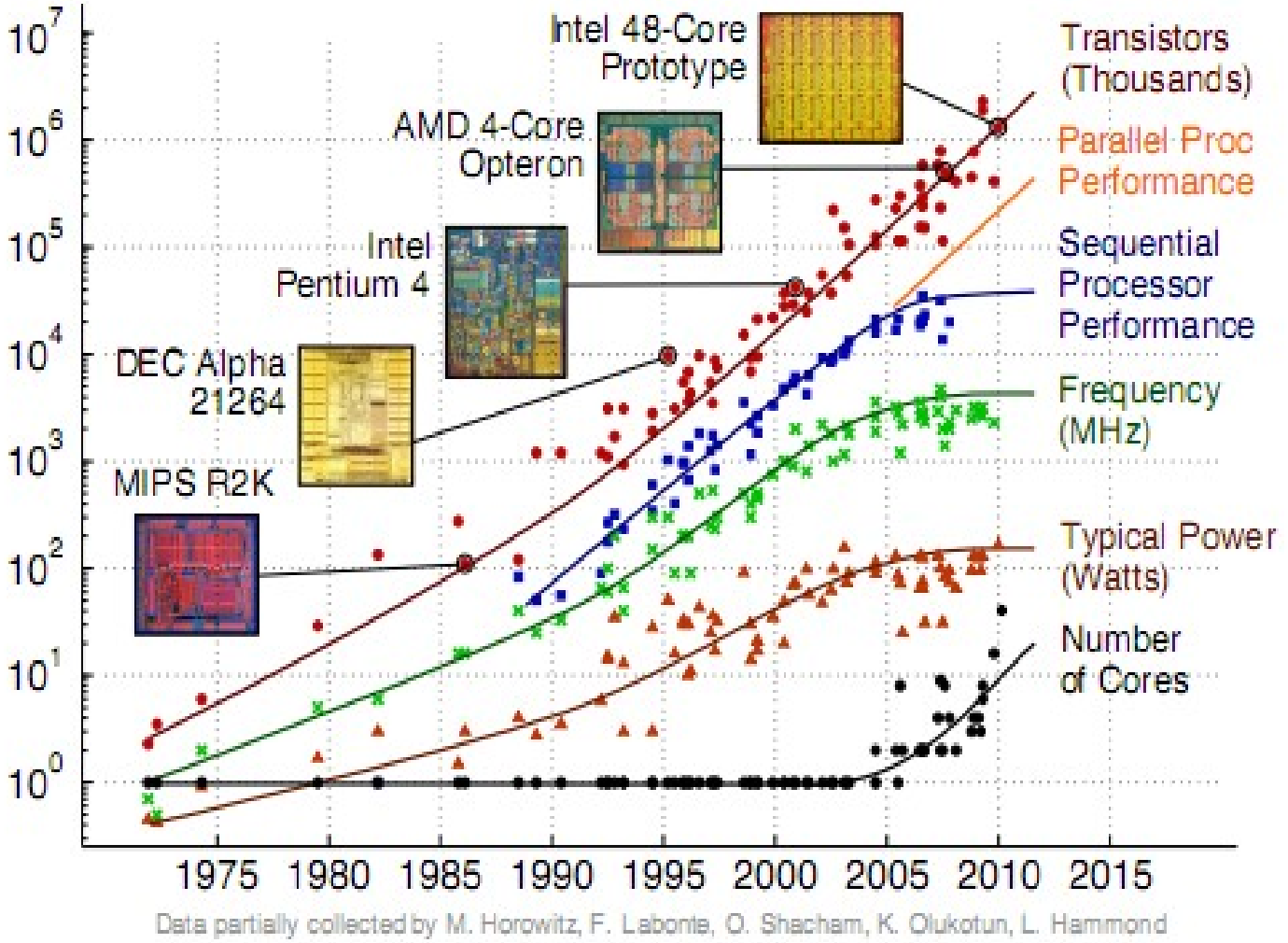
Threads

The Free Lunch Is Over



Parallelität

Verschiedene Programmteile laufen gleichzeitig

- Aufwendige Berechnungen im Hintergrund, trotzdem benutzbare GUI
- Verschiedene Netzwerkverbingungen gleichzeitig, auf alle soll reagiert werden
- Falls mehrere Prozessoren vorhanden sind: Jeder Programmteil bekommt volle Rechenleistung eines Prozessors, damit insgesamt schnellere Berechnung

Motivation

- Graphische Oberflächen benötigen einen unabhängigen Kontrollfluss
- Die Simulation realer Objekte wird erleichtert
- Mehrere Eingabequellen müssen effizient beantwortet werden
- Blockierende (oder langsame) Kommunikation soll nicht das gesamte Programm blockieren
- Effiziente und parallel arbeitende Webserver
- Verteilte Systeme

Geschichte der Verwendung von Nebenläufigkeit

- Die effiziente Ausnutzung von langsamer Peripherie führte zum Multitasking.
- Die ersten interaktiven Mehrbenutzersysteme (multics, tops10, unix) basierten auf Multiprocessing
- Echtzeitsysteme benötigen zumindest eine einfache Form von Nebenläufigkeit (Interrupt-Technik)
- Die ersten OO-Sprachen (Simula, Smalltalk) enthielten Nebenläufigkeit

Geschichte der Verwendung von Nebenläufigkeit

- In den 70ern und 80ern wurden unterschiedliche Modelle von OO und nicht OO Nebenläufigkeit untersucht.
- Seit den 70ern gibt es intensive Entwicklungen und Anwendungen im Bereich der Parallelverarbeitung
- Fast jeder Processor verfügt heute über mehrere Rechenerkerne
- Heute ist Multiprocessing und Multithreading (praktisch) auf jedem System und in jeder Programmiersprache möglich (und nötig)

Sequentielle Programmausführung

- Ein sequentielles Java-Programm entspricht dem prozeduralen Paradigma. Soweit man die Ergebnisse beobachten kann, ist die Ausführungsreihenfolge exakt durch das Programm vorgegeben.
- Der Compiler, die Java-Laufzeitumgebung und der Prozessor können innerhalb dieser Grenzen Modifikationen vornehmen um die Effizienz zu steigern.

Sequentielle Programmausführung

- Soweit es keinen Einfluss auf die Ergebnisse hat, kann die Reihenfolge von Befehlen verändert werden (reordering).
- Daten können zeitweise in Registern und Cache-Speichern gehalten werden. In dieser Zeit ist der Inhalt des Hauptspeichers nicht korrekt. Es ist denkbar, dass ein Objekte niemals im Hauptspeicher erscheint (visibility).
- Im Ergebnis kann erst durch diese Maßnahmen die Leistungsfähigkeit moderner Computer erreicht werden.

Nebenläufigkeit

- Unter Nebenläufigkeit versteht man die gleichzeitige Ausführung von Programmen oder Programmteilen.
- In einem nebenläufigen Programm ist die Reihenfolge der Befehlsausführung nicht vollständig festgelegt. Das Programm kann durch einen oder mehrere Prozessoren ausgeführt werden.
- Nebenläufige Abläufe können über gemeinsamen Speicher verfügen.

