ADVANCED JAVA

Modern Java: Lambdas and Streams

Vivek Shah bonii@di.ku.dk

August 24, 2018

DIKU, University of Copenhagen

CREDITS

The code snippets and the images for the slide have been taken from the book **Modern Java In Action**

New concepts and functionality to ease the writing of programs that are both effective and concise

New concepts and functionality to ease the writing of programs that are both effective and concise

- Lambdas - Use functions as values
- Streams - Declarative and functional data processing

BEHAVIOR PARAMETERIZATION

Change is the only constant in software development

Programming languages can go a long way to help deal with software changes

Consider the following simple filter to filter green apples

```
enum Color { RED, GREEN }

public static List<Apple> filterGreenApples
                        (List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory){
        if( GREEN.equals(apple.getColor() ) {
            result.add(apple);
        }
    }
    return result;
}
```

Consider the following simple filter to filter green apples

```
enum Color { RED, GREEN }

public static List<Apple> filterGreenApples
                         (List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory){
        if( GREEN.equals(apple.getColor() ) {
            result.add(apple);
        }
    }
    return result;
}
```

What if you need to filter red apples or filter apples by weight or some other criteria?

```java
public interface ApplePredicate{
    boolean test (Apple apple);
}

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

BEHAVIOR PARAMETERIZATION USING STRATEGY PATTERN (CONTD.)

```java
public class AppleHeavyWeightPredicate implements
                                    ApplePredicate {
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}

public class AppleGreenColorPredicate implements
                                    ApplePredicate {
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}

public class AppleRedAndHeavyPredicate implements
                                    ApplePredicate {
    public boolean test(Apple apple){
        return RED.equals(apple.getColor())
        && apple.getWeight() > 150;
    }
}

List<Apple> redAndHeavyApples = filterApples(inventory,
                  new AppleRedAndHeavyPredicate());
```

```java
List<Apple> redApples = filterApples(inventory,
       new ApplePredicate() {
         public boolean test(Apple apple){
           return RED.equals(apple.getColor());
       }
});
```

```
List<Apple> redApples = filterApples(inventory,
        new ApplePredicate() {
          public boolean test(Apple apple){
            return RED.equals(apple.getColor());
        }
});
```

Can we make it even more concise ?

```java
List<Apple> redApples = filterApples(inventory,
       new ApplePredicate() {
         public boolean test(Apple apple){
           return RED.equals(apple.getColor());
       }
});
```

Can we make it even more concise ?

```java
List<Apple> result =  filterApples(inventory,
     (Apple apple) -> RED.equals(apple.getColor()));
```

BEHAVIOR PARAMETERIZATION USING GENERICS AND LAMBDAS

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

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

List<Apple> redApples =
  filter(inventory, (Apple apple) -> RED.equals(apple.getColor()));
List<Integer> evenNumbers =
  filter(numbers, (Integer i) -> i % 2 == 0);
```

SOME CONCISE USAGE OF LAMBDAS

- Creating ad-hoc Comparator

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

SOME CONCISE USAGE OF LAMBDAS

- Creating ad-hoc Comparator

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

- Executing a block of code using Runnable

```
Thread t = new Thread(
            () -> System.out.println("Hello world"));
```

- Creating ad-hoc Comparator

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

- Executing a block of code using Runnable

```
Thread t = new Thread(
            () -> System.out.println("Hello world"));
```

- Returning a Future result using Callable

```
Future<String> threadName = executorService.submit(
            () -> Thread.currentThread().getName());
```

LAMBDA EXPRESSIONS

A lambda expression is a concise representation of an anonymous function that can be passed around

A lambda expression is a concise representation of an anonymous function that can be passed around

Properties of lambda expressions:

- Anonymous - Does not have an explicit name
- Function - Is not associated with a particular class but similar to methods in classes
- Passed As Values - Can be passed as argument to method or stored in a variable
- Concise - Minimal boilerplate code is required

ANATOMY OF A LAMBDA

$$(\textit{parameters}) \rightarrow \textit{expression}$$
$$(\textit{parameters}) \rightarrow \{\ \textit{statements};\ \}$$

$$(\textit{parameters}) \rightarrow \textit{expression}$$
$$(\textit{parameters}) \rightarrow \{\ \textit{statements};\ \}$$

A lambda expression consists of:

- A list of parameters
- The body of the lambda consisting of either an expression (considered the return value) or a set of statements
- An arrow separating the list of parameters and the body

Syntax chosen based on popularity of the concept in C# and Scala

EXAMPLE USES OF LAMBDAS

Use Case	Example of Lambda

EXAMPLE USES OF LAMBDAS

Use Case	Example of Lambda
A boolean expression	`(List<String> list) -> list.isEmpty()`

EXAMPLE USES OF LAMBDAS

Use Case	Example of Lambda
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`

EXAMPLE USES OF LAMBDAS

Use Case	Example of Lambda
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`
Consuming from an object	`(Apple a) -> { System.out.println(a.getWeight()); }`

Use Case	Example of Lambda
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`
Consuming from an object	`(Apple a) -> { System.out.println(a.getWeight()); }`
Select/extract from an object	`(String s) -> s.length()`

Use Case	Example of Lambda
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`
Consuming from an object	`(Apple a) -> { System.out.println(a.getWeight()); }`
Select/extract from an object	`(String s) -> s.length()`
Combine two values	`(int a, int b) -> a * b`

Use Case	Example of Lambda
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`
Consuming from an object	`(Apple a) -> { System.out.println(a.getWeight()); }`
Select/extract from an object	`(String s) -> s.length()`
Combine two values	`(int a, int b) -> a * b`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`

Where and how can we use a lambda

Where and how can we use a lambda (what is its type) ?

You can use a lambda in the context of any *functional interface*

FUNCTIONAL INTERFACES

Functional interface - An interface that specifies exactly one abstract method

Functional interface - An interface that specifies exactly one abstract method

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
public interface Runnable {
    void run();
}
public interface ActionListener extends EventListener {
    void actionPerformed(ActionEvent e);
}
public interface Callable<V> {
    V call() throws Exception;
}
public interface PrivilegedAction<T> {
    T run();
}
```

FUNCTIONAL INTERFACES (CONTD.)

- Lambda expressions provide the implementation of the abstract method of a functional interface directly inline

FUNCTIONAL INTERFACES (CONTD.)

- Lambda expressions provide the implementation of the abstract method of a functional interface directly inline
- Type signature of the abstract method describes the signature of the lambda expression

- Lambda expressions provide the implementation of the abstract method of a functional interface directly inline
- Type signature of the abstract method describes the signature of the lambda expression

```
public void process(Runnable r) {
  r.run();
}
process(() -> System.out.println("This is awesome!!"));

Runnable r = () -> System.out.println("Is this awesome?");
process(r);
```

The Java library provides quite a few functional interfaces to help out

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
public <T> List<T> filter(List<T> list, Predicate<T> p) {
  List<T> results = new ArrayList<>();
  for(T t: list) {
    if(p.test(t)) {
      results.add(t);
    }
  }
  return results;
}
Predicate<String> nonEmptyStringPredicate =
                             (String s) -> !s.isEmpty();
List<String> nonEmpty =
              filter(listOfStrings, nonEmptyStringPredicate);
```

