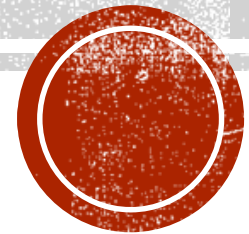


NEUES IN JAVA 8



Sprache

JDK





Sprache

JDK





Lambdas



Standard-
methoden

Sprache

JDK





Lambdas



Standard-
methoden



Funktionale
Interfaces

Sprache

JDK





Lambdas



Standard-
methoden



Funktionale
Interfaces



Annotationen

Sprache

JDK





Lambdas



Standard-
methoden



Funktionale
Interfaces



Annotationen

Sprache



Stream API

JDK





Lambdas



Standard-
methoden



Funktionale
Interfaces



Annotationen

Sprache



Stream API

javaFx

JavaFX 8

JDK





Lambdas



Standard-
methoden



Funktionale
Interfaces



Annotationen

Sprache



Stream API

javaFx

JavaFX 8



Nashorn

JDK





Lambdas



Standard-
methoden



Funktionale
Interfaces



Annotationen

Sprache



Stream API

javaFx

JavaFX 8



Nashorn



DateTime API

JDK





Lambdas



Standard-
methoden



Funktionale
Interfaces



Annotationen

Sprache



Stream API

javaFx

JavaFX 8



Nashorn



DateTime API



Weitere

JDK



URSPRUNG DER LAMBDA AUSDRÜCKE

- Abstraktion
 - Nur über Daten, nicht über Verhalten
- Parallelisierung
 - Komplexitäten von „Thread-Sicherheit“
 - Komplexe API zur Erstellung von Threads
 - Stichwort: volatile



URSPRUNG DER LAMBDA AUSDRÜCKE

- Datenverarbeitung
 - Redundante Datenverarbeitungsprozesse
 - Fehlendes „Lazy“ –Prinzip
 - viele Schleifen und hohes Ansammeln nicht gesuchter Daten
- Lambda Ausdrücke bezeichnen anonyme Funktionen
 - bereits in 1930 Jahren in der Mathematik entdeckt



URSPRUNG DER LAMBDA AUSDRÜCKE

- Funktionale Programmiersprachen
 - Lisp, Haskell -> echt funktional
 - haben sich für Parallelisierung als nützlich erwiesen
 - darum haben einige objektorientierte Sprachen Prinzipien daraus übernommen
 - C++, C#, Scala
 - Java 8 reiht sich nur ebenfalls ein



LAMBDA IN JAVA 8



- Idee
 - Definition eines Blocks von Code zur späteren Ausführung
 - Früher das Konzept von **Runnable**, **EventHandler**

Interface(Object)

Consumer <String> c = (String text) -> System.out.println(text);

Codeabschnitt

Variablen

- Anwendungsfall
 - „Event-Driven“ Programming (Swing/JFX Events)
 - „Reactive“ Programming
- Trennung von Definierung Wie von Was
 - weniger Fehleranfällig



LAMBDA IN JAVA 8

- Verkürzung der Lambda-Syntax
 - Typen sind optional

```
(text) -> System.out.println(text);
```

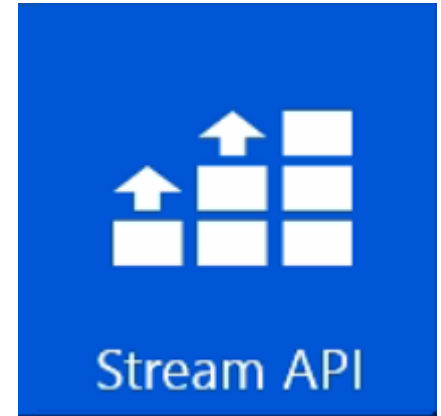
- Methodenaufruf
 - :: um die Methode zu adressieren
 - Variablenangabe überflüssig

```
Consumer<String> c = System.out::println;
```



STREAM API

- Stream -> sequenzieller Ablauf on Objekten in der Eingabe
- Bsp: Verarbeitung von Collections
 - Finden einer Person in einer Liste

