
Java™

# Lambda Programming Lab

Simon Ritter
Angela Caicedo
Stephen Chin

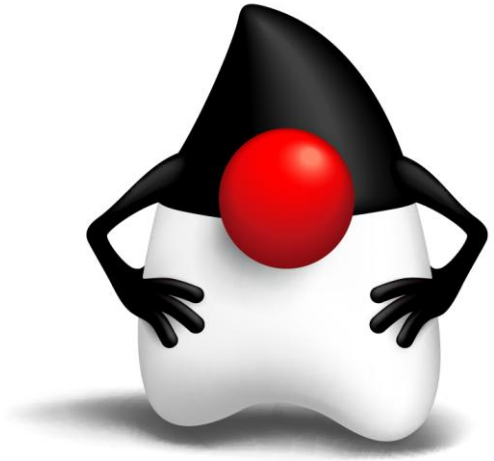
# Setting Up

- All lab software and materials are on USB keys
  - Please give them back
- Windows, Mac, Linux versions
- PDF with lab instructions

# Software Install

- Install JDK8
- Unpack API documentation
- Install NetBeans
- Install JUnit NetBeans Modules

# Lambdas and Functions Library Review



# Lambda Expressions

- Lambda expression is an anonymous function
- Think of it like a method
  - But not associated with a class
- Can be used wherever you would use an anonymous inner class
  - Single abstract method type
- Syntax
  - `([optional-parameters]) -> body`
- Types can be inferred (parameters and return type)

# Lambda Examples

```
SomeList<Student> students = ...
double highestScore =
  students.stream().
           filter(Student s -> s.getGradYear() == 2011).
           map(Student s -> s.getScore()).
           max();
```

# Method References

- Method references let us reuse a method as a lambda expression

```
FileFilter x = new FileFilter() {
  public boolean accept(File f) {
    return f.canRead();
  }
};
```

```
FileFilter x = (File f) -> f.canRead();
```

```
FileFilter x = File::canRead;
```

# The Stream Class

## java.util.stream

- **`Stream<T>`**
  - A sequence of elements supporting sequential and parallel operations
- A Stream is opened by calling:
  - **`Collection.stream()`**
  - **`Collection.parallelStream()`**
- Many Stream methods return Stream objects
  - Very simple (and logical) method chaining

# Stream Basics

- Using a Stream means having three things
- A source
  - Something that creates a **`Stream`** of objects
- Zero or more intermediate objects
  - Take a **`Stream`** as input, produce a **`Stream`** as output
  - Potentially modify the contents of the **`Stream`** (but don't have to)
- A terminal operation
  - Takes a **`Stream`** as input
  - Consumes the **`Stream`**, or generates some other type of output

# java.util.function Package

- **`Predicate<T>`**
  - Determine if the input of type T matches some criteria
- **`Consumer<T>`**
  - Accept a single input argumentof type T, and return no result
- **`Function<T, R>`**
  - Apply a function to the input type T, generating a result of type R
- Plus several more

# Using A Consumer (1)

## java.util.function

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

public void processPeople(List<Person> members,
                          Predicate<Person> predicate,
                          Consumer<Person> consumer) {
  for (Person p : members) {
    if (predicate.test(p))
      consumer.accept(p);
  }
}
```

# Using A Consumer (2)

```
processPeople(membership,
              p -> p.getGender() == Person.Gender.MALE && p.getAge() >= 65,
              p -> p.printPerson());
```

```
processPeople(membership,
              p -> p.getGender() == Person.Gender.MALE && p.getAge() >= 65,
              Person::printPerson);
```

# Using A Return Value (1)

## java.util.function

```
interface Function<T, R> {
  public R apply(T t);
}

public static void processPeopleWithFunction(
        List<Person> members,
        Predicate<Person> predicate,
        Function<Person, String> function,
        Consumer<String> consumer) {
  for (Person p : members) {
    if (predicate.test(p)) {
      String data = function.apply(p);
      consumer.accept(data);
    }
  }
}
```

# Using A Return Value (2)

```
processPeopleWithFunction(
  membership,
  p -> p.getGender() == Person.Gender.MALE && p.getAge() >= 65,
  p -> p.getEmailAddress(),
  email -> System.out.println(email));

processPeopleWithFunction(
  membership,
  p -> p.getGender() == Person.Gender.MALE && p.getAge() >= 65,
  Person::getEmailAddress,
  System.out::println);
```

# The iterable Interface

## Used by most collections

- One method
  - **`forEach()`**
  - The parameter is a **`Consumer`**

