JAVA 8 New Features

BY:

SUJATA BATRA

Introduction

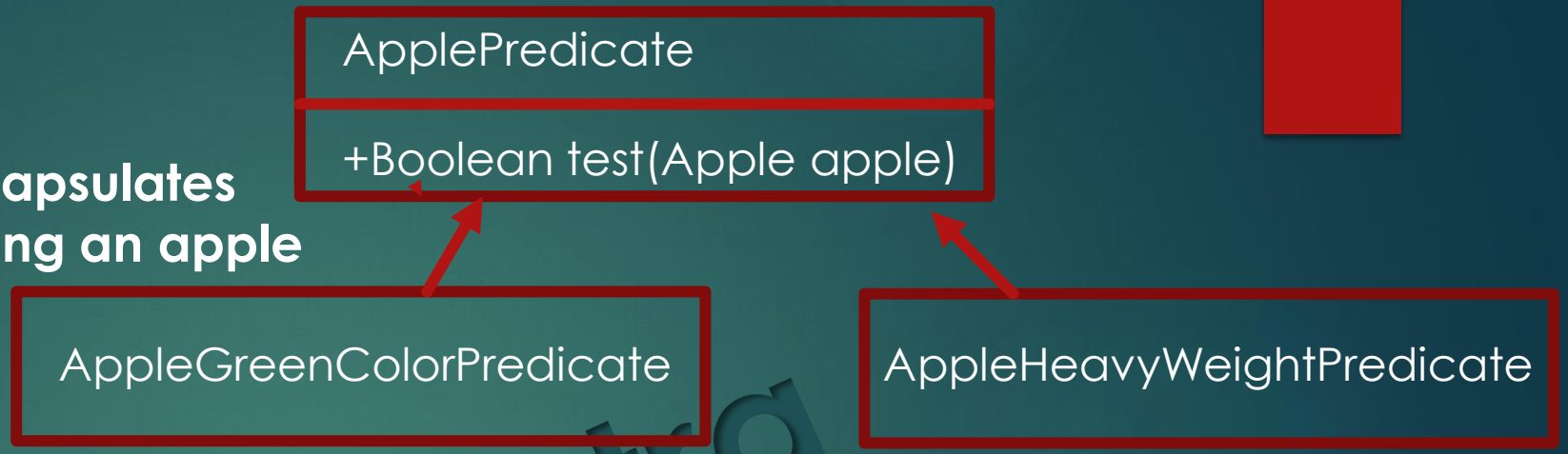
- JAVA 8 is a major feature release of JAVA programming language development.
- Its initial version was released on 18 March 2014.
- With the Java 8 release, Java provided supports for
  - functional programming,
  - new JavaScript engine,
  - new APIs for date time manipulation,
  - new streaming API, etc.

Behaviour Parameterization

- Behavior parameterization is the ability for a method to take multiple different behaviors as parameters and use them internally to accomplish different behaviors.
- Behavior parameterization lets us make your code more adaptive to changing requirements and saves on engineering efforts in the future.

Example

ApplePredicate encapsulates a strategy for selecting an apple




```
//Behaviour Parameterization
public static List<Apple> filterApple(List<Apple> inventory,ApplePredicate p){
        List<Apple> result =new ArrayList<>();
        for(Apple apple:inventory){
                if(p.test(apple)){
                    result.add(apple);
                }
         }
          return result;
}
```

Note: The only code really matters is implementation of the test method. Unfortunately,because filterApple method can only take objects,we have to wrap code inside an ApplePredicate object.

Anonymous classes

- Declare and instantiate a class at the same time.
- Don't have name.
- Used to reduce verbosity and time.

Note: Passing code is a way to give new behaviors as arguments to a method. But it's verbose prior to Java 8.
Anonymous classes helped a bit before Java 8 to get rid of the verbosity associated with declaring multiple concrete classes for an interface that are needed only once.

- The Java API contains many methods that can be parameterized with different behaviors, which include sorting, threads etc.

Lambdas

Sujata Batra

Lambda Introduction

- Representation of an 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.
- Can be passed around as a parameter thus achieving **behavior parameterization.**
- Let you pass code in a concise way.

- An instance of a lambda can be assigned to any **functional interface** whose single abstract method's definition matches the definition of the lambda.
- Lamda can be
  - assigned to variables
  - passed to functions

What are Lambda's good for

- Forms the basis of functional programming
- Make parallel programming easier
- Write more compact code
- Richer data structure collection
- Develop cleaner APIs

Syntax

- Lambda expression is composed of parameters,an arrow and a body.
- parameter -> expression body
  Or
  (parameters) -> { statements; }
  or
  () -> expression

- Following are Some examples of Lambda


```
(int a, int b) ->    a * b                    // takes two integers and returns their multiplication
(a, b)          ->   a - b                    // takes two numbers and returns their difference
() -> 99                                      // takes no values and returns 99
(String a) -> System.out.println(a)           // takes a string, prints its value to the console, and returns nothing
a -> 2 * a                                    // takes a number and returns the result of doubling it
c -> { //some complex statements }            // takes a collection and do some procesing
```

Rules for writing lambda expressions

- A lambda expression can have zero, one or more parameters.
- The type of the parameters can be explicitly declared or it can be inferred from the context.
- Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.
- When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a*a.
- The body of the lambda expressions can contain zero, one or more statements.
- If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

Functional Interface

Functional Interfaces in Java 8

- **A functional interface**, is an interface which has only a single abstract method.

