Java Reborn

Martijn Blankestijn
&
Remko de Jong

16:00 – 17:30

Presentation

17:30 – 18:15

Food

18:15 – 20:30

Lab

20:30 – 22:00

Drinks

Presentation

- JSR 310 – DateTime API
- Enhanced Metadata (inc. JSR 308)
- Miscellaneous changes
- JSR 335 – Project Lambda
- Completable Futures

1900 + 2013

0-based

```
new Date(2013, 12, 31);
```

```
Sat Jan 31 00:00:00 CET 3914
```

Who uses

Date?

Calendar?

Joda-time?

Why not Joda-Time?

- No separation between human and machine timelines
- Pluggable chronology (e.g. Koptic calendar)
- Nulls

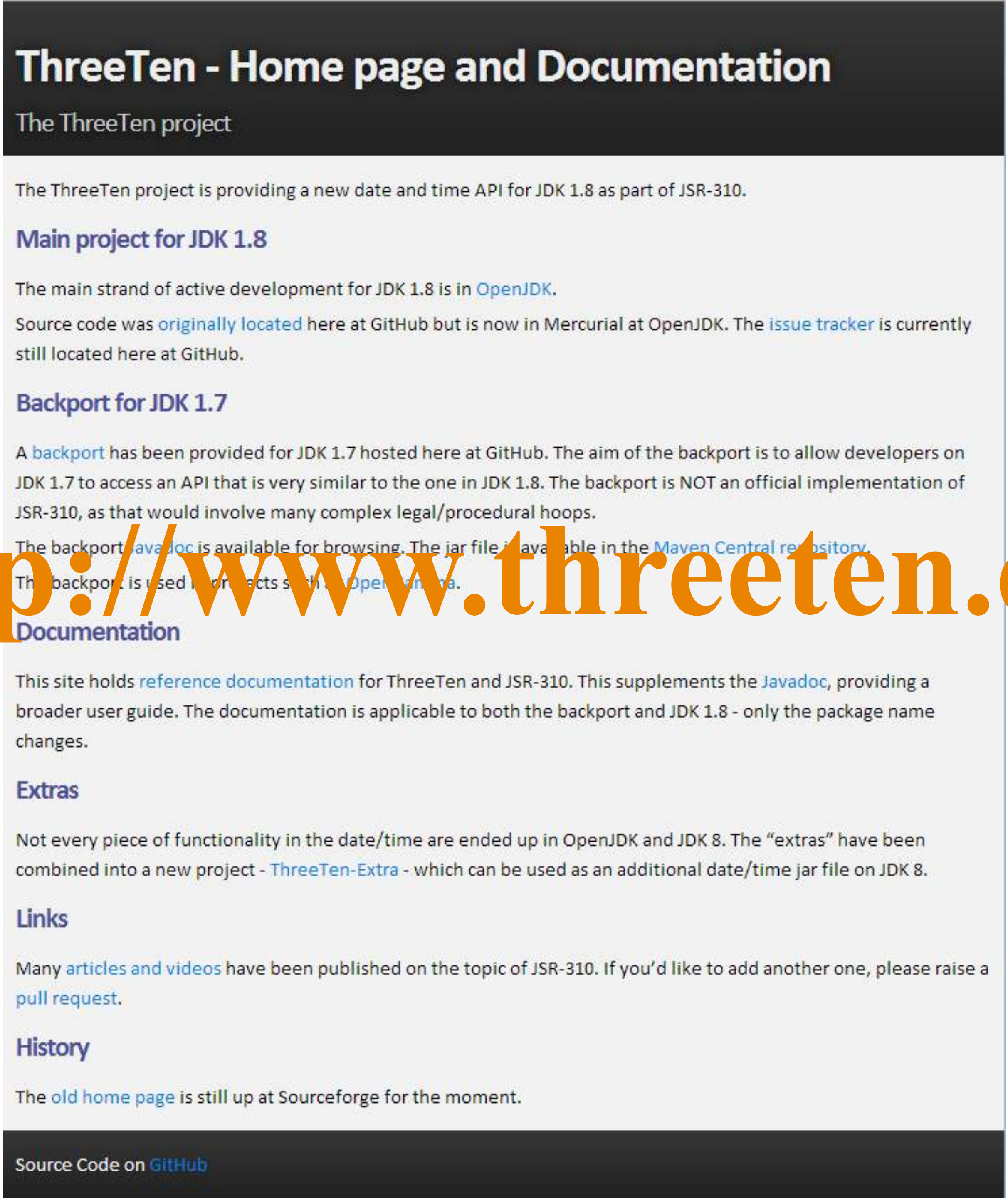
Goal of new Date-Time API

Immutable

Type-safe

```
@Deprecated
java.util.Date
java.util.Calendar
java.util.Timezone
java.text.DateFormat
```

http://www.threeten.org



ThreeTen - Home page and Documentation

The ThreeTen project

The ThreeTen project is providing a new date and time API for JDK 1.8 as part of JSR-310.

Main project for JDK 1.8

The main strand of active development for JDK 1.8 is in OpenJDK.

Source code was originally located here at GitHub but is now in Mercurial at OpenJDK. The issue tracker is currently still located here at GitHub.

Backport for JDK 1.7

A backport has been provided for JDK 1.7 hosted here at GitHub. The aim of the backport is to allow developers on JDK 1.7 to access an API that is very similar to the one in JDK 1.8. The backport is NOT an official implementation of JSR-310, as that would involve many complex legal/procedural hoops.

The backport javadoc is available for browsing. The jar file is available in the Maven Central repository.

The backport is used in projects such as OpenGamma.

Documentation

This site holds reference documentation for ThreeTen and JSR-310. This supplements the Javadoc, providing a broader user guide. The documentation is applicable to both the backport and JDK 1.8 - only the package name changes.

Extras

Not every piece of functionality in the date/time are ended up in OpenJDK and JDK 8. The "extras" have been combined into a new project - ThreeTen-Extra - which can be used as an additional date/time jar file on JDK 8.

