1.– 4. September 2014
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Im Schatten der Lambdas

Weitere interessante Neuerungen in Java 8

Ralph Henze

Sparda-Datenverarbeitung eG

Java 8 ist da!

Lambda-Ausdrücke, Methoden-Referenzen und Streams

- Wesentliche Neuerungen der Sprache Java
- Elemente der Funktionalen Programmierung halten Einzug in Java.
- Lange erwartet … endlich zieht Java mit anderen Sprachen (z. B. C#, C++11) gleich!

Image courtesy of Ambro at FreeDigitalPhotos.net

Java 8 ist da!

Aber Java 8 bringt eine Reihe weiterer Neuerungen mit sich.

- Viele spielen sich auf Ebene der Klassenbibliothek ab.
- Es gibt jedoch auch weitere Neuerungen an JVM und Sprache.

Image courtesy of imagerymajestic at FreeDigitalPhotos.net

Überblick von Oracle:

Überblick: JSRs in Java 8

- Neue Funktionalität:
  - JSR 308: Annotations on Java Types
  - JSR 310: Date and Time API
  - JSR 335: Lambda Expressions
- Updates bestehender Funktionalität:
  - JSR 114: JDBC Rowsets
  - JSR 160: JMX Remote API
  - JSR 199: Java Compiler API
  - JSR 173: Streaming API for XML Processing (StAX)
  - JSR 206: Java API for XML Processing (JAXP)
  - JSR 221: JDBC 4.0
  - JSR 269: Pluggable Annotation Processing API

Themen dieses Vortrags

- Klassenbibliothek
  - Neue Date &Time API (JSR 310)
  - Optionals (`java.util.Optional<T>`)
  - Concurrency Utilities
  - Nette Kleinigkeiten
- Programmiersprache
  - Erweiterte Interfaces
  - Annotationen auf Typen
- Java VM / Java Runtime
  - Compact Profiles
  - Wegfall der Permanent Generation

Neue Date & Time API (JSR 310)

Image courtesy of winnond at FreeDigitalPhotos.net

```
java.time.*
```

Date & Time API (JSR 310)

Warum brauchen wir eine neue API für Datum und Zeit?

- Die bisherige API (java.util.Date, Calendar, etc.) ist problematisch und krankt an mangelhaftem Design:
  - Nicht Thread-sicher; Mutable.
  - Namensgebung: Date ist kein Datum, sondern ein Zeitpunkt; Calendar ist eine Mixtur aus Datum und Zeit; usw.
  - API-Inkonsistenzen: Monate starten bei 0, Tage bei 1 und Jahre bei 1900.
  - Eine Reihe von Dingen sind mit dieser API nicht möglich, z. B. Datum ohne Minuten und Sek.

Design-Prinzipien der neuen Date & Time-API

- Alle Klassen sind immutable
  (und damit automatisch Thread-sicher).
- Fluent API
  Leicht zu lesen und zu verstehen.
- Domain-Driven Design
  Ausrichtung an der natürlichen Vorstellung von Datum und Zeit bzw. Zeitpunkten im Kontinuum.
- Erweiterbarkeit
  API-Schichten (High Level und Low Level APIs)
  z. B. Unterstützung unterschiedlicher Chronologien
  Neben dem Standardfall des ISO-8601-Kalenders werden auch andere Kalendersysteme/Zeitrechnungen unterstützt.

Einfaches Datum: LocalDate

- Repräsentiert ein Datum/einen bestimmten Tag (aber keinen bestimmten Zeitpunkt an diesem Tag).
- Speichert Jahr, Monat, Tag.
- Z. B. Urlaubstage, Feiertage, etc.

```
LocalDate date =
    LocalDate.of(2014, Month.SEPTEMBER, 3);
date = LocalDate.now();

boolean isLeap = date.isLeapYear();
int monthLength = date.lengthOfMonth();
date = date.plusMonths(3).minusDays(1);
date = date.with(Month.OCTOBER);
```

Einfache Uhrzeit: LocalTime

- Repräsentiert eine bestimmte Uhrzeit (aber an keinem bestimmten Tag).
- Speichert Stunde, Minute, Sekunde, Nanosekunde.
- Z. B. Öffnungszeiten, etc.

```
LocalTime time = LocalTime.now();
time = LocalTime.of(13, 30);

if(time.getHour() >= 12) ...

time = time.plusHours(2).minusMinutes(10);
time = time.truncatedTo(ChronoUnit.SECONDS);
```

