



# 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.

# New Features in Java 8

- ▶ **Lambda Expressions** enable you to treat functionality as a method argument, or code as data.
- ▶ **Method Reference** provide easy-to-read lambda expressions for methods that already have a name.
- ▶ **Default Method** enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
- ▶ Java 8's **new package `java.util.function`** provides many useful functional interfaces for the most common scenarios. The 4 most important functional interface among them are – **Predicate**, **Consumer**, **Function** and **Supplier**.
- ▶ **Repeating Annotations** provide the ability to apply the same annotation type more than once to the same declaration or type use.
- ▶ **New `java.util.stream` package** provides a new **Streams API** to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API.

# New Features in Java 8

- ▶ Java 8's **new Collector interface** and its multiple predefined implementations provide an efficient way to terminate the Stream operations and collect the result in a collection.
- ▶ A **new Date-Time package – java.time** – with a new comprehensive set of date and time utilities.
- ▶ **Concurrency related important changes** which include –
  - ▶ Changes to ConcurrentHashMap to support aggregate operations based on the newly added streams facility and lambda expression.
  - ▶ Addition of classes to the java.util.concurrent.atomic package to support scalable updatable variables.
  - ▶ Support for a common pool in ForkJoinPool.
  - ▶ New **StampedLock** class to provide a capability-based lock with three modes for controlling read/write access.
- ▶ **Type Annotations** provide the ability to apply an annotation anywhere a type is used, not just on a declaration.

# New Features in Java 8

- ▶ **JDBC 4.2** with new features and notably the JDBC-ODBC bridge has been removed.
- ▶ **Nashorn Javascript Engine** enhanced to provide a version of javascript which would run within the JVM.

Sujata Batra

# Behaviour Parameterization

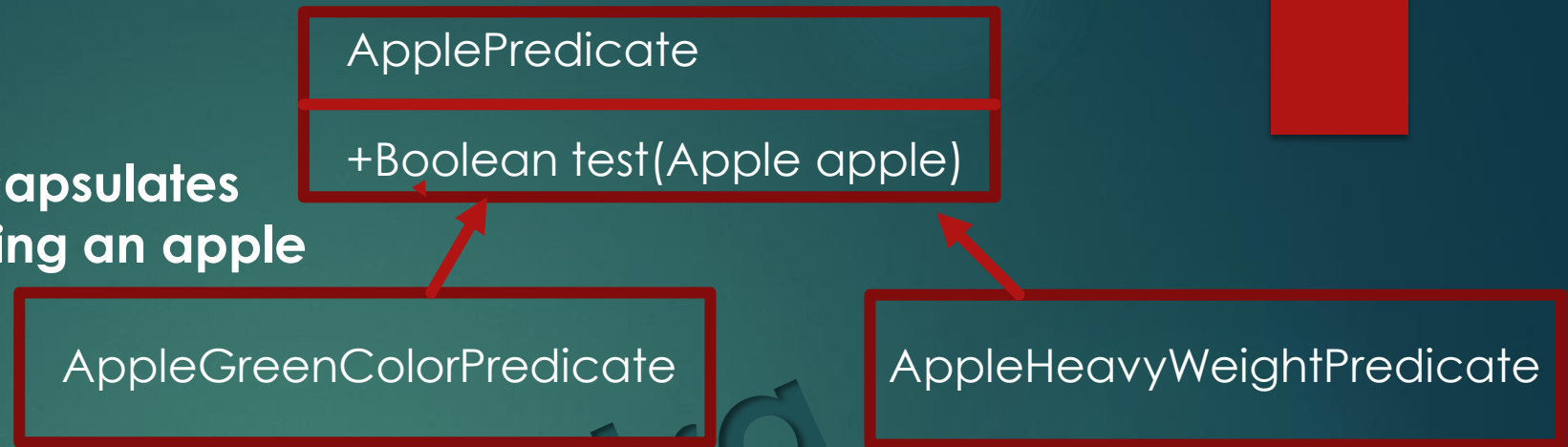
- ▶ Behavior parameterization is the ability for a method to take multiple different behaviors as parameters and use them internally to accomplish different behaviors.
- ▶ Behavior parameterization lets us make your code more adaptive to changing requirements and saves on engineering efforts in the future.

