2.– 5. September 2013
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Schwarze 8

Änderungen in Java 8

Michael Wiedeking

MATHEMA Software GmbH

JDK 8 Features openjdk.java.net/projects/jdk8/features

- --/--
- vm/--
- vm/comp
- vm/gc
- vm/rt
- core/--
- core/i18n
- core/lang
- core/libs
- core/sec

vm/*

- vm/--
  - 142 Reduce Cache Contention on Specified Fields
- vm/comp
  - 165 Compiler Control
- vm/gc
  - 122 Remove the Permanent Generation
  - 173 Retire Some Rarely-Used GC Combinations
- vm/rt
  - 136 Enhanced Verification Errors
  - 143 Improve Contended Locking
  - 147 Reduce Class Metadata Footprint
  - 148 Small VM
  - 171 Fence Intrinsics

core/--, core/i18n

- core/--
  - 153 Launch JavaFX Applications
- core/i18n
  - 127 Improve Locale Data Packaging and Adopt Unicode CLDR Data
  - 128 BCP 47 Locale Matching
  - 133 Unicode 6.2

core/lang

- 101 Generalized Target-Type Inference
- 104 Annotations on Java Types
- 105 DocTree API
- 106 Add Javadoc to javax.tools
- 117 Remove the Annotation-Processing Tool (apt)
- 118 Access to Parameter Names at Runtime
- 120 Repeating Annotations
- 139 Enhance javac to Improve Build Speed
- 172 DocLint

core/libs

- 113 MS-SFU Kerberos 5 Extensions
- 114 TLS Server Name Indication (SNI) Extension
- 115 AEAD CipherSuites
- 121 Stronger Algorithms for Password-Based Encryption
- 123 Configurable Secure Random-Number Generation
- 124 Enhance the Certificate Revocation-Checking API
- 129 NSA Suite B Cryptographic Algorithms
- 130 SHA-224 Message Digests
- 131 PKCS#11 Crypto Provider for 64-bit Windows
- 140 Limited doPrivileged
- 166 Overhaul JKS-JCEKS-PKCS12 Keystores

101: Generalized Target-Type Inference

Generalized Target-Type Inference

```
class List<E> {
   static <Z> List<Z> nil() { ... };
   static <Z> List<Z> cons(Z head, List<Z> tail) { ... };
   E head() { ... }
}
List<String> ls = List.nil();
List.cons(42, List.<Integer>nil());
```

- List.cons(42, List.nil());

```
String s = List.<String>nil().head();
```

- String s = List.nil().head();

120: Repeating Annotations

Wiederholung von Annotationen

```
public @interface Composer {
   String value() default "";
}
```

```
@Composer("F. Mendelssohn Bartholdy")
class Hochzeitsmarsch { … }
```

```
@Composer("F. Mendelssohn Bartholdy")
@Composer("R. Wagner")
class Hochzeitsmarsch { … }
```

Wiederholung von Annotationen

```
public @interface Composers {
    Composer[] value();
}
```

```
@Comporsers({
    @Composer("F. Mendelssohn Bartholdy"),
    @Composer("R. Wagner")
})
class Hochzeitsmarsch { … }
```

Wiederholung von Annotationen

```
public @interface Composers {
   Composer[] value();
}
```

```
import java.lang.annotation.Repeatable;

@Repeatable(Composers.class)
public @interface Composer {
   String value();
}
```

Wiederholung von Annotationen

- Alle anderen Methoden müssen ein *default* haben
- Keine Zyklen
- RetentionType muss verträglich sein (R(A) ⊆ R(AC))

```
@Retention(RetentionType.RUNTIME) // AC
public @interface Composers {
   Composer[] value();
}
```

```
@Retention(RetentionType.RUNTIME) // A
@Repeatable(Composers.class)
public @interface Composer {
   String value();
}
```

- Verträglichkeit bzgl. @Documented (=) und @Inherited (⊆)

Entsprechende *Reflection*-Methoden

Für <*A* **extends** Annotation> gibt es …

… Annotationen, die für die Klasse gelten:

- *A* getAnnotation(Class<*A*> *annotationClass*)
- *A*[] getAnnotations()
- *A*[] getAnnotationsByType(Class<*A*> *annotationClass*)