Ablaufreihenfolge bei Nebenläufigkeit

Die genaue Ablaufreihenfolge ist nicht festgelegt, nur innerhalb eines Teil

Teil 1

```
foo = 0;
foo = foo + 1;
foo = foo + 2;
```

Teil 2

```
foo = 1;
foo = foo * 3;
foo = foo * 2;
```

Ablaufmöglichkeit

```
foo = 0;
foo = 1;
foo = foo + 1;
foo = foo * 3;
foo = foo + 2;
foo = foo * 2;

=> foo = 10
```

Ablaufmöglichkeit

```
foo = 0;
foo = foo + 1;
foo = foo + 2;

foo = 1;
foo = foo * 3;
foo = foo * 2;

=> foo = 6
```

Ein nebenläufiges Programm kann bei verschiedenen Laufen verschiedene Ergebnisse haben.

Probleme

Bei nebenläufigen Programmen können verschiedene Probleme auftreten, die sonst nicht auftreten können Arbeiten Programmteile gleichzeitig auf den gleichen Variablen, können Programme Fehler enthalten, die nur im Zusammenspiel entstehen

- Sie können steckenbleiben
- Sie können auf nicht initialisierte Werte zugreifen
- Sie können unsaubere Daten sehen
- Fehler treten nur mit gewisse Wahrscheinlichkeit auf

Probleme

Teil 1

```
foo = 0;
while(foo != 10) {
   foo = foo + 1;
}
```

Teil 2

```
foo = 0;
while(foo != 20) {
   foo = foo + 1;
}
```

Laufen die Teile gleichzeitig, können beide Threads steckenbleiben; oft bleibt nur einer oder keiner stecken

Probleme

```
class Foo {
    String name;
    public Foo() {}
    public void setName(name) {this.name = name;}
    public String getName() {return name;}
}
```

Teil 1

```
foo = new Foo();
foo.setName("Hallo Welt");
```

Teil 2

```
if(null != foo){
    System.out.println(foo.getName);
}
```

Probleme

```
public class Person {
   int alter; double gewicht;
   public Person(int alter, double gewicht) {
      this.alter = alter; this.gewicht = gewicht; }
   public Person kopie() {
      return(new Person(alter, gewicht)); }
   public update(int alter, double gewicht) {
      this.alter = alter; this.gewicht = gewicht; }
}
```

Teil 1

```
Person foo = new Person(26, 77);
foo.update(27,85);
```

Teil 2

```
Person bar = foo.kopie();
```

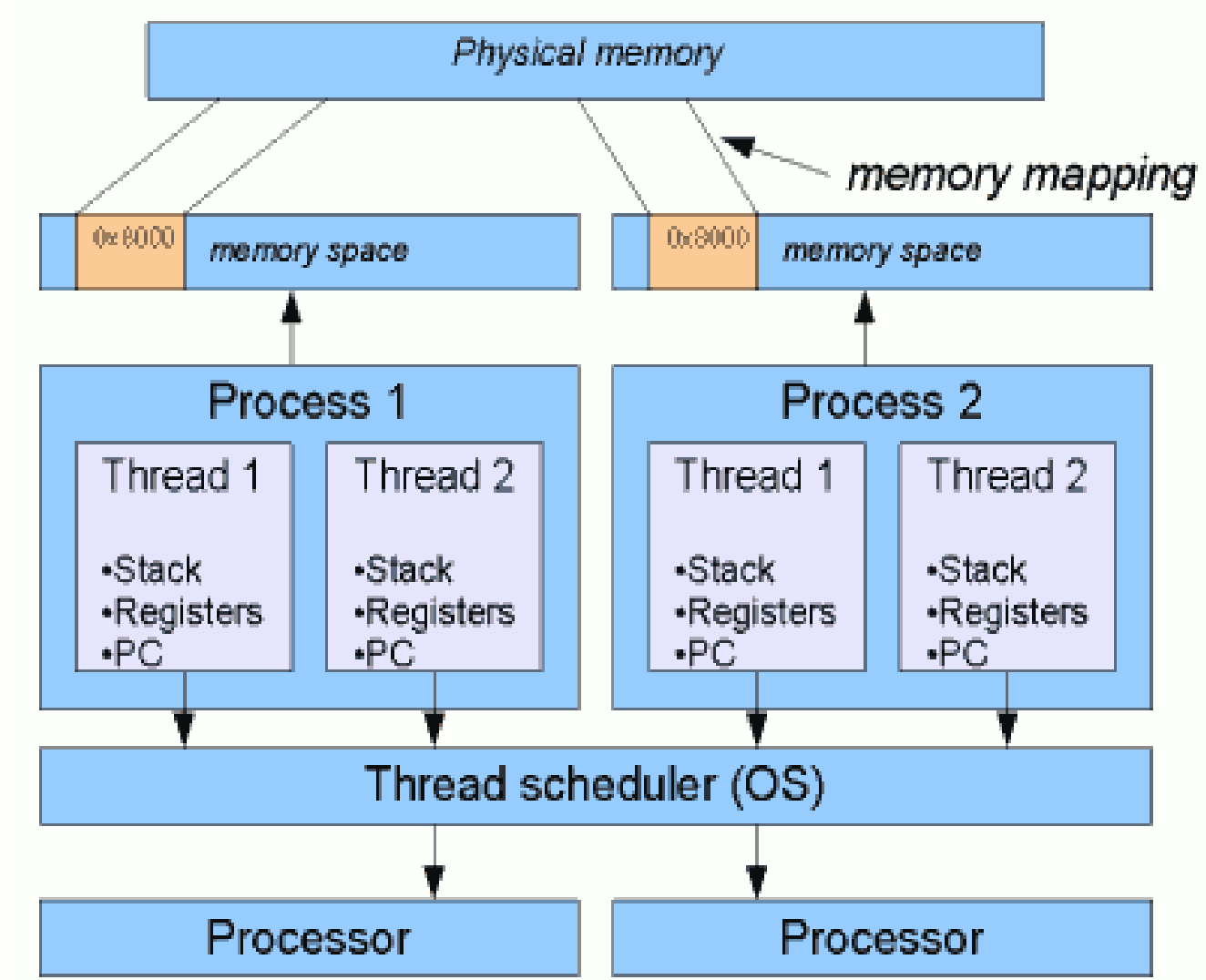
Prozesse

- Ein Prozess ist ein in der Ausführung begriffenes Programm
- Ein Programm beschreibt statisch Struktur und Ablauf
- Ein Prozess ist die dynamische Ausführung des Programm
- Prozesse werden von dem Betriebssystem verwaltet
- Interprozesskommunikation über das Betriebssystem ist langsam!