Sujata Bhatia



# 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.
- ▶ Lambda 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

Sujata Beotra



# Functional Interfaces in Java 8

- ▶ **A functional interface**, is an interface which has only a single abstract method.
- ▶ **@FunctionalInterface annotation**
  - ▶ used to explicitly 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)  
}`

Sujata Batra

# 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 Expression**

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

**//Method Reference**

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

# Method References

Sujata Batra

# Types of Method Reference

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

Sujata Batra



# Type 1: Reference to a static method

## ► Lambda Syntax:

- (arguments) -> <ClassName>.<staticMethodName>(arguments);

## ► Equivalent Method Reference:

- <ClassName> :: <staticMethodName>

## Example

```
//Lambda Expression
```

```
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>

Sujata Batra

# 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;  
    //Constructor 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

Sujata Batra

# 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.

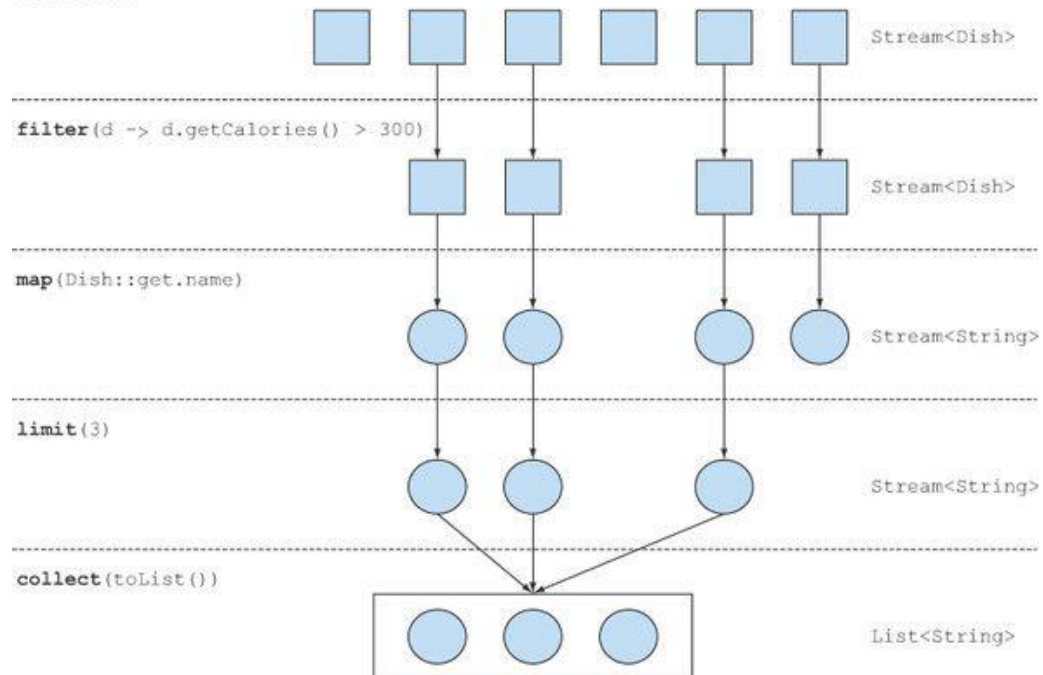
Get the names of the dishes.

The result is [pork, beef, chicken].

Select only the first three.

Store the results in another List.

Menu stream



# 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	Return Type	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

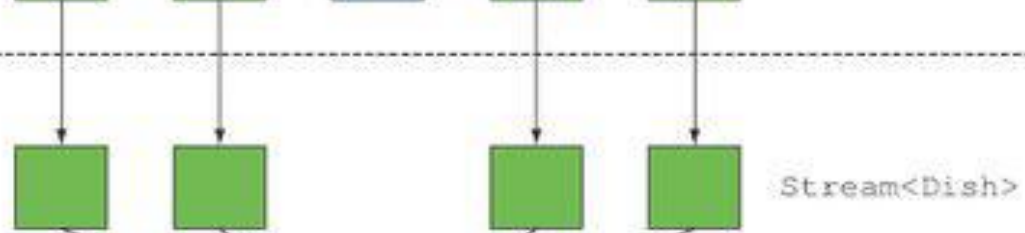
```
List<Dish> vegetarianMenu = menu.stream()
    .filter(Dish::isVegetarian)
    .collect(toList());
```

A method reference  
to check if a dish is  
vegetarian friendly

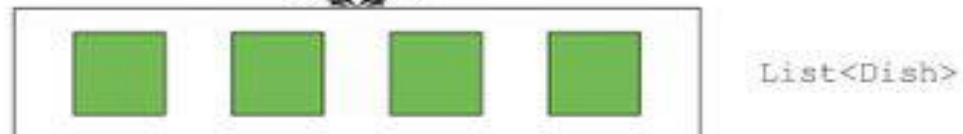
Menu stream



`filter(Dish::isVegetarian)`

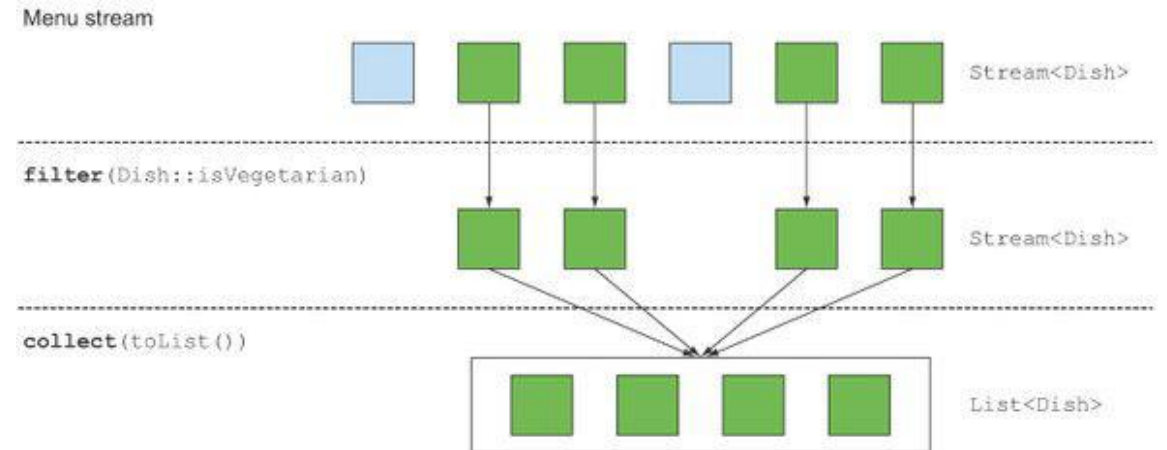


`collect(toList())`



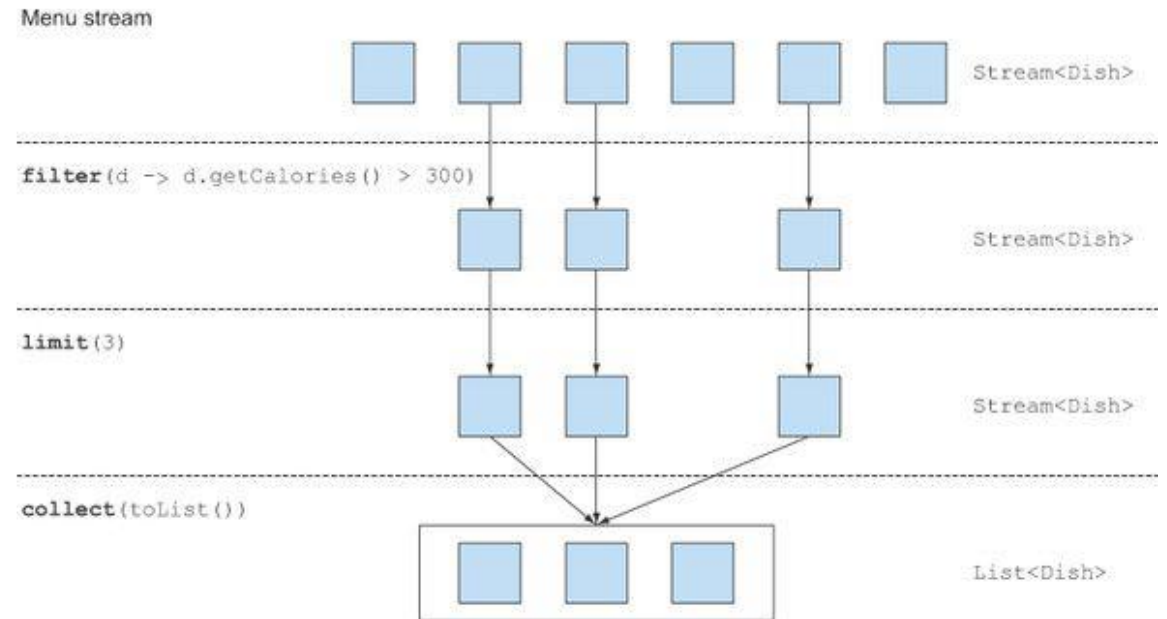
# Filtering Unique Elements

