Lambda NightHacking Lab

Stuart Marks
Brian Goetz
Stephen Chin
Angela Caicedo

Original Content Created with Simon Ritter

I AM A NIGHTHACKER

NIGHTHACKING.COM

**REAL GEEKS
LIVE HACKING**

NIGHTHACKING.COM

How to Participate

- Watch the live (or recorded) stream
    - http://nighthacking.com
- Join the conversation on Twitter
    - Message #nighthacking to ask questions
    - Watch the chat on twubs: http://twubs.com/NightHacking

- For important announcements follow @_NightHacking

Software Install

- Install JDK8 Build 121
  - https://jdk8.java.net/download.html
- Unpack API documentation
- Install NetBeans 7.4
  - https://netbeans.org/downloads/
- Install JUnit NetBeans Modules

Setting Up

- All lab software and materials are in GitHub
  - https://github.com/NightHacking/LambdasHacking
- Includes:
  - Source code
  - PDF with lab instructions
  - PDF with this presentation

Lambdas and Functions
Library Review
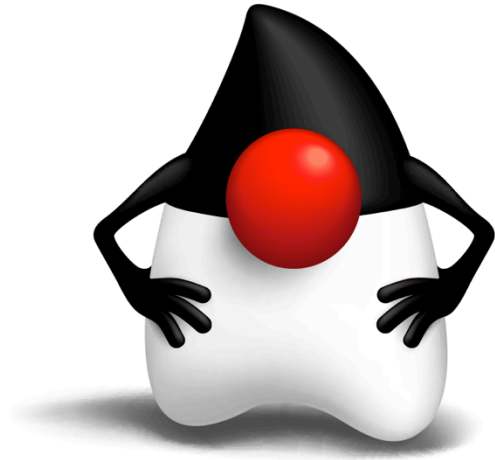
Lambda Expressions

- 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

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

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

```java
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 Usage

- Multiple operations available
  - **collect**, **filter**, **count**, **skip**, **limit**, **sorted**
  - **map** (and map to types, e.g. **mapToInt**)
  - **flatMap** maps each element in a **Stream** to possibly multiple elements
    - e.g. **flatMap(line -> Stream.of(line.split(REGEXP));**

```
List<String> names = Arrays.asList("Bob", "Alice", "Charlie");
System.out.println(names.
  stream().
  filter(e -> e.getLength() > 4).
  findFirst().
  get());
```

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

```java
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

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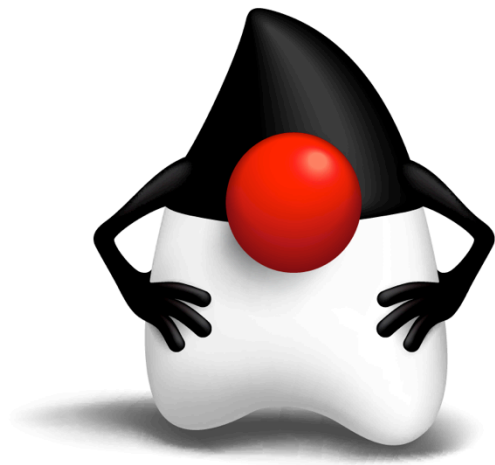
```
wordList.forEach(s -> System.out.println(s));

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

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!

Java™

ORACLE®

Exercise 1: Solution

Print all words in a list

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

Java™

ORACLE®

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

```java
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();
```

ORACLE

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

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

Java™

ORACLE®

Exercise 7: Solution

Collect all words in a file into a list

```java
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

```java
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

```java
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