Java 8 Workshop

Java Language Changes

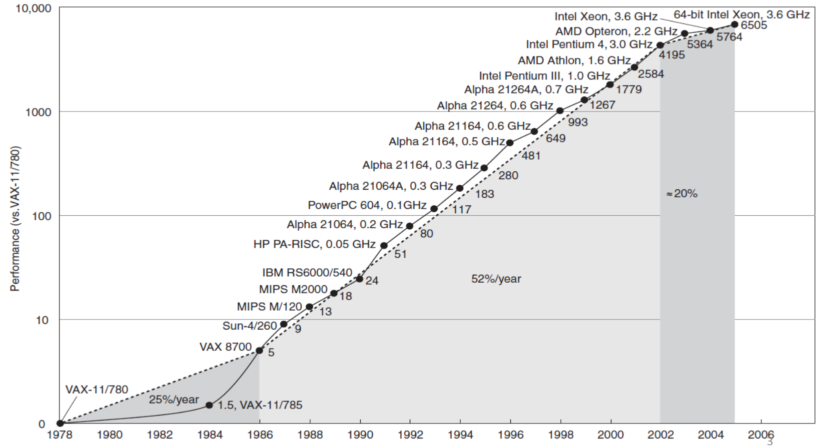
Workshop setup

- Complete package
  - workshop-eclipse-pack.zip
    or workshop-intellij-pack.zip
  - Run eclipse.bat or intellij.bat
- Java 8 + IDE already installed
  - workshop-workspace.zip
  - Source folders:
    - src/java-core/exercises
    - src/java-core/data
  - Additional libraries: lib/**

Workshop agenda

- Project Lambda
  - Introduction
  - Lambda expressions & functional interfaces
  - Default methods
  - Stream API
  - Optional Values
- Other novelties
  - Time API
  - Base64 API
  - Repeatable & Type annotations

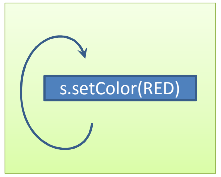
Evolution of processor speed

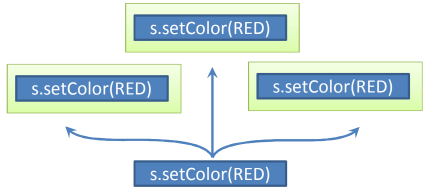


Go Parallel

- Moore's Law
  – More cores but not faster cores
- Serial API's
  – Limited to a shrinking fraction of available processing power
- Parallel API's
  – Multi-threading is hard
  – Little support in mainstream languages

Sequential vs. Parallel Execution



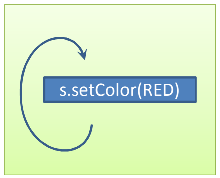


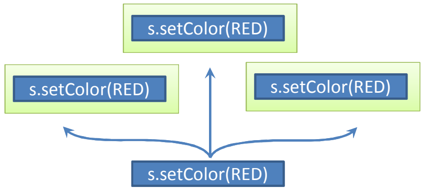
Example
External vs. Internal Iteration

```
for (Shape s : shapes) {
    s.setColor(RED)
}
```

```
shapes.forEach( s.setColor(RED) );
```


Sequential vs. Parallel Execution



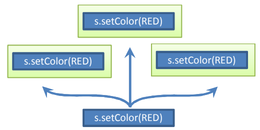


- Sequential Execution
  – For-loop is inherently sequential
  – Must process elements in the specified order
- Parallel Execution
  – Code expressed independently of the thread in which it will run
  – Requires code to be modelled as data

Internal Iteration

```
shapes.forEach( s.setColor(RED) );
```

- Requires code to be modelled as data
- The client delegates the iteration to the library
  – Allow for performance optimizations
    - Reordering of data
    - Parallelism
    - Short-circuiting
    - Laziness
- Client code can be clearer
  – Focus on stating the problem



Modelling Code as Data

- « Callback interface »

```
button.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent e) {
      System.out.println(e);
    }

});
```


Anonymous Inner Classes

- Various problems
  - Bulky syntax
  - Confusion wrt. *this*
  - Inflexible class-loading and instance-creation semantics
  - Inability to capture non-final local variables
  - Inability to abstract over control flow
- Advantage
  - Cleanly integrated in Java's type system: a *function value* with an *interface type*