Kombination: LocalDateTime

- Repräsentiert eine bestimmte Uhrzeit an einem bestimmten Tag.
- Kombination von LocalDate und LocalTime.
- Speichert Jahr, Monat, Tag, Stunde, Minute, Sekunde, Nanosekunde.
- Z. B. Termine im Terminkalender, etc.

```
LocalDateTime dt = LocalDateTime.now();
dt = LocalDateTime.of(2014, Month.SEPTEMBER, 3,
13, 30);

dt = dt.plusDays(2).minusHours(3).plusNanos(5);
dt = dt.with(next(DayOfWeek.TUESDAY));
```

TemporalAdjuster

- Interface, das dem Zweck dient, Justierungen bzw. Änderungen vorzunehmen.
- **`LocalDate`**, **`LocalTime`**, **`LocalDateTime`**, etc. implementieren das Interface.
- Die Hilfsklasse **`TemporalAdjusters`** enthält eine Reihe vordefinierter TemporalAdjuster:

```
dayOfWeekInMonth()
firstDayOfMonth()
firstDayOfNextMonth()
firstDayOfYear()
lastDayOfMonth()
lastDayOfNextMonth()
lastDayOfYear()
next()
nextOrSame()
previous()
previousOrSame()
```

Unterstützung von Zeitzonen: ZonedDateTime

- Repräsentiert eine bestimmte Uhrzeit an einem bestimmten Tag in einer definierten Zeitzone.
- Erweitert LocalDateTime um eine Zeitzone (d. h. um ZoneId, ZoneOffset und ZoneRule).
- Unterstützt Sommer-/Winterzeit (DST), incl. Gaps und Overlaps.

```
ZoneId zone = ZoneId.of("Europe/Berlin");
ZonedDateTime dt = ZonedDateTime.of(2014,
Month.SEPTEMBER, 3, 13, 30, 0, 0, zone);

ZoneId greenwich = ZoneId.of("UTC");
dt = dt.withZoneSameInstant(greenwich);
```

Zeitpunkt: Instant

- Repräsentiert einen zonenunabhängigen Zeitpunkt/Timestamp.
- Speichert die Anzahl an Nanosekunden seit dem 01.01.1970 GMT („The Epoch“).
- Äquivalent zu java.util.Date.

```
Instant instant1 = Instant.now();
Instant instant2 = Instant.now();

if(instant1.isAfter(instant2)) {...}

long timeValue = instant1.toEpochMilli();
```

Kurze Zeitspanne: Duration

- Repräsentiert einen Zeitmenge mit hoher Genauigkeit (aber keinen definierten Anfangs- oder Endezeitpunkt).
- Speichert Stunden, Minuten, Sekunden und Nanosekunden.

```
Duration duration = Duration.ofHours(12);
duration = duration.plusMinutes(30);

LocalDateTime dt = LocalDateTime.now();
dt = dt.plus(duration);
```

Längere Zeitspanne: Period

- Repräsentiert eine taggenaue Zeitmenge (aber keinen definierten Anfangs- oder Endezeitpunkt).
- Speichert Jahre, Monate und Tage.

```
Period period = Period.ofMonths(9);
period = period.plusDays(10);

LocalDateTime dt = LocalDateTime.now();
dt = dt.plus(period);
```

Rechnen mit Zeitdifferenzen

```
LocalDate date1 = LocalDate.of(2014, 3, 4);
LocalDate date2 = LocalDate.now();

Period period = Period.between(date1, date2);
```

Weitere konkrete Klassen

- **`Year`** steht für ein Jahr nach ISO-8601, z. B. 2014.
- **`YearMonth`** steht für einen bestimmten Monat eines Jahres, z. B. September 2014.
- **`MonthDay`** steht für einen Tag, unabhängig vom Jahr, z. B. Geburtstag, Jahrestag, …
- **`OffsetDateTime`** steht für ein Datum/Uhrzeit mit Offset, allerdings ohne konkrete Zeitzone.
- **`OffsetTime`** steht für eine Uhrzeit mit Offset, jedoch ohne konkrete Zeitzone.

