Java 8
Lambda Expressions and You

Technion – Institute of Technology
Software Design (236700)

Based on slides by: Sagie Davidovich, Assaf Israel

Author: Gal Lalouche - Technion 2016 ©
Life before Java 8 (cont.)

Extracting employee names

```java
public List<String> empNames(List<Employee> employees) {
    List<String> result = new ArrayList<>();
    for (Employee emp: employees) {
        result.add(emp.getName());
    }
    return result;
}
```

Extracting employee ages

```java
public List<Integer> empAges(List<Employee> employees) {
    List<Integer> result = new ArrayList<>();
    for (Employee emp: employees) {
        result.add(emp.getAge());
    }
    return result;
}
```
Life before Java 8 (cont.)

Let's identify the control structure, and extract the behavior into an object.

```java
public List<String> empNames(List<Employee> employees) {
    List<String> result = new ArrayList<>();
    for (Employee emp: employees) {
        result.add(emp.getName());
    }
    return result;
}
```

```java
public interface Mapper<U, T> {
    public T map(U u);
}
```

```java
public <U, T> List<T> map(List<U> aList, Mapper<? super U, ? extends T> mapper) {
    List<T> result = new ArrayList<>();
    for (U u: aList) {
        result.add(mapper.map(u));
    }
    return result;
}
```
Extracting employee names

List<String> empNames = map(employees, new Mapper<Employee, String>() {
    public String map(Employee e) {
        e.getName();
    }
});

Extracting employee ages

List<Integer> empAges = map(employees, new Mapper<Employee, Integer>() {
    public Integer map(Employee e) {
        e.getAge();
    }
});
In the Kingdom of Nouns

We removed the code duplication, but this is still very verbose…

➤ Semantically, map is a higher level function
  ➤ This means that it accepts a function as an argument (or returns a function)

➤ Syntactically, functions do not exist as first class entities
  ➤ All verbs (functions) have be accompanied by a noun (class)
  ➤ http://steve-yegge.blogspot.co.il/2006/03/execution-in-kingdom-of-nouns.html

➤ Prior to Java 8, Java was the only programming language in popular use without anonymous functions / blocks / lambdas / function pointers
  ➤ This is not purely a syntactic issue; Java also lacked proper support for such function in its collections and standard libraries
  ➤ Some libraries, like Guava, attempted to fill the void
Enter Java 8!

Extracting employee names

```
List<String> empNames = employees.stream()
    .map(x -> x.getName())
    .collect(Collectors.toList());
```

Extracting employee ages

```
List<Integer> empAge = employees.stream()
    .map(x -> x.getAge())
    .collect(Collectors.toList());
```
Let’s take a deeper look…

List<String> empNames = employees.stream()
    .map(x -> x.getName())
    .collect(Collectors.toList());

- stream() is a **default** method of List
- map is a higher level function of Stream
- x -> x.getName() is a **lambda expression**
- collect turns the Stream back to a normal Collection (in our case, a List)
- But what does each of these terms mean?
**default Methods**

List<String> empNames = employees.stream()
.map(x -> x.getName())
.collect(Collectors.toList());

- **default** methods are default implementations for **interfaces**
- This way you can add **new** functionality to an existing interface without **breaking** all depending code
  - In our case, we added the `stream()` method to `Collection`
- So is this the same as **multiple inheritance**?
  - Not exactly, the behavior more closely resembles **Traits**, which we will discuss in detail when we talk about **Scala**
  - For now, suffice to say that since there is neither conflict resolution nor constructors, the model is much **simpler**
- So are these **extension methods** (a la C#)?
  - No, because extension methods are actually **syntactic sugar** for **static decorators**
  - You can’t add methods yourself to existing classes (e.g., in C# you can add extension methods to `String`).
Higher order functions

List<String> empNames = employees.stream()
  .map(x -> x.getName())
  .collect(Collectors.toList());

- **map** is a higher order function in stream
  - That is, a function that takes a function
- There are other similar higher order functions in stream
  - filter, map, flatMap, sorted, reduce...
- Similar solution exist in other languages
  - LINQ in C#, itertools in Python, Enumerable in Ruby, etc.
Higher order functions – API examples