Modelling Code as Data

- « Callback interface »

```
button.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent e) {
      System.out.println(e);
    }

});
```

- Lambda expressions

```
button.addActionListener( e -> System.out.println(e) );
```

  - Lighter-weight
  - « Anonymous Methods »

Lambda Expressions

```
(String s) -> { System.out.println(s);  … }
(String s) -> System.out.println(s)
s -> System.out.println(s);

(int x, int y) -> return x + y
(int x, int y) -> x + y
(x, y) -> x + y
() -> 42
```

Type of a Lambda Expression

```
button.addActionListener( e -> System.out.println(e) );
```

```
??? listener = e -> System.out.println(e) ;
```

- Functional Interfaces
  - Lambda expression represented as an interface
  - *Not* simply syntactic sugar

```
ActionListener listener = e -> System.out.println(e) ;
```

functional interface — function

Functional Interfaces

```
@FunctionalInterface
public interface ActionListener {

    public void actionPerformed(ActionEvent e);

}
```

```
ActionListener listener = e -> System.out.println(e) ;
```

- Single abstract method
- Optional annotation: @FunctionalInterface
  - Captures design intent
  - Checked at compile-time

Functional Interfaces

- No changes in Java type system
- Existing API's can profit from lambda expressions
  - java.lang.Runnable
  - java.util.Comparator
  - java.io.FileFilter
  - commons-collections Transformer

Functional Interfaces

- Package java.util.function provides new commonly used functional interfaces
  - `Predicate` T → boolean
  - `Function` T → R
  - `BiFunction` (T, S) → R
  - `UnaryOperator` T → T
  - `BinaryOperator` (T, T) → T
  - `Consumer` T → (void)
  - `Supplier` () → R
- Variations for primitive types

Target Typing

- Type of a lambda is inferred from the surrounding context

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

```
ActionListener listener   = e -> System.out.println(e) ;
Consumer<String> consumer = e -> System.out.println(e) ;
```

```
public interface Consumer<T> {
    void accept(T t);
}
```


Target Typing

- Lambda expression can be assigned *if*
  - Target is a functional interface type
  - Same number of parameters
  - Same parameter types
  - Compatible return type
  - Compatible exceptions

```
public interface BiFunction<T, U, R> {
    R apply (T t, U u);
}
```

```
BiFunction<String, Integer, Character> =
```

```
(String string, Integer i) -> string.charAt(i)
```


Method References

- When you want to call an existing method
- More compact & readable
- ± Shorthand for lambda expressions

- Static Methods

```
Function<String, Integer> parse =
                          (String s) -> Integer.parseInt(s);
```

≈

```
Function<String, Integer> parse = Integer::parseInt;
```


Method References

- Instance Methods

```
String.toUpperCase()
```

```
Function<String, String> toUpperCase = (String s) -> s.toUpperCase();
```

```
Function<String, String> toUpperCase = String::toUpperCase;
```

  - Invocation target is the first parameter

```
String.charAt(Integer)
```

```
BiFunction<String, Integer, Character> charAt =
                              (String s, Integer i) -> s.charAt(i)
```

```
BiFunction<String, Integer, Character> charAt = String::charAt
```

Method References

- Instance Methods
  - Specific instance

```
Set<String> knownNames = …
Predicate<String> isKnown = (String s) -> knownNames.contains(s);
```

```
Predicate<String> isKnown = knownNames::contains;
```

  - Constructors

```
Supplier<Set<String>> setFactory = () -> new TreeSet<String>();
```

```
Supplier<Set<String>> setFactory = TreeSet::new;
```


Exercises

1

- Lambda Expressions
- Functional Interfaces
- Method References

Go Parallel

- So now we can finally do this:

```
Collection shapes = …;
shapes.forEach( s -> s.setColor(RED));
```

?