Formatieren und Parsen

- Aufrufen der toString()-Methoden liefert immer eine ISO-8601 konforme Darstellung.
- **`DateTimeFormatter`** ist die Hauptklasse für die formatierte Ausgabe und das Parsen.
- Es gibt eine Reihe vordefinierte Formatter für ISO-8601, incl. Locales und ZoneIds.

```
DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd.MM.yyyy");

String text = date.toString(formatter);
LocalDate date = LocalDate.parse(text, formatter);
```

Date & Time API: Sonstiges

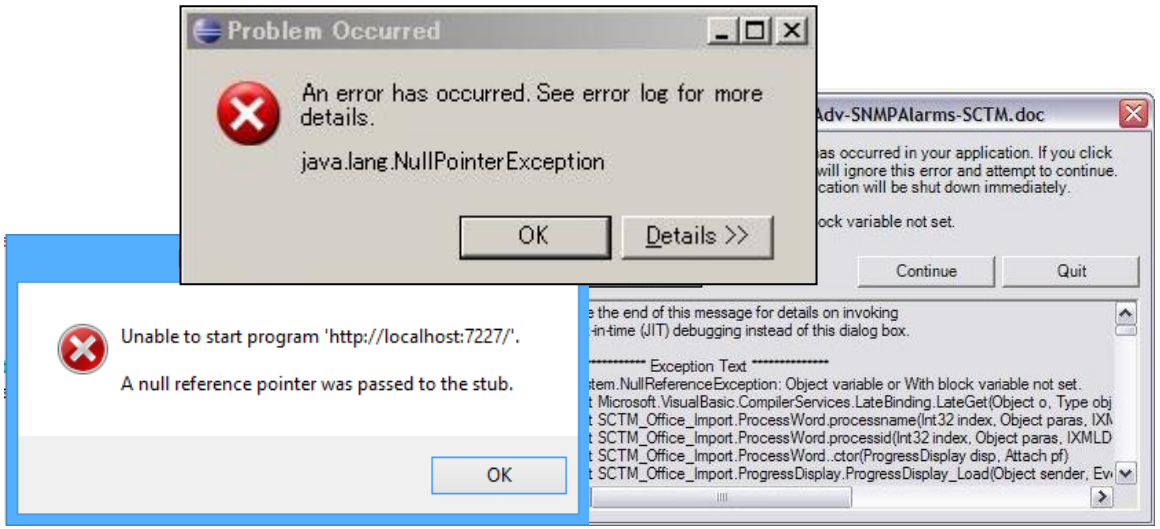
- Unterstützung verschiedener Zeitrechnungen
  - Eingebaut sind neben ISO-8601 auch Japan, Minguo (China), ThaiBuddhist und Hijrah (Islamische Zeitrechnung)
- Low Level API (Package `java.util.time.temporal`)
  - Entwicklung eigener Kalendersysteme/Zeitrechnungen
  - Eigene Date/Time-Klassen
  - Eigene Uhren/Uhrsysteme

Wer das API mit älteren Java-Versionen verwenden möchte:

- Das Open Source-Projekt ThreeTen stellt einen Backport der JSR 310 Referenzimplementierung für Java SE 6 und 7 bereit.
- Package-Name ist **`org.threeten.bp.*`**, ansonsten ist die API identisch mit Date/Time in Java 8.
- Project Lead ist Stephen Colebourne, der auch Spec Lead für JSR-310 und JodaTime ist.
- http://www.threeten.org/threetenbp/

```
<groupId>org.threeten</groupId>
<artifactId>threetenbp</artifactId>
<version>1.0</version>
```

Optionals `java.util.Optional<T>`



Tony Hoare auf der Konferenz QCon 2009:

"**I call it my billion-dollar mistake**. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. **This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.**"

Was ist ein `Optional<T>`?

- Ein Optional ist ein Container-Objekt, das eine Objektreferenz enthalten kann (oder auch nicht).
  - „präsent“ ➔ Das Optional enthält eine Referenz
  - „absent“ ➔ Das Optional enthält keine Referenz
- Optionals werden als Return-Typ von Methoden verwendet, die u. U. keinen Wert zurückliefern.
- Hilft, das Auftreten von unerwarteten NullPointerExceptions vermeiden.
- „Optional-Return“ Idiom.
- Unterscheidung zwischen „unbekannt“ und „bekannt, dass es keinen Wert hat“.
- Optional ist aber nicht Serializable!

