Lambda Expressions

Streams

Lambda Expressions – Definition and Syntax

- **Definition:** A lambda expression is a block of code with parameters which can be executed once/multiple times at a later point in time [1]
  - Has no name, no return type (inferred from the context of its use/body), no throws clause (inferred from the context of its use/body), no generics
- **Syntax: (<LambdaParametersList>) -> { <LambdaBody> }** [2]
  - **(<LambdaParametersList>)** - a comma-separated list of formal parameters enclosed in parentheses
  - **->** - the arrow token
  - **{ <LambdaBody> }** - consists of a single expression or a statement block
    - May declare local variables
    - May use statements including break, continue and return
    - May throw exceptions

Lambda Expressions – Why use them?

```
public class Person {
    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    Person(String nameArg, LocalDate birthdayArg, Sex genderArg, String emailArg) {
        name = nameArg;
        birthday = birthdayArg;
        gender = genderArg;
        emailAddress = emailArg;
    }

    public int getAge() {

    public Sex getGender() {

    public String getName() {

    public String getEmailAddress() {

    public LocalDate getBirthday() {

    public void printPerson() {
        System.out.println(name + ", " + this.getAge());
    }

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

```
public static List<Person> createRoster() {

    List<Person> roster = new ArrayList<>();
    roster.add(
        new Person(
        "Fred",
        IsoChronology.INSTANCE.date(1980, 6, 20),
        Person.Sex.MALE,
        "fred@example.com"));
    roster.add(
        new Person(
        "Jane",
        IsoChronology.INSTANCE.date(1990, 7, 15),
        Person.Sex.FEMALE, "jane@example.com"));
    roster.add(
        new Person(
        "George",
        IsoChronology.INSTANCE.date(1991, 8, 13),
        Person.Sex.MALE, "george@example.com"));
    roster.add(
        new Person(
        "Bob",
        IsoChronology.INSTANCE.date(2000, 9, 12),
        Person.Sex.MALE, "bob@example.com"));

    return roster;
}
```

(Examples taken from [2])

Lambda Expressions – Why use them?

```java
// Approach 1: Create Methods that Search for Persons that Match One
// Characteristic

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

```java
// Approach 1: Create Methods that Search for Persons that Match One
// Characteristic

System.out.println("Persons older than 20:");
printPersonsOlderThan(roster, 20);
```

```java
// Approach 2: Create More Generalized Search Methods

public static void printPersonsWithinAgeRange(
    List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

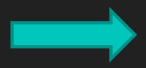
```java
// Approach 2: Create More Generalized Search Methods

System.out.println("Persons between the ages of 14 and 30:");
printPersonsWithinAgeRange(roster, 14, 30);
```

(Examples taken from [2])

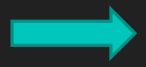
Lambda Expressions – Why use them?

```
// Approach 3: Specify Search Criteria Code in a Local Class
// Approach 4: Specify Search Criteria Code in an Anonymous Class
// Approach 5: Specify Search Criteria Code with a Lambda Expression

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

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



```
// Approach 3: Specify Search Criteria Code in a Local Class

System.out.println("Persons who are eligible for Selective Service:");

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

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

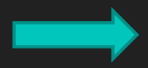


```
// Approach 4: Specify Search Criteria Code in an Anonymous Class

System.out.println("Persons who are eligible for Selective Service " +
    "(anonymous class):");

printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

(Examples taken from [2])

Lambda Expressions – Why use them?

```
// Approach 3: Specify Search Criteria Code in a Local Class
// Approach 4: Specify Search Criteria Code in an Anonymous Class
// Approach 5: Specify Search Criteria Code with a Lambda Expression

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



```
// Approach 5: Specify Search Criteria Code with a Lambda Expression

System.out.println("Persons who are eligible for Selective Service " +
    "(lambda expression):");

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

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

(Examples taken from [2])

Lambda Expressions – Why use them?

```
// Approach 6: Use Standard Functional Interfaces with Lambda Expressions

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

```
// Approach 6: Use Standard Functional Interfaces with Lambda
// Expressions

System.out.println("Persons who are eligible for Selective Service " +
    "(with Predicate parameter):");

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

```
// Approach 7: Use Lambda Expressions Throughout Your Application

public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            block.accept(p);
        }
    }
}
```

```
// Approach 7: Use Lamba Expressions Throughout Your Application

System.out.println("Persons who are eligible for Selective Service " +
    "(with Predicate and Consumer parameters):");

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

(Examples taken from [2])

Lambda Expressions – Why use them?

