Lambdas and Streams

Java How to Program, 10/e

17.1 Introduction

- Prior to Java SE 8, Java supported three programming paradigms—procedural programming, object-oriented programming and generic programming. Java SE 8 adds functional programming.
- The new language and library capabilities that support functional programming were added to Java as part of Project Lambda.
- This chapter presents many examples of functional programming, often showing simpler ways to implement tasks that you programmed in earlier chapters (Fig. 17.1).

17.2 Functional Programming Technologies Overview

- Prior to functional programming, you typically determined what you wanted to accomplish, then specified the precise steps to accomplish that task.
- External iteration
  - Using a loop to iterate over a collection of elements.
  - Requires accessing the elements sequentially.
  - Requires mutable variables.

17.2 Functional Programming Technologies Overview (Cont.)

- Functional programming
  - Specify what you want to accomplish in a task, but not how to accomplish it
- Internal iteration
  - Let the library determine how to iterate over a collection of elements is known as.
  - Internal iteration is easier to parallelize.
- Functional programming focuses on immutability—not modifying the data source being processed or any other program state.
- **Programming Languages that support functional programming:** Haskell, JavaScript, Python, Scala, Erlang, Lisp, ML, Clojure, OCaml, Common Lisp, Racket.

Lambda Expression

- Lambda expression is a new and important feature of Java which was included in Java SE 8.
- It provides a clear and concise way to represent one method interface using an expression.
- It is very useful in collection library. It helps to iterate, filter and extract data from collection.

Functional Interface

- Lambda expression provides implementation of *functional interface*.
- An interface which has only one abstract method is called functional interface.
- Java provides an annotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Ideal Use Case for Lambda Expressions

- Suppose that you are creating a social networking application. You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria.

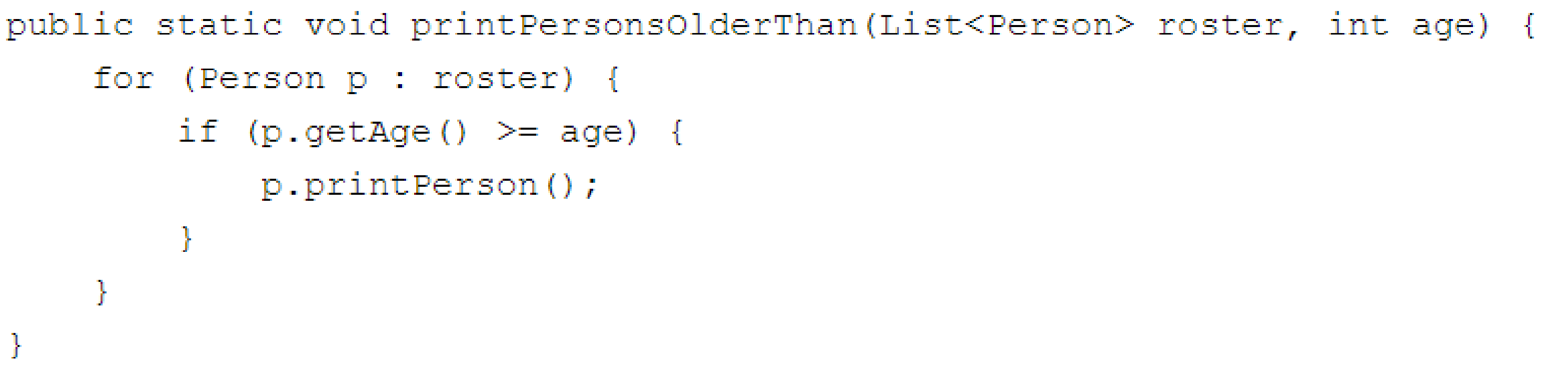
- Suppose that members of this social networking application are represented by the following Person class:

```
public class Person {
    public enum Sex {
        MALE, FEMALE
    }
    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

- Suppose that the members of your social networking application are stored in a List<Person> instance.

Approach 1: Create Methods That Search for Members That Match One Characteristic

- One simplistic approach is to create several methods; each method searches for members that match one characteristic, such as gender or age. The following method prints members that are older than a specified age:



```java
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

- This approach can potentially make your application brittle, which is the likelihood of an application not working because of the introduction of updates (such as newer data types).
- Suppose that you upgrade your application and change the structure of the Person class such that it contains different member variables; perhaps the class records and measures ages with a different data type or algorithm.
- You would have to rewrite a lot of your API to accommodate this change.
- In addition, this approach is unnecessarily restrictive; what if you wanted to print members younger than a certain age, for example?

