med scrollbars.



9 JFrame, JPanel, och LayoutManagers

```
JFrame window = new JFrame("JavaPonies");
window.setSize(width, height);
```

```
JPanel canvas = new JPanel();
window.getContentPane().add(canvas);
canvas.setLayout(new BorderLayout());
canvas.add("North", /* ... */);
```

- JFrame för att skapa ett fönster
 - Stoppa in en JPanel som innehåller allt GUI
 - Lägg till Menyer osv.
- Kan ha godtyckligt många JPanel; en JPanel kan i sin tur också innehålla flera JPanel.
- Varje Container kan ha en LayoutManager som styr hur man vill lägga in komponenter i den.

BorderLayout – North, South, East, West, Center

- BorderLayout.PAGE_START, PAGE_END, LINE_START, LINE_END, CENTER

BoxLayout En rad eller en kolumn

- new BoxLayout(thePanel, BoxLayout.PAGE_AXIS) eller BoxLayout.LINE_AXIS

CardLayout olika "kort", där ett i taget är synligt.

FlowLayout en lång rad med komponenter

GridLayout en MxN matris av "hål" att stoppa komponenter i.

GridBagLayout Mer komplicerad GridLayout

SpringLayout Låter begränsningar för varje komponent styra hur de läggs ut.

10 En egen Panel

- Man kan ärva och utöka alla Swing-komponenter.
- Mest användbart är det att ärva från `JPanel` och bygga sin egen komponent
 - Till exempel om man vill ha ett "kort" på en Ponny som man kan använda i ett fönster, se `view.PonyCard`
 - Vill man ha ett helt eget fönster så kan man ärva och utöka från `JFrame`, se t.ex. klassen `view.PonyWindow`

```
private class Banner extends JPanel {
    public Banner() {
        super();          // Se till att all JPanel-initiering görs.
        setPreferredSize(new Dimension(width, 50));
    }

    public void paintComponent(Graphics g) {
        g.setColor(Color.black);
        g.setFont(new Font("SansSerif", Font.BOLD, 20));
        g.drawString("Java Ponies", 20, 40);
        /* ... */
    }
}
```

11 paint()

- Vid behov anropas `repaint()`
 - I sin tur anropar denna `paint()`
 - * ... som anropar `paintComponent()`, `paintBorder()`, och `paintChildren()`
- `paint()` kan överlagras för att rita sin egen komponent.

```
public void paint(Graphics g) {
    super.paint(g);        // Om man vill att alla dekorationer och underkomponenter skall rita

    // Graphics (eller Graphics2D) har en massa metoder för att rita former och text
    // läs gärna mer i dokumentationen om dessa.
    g.drawImage(myImage, 0, 50, getBackground(), this);
    g.setColor(Color.black);
    g.drawRoundRect(0,0,width-1, height-1, 5, 5);
    g.drawString(myPony.getName(), 5, 20);

    g.dispose()           // Var snäll och "släpp" Graphics-objektet efter dig, så vet Java a
}
```

12 EventListerners

- En `Component` kan skapa olika typer av *Händelser* (Events), till exempel:
 - Focus
 - Key
 - Mouse
 - MouseMotion
 - Action
- Till var och en av dessa händelser kan man koppla en *EventListener*
 - `FocusListener`, `KeyListener`, `MouseListener`, `ActionListener`, osv.
 - `addMouseListener(...)`, `addActionListener(...)`, ...
 - En *EventListener* har ett antal metoder som anropas när händelsen inträffar
 - Varje metod i *EventListnern* tar en parameter, som representerar händelsen,
 - * e.g. `ActionEvent`, `MouseEvent`, osv.
- `ActionListener`, `MouseListener`, osv. är *interface*, du måste skriva implementationen för dem.
 - Ibland bara en enda metod, ibland en handfull.
 - Det blir otydligt att skriva en massa tomma metoder för de händelser du *inte* är intresserad av.
 - Numera finns färdiga implementationer *xxxAdapter* som du kan utöka med bara de beteenden du vill.
 - * (Inkonsekvent nog finns bara *xxxAdapter* för de Listeners som har mer än en metod)

13 EventListeners och Klasser

Tänk dig följande scenario:

- Du har ett gränssnitt, med två knappar i.
- Du vill (såklart) veta när man trycker på en knapp, och helst vilken knapp man trycker på.