Threads

- Ein Thread ist eine Ausführungsreihenfolge innerhalb eines Prozesses.
- Ein Prozess enthält mindestens 1 Thread.
- Jeder Thread verfügt über einen separaten Stack (Rücksprung, lokale Variable)
- Jeder Thread verfügt über einen eigenen Programmzähler, d.h. eine eigene Kontrolle des Ablaufs

Threads


Physical memory
memory mapping
0x8000
memory space
0x8000
memory space
Process 1
Thread 1
•Stack
•Registers
•PC
Thread 2
•Stack
•Registers
•PC
Process 2
Thread 1
•Stack
•Registers
•PC
Thread 2
•Stack
•Registers
•PC
Thread scheduler (OS)
Processor
Processor

Threads und Objektorientierung

- In sequentiellen OO-Programmiersprachen ist zu jedem Zeitpunkt genau eine Methode aktiv.
- In dem entgegengesetzten Actor-Konzept sind alle Objekte gleichzeitig aktiv.
- Kommunikation erfolgt durch den Austausch von Nachrichten.
- Thread-Modelle der prozeduralen Welt verlegen die Nebenläufigkeit auf die Ebene von Prozeduren.
- Java: run() -> Thread gestartet werden

Synchronisation

- Stellt sicher, dass Daten nicht gelesen werden, während sie geschrieben werden
- Geschieht, indem angegeben wird, welche Programmteile nicht gleichzeitig ablaufen dürfen
- Kann prinzipiell selbst implementiert werden, Java stellt aber saubere Mechanismen dafür zur Verfügung: `synchronized`

Java

Es gibt verschiedene Möglichkeiten Nebenläufigkeit in Java zu implementieren

- Thread
- Timer
- Runnable

Java

Nebenläufigkeit durch die Klasse `Thread`

- Klasse `Thread` erweitern
- Methode `void run()` implementieren; ihr Inhalt bestimmt, was in dem Thread läuft
- Methode `void start()` startet den Thread nebenläufig

Beispiel

```
public class Runner {


    public void doit() {
        for(int i = 0; i < 7; i++) {
            System.out.println("Schleife " + i);
        }
    }
}
```

Beispiel

```
public class Main extends Thread {
    Runner runner;
    public Main(Runner run) { runner = run; }
    public void run() {
        runner.doit();
    }

    public static void main(String[] args) {
        Runner runner = new Runner();
        Main foo = new Main(runner);
        Main bar = new Main(runner);
        foo.start();
        bar.start();
    }
}
```

Ausgabe

Programm ergibt bei jedem Ablauf ein andere Ergebnis...

-

```
Schleife 0
Schleife 1
Schleife 2
Schleife 3
Schleife 4
Schleife 5
Schleife 6
Schleife 0
```

```
Schleife 0
Schleife 0
Schleife 1
Schleife 1
Schleife 2
Schleife 2
Schleife 3
Schleife 3
```

```
Schleife 0
Schleife 1
Schleife 0
Schleife 2
Schleife 1
Schleife 3
Schleife 2
Schleife 4
```

Synchronisation

- Synchronisation kann über synchronized implementiert werden
- synchronized-Methoden: Wird eine Methode als synchronized deklariert, so wird diese bei einem Objekt maximal einmal gleichzeitig ausgeführt Bei verschiedenen Objekten kann sie immernoch gleichzeitig ausgeführt werden
- Synchronized-Blöcke: Enthalten in ihrem Aufruf ein Objekt. Es wird gewartet, bis kein anderer Thread mehr in einem synchronized-Block mit diesem Objekt ist

Beispiel

```
public class Runner {


    public synchronized void doit() {
        for(int i = 0; i < 7; i++) {
            System.out.println("Schleife " + i);
        }
    }
}
```

Beispiel

```
public class Main extends Thread {
    Runner runner;
    public Main(Runner run) { runner = run; }
    public void run() {
        runner.doit();
    }

    public static void main(String[] args) {
        Runner runner = new Runner();
        Main foo = new Main(runner);
        Main bar = new Main(runner);
        foo.start();
        bar.start();
    }
}
```

Synchronisation

Nur zwei Abläufe möglich:

- foo arbeitet runner.doit() vollständig ab, danach bar
- bar arbeitet runner.doit() vollständig ab, danach foo
- doit() als synchronized deklariert ist
- foo und bar das gleiche Runner-Objekt verwenden

Beispiel

```
public class Main extends Thread {
    Runner runner;
    public Main(Runner run) { runner = run; }
    public void run() {
        runner.doit();
    }
    public static void main(String[] args) {
        Runner runnerfoo = new Runner();
        Runner runnerbar = new Runner();
        Main foo = new Main(runnerfoo);
        Main bar = new Main(runnerbar);
        foo.start();
        bar.start();
    }
}
```

Beispiel

```
public class Runner {


    public void doit() {
        synchronized(this) {
            for(int i = 0; i < 7; i++) {
                System.out.println("Schleife " + i);
            }
        }
    }
}
```

Beispiel

```
public class Main extends Thread {
    Runner runner;
    public Main(Runner run) { runner = run; }
    public void run() {
        runner.doit();
    }

    public static void main(String[] args) {
        Runner runner = new Runner();
        Main foo = new Main(runner);
        Main bar = new Main(runner);
        foo.start();
        bar.start();
    }
}
```

Synchronisation

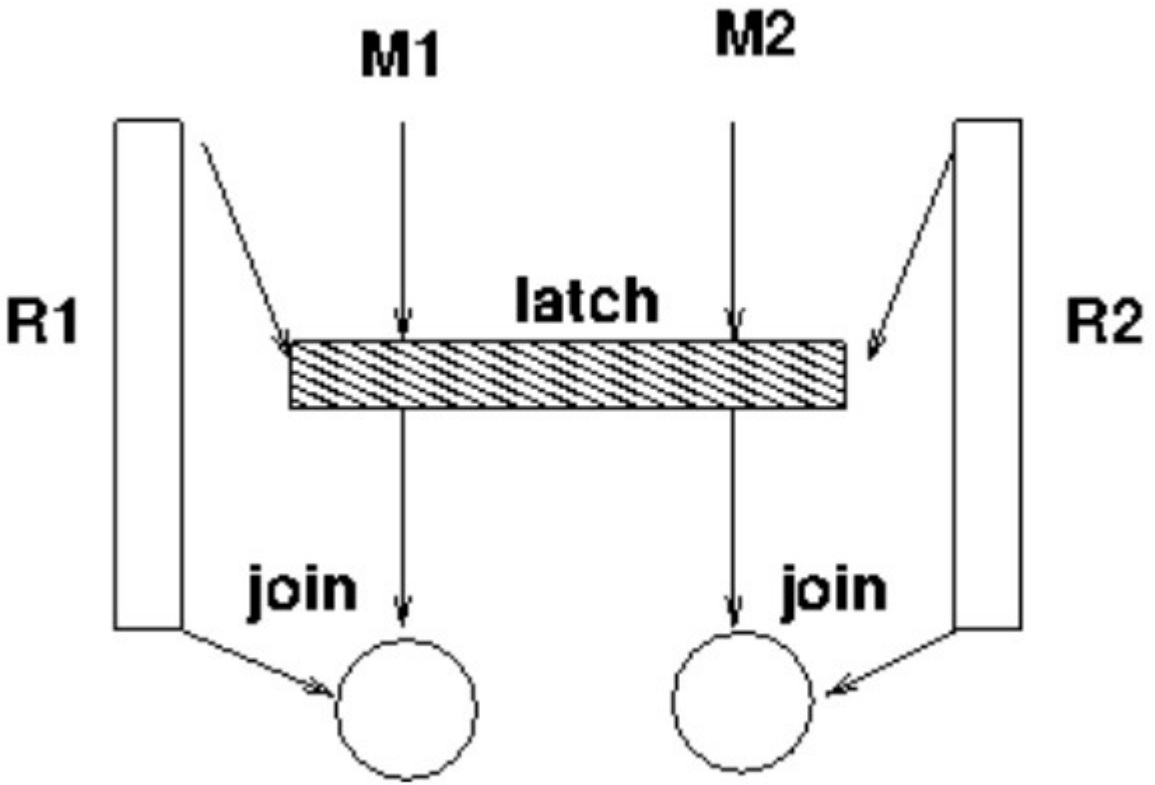
Nur zwei Abläufe möglich:

- foo arbeitet runner.doit() vollständig ab, danach bar
- bar arbeitet runner.doit() vollständig ab, danach foo
- Die Schleife liegt in einem synchronized-Block
- In beiden Threads ist das Objekt zum Block das gleiche

Beenden von Threads

- Ein Thread endet, wenn die run-Methode beendet wird
- Ein Java-Prozess endet, wenn alle Threads beendet sind
- Durch den Aufruf System.exit() wird immer der Prozess sofort beendet
- Ansonsten kann man den Thread enden nur durch entsprechende Mitteilung
- Mittels der Methode join() kann man darauf warten, dass ein bereits ausgeführter Thread beendet ist

Beispiel



Beispiel

```
class MyRunnable implements Runnable{

    @Override
    public void run() {
        System.out.println("Thread started:::"+Thread.currentThread().getName());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread ended:::"+Thread.currentThread().getName());
    }

}
```

```
public static void main(String[] args) {
    Thread t1 = new Thread(new MyRunnable(), "t1");
    Thread t2 = new Thread(new MyRunnable(), "t2");
    Thread t3 = new Thread(new MyRunnable(), "t3");

    t1.start();

    //start second thread after waiting for 2 seconds or if it's dead
    try {
        t1.join(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    t2.start();

    //start third thread only when first thread is dead
    try {
        t1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    t3.start();

    //let all threads finish execution before finishing main thread
    try {
        t1.join();
        t2.join();
        t3.join();
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    System.out.println("All threads are dead, exiting main thread");
}
```

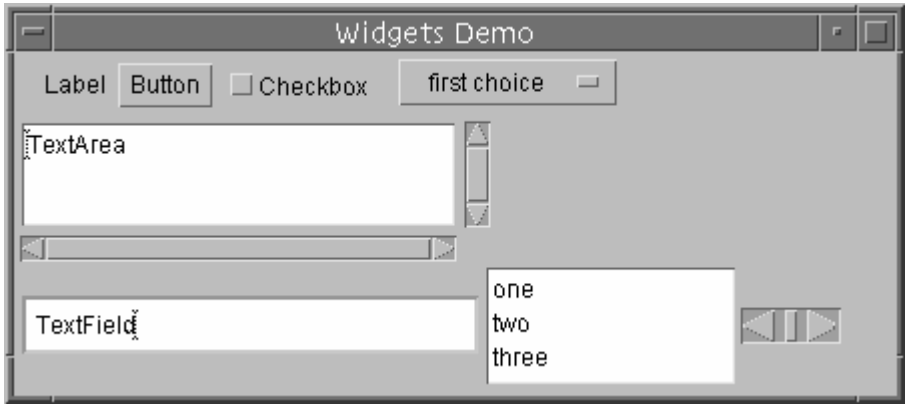
GUI

Grafische Benutzeroberflächen

- Windows Presentation Foundation (.NET)
- Qt,Wx (C++)
- Cocoa (Objective C, Swift, C#, Python,...)
- GTK (C++, Java, Python,...)
- Tk (Perl, Python,...)
- Swing, SWT, JavaFX (Java)

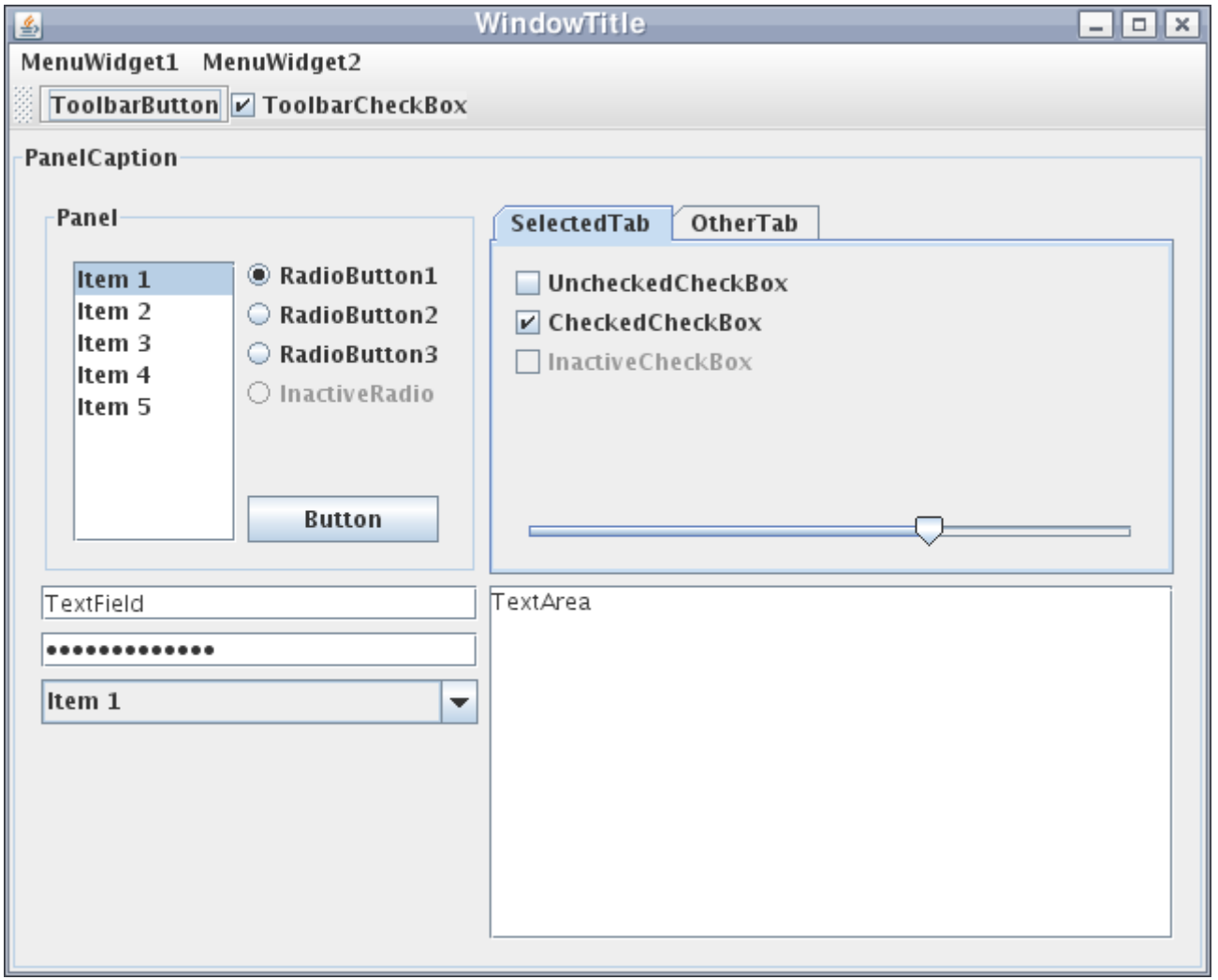
JAVA

- Der Anfang von Java GUIs: AWT



JAVA

- Kurz darauf: Swing



JavaFX

- FXML
- Markup Sprache, die auf XML basiert
  - Einfacheres Layouting von GUIs
- CSS für Styling ist sehr einfach
- Moderne Plattformen werden unterstützt
- Aktuelles Java wird unterstützt
- Scene Builder