Erzeugen eines `Optional`

- Ein leeres Optional (absent) erzeugen:

```
Optional<Integer> opt = Optional.empty();
```

- Ein neues Optional mit non-null Wert (präsent):

```
Optional<Integer> opt = Optional.of(5);
```

- Ein neues Optional, das einen Nullwert tragen kann:

```
Optional<Integer> opt =
    Optional.ofNullable(mightBeNull);
```

Anwendungsfall: Referenztest

Etwas nur tun, falls eine Referenz vorhanden ist.

```
Optional<Integer> opt =
   Optional.ofNullable(mightbeNull);

// alt: mit if()-Konstrukt
if(opt.isPresent()){
   System.out.println(opt.get());
}

// neu: mit Lambda-Ausdruck oder Methoden-Ref.
opt.ifPresent(System.out::println);
```

Anwendungsfall: Default-Wert

Einen Default-Wert zurückgeben, falls absent.

```
Optional<Company> opt = businessMethod();

// Falls present zurueckliefern, sonst Default
Company company = opt.orElse(new Company());

// Falls present zurueckliefern, sonst Exception
Company company =
   opt.orElseThrow(IllegalStateException::new);
```

Anwendungsfall: Prüfen einer Bedingung

Prüfen von Werten mit der Filter-Methode:

- Die Methode filter() nimmt ein Prädikat als Argument und gibt ein Optional zurück.
- Falls ein Wert präsent ist und das Prädikat erfüllt, wird das Optional zurückgeliefert; ansonsten das leere Optional.

```
Optional<Company> opt = businessMethod();
opt.filter(department -> "Finance".equals(department.getName()))
.ifPresent(() -> System.out.println("Finance is present"));
```

Anwendungsfall: Ausführen einer Funktion

Ausführen einer Funktion mit der map()-Methode:

- Falls präsent, wird der Lambda-Ausdruck auf dem Wert ausgeführt.
- Rückgabe ist ein Optional, dessen Wert das Ergebnis des Lambda-Ausdrucks ist (oder das leere Optional, falls das Optional absent war).

```
Optional<String> opt1 = Optional.of("Eine Zeichenkette");
Optional<Integer> opt2 = opt1.map(String::length);
opt2.ifPresent(System.out::println);
```

Wann sind Optionals ungeeignet?

- In Remote-Schnittstellenklassen
  (z. B. Remote EJBs, JAX-WS/JAX-RS Services, …)
- Im Domain Model Layer
- Bei Eingabeparametern
  (auch Konstrukturparameter).
- In DTOs/VOs (nicht Serializable!)

Optionals in Java 6 und 7: Google Guava

- Googles Guava-Library bietet schon seit längerem eine beinahe identische Optional-Klasse an.
- Diese unterstützt keine Lambda-Ausdrücke, sondern das in Guava integrierte FP-Konzept.
-
```
<groupId>com.google.guava</groupId>
<artifactId>guava</artifactId>
<version>18.0</version>
```

Neue Concurrency Utilities

Image courtesy of Piyachok Thawornmat
at FreeDigitalPhotos.net

Concurrent Adders und Accumulators

- Optimierte Klassen für Counter, die von mehreren Threads nebenläufig gelesen und geschrieben werden.
- `java.util.concurrent.atomic.LongAdder`
  `java.util.concurrent.atomic.DoubleAdder`
  `java.util.concurrent.atomic.LongAccumulator`
  `java.util.concurrent.atomic.DoubleAccumulator`
- Bisher gab es verschiedene „Alternativen“:
  - Unsynchronisiert: häufiger Fehler!
  - synchronized-Block: korrekt, ist aber langsam.
  - Volatile: OK, wenn nur ein schreibender Thread vorhanden ist.
  - AtomicInteger/AtomicLong: bislang beste Alternative.

Concurrent Adders und Accumulators API

- Adders: LongAdder und DoubleAdder.
  - Wert zunächst 0, kann dann mit Methoden add(…), increment(), decrement() verändert werden.
  - Der Wert kann mit longValue(), doubleValue() etc. gelesen werden.
  - Erben von java.lang.Number.
- Accumulators: LongAccumulator und DoubleAccumulator.
  - Bei der Instanzierung muss eine seiteneffektfreie Update-Funktion definiert werden.
  - Die Update-Funktion ist als Lambda-Ausdruck formuliert.
  - Danach ruft man zum Update nur noch die Methode void accumulate(long i) auf.
  - Erben von java.lang.Number.