```
wordList.forEach(s -> System.out.println(s));

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

# Files and Lines of Text

- BufferedReader has new method
  - `Stream<String> lines()`

- HINT: Test framework creates a **`BufferedReader`** for you

# Maps and FlatMaps

## Map Values in a Stream

- One-to-one mapping
  - `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
  - `mapToDouble`, `mapToInt`, `mapToLong`
- One-to-many mapping
  - `<R> Stream<R> flatMap(`
    `Function<? super T, ? extends Stream<? extends R> mapper)`
  - `flatMapToDouble`, `flatMapToInt`, `flatMapToLong`

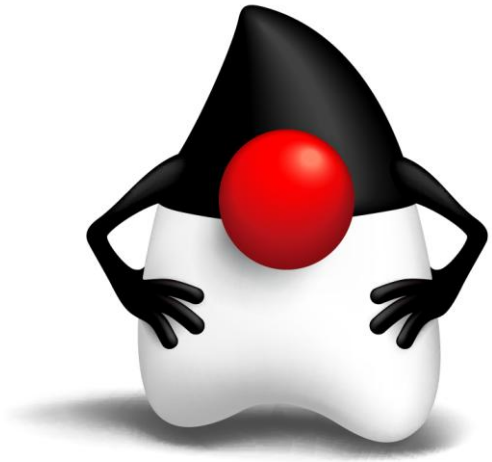
# Useful Stream Methods

- **`collect`** (terminal)
- **`filter`** (intermediate)
- **`count`** (terminal)
- **`skip`**, limit (intermediate)
- **`max`** (terminal)
- **`getAsInt`** (terminal)

# Getting Started

- Open the LambdasHOL project in NetBeans
- The exercises are configured as tests
- Edit the tests
  - Remove the `@Ignore` annotation
- Run the tests (Ctrl F6, or from the menu)
- Make the tests pass
- Simple!

# Let’s Go!



# Exercise 1: Solution

## Print all words in a list

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

# Exercise 2: Solution

## Convert words in list to upper case

```
List<String> output = wordList.
    stream().
    map(String::toUpperCase).
    collect(toList());
```

`toList` is a static method in the `Collectors` utiltity class

# Exercise 3: Solution

## Find words in list with even length

```
List<String> output = wordList.
  stream().
  filter(w -> (w.length() & 1 == 0).
  collect(toList());
```

# Exercise 4: Solution

## Count lines in a file

```
long count = reader.
  lines().
  count();
```

# Exercise 5: Solution

## Join lines 3-4 into a single string

```
String output = reader.
  lines().
  skip(2).
  limit(2).
  collect(joining());
```

**`joining`** is a static method in the **`Collectors`** utiltity class

# Exercise 6: Solution

## Find the length of the longest line in a file

```
int longest = reader.
  lines().
  mapToInt(String::length).
  max().
  getAsInt();
```

# Exercise 7: Solution

## Collect all words in a file into a list

```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  collect(toList());
```

# Exercise 8: Solution

List of words lowercased, in aphabetical order

```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  map(String::toLowerCase).
  sorted().
  collect(toList());
```

# Exercise 9: Solution

## Sort unique lower-case words by length then alphabetically

```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  map(String::toLowerCase).
  distinct().
  sorted(comparingInt(String::length).
    thenComparing(naturalOrder())).
  collect(toList());
```

# Exercise 10: Solution

## Categorize words into a map, key is length of each word

```
Map<Integer, List<String>> map = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  collect(groupingBy(String::length));
```

**groupingBy** is a static method in the **Collectors** utiltity class

# Exercise 11: Solution

## Gather words to map, with count of each words occurence

```
Map<String, Long> map = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  collect(groupingBy(Function.identity(), counting()));
```

**`counting`** is a static method in the **`Collectors`** utiltity class

# Exercise 12: Solution

## Nested grouping

```
Map<String, Map<Integer, List<String>>> map = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  collect(groupingBy(word -> word.substring(0, 1),
    groupingBy(String::length)));
```

# MAKE THE FUTURE JAVA


ORACLE®