JavaFX

- JavaFX-Programme erben von der abstrakten Klasse Application
- muss implementiert werden

  public void start (Stage stage)

  - der Ausgangspunkt für alle JavaFX Programme
  - diese Methode wird von uns überschrieben

  public static void launch(String... args)

  - wird meistens direkt in der main() Methode aufgerufen

JavaFX

- Beim Start des Programms wird ein Fenster (Stage) erzeugt
- Dort kann eine Szene hinzugefügt werden und das Fenster dargestellt werden
- Die Anwendung wird automatisch beendet, wenn das letzte Fenster geschlossen wird

JavaFX

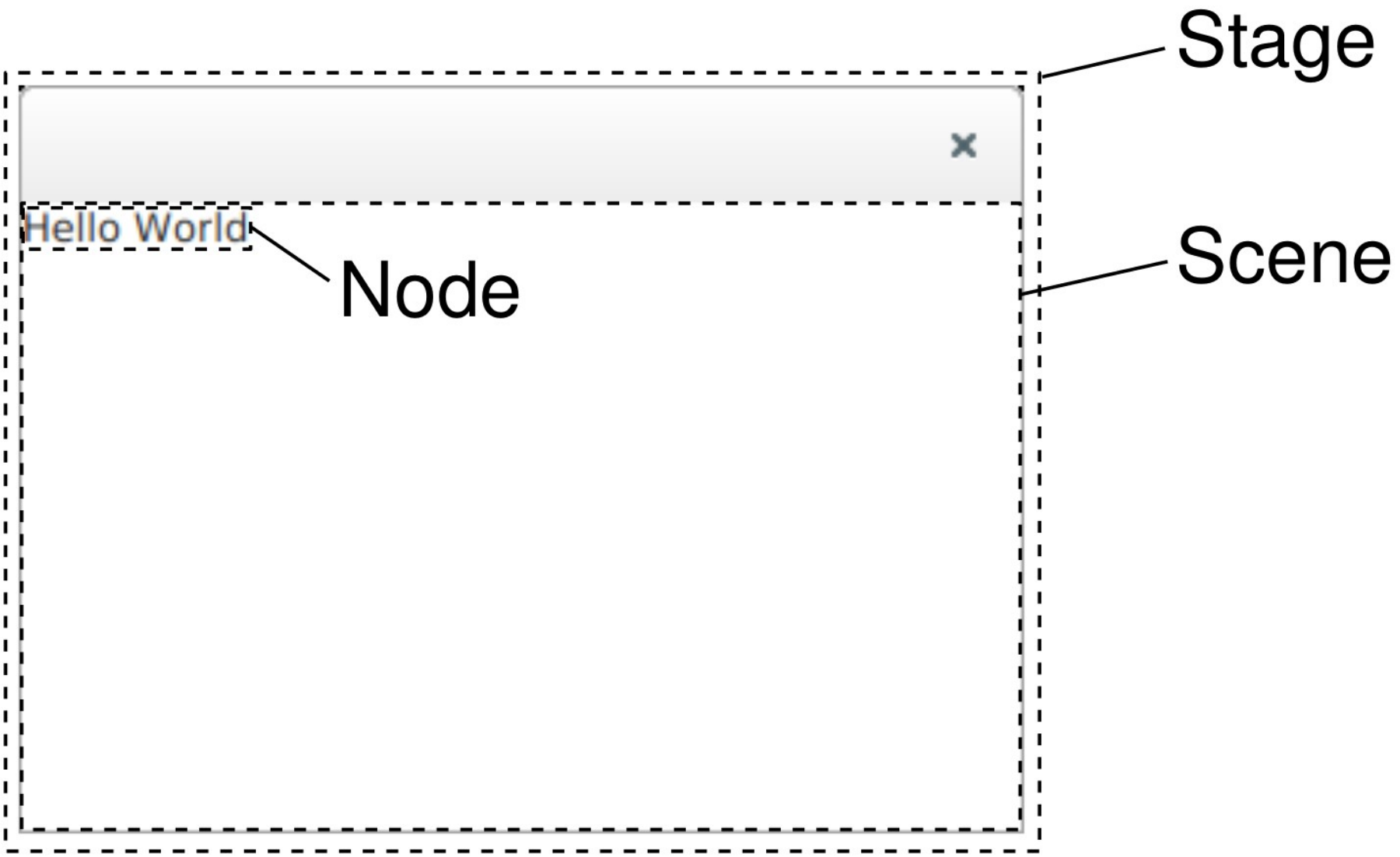
```
import javafx.application.Application;
import javafx.stage.Stage;

public class HelloWorld extends Application{
    public void start(Stage primaryStage)
        throws Exception {
    }

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

Stage & Scene

- GUI Elemente brauchen eine Scene
- Ein Programm hat Stages, die aus Scenes bestehen

```
Scene scene = new
Scene(root,width,height);
```

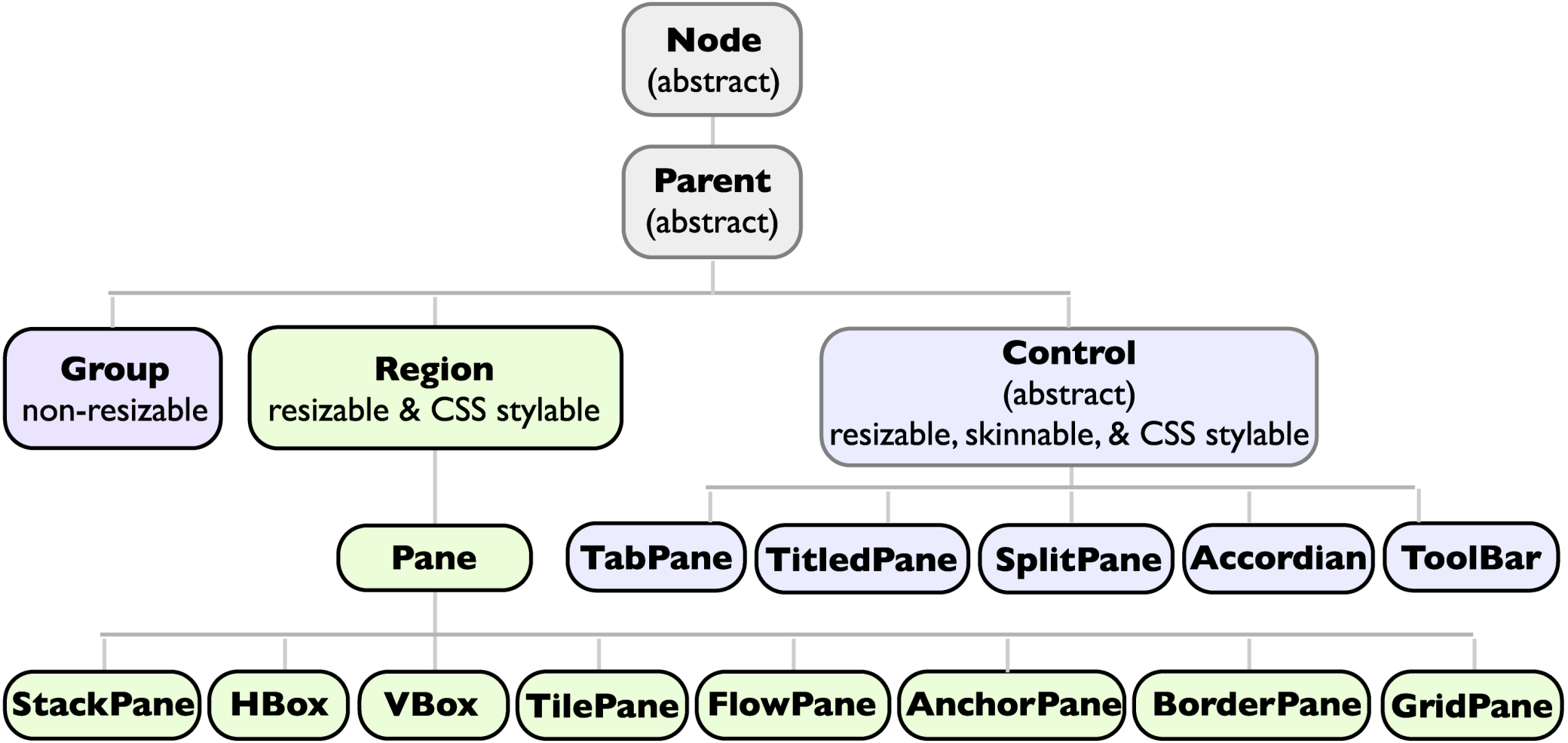
JavaFX



JavaFX

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

public class Main extends Application {

  @Override
  public void start (Stage stage) throws Exception {
    Node node = new Label ("Hello World");

    Group root = new Group ();
    root.getChildren().add (node);

    Scene scene = new Scene (root, 300 , 250);

    stage.setScene (scene);
    stage.show ();
  }
}
```