Beispiel: LongAccumulator

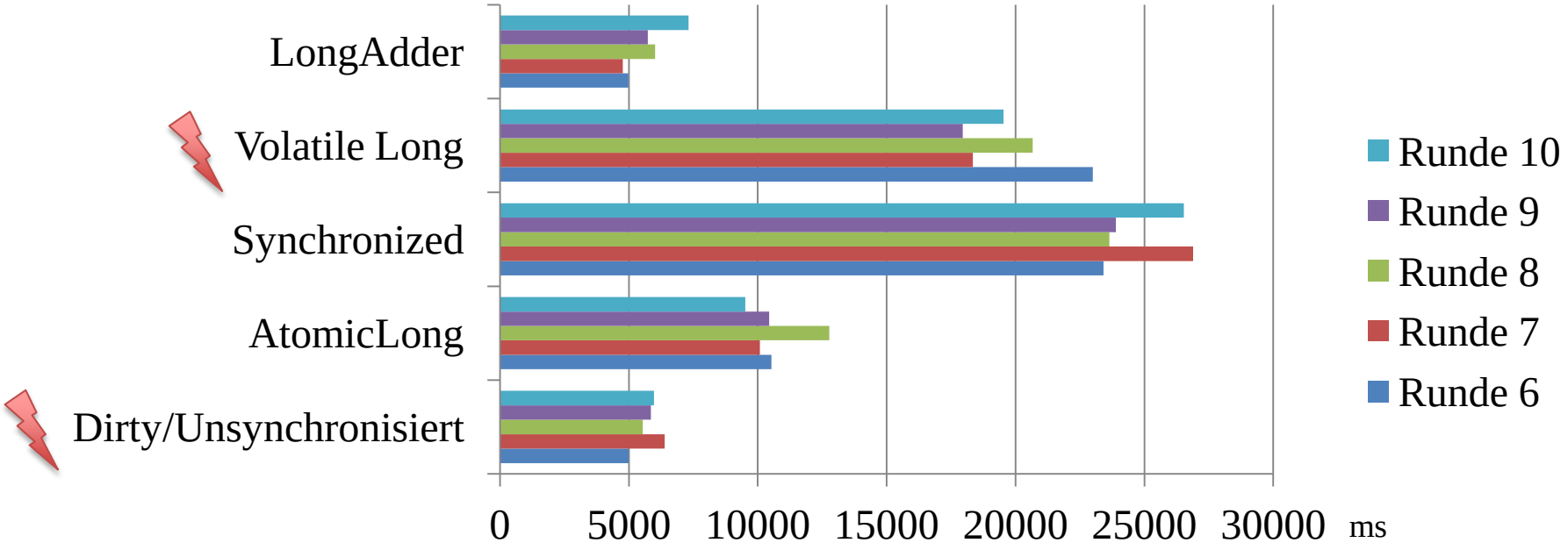
```
LongAccumulator la =
    new LongAccumulator((x, y) -> x + y, 2L);
la.accumulate(40L);
System.out.println("The answer is " + la.get());
```

Ausgabe:

```
The answer is 42
```

Benchmark – Concurrent Counters

- Intel Core2 Quad CPU (Q9550), 2.83 GHz, 4 GB RAM
- 10 Threads, 10 Runden (nur die letzten 5 werden gezählt)
- Win 7 x64, Oracle Java SE 8u11, 64 Bit



https://github.com/takipi/counters-benchmark

Warum sind Concurrent Adders schneller?

- AtomicLong und AtomicInteger haben eine einzige Instanzvariable, auf die mittels CAS zugegriffen wird.
- Concurrent Adders und Accumulators haben intern mehrere Speicherzellen.
- In Situationen mit hoher Nebenläufigkeit, bei denen mehrere Threads konkurrieren, wird einfach eine „freie“ Speicherzelle verwendet.
- Der Wert des Adders/Accumulators ist über mehrere Zellen verteilt und wird erst ausgerechnet, wenn er angefordert wird.
- Die Größe ist dynamisch, aber immer Zweierpotenz.

StampedLock

- Eine in vielen Fällen performantere Alternative zum ReentrantReadWriteLock.
- Das StampedLock ist ein optional optimistisches Read Lock (Write Lock ist immer pessimistisch).
- Optimistisch bedeutet: Es blockiert bei Leseoperationen keine anderen Threads.
- Nach der Leseoperation muss man jedoch nachsehen, ob der Optimismus gerechtfertigt war.
- Gute Lösung in Situationen mit häufigen Reads.
- Fähigkeit, Read Locks in Write Locks zu konvertieren, falls Schreiben erforderlich wird.

Beispiel / Idiom: StampedLock Read

```
final StampedLock lock = new StampedLock();
// ...
long stamp = lock.tryOptimisticRead(); // non blocking
read();

// if a write occurred, try again with a read lock
if(!lock.validate(stamp))
{
   long stamp = lock.readLock();

   try
   {
      read();
   }
   finally
   {
      lock.unlockRead(stamp);
   }
}
```

Nette Kleinigkeiten

Image courtesy of sattva at FreeDigitalPhotos.net

Operationen mit explizitem Overflow

- Arithmetische Overflows in numerischen Operationen waren bislang implizit.
- Nun gibt es in **`java.lang.Math`** und **`java.lang.StrictMath`** Operationen, die im Falle eines Overflow eine ArithmeticException werfen.
- Das kann für höhere numerische Stabilität sorgen.
- **`java.lang.ArithmeticException`** ist eine RuntimeException.
- Der Overhead dafür ist gering: Die Implementierung in Math hat einen zusätzlichen Test, der auf Bit-Operationen beruht.