```
// Approach 7, second example

public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

```
// Approach 7, second example

System.out.println("Persons who are eligible for Selective Service " +
    "(with Predicate, Function, and Consumer parameters):");

processPersonsWithFunction(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

```
// Approach 8: Use Generics More Extensively

public static <X, Y> void processElements(
    Iterable<X> source,
    Predicate<X> tester,
    Function<X, Y> mapper,
    Consumer<Y> block) {
        for (X p : source) {
            if (tester.test(p)) {
                Y data = mapper.apply(p);
                block.accept(data);
            }
        }
}
```

```
// Approach 8: Use Generics More Extensively

System.out.println("Persons who are eligible for Selective Service " +
    "(generic version):");

processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

(Examples taken from [2])

Lambda Expressions – Type

- The type of a Lambda Expression is **Functional Interface**
  - **Functional interface = an interface that has exactly one abstract method**
  - The exact type of a lambda expression is determined by the Java compiler based on the target type of the context or situation in which the lambda expression was found [2]

- T t = <LambdaExpression>; [3]
  - The target type of the Lambda expression is T
  - T must be a Functional Interface type
  - The Lambda expression has the same number and type of parameters as the abstract method of T
  - The type of the returned value from the body of the Lambda expression should be assignment compatible to the return type of the abstract method of T

```
@FunctionalInterface
  public interface Adder {
       double add(double n1, double n2);
}

Adder adder = (x, y) -> x + y;
double sum = adder.add(10.34, 89.11);
```

(Example taken from [3])

Lambda Expressions – Functional Interfaces (1)

- **Functional Interface - an interface that has exactly one abstract method**
  - Default/static methods and methods inherited from the Object class do not count for defining a Functional Interface
  - A Functional Interface represents one type of functionality/operation in terms of its single abstract method => the target type of a lambda expression is always a Functional Interface
  - A lambda expression can be supplied whenever an object of an interface with a single abstract method is expected [1]
- Annotation of functional interfaces: **@FunctionalInterface**
- **Example [1]**

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

```
// Consider the Arrays.sort method:
// public static <T> void sort(T[] a, int fIndex, int toIndex, Comparator<? super T> c)
Arrays.sort(words, (first, second) -> Integer.compare(first.length(),second.length()));
```

Lambda Expressions – Functional Interfaces (2)

- The java.util.function package defines a number of general purpose functional interfaces [3]

Interface Name	Method	Description
`Function<T,R>`	`R apply(T t)`	Represents a function that takes an argument of type T and returns a result of type R.
`BiFunction<T,U,R>`	`R apply(T t, U u)`	Represents a function that takes two arguments of types T and U, and returns a result of type R.
`Predicate<T>`	`boolean test(T t)`	In mathematics, a predicate is a boolean-valued function that takes an argument and returns true or false. The function represents a condition that returns true or false for the specified argument.
`BiPredicate<T,U>`	`boolean test(T t, U u)`	Represents a predicate with two arguments.
`Consumer<T>`	`void accept(T t)`	Represents an operation that takes an argument, operates on it to produce some side effects, and returns no result.
`BiConsumer<T,U>`	`void accept(T t, U u)`	Represents an operation that takes two arguments, operates on them to produce some side effects, and returns no result.
`Supplier<T>`	`T get()`	Represents a supplier that returns a value.
`UnaryOperator<T>`	`T apply(T t)`	Inherits from `Function<T,T>`. Represents a function that takes an argument and returns a result of the same type.
`BinaryOperator<T>`	`T apply(T t1, T t2)`	Inherits from `BiFunction<T,T,T>`. Represents a function that takes two arguments of the same type and returns a result of the same.