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}
public <T, R> List<R> map(List<T> list, Function<T, R> f) {
  List<R> result = new ArrayList<>();
  for(T t: list) {
    result.add(f.apply(t));
  }
  return result;
}
List<Integer> l = map(
                  Arrays.asList("lambdas", "in", "action"),
                  (String s) -> s.length());
```

- There are many more functional interfaces

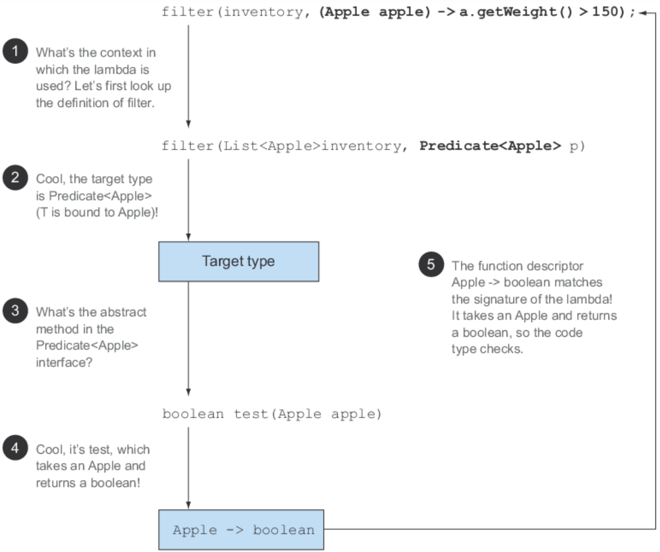
- There are many more functional interfaces
  - Some common examples are `Consumer<T>, Supplier<T>, UnaryOperator<T>, BinaryOperator<T>, BiPredicate<T, U>, BiConsumer<T, U>, BiFunction<T, U, R>`
  - Primitive specializations for autoboxing performance issues
  - Explore them and many more

- There are many more functional interfaces
    - Some common examples are `Consumer<T>, Supplier<T>, UnaryOperator<T>, BinaryOperator<T>, BiPredicate<T, U>, BiConsumer<T, U>, BiFunction<T, U, R>`
    - Primitive specializations for autoboxing performance issues
    - Explore them and many more
    - Add your own based on your needs.
    - Good practice to annotate the interface with `@FunctionalInterface`

- A lambda expression itself does not contain the information about which functional interface it is implementing

- A lambda expression itself does not contain the information about which functional interface it is implementing
- The type of a lambda is deduced from the context in which the lambda is used

- A lambda expression itself does not contain the information about which functional interface it is implementing
- The type of a lambda is deduced from the context in which the lambda is used
- Type inference of lambda expressions in a nutshell
    - Deduce the functional interface based on the usage, bind the generic type parameters
    - Find the abstract method in the interface and bind the generic type parameters
    - Match the parameter and return types of the lambda expression with the abstract method



Example of Type Checking in Lambda (Modern Java in Action)

Because of type inference of lambda expressions you can assign the same lambda expression to different functional interfaces

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

Comparator<Apple> c1 = (Apple a1, Apple a2) ->
                  a1.getWeight().compareTo(a2.getWeight());
ToIntBiFunction<Apple, Apple> c2 = (Apple a1, Apple a2) ->
                  a1.getWeight().compareTo(a2.getWeight());
//You can omit the type of the parameters as well
BiFunction<Apple, Apple, Integer> c3 = (a1, a2) ->
                  a1.getWeight().compareTo(a2.getWeight());
```

Lambda expressions can reference the following variables in their body

- Parameters of the lambda expression
- Instance variables or static variables
- Local variables *that are either final or assigned only once*

Lambda expressions can reference the following variables in their body

- Parameters of the lambda expression
- Instance variables or static variables
- Local variables *that are either final or assigned only once*
- Restrictions based on the implementation of variables in Java and to prevent imperative programming practices
- Are not closures

METHOD REFERENCES

Method references allow reuse of existing method definitions and pass them just like lambdas

Syntactic sugar for lambdas that refer only to a single method

Method references allow reuse of existing method definitions and pass them just like lambdas

Syntactic sugar for lambdas that refer only to a single method

Lambda Expression	Method Reference

Method references allow reuse of existing method definitions and pass them just like lambdas

Syntactic sugar for lambdas that refer only to a single method

Lambda Expression	Method Reference
`(Apple apple) -> apple.getWeight()`	`Apple::getWeight`

Method references allow reuse of existing method definitions and pass them just like lambdas

Syntactic sugar for lambdas that refer only to a single method

Lambda Expression	Method Reference
`(Apple apple) -> apple.getWeight()`	`Apple::getWeight`
`(str, i) -> str.substring(i)`	`String::substring`

Method references allow reuse of existing method definitions and pass them just like lambdas

Syntactic sugar for lambdas that refer only to a single method

Lambda Expression	Method Reference
`(Apple apple) -> apple.getWeight()`	`Apple::getWeight`
`(str, i) -> str.substring(i)`	`String::substring`
`(String s) -> this.isValidName(s)`	`this::isValidName`

Several of the functional interfaces in Java contain convenience methods to allow composition and combination of lambda expressions

Composing Predicates

```
Predicate<Apple> redApple = (a) -> RED.equals(a.getColor());
Predicate<Apple> redAndHeavyAppleOrGreen =
      redApple.and(apple -> apple.getWeight() > 150)
              .or (apple -> GREEN.equals(a.getColor()));
```

Composing and chaining comparators

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

Composing Functions

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

OVERVIEW OF STREAMS

- Streams are an update to Java API to allow declarative, functional style processing over collections of data

WHAT ARE STREAMS ?

