Lecture 8

Lambda Expressions

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

1. Introduction
2. Functional interfaces
3. Type Checking and Inference
4. Method References
5. Application
6. Composing lambda expressions
7. Summary
8. Notes and Further Reading

René Witte

Introduction
- Lambda Expressions
- Functional interfaces
- Function Descriptors
- Execute Around Pattern

Functional interfaces
- Predicate
- Consumer
- Function
- Primitive Specializations
- Examples

Type Checking and Inference
- Type checking
- Type Inference
- Closures

Method References
- Constructor references

Application
- Step 1: Pass code
- Step 2: Anonymous class
- Step 3: Lambda expression
- Step 4: Method references

Composing lambda expressions
- Composing Comparators
- Composing Predicates
- Composing Functions

Summary

Notes and Further Reading

Outline

1. **Introduction**
   - Lambda Expressions
   - Functional interfaces
   - Function Descriptors
   - Execute Around Pattern
2. **Functional interfaces**
3. **Type Checking and Inference**
4. **Method References**
5. **Application**
6. **Composing lambda expressions**
7. **Summary**
8. **Notes and Further Reading**

Sorting with a Comparator

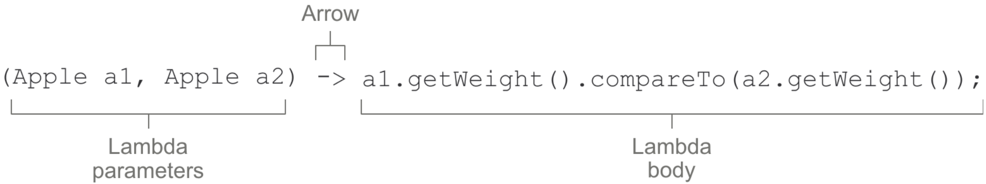
Creating a `Comparator`

```
Comparator<Apple> byWeight = new Comparator<Apple>() {
  public int compare(Apple a1, Apple a2){
    return a1.getWeight().compareTo(a2.getWeight());
  }
};
```

With a lambda expression

```
Comparator<Apple> byWeight =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Lambda expressions



```
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Arrow

Lambda parameters

Lambda body

Lambda expressions (II)

Anonymous Functions

Anonymous: no function name

Function: has parameters, return type, function body, exceptions; but **not** associated with a class

Passed around: can be passed as an argument to a method or stored in a variable

Concise: no need for boilerplate code, like for anonymous classes

Name comes from lambda calculus, see https://en.wikipedia.org/wiki/Lambda_calculus.

Java 8 Syntax

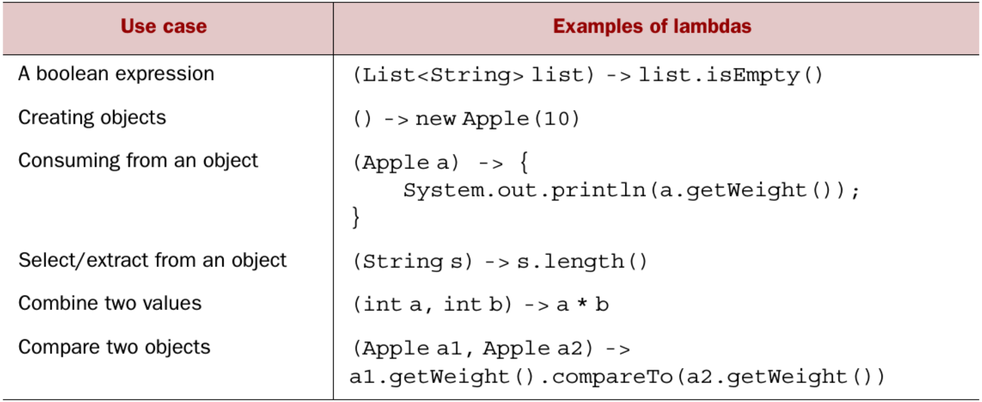
Single expressions:

```
(parameters) -> expression
```

List of statements:

```
(parameters) -> { statements; }
```

Lambda expressions (III)



Use case	Examples of lambdas
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`

Consuming from an object	`(Apple a) -> {`
	`    System.out.println(a.getWeight());`
	`}`

Select/extract from an object	`(String s) -> s.length()`
Combine two values	`(int a, int b) -> a * b`

Compare two objects	`(Apple a1, Apple a2) ->`
	`a1.getWeight().compareTo(a2.getWeight())`

Copyright 2015 by Manning Publications Co., [UFM14]

Functional Interfaces

Definition

A functional interface is an interface that specifies exactly one abstract method.

