CS209

Computer system design and application

Stéphane Faroult
faroult@sustc.edu.cn

Zhao Yao zhaoy6@sustc.edu.cn

MidTerm Exam

April 24th, usual room, lecture time

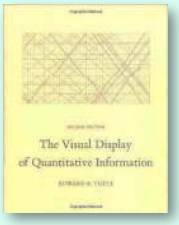
Next week: review

Graphics in Java

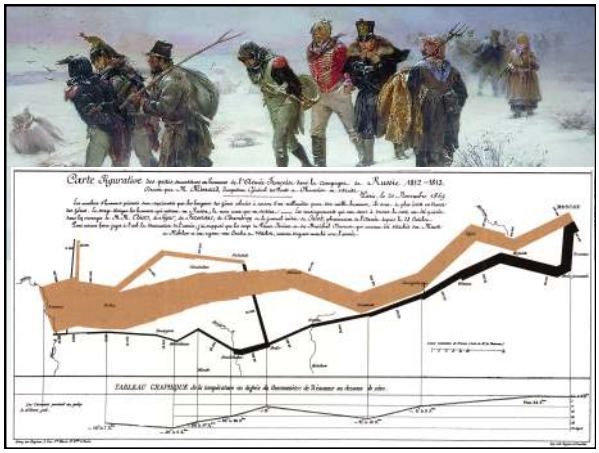
Let's now switch to what most monitoring tools use intensely: graphics. We'll only talk of relatively classic graphics that are often used in business applications.

Excellent book

Flickr:
Leonard Ling



If you have the opportunity to find this book in a library, take a look at it. The title isn't glamorous but the book is remarkable. What Tufte cites as one of the best graphics ever created is on the next slides and displays the 1812 disastrous French Russian campaign (numbers have been debated, but it's not the point). On a 2D surface you have a map, the size of the army, time, temperature (when retreating) and it remains remarkably legible. It wasn't done by a program ...



Charts

Much used in business applications...

```
import javafx.scene.chart.*;

class PieChart
class XYChart
```

Pie charts are hated by Tufte.

X can be a real X or the name of a category.

- AreaChart
- BarChart
- BubbleChart
- LineChart
- ScatterChart
- StackedAreaChart
- StackedBarChart

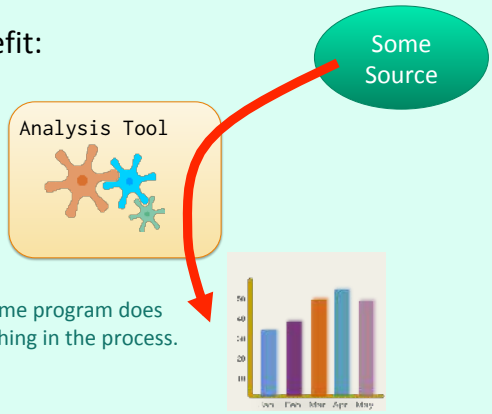
Benefit:

Very often people collect data with a program, generate a .csv or .txt file, load the data in Excel and create charts in Excel.

Writing a tool that collects source data and generates graphics for reports without having to use Excel

This is pretty time-consuming (even if you have macros, because you have to open files and so forth). Charts are therefore useful not only in an interactive application, but also to help generate reports.

Benefit:



The same program does everything in the process.

A Bar Chart example

Charts are widgets backed by data (they can also be updated dynamically), which means as usual Observable Lists.

Backed by an ObservableList

```
ObservableList<XYChart.Data<XType,YType>>
```

```
#EVENT TIME_WAITED PCT_WAITS WAIT_CLASS
"db file sequential read",2172313,87.370,"User I/O"
"db file scattered read",130611,5.250,"User I/O"
"log file switch (checkpoint incomplete)",84041,3.380,"Configuration"
"enq: TX - row lock contention",34906,1.400,"Application"
"log file switch completion",19113,0.770,"Configuration"
"direct path write temp",11049,0.440,"User I/O"
"log file sync",10550,0.420,"Commit"
"read by other session",8530,0.340,"User I/O"
"control file sequential read",3638,0.150,"System I/O"
"db file parallel read",2477,0.100,"User I/O"
"direct path read temp",2332,0.090,"User I/O"
"SQL*Net more data to client",2234,0.090,"Network"
"SQL*Net message to client",1453,0.060,"Network"
"buffer busy waits",566,0.020,"Concurrency"
"Streams AQ: qmn coordinator waiting for slave to start",490,0.020,"Other"
"SQL*Net more data from client",437,0.020,"Network"
"control file heartbeat",392,0.020,"Other"
"direct path read",240,0.010,"User I/O"
"latch free",147,0.010,"Other"
"enq: CF - contention",127,0.010,"Other"
```