- Streams are an update to Java API to allow declarative, functional style processing over collections of data
- Leads to elegant and concise specification (what instead of how)

WHAT ARE STREAMS ?

- Streams are an update to Java API to allow declarative, functional style processing over collections of data
- Leads to elegant and concise specification (what instead of how)
- Allows composition and flexibility

- Streams are an update to Java API to allow declarative, functional style processing over collections of data
- Leads to elegant and concise specification (what instead of how)
- Allows composition and flexibility
- Parallelizable without any need to write multi-threaded code

The following class will be used extensively in the code snippets

```java
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;

    public String getName() {
        return name;
    }
    public boolean isVegetarian() {
        return vegetarian;
    }
    public int getCalories() {
        return calories;
    }
    public Type getType() {
        return type;
    }
    public enum Type { MEAT, FISH, OTHER }
}
```

Counting words with length greater than 30 from a list of words

- Using iterations

```java
long count = 0;
for (String w : words) {
    if (w.length() > 30)
        count++;
}
```

Counting words with length greater than 30 from a list of words

- Using iterations

```
long count = 0;
for (String w : words) {
    if (w.length() > 30)
        count++;
}
```

- Using streams

```
words.stream().filter(w -> w.length() > 30).count()
```

Counting words with length greater than 30 from a list of words

- Using iterations

```java
long count = 0;
for (String w : words) {
    if (w.length() > 30)
        count++;
}
```

- Using streams

```java
words.stream().filter(w -> w.length() > 30).count()
```

- Parallelizing using streams

```java
words.parallelStream().filter(w -> w.length() > 30)
                      .count()
```

Collections in Java 8 support a new `stream` method that returns a stream (see the interface definition in `java.util.stream.Stream`)

- Streams consist of a sequence of elements

Collections in Java 8 support a new `stream` method that returns a stream (see the interface definition in `java.util.stream.Stream`)

- Streams consist of a sequence of elements
- Streams consume from a data-providing source like collections, arrays, I/O resources etc.

Collections in Java 8 support a new `stream` method that returns a stream (see the interface definition in `java.util.stream.Stream`)

- Streams consist of a sequence of elements
- Streams consume from a data-providing source like collections, arrays, I/O resources etc.
- Streams support data processing operations (commonly seen in databases or functional programming languages) e.g. filter, map, reduce, find, match, sort etc.

CONCEPTUALIZING STREAMS (CONTD.)

- Streams are operated on by methods contained in functional interfaces

CONCEPTUALIZING STREAMS (CONTD.)

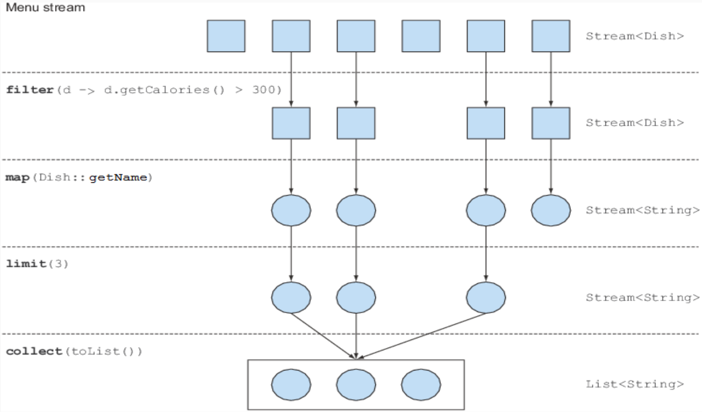
- Streams are operated on by methods contained in functional interfaces
- Streams provide internal iteration where the stream operations do the iterations behind the scene

- Streams are operated on by methods contained in functional interfaces
- Streams provide internal iteration where the stream operations do the iterations behind the scene
- Streams support pipelining where the stream operations return streams allowing chaining of operations

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

VISUALIZING STREAMS

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



Streams in a nutshell (Modern Java in Action)

STREAMS VS COLLECTIONS

Some of the main conceptual differences between streams and collections

- Collections are for storing data, streams are for specifying computations $\rightarrow$ A stream does not store its elements but computes/generates it on demand

