Java8

Per Kristian Solheim
Somaiah Kumbera

Introduction

- About us
- What’s the big deal?
- New Features in Java8
- Functional Interfaces
- Lambdas
- Method References
- Streams

About us

- Somaiah Kumbera
  - Been with Capgemini since October 2011
  - Worked with Java since 1996(!)
  - Been in love with Guava since forever
- Per Kristian Solheim
  - Capgemini since November 2011
  - Worked with Java since 2006
  - Currently located in Sparebank1 Nettbankutvikling

What’s the big deal?

Major release, out in March 2014!

Focus on concise readable code, remove code bloat

Focus on parallelism (use with caution!), multicores

New features in Java8

- Lambdas
- Method References
- Streams
- Enhanced Interfaces
- New Date/Time API
- Optional
- Nashorn Javascript engine

See the full list here: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

New features - Enhanced Interfaces

```
interface Foo {
default void talk() {
                out.println("Foo!");
        }
}
```

New features - Enhanced Interfaces

Exercise 1:

- Create 2 interfaces
- Both interfaces have a default method with the same signature
- Create a class that implements these interfaces

New features - Date/Time API

- LocalDate – Day, month, year.
- LocalTime – Time of day only.
- LocalDateTime – Both date and time.

For timezone specific times you use ZonedDateTime.

New features - Date/Time API

Ancient times:

```
Calendar cal = Calendar.getInstance();
cal.add(Calendar.HOUR, 8);
cal.getTime(); // actually returns a Date
```

JDK8:

```
LocalTime now = LocalTime.now();
LocalTime later = now.plus(8, HOURS);
```

Combine Date and time:

```
Localdate date = LocalDate.of(2014, Month.MARCH, 15);
LocalTime time = LocalTime.of(12, 15, 0);
LocalDateTime datetime = date.atTime(time);
```

New features - Date/Time API

Typesafe :)

```
LocalDate today = LocalDate.now();
LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);
LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
```

Period and Duration

```
Period p = Period.between(date1, date2);
Duration d = Duration.between(time1, time2);
Duration twoHours = Duration.ofHours(2);
Duration tenMinutes = Duration.ofMinutes(10);
Duration thirtySecs = Duration.ofSeconds(30);
```

New features - Date/Time API

Exercise 2: Play around with java.time:

Objectives - learn the difference between LocalDate, LocalDateTime and Instant

- Create a credit card with an expiry date 1 week from the date of creation (now)
- Write tests to check that the date is created correctly
- Find the number of days between today and the expiry date (should be 7 :))

New features - Optional

Ancient Times:

```
getEventWithId(10).getLocation().getCity();

public String getCityForEvent(int id) {
    Event event = getEventWithId(id);
    if(event != null) {
        Location location = event.getLocation();
        if(location != null) {
            return location.getCity();
        }
    }
    return "TBC";
}
```

New features - Optional

JDK 8:

```
public String getCityForEvent(int id) {
    Optional.ofNullable(getEventWithId(id))
            .flatMap(this::getLocation)
            .map(this::getCity)
            .orElse("TBC");
}
```

Lambdas

- Project lambda (JSR 335) started in 2009.
- The biggest new feature of Java 8.
- Enables functional programming principles in Java.
- A lambda expression is a block of code with parameters.
- Pre Java 8 we used anonymous inner classes.
- Reduce boilerplate, simplifies code and improves reusability.
- Functional interfaces
- Lambdas are single method classes that represent behavior.
- They can either be assigned to a variable or passed around to other methods just like we pass data as arguments.
- Use a lambda expression whenever you want a block of code executed at a later point in time
- More possibilities for optimization (Laziness, parallelism, Out-of-order execution).

Lambdas vs anonymous inner class

```
//Old school style (anonymous inner class)
Runnable r = new Runnable(){
            @Override
            public void run() {
                System.out.println("My Runnable");
            }};


//New kid on the block
Runnable r1 = () -> System.out.println("My Runnable");
```

Lambda - syntax

```
(final String name) -> {
     return “Hello, “ + name;
}
```

Input arguments
Arrow
Code block

Lambda - syntax

```
(final String name) -> {
    return "Hello, " + name;
}
```

```
(name) -> {
    return "Hello, " + name;
}
```

Compiler will figure out the types

Lambda - syntax

```
(final String name) -> {
    return "Hello, " + name;
}
(name) -> {
    return "Hello, " + name;
}
name -> {
    return "Hello, " + name;
}
```

There's no need for parentheses with only one parameter

Lambda - syntax

```
(final String name) -> {
        return "Hello, " + name;
}
(name) -> {
        return "Hello, " + name;
}
name -> {
        return "Hello, " + name;
}
name ->
return "Hello, " + name;
```

No curlybraces needed for one-liners

Lambda - syntax

```
(final String name) -> {
return "Hello, " + name;
}
(name) -> {
return "Hello, " + name;
}
name -> {
return "Hello, " + name;
}
name ->
return "Hello, " + name;
name -> "Hello, " + name;
```

Return type are auto inferred
No return needed for one-liners

Lambda - scope

```
public class Main {
  public Main(){
    //int x= 0;
    Function<String,String> func1 = x -> {System.out.println(this);return x ;};
    System.out.println(func1.apply(""));
  }
  public String toString(){
    return "Main";
  }
  public static void main(String[] args) {
    new Main();
  }
}
```

We cannot create variables with the same name as a variable in the enclosing method!!
(uncommenting int x = 0; will cause compile time error)

A lambda expression has the same scope as its outside method.
The lambda expression doesn't create its own scope.

Functional Interfaces

- Interface with one and only one abstract method

- Can also declare the abstract methods from the java.lang.Object class.