My data comes from this file. What I want to chart is highlighted.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.collections.*;

import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;

import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.BufferedReader;
import java.io.IOException;
```

This is what I need for the chart

This is what I need for reading the file

```
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.BufferedReader;
import java.io.IOException;

public class BarChartExample extends Application {
    private final String dataFile = BarChartExample.class
                                .getClassLoader()
                                .getResource("data.txt")
                                .toString().replace("file:",
                                                    "");

    private static ObservableList<XYChart.Data<String,Number>>
         data = FXCollections.observableArrayList();
```

I prepare the name of my file and an ObservableList where I'm going to store what I read from the file.

```
private static ObservableList<XYChart.Data<String,Number>>
     data = FXCollections.observableArrayList();

static void loadData(String file) {
    // Here it's loaded from a file, it could
    // as well be queried from a database (this type
    // of data is obtained by querying system tables).
    // We are only interested by the first and third
    // fields from each line (event name and percentage)
    try (BufferedReader reader
          = Files.newBufferedReader(Paths.get(file))) {
      String   line = null;
```

Going to split each line on commas.

```
          String[] fields;
          while ((line = reader.readLine()) != null) {
            if (!line.startsWith("#")) { // not a comment
              fields = line.split(",");
              data.add(new
XYChart.Data<String,Number>(fields[0].replace("\"",
                                                  ""),
                                  new Float(fields[2])));
            }
          }
        } catch (IOException x) {
          System.err.format("IOException: %s%n", x);
        }
    }
```

Adding the first field (X) and 3rd field (Y) of each row to the collection. Note that each object in the collection is XYChart.Data<X type, Y type>

```
@Override
public void start(Stage stage) {
     stage.setTitle("Technical Bar Chart");
     final CategoryAxis xAxis = new CategoryAxis();
     final NumberAxis yAxis = new NumberAxis();
     final BarChart<String,Number> bc =
                new BarChart<String,Number>(xAxis,yAxis);
     bc.setTitle("Database Waits");
     xAxis.setLabel("Event");
     yAxis.setLabel("Percentage of Waits");
     bc.setLegendVisible(false);
```

Because I only have ONE series of data

You can have several values (series) for Y associated with every X, it's not my case.

```
         loadData(dataFile);
         XYChart.Series<String,Number> series =
                   new XYChart.Series<String,Number>();
         series.setData(data);

         Scene scene  = new Scene(bc,1000,800);
         bc.getData().add(series);
         stage.setScene(scene);
         stage.show();
       }

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

Used as root node

There is a setData() wich reminds of setItems(). Note that here the BarChart is used as root node. It works because a Chart is a child of Region, like a Pane.

Here is the result.
Note that I have specified nothing about how everything should appear (it can be changed with CSS)

Generating Image Files

Having an image on screen is nice, but if you want to include it into a report taking a screenshot isn't the most convenient. You can save a chart to a file, using a component that actually comes from Swing ...

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.VBox;
import javafx.scene.layout.HBox;
import javafx.scene.control.Button;
import javafx.stage.Stage;
import javafx.collections.*;

import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.File;
```

We cannot add to the chart a button to save the image, so we are going to put chart and button inside boxes.

```
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.File;

import javafx.embed.swing.SwingFXUtils;
import javafx.scene.image.WritableImage;
import javax.imageio.ImageIO;
import javafx.stage.FileChooser;

import javafx.geometry.Insets;
import javafx.geometry.Pos;

public class SaveChartExample extends Application {
    private final String dataFile = ...;

    private static ObservableList ...;

    static void loadData(String file) { ... }
```

Here is the magical Swing package. We also need to write the image and a Dialog for saving it.

```java
@Override
public void start(Stage stage) {
    stage.setTitle("Technical Bar Chart");
    VBox  box = new VBox();
    final CategoryAxis xAxis = new CategoryAxis();
    final NumberAxis yAxis = new NumberAxis();
    final BarChart<String,Number> bc =
              new BarChart<String,Number>(xAxis,yAxis);
    bc.setTitle("Database Waits");
    xAxis.setLabel("Event");
    yAxis.setLabel("Percentage of Waits");
    bc.setLegendVisible(false);
```

Here is the box where we are going to stuff Chart and Button.