Approach 2: Create More Generalized Search Methods

- For eg. Write methods to filter by range of age

Approach 3: Use interface for specifying search criteria

- To specify the search criteria, you implement the CheckPerson interface:

```
interface CheckPerson {
    boolean test(Person p);
}
```

```
Then use the method:
```

- The following method prints members that match search criteria that you specify:

```
public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

- The following class implements the CheckPerson interface by specifying an implementation for the method test. It returns a true value if its Person parameter is male and between the ages of 18 and 25:

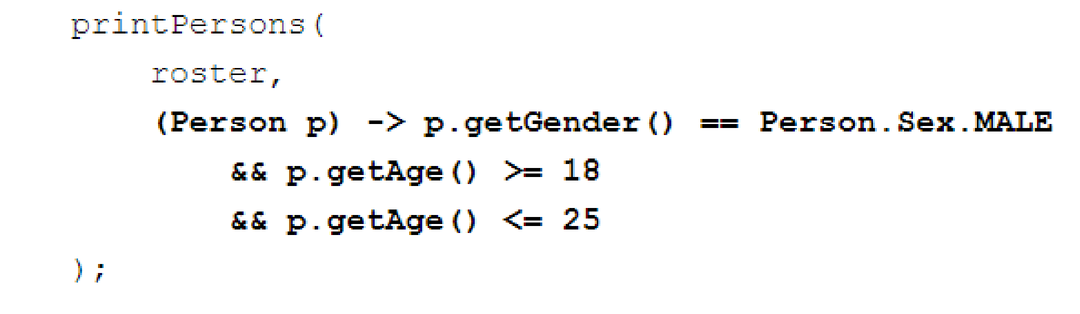
```
   class CheckPersonEligible implements
CheckPerson {
       public boolean test(Person p) {
           return p.gender == Person.Sex.MALE &&
               p.getAge() >= 18 &&
               p.getAge() <= 25;
       }
   }
```

- To use this class, you create a new instance of it and invoke the printPersons method:

```
printPersons(roster, new CheckPersonEligible());
```

Approach 4: Specify Search Criteria Code with a Lambda Expression

- The CheckPerson interface is a functional interface.
- A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.)
- Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it.
- To do this, you use a lambda expression, which is highlighted in the following method invocation:



```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Why Lambdas?

- A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times.
- **The Syntax of Lambda Expressions**
- A lambda expression consists of the following:
  - A comma-separated list of formal parameters enclosed in parentheses. The CheckPerson.test method contains one parameter, p, which represents an instance of the Person class.
  - The arrow token ->
  - A body, which consists of a single expression or a statement block.

```
p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

- If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