Some of the main conceptual differences between streams and collections

- Collections are for storing data, streams are for specifying computations $\rightarrow$ A stream does not store its elements but computes/generates it on demand
- Streams operations do not mutate streams but generate new output streams by applying transformations on the source stream

Some of the main conceptual differences between streams and collections

- Collections are for storing data, streams are for specifying computations $\rightarrow$ A stream does not store its elements but computes/generates it on demand
- Streams operations do not mutate streams but generate new output streams by applying transformations on the source stream
- Stream operations are lazy and hence only executed when the result is needed

Some of the main conceptual differences between streams and collections

- Collections are for storing data, streams are for specifying computations $\rightarrow$ A stream does not store its elements but computes/generates it on demand
- Streams operations do not mutate streams but generate new output streams by applying transformations on the source stream
- Stream operations are lazy and hence only executed when the result is needed
- Streams are traversable only once and the traversal is done internally by the stream operations

STREAMS VS COLLECTIONS

Some of the main conceptual differences between streams and collections

- Collections are for storing data, streams are for specifying computations $\rightarrow$ A stream does not store its elements but computes/generates it on demand
- Streams operations do not mutate streams but generate new output streams by applying transformations on the source stream
- Stream operations are lazy and hence only executed when the result is needed
- Streams are traversable only once and the traversal is done internally by the stream operations
- Once a stream has been reduced to a result, it cannot be reused

In order to work with streams the following three things are needed

1. Stream creation operation (e.g., `.stream()`, `.generate()`, etc.)
2. Intermediate stream transformation operation (e.g., `.filter()`, `.map()`, etc.)
3. Terminal stream consumption operation (e.g., `.reduce()`, `.forEach()`, `.collect()`, etc.)

CREATING STREAMS

In order to transform and consume streams, we need to first create a stream

Streams can be created from different sources e.g., In order to transform and consume streams, we need to first create a stream

Streams can be created from different sources e.g.,

- From `Collections`
- From `Arrays`
- From values
- From functions

stream creation (finite streams)

You can create finite streams from a finite set of values

STREAM CREATION (FINITE STREAMS)

You can create finite streams from a finite set of values

- From `Collections` using `.stream`

```
someList.stream()
        .forEach(System.out::println);
```

STREAM CREATION (FINITE STREAMS)

You can create finite streams from a finite set of values

- From Collections using `.stream`

```
someList.stream()
        .forEach(System.out::println);
```

- From Arrays using `Arrays.Stream`

```
int[] someNumbers = {12, 59, 1, 91, 17};
Arrays.stream(someNumbers)
      .forEach(System.out::println);
```

You can create finite streams from a finite set of values

- From Collections using `.stream`

```
someList.stream()
        .forEach(System.out::println);
```

- From Arrays using `Arrays.Stream`

```
int[] someNumbers = {12, 59, 1, 91, 17};
Arrays.stream(someNumbers)
      .forEach(System.out::println);
```

- From values using `Stream.of`

```
Stream.of("Is", "Java", "cool", "enough", "?")
      .forEach(System.out::println);
```

You can create infinite streams by specifying functions that would generate the values which will be consumed lazily

You can create infinite streams by specifying functions that would generate the values which will be consumed lazily

- Using `Stream.iterate` that takes a `Functional Interface UnaryOperator<T>`

```
Stream.iterate(1, n -> n + 2)
      .forEach(System.out::println);
```

You can create infinite streams by specifying functions that would generate the values which will be consumed lazily

- Using `Stream.iterate` that takes a `Functional Interface UnaryOperator<T>`

```
Stream.iterate(1, n -> n + 2)
      .forEach(System.out::println);
```

- Using `Stream.generate` that takes a `Functional Interface Supplier<T>`

```
Stream.generate(Math::Random)
      .forEach(System.out::println);
```

You can create infinite streams by specifying functions that would generate the values which will be consumed lazily

- Using `Stream.iterate` that takes a `Functional Interface UnaryOperator<T>`

```
Stream.iterate(1, n -> n + 2)
      .forEach(System.out::println);
```

- Using `Stream.generate` that takes a `Functional Interface Supplier<T>`

```
Stream.generate(Math::Random)
      .forEach(System.out::println);
```

`Stream.iterate` is sequential by definition while `Stream.generate` is not

TRANSFORMING STREAMS

STREAM TRANSFORMATION

- Once a stream is created, it can be transformed into other streams
- Operations that transform streams into another stream are called intermediate operations