Links

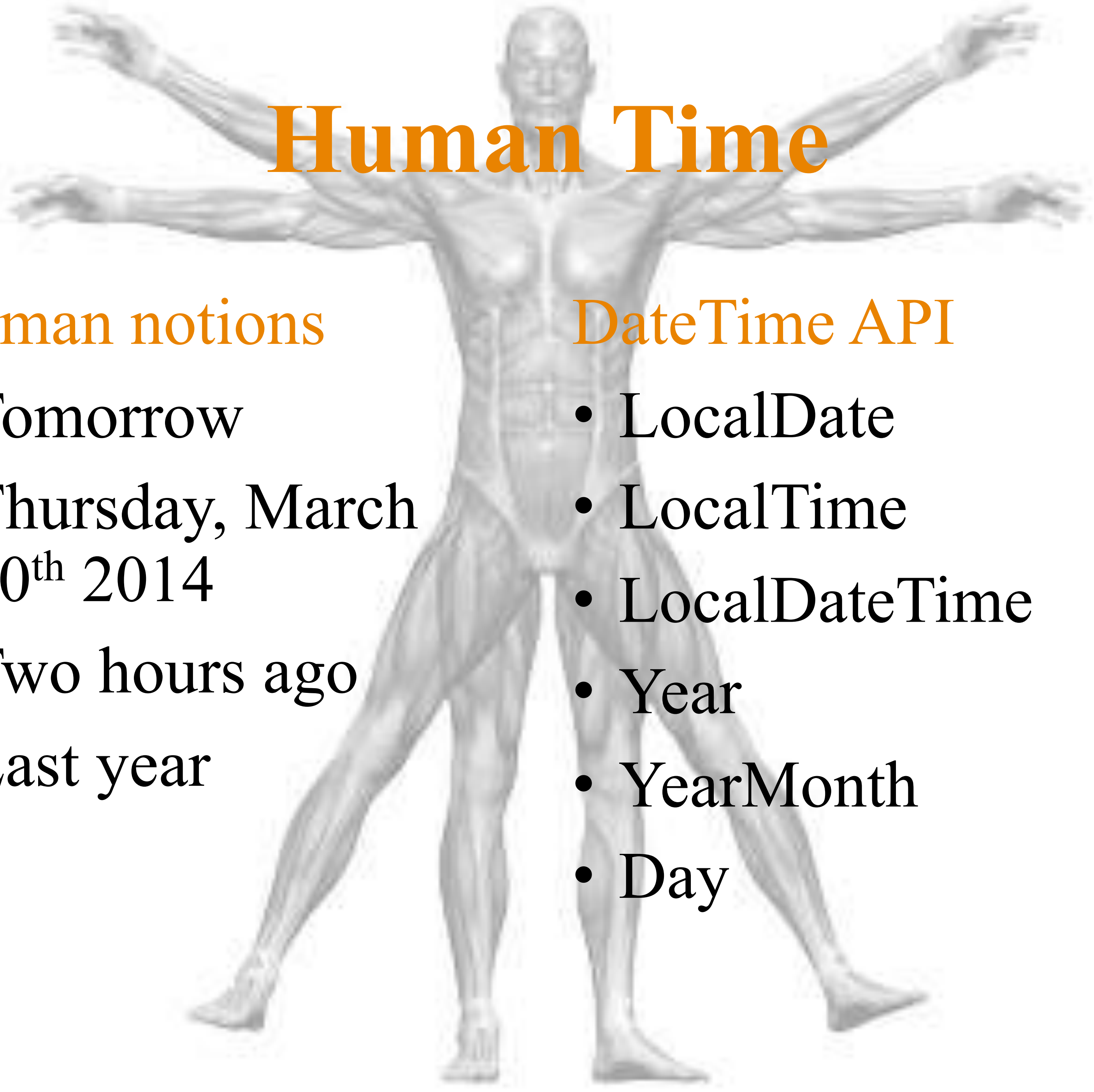
Many articles and videos have been published on the topic of JSR-310. If you'd like to add another one, please raise a pull request.

History

The old home page is still up at Sourceforge for the moment.

Source Code on GitHub

Human vs Machine (time)



Human Time

Human notions

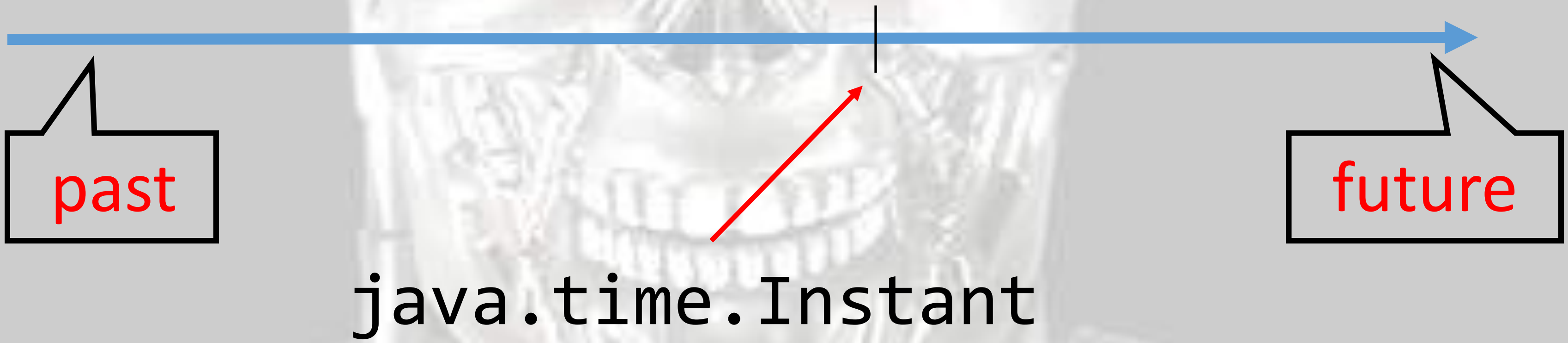
- Tomorrow
- Thursday, March 20th 2014
- Two hours ago
- Last year

DateTime API

- LocalDate
- LocalTime
- LocalDateTime
- Year
- YearMonth
- Day

Machine Time

Java Time Scale (in nanos)



Date arithmetic

```
date.plusDays(40);
date.plus(40, ChronoUnit.DAYS);

date.withDayOfMonth(5);
date.with(TemporalAdjusters.lastDayOfMonth());
date.with(lastDayOfMonth());
date.with(nextOrSame(FRIDAY));
```

build your own!

Composing

```java
Year year = Year.of(2014);
YearMonth yMonth = year.atMonth(MARCH);
LocalDate date = yMonth.atDay(25);
```

```java
Year.of(2014)
     .atMonth(MARCH)
     .atDay(25);
```

New objects

```java
== LocalDate.of(2014, MARCH, 25);
```

Amounts - Duration

Time based (Java Time Scale)

Storage in nanoseconds

```
Duration d = Duration.ofHours(6);

// After 6 hours
LocalDateTime.now().plus(d);
```

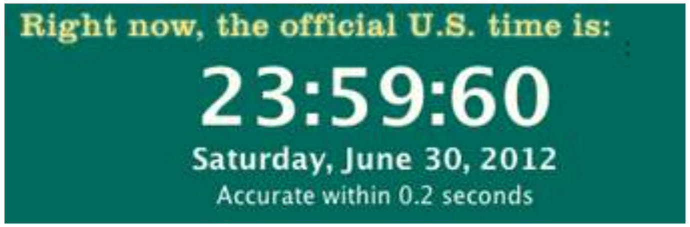
Amounts - Period

Date based

Years, months and days (ISO Calendar)

```
Period p = Period.ofDays(10)
                 .plusMonths(1);

LocalDate.now().plus(p);
```

How many seconds in a day?