- **@FunctionalInterface annotation**
  - used to explicitely specify that a given interface is to be treated as a functional interface.
  - is an *informative annotation.*

Custom Or User defined Functional Interfaces

- Interfaces defined by the user and have a single abstract method. These may/may not be annotated by @FunctionalInterface.

```
package com.sujata.java8training;

@FunctionalInterface
public interface MyCustomFunctionalInterface {

    //This is the only abstract method.Hence, this
    //interface qualifies as a Functional Interface
    public void firstMethod();

}
```

Pre-existing functional interfaces in Java prior to Java 8

- interface java.lang.Runnable{
  void run();
  }
- interface java.util.Comparator<T>{
  int compare(T o1,T o2)
  }

Newly defined functional interfaces in Java 8

- These are pre-defined Functional Interfaces introduced in Java 8 in java.util.function package.
- They are defined with generic types and are re-usable for specific use cases.
- One such Functional Interface is the Predicate<T> interface which is defined as follows –


```
//java.util.function.Predicate<T>
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Java 8 java.util.function package

- Provides a set of re-usable common functional interfaces(and their corresponding lambda) definitions which can be used by the programmer in his code instead of creating brand new functional interfaces.
- 4 major Categories
  - Consumer<T>
    - Used in all contexts where an object T needs to be consumed,i.e. taken as input, and some operation is to be performed on the object without returning any result.
  - Function<T,R>
    - Used when an object of a type T is taken as input and it is converted(or mapped) to another type R.
  - Predicate<T>
    - Used wherever an object T needs to be evaluated and a boolean value needs to be returned
  - Supplier<T>
    - Used in all contexts where there is no input but an output T is expected.

Common Functional Interfaces in Java8

Functional Interface	Functional Descriptor	Primitives Specializations
Predicate<T>	T -> Boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator

Common Functional Interfaces in Java8

Functional Interface	Functional Descriptor	Primitives Specializations
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<L, R>	(L, R) ->boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

Type Checking,type inference

- Type Checking:
  - Type of lambda is deduced from the context in which lambda is used.

Note: The type expected for the lambda expression inside the context is called the **target type.**

- Type Inference:
  - Java compiler also deduce an appropriate signature from the lambda because the function descriptor is available through the target type.

Using Local variables in Lambda

- lambda expressions are also allowed to use free variables (variables that aren't the parameters and defined in an outer scope), such lambdas are called **capturing lambdas**.

- Lambdas are allowed to capture (that is, to reference in their bodies) instance variables and static variables **without restrictions**.

- local variables have to be explicitly declared **final** or are **effectively final**.

Method References

- Used to refer method of functional interface.
- Compact and easy form of lambda expression.
- when using lambda expression to just referring a method, replace lambda expression with method reference.xxxxxxx

Example

```
//Lambda Expession
Function<String,Integer> intParser = (String str,Integer integer)->Integer.parseInt(str)

//Method Reference
Function<String,integer> intParser=Integer::parseInt
```

Method References

Types of Method Reference

- Reference to a static method.
- Reference to an instance method.
- Reference to a constructor.

Type 1: Reference to a static method

- **Lambda Syntax:**
  - (arguments) -> <ClassName>.<staticMethodName>(arguments);
- **Equivalent Method Reference:**
  - <ClassName> :: <staticMethodName>

Example

```
//Lambda Expession
Function<String, Double> doubleConvertorLambda=(String s) ->Double.parseDouble(s);
//Equivalent Method Reference
Function<String, Double> doubleConvertor=Double::parseDouble;
```

Type 2: Reference to an instance method of an arbitrary type

- Lambda Syntax
  - (arg0,rest)->arg0.instanceMethod(rest)

  Note: arg0 is of type ClassName
- **Equivalent Method Reference:**
  - ClassName of arg0 ::instanceMethod

Type 2: Example

```
List <String> str=Arrays.asList("a","b","A","B");
//Lambda
str.sort((s1,s2)->s1.compareToIgnoreCase(s2));

//Equivalent Method Reference
str.sort(String::compareToIgnoreCase);
```

Type 2: Reference to an instance method

- **Lambda Syntax:**
  - (arguments) -> <expression>.<instanceMethodName>(arguments)
- **Equivalent Method Reference:**
  - <expression> :: <instanceMethodName>

Type 2 :Example

```
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
    public static void main(String[] args) {
        InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating object
       //lambda
        Sayable sayablex=()->{methodReference.saySomething()};
        // Referring non-static method using reference
            Sayable sayable = methodReference::saySomething;
        // Calling interface method
            sayable.say();
            // Referring non-static method using anonymous object
            Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use anonymous object also
            // Calling interface method
            sayable2.say();
    }
}
```

Type 3:Reference to a Constructor

- **Syntax of Constructor References:**
  - <ClassName>::new

- Type 3: Example

```
public class Employee{
 String name;
 Integer age;
 //Contructor of employee
 public Employee(String name, Integer age){
  this.name=name;
  this.age=age;
 }
}
```

Type 3 : Example

```
public interface EmployeeFactory{
    public Employee getEmployee(String name,Integer age);
}
```

//Client Code for invoking Factory Interface

//lambda Example

```
EmployeeFactory empFactory=(name,age)-> new Employee(name,age);
```

//Equivalent Method Reference

```
EmployeeFactory empFactory=Employee::new;

Employee emp= empFactory.getEmployee("John Hammond", 25);
```

Streams

Stream API in JAVA 8

- The Stream API in Java 8 lets you write code that's
  - **Declarative** : More concise and readable
  - **Composable** : Greater Flexibility
  - **Parallelizable** : Better performance

What is Stream

A **sequence of elements** from a **source** that supports **data processing operations**.

- ***Sequence of elements—*** Like a collection, a stream provides an interface to a sequenced set of values of a specific element type.
- ***Source—*** Streams consume from a data-providing source such as collections, arrays, or I/O resources.
- ***Data processing operations—*** Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on. Stream operations can be executed either sequentially or in parallel.

Stream operations have two important characteristics:

- ***Pipelining—*** Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline. A pipeline of operations can be viewed as a database-like query on the data source.

- ***Internal iteration—*** In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration behind the scenes for you.

Example