JavaFX 2.0 Layout Classes

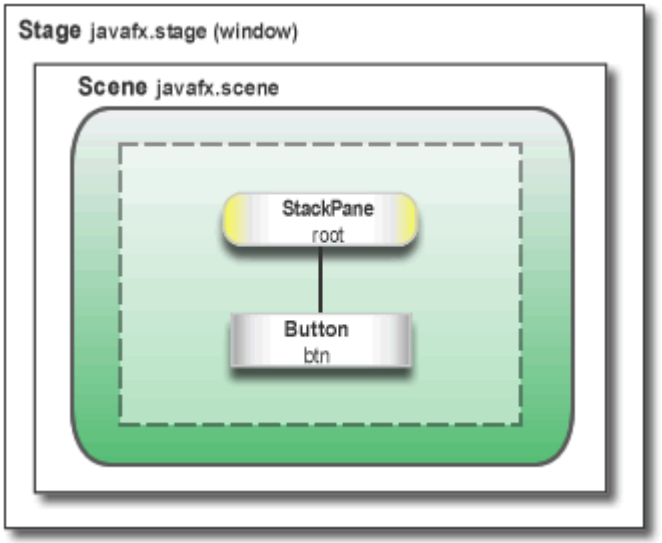


Nodes

- Die verschiedenen graphischen Elemente sind durch Unterklassen von Node repräsentiert
- Steuerelemente: Label, TextField, Button, Slider, . . .
- Geometrische Formen: Line, Circle, Polygon, ...
- Mehrere Node-Objekte können zu einem einzigen Node-Objekt zusammengefasst werden.
- Beispiele: Group, Region, Pane, BorderPane, HBox, VBox

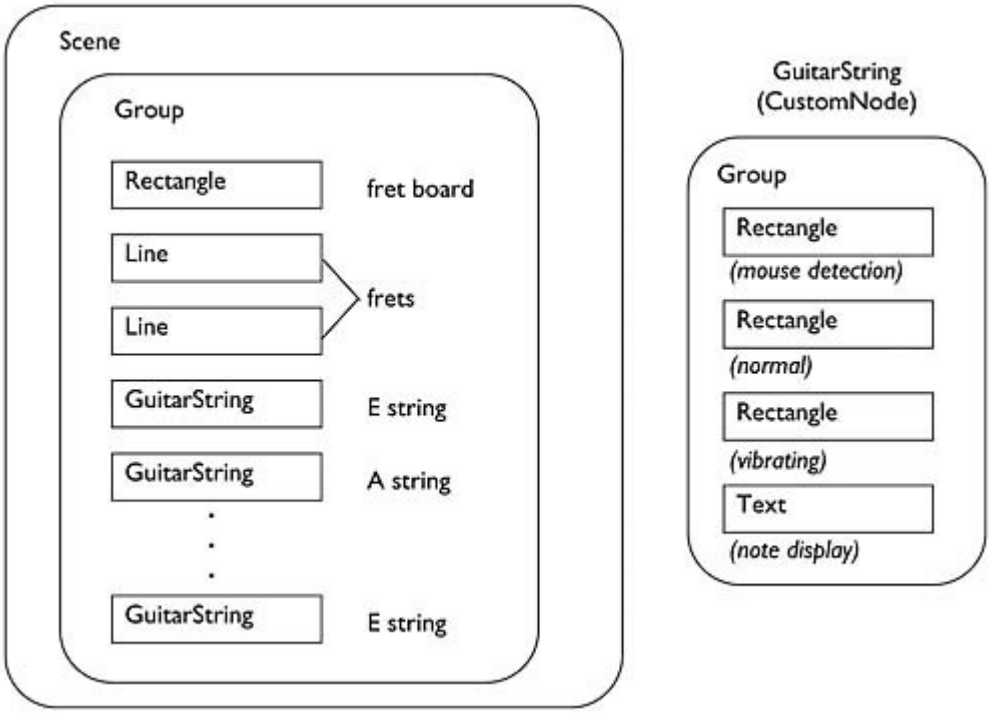
Layout

- Die Anordnung der Elemente innerhalb einer Szene wird objektorientiert modelliert
- Es gibt verschiedene Klassen, die als Container für andere Elemente eingesetzt werden können



Container-Nodes

- Group
- Unterklasse von Node
- Kind-Nodes erhalten ihre gewünschte Größe
- Gruppe ist gerade groß genug, um alle enthaltenen Objekte zu umschließen



Container-Nodes

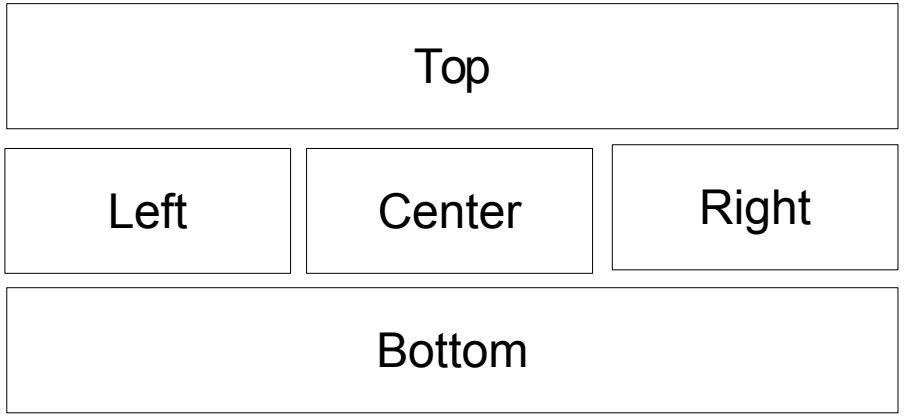
- Region
- Unterklasse von Node
- Größe unabhängig von Größe der Kinder
- jede Region definiert minimale, gewünschte und maximale Größe
- bei Größenänderungen werden die Kinder neu angeordnet und evtl. ihre Größe angepasst
- verschiedene Unterklassen implementieren verschiedene Layouts
- Pane - manuelle Größe und manuelles Layout der Kinder