Example

```
public interface Predicate<T>{
  boolean test (T t);
}
```

Functional Interface Examples

java.util.Comparator

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

java.lang.Runnable

```
public interface Runnable{
  void run();
}
```

java.awt.event.ActionListener

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

Functional Interfaces and Lambdas

```
Runnable r1 = () -> System.out.println("Hello␣World␣1");

Runnable r2 = new Runnable(){
  public void run(){
    System.out.println("Hello␣World␣2");
  }
};

public static void process(Runnable r){
  r.run();
}

process(r1);
process(r2);
process(() -> System.out.println("Hello␣World␣3"));
```

Function Descriptors

Definition

The abstract method in the functional interface is called a function descriptor. The signature of the abstract method describes the signature of the lambda expression.

Example

```
public void process(Runnable r){
  r.run();
}

process(() -> System.out.println("This␣is␣awesome!!"));
```

Note

You can pass lambdas only where a functional interface is expected

The Execute Around Pattern

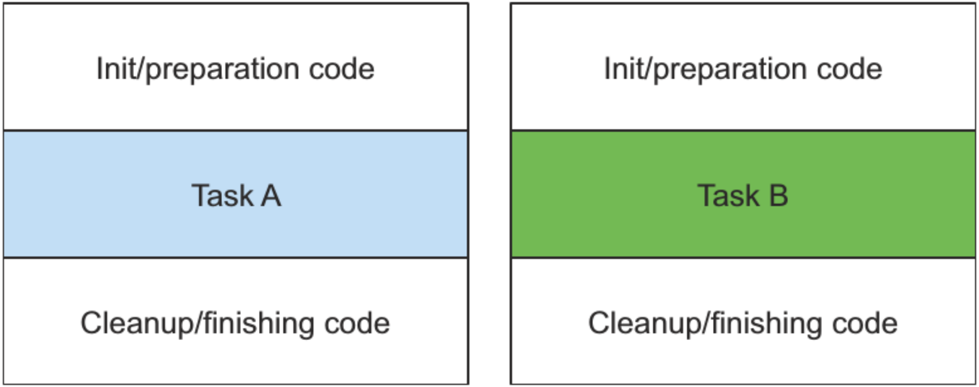
Pattern in resource processing (file, database, . . .)

1. Open a resource
2. Process the resource
3. Close the Resource

Example: File processing in Java 7

```
public static String processFile() throws IOException {
  try (BufferedReader br = new BufferedReader(
                               new FileReader("data.txt"))) {
    return br.readLine();
  }
}
```

Step 1: Behavior parameterization



Copyright 2015 by Manning Publications Co., [UFM14]

Example: Process two lines

```
String result = processFile((BufferedReader br) ->
                              br.readLine() + br.readLine());
```

Step 2: Use a functional interface

Define the Interface

```
@FunctionalInterface
public interface BufferedReaderProcessor {
  String process(BufferedReader b) throws IOException;
}
```

Usage

```
public static String processFile(BufferedReaderProcessor p)
throws IOException {
  ...
}
```

Step 3: Execute a behavior

```java
public static String processFile(BufferedReaderProcessor p)
       throws IOException {
  try (BufferedReader br = new BufferedReader(
                               new FileReader("data.txt"))) {
    return p.process(br);
  }
}
```

Step 4: Pass lambdas

Process one line

```
String oneLine = processFile((BufferedReader br) -> br.readLine());
```