Example [3] using Function

```
Function<Long, Long> square = x -> x * x;
Function<Long, Long> addOne = x -> x + 1;

Function<Long, Long> squareAddOne =
                    square.andThen(addOne);

System.out.println(squareAddOne.apply(5L));
```

Example [3] using Predicate

```
Predicate<Integer> greaterThanTen = x -> x > 10;
Predicate<Integer> lessThanOrEqualToTen =
                              greaterThanTen.negate();

System.out.println(greaterThanTen.test(10));
System.out.println(lessThanOrEqualToTen.test(10));
```

Lambda Expressions – Method References (1)

- Method references are compact, easy-to-read lambda expressions for methods that already have a name [4]
- **Syntax [3]: <Qualifier>::<MethodName>**
  - <Qualifier> depends on the type of the method reference
  - <MethodName> is the name of the method
- Example [3]

```
import java.util.function.ToIntFunction;
...
ToIntFunction<String> lengthFunction =
                                        str -> str.length();
String name = “Popescu";
int len = lengthFunction.applyAsInt(name);
System.out.println("Name = " + name + ",
                                          length = " + len);
```

```
import java.util.function.ToIntFunction;
...
ToIntFunction<String> lengthFunction = String::length;
String name = “Popescu";
int len = lengthFunction.applyAsInt(name);
System.out.println("Name = " + name + ",
                                          length = " + len);
```

Lambda Expressions – Method References (2)

- Types of Method References [3]

Syntax	Description
`TypeName::staticMethod`	A method reference to a static method of a class, an interface, or an enum
`objectRef::instanceMethod`	A method reference to an instance method of the specified object
`ClassName::instanceMethod`	A method reference to an instance method of an arbitrary object of the specified class
`TypeName.super::instanceMethod`	A method reference to an instance method of the supertype of a particular object
`ClassName::new`	A constructor reference to the constructor of the specified class
`ArrayTypeName::new`	An array constructor reference to the constructor of the specified array type

Lambda Expressions – Method References (3)

- Static Method References (TypeName::staticMethod) - Example [3]

```
// Consider the class Integer with the method static String toBinaryString(int i)

//Using a lambda expression with toBinaryString
Function<Integer, String> func1 = x -> Integer.toBinaryString(x);
System.out.println(func1.apply(17));

//Using a static Method Reference with toBinaryString
Function<Integer, String> func2 = Integer::toBinaryString;
System.out.println(func2.apply(17));
```

Lambda Expressions – Method References (4)

- Instance Method References – Examples [4]

```
// Example with bound receiver – objectRef::instanceMethod
class ComparisonProvider {
    public int compareByName(Person a, Person b) {
        return a.getName().compareTo(b.getName());
    }
}
ComparisonProvider myComparisonProvider = new ComparisonProvider();
Arrays.sort(rosterAsArray, myComparisonProvider::compareByName);
```

```
// Example with unbound receiver – ClassName::instanceMethod
String[] stringArray = { "Barbara", "James", "Mary", "John",
    "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase)
```

Lambda Expressions – Method References (5)

- Supertype Method References (TypeName.super::instanceMethod) – Example [3]

```
public interface Priced{
        default double getPrice() { return 1.0;}
}

public class Item implements Priced{
…
@Override
public double getPrice() { return price; }

@Override
public String toString() {
  return "name = "+getName()+", price ="+getPrice();}
}
```

```
//Method test in Class Item:
public void test() {
  // Uses the Item.toString() method
  Supplier<String> s1 = this::toString;
  // Uses Object.toString() method
  Supplier<String> s2 = Item.super::toString;
  // Uses Priced.getPrice() method
  Supplier<Double> s3 = Priced.super::getPrice;
  System.out.println(s1.get());
  System.out.println(s2.get());
  System.out.println(s3.get());
}
```

Lambda Expressions – Method References (6)

- Constructor References (ClassName::new or ArrayTypeName::new) – Example [3]

```
Supplier<Item> func1 = () -> new Item();
Function<String,Item> func2 = name -> new Item(name);
BiFunction<String,Double, Item> func3 =
            (name, price) -> new Item(name, price);
System.out.println(func1.get());
System.out.println(func2.apply("Apple"));
System.out.println(func3.apply("Apple", 0.75));
```

Output

```
//Constructor Item() called.
name = Unknown, price = 0.0
//Constructor Item(String) called.
name = Apple, price = 0.0
//Constructor Item(String, double) called.
name = Apple, price = 0.75
```

Lambda Expressions – Method References (7)

- Generic method references - Example

Example

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
...
Function<String[], List<String>>asList = Arrays::<String>asList;
List<String> stringsList = asList.apply(new String[] {"a", "b", "c"});
for(String s : stringsList) { System.out.println(s); }
```

Lambda Expressions – Method References (8)

- Comparing Objects
  - Methods of the Comparator interface
    - static <T,U extends Comparable<? super U>> Comparator<T> **comparing** (Function<? super T,? extends U> keyExtractor)
    - default <U extends Comparable<? Super U>>Comparator<T> **thenComparing** (Function<? super T,? extends U> keyExtractor)