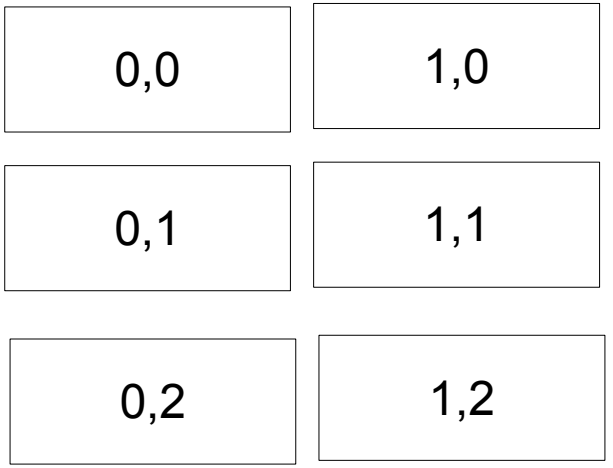
Region

- Hbox
- horizontale Anordnung der Kinder
- Verhalten der Kinder bei Skalierung kontollierbar
- die Kinder behalten ihre Größe und werden gleichmäßig angeordnet
- die Kinder werden zu gleichen Teilen vergrößert, um den zusätzlichen Platz zu füllen
- Vbox - analog zu HBox, nur vertikal

Region

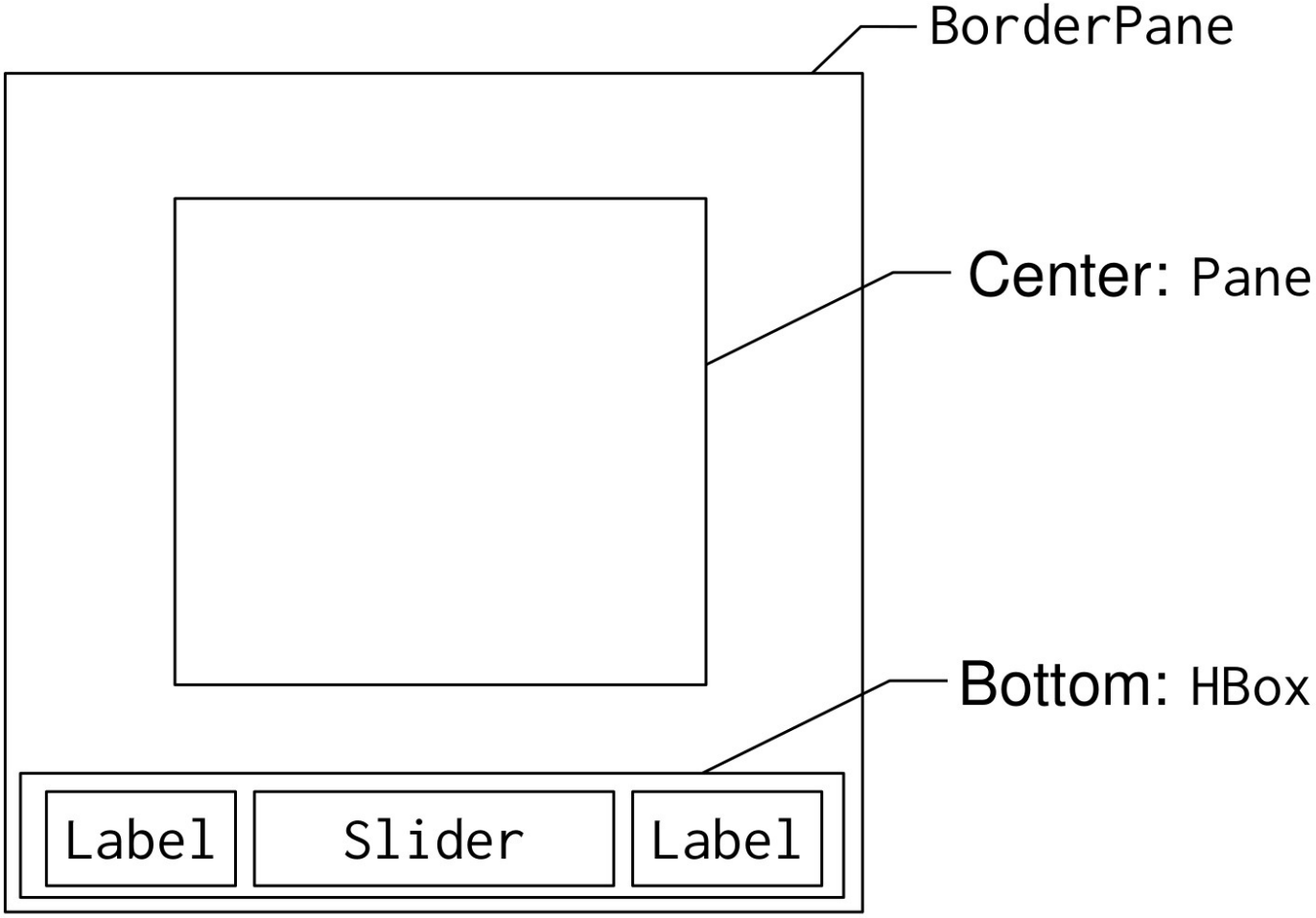
- BorderPane
- GridPane





Automatisches Layout

- Das Layout der GUI-Elemente wird durch die geeignete Schachtelung von Container-Nodes bestimmt



CASCADING STYLE SHEETS

- eine Stylesheet-Sprache
- WWW?
- CSS ermöglichen die Beschreibung von Layout und Präsentation
- CSS wurde entworfen, um Darstellungsvorgaben weitgehend von den Inhalten zu trennen

Styling mit CSS

- Details des Aussehens können über Stylesheets gesteuert werden
- style.css

```
.hbox {
  -fx-background-color: lightblue;
  -fx-alignment: center;
  -fx-padding: 15;
  -fx-spacing: 5;
}
#rect {
  -fx-fill: linear-gradient(from 0% 0% to 100% 100%, red 0%,
    black 100%);
  -fx-stroke: darkblue;
  -fx-stroke-width: 5;
}
#line {
  -fx-stroke: darkblue;
  -fx-stroke-width: 5;
}
```

Styling mit CSS

- JavaFX CSS Reference
- Auswahl des Stylesheets

```
hbox.getStyleClass().add("hbox");
line.setId("line");

rectangle.setId("rect");
borderPane.getStylesheets().add("path/to/style.c
ss");
```

fxml

- Oberfläche wird mit xml beschrieben
- Oberfläche wird in der Starterclass generiert
- Oberfläche kann durch Änderungen des xml-Files modifiziert werden, ohne neu compilieren zu müssen
- Die Verbindung zum Java-Programm - Eventhandler