… Annotationen, die direkt an der Klasse angegeben wurden

- *A* getDeclaredAnnotation(Class<*A*> *annotationClass*)
- *A*[] getDeclaredAnnotations()
- *A*[] getDeclaredAnnotationsByType(Class<*A*> *annotationC*)

150 Date & Time API

ISO-8601 Kalendersystem – Datum

- Year
  Jahr: “2013”
- YearMonth
  Monat eines Jahres (ohne Tag): “2013-07”
- Month (enum)
  Monat: “July”
- MonthDay
  Tag eines Monats: “--07-04”
- DayOfWeek (enum)
  Wochentag: “Thursday”
- LocalDate
  Datum (ohne Zeit): “2013-07-04”

ISO-8601 Kalendersystem – Zeit

- LocalTime
  Eine Zeit (ohne Datum): “09:50:00”
- OffsetTime
  Ein Zeitpunkt (mit Zeitzone): “12:30:00+01:00”
- LocalDateTime
  Ein Zeitpunkt (ohne Zeitzone): “2013-07-04T09:50:00”
- OffsetDateTime
  Ein Zeitpunkt (mit Zeitzone):
  “2013-07-04T09:50:00+02:00”
- ZonedDateTime
  Ein Zeitpunkt mit Zeitzonen-Id:
  “2013-07-04T09:50:00+02:00 Europe/Berlin”

ISO-8601 Kalendersystem – Zeitpunkt

- Instant
  Zeitpunkt in UTC (repräsentiert durch Sekunden seit “1970-01-01T00:00:00Z” mit einer Auflösung in Nanosekunden):
  “2013-07-04T09:50:00Z”
- ZoneId
  Zeitzone: “Europe/Berlin”
- ZoneOffset
  Zeitversatz (einer Zeitzone): “+02:00”
- Clock
  Liefert den aktuellen Zeitpunkt

ISO-8601 Kalendersystem – Zeitdauer

- Duration

  (Zeit-baiserte) Dauer:

  “23.5 seconds”

- Period

  (Datums-basierte) Dauer:

  “2 years, 3 months and 7 days”

126: Lambda Expressions and Virtual Extension Methods

Funktionale Schnittstellen (Prä-Java-8-Ära)

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

```
public interface Callable<V> {
    V call() throws Exception;
}
```

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

```
public interface Runnable {
    public void run();
}
```

Funktionale Schnittstellen (Java 8)

- **public interface** Predicate<T> {
  boolean test(T *t*);
  }
- **public interface** Function<T, R> {
  R apply(T *t*);
  }
- **public interface** Consumer<T> {
  void sccept(T *t*);
  }

- … und viele mehr, z. B. unter java.util.function

Nutzung funktionaler Schnittstellen

- Benannte Klasse

```
public class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        …
    }
}

ActionListener a = new MyActionListener();
```

Nutzung funktionaler Schnittstellen

- Anonyme Klasse

```
ActionListener a =  new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        …
    }
};
```

- Lambda-Ausdruck

```
ActionListener a = (ActionEvent e) -> … ;
ActionListener a = (ActionEvent e) -> { … };
ActionListener a = (e) -> …;
```

Lambda-Ausdrücke

Lambda-Ausdruck ::= *Parameter-Liste* -> *Rumpf*

- *Parameter-Liste*
  - Keine oder mehrere mit “(” und “)” geklammerte Parameter
  - Typ kann explizit angegeben oder inferiert werden
  - Bei einer Liste mit genau einem Parameter mit inferiertem Typ kann auf die Klammerung verzichtet werden
- *Rumpf*
  - Keine oder mehrere mit “{” und “}” geklammerte Anweisungen
  - Typ des Lambda-Ausdrucks ist der des *return*-Werts
  - Bei genau einer Anweisung ist keine Klammerung nötig; Typ des Lambda-Ausdrucks ist der dieser Anweisung

- *x* -> 2 * *x*
- *x* -> (2 * *x*)
- (int *x*) -> 2 * *x*
- (int *x*) -> (2 * *x*)
- *x* -> {
  return 2 * *x*;
  }
- (*x*, *y*) -> *x* + *y*
- (int *x*, long *y*, double *z*) -> {
  return *x* * *y* * *z*;
  }
- (int *n*) -> {
  long *p* = 1;
  for (int *i* = 1; *i* < *n*; *i*++) {
  p *= *i*;
  }
  return *p*;
  }
- () -> *out*.println("Hi!")
- () -> 3
- (*x*) -> 3