```
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;
    //argumented constructor
    //setters & getters

    public enum Type { MEAT, FISH, OTHER }
}
```

In Java 7

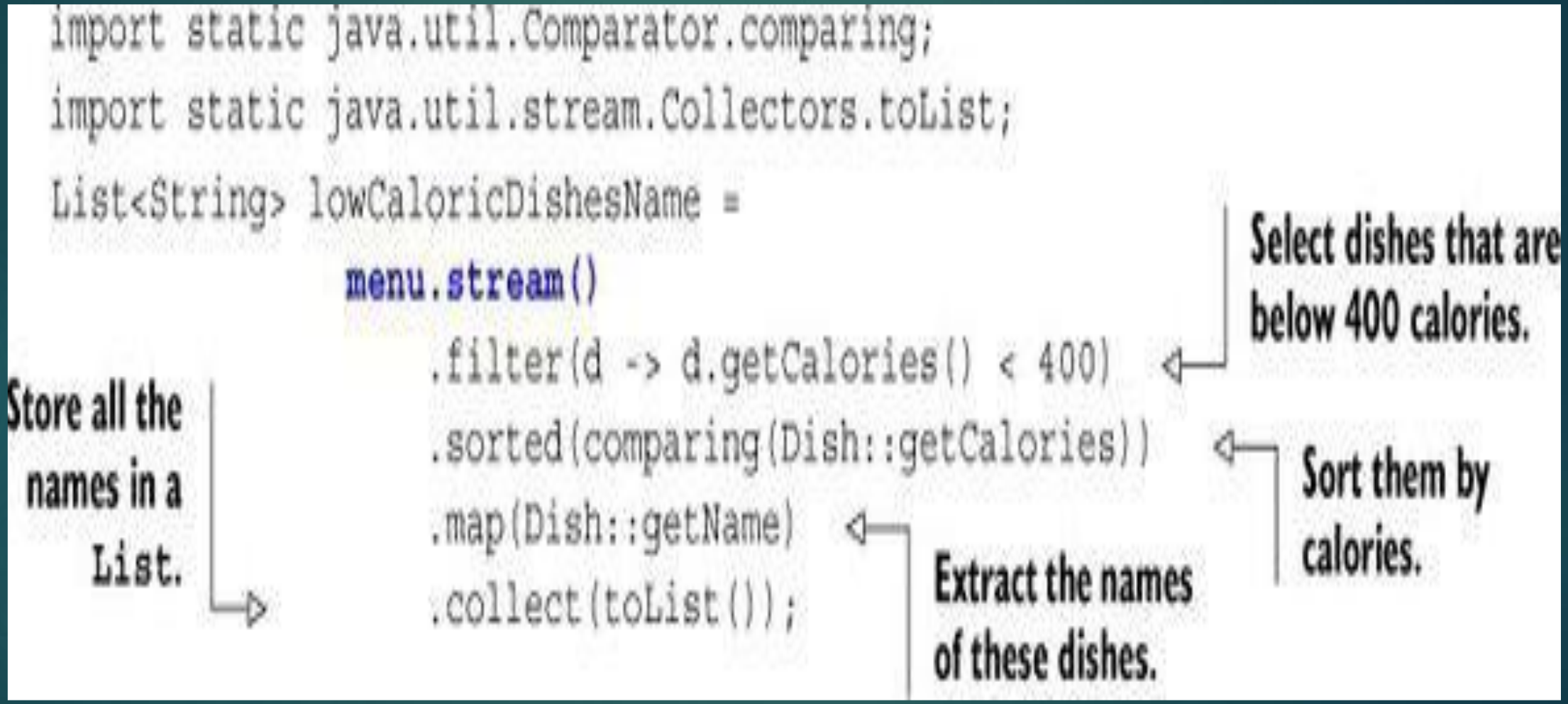
```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: menu){
    if(d.getCalories() < 400){
        lowCaloricDishes.add(d);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(), d2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d: lowCaloricDishes){
    lowCaloricDishesName.add(d.getName());
}
```

Filter the elements using an accumulator.

Sort the dishes with an anonymous class.

Process the sorted list to select the names of dishes.

In Java 8


import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
List<String> lowCaloricDishesName =
menu.stream()
.filter(d -> d.getCalories() < 400)
.sorted(comparing(Dish::getCalories))
.map(Dish::getName)
.collect(toList());
Select dishes that are below 400 calories.
Sort them by calories.
Extract the names of these dishes.
Store all the names in a List.

```
import static java.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames =
  menu.stream()
      .filter(d -> d.getCalories() > 300)
      .map(Dish::getName)
      .limit(3)
      .collect(toList());
System.out.println(threeHighCaloricDishNames);
```

Get a stream from menu (the list of dishes).

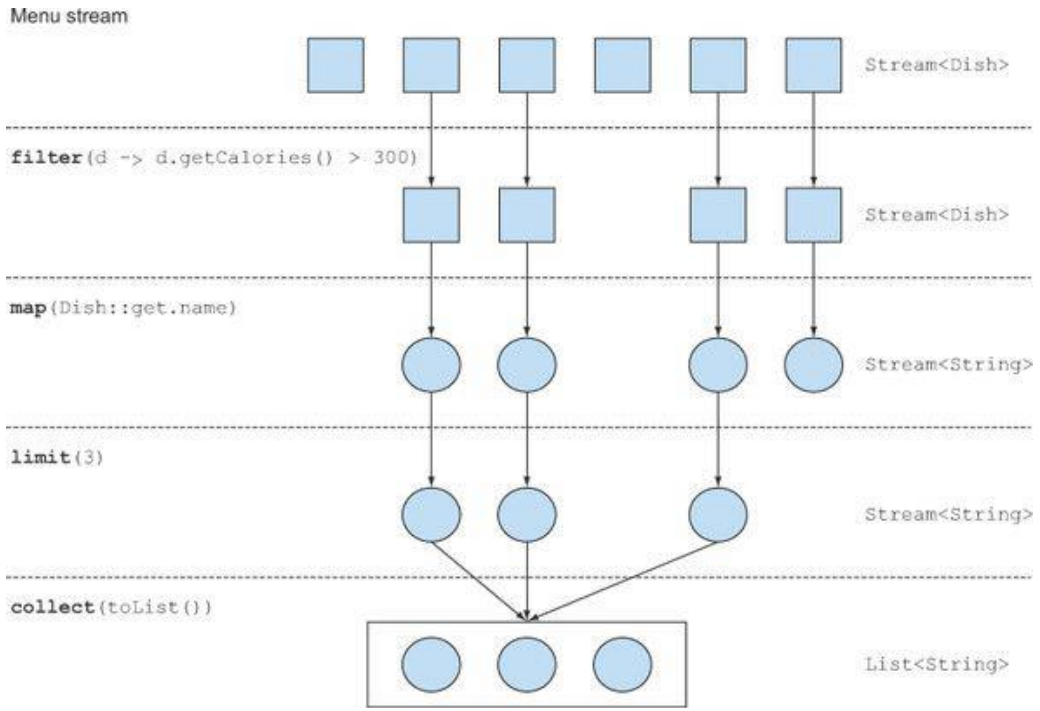
Create a pipeline of operations: first filter high-calorie dishes.

Get the names of the dishes.

Select only the first three.

Store the results in another List.

The result is [pork, beef, chicken].



Another Example Java 8

Stream Operations

- Intermediate Operation
  - Allows the operations to be connected to form a query.
  - Don't perform any processing until a terminal operation is invoked.

- Terminal Operation
  - produce a result from a stream pipeline.
  - A result is any nonstream value such as a List, an Integer, or even void.

Working with Streams

- A *data source* (such as a collection) to perform a query on
- A chain of *intermediate operations* that form a stream pipeline
- A *terminal operation* that executes the stream pipeline and produces a result

List of Intermediate Operations

Operation	ReturnType	Argument of Operation	Function Description
filter	Stream<T>	Predicate<T>	T->boolean
map	Stream<R>	Function<T,R>	T->R
limit	Stream<T>		
sorted	Stream<T>	Comparator<T>	(T,T)->int
distinct	Stream<T>		

List of Terminal operation

Operation	Purpose
forEach	Consumes each element from a stream and applies a lambda to each of them. The operation returns void.
count	Returns the number of elements in a stream. The operation returns a long.
collect	Reduces the stream to create a collection such as a List, a Map, or even an Integer.

External vs. internal iteration

Collections: external iteration with a for-each loop