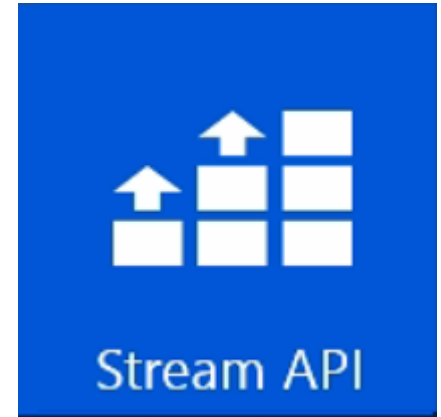


```
List<Person> personenListe = ...
personenList
    .stream()
    .limit(1)
    .filter( p -> p.getCountry().equals(„Germany“))
        lambda Ausdruck für Kriterien
    .forEach( p -> {System.out.println(p);
    });
```



STREAM API

- Stream
 - Speichert Elemente nicht zusätzlich
 - „On Demand“ – Berechnung = Lazy Evaluation
- Besonders geeignet für Datenbankabfragen
 - Spring JDBC, Apache DbUtils



DATE/TIME API

- Repräsentation **ISO 8601** Standard angepasst
- An **JSR 310 (JodaTime)** orientiert
- Intuitivere Benutzung
- Einfachere Konvertierung zwischen Datumsangaben und Maschinenzeit
 - oder zu alten Formaten < Java 8
- Nicht mehr den Millisekunden seit 1970 nachempfunden
- Neu → **ZonedDateTime** – repräsentiert Zeit in einer anderen Zeitzone
- **Instant** entspricht der Maschinenzeit
- nun auch „Nano“ als Wert möglich



DATE/TIME API



`LocalDate.now()` = Aktuelles Datum

`LocalTime.now()` = Aktuelle Zeit

`LocalDateTime.now()` = Aktuelle Zeit + Datum

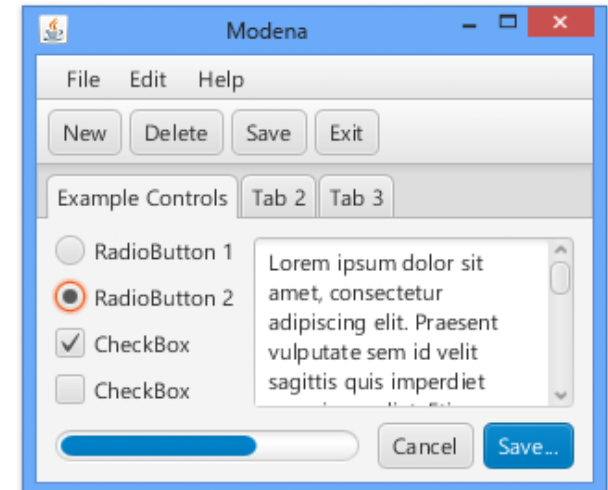
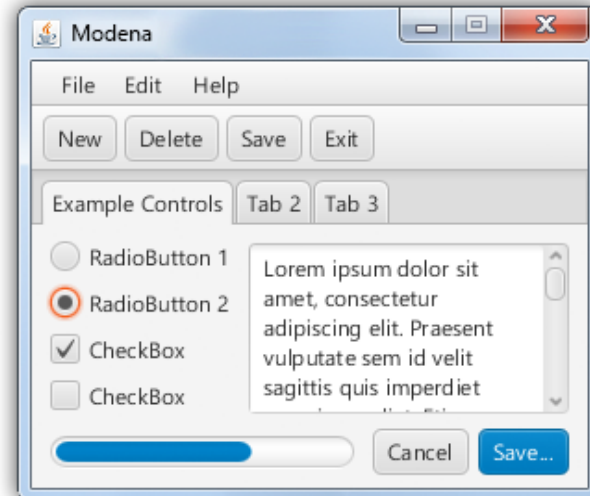
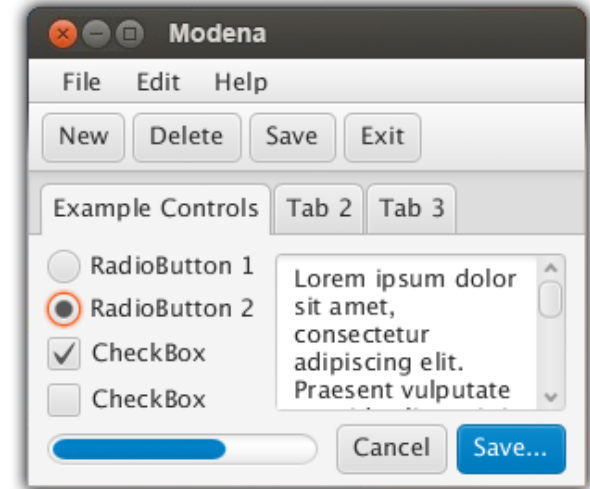
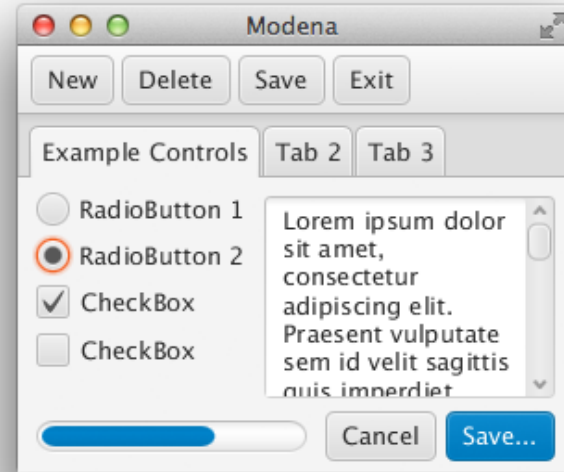
`Instant.now()` = Aktuelle Maschinenzeit