- Previously known as Single Abstract Method(SAM)

- @FunctionalInterface

- Also had one-method interfaces such as Runnable, Callable, Comparator, ActionListener and others in Java pre 8.

- New built in functional interfaces such as Predicate, Consumer, Function, UnaryOperator, BinaryOperator (The java.util.function Package)

Functional Interfaces

```
@FunctionalInterface
public interface ITrade {
  public boolean check(Trade t);
}
```

```
ITrade newTradeChecker = (Trade t) -> t.getStatus().equals("NEW");

// Or without type!
ITrade newTradeChecker = (t) -> t.getStatus().equals("NEW");
```

Functional Interfaces - Built in

```
// Predicate for evaluation. Here is a lambda to check an empty string
Predicate<String> emptyStringChecker = s -> s.isEmpty();

// Function for transformation purposes. Her is a String to an integer transformation
Function<String, Integer> stringToInt = x -> Integer.valueOf(x);

// Supplier takes no arguments and pruduces a result of type. Eg. the new operator
Supplier<List<String>> listSupplier = ArrayList::new;

//Consumer takes a single argument and produce no result. Take an object and perform operations on it.
Consumer<Student> style = (Student s) -> System.out.println("Name:"+s.name +" and Age:"+s.age);
list.forEach(style);

//UnaryOperator takes a type and returns a value of the same type. Can replace Function if types are the same.
UnaryOperator<String> toLowerUsingUnary = (s) -> s.toLowerCase();
```

See docs for more information https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Method references

- When calling an already existing method existing method
- Double colon (ClassName::methodName)
- Refer to methods or constructors without invoking them.
- Four types:
  - Reference to a static method (`ContainingClass::staticMethodName`)
  - Reference to an instance method of a particular object (`containingObject::instanceMethodName`)
  - Reference to an instance method of an arbitrary object of a particular type (`ContainingType::methodName`)
  - Reference to a constructor (`ClassName::new`)

Method references - Reference to a static method

```
public class Person {
    String name;
    LocalDate birthday;

    public Calendar getBirthday() {
        return birthday;
    }

    public static int compareByAge(Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }
}
Arrays.sort(arrayOfPeople,
    (a, b) -> Person.compareByAge(a, b)    //Lambda
);
Arrays.sort(arrayOfPeople, Person::compareByAge);    //Java 8 Method reference
```

Method references - Reference to an instance method of a particular object

```
ComparisonProvider myComparisonProvider = new ComparisonProvider();

Arrays.sort(arrayOfPeople, myComparisonProvider::compareByName);
```

Method references - Reference to an instance method of an arbitrary object of a particular type

```
ContainingType::methodName
```

```
String[] stringArray = {
                "Barbara",
                "James",
                "Mary",
                "John",
                "Patricia",
                "Robert",
                "Michael",
                "Linda" };

Arrays.sort(stringArray, String::compareToIgnoreCase);
```

Method references - Reference to a constructor

```
public class Main {

    public String toString(){
        return "Main";
    }

    public static void main(String[] argv) {
       // Supplier<Main> supplier = () -> new Main(); //Lambda way
        Supplier<Main> supplier = Main::new;   //Java 8 method refernce
        System.out.println(supplier.get());
    }
}
```

Exercise 3

Objectives: Use lambdas, method references in code

https://github.com/pksolheim/java8kurs

a) Refactor com.capgemini.exercise3.Exercise3a.java to use lambda instead of an anonymous inner class
b) Make the implementation of com.capgemini.exercise3.Exercise3b (Exercise3bImpl) run using pre built funcitonal interfaces.

See https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html for information about the java.util.function package.

Streams

A sequence of objects somewhat like the Iterator interface. However, unlike the Iterator, it supports parallel execution.

- map
- sorted
- filter
- reduce
- findFirst
- allMatch

Executes lazily

Streams

Oops!

- Don’t modify the backing collection of a stream
- Always remember the terminating opertation

Streams

Files

```
try (Stream stream = Files.lines(Paths.get("file"))) {
    stream.forEach(System.out::println);
}
```

Text Patterns

```
Pattern pattern = Pattern.compile(",");
pattern.splitAsStream("a,b,c")
    .forEach(System.out::println);
```

Streams

Infinite/Finite streams

```
Stream.generate(() -> Math.random());

Stream.iterate(1, i -> i+1)
        .forEach(System.out::print);

IntStream.range(1, 11)
     .forEach(System.out::println);
```

Create streams

```
Stream<Integer> s = Stream.of(1, 2, 3);
Stream<Object> s2 = Arrays.stream(array);
```

Streams

Assume:

```
List<Invoice> invoices;
...
List<Invoice> expensiveInvoices =
        invoices.stream()
                    .filter(inv -> inv.getAmount() > 10000)
                .limit(5)
                .collect(Collectors.toList());
```

```
boolean expensive =
    invoices.stream()
            .allMatch(inv -> inv.getAmount() > 10000);
```

```
Optional<Invoice> =
    invoices.stream()
            .filter(inv -> inv.getCustomer() == Customer.MR_MONEYBAGS)
            .findAny();
```

Exercise 4

Objective: Putting it all together

1. Read the superheros-revealed.json file
2. Produce a comma separated string of first names
3. Find the least powerful superhero based on power rating
4. Is Blade in the list? Check for superheros with first or last name “Blade”
5. Convert all the superheros to muggles (set power rating to 0)

Further reading

https://leanpub.com/whatsnewinjava8/read

https://www.oreilly.com/learning/introducing-java-8

https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

http://zeroturnaround.com/rebellabs/java-parallel-streams-are-bad-for-your-health/

https://blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api/