```
List<String> names = new ArrayList<>();
for(Dish d: menu){
    names.add(d.getName());
}
```

Explicitly iterate the list of menu sequentially.

Extract the name and add it to an accumulator.

Streams: internal iteration

```
List<String> names = menu.stream()
                      .map(Dish::getName)
                      .collect(toList());
```

Start executing the pipeline of operations; no iteration!

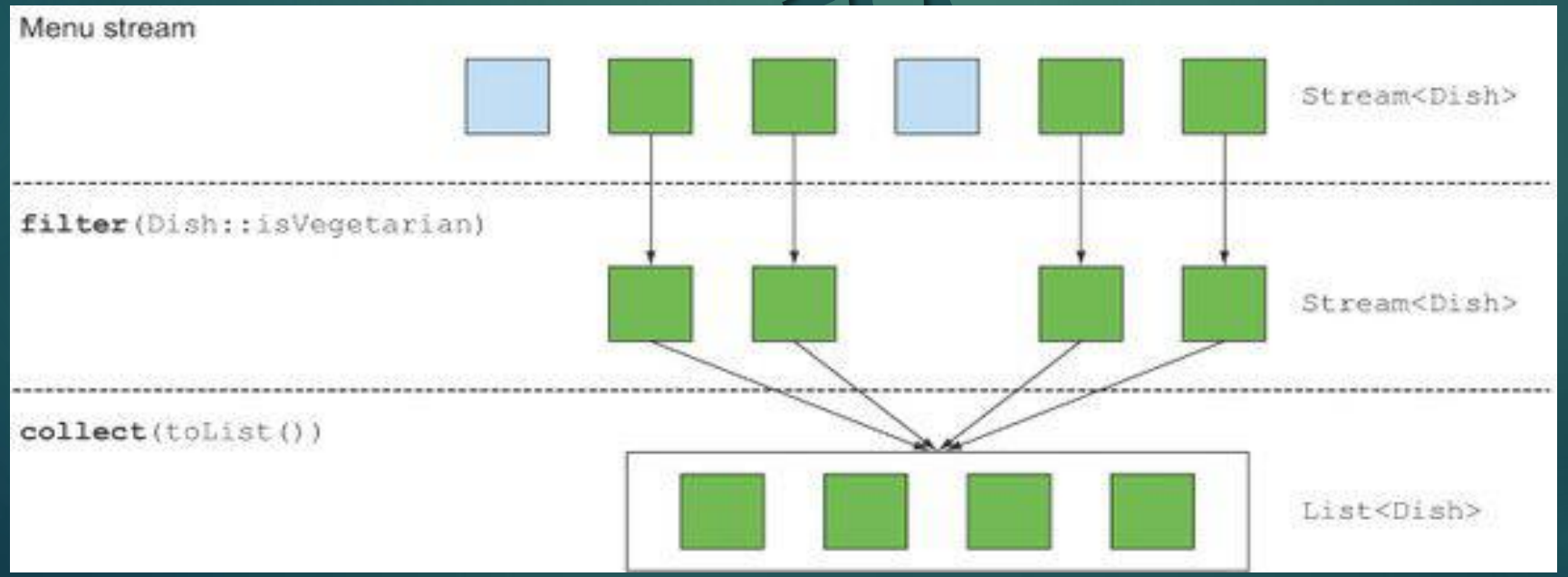
Parameterize map with the getName method to extract the name of a dish.

Filtering

- Filtering a stream with predicate

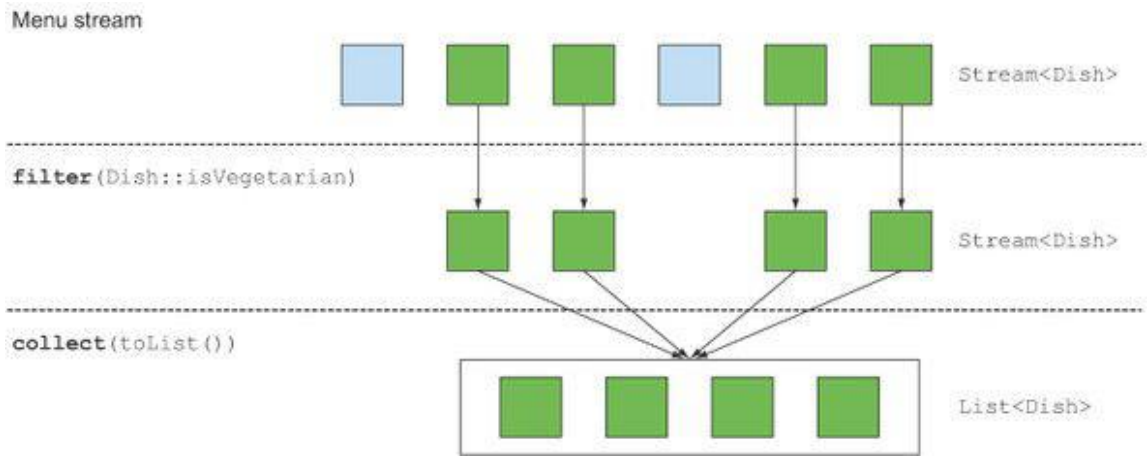
```
List<Dish> vegetarianMenu = menu.stream()
                            .filter(Dish::isVegetarian)
                            .collect(toList());