Process two lines

```
String twoLines =
  processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

Outline

1 Introduction

2 Functional interfaces
- Predicate
- Consumer
- Function
- Primitive Specializations
- Examples

3 Type Checking and Inference

4 Method References

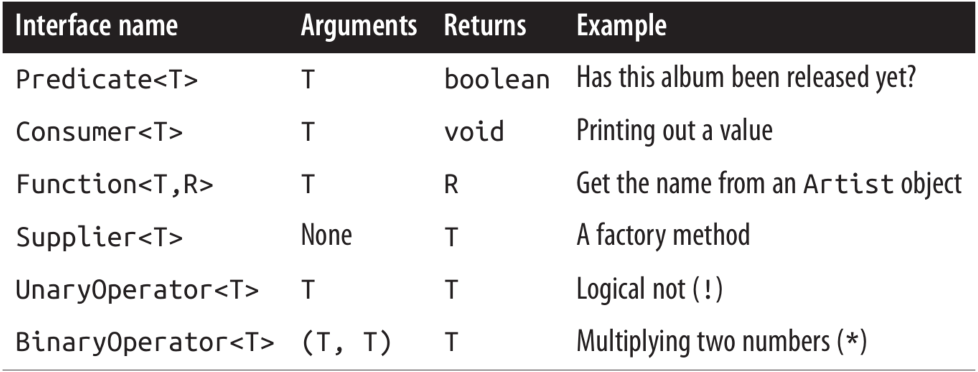
5 Application

6 Composing lambda expressions

7 Summary

8 Notes and Further Reading

New Functional Interfaces in Java 8: Examples



Interface name	Arguments	Returns	Example
`Predicate<T>`	T	boolean	Has this album been released yet?
`Consumer<T>`	T	void	Printing out a value
`Function<T,R>`	T	R	Get the name from an `Artist` object
`Supplier<T>`	None	T	A factory method
`UnaryOperator<T>`	T	T	Logical not (!)
`BinaryOperator<T>`	(T, T)	T	Multiplying two numbers (*)

Copyright 2014 by Richard Warburton, [War14]

Predicate

```
@FunctionalInterface
public interface Predicate<T>{
  boolean test(T t);
}

public static <T> List<T> filter(List<T> list, Predicate<T> p) {
  List<T> results = new ArrayList<>();
  for(T s: list){
    if(p.test(s)){
      results.add(s);
    }
  }
  return results;
}

Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
```

Consumer

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

public static <T> void forEach(List<T> list, Consumer<T> c){
  for(T i: list){
    c.accept(i);
  }
}

forEach(Arrays.asList(1,2,3,4,5), (Integer i) -> System.out.println(i));
```

Function

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

public static <T, R> List<R> map(List<T> list, Function<T, R> f) {
  List<R> result = new ArrayList<>();
  for(T s: list){
    result.add(f.apply(s));
  }
  return result;
}

List<Integer> l = map(Arrays.asList("lambdas","in","action"),
                      (String s) -> s.length()
                     );
// l = ???
```

Primitive Types

Primitive Types vs. Reference Types

Primitive Types: `int, double, byte, char, ...`

Reference Types: `Byte, Integer, Object, List, ...`

Boxing: converting a primitive type into a reference type

Unboxing: converting a reference type into a primitive type

Autoboxing

```java
List<Integer> list = new ArrayList<>();
for (int i = 300; i < 400; i++){
  list.add(i);
}
```

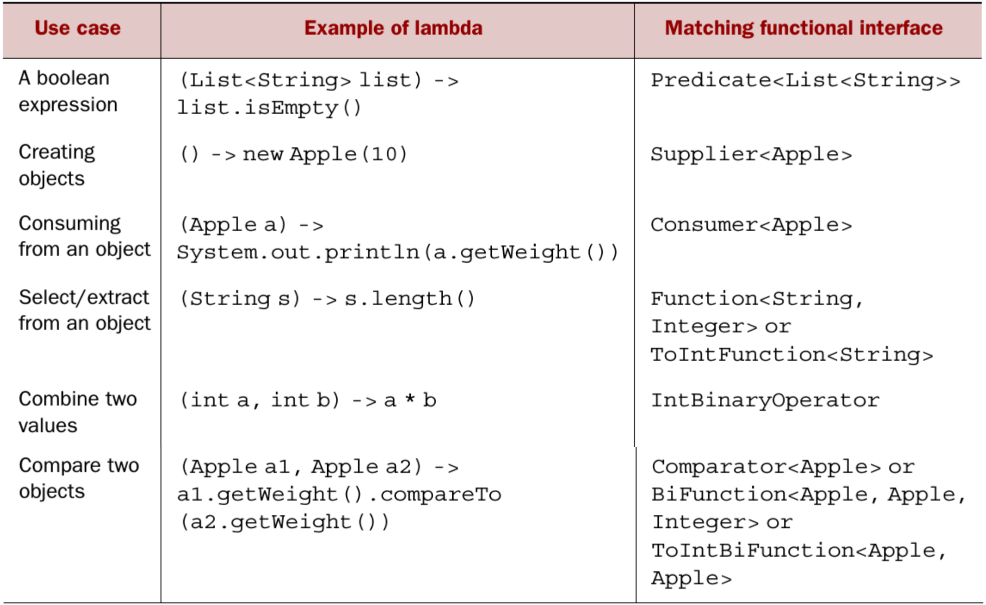
Primitive Specializations

