Bài 12

Streams API

Agenda

- 1. Coping with changing requirements
- 2. Lambda expressions
  - Functional Interface
  - Method references
- 3. Introducing streams
  - What are streams
  - Streams vs collections
  - Stream operation
- 4. Working with streams
  - Filtering and slicing
  - Mapping
  - Finding and matching
  - Reducing
  - Numeric streams

Agenda

- 5. Collecting data with streams
  - Collectors in a nutshell
  - Reducing and summarizing
  - Grouping
  - Partitioning
  - The Collector interface
- 6. Using Optional as a better alternative to null
  - How do you model the absence of a value
  - Optional class
  - Patterns for adopting Optional

1. Coping with changing requirements

- Writing code that can cope with changing requirements is difficult
- For example, imagine an application to help a farmer understand his inventory. App will have the behaviors below:
- 1st : find all green apples in his inventory
- 2nd: find all red apples in his inventory
- 3rd: find all apples heavier than 150 g
- 4th: find all apples that are green and heavier than 150 g

1. Coping with changing requirements

- 1st : find all <u>green apples</u> in his inventory

```java
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;
}
```

- 2nd: find all <u>red apples</u> in his inventory

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

1. Coping with changing requirements

- Attempt: parameterizing the color

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

- You can now make the famer happy and call your method as follows
  - List<Apple> **greenApples** = filterApplesByColor(inventory, "**green**");
  - List<Apple> **redApples** = filterApplesByColor(inventory, "**red**");

1. Coping with changing requirements

- 3rd: find all apples heavier than 150 g

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory, int weight) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory) {
        if (apple.getWeight() > weight) {
            result.add(apple);
        }
    }
    return result;
}
```

- 4th: find all apples that are green and heavier than 150 g

```
public static List<Apple> filterApplesByColorAndWeight(List<Apple> inventory,
        String color, int weight) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory) {
        if (apple.getColor().equals(color) &&  apple.getWeight() > weight) {
            result.add(apple);
        }
    }
    return result;
}
```

1. Coping with changing requirements

- This solution is extremely bad.
- In addition, this solution doesn’t cope well with changing requirements. What if the farmer asks you to filter with different attributes of an apple, for example, its size, its shape, its origin, and soon?
- Furthermore, what if the farmer asks you for more complicated queries that combine attributes, such as green apples that are also heavy? You’d either have multiple duplicated filter methods or one giant, very complex method

1. Coping with changing requirements

- Behavior parameterization

```
apple.getWeight() > weight
apple.getColor().equals(color)
"green".equals(apple.getColor())
"red".equals(apple.getColor())
```

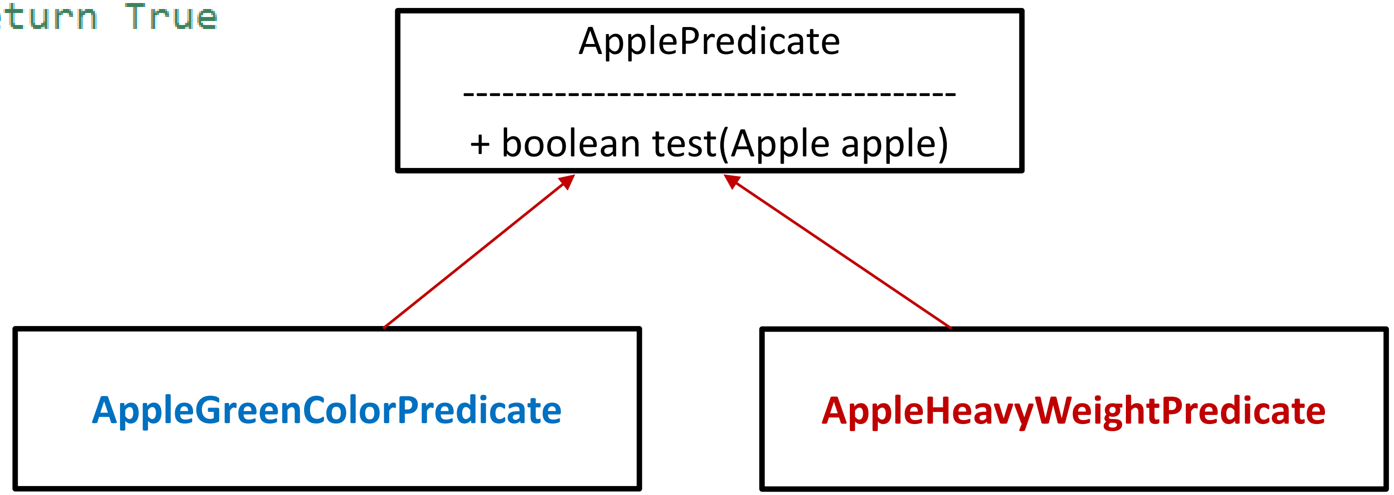
```
public static List<Apple> filterApples(List<Apple> inventory, ... behavior) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory) {
        if (behavior(apple)) { // return True
            result.add(apple);
        }
    }
    return result;
}
```

1. Coping with changing requirements

```
"green".equals(apple.getColor())
"red".equals(apple.getColor())
```

- Behavior parameterization

```java
public static List<Apple> filterApples(List<Apple> inventory, ... behavior) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory) {
        if (behavior(apple)) { // return True
            result.add(apple);
        }
    }
    return result;
}
```



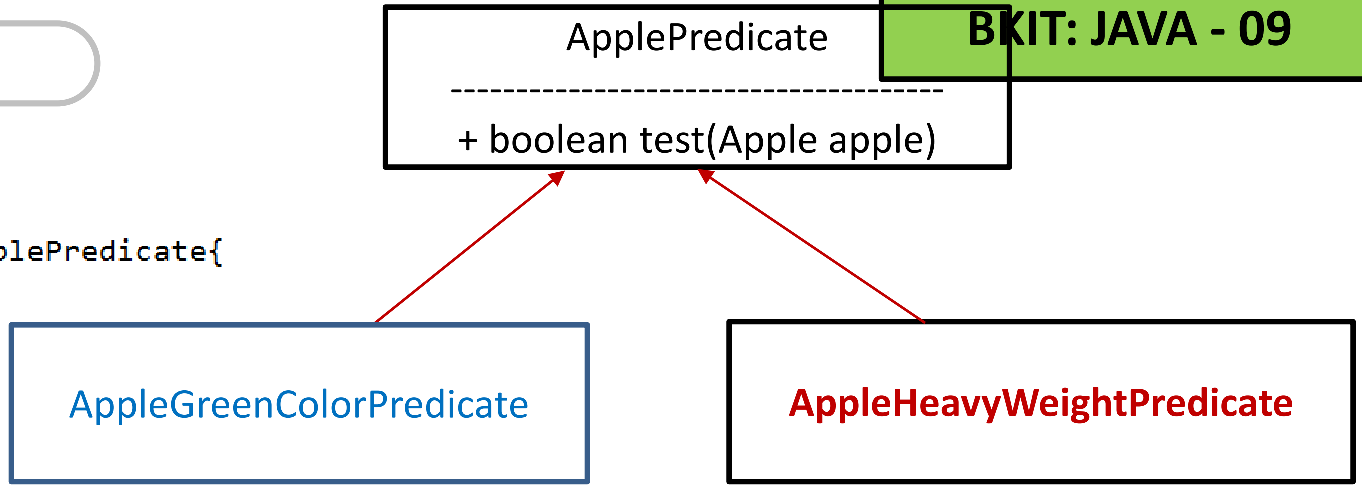
- Solution

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

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

1. Coping with changing requirements

- Behavior parameterization




```java
public class AppleGreenColorPredicate implements ApplePredicate{
    @Override // Select only green apples
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
}
```

2

```java
public class AppleHeavyWeightPredicate implements ApplePredicate{
    @Override // Select only heavy apples
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
```

3