- What's this?
  ```java
  employees.stream().noneMatch(x -> x.age < 18);
  ```
- Find the highest paid individual in the company
  ```java
  Optional<Employee> $ = employees.stream().max((x,y) -> x.salary - y.salary);
  ```
- What is returned if the list is empty?
- Instead of working with `null`, a new type `Optional<T>` is returned
  ```java
  Optional<T> can be present (i.e. not `null`) or not (i.e. `null`)
  ```
  ```java
  Has a method `get()` that returns `T` or throws an exception
  ```
- `Optional<T>` itself has higher order functions
  ```java
  Optional<Integer> ageOfRichest = $.map(x -> x.getAge);
  ```

Author: Gal Lalouche - Technion 2016©
Higher order functions – Map-Reduce

- Sum of all salaries in the company

```java
employees.stream()
    .mapToInt(Employee::getSalary) // note the mapToInt... why?
    .reduce(0, Integer::sum)
// could also be done with Lambdas, or simply .sum()
```

- There are programming models that work on the MapReduce principle
  - e.g. Hadoop
Higher Order function – A more complex example

- Get Israeli students with a top grade sorted by name (in Java 7)

```java
List<Student> topGrades = new ArrayList<>();
Collections.sort(students, new Comparator<Student>() {
    public int compare(Student student1, Student student2) {
        return student1.getName().compareTo(student2.getName());
    }
});
for (Student student: students)
    if (student.getCountry() == "Israel")
        if (student.getGrade() >= 90)
            topGrades.add(student);
```

- In Java 8:

```java
List<Students> topStudents = students.stream()
    .filter(x -> x.getCountry() == "Israel")
    .filter(x -> x.getGrade() > 90)
    .sorted((x, y) -> y.getGrade() - x.getGrade())
    .collect(Collectors.toList());
```

Sorts in place! Why is this bad?

Depth of 3!
Streams

- We only iterate over a stream once, even if we have two (or more) higher level functions
- This is because streams are **lazily evaluated**
  - until we call `collect`, no iteration takes place
  - `collect` is actually a form of **mutable reduction**
    - i.e., it reduces to a mutable container
- They also provide us with a **uniform API**
- Streams also give us “free” **parallelization** (why is it so easy?)

```java
List<Students> topStudents = students.stream().parallel()
  .filter(x -> x.getCountry() == "Israel")
  .filter(x -> x.getGrade() > 90)
  .sorted((x, y) -> y.getGrade() - x.getGrade())
  .collect(Collectors.toList());
```
Lambdas and Default Methods

The signature for `map` is:

```java
map(Function<? super T, ? extends R> mapper)
```

And here is the signature for `Function`:

```java
interface Function<T, R> {
    R apply(T t);
    // default methods retracted
}
```

- An interface which has single abstract (i.e., non-default) method can be called a Functional Interface
- So is Java now a functional language?
  - Nope
  - Lambdas are just syntactic sugar for implementing Functional Interfaces
  - Functions aren’t first-class citizens, since functions aren’t even a proper part of the Java language, just a standard library interface

Author: Gal Lalouche - Technion 2016©
We can also use lambda with "old" API!

- **Old code**

```java
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Kill me :([";
    }
}).start();
```

- **New code**

```java
new Thread(() -> System.out.println("PARTEH! :D|< <D/-- :D"<D").start();
```

- **We can use the convenience `@FunctionalInterface` annotation to tell the compiler that the interface should be functional (à la `@Override`)**

```java
@FunctionalInterface
interface I { void A(); void B(); } // won't compile
```
What else is new in Java 8?

- **New Date and Time APIs**
  - Everything is now immutable, and **immutable is good**
- **Support for unsigned arithmetic** (no `uint` type)
- **Embedding JavaScript code in Java**

```java
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval("console.log('Hello World!');");
```

- **Better integration with JavaFX**
  - Java library for developing **rich client applications**
  - Alternative to swing, which is no longer in active development