Typen von Lambda-Ausdrücken

- Funktionale Schnittstellen
  - siehe auch java.lang.@FunctionalInterface
- Schnittstellen mit genau einer abstrakten Methode

```
public interface Runnable {
    public void run();
}

Runnable r = () -> {
    …
};
```

```
public interface Predicate<T> {
    boolean test(T input);
}
```

```
<T>   void   filter(LinkedList<T> list, Predicate<T> p) {
      Iterator<T> iter = list.iterator();
      while (iter.hasNext()) {
            T v = iter.next();
            if (!p.test(v)) {
                  iter.remove();
            }
      }
}
```

```
LinkedList<String> list = … ;
filter(list, s -> hasSurrogates(s));
```

```
public interface IntBinOp {
    public int op(int x, int y) throws SomeCheckedException;
}
```

```
public int doIt(IntBinOp o, int x, int y)
    throws SomeCheckedException
{
    return o.op(5, 7);
}
```

```
IntBinOp f = (x, y) -> x + y;

int z = doIt(f, 2, 3);
```

```
IntBinOp createScaledOp(int scale, IntBinOp op) {
    return (x, y) -> op(scale * x, scale * y);
}

IntBinOp f = (x, y) -> x + y;
f = createScaledOp(100, f);
out.println(f.op(2, 3));
```

```
Function<Integer, Integer> fibonacci = (Integer n) -> {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci.apply(n – 2) + fibonacci.apply(n – 1);
};
```

- Nur bei statischen Variablen und Instanzvariablen (assignment before use)

```
class M {
    final double m;
    M(double m) { this.m = m; }
    double fm(double x, double y) { return x + y + m; }
    static double fs(double x, double y) { return x + y; }
}

DoubleBinaryOperator binop;

binop = (x, y) -> x + y
… binop.applyAsDouble(2.0, 3.1) …

binop = M::fs;
…

binop = new M(1.7)::fm;
…
```

```
class M {
    …
    M addm(M x) { return new M(this.m + x.m); }
    static M adds(M x, M y) { return new M(x.m + y.m); }
}

BinaryOperator<M> mop;

mop = (x, y) -> new M(x.m + y.m);
mop = M::addm
mop = M::adds
```

```
class M {
    …
    M add(M x) { return new M(this.m + x.m); }
    static M add(M x, M y) { return new M(x.m + y.m); }
}

BinaryOperator<M> mop;

mop = (x, y) -> new M(x.m + y.m);
mop = M::add                    // Fehler: Nicht eindeutig
```

- ArrayList::**new**
- File::**new**

- ```
  interface Factory<T> {
      T make();
  }
  ```

- ```
  Factory<ArrayList<String>> f1 = ArrayList::<String>new;
  ```

- ```
  List<String> strList = Arrays.asList("1","2","3");
  ```
- ```
  ArrayList<Integer> intList = strList
      .map(Integer::new)
      .into(new ArrayList<Integer>());
  ```

```
int count = 0;
list.forEach(s -> {
    out.println(s);
    count++;        // Fehler: Variable muss (effektiv) final sein
});
out.println(count);
```

```
MutableInteger count = new MutableInteger(0);
list.forEach(s -> {
    out.println(s);
    count.increment();
});
out.println(count.intValue());
```

```
interface Iterable<E> {
    Iterator<E> iterator();
}

interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

```
interface Iterable<E> {

    Iterator<E> iterator();

    default void  forEach(Consumer<? super T> action);
}

interface Iterator<E> {

    boolean hasNext();
    E next();

    default void remove();

    default void forEachRemaining(
        Consumer<? super E> action
    );
}
```

```
interface Ordered<T extends Ordered<T>> {
    boolean less(T that);
}

interface Trichotomy<T> extends Ordered<T> {

    default boolean lessOrEqual(T that) {
        return less(that) || equals(that);
    }

    default boolean greaterOrEqual(T that) {
        return !less(that);
    }

    …
}
```

```
public interface A {
    void default hello() { System.out.println("A"); }
}

public interface B extends A {
    void default hello() { System.out.println("B"); }
}

public class C implements B, A {
    public static void main(String ... args) {
        new C().hello();            // “B”
    }
}
```