We cannot add methods to interfaces

```
Collection shapes = …;
Collections.forEach( shapes, s -> s.setColor(RED));
```


Default Methods

- Add *new* methods to *existing* interfaces
- Provide a *default* implementation

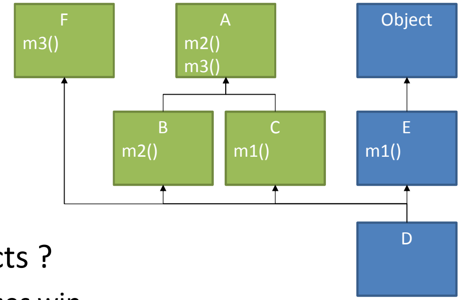
```
public interface Collection<T> {
 default void forEach(Consumer<T> action) {
      for (T t : this) {
          action.accept(t);
      }
    }
}
```

- Concrete classes can *override* default methods
- Previously compiled classes inherit default implementation

Default Methods

- Allows evolution if API's
- NOT « Traits »
- Java remains statically typed
- Multiple inheritance ?
  - Type (interfaces)
  - Behaviour (default methods)
  - ~~State~~

Default Methods



- Conflicts ?
  - Classes win
  - Most specific ancestor wins
  - Ambiguity is a compile-time error

Fluent API's

- Default and Static methods on interfaces
- Allow for « fluent-style » API's

```
Comparator<Person> comparator =
       Comparator.comparing(Person::getLastName)
                 .thenComparing(Person::getFirstName)
                 .thenComparingInt(Person::getAge);
```


Exercises

2

- Default Methods
- New methods in java core classes

Putting it all together

```
List<Person> people = ...
Collections.sort(people, new Comparator<Person>() {
  public int compare(Person x, Person y) {
    return x.getLastName().compareTo(y.getLastName());
  }
});
```


Putting it all together

```
Collections.sort(people, new Comparator<Person>() {
  public int compare(Person x, Person y) {
    return x.getLastName().compareTo(y.getLastName());
  }
});
```

```
Collections.sort(people, (Person x, Person y) ->
                  x.getLastName().compareTo(y.getLastName()));
```

```
Collections.sort(people,
         Comparator.comparing((Person p) -> p.getLastName()));
```

```
Collections.sort(people, comparing(p -> p.getLastName()));
```

```
Collections.sort(people, comparing(Person::getLastName));
```

```
people.sort(comparing(Person::getLastName));
```


Go Parallel

- So now we can finally do this:

```
Collection shapes = …;
shapes.forEach( s -> s.setColor(RED));
```

- Or this:

```
Collection shapes = …;

Collection greenShapes = shapes.filter( s -> s.getColor()==GREEN);

Collection greenShapeSizes = greenShapes.map( s -> s.getSize());

Double average = greenShapeSizes.average();
```


Go Parallel

- Utility methods on Collections would inherently have a performance impact
  Even written like this :

```
Double average = shapes
                    .filter( s -> s.getColor()==GREEN)
                    .map( s -> s.getSize())
                    .average();
```

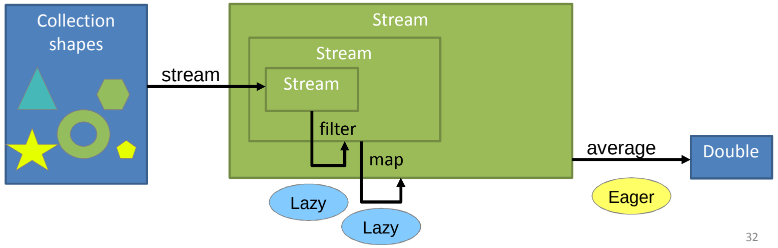
- What we want is :

```
Double average = shapes
                    .filter( s -> s.getColor()==GREEN)
                      and also map( s -> s.getSize())
                      and also average();
```

Introducing Streams

- Need for a new concept : *Stream*

```
Double average = shapes.stream()
                       .filter( s -> s.getColor()==GREEN)
                       .map( s -> s.getSize())
                       .average();
```



Stream Optimizations

- Laziness-seeking

```
shapes.filter( s -> s.getSize() > 42 )
      .findFirst()
```

- Stream Properties
  - Sized (Collection)
  - Distinct (Set)
  - Ordered (List)
  - Sorted

```
treeSet.stream().sorted()
set.stream().distinct()
```

NOP

Support filter-map-reduce pattern

```
Collection<T>
stream()
filter( Predicate<T> )
map( Function<T, U> )
    mapToInt( ToIntFunction<T> )
reduce( BinaryOperator<U> )
    sum( )
    max( )
    collect( toList() )
```

- Collection<T>
- Stream<T>
- Stream<T>
- Stream<U>
  - IntStream
- U
  - int
  - int*
  - List<U>