Operationen mit explizitem Overflow

`java.lang.Math` und `java.lang.StrictMath` bieten in Java 8 folgende Operationen mit explizitem Overflow:

```
• public static int addExact(int x, int y);
  public static long addExact(long x, long y);
• public static int subtractExact(int x, int y);
  public static long subtractExact(long x, long y);
• public static int multiplyExact(int x, int y);
  public static long multiplyExact(long x, long y);
• public static int incrementExact(int a);
  public static long incrementExact(long a);
• public static int decrementExact(int a);
  public static long decrementExact(long a);
• public static int negateExact(int a);
  public static long negateExact(long a);
• public static int toIntExact(long value);
```

Beispiel für garantierten impliziten Overflow:

```
int a = Integer.MAX_VALUE;
a++;
System.out.println(a);
```

- Es wird keine Exception geworfen.
- Ausgabe ist
  `-2147483648`

Beispiel mit Overflow-Operation:

```
int a = Integer.MAX_VALUE;
a = Math.incrementExact(a);
System.out.println(a);
```

```
Exception in thread "main" java.lang.ArithmeticException: integer overflow
  at java.lang.Math.incrementExact(Math.java:909)
```

Base64 Encoding und Decoding

- Java 8 bringt einen Encoder und Decoder für Base64 mit!
- **`java.util.Base64.Encoder`**
  `java.util.Base64.Decoder`
- Es ist nicht mehr nötig, die inoffizielle API **`sun.misc.Base64Encoder`** bzw. **`sun.misc.Base64Decoder`** zu verwenden.
- Es ist nicht mehr nötig, Commons Codec einzubinden, nur um Base64 zu verarbeiten.
- Übrigens: Schon seit Java 6 gibt es eine „offizielle“ API für Base64 Encoding/Decoding: **`javax.xml.bind.DatatypeConverter`**

Image courtesy of stockimages at FreeDigitalPhotos.net

Paralleles Sortieren von Arrays

- Neue Methoden in java.util.Arrays
- Die Klasse java.util.Arrays bietet schon seit Java 1.2 Hilfsmethoden für die Verarbeitung von Arrays an.
- Das wird häufig zum Sortieren von Arrays aus primitiven Typen verwendet.

```
public static void sort(int[] a);
```

- Neu in Java 8: Jetzt gibt es alle 18 Sortiermethoden auch in Varianten mit paralleler Sortierung.

```
public static void parallelSort(int[] a);
```

- Verwendet ein paralleles Mergesort-Verfahren.
- Basiert intern auf dem Fork-Join-Framework aus java.util.concurrent (seit Java 7).

Auswahl eines starken Zufallszahlengenerators

- Neue statische Methode in `java.security.SecureRandom`
- `public static SecureRandom getInstanceStrong()`
  `throws NoSuchAlgorithmException;`
- Liefert den stärksten Zufallszahlengenerator, der auf der Java VM verfügbar ist.

StringJoiner

- Klasse **`java.util.StringJoiner`**
  - ```
    StringJoiner sj = new StringJoiner(":", "[", "]");
    sj.add("George").add("Sally").add("Fred");
    String desiredString = sj.toString();
    // desiredString is: "[George:Sally:Fred]"
    ```
- Neue statische Methode join() in **`java.lang.String`**
  - ```
    public static String join(CharSequence delimiter,
    CharSequence... elements);
    ```
  - ```
    String message = String.join("-", "Java", "is", "cool");
    // message returned is: "Java-is-cool"
    ```
    ```

Java Programmiersprache

Image courtesy of sippakorn at FreeDigitalPhotos.net

Erweiterte Interfaces

- Interfaces können in Java 8 nicht nur Methodensignaturen und Konstanten enthalten, sondern auch
  - nicht-abstrakte Default-Methoden und
  - statische Methoden.
- Damit können Interfaces erstmals auch Verhalten tragen (aber keinen Zustand!).
- Implementierende Klassen müssen Default-Methoden nicht überschreiben/implementieren.
- Mehrfachvererbung von Logik ist nun möglich.
- Gut zur Erweiterung bestehender Interfaces z. B. in Libraries!

Default-Methoden in Interfaces