```
p -> {
    return p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25;
}
```

- A return statement is not an expression; in a lambda expression, you must enclose statements in braces ({}). However, you do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:

- email -> System.out.println(email)

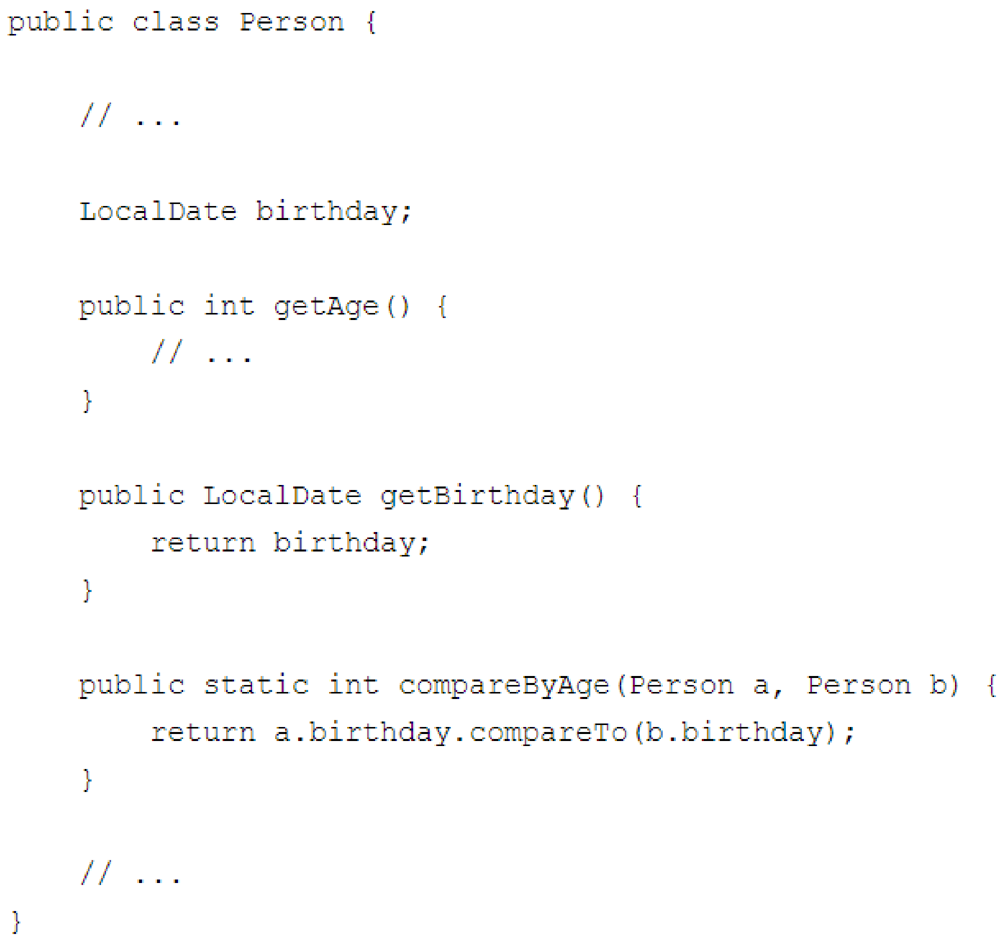
Lambdas usage

- You can supply a lambda expression whenever an object of an interface with a single abstract method is expected. Such an interface is called a *functional interface*.
- Example: public static <T> void sort(T[] a, Comparator<? super T> c)

```
Arrays.sort(myArray, (first, second) ->
first.length() - second.length());
```

Method References

- You use lambda expressions to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method.
- In those cases, it's often clearer to refer to the existing method by name.
- Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.



```
public class Person {

    // ...

    LocalDate birthday;

    public int getAge() {
        // ...
    }

    public LocalDate getBirthday() {
        return birthday;
    }

    public static int compareByAge(Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }

    // ...
}
```

Kinds of method references

Kind	Syntax	Examples
Reference to a static method	ContainingClass::staticMethodName	Person::compareByAge String::appendStrings
Reference to an instance method of a particular object	containingObject::instanceMethodName	myComparison::compareByName myApp::appendStrings2
Reference to an instance method of an arbitrary object of a particular type	ContainingType::methodName	String::compareToIgnoreCase String::concat
Reference to a constructor	ClassName::new	HashSet::new

- Suppose that the members of your social networking application are contained in an array, and you want to sort the array by age. you could use a lambda expression

```
Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday().compareTo(b.getBirthday());
    }
);
```

- However, this method to compare the birth dates of two Person instances already exists as Person.compareByAge. You can invoke this method instead in the body of the lambda expression:

```
Arrays.sort(rosterAsArray,
    (a, b) -> Person.compareByAge(a, b)
);
```

Because this lambda expression invokes an existing method, you can use a method reference instead of a lambda expression:

```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

Aggregate Operations

- For what do you use collections? You don't simply store objects in a collection and leave them there. In most cases, you use collections to retrieve items stored in them.
- The following example prints the name of all members contained in the collection roster with a for-each loop:

```
for (Person p : roster) {
    System.out.println(p.getName());
}
```

- The following example prints all members contained in the collection roster but with the aggregate operation forEach:

```
     roster
    .stream()
    .forEach(e -> System.out.println(e.getName());
```

- Although, in this example, the version that uses aggregate operations is longer than the one that uses a for-each loop, you will see that versions that use bulk-data operations will be more concise for more complex tasks.