```java
bc.setAnimated(false);
          // IMPORTANT. Must be done before
          // you start plotting.

loadData(dataFile);
XYChart.Series<String,Number> series
             = new XYChart.Series<String,Number>();
series.setData(data);
bc.getData().add(series);
bc.setPrefWidth(800);
bc.setPrefHeight(700);
box.setPadding(new Insets(10));
box.setAlignment(Pos.CENTER);
box.getChildren().add(bc);
```

By default a chart can be animated. It musn't be if you want to save it as an image. Then nothing new apart from adding the chart to the vertical box.

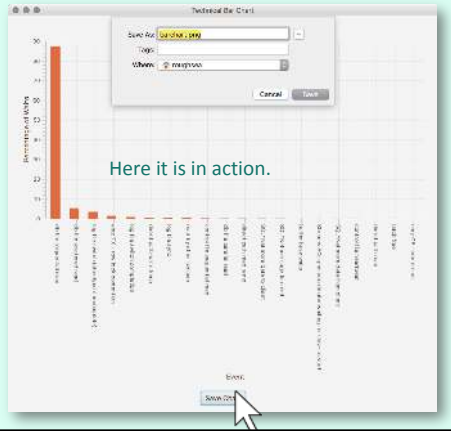
```java
HBox hbox = new HBox();
hbox.setPadding(new Insets(5));
hbox.setAlignment(Pos.CENTER);
Button saveButton = new Button("Save Chart");
hbox.getChildren().add(saveButton);
box.getChildren().add(hbox);

saveButton.setOnAction((e)->{
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Save Chart");
    fileChooser.setInitialFileName("barchart.png");
    File selectedFile =
              fileChooser.showSaveDialog(stage);
```

Another box (mostly to control padding) to add the button, then the action: when you click on the button, the dialog opens with a default filename.

```java
saveButton.setOnAction((e)->{
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("Save Chart");
    fileChooser.setInitialFileName("barchart.png");
    File selectedFile =
              fileChooser.showSaveDialog(stage);
    if (selectedFile != null) {
      try {
        WritableImage snap = bc.snapshot(null, null);
        ImageIO.write(SwingFXUtils.fromFXImage(snap,
                      null),
                      "png", selectedFile);
      } catch (IOException exc) {
        System.err.println(exc.getMessage());
      }
    }
  });
```

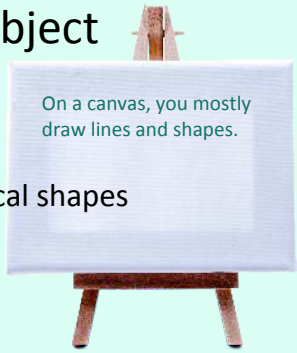
If you didn't click "Cancel" in the dialog (which would return null) you can take a snapshot in the program and save it.
... Skipping the end of the program ...



2D Graphics

Charts are 2D graphics (you also have 3D charts but if you ever read Tufte you'll never want to use them), but in charts you haven't full freedom to draw whatever you want on the screen. If you want to draw you should you a Canvas object. "Canvas" was the name of the cloth used in the old days for making ship sails. Put on a wooden frame, this is what western artists started to use around the 17th century for painting, hence the name in graphical interfaces.

Canvas object



Lines

Geometrical shapes

On a canvas, you mostly draw lines and shapes.

Examples you usually find

Check figure 15.32 in the book (close)

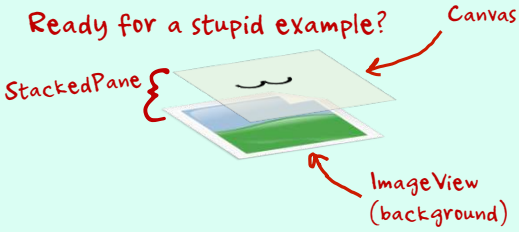
Whenever you look for a Canvas example, you find a program generating something like this.

From an article by Manoj Debnath on www.developer.com

I have decided to have a stupid example of my own. Background image, and Canvas on which I draw two half circles close to each other (part-circles shapes are called "arc", which is French for "bow" )

arc (de triomphe), originally to honor victorious armies in Rome, idea copied in Paris

Ready for a stupid example?