```java
// main method
List<Apple> greenApples = filterApples(inventory, new AppleGreenColorPredicate());
List<Apple> heavyApples = filterApples(inventory, new AppleGreenColorPredicate());
```

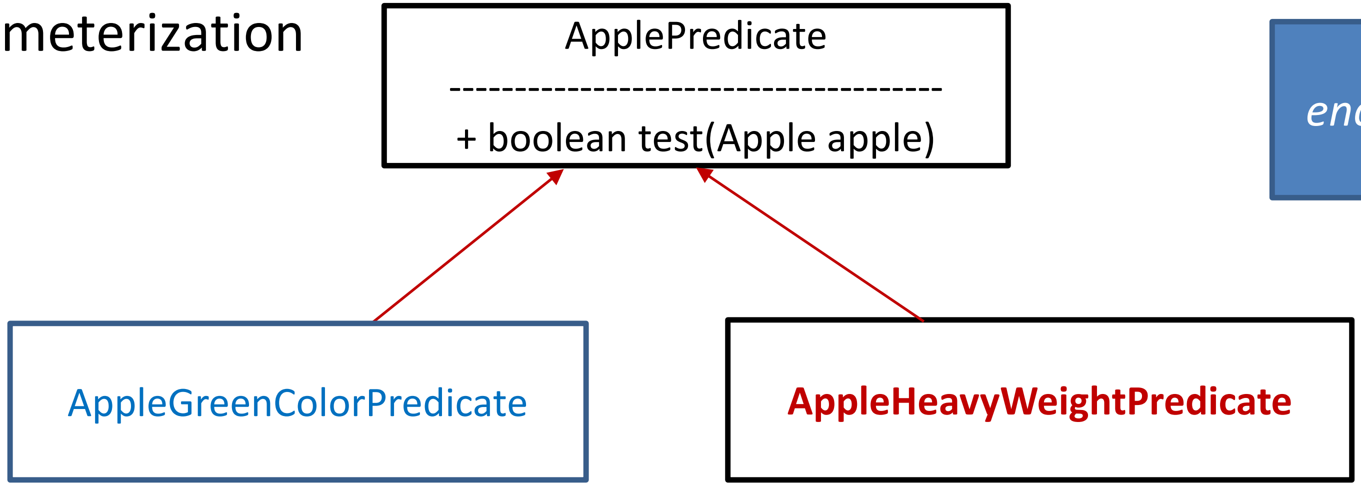
```java
public interface ApplePredicate {
    boolean test(Apple apple);
}
```

1

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

1. Coping with changing requirements

- Behavior parameterization



ApplePredicate encapsulates a strategy for selecting apple

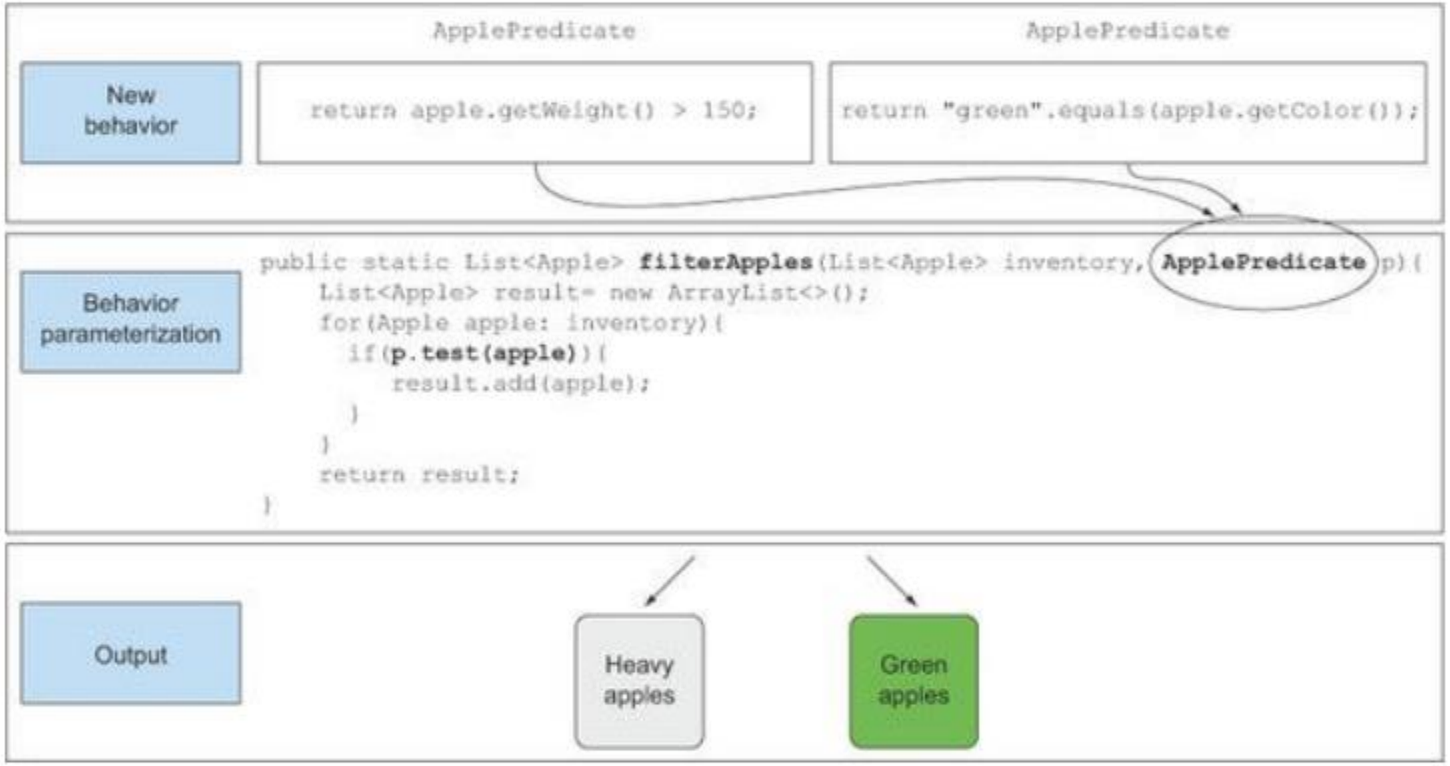
- What you just did is related to the strategy design pattern, which lets you define a family of algorithms, encapsulate each algorithm (called a strategy), and select an algorithm at **run-time**. In this case the family of algorithms is ApplePredicate and the different strategies are AppleHeavyWeightPredicate and AppleGreenColorPredicate.

- See https://en.wikipedia.org/wiki/Strategy_pattern

1. Coping with changing requirements

- Behavior parameterization

Figure 2.3. Parameterizing the behavior of filterApples and passing different filter strategies



1. Coping with changing requirements

- **Anonymous classes** are like the local classes (a class defined in a block) that you're already familiar with in Java. But anonymous classes don't have a name. They allow you to declare and instantiate a class at the same time. In other words, they allow you to create ad hoc implementations.

```java
public class AppleGreenColorPredicate implements ApplePredicate{
    @Override // Select only green apples
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
}

public class AppleHeavyWeightPredicate implements ApplePredicate{
    @Override // Select only heavy apples
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
```

```java
// anonymous class
List<Apple> greenApples = filterApples(inventory, new ApplePredicate() {
    @Override // Select only green apples
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
});
```

```java
// main method
List<Apple> greenApples = filterApples(inventory, new AppleGreenColorPredicate());
List<Apple> heavyApples = filterApples(inventory, new AppleGreenColorPredicate());
```

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

1. Coping with changing requirements

- **Anonymous classes**

```
// anonymous class
List<Apple> greenApples = filterApples(inventory, new ApplePredicate() {
    @Override // Select only green apples
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
});

List<Apple> heavyApples = filterApples(inventory, new ApplePredicate() {
    @Override // Select only heavy apples
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
});
```

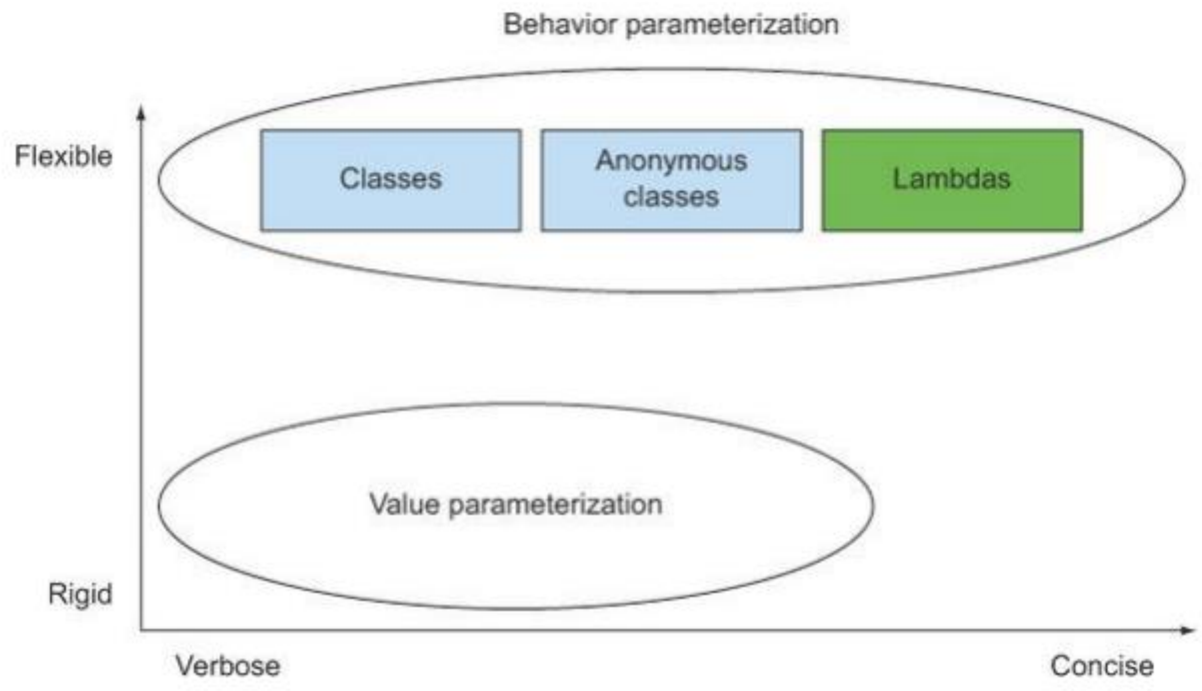
- But anonymous classes are still not good enough, they take a lot of space

1. Coping with changing requirements

- **Anonymous functions:** Lambda expression
- The previous code can be rewritten as follows in Java 8 using Lambda expression

```
// Anonymous functions
List<Apple> greenApples = filterApples(inventory,
        (Apple apple) -> "green".equals(apple.getColor()));
```

Figure 2.4. Behavior parameterization vs. value parameterization



2. Lambda expression

- In the previous chapter, you saw that passing code with behavior parameterization is useful for coping with frequent requirement changes in your code

- But you saw that using anonymous classes to represent different behaviors is unsatisfying: it's verbose, which doesn't encourage programmers to use behavior parameterization in practice. In this chapter, we teach you about a new feature in Java 8 that tackles this problem: **lambda expressions**

```
// Anonymous functions
List<Apple> greenApples = filterApples(inventory,
        (Apple apple) -> "green".equals(apple.getColor()));
```

2. Lambda expression

- A lambda expression can be understood as a concise representation of an **anonymous function** that can be **passed around**: 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

2. Lambda expression

- Anonymous: it doesn't have an explicit name like a method would normally have: less to write and think about!
- Function— A lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.
- Passed around— A lambda expression can be passed as argument to a method or stored in a variable.
- Concise— You don't need to write a lot of boilerplate like you do for anonymous classes. Will be replaced for an instance of **<u>functional interface</u>**

```
// anonymous class
ApplePredicate appleGreenPredicate = new ApplePredicate() {
    @Override // Select only green apples
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
};
```

(parameters) -> { body }

```
// anonymous function: lambda expression
ApplePredicate appleGreenPredicate = (Apple apple) -> {
    return "green".equals(apple.getColor());
};
```

2. Lambda expression

(parameters) -> { body }

- Valid lambda expressions in Java 8