```

A method reference to check if a dish is vegetarian friendly



Filtering Unique Elements

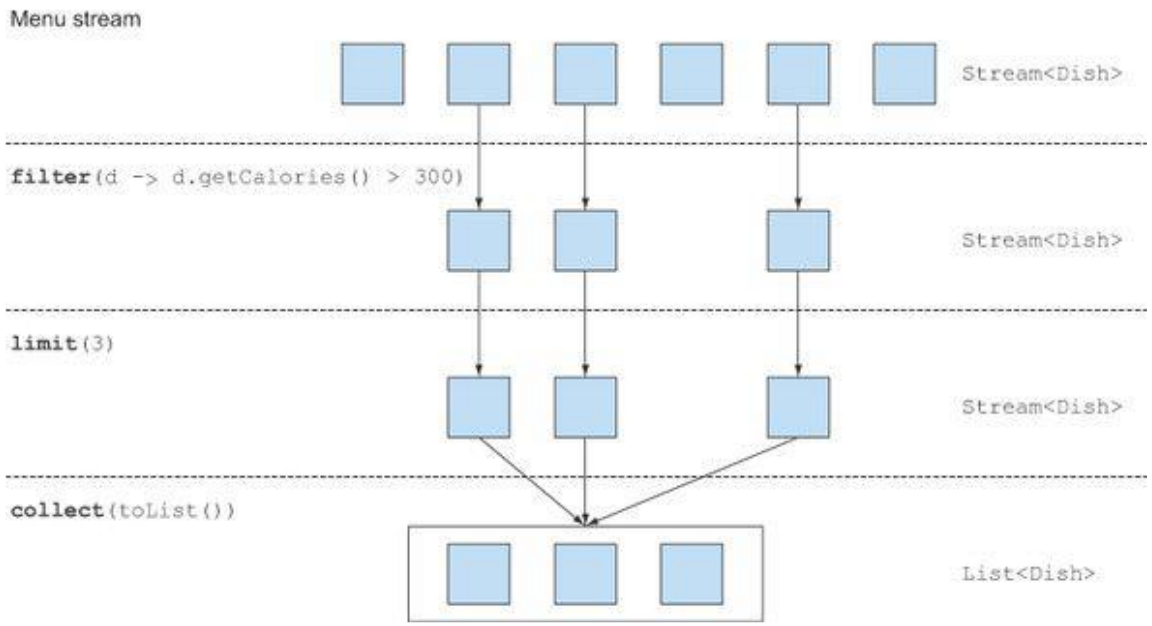
```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
.filter(i -> i % 2 == 0)
.distinct()
.forEach(System.out::println);
```



Truncating a stream

- Streams support the limit(n) method, which returns another stream that's no longer than a given size.

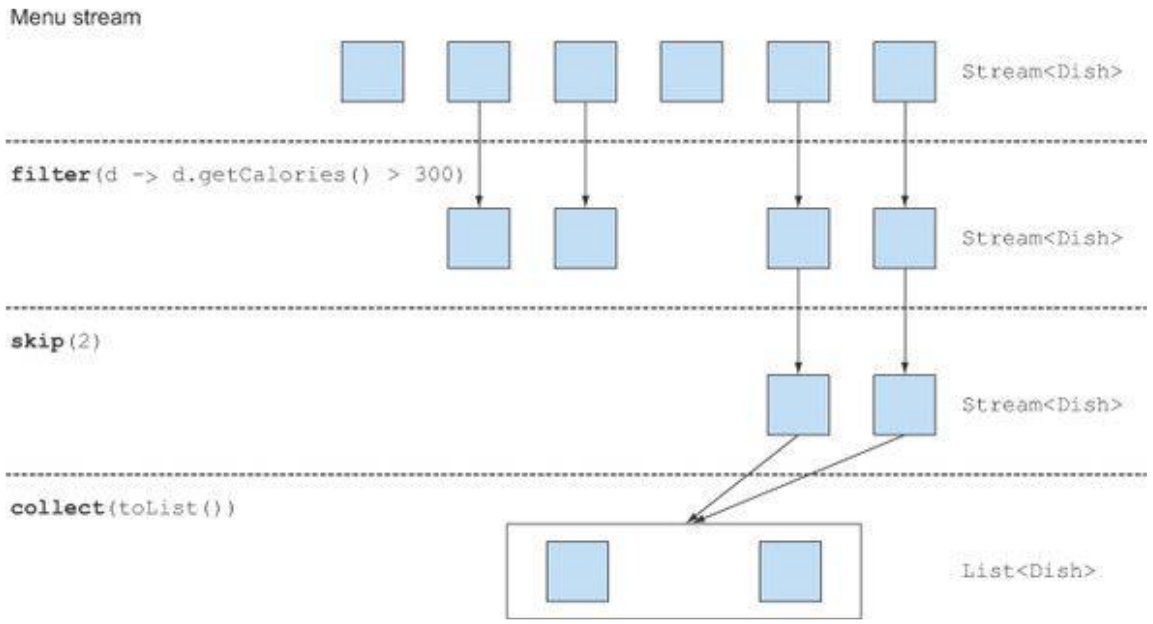
```
List<Dish> dishes =
menu.stream()
.filter(d -> d.getCalories() >
300)
.limit(3)
.collect(toList());
```



Skipping elements

- Streams support the skip(n) method to return a stream that discards the first n elements.
- If the stream has fewer elements than n, then an empty stream is returned.