- Once a stream is created, it can be transformed into other streams
- Operations that transform streams into another stream are called intermediate operations
- Transformation operations can compose and decide the optimal way to process the operation pipeline
- Categories of transformation operations

- Once a stream is created, it can be transformed into other streams
- Operations that transform streams into another stream are called intermediate operations
- Transformation operations can compose and decide the optimal way to process the operation pipeline
- Categories of transformation operations
    1. Filtering
    2. Slicing
    3. Sorting
    4. Mapping

The `Stream` interface supports the filter method that takes a `Functional Interface Predicate<T>` (function returning a boolean) and returns a stream with elements satisfying the predicate

The `Stream` interface supports the filter method that takes a `Functional Interface Predicate<T>` (function returning a boolean) and returns a stream with elements satisfying the predicate

1. Filtering with a predicate using `filter`

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

The `Stream` interface supports the filter method that takes a `Functional Interface Predicate<T>` (function returning a boolean) and returns a stream with elements satisfying the predicate

1. Filtering with a predicate using `filter`

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

2. Filtering Unique Elements using `distinct`

```
List<Integer> numbers = Arrays.asList(2,7,3,5,3,1,2,91);
           numbers.stream()
                  .filter(i -> i % 2 != 0)
                  .distinct()
                  .forEach(System.out::println);
```

Selecting or dropping elements with a predicate

Selecting or dropping elements with a predicate

- Using `filter` to process the entire stream can often be improved

Selecting or dropping elements with a predicate

- Using `filter` to process the entire stream can often be improved
- Using `takeWhile` for early termination (since Java 9)

```
List<Dish> filteredMenu =
  specialMenu.stream()
             .takeWhile(dish -> dish.getCalories() < 320)
             .collect(toList());
```

Selecting or dropping elements with a predicate

- Using `filter` to process the entire stream can often be improved
- Using `takeWhile` for early termination (since Java 9)

```
List<Dish> filteredMenu =
  specialMenu.stream()
             .takeWhile(dish -> dish.getCalories() < 320)
             .collect(toList());
```

- Using `dropWhile` for early termination (since Java 9)

```
List<Dish> filteredMenu =
  specialMenu.stream()
             .dropWhile(dish -> dish.getCalories() < 320)
             .collect(toList());
```

TRUNCATING AND SKIPPING STREAMS

- Truncating a stream using `limit(n)`

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

- Truncating a stream using `limit(n)`

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

- Skipping elements in a stream using `skip(n)`

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

The `Stream` interface supports the method `sorted`, which takes a `Functional Interface Comparator<T>` (function to compare two values) as argument and produces a sorted stream using it

Sorting a stream using `sorted`

```
List<Integer> dishNameLengths =
        menu.stream()
            .sorted(comparing(Dish::getCalories)
                                 .andThen(Dish::getName))
            .collect(toList());
```

The `Stream` interface supports the method `map`, which takes a `Functional Interface Function<T,R>` (function) as argument and applies it to each element, thus mapping it into a new element

- Mapping a function using map

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

MAPPING IN A STREAM (CONTD.)

- Use `flatMap` when you need to flatten a stream

```
List<String> uniqueCharacters =
        words.stream()
              .map(word -> word.split(""))
              .flatMap(Arrays::stream)
              .distinct()
              .collect(toList());
```

Stream operations may need to maintain some history (state) for their internal iterations

Stream operations may need to maintain some history (state) for their internal iterations

- Stateless (`filter`, `map`, `flatMap`, `takeWhile`, `dropWhile`)

Stream operations may need to maintain some history (state) for their internal iterations

- Stateless (`filter`, `map`, `flatMap`, `takeWhile`, `dropWhile`)
- Stateful

Stream operations may need to maintain some history (state) for their internal iterations

- Stateless (`filter`, `map`, `flatMap`, `takeWhile`, `dropWhile`)
- Stateful
  - Bounded state (`limit`, `skip`)

Stream operations may need to maintain some history (state) for their internal iterations

- Stateless (`filter`, `map`, `flatMap`, `takeWhile`, `dropWhile`)
- Stateful
    - Bounded state (`limit`, `skip`)
    - Unbounded state (`distinct`, `sorted`)