```
Function<String, Integer> function = (String s) -> {
    return s.length();
};
```

```
Function<String, Integer> function = (String s) -> s.length();
Predicate<String> predicate = (String s) -> s.startsWith("Adam");
Runnable runnable = () -> {};
Comparator<String> comparator = (String s1, String s2) -> s1.compareTo(s2);
```

- (parameters) -> expression
- or (note the curly braces for statements)
- (parameters) -> { statements; }

2. Lambda expression

- Valid lambda expressions in Java 8

Quiz 3.1: Lambda syntax

Based on the syntax rules just shown, which of the following are not valid lambda expressions?

1. `() -> {}`
2. `() -> "Raoul"`
3. `() -> {return "Mario";}`
4. `(Integer i) -> return "Alan" + i;`
5. `(String s) -> {"Iron Man";}`

2. Lambda expression

- **Where and how to use lambda expressions**
- You may now be wondering where you're allowed to use lambda expressions. In the previous example, you assigned a lambda to a variable of type Comparator<Apple>. You could also use another lambda with the filter method you implemented in the previous chapter

```
// Anonymous functions
List<Apple> greenApples = filterApples(inventory,
        (Apple apple) -> "green".equals(apple.getColor()));
```

- So where exactly can you use lambdas? **You can use a lambda expression in the context of a functional interface**. In the code shown here, you can pass a lambda as second argument to the method filter because it expects a Predicate<T>, which is a functional interface

2. Lambda expression

- **Functional interface**
- Remember the interface ApplePredicate | Predicate<T> you created, so you could parameterize the behavior of the filter apples method. It’s a functional interface
- Why ? Because Predicate specifies only one abstract method

```java
 * @since 1.8
 */
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate or
     *
     * @param t the input argument
     * @return {@code true} if the
     * otherwise {@code false}
     */
    boolean test(T t);

    /**
```

```java
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();
}

public interface PrivilegedAction<V>{
    T run();
}
```

java.util.Comparator

java.lang.Runnable

java.awt.event.ActionListener

java.util.concurrent.Callable

java.security.PrivilegedAction

- In a nutshell, a functional interface is an interface that specifies exactly one abstract method

2. Lambda expression

- **Functional interface**

Quiz 3.2: Functional interface

Which of these interfaces are functional interfaces?

```
public interface Adder{
    int add(int a, int b);
}
public interface SmartAdder extends Adder{
    int add(double a, double b);
}
public interface Nothing{
}
```

2. Lambda expression

- **Functional interface**
- What can we do with functional interface ?
- Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and treat the whole expression as an instance of a functional interface
- You can achieve the same thing with an anonymous inner class

```java
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing inter
     * to create a thread, starting the
     * <code>run</code> method to be cal
     * thread.
     * <p>
     * The general contract of the metho
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

```java
Runnable r1 = () -> System.out.println("Hello World 1");

Runnable r2 = new Runnable(){
    public void run(){
        System.out.println("Hello World 2");
    }
};

public static void process(Runnable r){
    r.run();
}
process(r1);
process(r2);
process(() -> System.out.println("Hello World 3"));
```

Using a lambda

Using an anonymous class

Prints "Hello World 1"

Prints "Hello World 2"

Prints "Hello World 3" with a lambda passed directly

2. Lambda expression

- **Function descriptor**
- The signature of the abstract method of the functional interface essentially describes the signature of the lambda expression. We call this abstract method a function descriptor

Functional interface	Function descriptor	Primitive 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
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>

```java
 * @since 1.8
 */
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on
     *
     * @param t the input argument
     * @return {@code true} if the
     * otherwise {@code false}
     */
    boolean test(T t);

    /**
```

2. Lambda expression

- **Function descriptor**

Quiz 3.3: Where can you use lambdas?

Which of the following are valid uses of lambda expressions?

1.

```
execute(() -> {});
public void execute(Runnable r){
r.run();
}
```

2.

```
public Callable<String> fetch() {
return () -> "Tricky example ;-)";
}
```

3. `Predicate<Apple> p = (Apple a) -> a.getWeight();`

2. Lambda expression

- **Using local variable**
- All the lambda expressions we've shown so far used only their arguments inside their body. But lambda expressions are also allowed to use free variables (variables that aren't the parameters and defined in an outer scope) just like anonymous classes can.

```
private static Runnable run(int time) {
    String running = "running ...";
    return () -> {
        String student = "Adam";
        System.out.println(student + " is " + running + " in " + time + "(s)");
    };
}
```

2. Lambda expression

- **Using local variable**
- Restrictions: Lambdas are allowed to capture instance and static variables without restrictions. But local variables have to be explicitly declared final.

```
private static Runnable run(int time) {
    // closure
    String running = "running ...";
    Apple apple = new Apple();
    return () -> {
        String student = "Adam";
        System.out.println(student + " is " + running + " in " + time + "(s)");
        apple.setColor("blue");
        running = " stopped ..."; // Local variable 'running, time' defined
        time = 10; // in a enclosing scope must be final
        apple = null;
    };
}
```

- Why can we set a value for the instance variable ‘apple’. But we cannot deallocated it or assign the instance variable to another reference ?

2. Lambda expression

- **Using local variable**
- Why can we set a value for the instance variable 'apple'. But we cannot deallocated it or assign the instance variable to another reference ?

```
private static Runnable run(int time) {
    // closure
    String running = "running ...";
    Apple apple = new Apple();
    return () -> {
        String student = "Adam";
        System.out.println(student + " is " + running + " in " + time + "(s)");
        apple.setColor("blue");
        running = " stopped ..."; // Local variable 'running, time' defined
        time = 10; // in a enclosing scope must be final
        apple = null;
    };
}
```

- Instance variables are stored on the heap, whereas local variables live on the stack. If a lambda could access the local variable (ref) directly and the lambda were used in a thread, then the thread using the lambda could try to access the variable **after** the thread that allocated the variable had deallocated it. Hence, **Java implements access to a free local variable as access to a copy of it rather than access to the original variable**. This makes no difference if the local variable is assigned to only once—hence the restriction

2. Lambda expression

- <u>Using local variable</u>

```java
private static Runnable run(int time) {
    // closure
    String running = "running ...";
    Apple apple = new Apple("green", 50);
    // Expected: update apple color value
    Runnable runnable = () -> {
        String student = "Adam";
        System.out.println(student + " is " + running + " in " + time + "(s)");
        apple.setColor("blue");
        running = " stopped ..."; // Local variable 'running, time' defined
        time = 10; // in a enclosing scope must be final
        apple = null;
    };
    Thread thread1 = new Thread(runnable); // apple = new Apple('pink', 99)
    Thread thread2 = new Thread(runnable); // apple.setColor('black')
    thread1.start();
    thread2.start();
    System.out.println("Color: " + apple.getColor());
    return runnable;
}
```

<u>Expected:</u> After a local variable was called in a lambda expression. The value of instance variable could be changed. But the reference should not.

2. Lambda expression

- **Closure:**
- You may have heard of the term closure and may be wondering whether lambdas meet the definition of a closure (not to be confused with the Clojure programming language). To put it scientifically, a closure is an instance of a function that can reference nonlocal variables of that function with no restrictions. 1

- For example, a closure could be passed as argument to another function. It could also access and modify variables defined outside its scope. 2 Now Java 8 lambdas and anonymous classes do something similar to closures: they can be passed as argument to methods and can access variables outside their scope. But they have a restriction: they can't modify the content of local variables of a method in which the lambda is defined. Those variables have to be implicitly final. It helps to think that lambdas close over values rather than variables

2. Lambda expression

- **Closure:**

1

```
public static void main(String[] args) {
    Runnable runnable = run(20);
    runnable.run();
}

private static Runnable run(int time) {
    // closure
    String running = "running ..."; // outer scope
    return () -> {
        // inner (local) scope
        String student = "Adam";
        System.out.println(student + " is " + running + " in " + time + "(s)");
    };
}
```

Closure

2

```
public static void main(String[] args) {
    int lenth = run(20, (String s) -> s.length());
    System.out.println("length: " + lenth);
}

private static int run(int time, Function<String, Integer> function) {
    String text = "Welcome to Java class";
    return function.apply(text);
}
```

2. Lambda expression

- **Method reference:**
- Let you reuse existing method definitions and pass them just like lambdas. In some cases they appear more readable and feel more natural than using lambda expressions.
- Here's our example written with a method reference and a bit of help from the updated

```
greenApples.forEach((Apple apple) -> System.out.println(apple));
greenApples.forEach(System.out::println);

greenApples.sort((Apple a1, Apple a2) -> a1.getWeight() - a2.getWeight());
greenApples.sort(Comparator.comparing(Apple::getWeight));
```

- Method references can be seen as shorthand for lambdas calling only a specific method. The basic idea is that if a lambda represents "call this method directly," it's best to refer to the method by name rather than by a description of how to call it.
- Indeed, a method reference lets you create a lambda expression from an existing method implementation

2. Lambda expression

- **Method reference:**
- When you need a method reference, the target reference is placed before the delimiter :: and the name of the method is provided after it.
- For example, Apple::getWeight is a method reference to the method getWeight defined in the Apple class. Remember that no brackets are needed because you're not actually calling the method. The method reference is shorthand for the lambda expression (Apple a) -> a.getWeight().