Streams

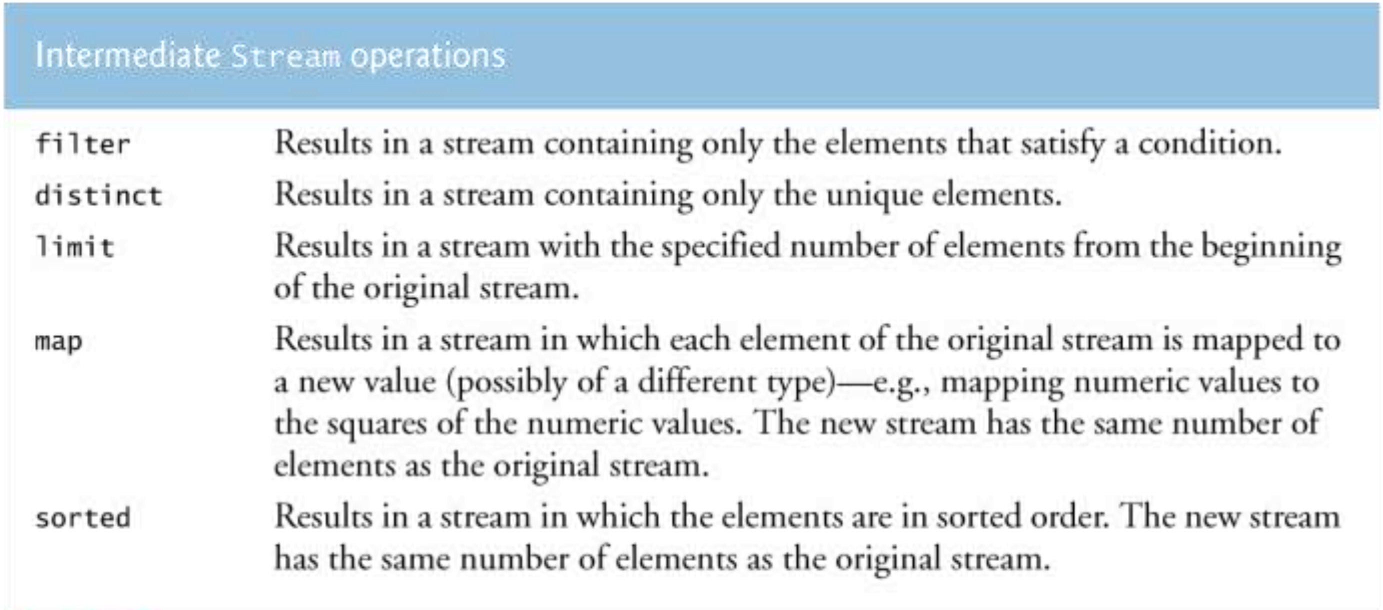
- Streams are objects that implement interface `Stream` (from the package `java.util.stream`
  - Enable you to perform functional programming tasks
- Specialized stream interfaces for processing `int`, `long` or `double` values
- Streams move elements through a sequence of processing steps—known as a stream pipeline
  - Pipeline begins with a data source, performs various intermediate operations on the data source’s elements and ends with a terminal operation.
- A stream pipeline is formed by chaining method calls.

Streams (Cont.)

- Streams do not have their own storage
  - Once a stream is processed, it cannot be reused, because it does not maintain a copy of the original data source.
- An intermediate operation specifies tasks to perform on the stream’s elements and always results in a new stream.
- Intermediate operations are lazy—they aren’t performed until a terminal operation is invoked.
  - Allows library developers to optimize stream-processing performance.