Stream API: Creation

- Collection<T> Stream<T>
- Arrays.stream(T[]) Stream<T>
- IntStream.range(from, to) IntStream
- Stream.iterate(T, UnaryOperator) Stream<T>
- BufferedReader.lines() Stream<String>
- Files.walk(Path) Stream<Path>
- Random.ints() IntStream

Stream API: Operations

- stream Collection<T> → Stream<T>
- parallelStream Collection<T> → Stream<T>
- filter Stream<T> → Stream<T>
- distinct Stream<T> → Stream<T>
- sorted Stream<T> → Stream<T>
- map Stream<T> → Stream<U>
- reduce Stream<T> → T
- collect Stream<T> → Collection<T>
- flatMap Stream<Collection<T>> → Stream<T>

Stream API: Reduction

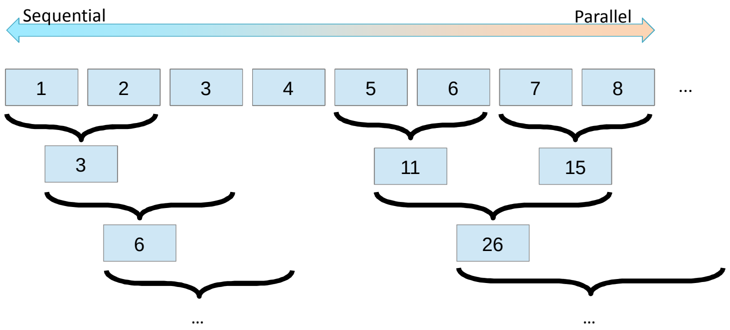
- Reduction of a Stream to a single end result

```
int sum = intStream.sum();
```

- Streams provide an abstraction over sequential / parallel behaviour
- How does reduction work in parallel ?

```
int sum = intStream.parallel().sum();
```


Reduction Operator




```
T result = stream.reduce(BinaryOperator<T> accumulator)
```

Associativity (a+b)+c == a+(b+c)

Reduction: Empty Streams ?

- Identity element

```
IntStream.of().sum() == 0
```

- Explicit identity

```
IntBinaryOperator product = (i,j) -> i * j;
IntStream.of().reduce(1, product) == 1
```

```
IntStream.of().max() == ?
```

?

Optional Values

- `java.util.Optional`
- « The operation may not produce a result »

```
OptionalInt result = IntStream.of().max();
```

- Alternative to **`null`**

> **" I call it my billion-dollar mistake. It was the invention of the null reference in 1965"**
> **– Tony Hoare**

- Auto-documenting
- Fluent-style API
- Cope explicitly with the absence of a value

Optional Values: Usage

- Provide a default

```
Optional<String> optional = getOptionalValue();
System.out.println(optional.orElse("default"));
```

- Fail

```
optional.orElseThrow(() -> new IllegalStateException("no value"))
```

- Use an alternative

```
optional.orElseGet(() -> requestUserInput())
```

- Provide optional behaviour

```
optional.ifPresent(value -> doSomethingWith(value));
```


Optional Values: Propagation

- Like a stream with 0 or 1 element
- Optional value mapping

```
Optional<Integer> length = optional.map(String::length);
```

- Filtering values

```
Optional<String> notEmptyString =
           optional.filter(string -> string.length() > 0);
```

- Flat mapping

```
Optional<String> punctuation =
optional.flatMap( string -> findPunctuationMarks(string) );
```


Exercises

3

- Creating Streams
- Map-Filter-Reduce operations

Collect

- « Mutable » Reduction
- Collectors utility class

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

- Custom collections: Stream.collect
  - supplier — create result — `new ArrayList<T>();`
  - accumulator — modify result — `list.add(t)`
  - combiner — combine results — `list1.addAll(list2)`

Collectors

- toList() — List<T>
  toSet() — Set<T>
- toMap — Map<K, V>
  - Function<T, K> keyMapper
  - Function<T, V> valueMapper
- Joining — String
  - String delimiter

Collectors

- groupingBy — Map<K, List<T>>
  - Function<T, K> classifier
- groupingBy — Map<K, D>
  - Function<T, K> classifier
  - Collector<T, ?, D> downStream
- partitioningBy — Map<Boolean, List<T>>
  - Predicate<T>

Exercises

4

- Collectors

Go Parallel ?

- Overhead!