Lambda	Method reference equivalent
(Apple a) -> a.getWeight()	Apple::getWeight
() -> Thread.currentThread().dumpStack()	Thread.currentThread()::dumpStack
(str, i) -> str.substring(i)	String::substring
(String s) -> System.out.println(s)	System.out::println

2. Lambda expression

- **Method reference:**

Recipe for constructing method references

There are three main kinds of method references:

1. A method reference to a *static method* (for example, the method parseInt of Integer, written Integer::parseInt)

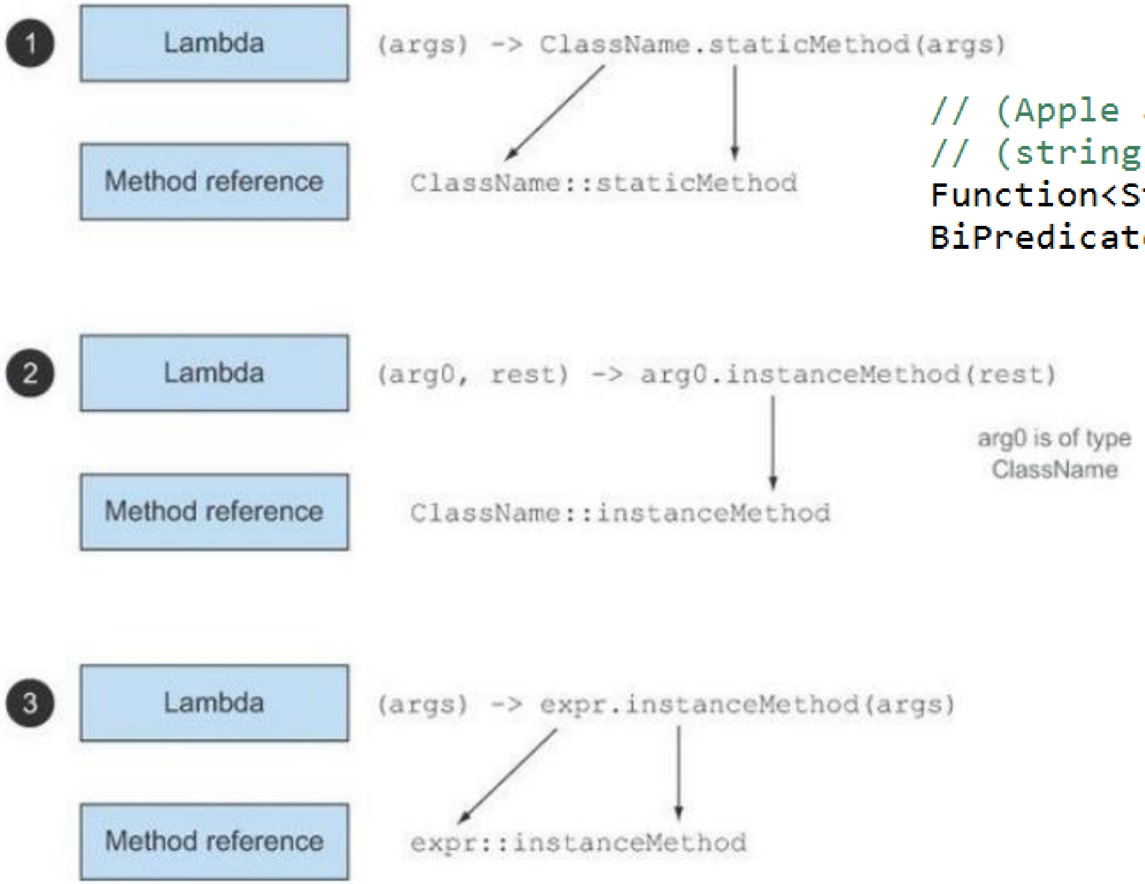
2. A method reference to an *instance method of an arbitrary type* (for example, the method length of a String, written String::length)

3. A method reference to an *instance method of an existing object* (for example, suppose you have a local variable expensiveTransaction that holds an object of type Transaction, which supports an instance method getValue; you can write expensiveTransaction::getValue)

2. Lambda expression

- **Method reference:**

Figure 3.5. Recipes for constructing method references for three different types of lambda expressions




```java
// (Apple apple) -> apple.getWeight() => Apple::getWeight
// (string, rest) -> string.subString(rest) => String::subString
Function<String, Integer> stringToIntegers = Integer::parseInt;
BiPredicate<List<String>, String> contains = List::contains;
```

2. Lambda expression

- **Constructor reference**:
- You can create a reference to an existing constructor using its name and the keyword new as follows: ClassName::new. It works similarly to a reference to a static method

```
Supplier<Apple> c1 = Apple::new;
Apple a1 = c1.get();
```

A constructor reference to the default `Apple()` constructor.

Calling `Supplier`'s `get` method will produce a new `Apple`.

which is equivalent to

```
Supplier<Apple> c1 = () -> new Apple();
Apple a1 = c1.get();
```

A lambda expression creating an `Apple` with the default constructor.

Calling `Supplier`'s `get` method will produce a new `Apple`.

If you have a constructor with signature Apple(Integer weight), it fits the signature of the Function interface, so you can do this,

```
Function<Integer, Apple> c2 = Apple::new;
Apple a2 = c2.apply(110);
```

A constructor reference to `Apple(Integer weight)`.

Calling the `Function`'s `apply` method with the requested weight will produce an `Apple`.

2. Lambda expression

- **Constructor reference:**
- Hers is our example about the sorting a map and return LinkedHashMap

```
Map<String, Integer> sortedMap = budget.entrySet()
    .stream()
    .sorted(Entry.comparingByValue())
    .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
```

Which is equivalent to

```
Supplier<LinkedHashMap<String, Integer>> lkhMap = new Supplier<LinkedHashMap<String,Integer>>() {
    @Override
    public LinkedHashMap<String, Integer> get() {
        return new LinkedHashMap<String, Integer>();
    }
};

Map<String, Integer> sortedMap = budget.entrySet()
    .stream()
    .sorted(Entry.comparingByValue())
    .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
```

2. Lambda expression

- **Constructor reference:**
- Examples:

In the following code, each element of a List of Integers is passed to the constructor of Apple using a similar map method we defined earlier, resulting in a List of apples with different weights:

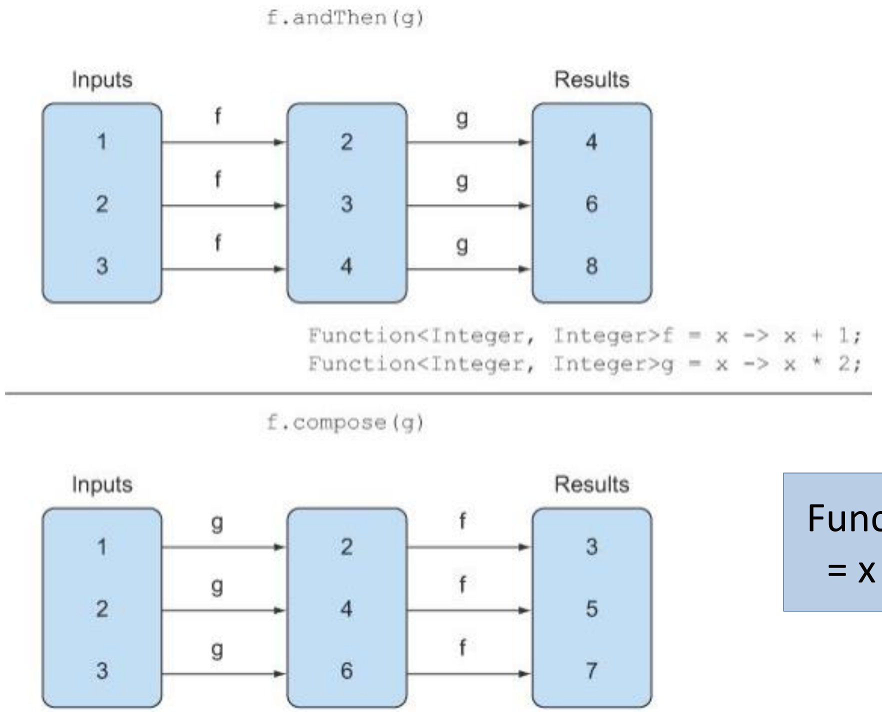
```
List<Integer> weights = Arrays.asList(7, 3, 4, 10);
List<Apple> apples = map(weights, Apple::new);
public static List<Apple> map(List<Integer> list,
                              Function<Integer, Apple> f){
    List<Apple> result = new ArrayList<>();
    for(Integer e: list){
        result.add(f.apply(e));
    }
    return result;
}
```

Passing a constructor reference to the map method

2. Lambda expression

▪ Useful methods to compose lambda expressions

- Several functional interfaces in the Java 8 API contain convenient methods. Specifically, many functional interfaces such as Comparator, Function, and Predicate that are used to pass lambda expressions provide methods that allow composition. What does this mean? In practice it means you can combine several simple lambda expressions to build more complicated ones.
- For example, you can combine two predicates into a larger predicate that performs an or operation between the two predicates. Moreover, you can also compose functions such that the result of one becomes the input of another function
- **Chaining Comparators**
- **Composing Predicates**
- **Composing Functions**



Function<Integer, Integer> r = x -> (x+1)*2 || (x*2) + 1

2. Lambda expression

- **Chaining Comparators**
- When two apples have the same weight. Which apple should have priority in the sorted list? You may want to provide a second Comparator to further refine the comparision.
- For example: after two apples are compared based on their weight, you may want to sort them by country of origin. The **thenComparing** method allows you to do just that

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

Sorting by decreasing weight

Sorting further by country when two apples have same weight

2. Lambda expression

- **Composing Predicates**
- The Predicate interface includes three methods that let you **reuse an existing Predicate** to create more complicated ones: negate, and, and or. For example, you can use the method negate to return the negation of a Predicate, such as an apple **that is not red**

```
Predicate<Apple> redApple = (Apple a) -> "red".equals(a.getColor());

Predicate<Apple> notRedApple = (Apple a) -> !"red".equals(a.getColor());
Predicate<Apple> notRedApple = redApple.negate();
```

- You may want to combine two lambdas to say that an apple is **both red and green or heavy** with the and, or method

```
Predicate<Apple> predicate = redApple
        .and(a -> a.getWeight() > 150)
        .or(a -> "green".equals(a.getColor()));
```

2. Lambda expression

- **Composing Functions**
- The Function interface comes with two default methods for this, **andThen** and **compose**, which both return an instance of Function.