Olika lösningar:

1. Skriv en separat klass som **implements** `ActionListener`
 - Du kollar noga på det `ActionEvent` du får som parameter i `actionPerformed()` - metoden.
 - Den här klassen måste alltså veta exakt vad alla knappar heter.
 - `ActionListener` är enkel, den har bara en metod; titta på t.ex. `MouseListener` med fem metoder.

- Ditt paket får minst en sådan här klass som bara finns för att ta emot en enda händelse.
 - Jobbigare att läsa; din GUI-klass har inte längre hela ansvaret.
2. Skriv en *nästlad och privat klass* som **implements** `ActionListener` på samma sätt
 - Inga fler separata klasser; den här koden kommer skrivas som en del av din GUI-klass.
 - I övrigt samma som (1)
 3. Använd `ActionListener` som bas och överlagra metoden med lite modifiera genvägar:

```

JButton b1 = new JButton("Confirm");
JButton b2 = new JButton("Cancel");
JTextField someText = new JTextField();

b1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        // Read data from window (all local variables are available)
        someText.getText();
    } });

b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        someText.setText("");
    } });

// MouseListener har fler metoder, alltså finns det en MouseAdapter också...
myCanvas.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent evt) {
        if (null == myWindow) {
            myWindow = new PonyWindow(myPony);
            myWindow.setVisible(true);
        } else {
            myWindow.toggleVisible();
        }
    }
});

```

14 File Output in Java

- Use `java.io.FileWriter`
- There are many other ways, e.g. `java.io.PrintWriter`
- Careful about the character set, åäö may trip you up.

Basic pattern:

- try { 1. Open, 2. Write, 3. Close } catch(IOException e) {}
- What if we fail while writing?
- What happens with close?

Preferred pattern try-with

- try (resources to use) { } catch(...) { }
- Calls close() for you even if there is an exception.

```
import java.io.FileWriter;
import java.io.PrintWriter;

public class Outputter {
    public void basicPattern(String filename) {
        try {
            FileWriter fw = new FileWriter(filename);
            fw.write("Some text\n");
            fw.append("Some more text\n");
            fw.close();
        } catch(IOException e) {
            System.err.println("Error writing file " + filename);
            e.printStackTrace();
        }
    }

    public void preferredPattern(String filename) {
        try(FileWriter fw = new FileWriter(filename);
            BufferedWriter buf = new BufferedWriter(fw); ) {
            buf.write("Some preferred text");
            buf.newLine();
            buf.write("Some more preferred text");
        } catch (IOException e) {
            System.err.println("Error writing file " + filename);
            e.printStackTrace();
        }
    }

    public void otherWriter(String filename) {
        try ( PrintWriter out = new PrintWriter(filename) ) {
            out.println("Some more more text");
            out.println("and yet some more");
        } catch(IOException e) {
            System.err.println("Error writing file " + filename);
            e.printStackTrace();
        }
    }

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

```

        Outputter o = new Outputter();

        o.basicPattern("tst.txt");
        o.preferredPattern("tst2.txt");
        o.otherWriter("tst3.txt");
    }
}

```

15 File Input in Java

- `java.io.FileReader` ?
 - Only has one method to read a single character.
 - `java.io.BufferedReader` to get `readLine()` method
- Two ways here too. One old-school and one newer.
- As usual, there are many other ways too.