# numbers in list	Avg. time SERIAL	Avg. time PARALLEL	NUM_RUNS
3	0.02s	0.37s	100,000
30	0.02s	0.46s	100,000
300	0.07s	0.53s	100,000
3,000	1.98s	2.76s	100,000
30,000	0.67s	1.90s	10,000
300,000	1.71s	1.98s	1,000
3,000,000	1.58s	0.93s	100

- Useful for large datasets and/or heavy calculations
- Don't optimize blindly ➔ Benchmark your applications
- Streams prepare code for behind-the-scenes optimizations

http://www.javacodegeeks.com/2014/05/the-effects-of-programming-with-java-8-streams-on-algorithm-performance.html

Time API (JSR-310)

- java.util.Date & java.util.Calendar : flawed API
- java.time.*
- Inspired by Joda Time – with improvements

Time API – Powerful & Exact API

- Immutable
- Timezone is optional
- Distinction Date - Time
- « Partial » Classes : **`MonthDay`** & **`YearMonth`**
- Range Classes : **`Duration`** & **`Period`**
- Calculations : plus, minus

Time API – Complex Model

- **`Clock, Instant`**
  Machine-oriented notion of time
- **`LocalDate, LocalTime, LocalDateTime`**
  Human-oriented notion of time
- **`ZonedDateTime, OffsetTime, OffsetDateTime`**
- Complex abstract model
- Support for other Chronologies
  Hijrah, Buddhist, Chinese, ...

JSR-310 vs. Joda-Time

- JSR-310 was inspired by Joda-Time
- Same Spec-lead (Stephen Colebourne)
- Distinction Machine ↔ Human timeline
- Calendar systems no longer “transparent”
- No defaults for null
- Revised implementation

http://blog.joda.org/2009/11/why-jsr-310-isn-joda-time_4941.html

Time API - Examples

- Clock: useful for testing

```
Clock clock = Clock.systemDefaultZone();

Clock testingClock = Clock.fixed(Instant.now(),
                                 ZoneId.systemDefault());
```

- Local Date and Time
  not bothering with timezones

```
LocalDate today = LocalDate.now();

LocalDate january7 = LocalDate.of(2015, Month.JANUARY, 7);

LocalDate deadline = LocalDate.now().plus(Period.ofDays(15));
```


Time API - Examples

- Creation: now / of
- Adding information: at
- Changing information: with

```
LocalDate date = LocalDate.now();

LocalTime time = LocalTime.of(11, 30);

LocalDateTime dateTime = date.atStartOfDay() ;
dateTime = date.atTime(time) ;
dateTime = time.atDate(date) ;

LocalDateTime food = dateTime.withHour(12);
```


Time API - Examples

- « Partial » notions of dates:
  get rid of unnecessary information

```
MonthDay christmas = MonthDay.of(Month.DECEMBER, 25) ;

YearMonth februari2015 = YearMonth.of(2015, Month.FEBRUARI);

LocalDate lastDay = februari2015.atEndOfMonth() ;
```

Base-64 API

- java.util.Base64

```
Encoder encoder = Base64.getEncoder();

encoder.encode("ABC".getBytes());

OutputStream stream = encoder.wrap(outputStream);
```

```
Decoder decoder = Base64.getDecoder();

decoder.decode(bytes);

InputStream stream = decoder.wrap(inputStream);
```


Repeatable Annotations

```
@Aliases({ @Alias("One"), @Alias("Two")})
class Test {
}
```

```
@Alias("One")
@Alias("Two")
class Test {
}
```

- No more explicit “container” annotations
- Compiler takes care of implicit container annotations
- See java.lang.@Repeatable
- Expected usage in frameworks or custom annotations

Type Annotations

- Annotations on the appearance of a *Type*
- Practically anywhere
- Elements : Type, Type Parameter, Type Use
- Expected usage in frameworks or custom annotations

```
@Here
class Test<@Here T> extends @Here Object {

    public @Here String getValues(@Here List< @Here String> List)
                          throws @Here IllegalArgumentException {

        return new @Here String("Hello");
    }
}
```


References

- Brian Goetz
  - State of the Lambda
    http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html
  - State of the Lambda: Libraries Edition
    http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html
- José Paumard
  - http://www.slideshare.net/jpaumard
- Angelika Langer
  - http://www.angelikalanger.com/Lambdas/Lambdas.html