```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.paint.*;
import javafx.scene.canvas.*;
import javafx.scene.shape.*;
import javafx.stage.Stage;
import javafx.stage.Screen;
import javafx.scene.image.*;
import java.net.URL;

public class StupidCanvasExample extends Application {

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

A couple of new packages to import.

```
public void start(Stage stage) {
    double width;
    double height;
    double x;
    double y;

    stage.setTitle("StupidCanvasExample");
    stage.setResizable(false);
    Group root = new Group();
    Scene scene = new Scene(root);
    StackPane pane = new StackPane();
    URL url = this.getClass()
                .getClassLoader()
                .getResource("background.jpeg");
```

StackPane to put image and Canvas (transparent) on top of it.

```
    if (url != null) {
      Image image = new Image(url.toString());
      width = image.getWidth();
      height = image.getHeight();
      ImageView iv = new ImageView(image);
      pane.getChildren().add(iv);

      final Canvas canvas = new Canvas(width, height);
      GraphicsContext gc = canvas.getGraphicsContext2D();
```

To draw on a Canvas, you need the associated "GraphicsContext". This is where you define, among other things, line thickness and colours.

```
gc.setStroke(Color.BLACK);
gc.setLineWidth(height * 0.01);
x = 0.42 * width - width / 36.0;
y = 0.285 * height;
gc.strokeArc(x, y,
             width / 18.0, height / 40.0,
             180, 180, ArcType.OPEN);
x += width / 18.0;
gc.strokeArc(x, y,
             width / 18.0, height / 40.0,
             180, 180, ArcType.OPEN);
pane.getChildren().add(canvas);
// Make canvas disappear when clicked
canvas.setOnMouseClicked((e)->{
   canvas.setVisible(false);
});
```

There are multiple ways to define colours. For basic colors you can use an enum.

"Stroke" refers to lines. When you draw, you give the position of the top left corner, plus parameters that depend on the shape drawn.

```
        }
        root.getChildren().add(pane);
        stage.setScene(scene);
        stage.show();
    }
}
```

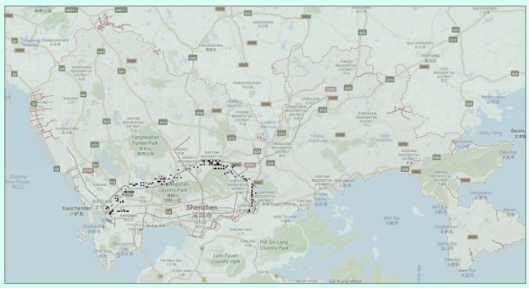
And there you go. All the art, of course, is in the choice of the suitable background image.

From stupid to usable in real life

Use a map as background

Draw routes

If instead of using Mona Lisa you use a map, you can create some really interesting applications with canvases (other than a drawing tool).



You could have for instance one Canvas per Metro line, stack all of them, and use buttons to make a line appear or disappear. That said, working with maps is not very easy because what you want to plot are usually places for which you know latitude and longitude.

Problems with maps

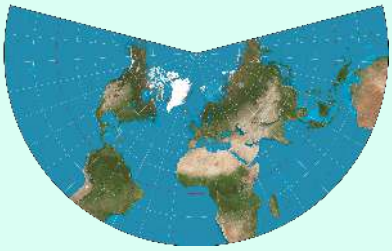
1. Projection

There are multiple ways to project a latitude and longitude on a flat screen, from the relatively simple cylindrical projection.

Problems with maps

1. Projection

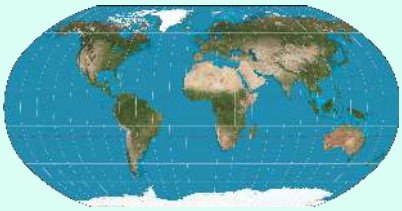
... or a conical projection (that makes here southern regions look far bigger than they are)



Problems with maps

1. Projection

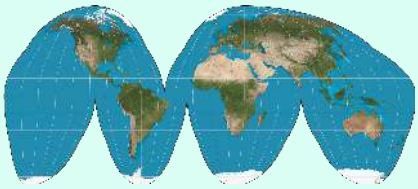
To projections that attempt to keep the surfaces right (but not the angles)



Problems with maps

1. Projection

To projections that try to achieve the "let's take an orange peel and lay it flat on the table" effect.

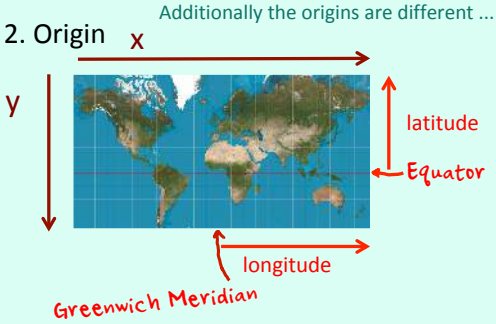


As you can guess, finding the (x,y) matching a given longitude and latitude can be mathematically challenging.

Problems with maps

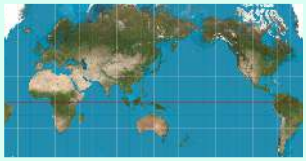
Additionally the origins are different ...