- The method andThen returns a function that first applies a given function to an input and then applies another function to the result of that application. For example, given a function f that increments a number (x -> x + 1) and another function g that multiples a number by 2, you can combine them to create a function h that first increments a number and then multiplies the result by 2:

```
Function<Integer, Integer> f = x -> x + 2;
Function<Integer, Integer> g = x -> x * 3;
Function<Integer, Integer> r = f.andThen(g);
int value = r.apply(2);
```

Math: r = g (f(x))

This returns 12

2. Lambda expression

- **Composing Functions**
- The Function interface comes with two default methods for this, **andThen** and **compose**, which both return an instance of Function.

- You can also use the method compose similarly to first apply the function given as argument to compose and then apply the function to the result. For example, in the previous example using compose, it would mean f(g(x)) instead of g(f(x)) using andThen:

```
Function<Integer, Integer> f = x -> x + 2;
Function<Integer, Integer> g = x -> x * 3;
Function<Integer, Integer> r = f.compose(g);
int value = r.apply(2);
```

Math: r = f (g(x))

This returns 8

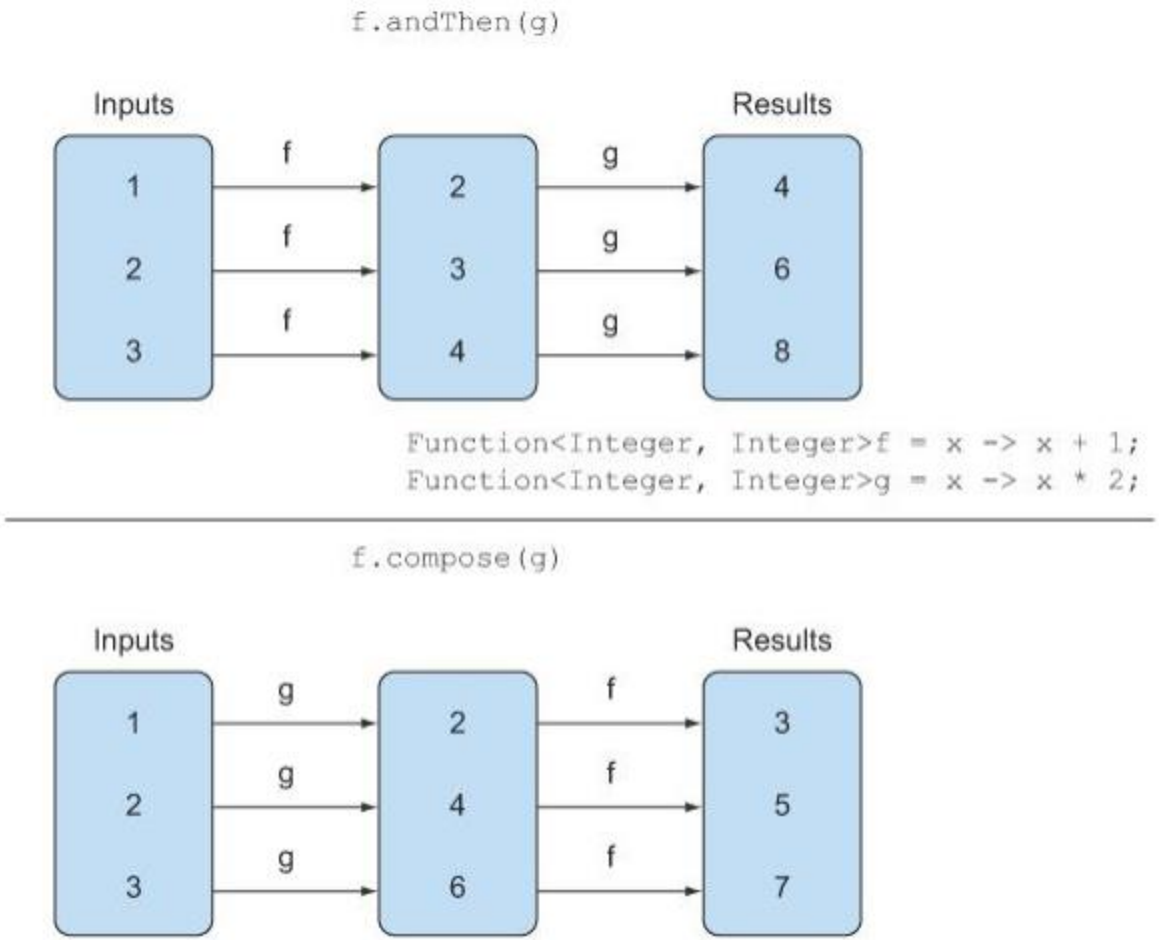
```
Function<Student, Grade> f1 = Student::getGrade;
Function<Grade, String> f2 = Grade::getId;
Function<Student, String> f3 = f1.andThen(f2);

students.sort(comparing(f3));
```

2. Lambda expression

- **Composing Functions**

Figure 3.6. Using andThen vs. compose



2. Lambda expression

▪ Summary

- A *lambda expression* can be understood as a kind of 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.
- Lambda expressions let you pass code concisely.
- A *functional interface* is an interface that declares exactly one abstract method.
- Lambda expressions can be used only where a functional interface is expected.
- Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and *treat the whole expression as an instance of a functional interface*.
- Java 8 comes with a list of common functional interfaces in the java.util .function package, which includes Predicate<T>, Function<T, R>, Supplier<T>, Consumer<T>, and BinaryOperator<T>, described in table 3.2.
- There are primitive specializations of common generic functional interfaces such as Predicate<T> and Function<T, R> that can be used to avoid boxing operations: IntPredicate, IntToLongFunction, and so on.
- The execute around pattern (that is, you need to execute a bit of behavior in the middle of code that's always required in a method, for example, resource allocation and cleanup) can be used with lambdas to gain additional flexibility and reusability.
- The type expected for a lambda expression is called the *target* type.
- Method references let you reuse an existing method implementation and pass it around directly.
- Functional interfaces such as Comparator, Predicate, and Function have several default methods that can be used to combine lambda expressions.

3. Introducing Streams

- What is a stream?
- Collections vs streams
- Internal vs external iteration
- Intermediate vs terminal operation

3. Introducing Streams

- **What is a stream?**
- Streams are an update to the Java API that lets you manipulate collections of data in a declarative way (you express a query rather than code an ad hoc implementation for it). For now, you can think of them as fancy iterators over a collection of data.

3. Introducing Streams

- **For example**: Lets select the names of dishes that are low in calories (less than 300) and sorting by calories

in-memory

```
private static List<Dish> mock() {
    // menu
    Dish d1 = new Dish("D01", true, 200, Type.OTHER);
    Dish d2 = new Dish("D02", false, 220, Type.MEAT);
    Dish d3 = new Dish("D03", false, 280, Type.MEAT);
    Dish d4 = new Dish("D04", false, 380, Type.FISH);
    return Arrays.asList(d1,d2,d3,d4);
}

// Before (Java07)
List<Dish> lowCaloricDishes = new ArrayList<>();
for (Dish d: menu) {
    if (d.getCalories() < 400) {
        lowCaloricDishes.add(d);
    }
}

// sorting by calories
menu.sort(new Comparator<Dish>() {
    @Override
    public int compare(Dish d1, Dish d2) {
        return d1.getCalories() - d2.getCalories();
    }
});

// get name of dish
List<String> lowCaloricDishesName = new ArrayList<>();
for (Dish d: lowCaloricDishes) {
    lowCaloricDishesName.add(d.getName());
}
```

How to implement

database

Table: dishes

ID	NAME	CALORIES
1	D01	200
2	D02	220
3	D03	280
4	D04	380

SELECT name
FROM dishes
WHERE calories < 300
ORDER BY CALORIES **ASC**

What you expected

3. Introducing Streams

- **For example**: Lets select the names of dishes that are low in calories (less then 400) and sorting by calories

in-memory

```java
private static List<Dish> mock() {
    // menu
    Dish d1 = new Dish("D01", true, 200, Type.OTHER);
    Dish d2 = new Dish("D02", false, 220, Type.MEAT);
    Dish d3 = new Dish("D03", false, 280, Type.MEAT);
    Dish d4 = new Dish("D04", false, 380, Type.FISH);
    return Arrays.asList(d1,d2,d3,d4);
}

// After (Java08)
List<String> lowCaloricDishesName = menu.stream()
        .filter(d -> d.getCalories() < 400)
        .sorted(Comparator.comparing(Dish::getCalories))
        .map(Dish::getName)
        .collect(Collectors.toList());
```

Database-like operations than programing language

What you expected

database

Table: dishes

NAME	VEGETERIAN	CALORIES
D01	1	200
D02	0	220
D03	0	280
D04	0	380

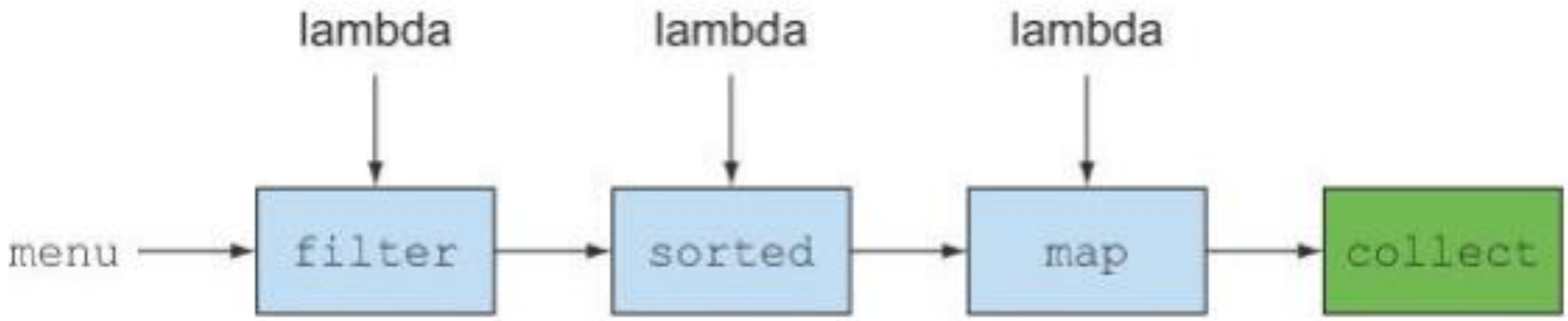
SELECT name
FROM dishes
WHERE calories < 400
ORDER BY CALORIES **ASC**