```

import java.io.FileReader;
import java.io.BufferedReader;
import java.nio.file.*;
import java.util.ArrayList;

public class Inputter {

    public ArrayList<String> oldSchool(String filename) {
        ArrayList<String> contents = new ArrayList<String>();
        try (
            FileReader file = new FileReader(filename);
            BufferedReader buf = new BufferedReader(file);
        ) {
            String line;
            while ((line = buf.readLine()) != null) {
                contents.add(line);
            }
        } catch (IOException e) {
            System.out.println(e);
        }

        return contents;
    }

    public ArrayList<String> hipster(String filename) {
        ArrayList<String> contents = new ArrayList<String>();
        Path path = Paths.get(filename);

        try(BufferedReader buf = Files.newBufferedReader(path)) {
            String line;

```

```

        while ((line = buf.readLine()) != null) {
            contents.add(line);
        }
    } catch (IOException e) {
        System.out.println(e);
    }

    return contents;
}

public static void main(String [] args) {
    Inputter in = new Inputter();
    ArrayList<String> contents;

    contents = in.oldSchool("tst.txt");
    for (String l : contents) { System.out.println(l); }

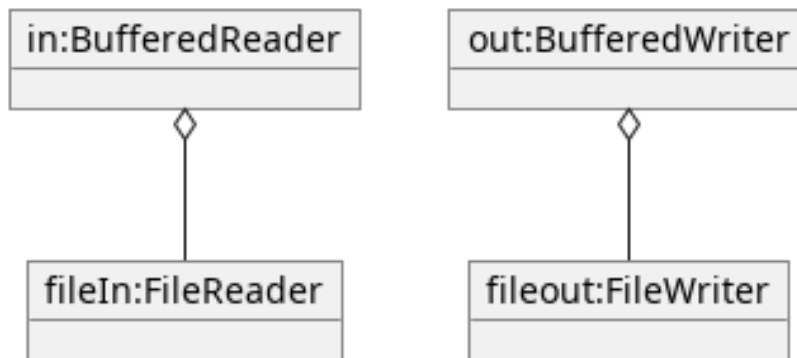
    contents = in.hipster("tst2.txt");
    for (String l : contents) { System.out.println(l); }

}
}

```

16 Java Readers and Writers

- Conceptually, you wrap different levels of readers until you get the functionality you want
 - Typically, you will want at least a `BufferedReader` and a `BufferedWriter`



- Reader
 - `BufferedReader`
 - `CharArrayReader`
 - `FilterReader`

- InputStreamReader
 - * FileReader
- PipedReader
- StringReader
- Writer
 - BufferedWriter
 - CharArrayWriter
 - FilterWriter
 - OutputStreamWriter
 - * FileWriter
 - PipedWriter
 - StringWriter
 - PrintWriter

17 When Things Go Wrong: Testing for Runtime Errors

- Compile-time errors covered previously:
 - Syntax Errors
 - * the compiler tells you what you have written wrong
 - * Read the compiler errors
 - Logical Errors
 - * the compiler does not know this is wrong
 - * Use Unit Testing to capture this.
 - * Debugger
- Runtime errors :
 - Input Testing
 - * Ensure that user enters “correct” information
 - Exception handling
 - * runtime exceptions
 - * e.g. “file not found”, “divide by zero”, ...

18 Handling Runtime Errors

- Handling compile-time errors is done during development
- Handling errors caught by automated tests is done during development
- Handling runtime errors *have to be* handled during runtime!

19 Detecting the Error

- Defensive Programming
 - Balance between checking everything and put reasonable trust in the program.
- Anticipating and avoiding the error
- Detecting the error state
- Deciding what to do about the error
- Deciding how much to report to the user

20 A Brief Word on Input Sanitisation

- Never ever *ever* assume that input is correct
 - User input
 - Command line parameters
 - File contents
 - Socket traffic
 - Fetched data (e.g., html, xml, json)
 - HTTP header
 - Properties/Config files
 - ...
 - <https://github.com/kuronpawel/big-list-of-naughty-strings>
- Java is safer than C++, but can still be really dangerous.
 - What do you do with the input?
 - * Pass it on as a field to the database?
 - * Display it in a web browser?
 - * Use it as a script and interpret it?
 - What do you do if not all data is present?
 - What do you do if the data is of the wrong type?

21 Method Input: Parameters

- Are the parameters to a method within bounds?
- Are the parameters not null?
 - Are there good defaults to use instead?
- Will the method complete even with incorrect parameters?
- Will the result be meaningful?

22 Error Reporting (to the user)

- Is the error quietly fixable?
- Should the user be *informed* , or *warned*
- Should the program *exit*?
- Write to a log to facilitate debugging