2. Origin



Problems with maps

2. Origin



... and it may be further moved if you don't want to see the Greenwich Meridian (or the Equator) right in the middle. You need to write methods that know how to translate latitude and longitude depending on several parameters.

Interaction?

You can only interact with a "Node". A Canvas is a node, and you can interact with it (I was able to make the Canvas over Mona Lisa invisible by clicking on it). However, you cannot directly interact with the shapes drawn over the canvas. If you want to interact with shapes, you need Shape objects, one by shape.

Shape objects

- Arc
- Circle
- Line
- Polygon
- Rectangle
- Text
- ...

You have a corresponding Shape object for every shape you can draw on a Canvas.

Stroke

Color .setStroke(col)

Width .setStrokeWidth(w)

+ other properties

You can set for them what you can set in the GraphicsContext of a Canwas (note that the Width of a Stroke is the line thickness)

Stroke

Color .setStroke(col)

Width .setStrokeWidth(w)

+ other properties

Fill

Color, gradient, image/pattern

You can fill shapes also and when you specify colours you can give the amount of Red, Green and Blue, as well as a parameter that specifies transparency (0 = completely transparent)

```
Color.rgb(255, 255, 0, 1.0)
```

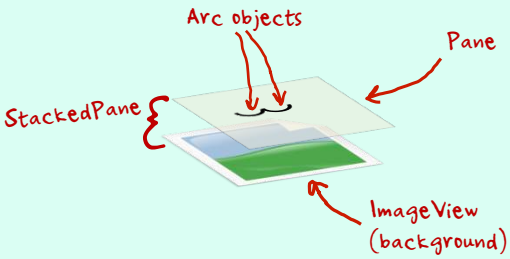
Shapes are nodes ...

CLICKABLE!

With Shapes you can click on every individual shape.

Let's redo the Mona Lisa example with shapes instead of a Canvas.

Stupid example, revisited



```
Arc arc = new Arc();
arc.setCenterX(0.42 * width);
arc.setCenterY(0.288 * height);
arc.setRadiusX(width / 36.0);
arc.setRadiusY(width / 36.0);
arc.setStartAngle(180.0);
arc.setLength(180.0);
arc.setType(ArcType.OPEN);
arc.setStroke(Color.BLACK);
arc.setStrokeWidth(height * 0.01);
arc.setFill(Color.rgb(255, 255, 255, 0.0));
shapePane.getChildren().add(arc);
```

More code than with a simple drawing (... but we could create a method for that)

Same with arc2

```
arc.setOnMouseClicked((e)->{
  Random rand = new Random();
  int r = rand.nextInt(256);
  int g = rand.nextInt(256);
  int b = rand.nextInt(256);
  Color col = Color.rgb(r, g, b);
  arc.setStroke(col);
  arc2.setStroke(col);
});
```

I associate the same action to a click on each arc, that changes the color for both arcs.

Canvas or individual shapes?

Depends on how many elements

Possible to check where a Canvas was clicked

Shapes are good if you have few of them. Otherwise everything can become slow (and it may become difficult to make sure you clicked at the right place).

3D Graphics

I won't talk about 3D graphics because it's beginning to become very specific to advanced applications. Let's just say that you have packages in JavaFX for 3D graphics as well.

Audio and Video

You can also play audio and video in JavaFX. It's not very different from images. With images you have an Image object, and the ImageView that shows it on screen. With audio and video, you have a Media object, you have a MediaView, and between the two you have a MediaPlayer object with controls allowing you to start, pause, stop, rewind and so forth.

Very much like Images

`Media`

`MediaPlayer` ← Controls

`MediaView`

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.geometry.Pos;
import javafx.util.Duration;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;

public class MediaDemo extends Application {
    private final String MEDIA_URL = this.getClass()
                                     .getClassLoader()
                                     .getResource("TestVid.mp4")
                                     .toString();
    private final String MEDIA_URL =
            "http://edu.konagora.com/video/TestVid.mp4";
```

Here are the new packages

OR

URL, Path and String

URL: **prefix://path**

PATH: **path**

A Media (like an Image) can take a URL as argument of a constructor. An URL (like an URI, basically the same thing) is a prefix + a path. If the prefix is "file:" it means that the resource (... name given to anything you can load) is accessible through the file system of your computer (it's not necessarily local, it can be a network disk). It can also be something else such as "http:" to mean that the resource is accessed from a web server through HTTP requests. You usually need to apply toString() to them.