```
// Example - creates a Comparator<Person> that sorts Person objects based on their last names and first names
Comparator<Person> lastFirstComp = Comparator.comparing(Person::getLastName)
                                                .thenComparing(Person::getFirstName);
```

Streams - Definition

- **Stream** = a sequence of elements from a source that supports aggregate operations [5]
  - ***Sequence of elements***: A stream provides an interface to a sequenced set of values of a specific element type
    - streams don't actually store elements; they are computed on demand
  - ***Source:*** Streams consume from a data-providing source such as collections, arrays, or I/O resources
  - ***Aggregate operations***: Streams support SQL-like operations and common operations from functional programing languages (e.g. filter, map, reduce, find, match, sorted, etc.)
- **Features of stream operations [5]**
  - ***Pipelining***: Many stream operations return a stream themselves => allows operations to be chained to form a larger pipeline
  - ***Internal iteration***: In contrast to collections, which are iterated explicitly (external iteration), stream operations do the iteration behind the scenes for you

Streams – Why to use them?

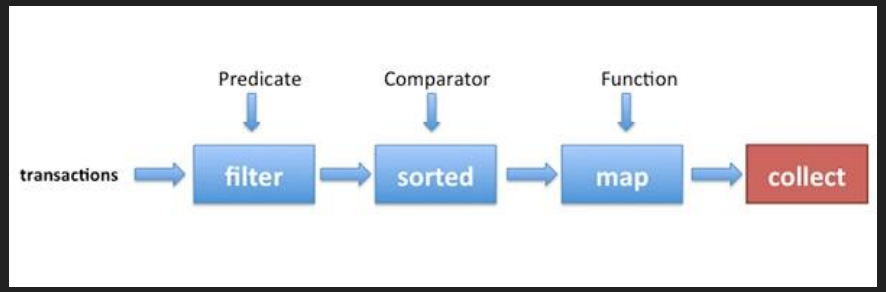
- Example [5]: Find all transactions of type grocery and return a list of transaction IDs sorted in decreasing order of transaction value

Java SE 7 implementation

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions){
  if(t.getType() == Transaction.GROCERY){
    groceryTransactions.add(t);
  }
}
Collections.sort(groceryTransactions, new Comparator(){
  public int compare(Transaction t1, Transaction t2){
    return t2.getValue().compareTo(t1.getValue());
  }});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
                transactionsIds.add(t.getId());}
```