```
int x = 10;
Logger l = Logger.getLogger("se.bth.example.system.Test");
l.entering("Main","main()");

l.setLevel(Level.ALL);
l.info("Current log level " + l.getLevel());

l.config("Using default configuration");
l.info("Starting...");

l.log(Level.FINE, "Still here...");
l.fine("Also still here");

l.warning("Not sure how to proceed here");

l.exiting("Main", "main()");
```

23 Error Reporting (within program)

- Magic return value that indicates error
 - `null`
 - `-1`
 - `false`
- Throw an Exception (Will it be caught)
- Set an error flag somewhere
- ...

24 Error Handling

- Check for magic error number
- Catch Exception
- Check error flags
- Deal with the error if possible
 - Maybe return some indication that there was an error

- Log the error
- Maybe re-try the same operation
 - How many times?
 - Delay between each try?
- When all fails: re-throw exception
- **Always** clean up when you detect an error!
 - If you tried to open a file, try to close it
 - If you had an open database connection, close it
 - ...

25 Sammanfattning

- Användarinteraktion kan göras i olika nivåer
 - start-parametrar och utskrifter
 - in- och utmatning i en konsol
 - text-baserat UI
 - Grafiska gränssnitt
 - web-applikationer
- `Java.Swing` för att bygga grafiska gränssnitt i Java
 - Finns naturligtvis verktyg för detta, men är ganska enkelt att skriva själv
- `EventListeners` för att binda samman GUI-kod med din applikation
- In-och utmatning till filer
- Felhantering, `try-catch`, felrapportering

26 Nästa Föreläsning

- Virtuella Maskiner och Containers
- Utveckla i en Container
- Utvecklingsprocess för Kompilerade respektive Tolkade programspråk

27 Övning: Mera Pony-Statistik

- I projektet JavaPonies finns en branch PonyStatistics där interfacet från tidigare föreläsningar implementeras.
- Checka ut JavaPonies på ett nytt ställe (om du vill spara din implementation) och byt branch:
 - `git clone https://codeberg.org/mickesv/JavaPonies.git`
 - `cd JavaPonies && git checkout PonyStatistics`
- Kontrollera vad som ändrats: `git diff origin/main`

27.1 En Ny Main

1. Skriv en ny klass `JavaPonyStatistics` extends `JavaPonies` (att ärva från `JavaPonies` gör att du kan spara mycket av uppstarten från `JavaPonies`).
2. Skriv en ny `main()` - funktion i `JavaPonyStatistics` :