```
    private final String MEDIA_URL =
            "http://edu.konagora.com/video/TestVid.mp4";

    @Override
    public void start(Stage primaryStage) {
        Media media = new Media(MEDIA_URL);
        int width = media.widthProperty().intValue();
        int height = media.heightProperty().intValue();
        MediaPlayer mediaPlayer =
                            new MediaPlayer(media);
        MediaView mediaView =
                            new MediaView(mediaPlayer);
        Button playButton = new Button(">");
```

Once the media is loaded, you associate it with a MediaPlayer, and the MediaPlayer with a MediaView. Controls will execute MediaPlayer methods.

```
        playButton.setOnAction(e -> {
            if (playButton.getText().equals(">")) {
              mediaPlayer.play();
              playButton.setText("||");
            } else {
              mediaPlayer.pause();
              playButton.setText(">");
            }
          });
        Button rewindButton = new Button("<<");
        rewindButton.setOnAction(e->
                        mediaPlayer.seek(Duration.ZERO));
        Slider slVolume = new Slider();
```

The text on the button tells us what is the current state, and whether we should play or pause. I'm also adding another button for rewinding, and a new widget (Slider) for setting the volume.

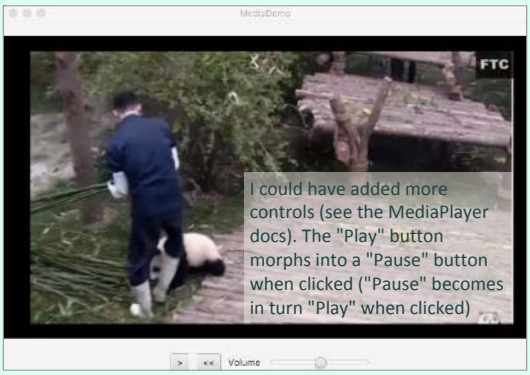
```
        slVolume.setPrefWidth(150);
        slVolume.setMaxWidth(Region.USE_PREF_SIZE);
        slVolume.setMinWidth(30);
        slVolume.setValue(50);
        mediaPlayer.volumeProperty()
                   .bind(slVolume.valueProperty()
                                 .divide(100));
        HBox hBox = new HBox(10);
        hBox.setAlignment(Pos.CENTER);
        hBox.getChildren().addAll(playButton,
                                  rewindButton,
                                  new Label("Volume"),
                                  slVolume);
        BorderPane pane = new BorderPane();
```

Other than geometry (size) I give the range and initial value for the slider (0 to 100, initially 50) and "bind" it to the MediaPlayer. There is an implicit ChangeListener behind, to change the volume when the slider moves.

```
        BorderPane pane = new BorderPane();
        pane.setCenter(mediaView);
        pane.setBottom(hBox);
        Scene scene = new Scene(pane, 750, 500);
        primaryStage.setTitle("MediaDemo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Controls are in a box, everything is added to a BorderPane (that controls placement as top/right/bottom/left and center) and we are ready to go.



Skinning a GUI using CSS

Last JavaFx subject, superficial in all meanings of the word but important (looks sell): how to change the appearance of a JavaFx application. I have already briefly talked about it, the best way is to do it through an external style sheet (.css file). People will be able to change, often in a very impressive way, the looks of your application by changing this file and without any need to access the code (in fact, they just need the .css and the .class to run the "modified" application).

If you want to see how far you go with "styling", you can visit http://csszengarden.com and click on designs on the right hand-side. The same page will look completely different.

I have already said it, CSS stands for Cascading Style Sheet and Cascade is French for waterfall.
The name emphasizes that styles "drip down", and are inherited from previous style sheets.

Cascading Style Sheet

It all starts with HTML

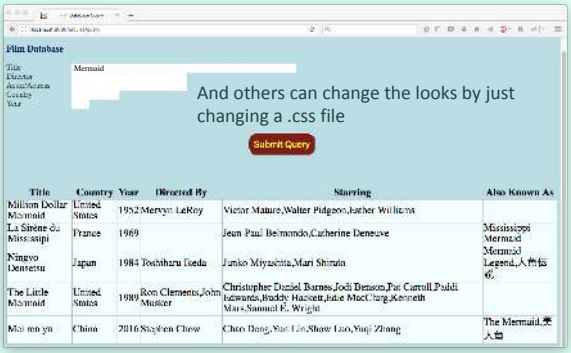
HyperTextMarkupLanguage

The idea is stolen from web applications. Web applications just send HTML pages to browsers that decode and display these pages. An HTML page is organized by sections between pairs of tags (<tag> at the beginning, </tag> at the end) that structure the document and can contain in turn other pairs of tags, thus defining a kind of hierarchical structure (note that some tags, such as those for images, act both as opening and closing tags). Tags pretty often also contain attributes (such as the image file name for an image tag).