```
List<Dish> dishes =
menu.stream()
.filter(d -> d.getCalories() > 300)
.skip(2)
.collect(toList());
```



Mapping

- Streams support the method map, which takes a **function** as argument.
- The function is applied to each element, mapping it into a new element
- Examples:

```
List<String> dishNames = menu.stream()
.map(Dish::getName)
.collect(toList());

List<String> words = Arrays.asList("Java8", "Lambdas", "In", "Action");
List<Integer> wordLengths = words.stream()
.map(String::length)
.collect(toList());

List<Integer> dishNameLengths = menu.stream()
.map(Dish::getName)
.map(String::length)
.collect(toList());
```

Finding And Matching

- The Streams API provides finding and matching through the allMatch, anyMatch, noneMatch, findFirst, and findAny methods of a stream.

- **Checking to see if a predicate matches at least one element:**

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
        System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

Note :The anyMatch method returns a boolean and is therefore a terminal operation.

- **Checking to see if a predicate matches all elements**

```
boolean isHealthy = menu.stream()
.allMatch(d -> d.getCalories() < 1000);
```

- **Note:** The allMatch method works similarly to anyMatch but will check to see if all the elements of the stream match the given predicate.

- **noneMatch**
  - ensures that no elements in the stream match thegiven predicate.

```
boolean isHealthy = menu.stream()
.noneMatch(d -> d.getCalories() >= 1000);
```

- **Finding an element**
  - The findAny method returns an arbitrary element of the current stream. It can be used in conjunction with other stream operations.

```
Optional<Dish> dish =menu.stream()
.filter(Dish::isVegetarian)
.findAny();
```

- **Note**: The Optional<T> class (java.util.Optional) is a container class to represent the existence or absence of a value. In the previous code, it's possible that findAny doesn't find any element. Instead of returning null, which is well known for being error prone, the Java 8 library designers introduced Optional<T>.

- Methods available in Optional that force to explicitly check for the presence of a value or deal with the absence of a value:
  - **isPresent() returns true if Optional contains a value, false otherwise.**
  - **ifPresent(Consumer<T> block) executes the given block if a value is present**
  - **T get() returns the value if present; otherwise it throws a NoSuchElement-Exception.**
  - **T orElse(T other) returns the value if present; otherwise it returns a default value.**

- **Finding the First Element**
  - ```
    List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
    Optional<Integer> firstSquareDivisibleByThree =
    someNumbers.stream()
    .map(x -> x * x)
    .filter(x -> x % 3 == 0)
    .findFirst(); // 9
    ```


- **Short-circuiting evaluation**
  - Certain operations such as allMatch, noneMatch, findFirst, and findAny don't need to process the whole stream to produce a result. As soon as an element is found, a result can be produced.
    ```

Reducing

- combine all elements of a stream iteratively to produce a result using the reduce method, for example, to calculate the sum or find the maximum of a stream.
- **Summing the elements**

```
int sum = 0;
for (int x : numbers) {
  sum += x;
}

int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

- //Multiplying using reduce

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

No Initial value

- An overloaded variant of reduce that doesn't take an initial value, but it returns an Optional object:
  - Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));

- Maximum & Minimum

  Optional<Integer> max = numbers.stream().reduce(Integer::max);

  Optional<Integer> min = numbers.stream().reduce(Integer::min)

  or

  Optional<Integer> min=numbers.stream().reduce((x,y)->x<y?x:y)

Numeric Streams

- Java 8 introduces three primitive specialized stream interfaces:
  - IntStream,
  - DoubleStream
  - LongStream

- Mapping to numeric stream
  - Methods use to convert a stream to a specialized version are mapToInt, mapToDouble, and mapToLong


```
int calories = menu.stream()
                    .mapToInt(Dish::getCalories)
                    .sum();
```

Returns an IntStream

Returns a Stream<Dish>

- **Note:** IntStream also supports other convenience methods such as max, min, and average.

Default Values: OptionalInt

```
OptionalInt maxCalories = menu.stream()
.mapToInt(Dish::getCalories)
.max();
```

```
int max = maxCalories.orElse(1);
```

Provide an explicit default maximum if there's no value.

Building streams

- **Streams from values**
  - Stream<String> stream = Stream.of("Sujata ", "Batra ", "Demonstrating ", "Java8");
    stream.map(String::toUpperCase).forEach(System.out::println);
- To get an empty stream
  - Stream<String> emptyStream = Stream.empty();

- **Streams from arrays**
  - **int numbers={2,3,5,7,11,15};**
    int sum=Arrays.stream(numbers).sum();

Streams from functions: creating infinite streams!

- The Streams API provides two static methods to generate a stream from a function:
  - Stream.iterate
  - Stream.generate.

Stream.iterate()

Example:

```
Stream.iterate(0, n -> n + 2)
.limit(10)
.forEach(System.out::println);
```

Fibonacci series

```
Stream.iterate(new int[]{0, 1},
t -> new int[]{t[1],t[0] + t[1]})
.limit(10)
.map(t -> t[0])
.forEach(System.out::println);
```

Stream.generate()

Example