```
public static void main(String[] args) {  
    JavaPonyStatistics ps = new JavaPonyStatistics();  
    ps.printStatistics();  
}
```

3. Implementera metoden `JavaPonyStatistics.printStatistics()` så att den:
 - itererar över alla Ponies (du hittar dem via `myPonies.findAll()`), och
 - skriver ut Categories, Behaviour Groups, Behaviours, Effects, Interactions, och Speakig Lines:

Printing Statistics for Apple Bloom

Categories:

Behaviour Groups:

Behaviours: stand, walk, follow_{aj}, spin_{merightround}, workout, aww, CMC, dance

Effects:

Interactions:

Speaking lines:

- CUTIE MARK CRUSADER DESKTOP PONIES!!!
- Did I get my cutie mark? Did I? Did I!?
- Scoot-Scootalooo!
- Aww!
- Aren't you gonna stay for brunch?
- But I want it now!
- I am a big pony!
- I'm not a baby, I can take care of myself!
- Likely story.
- Not the cupcakes!

- Some pony needs to put this thing out of its misery.
- You're not using power tools, are you?
- Scootaloo! Scoot-Scootaloo!
- Trust me.
- What a thing to say!

27.2 Bara en enda Pony

Notera signaturen för `main`: `public static void main(String [] args)`

public så att man kommer åt den utanför klassen

static så att man inte behöver först skapa ett objekt

void man kan inte returnera något

main så att runtime-java vet vilken metod den skall leta efter

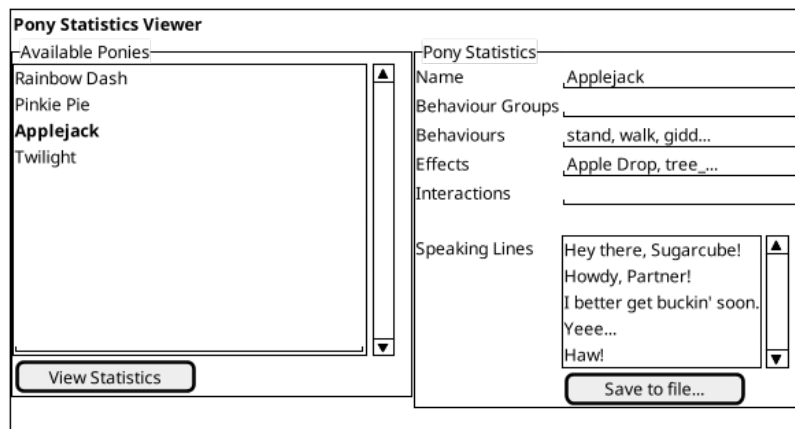
String [] args Här kommer alla kommandorads-parametrar.

Dags att lägga till lite interaktivitet:

1. Om (`0 == args.length`) , lista statistik för alla ponies (som tidigare)
2. Annars, hitta alla ponies som innehåller `arg[0]`.
 - Du kommer vilja se till att allting är antingen stora eller små bokstäver: `String::toLowerCase()` .
 - Det räcker att veta om ponnyns namn *innehåller* strängen, använd `String::contains()` .
 - `pony.getName().toLowerCase().contains(arg[0].toLowerCase())`

27.3 Ett Grafiskt Gränssnitt

1. Skriv en ny klass `view/PonyStatisticsViewer` som skapar en `JFrame` enligt nedan.
2. Skriv en funktion för att fylla listan med namnen på alla tillgängliga Ponies.
3. Skriv kod så att när man har valt en Pony och trycker på knappen "View Statistics", så visas statistiken till höger.
4. Vänta med "Save to file..." - knappen.



27.4 Save to File...

Nu är det dags att implementera "Save to file...":

1. När man trycker på knappen skall en `javax.swing.JFileChooser` öppnas.
2. Statistiken om den valda Ponyn skall sedan skrivas till den angivna filen.
3. Kontrollera att filen har rätt innehåll genom att öppna den (eller visa den i din terminal)
1. MWE for Save to File

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.filechooser.FileSystemView;

public class FiCH {

    public static void main(String [] args) {
        JFrame f = new JFrame("FiCH");
        f.setSize(500, 500);
        f.setVisible(true);
        JLabel l = new JLabel("no file selected");

        JButton button1 = new JButton("save");
        button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                JFileChooser j = new JFileChooser(FileSystemView.getFileSystemView().getHomeD
                int result = j.showSaveDialog(f);
                if (result == JFileChooser.APPROVE_OPTION) {
                    l.setText(j.getSelectedFile().getAbsolutePath());
                } else {
                    l.setText("the user cancelled the operation");
                }
            }
        });
    }
}
```

```
    }  
    } });  
  
    JPanel p = new JPanel();  
    p.add(button1);  
    p.add(l);  
    f.add(p);  
}  
}
```