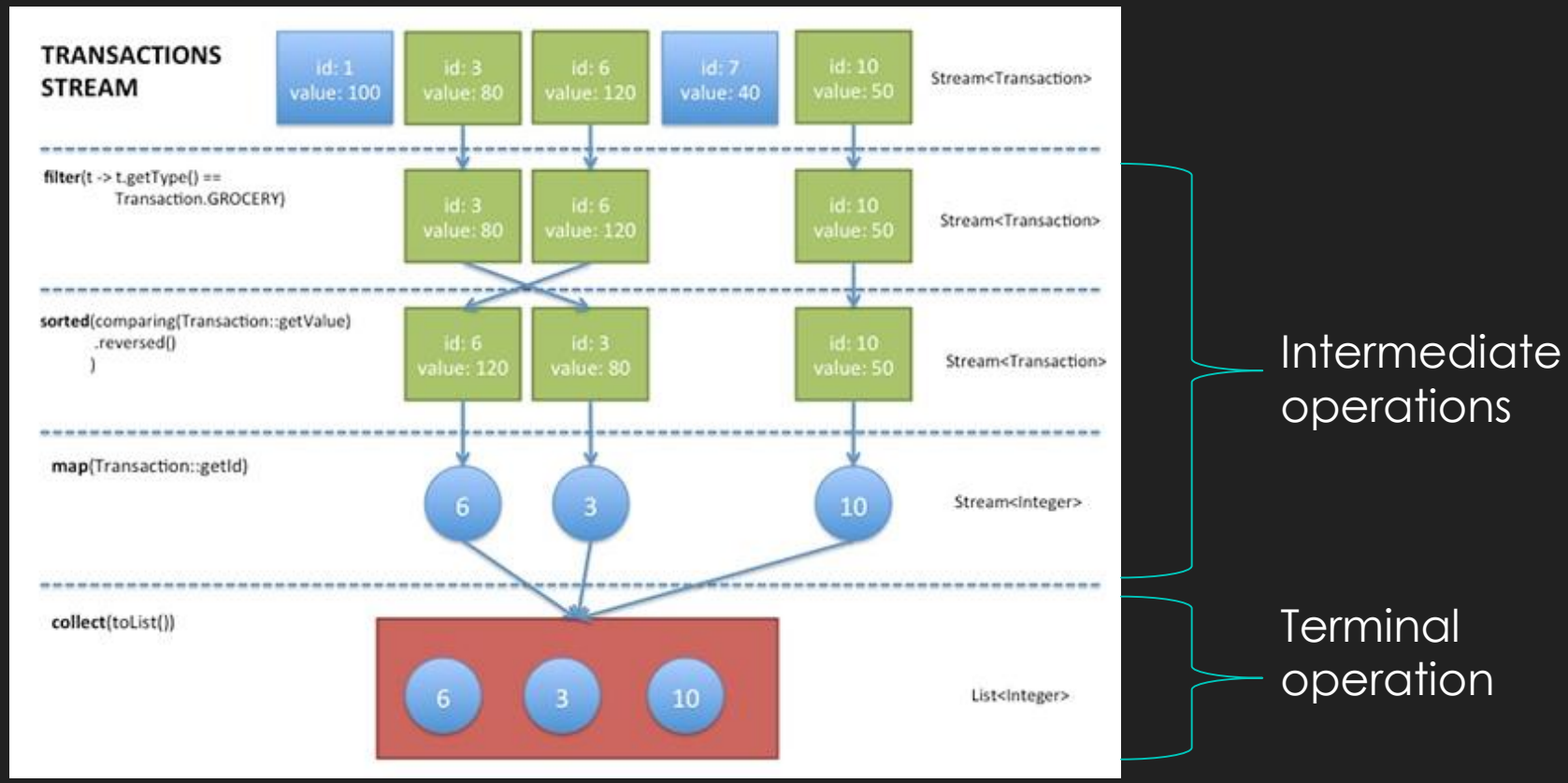
Java SE 8 implementation

```
List<Integer> transactionsIds =
    transactions.stream() //or transactions.parallelStream()
      .filter(t -> t.getType() == Transaction.GROCERY)
      .sorted(comparing(Transaction::getValue).reversed())
      .map(Transaction::getId)
      .collect(toList());
```



Streams – Why to use them?

- Example [5]: Find all transactions of type grocery and return a list of transaction IDs sorted in decreasing order of transaction value



Streams – Creation Examples (1)

```
// create sequential stream from values
int sum = Stream.of(1, 2, 3, 4, 5)
   .filter(n -> n % 2 == 1)
   .reduce(0, Integer::sum);

//create an empty stream
Stream<String> stream = Stream.empty();

//create a sequential stream from collections
Stream<String> sequentialStream = names.stream();

// create a stream from functions
Stream.generate(Math::random)
  .limit(5)
  .forEach(System.out::println);
```

Streams – Creation Examples (1)

```
// create a stream from files [9]
Stream<String> stream = Files.lines(Paths.get("test.txt")){
                        stream
                            .filter(line -> line.contains(“pattern"))
                            .map(String::trim)
                            .forEach(System.out::println);
                    }
```

Streams – Operations

Operation	Type	Description
Distinct	Intermediate	Returns a stream consisting of the distinct elements of this stream. Elements `e1` and `e2` are considered equal if `e1.equals(e2)` returns true.
filter	Intermediate	Returns a stream consisting of the elements of this stream that match the specified predicate.
flatMap	Intermediate	Returns a stream consisting of the results of applying the specified function to the elements of this stream. The function produces a stream for each input element and the output streams are flattened. Performs one-to-many mapping.
limit	Intermediate	Returns a stream consisting of the elements of this stream, truncated to be no longer than the specified size.
map	Intermediate	Returns a stream consisting of the results of applying the specified function to the elements of this stream. Performs one-to-one mapping.
peek	Intermediate	Returns a stream whose elements consist of this stream. It applies the specified action as it consumes elements of this stream. It is mainly used for debugging purposes.
skip	Intermediate	Discards the first `n` elements of the stream and returns the remaining stream. If this stream contains fewer than `n` elements, an empty stream is returned.
sorted	Intermediate	Returns a stream consisting of the elements of this stream, sorted according to natural order or the specified `Comparator`. For an ordered stream, the sort is stable.
allMatch	Terminal	Returns true if all elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
anyMatch	Terminal	Returns true if any element in the stream matches the specified predicate, false otherwise. Returns false if the stream is empty.
findAny	Terminal	Returns any element from the stream. An empty `Optional` object is for an empty stream.
findFirst	Terminal	Returns the first element of the stream. For an ordered stream, it returns the first element in the encounter order; for an unordered stream, it returns any element.
noneMatch	Terminal	Returns true if no elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
forEach	Terminal	Applies an action for each element in the stream.
reduce	Terminal	Applies a reduction operation to computes a single value from the stream.