What you expected

3. Introducing Streams

- Operations such as filter (or sorted, map, and collect) are available in Stream as high level building blocks

Figure 4.1. Chaining stream operations forming a stream pipeline



3. Introducing Streams

- To summarize, the Streams API in Java 8 lets you write code that’s
  - Declarative: More concise and readable
  - Compassable: Great flexibility
  - Parallelizable: Better performance

3. Introducing Streams

- **Getting started with streams**
- We start our discussion of streams with collections, because that's the simplest way to begin working with streams.
- **Collections in Java 8 support a new stream method that returns a stream** (the interface definition is available in java.util.stream.Stream).
- You'll later see that you can also get streams in various other ways (for example, generating stream elements from a numeric range or from I/O resources).
- **Stream is "a sequence of elements from a source that supports data processing operations"**

3. Introducing Streams

- **Stream is “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. Because collections are data structures, they’re mostly about **storing and accessing elements** with specific time/space complexities (for example, an ArrayList vs. a LinkedList).
- But streams are about expressing computations such as filter, sorted, and map that you saw earlier. **Collections are about data; streams are about computations**. We explain this idea in greater detail in the coming sections

3. Introducing Streams

- **Stream is "a sequence of elements from a source that supports data processing operations**

- **<u>Source:</u>** Streams consume from a data-providing source such as collections, arrays, or I/O resources. Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.

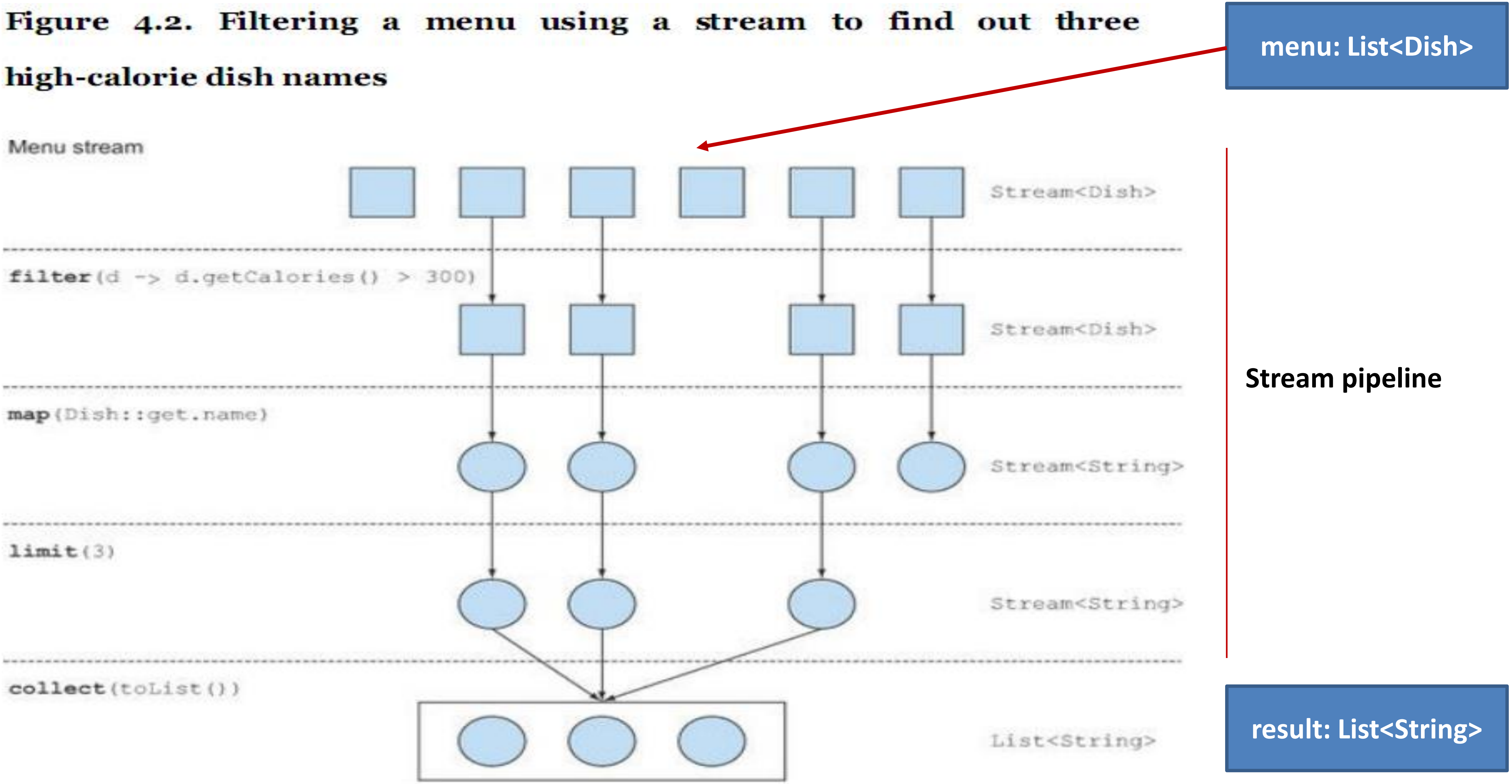
3. Introducing Streams

- **Stream is "a sequence of elements from a source that supports data processing operations**

- **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.

3. Introducing Streams

Figure 4.2. Filtering a menu using a stream to find out three high-calorie dish names



3. Introducing Streams

In addition, stream operations have two important characteristics:

- ***Pipelining***— Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline. This enables certain optimizations that we explain in the next chapter, such as *laziness* and *short-circuiting*. 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. We briefly mentioned this idea in chapter 1 and return to it later in the next section.

3. Introducing Streams

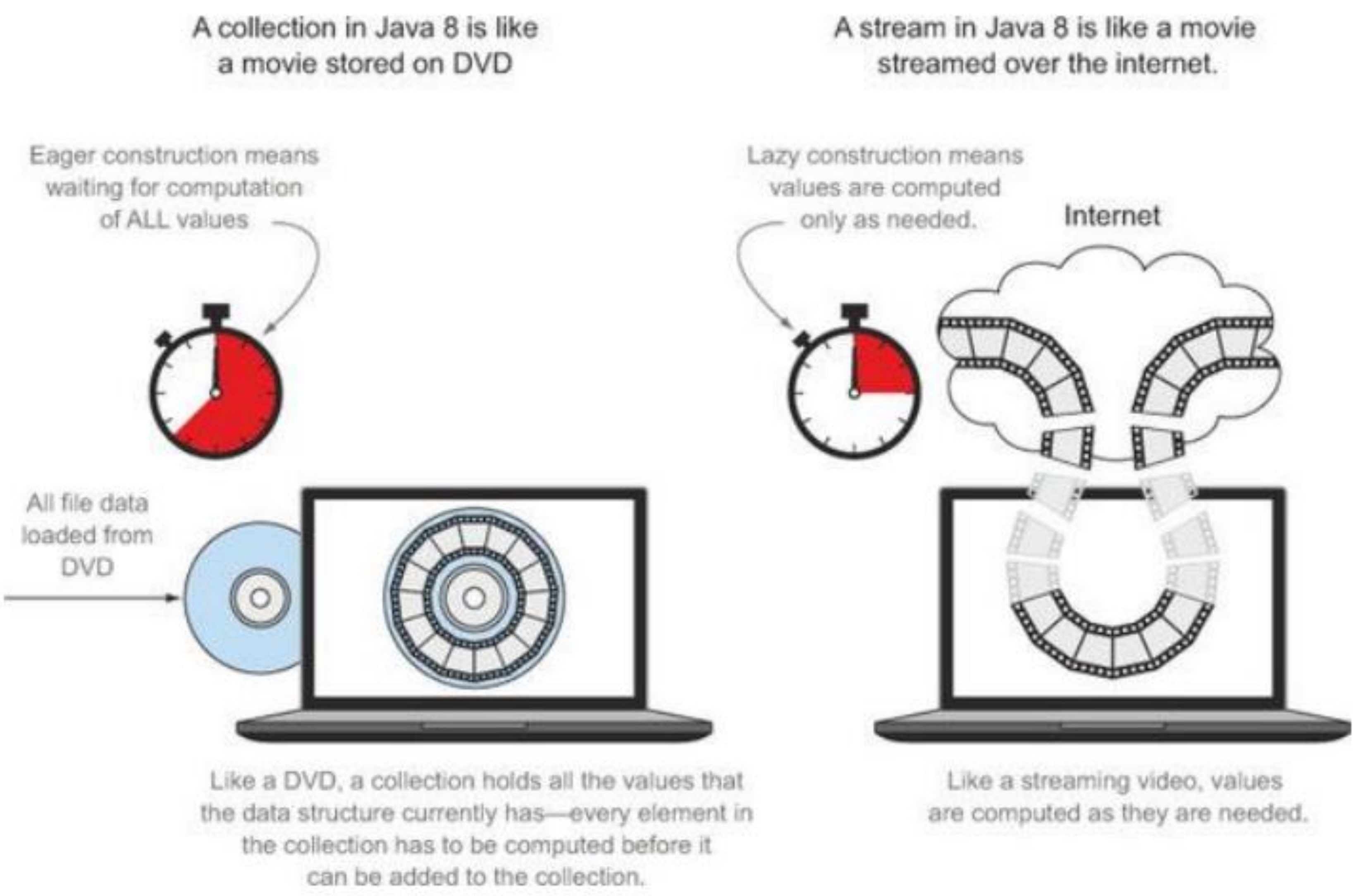
- **Streams vs collections**
- Both the existing Java notion of collections and the new notion of streams provide interfaces to data structures representing a sequenced set of values of the element type.
- By sequenced, we mean that we commonly step through the values in turn rather than randomly accessing the in any order. So what's the difference?