```
public interface A {
    void default hello() { System.out.println("A"); }
}

public interface B {
    void default hello() { System.out.println("B"); }
}

public class C implements B, A {
    public void hello() {
        A.super.hello();
    }
    public static void main(String ... args) {
        new C().hello();
    }
}
```

- Klassen haben immer Vorrang
- Immer die passendste default-Implementierung gewinnt
- Konflikte können nur dann aufgelöst werden,
  wenn wirklich welche bestehen

```
interface A {
    void m() default { ... }
}
interface B extends A { }
interface C extends A { }
class D implements B, C { }

C c = new D();
c.m();
```

```
interface A {
    void m() default { System.out.println("A"); }
}

interface B extends A {
    void m() default { System.out.println("B"); }
}

interface C extends A { }

class D implements B, C { }

…

C c = new D();

c.m();                                  // “B”
```

- “lazy”

```
IntStream ints = strings
    .stream()
    .map(s -> s.length())
    .filter(i -> i % 2 != 0);
```

- “eager”

```
ints.forEach(System.out::println);
```

- “eager”

```
strings.removeAll(s -> s.length() == 0);
strings.filter(s -> s.length() == 0);
```

- “lazy”

```
collection.filter(...).map(...).reduce(...);
collection.parallel().filter(...).map(...).reduce(...);
```

Streams

- boolean allMatch(Predicate<? **super** T> *predicate*)
- boolean anyMatch(Predicate<? **super** T> *predicate*)
- <R> R collect(Collector<? **super** T, R> *collector*)
- <R> R collect(
  Supplier<R> *resultFactory*,
  BiConsumer<R, ? **super** T> *accumulator*,
  BiConsumer<R, R> *combiner*
)
- long count()
- Stream<T> distinct()
- Stream<T> filter(Predicate<? **super** T> *predicate*)
- Optional<T> findAny()
- Optional<T> findFirst()

Streams

- <R> Stream<R> flatMap(
  Function<? super T, ? extends Stream<? extends R>> *mapper*
)
- DoubleStream flatMapToDouble(
  Function<? super T ,? extends DoubleStream> *mapper*
)
- IntStream flatMapToInt(
  Function<? super T, ? extends IntStream> *mapper*
)
- LongStream flatMapToLong(
  Function<? super T ,? extends LongStream> *mapper*
)
- void forEach(Consumer<? super T> *consumer*)
- void forEachOrdered(Consumer<? super T> *consumer*)

Streams

- boolean isParallel()
- Iterator<T> iterator()
- Stream<T> limit(long maxSize)
- <R> Stream<R> map(Function<? super T, ? extends R> mapper)
- DoubleStream mapToDouble(
  ToDoubleFunction<? super T> mapper
)
- IntStream mapToInt(ToIntFunction<? super T> mapper)
- LongStream mapToLong(ToLongFunction<? super T> mapper)
- Optional<T> max(Comparator<? super T> comparator)
- Optional<T> min(Comparator<? super T> comparator)

Streams

- boolean noneMatch(Predicate<? super T> predicate)
- S parallel()
- Stream<T> peek(Consumer<? super T> *consumer*)
- Optional<T> reduce(BinaryOperator<T> *accumulator*)
- T reduce(
  T *identity*,
  BinaryOperator<T> *accumulator*
)
- <U> U reduce(
  U *identity*,
  BiFunction<U, ? super T, U> *accumulator*,
  BinaryOperator<U> *combiner*
)
- S sequential()

Streams

- Stream<T> sorted()
- Stream<T> sorted(Comparator<? super T> *comparator*)
- Spliterator<T> spliterator()
- Stream<T> substream(long *startingOffset*)
- Stream<T> substream(
  long *startingOffset*,
  long endingOffset
)
- Object[] toArray()
- <A> A[] toArray(IntFunction<A[]> *generator*)
- S unordered()

Fragen!?

MATHEMA

Vielen Dank

MATHEMA Software GmbH
Henkestraße 91
91052 Erlangen
www.mathema.de
info@mathema.de