```
public interface IntPredicate{
  boolean test(int t);
}

IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);

Predicate<Integer> oddNumbers = (Integer i) -> i % 2 == 1;
oddNumbers.test(1000);
```

Examples



Use case	Example of lambda	Matching functional interface
A boolean expression	`(List<String> list) -> list.isEmpty()`	`Predicate<List<String>>`
Creating objects	`() -> new Apple(10)`	`Supplier<Apple>`
Consuming from an object	`(Apple a) -> System.out.println(a.getWeight())`	`Consumer<Apple>`
Select/extract from an object	`(String s) -> s.length()`	`Function<String, Integer>` or `ToIntFunction<String>`
Combine two values	`(int a, int b) -> a * b`	`IntBinaryOperator`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo (a2.getWeight())`	`Comparator<Apple>` or `BiFunction<Apple, Apple, Integer>` or `ToIntBiFunction<Apple, Apple>`

Copyright 2015 by Manning Publications Co., [UFM14]

Outline

1 Introduction

2 Functional interfaces

3 Type Checking and Inference
- Type checking
- Type Inference
- Closures

4 Method References

5 Application

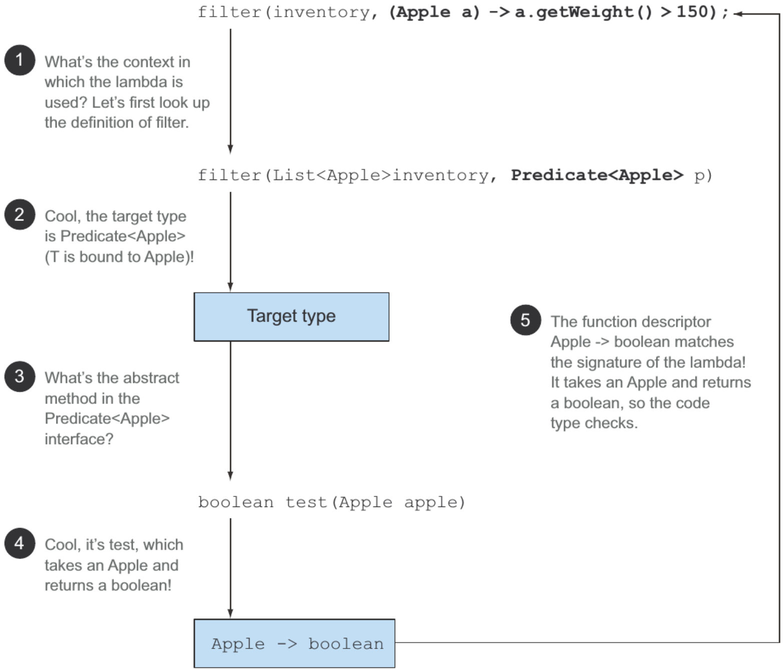
6 Composing lambda expressions

7 Summary

8 Notes and Further Reading

Type checking

```
List<Apple> heavierThan150g =
    filter(inventory, (Apple a) -> a.getWeight() > 150);
```



`filter(inventory, (Apple a) -> a.getWeight() > 150);`

1 What's the context in which the lambda is used? Let's first look up the definition of filter.

`filter(List<Apple>inventory, Predicate<Apple> p)`

2 Cool, the target type is Predicate<Apple> (T is bound to Apple)!

Target type

3 What's the abstract method in the Predicate<Apple> interface?

`boolean test(Apple apple)`

4 Cool, it's test, which takes an Apple and returns a boolean!

`Apple -> boolean`

5 The function descriptor Apple -> boolean matches the signature of the lambda! It takes an Apple and returns a boolean, so the code type checks.

Introduction
Lambda Expressions
Functional interfaces
Function Descriptors
Execute Around Pattern
Functional interfaces
Predicate
Consumer
Function
Primitive Specializations
Examples
Type Checking and Inference
Type checking
Type Inference
Closures
Method References
Constructor references
Application
Step 1: Pass code
Step 2: Anonymous class
Step 3: Lambda expression
Step 4: Method references
Composing lambda expressions
Composing Comparators
Composing Predicates
Composing Functions
Summary
Notes and Further Reading

Target Typing

Same lambda, different functional interfaces

```
Callable<Integer> c = () -> 42;
PrivilegedAction<Integer> p = () -> 42;
```

Compare with Java 7 Diamond Operator

```
List<String> listOfStrings = new ArrayList<>();
List<Integer> listOfIntegers = new ArrayList<>();
```

Special void-compatibility rule

```
// Predicate has a boolean return
Predicate<String> p = s -> list.add(s);
// Consumer has a void return
Consumer<String> b = s -> list.add(s);
```

Type Checking: Example

Does it compile?