```
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

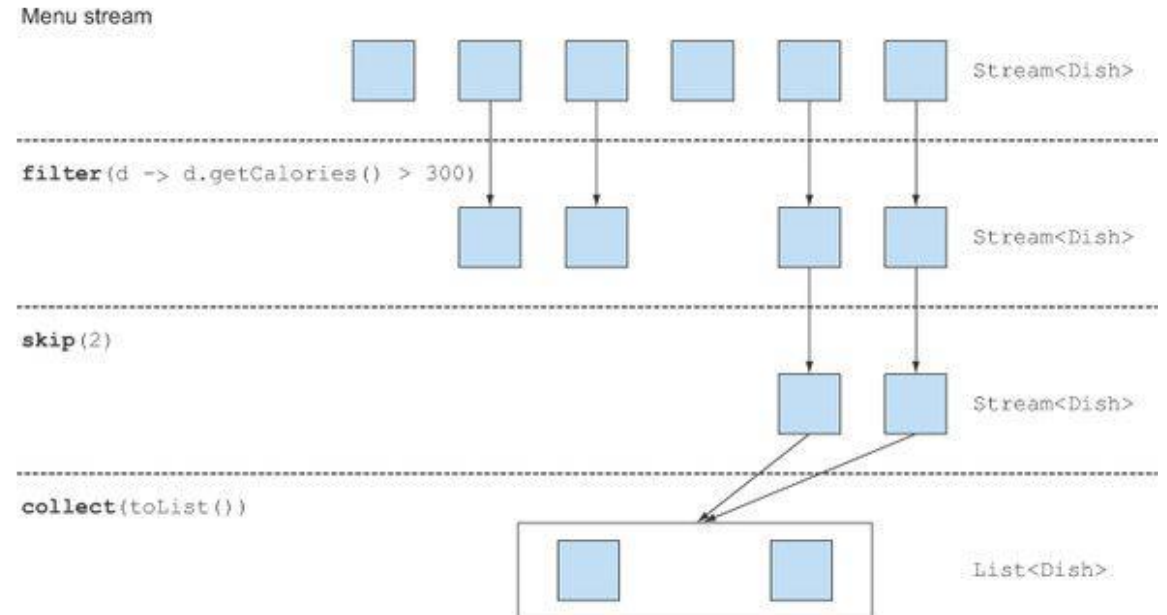
- Streams support the `limit(n)` method, which returns another stream that's no longer than a given size.  
`List<Dish> dishes = menu.stream().filter(d -> d.getCalories() > 300).limit(3).collect(toList());`



# 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 the given 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 `NoSuchElementException`.
- ▶ `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 a Stream<Dish>

← Returns an IntStream

- ▶ **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`.