- A collection is an **in-memory data structure that holds all the values the data structure** currently has—every element in the collection has to be computed before it can be added to the collection. (You can add things to, and remove them from, the collection, but at each moment in time, every element in the collection is stored in memory; elements have to be computed before becoming part of the collection (eager constructed)

- A stream is a conceptually **fixed data structure** (you can't add or remove elements from it) whose elements are computed on demand
- List<T> => Stream<T> => Stream#filter (lazy constructed)

3. Introducing Streams

Figure 4.3. Streams vs. collections



3. Introducing Streams

- **Traversable only once**
- Similarly to iterators, a stream can be traversed only once. After that a **stream is said to be consumed**. You can get a new stream from the initial data source to traverse it again just like for an iterator (assuming it's a repeatable source like a collection; if it's an I/O channel, you're out of luck). For example, the following code would throw an exception indicating the stream has been consumed

```java
List<Dish> menu = mock();
Stream<Dish> stream = menu.stream();
stream.forEach(System.out::println);
stream.forEach(System.out::println);
```

- Imagine that you cannot **re-watch** the video was streamed in the past (except - saved video)
- So keep in mind that you can consume a stream only once !

3. Introducing Streams

- External vs internal iteration
- Using the **Collection** interface requires iteration to be done by the user (for example, using for-each); this is called **external iteration**
- The **Streams** library by contrast uses **internal iteration**—it does the iteration for you and takes care of storing the resulting stream value somewhere; you merely provide a function saying what's to be done

```
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.

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

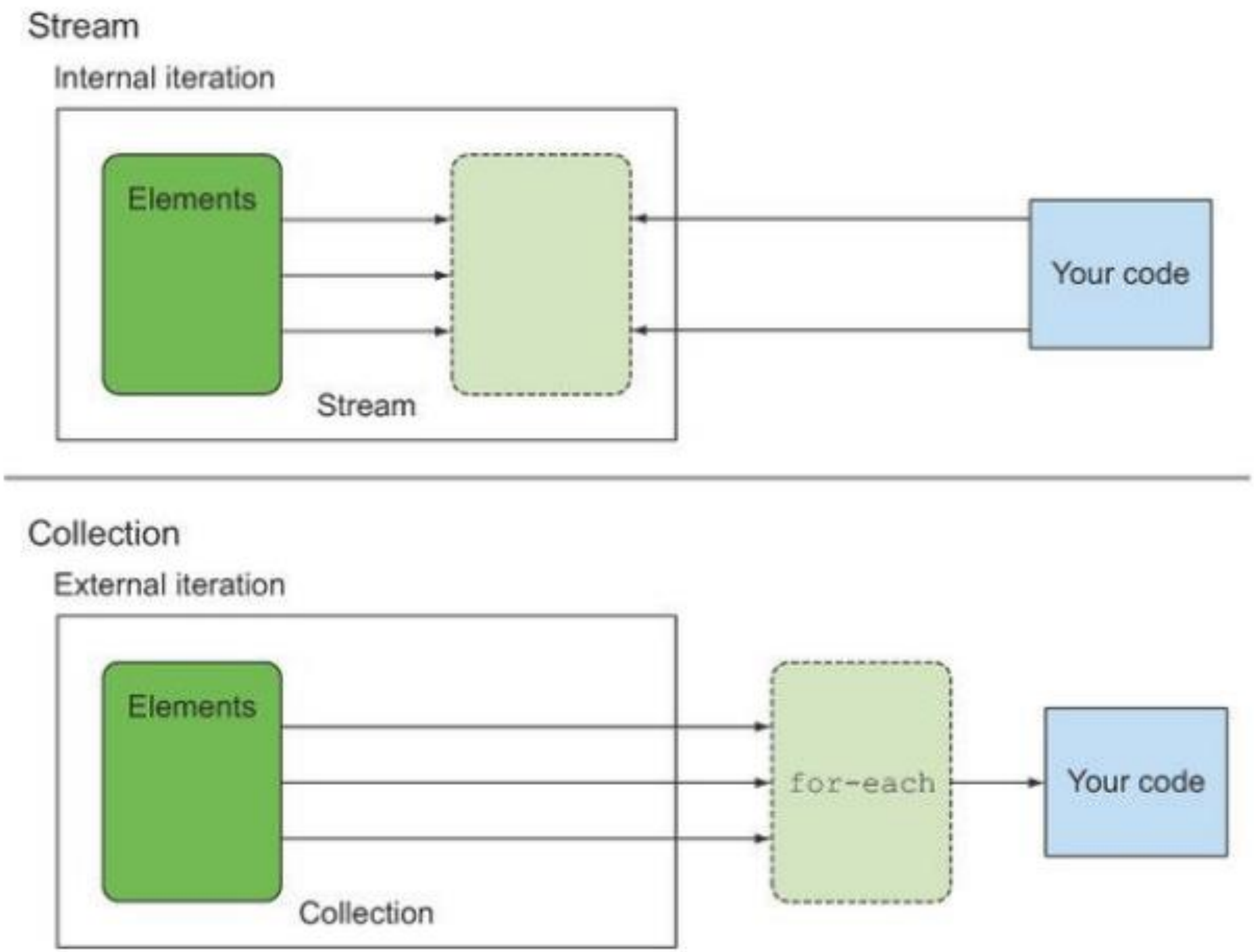
Start executing the pipeline of operations; no iteration!

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

3. Introducing Streams

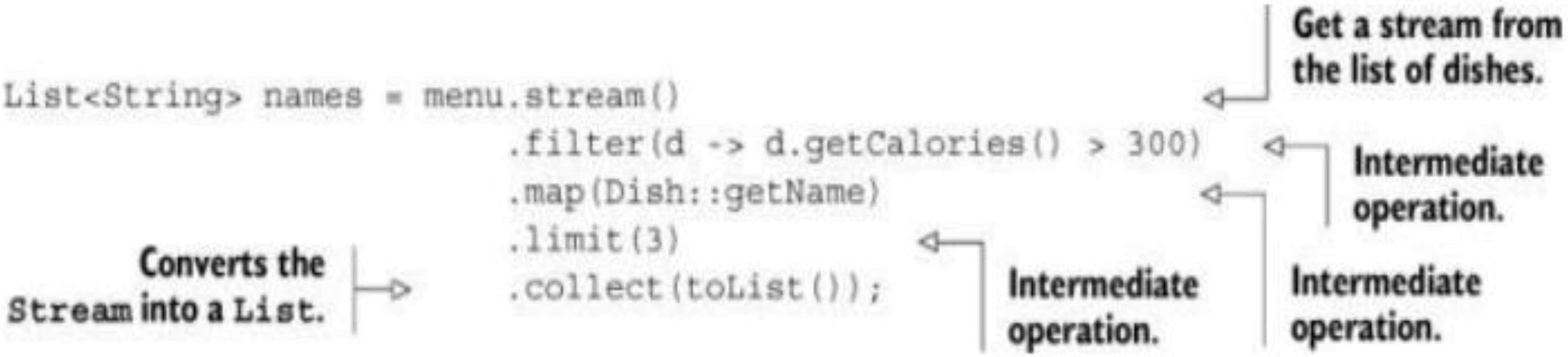
- **External vs internal iteration**

Figure 4.4. Internal vs. external iteration



3. Introducing Streams

- **Stream operations**
- The Stream interface in java.util.stream.Stream defines many operations. They can be classified into two categories. Let's look at our previous example once again:

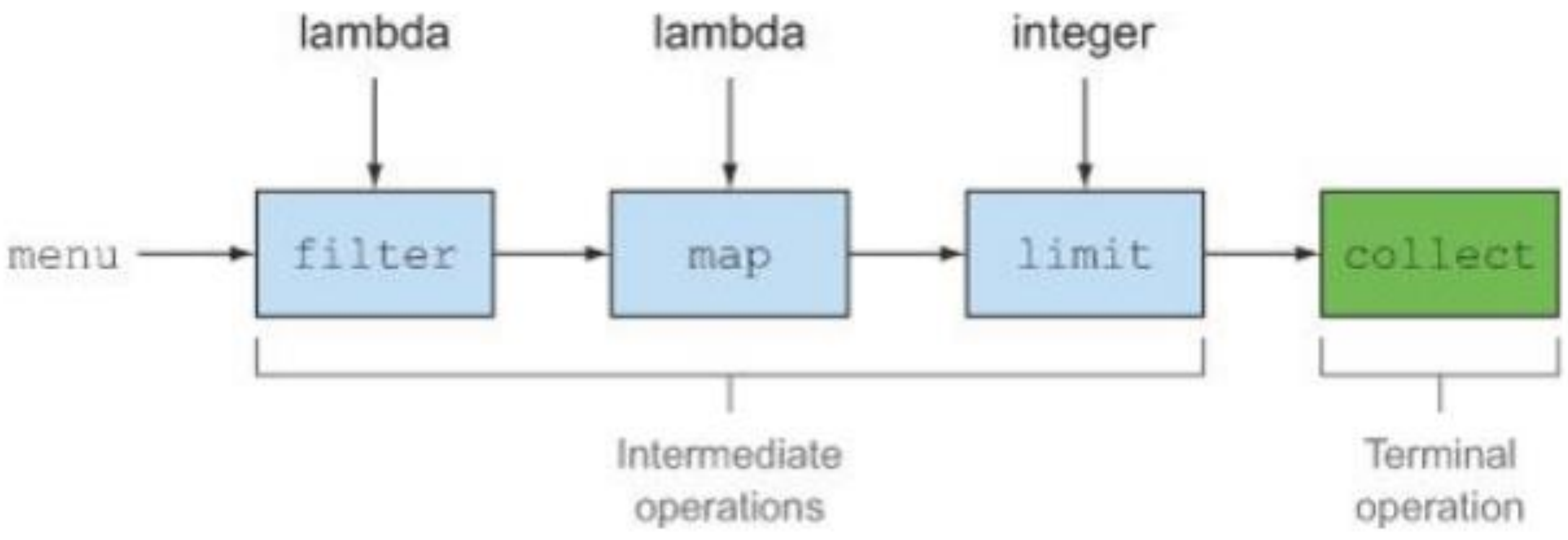


- You can see two groups of operations
  - filter, map, and limit can be connected together to form a pipeline.
  - collect causes the pipeline to be executed and closes it

3. Introducing Streams

- **Stream operation**
- Stream operations that can be connected are called intermediate operations, and operations that close a stream are called terminal operations

Figure 4.5. Intermediate vs. terminal operations



3. Introducing Streams

- Intermediate operations

- Intermediate operations such as **filter** or **sorted** **return another stream as the return type**. This allows the operations to be connected to form a query. What's important is that intermediate operations **don't perform any processing until a terminal operation is invoked** on the stream pipeline—**they're lazy**. This is because intermediate operations can usually be merged and processed into a single pass by the terminal operation

3. Introducing Streams

- **Intermediate operations**