```
public interface MyInterface
{
    default public void foo()
    {
        // …
    }
}
```

- Das Schlüsselwort **`default`** leitet eine Default-Methode ein.

Statische Methoden in Interfaces

```
public interface MyInterface
{
    public static void bar()
    {
        // ...
    }
}
```

- Wichtig: Statische Methoden werden nicht an implementierende Klassen vererbt!
- Die Verwendung der statischen Methode ist nur mittels **`MyInterface.bar()`** möglich.

Annotationen auf Typen

- Bislang konnten Annotationen in Java nur auf Deklarationen gesetzt werden.
- In Java 8 können Annotationen auch überall genutzt werden, wo Typen verwendet werden.
- Ziel ist es, weitere Möglichkeiten zur stärkeren Typprüfung in Frameworks oder Compilern zu schaffen.
- Beispiele: Bei Kontruktor-Ausdrücken, Type Casts, `implements`, `extends`, `throws`, ...

Beispiele: Typ-Annotationen

- `new @Interned MyObject();`
- `myString = (@NonNull String) str;`
- `class UnmodifiableList<T> implements`
  `@Readonly List<@Readonly T> { ... }`
- `void monitorTemperature() throws`
  `@Critical TemperatureException { ... }`

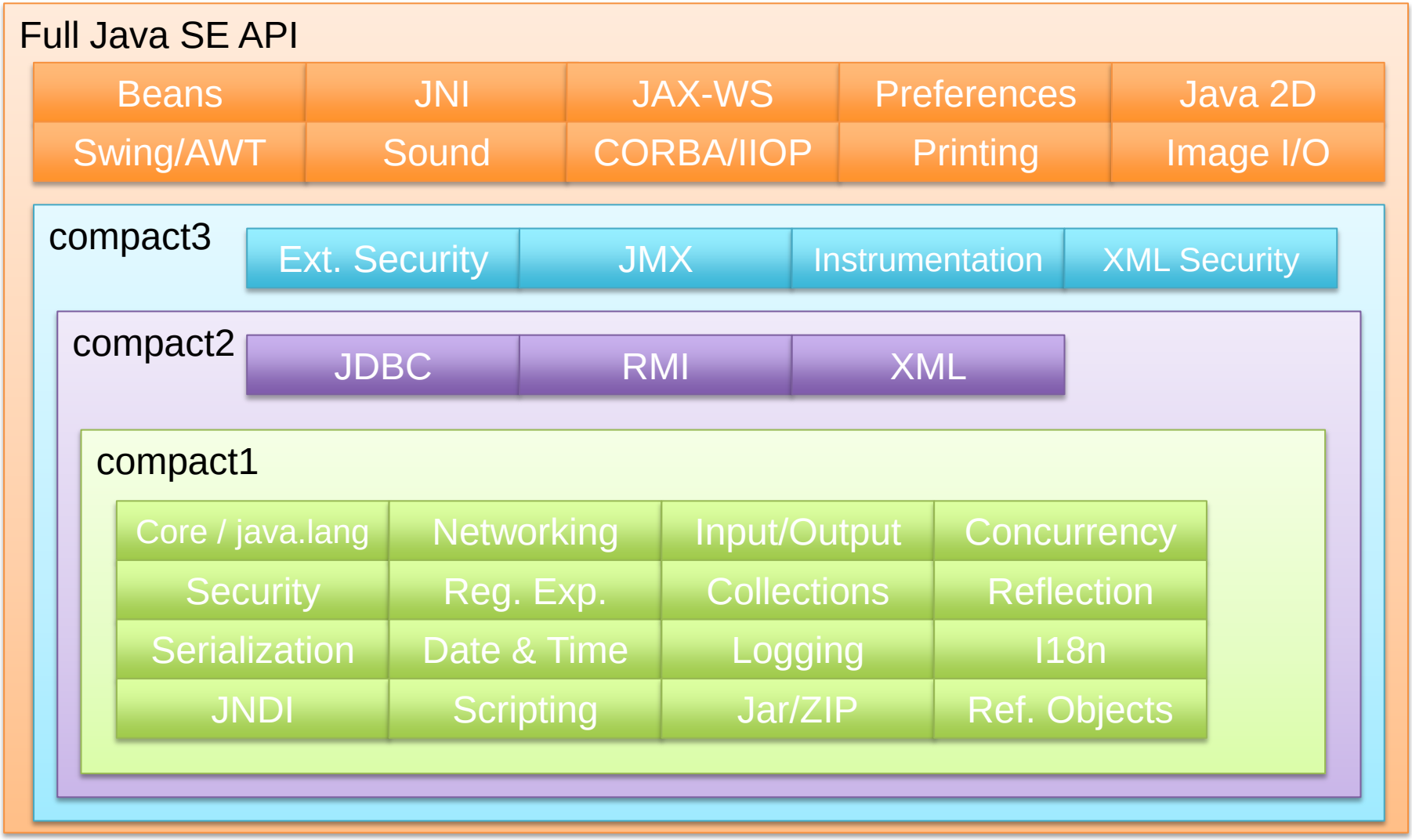
Java VM / Java Runtime

Image courtesy of sippakorn at FreeDigitalPhotos.net

Compact Profiles

- Das Java 8 Runtime definiert drei verschiedene Profile (compact1, compact2 und compact3), die additive Package-Mengen umfassen.
- Idee:
  - Geringerer Umfang ←→ geringerer Memory Footprint
  - Schnellerer Startup der JVM
- Profiles können als Vorbereitung auf die Modularisierung von Java (Project Jigsaw) in Java 9 verstanden werden.

Umfang der Compact Profiles



Memory Footprint der Profile

- compact1 ca. 10 MB
- compact2 ca. 17 MB
- compact3 ca. 24 MB
- Full JRE ca. 140 MB

Toolunterstützung für Compact Profiles

- Java Compiler
  javac –profile [compact1|compact2|compact3]