`LocalDate.of(...)` = Bestimmte Zeitpunkte aufbauen



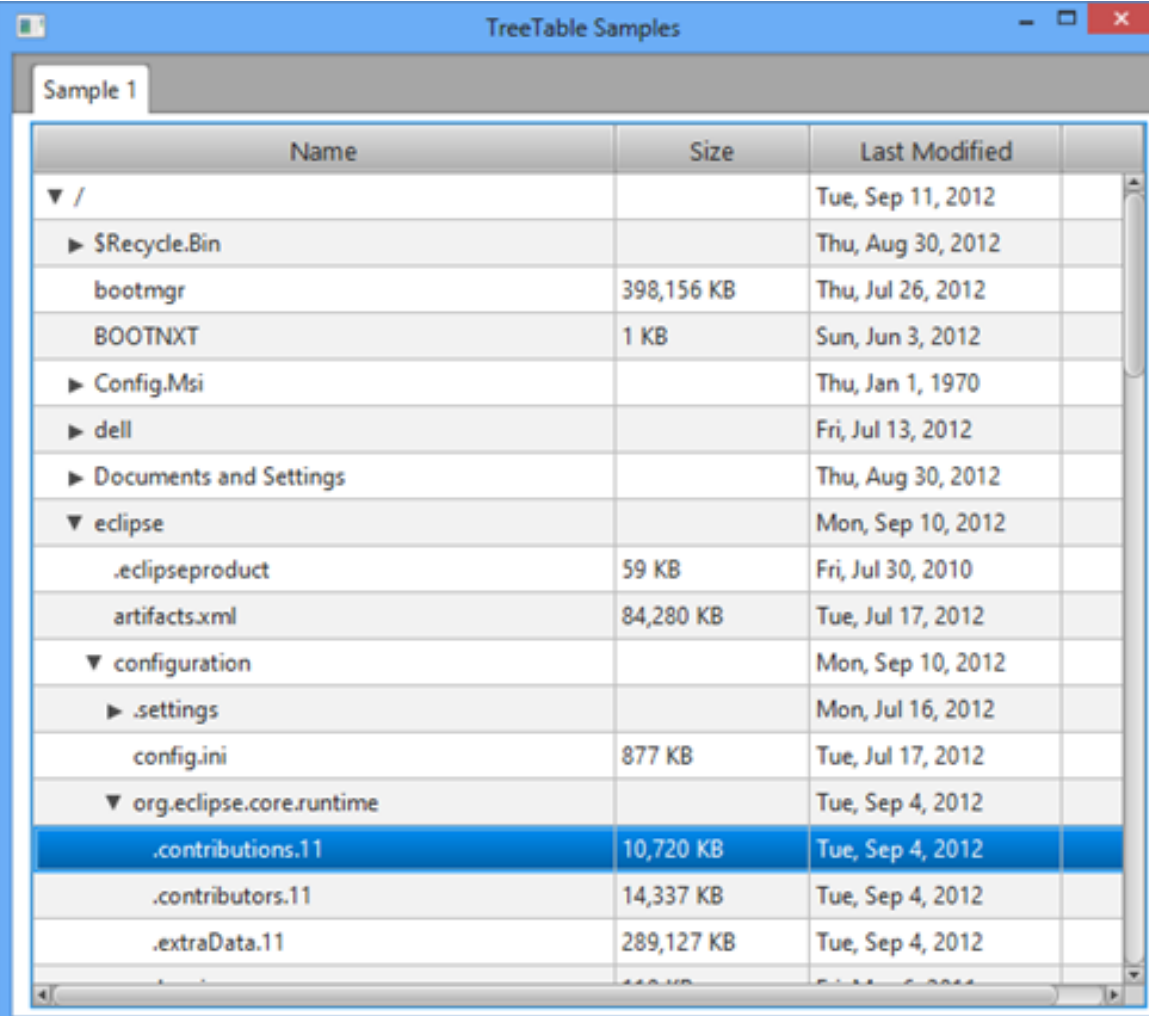
JAVA FX 8

■ Modena-Stil



JAVA FX 8

- **Neue Komponenten**
 - TreeTableView



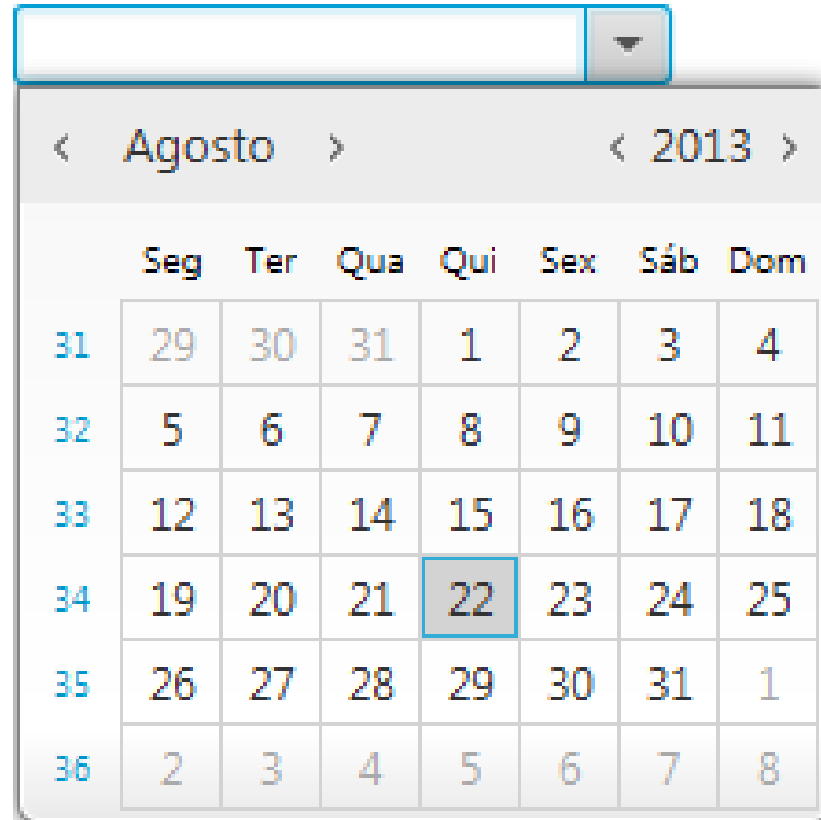
| Name | Size | Last Modified |
|----------------------------|------------|-------------------|
| ▼ / | | Tue, Sep 11, 2012 |
| ▶ \$Recycle.Bin | | Thu, Aug 30, 2012 |
| bootmgr | 398,156 KB | Thu, Jul 26, 2012 |
| BOOTNXT | 1 KB | Sun, Jun 3, 2012 |
| ▶ Config.Msi | | Thu, Jan 1, 1970 |
| ▶ dell | | Fri, Jul 13, 2012 |
| ▶ Documents and Settings | | Thu, Aug 30, 2012 |
| ▼ eclipse | | Mon, Sep 10, 2012 |
| .eclipseproduct | 59 KB | Fri, Jul 30, 2010 |
| artifacts.xml | 84,280 KB | Tue, Jul 17, 2012 |
| ▼ configuration | | Mon, Sep 10, 2012 |
| ▶ .settings | | Mon, Jul 16, 2012 |
| config.ini | 877 KB | Tue, Jul 17, 2012 |
| ▼ org.eclipse.core.runtime | | Tue, Sep 4, 2012 |
| .contributions.11 | 10,720 KB | Tue, Sep 4, 2012 |
| .contributors.11 | 14,337 KB | Tue, Sep 4, 2012 |
| .extraData.11 | 289,127 KB | Tue, Sep 4, 2012 |