Right now, the official U.S. time is:
23:59:60
Saturday, June 30, 2012
Accurate within 0.2 seconds

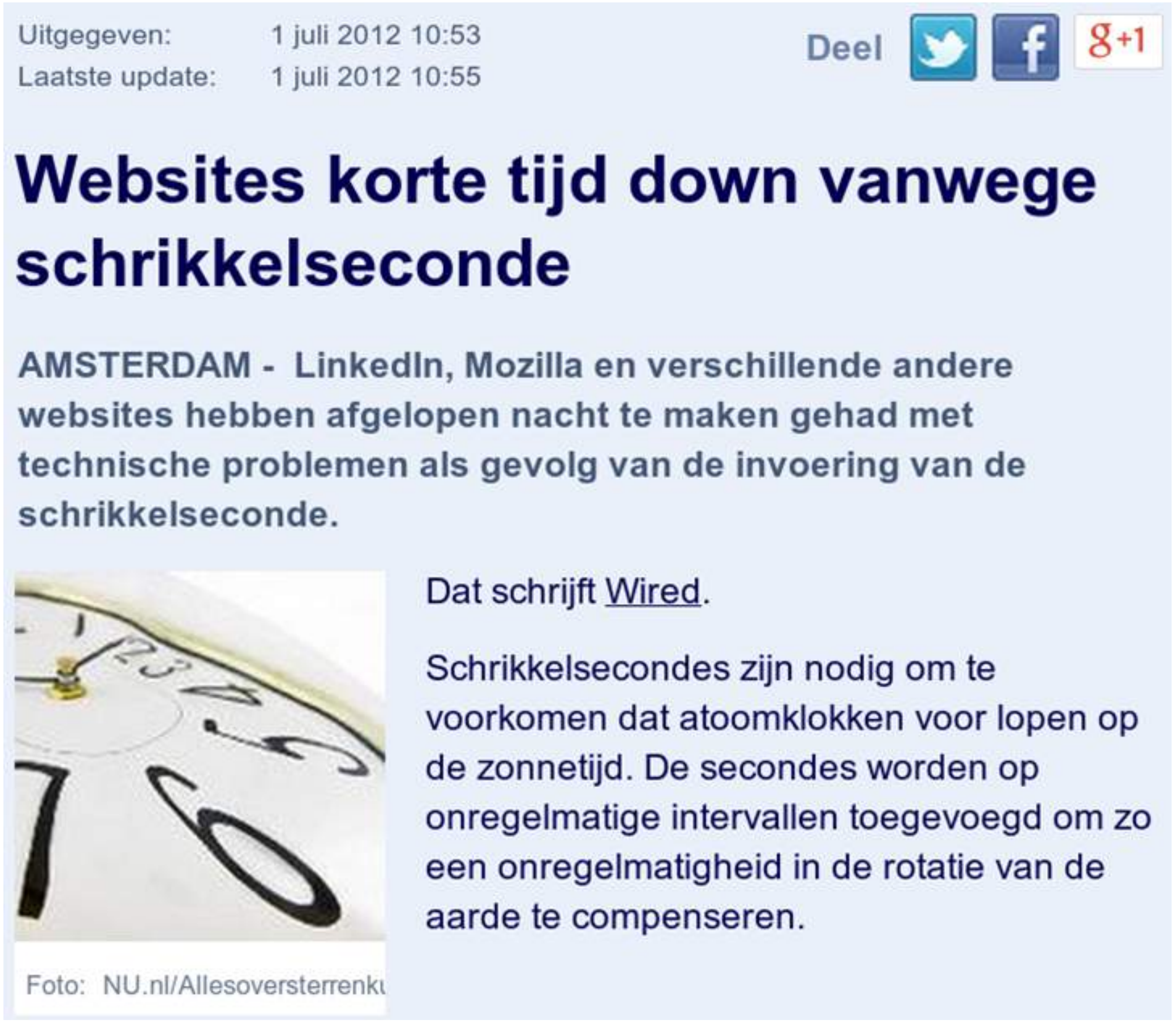
a) 86.401

b) 86.400

c) 90.000

d) 82.800

e) 86.399


Uitgegeven: 1 juli 2012 10:53
Laatste update: 1 juli 2012 10:55

Websites korte tijd down vanwege schrikkelseconde

AMSTERDAM - LinkedIn, Mozilla en verschillende andere websites hebben afgelopen nacht te maken gehad met technische problemen als gevolg van de invoering van de schrikkelseconde.

Foto: NU.nl/Allesoversterrenk

Dat schrijft Wired.

Schrikkelsecondes zijn nodig om te voorkomen dat atoomklokken voor lopen op de zonnetijd. De secondes worden op onregelmatige intervallen toegevoegd om zo een onregelmatigheid in de rotatie van de aarde te compenseren.

DateTime API Duration

```
ZoneId z = ZoneId.of("Europe/Amsterdam");

Duration duration = Duration.ofDays(1);


ZonedDateTime mrt30 =
  ZonedDateTime.of(
    LocalDateTime.of(2014,3,30,0,0,0), z);

mrt30.plus(duration);
```