```
javac -profile compact1 Test.java
Test.java:2: error: Point is not available in
profile 'compact1'
```

- Neues Tool zur Analyse von Klassenabhängigkeiten
  jdeps –profile

```
jdeps -profile Test2.class
Test2.class -> C:\jdk1.8.0\jre\lib\rt.jar (compact1)
   <unnamed> (Test2.class)
      -> java.io                    compact1
      -> java.lang                  compact1
```

Java Runtimes (JREs) für Compact Profiles

- OpenJDK
  - Compact Profile JREs müssen von Hand kompiliert werden.
  - Es gibt ein make target „profiles“. Das startet einen Build, der JRE Images für alle drei Profile erzeugt.
  - Das funktioniert nur für Linux/x86.
- Java SE Embedded (aka „EJDK“)
  - Tool jrecreate.sh –profile [compact1|compact2|compact3]
  - Erzeugt ein Custom JRE, das für die Anwendung und die Plattform optimiert ist.
  - Gibt es derzeit für Linux auf ARM, PowerPC und x86.
  - Download erfordert Oracle Account.

JavaScript Engine Nashorn

- Nashorn ist eine Java-basierte Runtime für JavaScript.
- Compiliert Java-Bytecode aus JavaScript. „invokedynamics are everywhere“
- Bis zu 10-mal bessere Performance gegenüber der bisherigen JavaScript Engine Rhino.
- 100 % ECMAScript-262 5.1-konform.
- Neue Werkzeuge für die Kommandozeile:
  - jjs – Nashorn JavaScript Console
  - jrunscript – Script Runner
- Unterstützt JavaFX-Anwendungen in JavaScript.

Wegfall der Permanent Generation

- Die Oracle Hotspot JVM für Java 8 hat keine Permanent Generation (PermGen) mehr.
- Stattdessen werden die bisher darin gehaltenen Objekte (Klassenmetadaten, statische Variablen) auf dem Heap bzw. in einem neuen Bereich des Native Memory, dem sog. Metaspace, gehalten.
- Das ist ein Nebenprodukt der Konvergenz von HotSpot und JRockit JVMs.
- Nie mehr

```
java.lang.OutOfMemoryError: PermGen space
```

Vielen Dank!

Ralph Henze
Sparda-Datenverarbeitung eG

Sparda-Datenverarbeitung eG

- Wir sind der IT-Dienstleister der 12 Sparda-Banken und der netbank und decken deren gesamten zentralen IT-Bedarf ab.
- Wir sind Spezialisten für Anwendungen wie Corebanking, Zahlungsverkehr, Vertriebslösungen, Baufinanzierung und IT-technische Lösungen wie Virtualisierung und Automatisierung.
- Mit unseren hochmodernen Rechenzentren und innovativen IT-Lösungen sorgen wir für höchste Sicherheitsstandards sowie eine ständige Verfügbarkeit und somit für verlässliche IT im Bankwesen.

Lambda-Ausdrücke und Methodenreferenzen

- Per Definition nicht Thema dieses Vortrags …