(From [3])

Streams – Other Examples (1)

- **Examples with the *collect* operation (using the Collectors class) [8]**

```
// Accumulate names into a List
    List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());

// Accumulate names into a TreeSet
    Set<String> set =
people.stream().map(Person::getName).collect(Collectors.toCollection(TreeSet::new));

 // Convert elements to strings and concatenate them, separated by commas
    String joined = things.stream()
                          .map(Object::toString)
                          .collect(Collectors.joining(", "));
```

Streams – Other Examples (2)

- **Examples with the *collect* operation (using the Collectors class) [8]**

```
// Group employees by department
    Map<Department, List<Employee>> byDept = employees.stream()
                                                .collect(Collectors.groupingBy(Employee::getDepartment));

// Compute sum of salaries by department
    Map<Department, Integer> totalByDept = employees.stream()
                                                .collect(Collectors.groupingBy(Employee::getDepartment,
                                                        Collectors.summingInt(Employee::getSalary)));

// Partition students into passing and failing
    Map<Boolean, List<Student>> passingFailing = students.stream()
                                .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

Streams – Other Examples (3)

- **Examples with the *collect* operation (using the Collectors class) [8]**

```
// Given a stream of Person, calculate tallest person in each city
Comparator<Person> byHeight = Comparator.comparing(Person::getHeight);
Map<City, Person> tallestByCity
        = people.stream().collect(groupingBy(Person::getCity, reducing(BinaryOperator.maxBy(byHeight))));
```

Streams – Other Examples (4)

- **Examples with the *reduce* operation [9]**

```
// Given a stream of Person, determine the oldest person
List<Person> persons = Arrays.asList(new Person("Max", 18),
                                     new Person("Peter", 23),
                                     new Person("Pamela", 23),
                                     new Person("David", 12));
persons.stream().reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
                .ifPresent(System.out::println);

// Given a stream of Person, build a new Person object with the aggregated names and ages from all other persons in the stream
Person result = persons.stream().reduce(new Person("", 0), (p1, p2) -> { p1.age += p2.age;
                                                                         p1.name += p2.name;
                                                                         return p1;});
```

Streams – Other Examples (5)

- **Example with the *map* operation [10]**

```
public class Staff {
    private String name;
    private int age;
    private BigDecimal salary;
    //…
}

public class StaffPublic {
    private String name;
    private int age;
    private String extra;
    //...
}
```

```
List<Staff> staff = Arrays.asList(
                new Staff("mkyong", 30, new BigDecimal(10000)),
                new Staff("jack", 27, new BigDecimal(20000)),
                new Staff("lawrence", 33, new BigDecimal(30000)));
// Convert a list of Staff objects into a list of StaffPublic objects
 List<StaffPublic> result = staff.stream().map(temp -> {
            StaffPublic obj = new StaffPublic();
            obj.setName(temp.getName());
            obj.setAge(temp.getAge());
            if ("mkyong".equals(temp.getName())) { obj.setExtra("extra");}
            return obj;
        }).collect(Collectors.toList());
```

Bibliography

Note: This presentation synthesizes the information presented in the following sources

[1] Cay S. Horstmann, Java SE 8 for the Really Impatient, Addison-Wesley, ISBN-13: 978-0-321-92776-7, 2014.

[2] https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[3] Kishori Sharan, Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams, Apress, ISBN 1430266597, 2014.

[4] https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

[5] http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html.

[6] http://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html

[7] http://winterbe.com/posts/2015/03/25/java8-examples-string-number-math-files/

[8] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

[9] http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

[10] https://www.mkyong.com/java8/java-8-streams-map-examples/