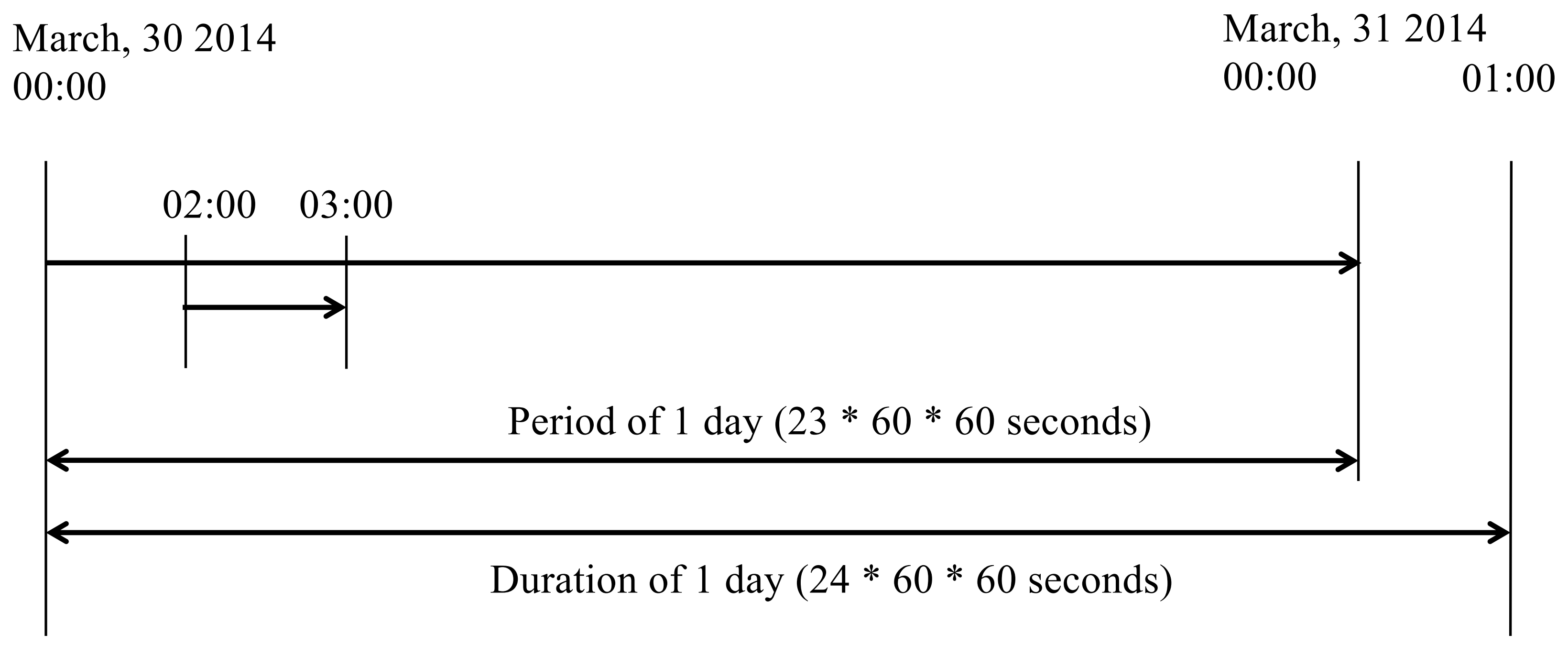
```
2014-03-31T01:00+02:00[Europe/Amsterdam]
```

DateTime API Period

```
Period period = Period.ofDays(1);


ZonedDateTime mrt31 = mrt30.plus(period);

  mrt31.toInstant().getEpochSecond()
- mrt30.toInstant().getEpochSecond();
```

$\boxed{\texttt{82.800 seconds !!!!}}$

Complexity of Time


March, 30 2014
00:00
March, 31 2014
00:00
01:00
02:00
03:00
Period of 1 day (23 * 60 * 60 seconds)
Duration of 1 day (24 * 60 * 60 seconds)

DateTimeFormatter

Thread-safe!!!

```
DateTimeFormatter
    .ofPattern("dd-MM-yyyy HH:mm:ss");
```

meta
data

Enhanced Metadata

Parameter Names

Repeating @nnotations

Type @nnotations

Parameter Names

```
@Path("persons/{id}")
public Person get(@PathParam("id") int id)
{
```

Parameter Names: Opt-in

```
void print(java.lang.reflect.Method m) {
   for (Parameter p : m.getParameters())
     if (p.isNamePresent()) {
       System.out.println(p.getName());
     }
   }
}
```

```
javac -target 1.8 -source 1.8 -parameters
```

Parameter Names: Example

```java
void invoke(Method method,
            Map<String,Object> params) {
  Object[] args =
    Stream.of(method.getParameters())
          .map(p->params.get(p.getName()))
          .toArray();
  method.invoke(resource, args);
}
```

Repeatable Annotations

```
 @AttributeOverride(name = "streetno" ...)
 private Address residentialAddress;
```

Container - pattern

```
@AttributeOverrides({
  @AttributeOverride(name = "streetno" ...,
  @AttributeOverride(name = "houseno" …)
})
private Address residentialAddress;
```

Type Annotations Goal

JSR 308 extends

Java’s annotation system

so that

annotations may appear

on **any use of a type**

Type annotations make Java's annotation system more expressive and uniform.

Type Annotations - Enablers

Syntax (JSR 308)

+

Annotation processing capability (JSR 269)

=

Pluggable type-checking

Libraries:

- http://types.cs.washington.edu/checker-framework/
- …

New Annotation Locations

```
@NonNull List<@Interned String> messages;

@Interned String @NonNull[] array;

LocalDate d = (@ReadOnly LocalDate) nu;

String toString(@ReadOnly ThisClass this) {}

public @Interned String intern() {}
```

Why Type Checkers?

Type checking prevents mistakes…

… but not enough

Null Pointer Exceptions

Wrong String Comparisons

Fake Enums

Units (meters/yards, kilogram/pounds)

Will I make it?

```
 /**
   * @param distance in kilometers
   * @return will I make it?
   */
boolean hasEnoughFuel(double distance) {
  return
    distance < velocity * maxFlightTime;
}
```

```
double distance = 1200; // km
plane.hasEnoughFuel(distance);
```

How about now?

```
 /**
   * @param distance in kilometers
   * @return will I make it?
   */
boolean hasEnoughFuel(@km double distance) {
  return
    distance < velocity * maxFlightTime;
}


  @km double distance = (@km double) 1200
  plane.hasEnoughFuel(distance);
```

other changes

Improved Type Inference

```
Set<String> x = new HashSet<>();
```

Diamond Operator

```
Set<String> s = new
  HashSet<>(Collections.<String>emptySet());
```