<tags> ← Associate formatting with each tag in a "stylesheet"

In the very early days of the web, people were using tags to format their pages, for instance what they wanted in bold was between **<bold>** and **</bold>** (inspired by previous document generation systems that were sending special signals to printers), and you could change the fonts with **<font** *attributes specifying the font, size and everything*> ... ****. As websites were growing in size and number of pages, and as increasingly pages were generated by programs instead of being created by hand, it became unmanageable, especially when the marketing department was deciding on new corporate colors. So the idea was to associate formatting to tags in one or several separate text files.

Some people could just focus on the "engine", such as here an application allowing to retrieve film information from a database ...



And others can change the looks by just changing a .css file

(I know it's ugly but I didn't work very hard on it)

The way it works in web pages

Before I discuss about CSS in JavaFx, I'm going to talk about CSS with HTML, because there is far more CSS written for HTML than for JavaFX.

There are three main ways to specify how to display what is between tags.

1. You can specify for a given tag, associating a tagname with visual characteristics
2. I have mentioned that tags can have attributes, one is "class" (unrelated to object-oriented programming) listing one or several categories. This allows to create a subcategory, or to give some common visual characteristics across different tags that have the same class (for instance "inactive")
3. You can give another attribute "id" to a tag, an this allows you to make one particular tag look really special.

<tag>

```
tag {
    attribute: value;
        ...
}
```

<tag class="category">

```
.category {
    attribute: value;
        ...
}
```

<tag id="name">

```
#name {
    attribute: value;
        ...
}
```

You have here how the tag looks in HTML and how styling looks in CSS.

I focus on simple tags in the next example.

<body>

In a HTML page, everything that is displayed is between <body> and </body> tags, so <body> really defines the global looks of the page.

Link <a>

Title <h1>

<table>

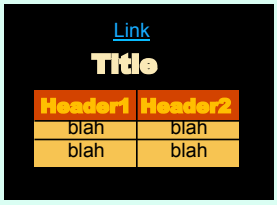
Header1 **Header2** <th>

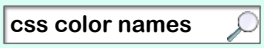
blah blah

blah blah <td>

In my example, I'll use tags <a> associated with a link, <h1> with a (first level) header, <table>, <th> (table header) and <td> (table data). I have omitted <tr> (table row) which usually surrounds the columns of a same row. The tag will be there on the page, but I'm not associating any special style with it in my example.

```
body {text-align: center;
      background-color: black;}
a {color: lightskyblue;}
a:visited {color: lightseagreen;}
a:hover {color: salmon;}
h1 {color: cornsilk;}
th {background-color: sienna;
    color: orange;}
td {background-color: burlywood;}
table {width: 80%;
       margin-left: auto;
       margin-right: auto;
       text-align: center;}
form {width: 50%;
      margin-left: auto;
      margin-right: auto;
      padding: 20px;
      background-color: silver;}
```





This will give you the names of colors understood by browsers.

Notice the special a:visited (link you have already clicked on) and a:hover (when the cursor moves over the link)

```
body {text-align: center;
      background-color: black;}
a {color: lightskyblue;}
a:visited {color: lightseagreen;}
a:hover {color: salmon;}
h1 {color: cornsilk;}
th {background-color: sienna;
    color: orange;}
td {background-color: burlywood;}
table {width: 80%;
       margin-left: auto;
       margin-right: auto;
       text-align: center;}
form {width: 50%;
      margin-left: auto;
      margin-right: auto;
      padding: 20px;
      background-color: silver;}
```

styles.css

Styling is written to a file that is included in the HTML page by a special tag in the <header>...</header> section that precedes the "body". Then you can change it when needed.

```
<link rel="stylesheet" href="styles.css"/>
```

The way it works in javaFX

Nodes are almost equivalent to tags

`.root` plays the same role as **body**

otherwise use class names prefixed with a dot

`.button`

You have of course no tags in a JavaFx application, but you have the same kind of hierarchy through nodes, containers and widgets. The JavaFx class names are used with the same syntax as the HTML classes in CSS, that means that they are prefixed by a dot.

The way it works in javaFX

Nodes are equivalent to tags

Attribute names are prefixed with **-fx-**

```
-fx-font-size: 150%;
```

CSS attributes also have a special name with JavaFx. The idea is to be able to have a single CSS files shared by a Web and a JavaFx application without having any conflict.