Scene Builder

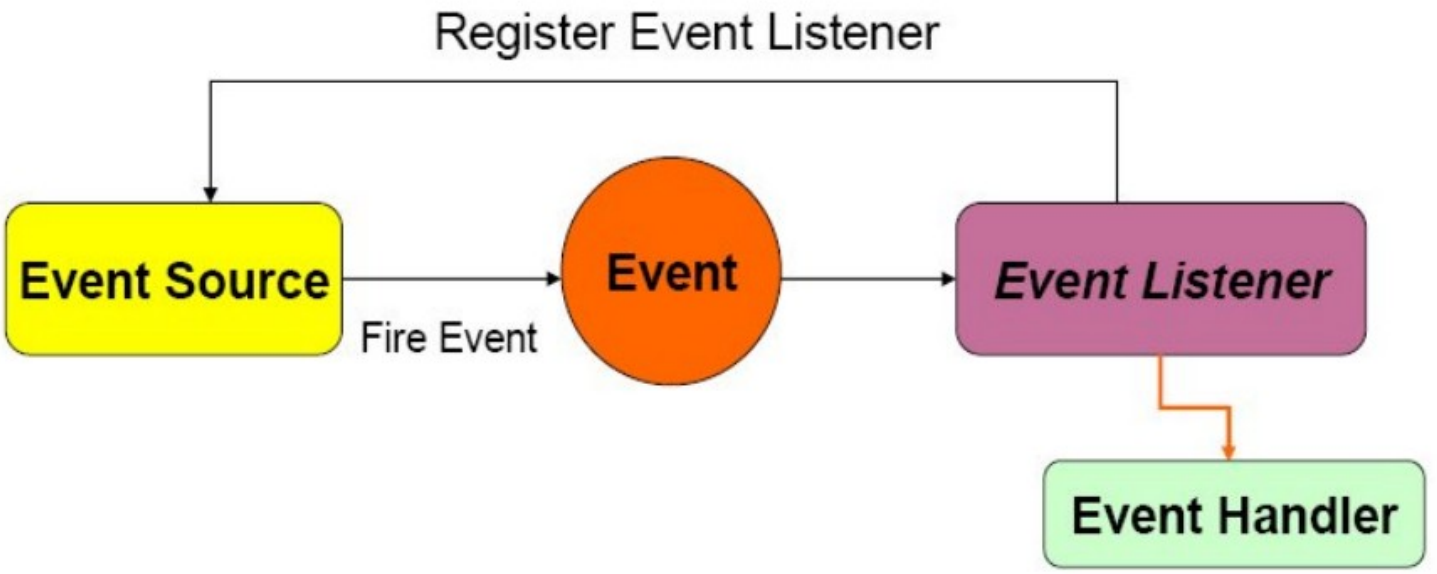
- Binary?
- You're out of luck, Sonny Jim!
- 3rd party
- Download von Oracle
- Kann verwendet werden mit
  - Netbeans
  - Eclipse

Events

- Die Interaktion mit einem GUI geschieht über Events
- Ein Event ist eine, durch einen Benutzer initialisierte Aktion
- Events können auf vielerlei Arten ausgelöst werden, sei es durch Drücken eines Buttons, Verschieben der Maus, Drücken einer Taste ...
- Damit Java auf ein Event reagiert, muss zuerst ein EventListener registriert und ein zugehöriger EventHandler implementiert werden

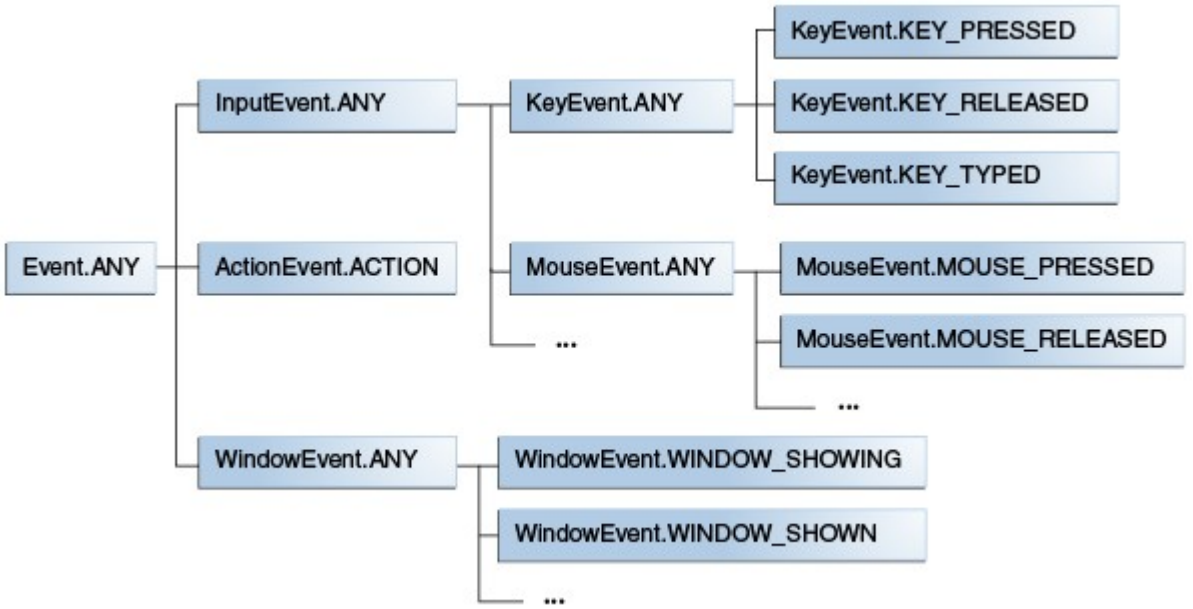
Events

- Für jedes Event wird ein EventListener registriert, welcher die entsprechende(n) EventHandler Methode(n) implementiert



Events

- Für jedes Event wird ein EventListener registriert, welcher die entsprechende(n) EventHandler Methode(n) implementiert



JavaFx Events

- ActionEvent
- InputEvent
- ScrollToEvent
- SortEvent
- MediaErrorEvent
- CheckBoxTreeItem.TreeModificationEvent
- TransformChangedEvent
- WindowEvent
- ...

Eventhandler

```
import javafx.event.*;


Button btn = new Button();
btn.setText("Say 'Hello World'");


btn.setOnAction(new EventHandler<ActionEvent>()
{
    @Override
    public void handle(ActionEvent event)
   {
        System.out.println("Hello World!");
    }
});
```

Lambda Ausdrucks

- Lambdas stammen von funktionalen Sprachen
- Anonyme Methode
- Code as Data
- Prägnante Syntax, weniger Code, lesbarer
- ad-hoc Implementierung von Funktionalität

lambda = ArgList "->" Body

(int x, int y) -> { return x+y; }

ArgList = Identifier

| "(" Identifier ["," Identifier]* ")"

| "(" Type Identifier ["," Type Identifier]* ")“

Body = Expression

| "{" [Statement ";"]+ "}"

Eventhandler

- Eventhandler in Form eines Lambda Ausdrucks

```
btn.setOnAction(event
        ->System.out.println("Hello World!"));


void myAction(ActionEvent e)
{
  System.out.println("Hello World!");
}
. . .
btn.setOnAction(e->myAction(e));
```

Eventhandler

- Eventhandler in Form einer Method-Reference

```
void myAction(ActionEvent e){System.out.println("Hello World!");}
. . .
btn.setOnAction(this::myAction);
```

- Lambda Ausdruck scheint etwas unpassend formuliert zu sein, es folgt eine Anweisung, sogar ein Block kann folgen.