JDK 8

```
Set<String> s =
    new HashSet<>(Collections.emptySet());
```

Corner cases

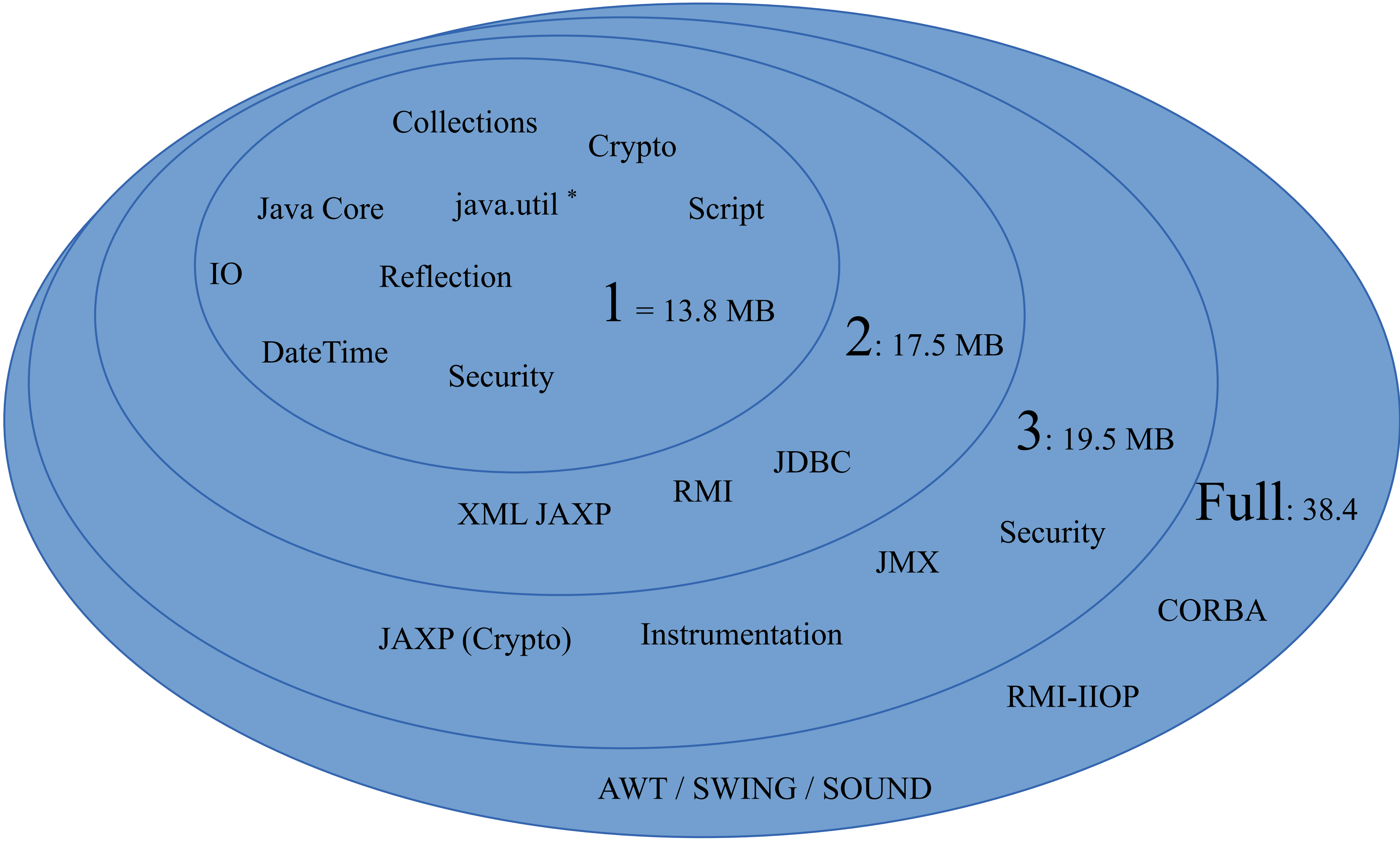
```
public static void main(String[] args) {
    print(Arrays.asList(1, 2, 3));
}

static void print(Object o) {
    out.println("Object o");
}

static void print(List<Number> ln) {
    out.println("List<Number> ln");
}
```

What does it print ??

JDK 1.7

JDK 1.8

Compact Profiles



Miscellaneous changes

- Nashorn
- PermGen
- Base64.Encoder / Base64.Decoder
- APT

http://openjdk.java.net/projects/jdk8/features

Generated by: https://www.jasondavies.com/wordcloud

Contents of JSR 335 or Lambda

+ Lambda expressions and method references

+ Enhanced type inference and target typing

+ Default and static methods in interfaces

Goal

“To enable **programming patterns** that require **modeling code as data** to be convenient and idiomatic in Java.”

idiomatic

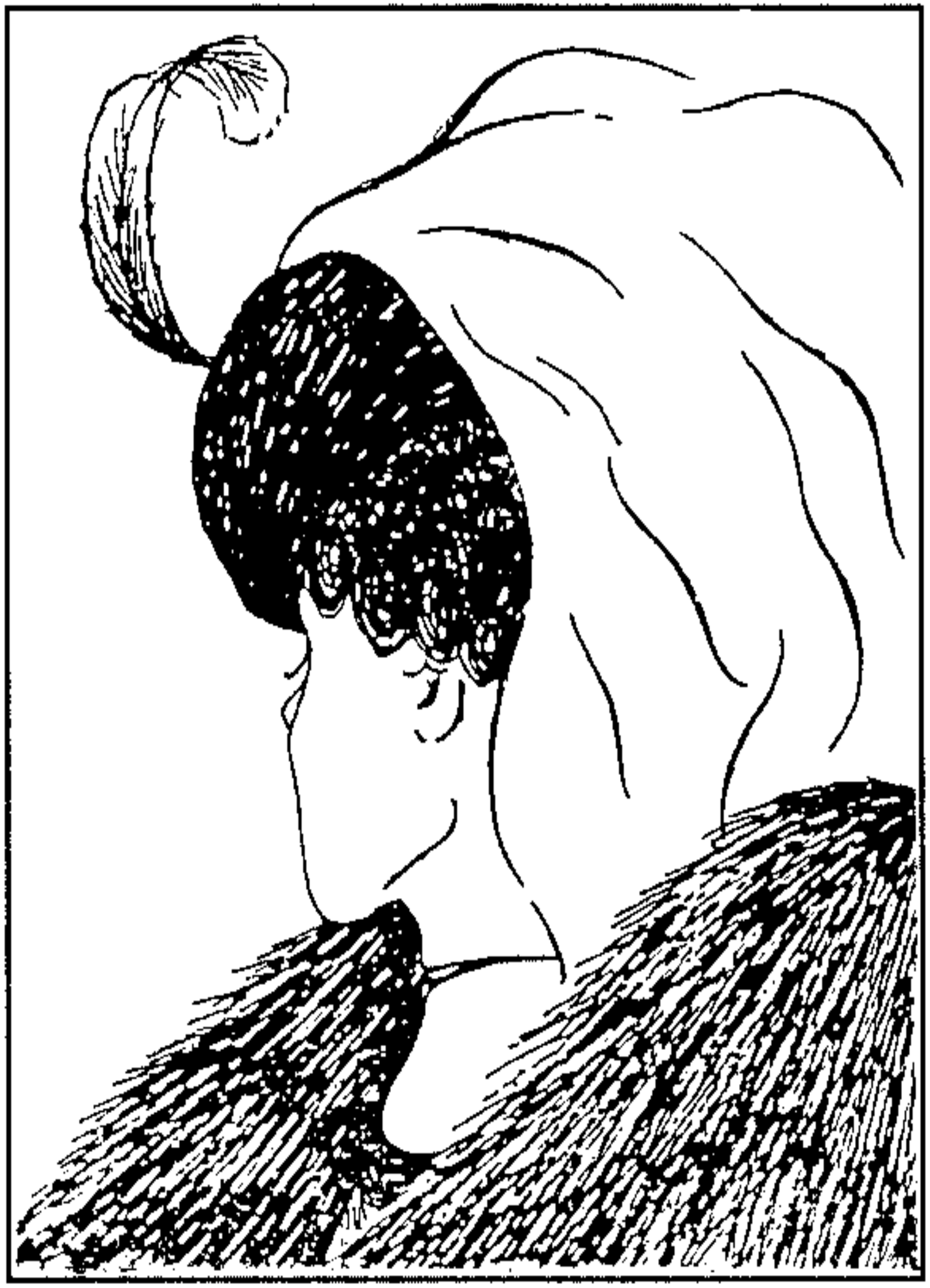
Line breaks: idiom|at¦ic

Pronunciation: /ˌɪdɪəˈmatɪk /

ADJECTIVE

1 Using, containing, or denoting expressions that are natural to a native speaker:

‘he spoke fluent, idiomatic English’