TERMINATING STREAMS

Stream creation and transformation operations do not consume the stream while termination operations do

Categories of termination operations:

- Finding and Matching
- Reducing
- Iterating
- Collecting

The `Stream` interface supports the `anyMatch`, `allMatch`, `noneMatch` operations that take a `Functional Interface` `Predicate<T>` (function returning a boolean) to match elements in a stream and return a boolean

The `Stream` interface supports the `anyMatch`, `allMatch`, `noneMatch` operations that take a `Functional Interface` `Predicate<T>` (function returning a boolean) to match elements in a stream and return a boolean

- Checking if any element matches the predicate using `anyMatch`

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

- Checking to see if all elements match the predicate match using `allMatch`

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

MATCHING ELEMENTS IN STREAMS (CONTD.)

- Checking to see if all elements match the predicate match using `allMatch`

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

- Checking to see if none of the elements match the predicate match using `noneMatch`

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

MATCHING ELEMENTS IN STREAMS (CONTD.)

- Checking to see if all elements match the predicate match using `allMatch`

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

- Checking to see if none of the elements match the predicate match using `noneMatch`

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

- All these operations take advantage of short circuited evaluation

The `Stream` interface supports the `findAny, findFirst` operations that take a `Functional Interface Predicate<T>` (function returning a boolean) and return an `Optional` if the predicate matches

The `Stream` interface supports the `findAny`, `findFirst` operations that take a `Functional Interface Predicate<T>` (function returning a boolean) and return an `Optional` if the predicate matches

- Finding the first element matching the predicate using `findFirst`

```
    Arrays.asList(9,10,11,12,67,91,52).
          .stream()
          .map(n -> n * n)
          .filter(n -> n % 2 == 0)
          .findFirst();
```

- Finding any element matching the predicate match using `findAny`

```
      menu.stream()
          .filter(Dish::isVegetarian)
          .findAny()
          .ifPresent(dish ->
               System.out.println(dish.getName()));
```

- Finding any element matching the predicate match using `findAny`

```
      menu.stream()
          .filter(Dish::isVegetarian)
          .findAny()
          .ifPresent(dish ->
               System.out.println(dish.getName()));
```

- `Optional<T>` is a container class to represent whether a value exists or not. It has methods `isPresent`, `ifPresent`, `get`, `orElse` which can be used appropriately

The `Stream` interface supports the `reduce` operation that takes an initial `value` and a `Functional Interface` `BinaryOperator<T>` (function combinining two elements to produce a new value) and returns a reduction of the stream

The `Stream` interface supports the `reduce` operation that takes an initial value and a `Functional Interface` `BinaryOperator<T>` (function combinining two elements to produce a new value) and returns a reduction of the stream

- Reducing a stream using `reduce`

```
int product = numbers.stream()
                     .reduce(1, (a, b) -> a * b);
int sum = numbers.stream().reduce(0, Integer::sum);
```

The `Stream` interface supports the reduce operation that takes an initial value and a `Functional Interface` `BinaryOperator<T>` (function combinining two elements to produce a new value) and returns a reduction of the stream

- Reducing a stream using `reduce`

```java
int product = numbers.stream()
                     .reduce(1, (a, b) -> a * b);
int sum = numbers.stream().reduce(0, Integer::sum);
```

- Using an overloaded `reduce` that does not take an initial value

```java
Optional<Integer> sum = numbers.stream()
                               .reduce(Integer::sum);
```

- Computing maximum and minimum in a stream using `reduce`

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

- Computing maximum and minimum in a stream using `reduce`

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

- `reduce` is a stateful bounded operation while the other terminal operations we saw so far are stateless

QUESTION ON REDUCTION

$\boxed{\text{How can we count the number of elements in a stream ?}}$

How can we count the number of elements in a stream ?

- Counting a stream using `map` and `reduce`

```
int count = menu.stream()
                .map(d -> 1)
                .reduce(0, (a, b) -> a + b);
```

How can we count the number of elements in a stream ?