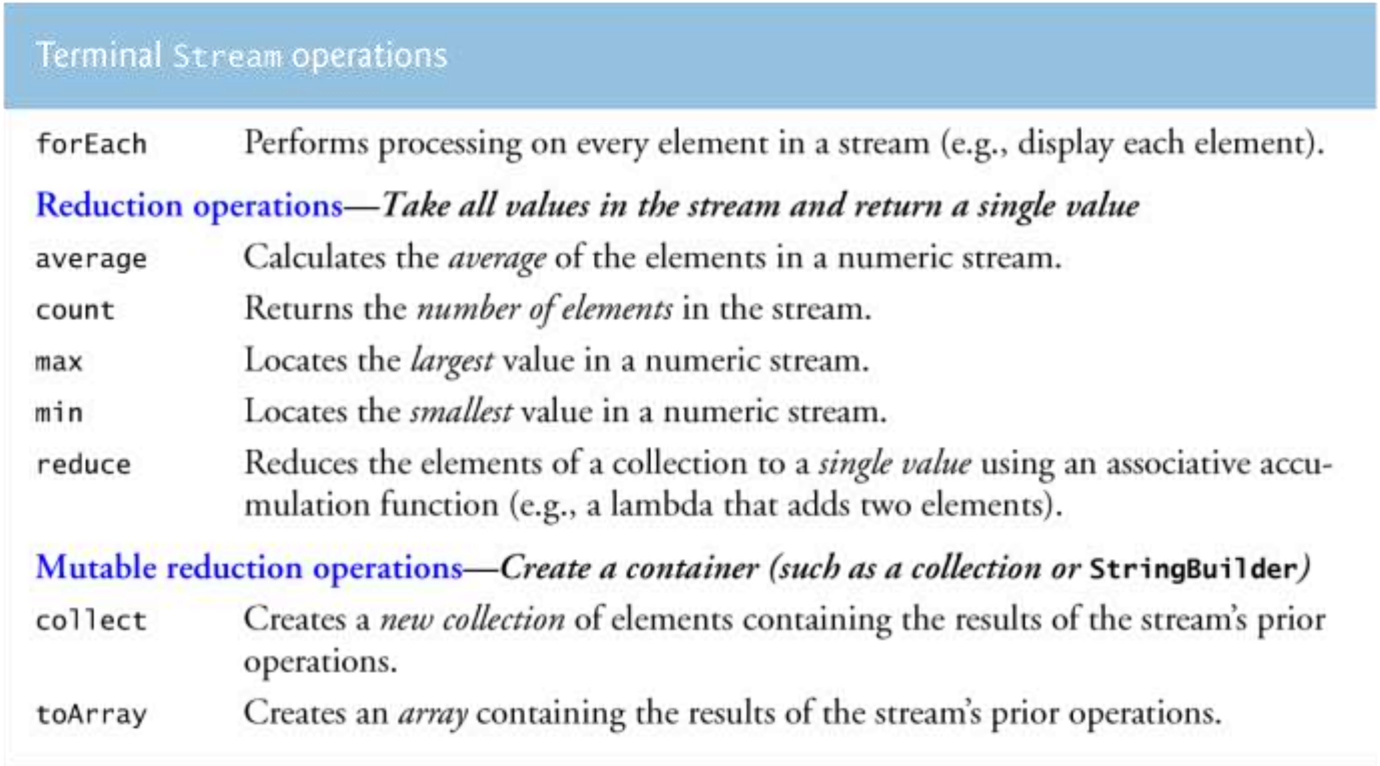
Streams (Cont.)

- Terminal operation
  - initiates processing of a stream pipeline's intermediate operations
  - produces a result
  - Terminal operations are eager—they perform the requested operation when they are called.
- Figure 17.3 shows some common intermediate operations.
- Figure 17.4 shows some common terminal operations.



Intermediate `Stream` operations	
`filter`	Results in a stream containing only the elements that satisfy a condition.
`distinct`	Results in a stream containing only the unique elements.
`limit`	Results in a stream with the specified number of elements from the beginning of the original stream.
`map`	Results in a stream in which each element of the original stream is mapped to a new value (possibly of a different type)—e.g., mapping numeric values to the squares of the numeric values. The new stream has the same number of elements as the original stream.
`sorted`	Results in a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream.

Fig. 17.3 | Common intermediate `Stream` operations.



Terminal `Stream` operations	
`forEach`	Performs processing on every element in a stream (e.g., display each element).
Reduction operations—*Take all values in the stream and return a single value*	
`average`	Calculates the *average* of the elements in a numeric stream.
`count`	Returns the *number of elements* in the stream.
`max`	Locates the *largest* value in a numeric stream.
`min`	Locates the *smallest* value in a numeric stream.
`reduce`	Reduces the elements of a collection to a *single value* using an associative accumulation function (e.g., a lambda that adds two elements).
Mutable reduction operations—*Create a container (such as a collection or* `StringBuilder`*)*	
`collect`	Creates a *new collection* of elements containing the results of the stream's prior operations.
`toArray`	Creates an *array* containing the results of the stream's prior operations.

Fig. 17.4 | Common terminal `Stream` operations.

Examples

- The following example prints the male members contained in the collection roster with a pipeline that consists of the aggregate operations filter and forEach:

```
roster
    .stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()))
```

Examples

- The following example calculates the average age of all male members contained in the collection roster with a pipeline that consists of the aggregate operations filter, mapToInt, and average:

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Examples

- Consider the following pipeline, which calculates the sum of the male members' ages in the collection roster. It uses the Stream.sum reduction operation:

```
Integer totalAge = roster
    .stream()
    .mapToInt(Person::getAge)
    .sum();
```