Paradigm *shift*

from

how → **what**

or

imperative → **declarative**

{ live coding; }

Quick Recap
“The Metamorphosis”

With inner class

```
Predicate<Integer> isEven =
   new Predicate<Integer>() {
      @Override
      public boolean test(Integer i) {
            return i % 2 == 0;
      }
   };
```

Full lambda notation

```
      Predicate<Integer> isEven =
(Integer i) -> { return i % 2 == 0; };
```

The type of a lambda is just a plain old interface

The type is inferred from the context

```
Predicate<Integer> isEven =
(i) -> { return i % 2 == 0; };
```

Implicit return for expressions

```
Predicate<Integer> isEven =
  (i) -> { i % 2 == 0; };
```

Removing parentheses

```
Predicate<Integer> isEven =
    i -> i % 2 == 0;
```

Using a method reference

```
Predicate<Integer> isEven =
        Class::isEven;
```

```
static boolean isEven(Integer i) {
     return i % 2==0;
}
```

functional interfaces

```java
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

<- backward compatible

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

public interface ActionListener … {
  void actionPerformed(ActionEvent e);
}

public interface PrivilegedAction<T> {
  T run();
}
```

@FunctionalInterface

```
public class ThreadExample {
  public static void main(String[] args) {
    new Thread(
      new Runnable() {
        @Override
        public void run() {
          System.out.println(“Hi!");
        }
      }
    ).start();
  }
}
```

@FunctionalInterface

```
public class ThreadExample {
  public static void main(String[] args) {
    new Thread(
      () -> System.out.println(“Hi!")
    ).start();
  }
}
```

@FunctionalInterface

General approach

Replace

```
new FunctionalInterface() {
   @Override
   public T someMethod(args) {
      body
   }
});
```

With

```
args -> { body }
```

@FunctionalInterface

`@FunctionalInterface`

```java
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Why?

- Catch errors @ compile time
- Communicate intention
- … but not required

`@FunctionalInterface`

```
package java.util.function;
```

```
Predicate<T>   -- boolean test(T t)
Consumer<T>    -- void accept(T t)
Function<T,R>  -- R apply(T t)
Supplier<T>    -- T apply()
```

```
@FunctionalInterface
```

So the type of a lambda is a functional interface…

```
() -> "done"; // ?
() -> "done";
```

```
Supplier<String> option1 = () -> "done";

Callable<String> option2 = () -> "done";

PrivilegedAction<String> option3 =
        () -> "done";
```

the type is inferred from the

context

The type is inferred from the context

```
Supplier<Runnable> c =
  () -> () -> out.println("hi");

// Illegal, cannot determine interface
Object o =
  () -> out.println("hi");

// Valid, explicit cast
Object o =
  (Runnable) () -> out.println("hi");
```

```
System.out::println
```

Types of Method Reference

```
1.ContainingClass::staticMethodName

2.ContainingObject::instanceMethodName

3.ContainingType::instanceMethodName

4.ClassName::new
```

```java
class Person {
  String name;
  LocalDate bday;

  public int getName() { return name; }
  public LocalDate getBirthday() {
    return bday;
  }
  public static int compareByAge(
    Person a, Person b) {
    return a.bday.compareTo(b.bday);
  }
}
```

Reference to a static method

```
Person::compareByAge
```

```
(a1, a2) -> Person.compareByAge(a1, a2)
```

```
class Person {
  …
  public static int compareByAge(
    Person a, Person b) {
    return a.bday.compareTo(b.bday);
  }
}
```

Reference to an Instance Method of a Particular Object

```
person::getBirthDay

p -> p.getBirthDay()
```

```
class Person {
  …
  public LocalDate getBirthday() {
    return birthday;
  }
}
```

Reference to an Instance Method of an Arbitrary Object of a Particular Type

```
String::startsWith

(s1, s2) -> s1.startsWith(s2)
```

BiFunction

```
public boolean startsWith(String prefix)
{
  return startsWith(prefix, 0);
}
```

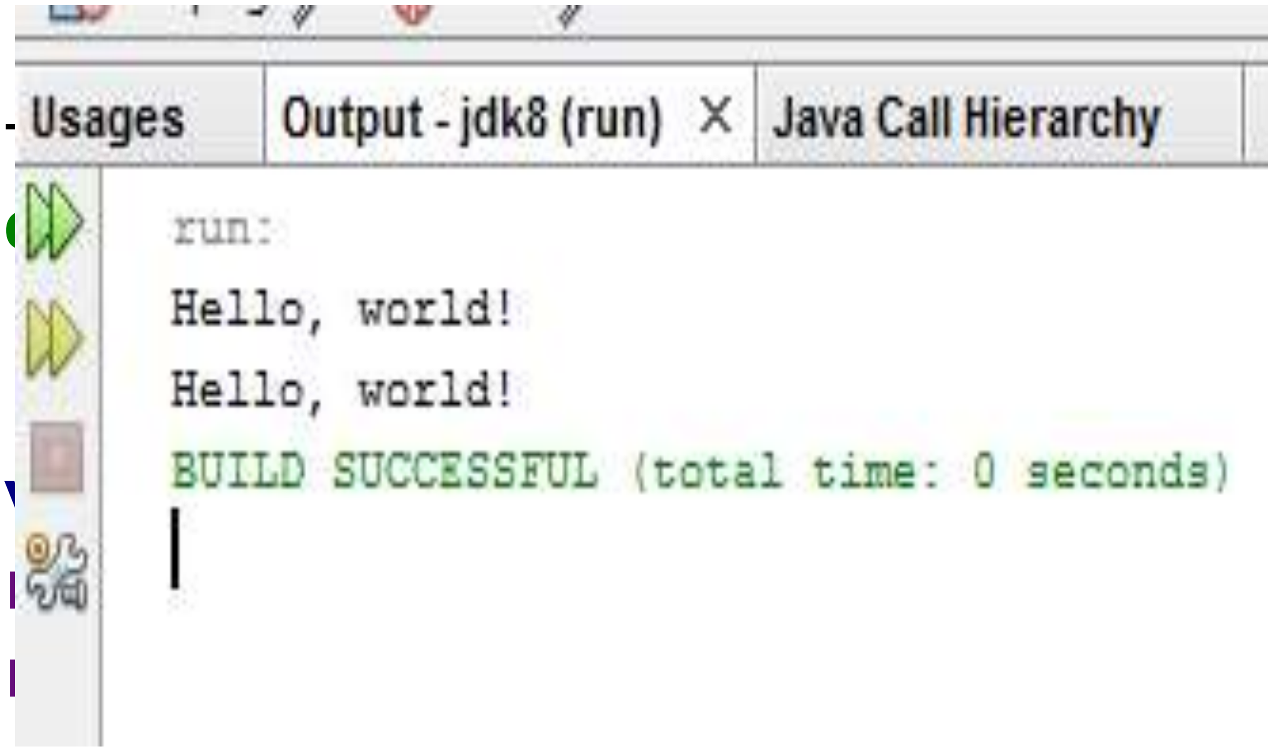
Reference to a constructor