- Counting a stream using map and `reduce`

```
int count = menu.stream()
                .map(d -> 1)
                .reduce(0, (a, b) -> a + b);
```

- Counting a stream using `count`

```
long count = menu.stream().count();
```

The `Stream` interface supports the `forEach` operation that takes an initial value and a `Functional Interface Consumer<T>` (function that produces no value) and consumes the stream

ITERATING OVER STREAMS

The Stream interface supports the forEach operation that takes an initial value and a Functional Interface Consumer<T> (function that produces no value) and consumes the stream

- Printing using forEach

```
menu.stream()
    .map(Dish::getName)
    .forEach(x -> System.out.println(x));

menu.stream()
    .map(Dish::getName)
    .forEach(System.out::println);
```

The `Stream` interface supports the `collect` operation that takes a `Functional Interface Collector<? super T,A,R>` (aggregation function) to perform a mutable reduction on the stream

The `Stream` interface supports the `collect` operation that takes a `Functional Interface Collector<? super T,A,R>` (aggregation function) to perform a mutable reduction on the stream

- A large number of static functions have been defined in `Collectors` which you can use and implement the interface if you need a new one

The `Stream` interface supports the `collect` operation that takes a `Functional Interface Collector<? super T,A,R>` (aggregation function) to perform a mutable reduction on the stream

- A large number of static functions have been defined in `Collectors` which you can use and implement the interface if you need a new one
- Counting using `Collectors.counting`

```
long howManyDishes = menu.stream()
                         .collect(Collectors.counting());
```

- Summing using `Collectors.summingInt`

```
int totalCalories = menu.stream()
                        .collect(summingInt(
                               Dish::getCalories));
```

- Summing using `Collectors.summingInt`

```
int totalCalories = menu.stream()
                        .collect(summingInt(
                                 Dish::getCalories));
```

- Averaging using `Collectors.averagingInt`

```
  double avgCalories = menu.stream()
                        .collect(averagingInt(
                                 Dish::getCalories));
```

- Finding maximum and minimum using `Collectors.maxBy`, `Collectors.minBy`

```
Comparator<Dish> dishCaloriesComparator =
  Comparator.comparingInt(Dish::getCalories);
Optional<Dish> mostCalorieDish =
  menu.stream().collect(maxBy(dishCaloriesComparator));
Optional<Dish> leastCalorieDish =
  menu.stream().collect(minBy(dishCaloriesComparator));
```

- Finding maximum and minimum using `Collectors.maxBy`, `Collectors.minBy`

```
Comparator<Dish> dishCaloriesComparator =
  Comparator.comparingInt(Dish::getCalories);
Optional<Dish> mostCalorieDish =
  menu.stream().collect(maxBy(dishCaloriesComparator));
Optional<Dish> leastCalorieDish =
  menu.stream().collect(minBy(dishCaloriesComparator));
```

- Summarizing using `Collectors.summarizingInt`

```
IntSummaryStatistics menuStatistics =
    menu.stream().collect(summarizingInt(
                              Dish::getCalories));
```

COLLECTING FROM STREAMS (CONTD.)

- Joining strings using `Collectors.joining`

```
String shortMenu =
  menu.stream().map(Dish::getName).collect(joining(", "));
```

COLLECTING FROM STREAMS (CONTD.)

- Joining strings using `Collectors.joining`

```
String shortMenu =
  menu.stream().map(Dish::getName).collect(joining(", "));
```

- Grouping elements using `Collectors.groupingBy`

```
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;} ));
```

- Joining strings using `Collectors.joining`

```
String shortMenu =
  menu.stream().map(Dish::getName).collect(joining(", "));
```

- Grouping elements using `Collectors.groupingBy`

```
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;} ));
```

- Grouping is a very versatile operation and the groups can be manipulated by passing other collectors to the overloaded `groupingBy` operation

The `Stream` interface supports `parallel`, `sequential` operations to convert sequential streams to parallel and vice-versa

- Creating a parallel stream from `Collection` using `parallelStream`

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

PARALLELIZING STREAMS (CONTD.)

- Making a stream parallel using `parallel`

```
long sum = Stream.iterate(1L, i -> i + 1)
                 .limit(n)
                 .parallel()
                 .reduce(0L, Long::sum);
```

- Making a stream sequential using `sequential`

```
    aStream.parallel()
           .filter(...)
           .sequential()
           .map(...)
           .parallel()
           .reduce();
```

PARALLELIZING STREAMS (CONTD.)

- Last usage of sequential/parallel in the pipeline is applied to all operations.
- In practice hard to get predictable performance, has its trade-offs