The way it works in javaFX

Nodes are equivalent to tags

Attribute names are prefixed with **–fx-**

```
Node.setId("name")
Node.getStyleClass().add("css class")
```

Finally, node methods allow you to associate with a node the same kind of attributes as with a HTML tag. You can only have a single Id, but you can have several classes, and therefore getSyleClass() returns a list.

The way it works in javaFX

Nodes are equivalent to tags

Attribute names are prefixed with **–fx-**

We have already seen how to load the CSS file into the JavaFX application.

```
Node.setId("name")
Node.getStyleClass().add("css class")
```

```
Node.setStyle("-fx-attribute: value")
```

```
Scene scene = new Scene(new Group(), 500, 400);
scene.getStylesheets().add("path/styles.css");
```

Everything cannot be styled with CSS in JavaFX

CSS styling allows you to go rather far in JavaFx, but not as far as you could go in HTML. Some elements may prove hard to style with CSS. You may sometimes have to code some styling in the Java application. However, if you still want this styling to be "externalized", don't forget that properties files also provide a way to read attributes at run-time. It's of course better to have all styling at one place, but it's better to use a properties file than to hard-code.

A Properties file might sometimes be a workaround

PERSISTENCE

After Graphical User Interfaces, our main "big" topic will be the topic of persistence.

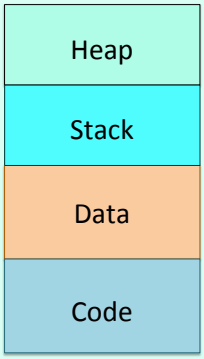
Another word that comes from Latin, and conveys an idea of continuity and robustness.

Data Persistence

per sistere

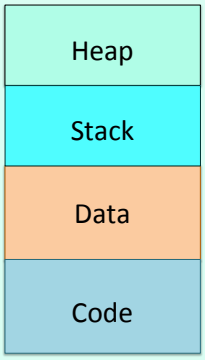
thoroughly

stand still

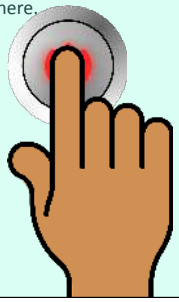


von Neumann Architecture

In the von Neuman architecture we have everything in memory.



But what happens when the power is cut? Poof, gone. We can reload programs when we restart, but if they modify data we'd rather save the work done somewhere.



save, keep

Need to persist program results somewhere

That's all the idea of persistence. "Somewhere" may take multiple shapes (including a remote computer)

Especially in the old days

Memory

volatile

Memory is like cars: fast goes hand-in-hand with expensive. Fast memory is "volatile", which means (that's the meaning of the word) that it's content can "fly away" (if you cut power).
Slow memory stays, but the problem is that when you access it your program works at its slow speed.

non-volatile

Flickr: Andrew Magill

So we have to work as much as we can in memory, and only in memory, for speed ...

Mostly work in memory for speed

Memory

... while we still need a safety net.

Write to file for safety

magnetic

Non-Volatile Memory Technology

There are several non-volatile technologies available. The oldest one, still alive and kicking, is the magnetic disk, which has done tremendous progress over the years in data density and transfer rates (not so much directly accessing data somewhere on the disk, called "random access")

optical

Non-Volatile Memory Technology

Optical technologies are no longer so hot as they used to be, but they are very good for archiving and, because readers/writers are really mass produced, fairly cheap. Data updates? Better to look elsewhere.

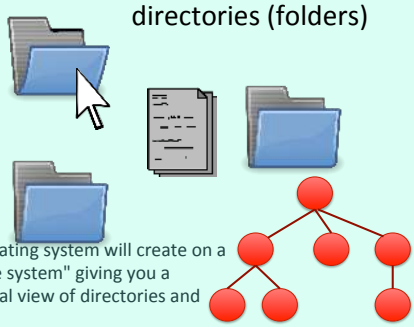
solid-state

Non-Volatile Memory Technology

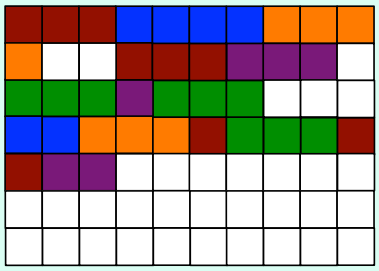
Solid-state technologies are very good for reading, much less so for writing.

File System

directories (folders)



Your operating system will create on a disk a "file system" giving you a hierarchical view of directories and files.



But in reality the system will reserve blocks, all multiples of a same unit, that it will associate with one file. Files will grow, deletions will create gaps soon filled by blocks from from other files, and the system will try to do all this as efficiently as possible. There is a strong disconnect between the view we have and physical reality.