Sujata Batra



# 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);
```

Sujata Batra

# 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 friesricesseason 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:**

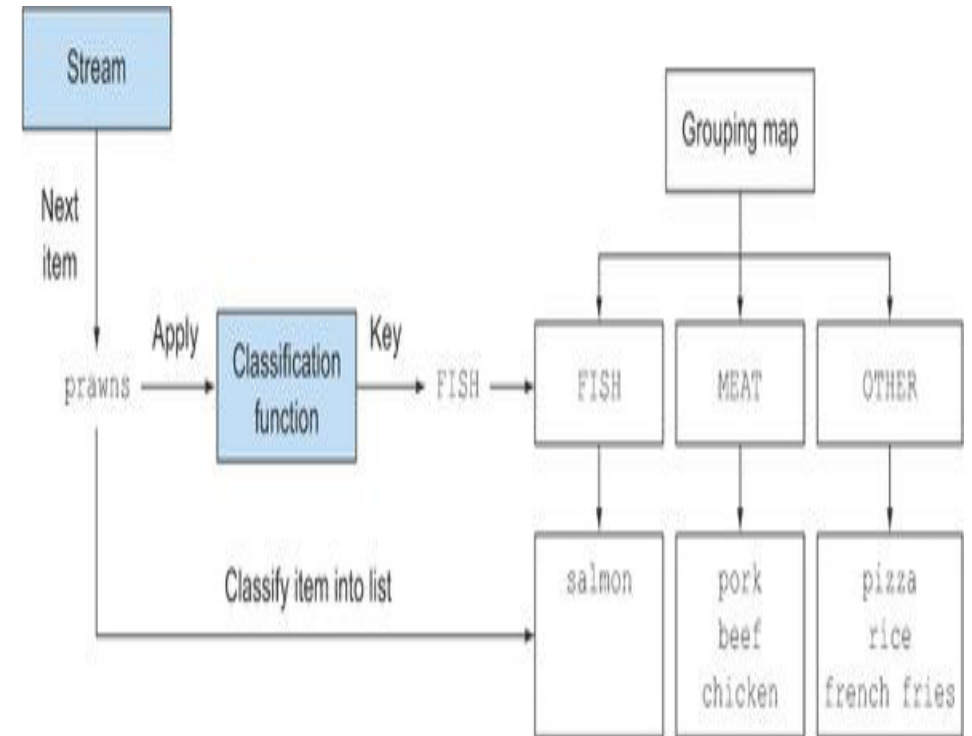
```
Map<Dish.Type, List<Dish>> dishesByType = menu.stream()  
.collect(groupingBy(Dish::getType));
```

#### **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))));
```

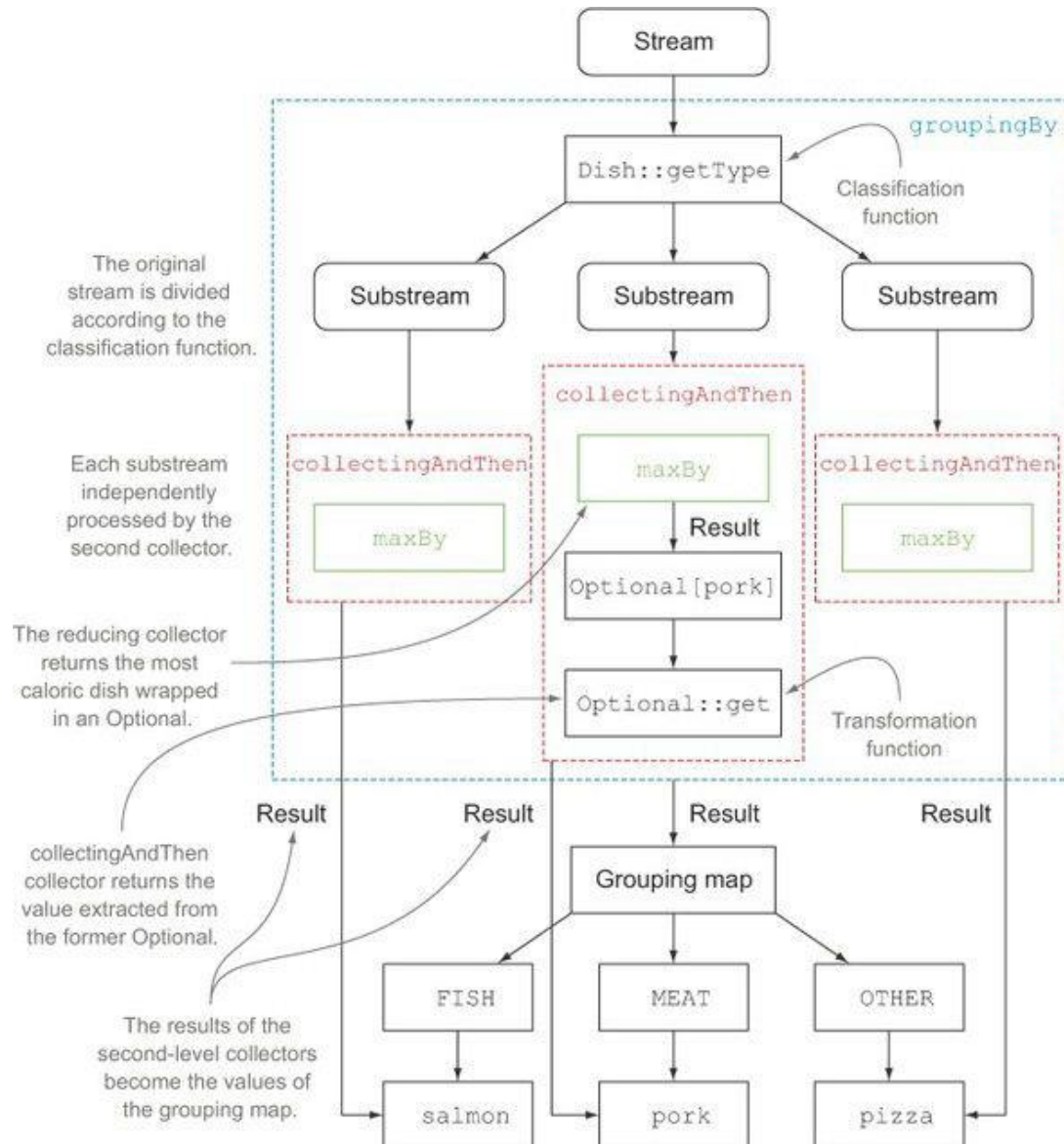
## Output:

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

# Adapting the collector result to a different type

```
Map<Dish.Type, Dish> mostCaloricByType =  
    menu.stream()  
        .collect(groupingBy(Dish::getType,  
                             ← Classification  
                             function  
                             collectingAndThen(  
                                 ← Wrapped  
                                 maxBy(comparingInt(Dish::getCalories)),  
                                 collector  
                                 Optional::get)));  
    ← Transformation  
    function
```

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*.

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

Partitioning function

## 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:

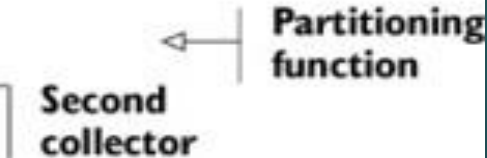
```
{ false={false=[chicken, prawns, salmon], true=[pork, beef]},  
  true={false=[rice, season fruit], true=[french fries, pizza]}}
```

```
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));
```



The diagram shows a horizontal line representing the lambda expression in the collect call. An arrow points from the text "Partitioning function" to the `partitioningBy` part of the lambda. Another arrow points from the text "Second collector" to the `groupingBy` part of the lambda.

**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))
```

# Default Methods

Sujata Patra

# Default Methods

- ▶ Interfaces in Java 8 can have implementation code through default methods and static methods.
- ▶ Default methods start with a default keyword and contain a body like class methods do.
- ▶ Adding an abstract method to a published interface is a source incompatibility.
- ▶ Default methods help library designers evolve APIs in a backward-compatible way.
- ▶ Default methods can be used for creating optional methods and multiple inheritance of behavior.
- ▶ There are resolution rules to resolve conflicts when a class inherits from several default methods with the same signature.
- ▶ A method declaration in the class or a superclass takes priority over any default method declaration.
- ▶ Otherwise, the method with the same signature in the most specific default-providing interface is selected.
- ▶ When two methods are equally specific, a class can explicitly override a method and select which one to call.

# Abstract classes VS interfaces in Java 8

- ▶ Abstract class can define constructor.
- ▶ Abstract classes are more structured and can have a state associated with them.
- ▶ Default method can be implemented only in the terms of invoking other interface methods, with no reference to a particular implementation's state.

# Default Method and Multiple Inheritance Ambiguity Problems

- ▶ Since java class can implement multiple interfaces and each interface can define default method with same method signature, therefore, the inherited methods can conflict with each other.

Sujata Batra



# Example

```
public interface InterfaceA {  
    default void defaultMethod(){  
        System.out.println("Interface A default method");  
    }  
}  
  
public interface InterfaceB {  
    default void defaultMethod(){  
        System.out.println("Interface B default method");  
    }  
}  
  
public class Impl implements InterfaceA, InterfaceB {  
}
```



- **In order to fix this class, we need to provide default method implementation:**

```
public class Impl implements InterfaceA, InterfaceB {  
    public void defaultMethod(){  
    }  
}
```

- **To invoke default implementation provided by any of super interface rather than our own implementation.**