```
Object o = () -> {System.out.println("Tricky␣example"); };
```

Error

```
Quiz.java:5: error: incompatible types: Object is not a functional interface
    Object o = () -> {System.out.println("Tricky␣example"); };
               ^
1 error
```

Type Inference

Java compiler can infer lambda parameter types

```
List<Apple> greenApples =
    filter(inventory, a -> "green".equals(a.getColor()));
```

Without type inference

```
Comparator<Apple> c =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

With type inference

```
Comparator<Apple> c =
    (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Using local variables

Capturing lambdas

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
```

Rules

Lambdas can capture:

- instance variables
- static variables
- `final` local variables
- effectively `final` local variables

Error

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
portNumber = 31337;
```

```
Error: local variables referenced from a lambda expression must
be final or effectively final.
```

Closures

Definition

A closure is an instance of a function that can reference nonlocal variables of that function with no restrictions.

Closures in Java 8?

- Java 8 does not allow lambdas to modify variables defined outside its scope
- However, they can be passed as method arguments
- and read variables outside their scope

We can say Java 8 lambdas close over values, rather than variables

Method References

Passing an existing method

```
inventory.sort((Apple a1, Apple a2) ->
    a1.getWeight().compareTo(a2.getWeight()));
```

With a method reference

```
inventory.sort(comparing(Apple::getWeight));
```

Improve code readability

Instead of

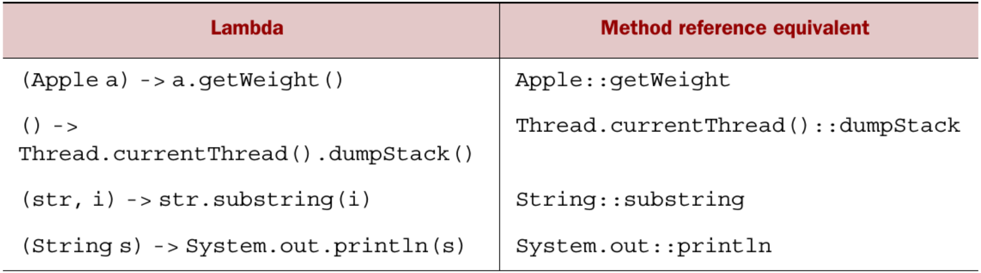
```
(Apple a) -> a.getWeight()
```

you can just say

```
Apple::getWeight
```

Note: no brackets `()` – we are not calling the method!

Method reference examples



Lambda	Method reference equivalent
`(Apple a) -> a.getWeight()`	`Apple::getWeight`
`() -> Thread.currentThread().dumpStack()`	`Thread.currentThread()::dumpStack`
`(str, i) -> str.substring(i)`	`String::substring`
`(String s) -> System.out.println(s)`	`System.out::println`

Constructing method references

1. Reference to a static method, e.g., `Integer::parseInt`
2. Reference to an instance method of an arbitrary type, e.g., `String::length`
3. Reference to an instance method of an existing object, e.g., with a variable `expensiveTransaction` of type `Transaction` that has an instance method `getValue`, write `expensiveTransaction::getValue`

Constructor references

Calling a constructor

```
Supplier<Apple> c1 = () -> new Apple();
Apple a1 = c1.get();
```

Now with constructor reference

```
Supplier<Apple> c1 = Apple::new;
Apple a1 = c1.get();
```

Introduction
Lambda Expressions
Functional interfaces
Function Descriptors
Execute Around Pattern
Functional interfaces
Predicate
Consumer
Function
Primitive Specializations
Examples
Type Checking and Inference
Type checking
Type Inference
Closures
Method References
Constructor references
Application
Step 1: Pass code
Step 2: Anonymous class
Step 3: Lambda expression
Step 4: Method references
Composing lambda expressions
Composing Comparators
Composing Predicates
Composing Functions
Summary
Notes and Further Reading

Constructor references (II)

Calling a non-default constructor