JAVA FX 8

- **Neue Komponenten**

- DatePicker
- Mit der neuen Date API kompatibel



JAVA FX 8

- **Neue Komponenten**
- Drucken
- JavaFX 3D-Verbesserungen
 - 3D Primitive
 - `javafx.scene.shape.Box`
 - `javafx.scene.shape.Cylinder`
 - `Javafx.scene.shape.Sphere`
- Touch-Verbesserungen
- Embedded-/ARM-Unterstützung (HTML 5)
- `SwingNode` < - > `JFXPanel` (Swing in JFX)
- nun Vollkommen Open Source



PROJECT NASHORN

- JavaScript Engine
- Entstanden aus Rhino
- Entwickelt zu 100% in Java
- Java-Interoperabilität
- Java Library **jjs** erlaubt Ausführen von Javascript in der Console



PROJECT NASHORN



```
ScriptEngineManager scriptManager = new ScriptEngineManager();  
nashornEngine = scriptManager.getEngineByName(„nashorn“);  
  
scriptResult = nashornEngine.eval(Files.newBufferedReader(„Script.js“));  
inlineScriptResult = nashornEngine.eval(„return ,Print Me!“);
```



PROJECT NASHORN

- Java Integration – Methodenaufruf

JavaScript -> Java:

```
var Klasse = Java.type(„com.package.Klasse“);  
Klasse.aufruf(„Test“);
```

Java -> JavaScript:

```
Invocable inv = (Invocable)nashornEngine;  
Inv.invokeFunction(„printText“, „Argument1“, „Argument2“, new Date());
```



OPTIONAL

- Umgehen der Null-Problematik
- Findet dort Anwendung wo Ergebnisse nicht garantiert werden

```
Optional<String> doSomething(){  
    try{  
        String txt = //...  
    } catch (IOException ex){  
        return Optional.empty();  
    }  
    return Optional.of(txt);  
}
```



OPTIONAL

```
Optional<String> result = doSomething();  
if(result.isPresent())  
    result.get();  
}
```

```
//Standardwert  
String result = doSomething().orElse(„Not existing“);
```

```
//Lambda Ausdruck  
doSomething().isPresent(System.out::println);
```

```
Optional<String> doSomething(){  
    try{  
        String txt = //...  
    } catch (IOException ex){  
        return Optional.empty();  
    }  
    return Optional.of(txt);  
}
```



VIELEN DANK FÜR DIE AUFMERKSAMKEIT



FRAGEN?