```
Stream.generate(Math::random)
.limit(5)
.forEach(System.out::println);
```

Collecting data with streams

- **collect** is a terminal operation that takes as argument various recipes (called collectors) for accumulating the elements of a stream into a summary result.

- Predefined collectors include reducing and summarizing stream elements into a single value, such as calculating the minimum, maximum, or average.

- Predefined collectors offer three main functionalities:
  - Reducing and summarizing stream elements to a single value
  - Grouping elements
  - Partitioning elements

Reducing and Summarizing

- ***Collectors.counting()***
  - *Counting* is a simple collector that allows simply counting of all *Stream* elements.
  - **Example:**
    long howManyDishes = menu.stream().collect(Collectors.counting());

- ***Collectors.maxBy()/minBy()***
  - *MaxBy/MinBy* collectors return the biggest/the smallest element of a *Stream*according to a provided *Comparator* instance.
  - **Example:**
    Comparator<Dish> dishCaloriesComparator =Comparator.comparingInt(Dish::getCalories);
    Optional<Dish> mostCalorieDish =menu.stream()
    .collect(maxBy(dishCaloriesComparator));

Summarization

- ***Collectors.summingDouble/Long/Int()***
  - *SummingDouble/Long/Int* is a collector that simply returns a sum of extracted elements.
  - **Example:**
    int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));

- ***Collectors.averagingDouble/Long/Int()***
  - *AveragingDouble/Long/Int* is a collector that simply returns an average of extracted elements.
  - **Example:**
    double avgCalories =menu.stream()
    .collect(averagingInt(Dish::getCalories));

- ***Collectors.summarizingDouble/Long/Int()***
  - *SummarizingDouble/Long/Int* is a collector that returns a special class containing statistical information about numerical data in a *Stream* of extracted elements.
  - **Example:**
    IntSummaryStatistics menuStatistics =menu.stream()
    .collect(summarizingInt(Dish::getCalories));

    DoubleSummaryStatistics result = givenList.stream()
    .collect(summarizingDouble(String::length));

Joining Strings

- ***Collectors.joining()***
  - *Joining* collector can be used for joining *Stream<String>* elements.
  - **Example**
    String shortMenu = menu.stream().map(Dish::getName).collect(joining());
    Output: porkbeefchickenfrench friesriceseason fruitpizzaprawnssalmon

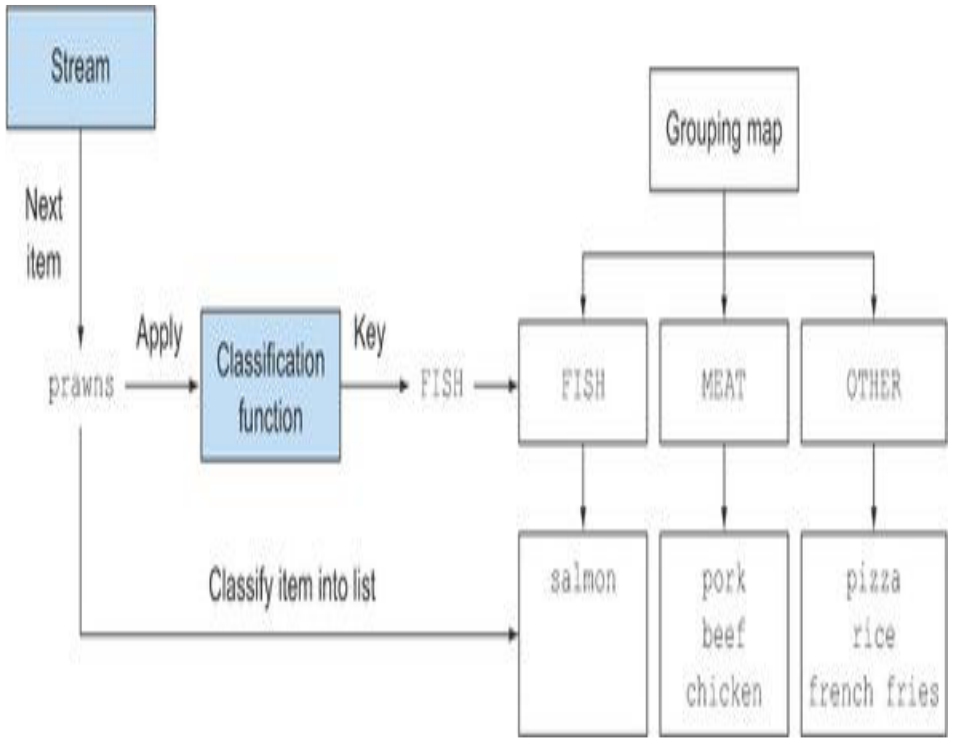
    String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
    Output: pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon

Grouping

- ***Collectors.groupingBy()***
  - *GroupingBy* collector is used for grouping objects by some property and storing results in a *Map* instance.
  - **Example:**
    Map<Dish.Type, List<Dish>> dishesByType =menu.stream()
    .collect(groupingBy(Dish::getType));
    Output:
    {FISH=[prawns, salmon], OTHER=[french fries, rice, season fruit, pizza],MEAT=[pork, beef, chicken]}

    public enum CaloricLevel { DIET, NORMAL, FAT }
    Map<CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream().collect(
    groupingBy(dish -> {
    if (dish.getCalories() <= 400) return CaloricLevel.DIET;
    else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
    else return CaloricLevel.FAT;
    }));

Classification of an item in the stream during the grouping process



MultiLevel grouping

- The result of this two-level grouping is a two-level Map like the following:
- {MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},
- FISH={DIET=[prawns], NORMAL=[salmon]},
- OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}}

```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
menu.stream().collect(
    groupingBy(Dish::getType,
        groupingBy(dish -> {
            if (dish.getCalories() <= 400) return CaloricLevel.DIET;
            else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
            else return CaloricLevel.FAT;
        } )
    )
);
```

First-level classification function

Second-level classification function

Collecting data in subgroups