```
Person::new

() -> new Person()
```

```
class Person {
   …
   public Person() {
      // Default constructor
   }
}
```

{ scope }

```java
import static java.lang.System.out;

public class Hello {
  Runnable r1 = () -> out.println(this);
  Runnable r2 = () -> out.println(toString());

  public String
    return "Hell
  }

  public static
    new Hello().
    new Hello().
  }
}
```



```
run:
Hello, world!
Hello, world!
BUILD SUCCESSFUL (total time: 0 seconds)
```

JDK 7

```
public static void main(String[] args) {
  int x = 5;
  Function<Integer, Integer> f1 =
    new Function<Integer, Integer>() {
      @Override
      public Integer apply(Integer i) {
        return i + x;
      }
    };
  f1.apply(1);
}
```

Variable ‘x’ is accessed from within inner class.
Needs to be declared final

JDK 8 – It compiles!

```
public static void main(String[] args) {
  int x = 5; // Effectively final
  Function<Integer, Integer> f1 =
    new Function<Integer, Integer>() {
      @Override
      public Integer apply(Integer i) {
        return i + x;
      }
    };
  f1.apply(1);
}
```

Effectively final

```
int x = 5; // effectively final
Function<Integer, Integer> f = i -> i + x;
f.apply(1); // 6

int x = 5; // not effectively final
Function<Integer, Integer> f = i-> i + ++x;
f.apply(1); // Does not compile
```

lambda == functional interface (SAM)

type depends on context

intuitive scoping, not like inner classes

enhanced type inference / effectively final variables
or
let the compiler work for you

method referencing for readability

Remember the first demo?

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

So where does this come from then?

```
public interface Iterable<T> {

    …

    default void forEach(
        Consumer<? super T> action) {
            Objects.requireNonNull(action);
            for (T t : this) {
                action.accept(t);
      }
}
```

And while we’re at it, why not add static methods as well then…

Static Interface Methods

Collection | Collections

Path | Paths

```java
@FunctionalInterface
public interface Comparator<T>

public static Comparator<T> reverseOrder() {
      return Collections.reverseOrder();
}

public static Comparator<T> naturalOrder() {
      return  ...;
}
```

streams

{ live coding; }

Stream API

1) Create

3) Terminal operations
Reduction
Collecting

2) Intermediary operations
Stateless transformations
Stateful transformations
Extract / combine

1) Creating streams

1) Call `Collection.stream();`

```
numbers.stream();
```

```
List<Integer> numbers;
```

2) Use a static factory

```
Stream.of("stream", "of", "strings");
```

3) Roll your own

```
public interface Spliterator<T>
```

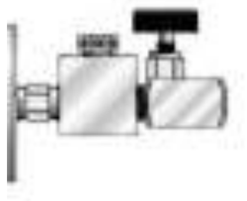
2) Transformations - stateful

```
Stream<String> chars =
  Stream.of("A","B","D","A","B");

// { "A","B","D" }
Stream<String> distinctChars =
  chars.distinct();

// {"A","A","B","B","D" };
Stream<String> sorted =
  chars.sorted(); // default natural order
```

2) Transformations - stateless

```
Stream<String> words =
  Stream.of("stream", "of", "strings");

// { "streams", "strings" }
Stream<String> longWords =
  words.filter(s -> s.length() > 4);

// { 6, 2, 7 };
Stream<Integer> lengths =
  words.map(s -> s.length());
```

3) Terminal operations



```
Stream<String> chars =
   Stream.of("AB","CDE","FGHI");

long numberOfChars = chars.count(); // 3


// "FGHI"
Optional<String> max =
   chars.max(comparing(String::length));
```

Optional values before JDK 8

```java
String street = "Unknown";
if (person != null
    && person.getAddress() != null
    && person.getAddress()
             .getStreet() != null) {
  street = person.getAddress().getStreet();
}
```

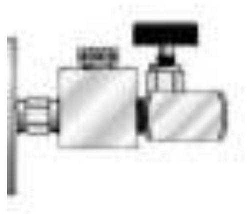
Optional

```
Optional<Person> person;

String street =
  person.map(Person::getAddress)
        .map(Address::getStreet)
        .orElse("Unknown");
```

Syntax

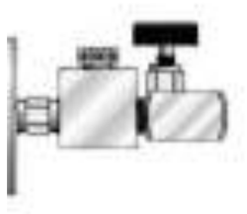
```java
// Creating a new Optional
Optional<String> value = Optional.of("Hi");
Optional<String> empty = Optional.empty();

// Most common operations
value.get(); // NoSuchElementException
value.orElse("something else");
value.ifPresent(v -> out.println(v));
value.isPresent();
```

Collecting



```
<R> R collect(
   Supplier<R> supplier,
   BiConsumer<R, ? super T> accumulator,
   BiConsumer<R, R> combiner);

Set<String> s = stream.collect(
   HashSet::new,
   HashSet::add,
   HashSet::addAll
);
```

More convenient collecting



```
List<String> list =
   stream.collect(Collectors.toList());

Set<String> set =
   stream.collect(Collectors.toSet());