```
Function<Integer, Apple> c2 = (weight) -> new Apple(weight);
Apple a2 = c2.apply(110);
```

Now with constructor reference

```
Function<Integer, Apple> c2 = Apple::new;
Apple a2 = c2.apply(110);
```

Outline

1. Introduction
2. Functional interfaces
3. Type Checking and Inference
4. Method References
5. Application
   - Step 1: Pass code
   - Step 2: Anonymous class
   - Step 3: Lambda expression
   - Step 4: Method references
6. Composing lambda expressions
7. Summary
8. Notes and Further Reading

Putting lambdas and method references into practice

```
inventory.sort(comparing(Apple::getWeight));
```

Step 1: Pass code

Java 8 `List.sort`

```
void sort(Comparator<? super E> c)
```

Implement Comparator (behavior parameterization!)

```
public class AppleComparator implements Comparator<Apple> {
  public int compare(Apple a1, Apple a2){
    return a1.getWeight().compareTo(a2.getWeight());
  }
}

inventory.sort(new AppleComparator());
```

Step 2: Use an anonymous class

```java
inventory.sort(new Comparator<Apple>() {
  public int compare(Apple a1, Apple a2){
    return a1.getWeight().compareTo(a2.getWeight());
  }
});
```

Step 3: Use lambda expressions

Goal: Passing Code!

- Can use lambda expression where a functional interface is expected
- `Comparator`: function descriptor `(T, T) -> int`
- here, `(Apple, Apple) -> int`

```
inventory.sort((Apple a1, Apple a2) ->
    a1.getWeight().compareTo(a2.getWeight()));
```

Shorter with type inference

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

More readable with `comparing` helper method

With

```
import static java.util.Comparator.comparing;
Comparator<Apple> c = Comparator.comparing((Apple a) -> a.getWeight());
```

we can write

```
inventory.sort(comparing((a) -> a.getWeight()));
```

Step 4: Use method references

```
inventory.sort(comparing(Apple::getWeight));
```

Outline

1 Introduction

2 Functional interfaces

3 Type Checking and Inference

4 Method References

5 Application

6 **Composing lambda expressions**
- Composing Comparators
- Composing Predicates
- Composing Functions

7 Summary

8 Notes and Further Reading

Introduction
Lambda Expressions
Functional interfaces
Function Descriptors
Execute Around Pattern
Functional interfaces
Predicate
Consumer
Function
Primitive Specializations
Examples
Type Checking and Inference
Type checking
Type Inference
Closures
Method References
Constructor references
Application
Step 1: Pass code
Step 2: Anonymous class
Step 3: Lambda expression
Step 4: Method references
Composing lambda expressions
Composing Comparators
Composing Predicates
Composing Functions
Summary
Notes and Further Reading

Composing Comparators

Reversed Order

```
inventory.sort(comparing(Apple::getWeight).reversed());
```

Chaining Comparators

```
inventory.sort(comparing(Apple::getWeight)
         .reversed()
         .thenComparing(Apple::getCountry));
```

Composing Predicates

`Predicate` interface

Additional methods: `negate, and, or`

Example: Apples that are *not* red

```java
Predicate<Apple> notRedApple = redApple.negate();
```

Example: Apples that are red *and* heavy

```java
Predicate<Apple> redAndHeavyApple
    = redApple.and(a -> a.getWeight() > 150);
```

Example: Apples that are red *and* heavy *or* green