```
Map<Dish.Type, Long> typesCount = menu.stream()
.collect(groupingBy(Dish::getType, counting()));
```

Output:

{MEAT=3, FISH=2, OTHER=4}

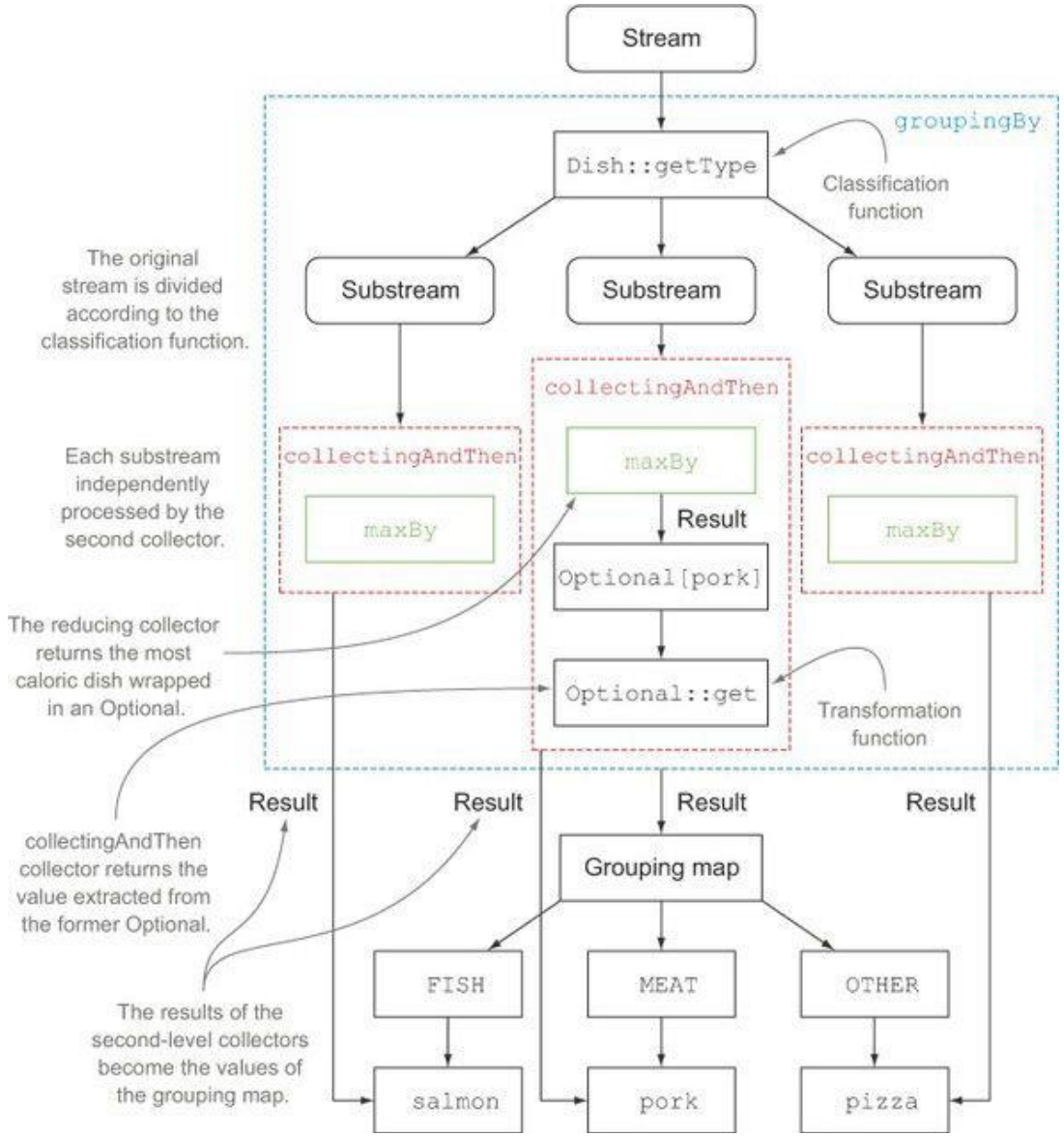
Note: regular one-argument groupingBy(f), where f is the classification function, is in reality just shorthand for groupingBy(f, toList()).

```
Map<Dish.Type, Optional<Dish>> mostCaloricByType =
menu.stream()
.collect(groupingBy(Dish::getType,
maxBy(comparingInt(Dish::getCalories))));
```

Output:
{FISH=Optional[salmon], OTHER=Optional[pizza], MEAT=Optional[pork]}

Adapting the collector result to a different type

The result is the following Map:
{FISH=salmon, OTHER=pizza, MEAT=pork}



Combining the effect of multiple collectors by nesting one inside the other

Partitioning

- ***Collectors.partitioningBy()***
- *PartitioningBy* is a specialized case of *groupingBy* that accepts a *Predicate*instance and collects *Stream* elements into a *Map* instance that stores *Boolean*values as keys and collections as values. Under the "true" key, we can find a collection of elements matching the given *Predicate*, and under the "false" key, we can find a collection of elements not matching the given *Predicate*.

Partitioning function

```
Map<Boolean, List<Dish>> partitionedMenu =
                menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

Output:

{false=[pork, beef, chicken, prawns, salmon],

true=[french fries, rice, season fruit, pizza]}

Examples

menu.stream()

.collect(partitioningBy(Dish::isVegetarian

,partitioningBy(d -> d.getCalories() > 500)));
Output:
{ false={false=[chicken, prawns, salmon], true=[pork, beef]},
true={false=[rice, season fruit], true=[french fries, pizza]}}

menu.stream().collect(partitioningBy(Dish::isVegetarian,counting()));

Output:
{false=5, true=4}


```
Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =
menu.stream().collect(
        partitioningBy(Dish::isVegetarian,
                       groupingBy(Dish::getType)));
```

Partitioning function

Second collector

Output: {false={FISH=[prawns, salmon], MEAT=[pork, beef, chicken]},
true={OTHER=[french fries, rice, season fruit, pizza]}}

- ***Collectors.toList()***
  - *ToList* collector can be used for collecting all *Stream* elements into a *List*instance.
    Example
    List<String> result = givenList.stream()
     .collect(toList());

- ***Collectors.toSet()***
  - *ToSet* collector can be used for collecting all *Stream* elements into a *Set* instance.
    Example
    Set<String> result = givenList.stream()
     .collect(toSet());
- ***Collectors.toCollection()***
  - *toCollection* collector with a provided collection of your choice.
    Example
    List<String> result = givenList.stream()
      .collect(toCollection(LinkedList::new))

Parallel Stream

Parallel Stream

- Internal iteration allows you to process a stream in parallel without the need to explicitly use and coordinate different threads in your code.
- Even if processing a stream in parallel is so easy, there's no guarantee that doing so will make your programs run faster under all circumstances. Behavior and performance of parallel software can sometimes be counterintuitive, and for this reason it's always necessary to measure them and be sure that you're not actually slowing your programs down.
- Parallel execution of an operation on a set of data, as done by a parallel stream, can provide a performance boost, especially when the number of elements to be processed is huge or the processing of each single element is particularly time consuming.
- From a performance point of view, using the right data structure, for instance, employing primitive streams instead of nonspecialized ones whenever possible, is almost always more important than trying to parallelize some operations.
- The fork/join framework lets you recursively split a parallelizable task into smaller tasks, execute them on different threads, and then combine the results of each subtask in order to produce the overall result.
- Spliterators define how a parallel stream can split the data it traverses.

Parallel Stream

- parallelStream() method is used to create a parallel stream of elements.
- parallel() intermediate method can also be called on a given stream to convert a sequential stream to a parallel counterpart.

Example

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

Extending the previous Example

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
        System.out.format("sort: %s <> %s [%s]\n",
            s1, s2, Thread.currentThread().getName());
        return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

sort()/parallelSort()

- sort on a parallel stream uses the new Java 8 method Arrays.parallelSort() under the hood. As stated in Javadoc this method decides on the length of the array if sorting will be performed sequentially or in parallel:

- If the length of the specified array is less than the minimum granularity, then it is sorted using the appropriate Arrays.sort method.