String joined =
   stream.collect(joining(","));
```

{ live coding; }

parallel | parallel | parallel

parallel | parallel | parallel

parallel | parallel | parallel

{ live coding; }

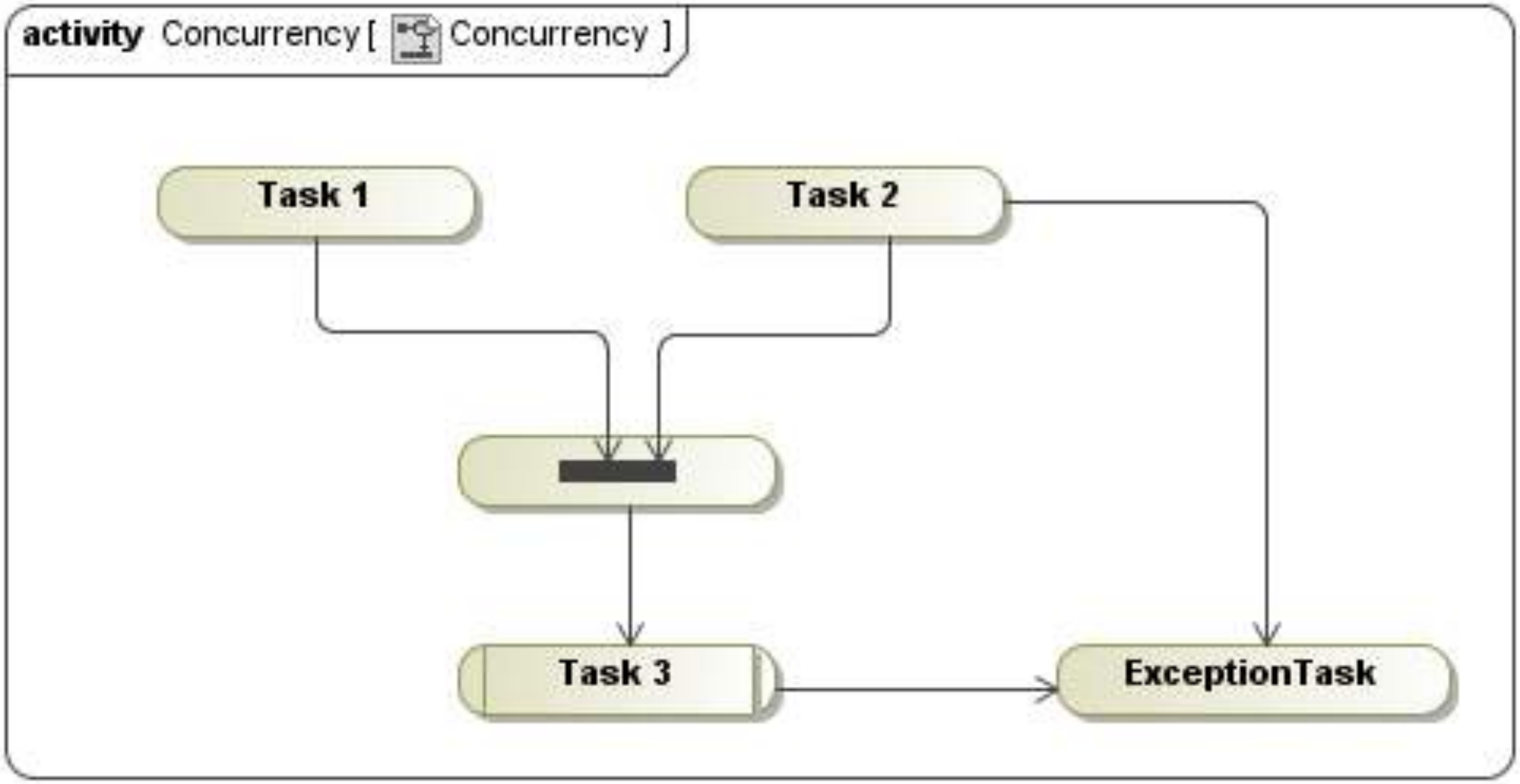
The ‘bun’ problem

```
List<Integer> doubled =
  numbers.stream()
         .map(i -> i * 2)
         .collect(toList());
```

```
List<Integer> doubled =
  numbers.map(i -> i * 2);
```

completable futures

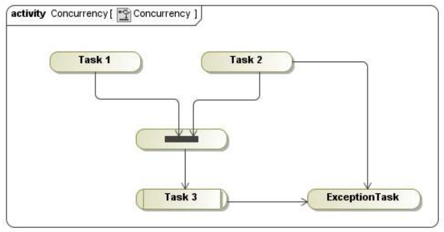
Problem Statement



java.util.concurrent.Future

```java
// @since 1.5
public interface Future<V> {
    boolean isDone();
    V get()
    V get(long timeout, TimeUnit unit)
}
```

Old School Future



```
ExecutorService es = newFixedThreadPool(2);

FutureTask<String> t1 = createTask("t1");
es.execute(t1); // and task 2

FutureTask<String> t3 =
        createTask(t1.get() + t2.get());
es.execute(t3);

t3.get(1, SECONDS);
```

Blocking

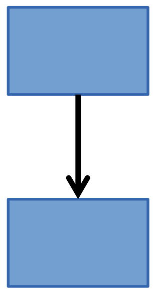
Blocking

Blocking

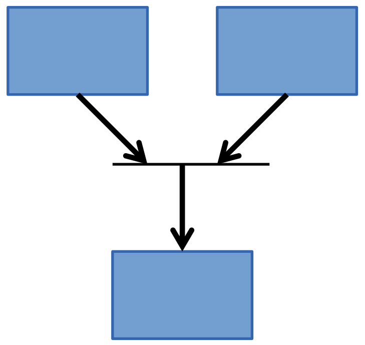
Composition

Composition / composing asynchronous operations

Seeing the calculation as a chain of tasks



thenApply
thenAccept
run

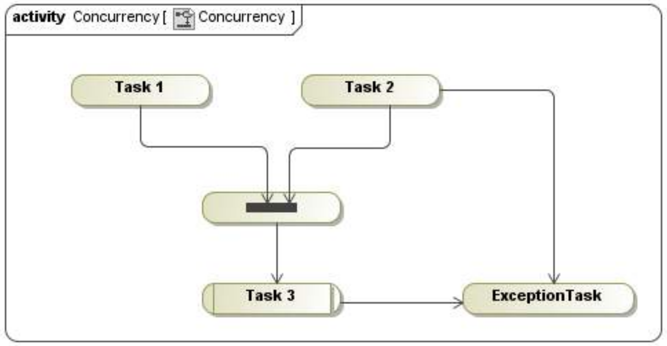


thenCombine
thenAcceptBoth
runAfterBoth
...

Promise Pipelining

```
public interface CompletableFuture<T>

…

CompletableFuture thenCombine(
  CompletableFuture other,
  BiFunction combiner)


<U,V> CompletableFuture<V> thenCombine(
  CompletableFuture<? extends U> other,
  BiFunction<? super T,? super U,? extends V>
    combiner)
```

CompletableFuture



```
CompletableFuture<String> task1 =
  CompletableFuture.supplyAsync(
    () -> doAction("t1"));
```

Blocking

```
CompletableFuture<String> task3 =
  (String p) -> supplyAsync(() -> doAction(p));


task1.thenCombine(task2, String::concat)
     .thenCompose(task3)
     .exceptionally(t -> "UNKNOWN“)
     .thenAccept(System.out::println)
```

Martijn Blankestijn
martijn.blankestijn@ordina.nl

Remko de Jong
remko.de.jong@ordina.nl

? ? ? ? ?

What’s next ?

~~16:00 – 17:30~~

~~Presentation~~

17:30 – 18:15

Food

18:15 – 20:30

Lab

20:30 – 22:00

Drinks

USB-stick

- Exercises
- JDK 8
  - mac, linux, windows, 32/64 bits
  - javadoc
- IDE
  - NetBeans 8
  - IntelliJ Community Edition 13.1

Hands-on Lab: Rooms

A11.06	A11.02	A12.05	A12.06	A12.07
Martijn	Philippe	Ivo	Pieter	Remko