```
public class Impl implements InterfaceA, InterfaceB {  
    public void defaultMethod(){  
        // existing code here..  
        InterfaceA.super.defaultMethod();  
    }  
}
```

# Appendix

Sujata Batra

# Java Predefined Functional Interface

Interface	Description
BiConsumer<T,U>	It represents an operation that accepts two input arguments and returns no result.
Consumer<T>	It represents an operation that accepts a single argument and returns no result.
Function<T,R>	It represents a function that accepts one argument and returns a result.
Predicate<T>	It represents a predicate (boolean-valued function) of one argument.
BiFunction<T,U,R>	It represents a function that accepts two arguments and returns a a result.
BinaryOperator<T>	It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands.
BiPredicate<T,U>	It represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	It represents a supplier of boolean-valued results.
DoubleBinaryOperator	It represents an operation upon two double type operands and returns a double type value.

# Java Predefined Functional Interface

DoubleConsumer	It represents an operation that accepts a single double type argument and returns no result.
DoubleFunction<R>	It represents a function that accepts a double type argument and produces a result.
DoublePredicate	It represents a predicate (boolean-valued function) of one double type argument.
DoubleSupplier	It represents a supplier of double type results.
DoubleToIntFunction	It represents a function that accepts a double type argument and produces an int type result.
DoubleToLongFunction	It represents a function that accepts a double type argument and produces a long type result.
DoubleUnaryOperator	It represents an operation on a single double type operand that produces a double type result.
IntBinaryOperator	It represents an operation upon two int type operands and returns an int type result.
IntConsumer	It represents an operation that accepts a single integer argument and returns no result.
IntFunction<R>	It represents a function that accepts an integer argument and returns a result.
IntPredicate	It represents a predicate (boolean-valued function) of one integer argument.
IntSupplier	It represents a supplier of integer type.

# Java Predefined Functional Interface

IntToDoubleFunction	It represents a function that accepts an integer argument and returns a double.
IntToLongFunction	It represents a function that accepts an integer argument and returns a long.
IntUnaryOperator	It represents an operation on a single integer operand that produces an integer result.
LongBinaryOperator	It represents an operation upon two long type operands and returns a long type result.
LongConsumer	It represents an operation that accepts a single long type argument and returns no result.
LongFunction<R>	It represents a function that accepts a long type argument and returns a result.
LongPredicate	It represents a predicate (boolean-valued function) of one long type argument.
LongSupplier	It represents a supplier of long type results.
LongToDoubleFunction	It represents a function that accepts a long type argument and returns a result of double type.
LongToIntFunction	It represents a function that accepts a long type argument and returns an integer result.
LongUnaryOperator	It represents an operation on a single long type operand that returns a long type result.



# Java Predefined Functional Interface

<code>ObjDoubleConsumer&lt;T&gt;</code>	It represents an operation that accepts an object and a double argument, and returns no result.
<code>ObjIntConsumer&lt;T&gt;</code>	It represents an operation that accepts an object and an integer argument. It does not return result.
<code>ObjLongConsumer&lt;T&gt;</code>	It represents an operation that accepts an object and a long argument, it returns no result.
<code>Supplier&lt;T&gt;</code>	It represents a supplier of results.
<code>ToDoubleBiFunction&lt;T,U&gt;</code>	It represents a function that accepts two arguments and produces a double type result.
<code>ToDoubleFunction&lt;T&gt;</code>	It represents a function that returns a double type result.
<code>ToIntBiFunction&lt;T,U&gt;</code>	It represents a function that accepts two arguments and returns an integer.
<code>ToIntFunction&lt;T&gt;</code>	It represents a function that returns an integer.
<code>ToLongBiFunction&lt;T,U&gt;</code>	It represents a function that accepts two arguments and returns a result of long type.
<code>ToLongFunction&lt;T&gt;</code>	It represents a function that returns a result of long type.
<code>UnaryOperator&lt;T&gt;</code>	It represents an operation on a single operand that returns a result of the same type as its operand.

# Java Stream Interface Methods

Methods	Description
<code>boolean allMatch(Predicate&lt;? super T&gt; predicate)</code>	It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
<code>boolean anyMatch(Predicate&lt;? super T&gt; predicate)</code>	It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.