```java
Predicate<Apple> redAndHeavyAppleOrGreen =
    redApple.and(a -> a.getWeight() > 150)
            .or(a -> "green".equals(a.getColor()));
```

Composing Functions

Math: function composition

Let $f(x) = x + 1$ and $g(x) = x \cdot 2$, then we can compose both functions, written $g(f(x))$ or $(g \circ f)(x)$.

Java 8: `andThen`

To implement $(g \circ f)(x)$, use `andThen`

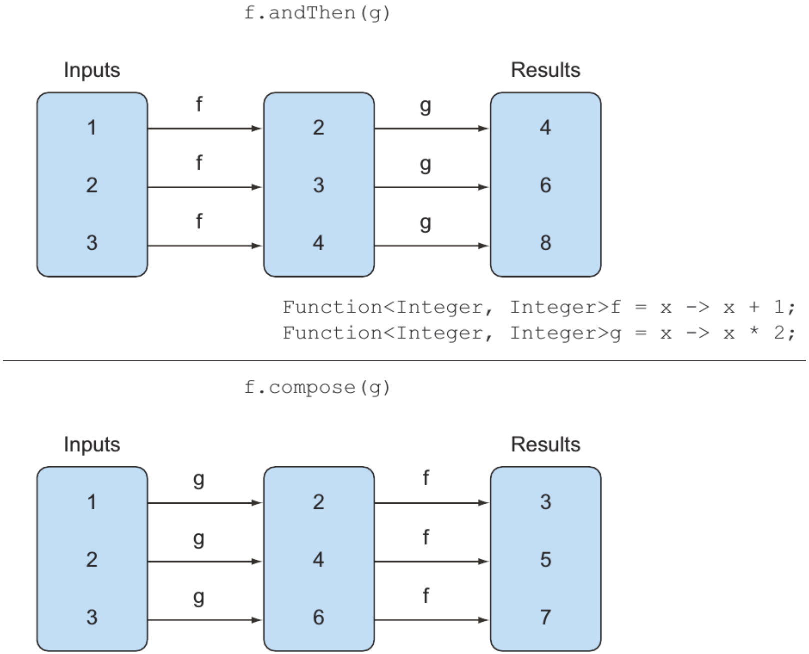
```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.andThen(g);
int result = h.apply(1);
```

Java 8: `compose`

To implement $(f \circ g)(x)$, use `compose`

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g);
int result = h.apply(1);
```

andThen vs. compose



```
Function<Integer, Integer>f = x -> x + 1;
Function<Integer, Integer>g = x -> x * 2;
```

Copyright 2015 by Manning Publications Co., [UFM14]

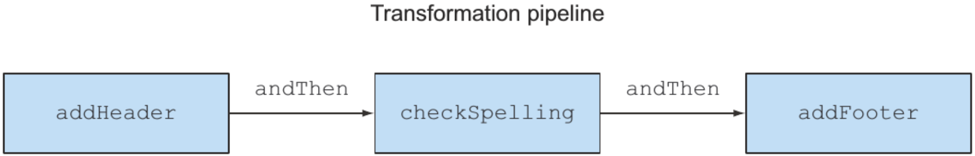
Example: Transformation Pipelines

```
public class Letter{
  public static String addHeader(String text){
    return "From␣Raoul,␣Mario␣and␣Alan:␣" + text;
  }

  public static String addFooter(String text){
    return text + "␣Kind␣regards";
  }

  public static String checkSpelling(String text){
    return text.replaceAll("labda", "lambda");
  }
}
```

Using function composition



Copyright 2015 by Manning Publications Co., [UFM14]

Creating the pipeline

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
    = addHeader.andThen(Letter::checkSpelling)
               .andThen(Letter::addFooter);
```

Summary

Java 8 Lambdas

- A lambda expression can be understood as a kind of anonymous function: it doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown.
- Lambda expressions let you pass code concisely
- A functional interface is an interface that declares exactly one abstract method
- Lambda expressions can be used only where a functional interface is expected
- Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and treat the whole expression as an instance of a functional interface

Summary (contd.)

Java 8 Lambdas (II)

- Java 8 comes with a list of common functional interfaces in the `java.util.function` package, which includes `Predicate<T>`, `Function<T,R>`, `Supplier<T>`, `Consumer<T>`, and `BinaryOperator<T>`
- There are primitive specializations of common generic functional interfaces such as `Predicate<T>` and `Function<T, R>` that can be used to avoid boxing operations: `IntPredicate`, `IntToLongFunction`, and so on
- The execute around pattern can be used with lambdas to gain additional flexibility and reusability
- The type expected for a lambda expression is called the target type
- Method references let you reuse an existing method implementation and pass it around directly
- Functional interfaces such as `Comparator`, `Predicate`, and `Function` have several default methods that can be used to combine lambda expressions.

Outline

1. Introduction
2. Functional interfaces
3. Type Checking and Inference
4. Method References
5. Application
6. Composing lambda expressions
7. Summary
8. Notes and Further Reading

Reading Material

Required

- [UFM14, Chapter 3] (Lambda Expressions)

Supplemental

- [War14, Chapter 2] (Lambda Expressions)

References

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.

[War14] Richard Warburton.
Java 8 Lambdas.
O'Reilly, 2014.