<code>static &lt;T&gt; Stream.Builder&lt;T&gt; builder()</code>	It returns a builder for a Stream.
<code>&lt;R,A&gt; R collect(Collector&lt;? super T,A,R&gt; collector)</code>	It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to <code>collect(Supplier, BiConsumer, BiConsumer)</code> , allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.



# Java Stream Interface Methods

<code>&lt;R&gt; R collect(Supplier&lt;R&gt; supplier, BiConsumer&lt;R,? super T&gt; accumulator, BiConsumer&lt;R,R&gt; combiner)</code>	It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.
<code>static &lt;T&gt; Stream&lt;T&gt; concat(Stream&lt;? extends T&gt; a, Stream&lt;? extends T&gt; b)</code>	It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.
<code>long count()</code>	It returns the count of elements in this stream. This is a special case of a reduction.
<code>Stream&lt;T&gt; distinct()</code>	It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
<code>static &lt;T&gt; Stream&lt;T&gt; empty()</code>	It returns an empty sequential Stream.

# Java Stream Interface Methods

<code>Stream&lt;T&gt; filter(Predicate&lt;? super T&gt; predicate)</code>	It returns a stream consisting of the elements of this stream that match the given predicate.
<code>Optional&lt;T&gt; findAny()</code>	It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
<code>Optional&lt;T&gt; findFirst()</code>	It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned.
<code>&lt;R&gt; Stream&lt;R&gt; flatMap(Function&lt;? super T,? extends Stream&lt;? extends R&gt;&gt; mapper)</code>	It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
<code>DoubleStream flatMapToDouble(Function&lt;? super T,? extends DoubleStream&gt; mapper)</code>	It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.)

# Java Stream Interface Methods

<code>IntStream flatMapToInt(Function&lt;? super T,? extends IntStream&gt; mapper)</code>	It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
<code>LongStream flatMapToLong(Function&lt;? super T,? extends LongStream&gt; mapper)</code>	It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
<code>void forEach(Consumer&lt;? super T&gt; action)</code>	It performs an action for each element of this stream.
<code>void forEachOrdered(Consumer&lt;? super T&gt; action)</code>	It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
<code>static &lt;T&gt; Stream&lt;T&gt; generate(Supplier&lt;T&gt; s)</code>	It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc.

# Java Stream Interface Methods

<code>static &lt;T&gt; Stream&lt;T&gt; iterate(T seed,UnaryOperator&lt;T&gt; f)</code>	It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
<code>Stream&lt;T&gt; limit(long maxSize)</code>	It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
<code>&lt;R&gt; Stream&lt;R&gt; map(Function&lt;? super T,? extends R&gt; mapper)</code>	It returns a stream consisting of the results of applying the given function to the elements of this stream.
<code>DoubleStream mapToDouble(ToDoubleFunction&lt;? super T&gt; mapper)</code>	It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
<code>IntStream mapToInt(ToIntFunction&lt;? super T&gt; mapper)</code>	It returns an IntStream consisting of the results of applying the given function to the elements of this stream.
<code>LongStream mapToLong(ToLongFunction&lt;? super T&gt; mapper)</code>	It returns a LongStream consisting of the results of applying the given function to the elements of this stream.



# Java Stream Interface Methods

<code>Optional&lt;T&gt; max(Comparator&lt;? super T&gt; comparator)</code>	It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction.
<code>Optional&lt;T&gt; min(Comparator&lt;? super T&gt; comparator)</code>	It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction.
<code>boolean noneMatch(Predicate&lt;? super T&gt; predicate)</code>	It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
<code>@SafeVarargs static &lt;T&gt; Stream&lt;T&gt; of(T... values)</code>	It returns a sequential ordered stream whose elements are the specified values.
<code>static &lt;T&gt; Stream&lt;T&gt; of(T t)</code>	It returns a sequential Stream containing a single element.
<code>Stream&lt;T&gt; peek(Consumer&lt;? super T&gt; action)</code>	It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

# Java Stream Interface Methods

Optional<T> reduce(BinaryOperator<T> accumulator)	It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T reduce(T identity, BinaryOperator<T> accumulator)	It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T> skip(long n)	It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned.
Stream<T> sorted()	It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.

# Java Stream Interface Methods

<code>Stream&lt;T&gt; sorted(Comparator&lt;? super T&gt; comparator)</code>	It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
<code>Object[] toArray()</code>	It returns an array containing the elements of this stream.
<code>&lt;A&gt; A[] toArray(IntFunction&lt;A[]&gt; generator)</code>	It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.