```
List<String> names =
    menu.stream()
        .filter(d -> {
                        System.out.println("filtering" + d.getName());
                        return d.getCalories() > 300;
                    })
        .map(d -> {
                    System.out.println("mapping" + d.getName());
                    return d.getName();
                })
        .limit(3)
        .collect(toList());
System.out.println(names);
```

Printing the dishes as they're filtered

Printing the dishes as you extract their names

This code when executed will print the following:

filtering pork
mapping pork
filtering beef
mapping beef
filtering chicken
mapping chicken
[pork, beef, chicken]

Console will print nothing in case without collect method (terminal operation)

3. Introducing Streams

- **Terminal operations**
- Terminal operations **produce a result from a stream pipeline**. A result is any **non-stream** value such as a List, an Integer, or even void
- Example: collect – count – forEach

```
long count = menu.stream()
                 .filter(d -> d.getCalories() > 300)
                 .distinct()
                 .limit(3)
                 .count();
```

3. Introducing Streams

- **Working with streams**
- To summarize, working with streams in general involves three items:
  - 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

3. Introducing Streams

- **Working with streams**

Table 4.1. Intermediate operations

Operation	Type	Return type	Argument of the operation	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T, R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Intermediate	Stream<T>		

3. Introducing Streams

- Working with streams

Table 4.2. Terminal operations

Operation	Type	Purpose
forEach	Terminal	Consumes each element from a stream and applies a lambda to each of them. The operation returns void.
count	Terminal	Returns the number of elements in a stream. The operation returns a long.
collect	Terminal	Reduces the stream to create a collection such as a List, a Map, or even an Integer. See chapter 6 for more detail.

3. Introducing Streams

- **Summary**

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

Streams make use of internal iteration: the iteration is abstracted away through operations such as filter, map, and sorted.

There are two types of stream operations: intermediate and terminal operations.

Intermediate operations such as filter and map return a stream and can be chained together. They're used to set up a pipeline of operations but don't produce any result.

Terminal operations such as forEach and count return a nonstream value and process a stream pipeline to return a result.

The elements of a stream are computed on demand.

4. Working with streams

- **This part covers**
- Filtering, slicing, and matching
- Finding, matching, and reducing
- Using numeric streams such as ranges of numbers
- Creating streams from multiple sources
- Infinite streams

4. Working with streams

- **4.1 Filtering and slicing**
- In this section, we look at how to select elements of a stream: filtering with a predicate, filtering only unique elements, ignoring the first few elements of a stream, or truncating a stream to a given size

4. Working with streams

- **4.1 Filtering and slicing**
- In this section, we look at how to select elements of a stream: **filtering with a predicate**, filtering only unique elements, ignoring the first few elements of a stream, or truncating a stream to a given size
- Example: create a vegetarian menu by filtering all vegetarian dishes from menu

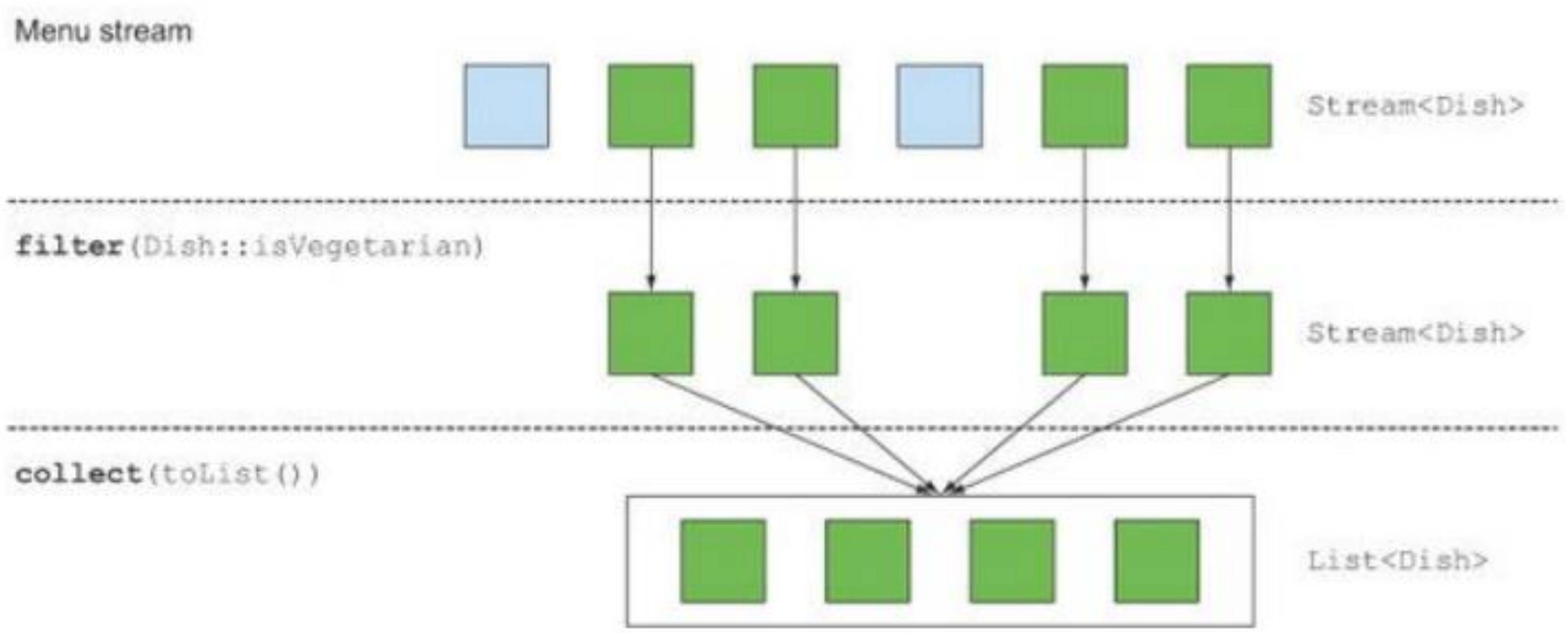
```
List<Dish> vegeterianMenu = menu.stream()
        .filter(Dish::isVegeterian)
        .collect(Collectors.toList());
```

- Exercise: create a MEAT menu be filtering all MEAT dishes from menu

4. Working with streams

- **4.1 Filtering and slicing**

Figure 5.1. Filtering a stream with a predicate



4. Working with streams

- 4.1 Filtering and slicing: **filtering only unique elements**
- Example: filter all even numbers from a list and makes sure that there are no duplicates

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

- Exercise: create a MEAT menu from initial menu and makes sure that there are no duplicates of calories
- Point: Distinct by properties using concurrent set or Collectors.toMap that key is property

4. Working with streams

- **4.1 Filtering and slicing: truncate and skip a stream**
- Streams support the **limit(n)** method, which returns another stream that's no longer than a given size. The requested size is passed as argument to limit. If the stream is ordered, the first elements are returned up to a maximum of n

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

4. Working with streams

- **4.1 Filtering and slicing: truncate and skip a stream**
- 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. Note that limit(n) and skip(n) are complementary

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

4. Working with streams

- **4.1 Filtering and slicing: truncate and skip a stream**

Quiz 5.1: Filtering

How would you use streams to filter the first two meat dishes?

4. Working with streams

- **4.2 Mapping**
- A very common data processing idiom is to select information from certain objects. For example, in SQL you can select a particular column from a table. The Streams API provides similar facilities through the **map** and **flatMap** methods.

```
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());
```

4. Working with streams

- **4.2 Mapping: flapMap**
- which is used to flatten a stream of collections to a stream of elements combined from all collections
- The flatMap() operation has the effect of applying a one-to-many transformation to the elements of the stream, and then flattening the resulting elements into a new stream.
- Syntax: flatMap method

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```

- Stream.flatMap helps in converting **Collection<Collection<T>>** to **Collection<T>**
- **Collection<Collection<T>>** to **Stream<Collection<T>>** **Stream<T>** to **Collection<T>**
- **flatMap() = map() + Flattening**

4. Working with streams

- **4.2 Mapping: flapMap**
- Flattening is referred by converting several lists of lists, and merge all those lists to create single list containing all the elements from all the lists.

```
Flattening example

Before flattening    : [[1, 2, 3], [4, 5], [6, 7, 8]]
After flattening     : [1, 2, 3, 4, 5, 6, 7, 8]
```

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```

```
Similar methods

IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)
LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)
DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)
```

4. Working with streams

- **4.2 Mapping: flapMap**
- It is an **intermediate operation** and return another stream as method output return value.
- Returns a stream consisting of the results of replacing each element of the given stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
- The function used for transformation in flatMap() is a stateless function and returns only a stream of new values.
- Each mapped stream is closed after its contents have been placed into new stream.
- flatMap() operation flattens the stream; opposite to map() operation which does not apply flattening.

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```

4. Working with streams

- **4.2 Mapping: flapMap**
- Example: Convert list of lists to single list

```
List<Integer> list1 = Arrays.asList(1,2,3);
List<Integer> list2 = Arrays.asList(4,5,6);
List<Integer> list3 = Arrays.asList(7,8,9);

List<List<Integer>> listOfLists = Arrays.asList(list1, list2, list3);

List<Integer> listOfAllIntegers = listOfLists.stream()
                            .flatMap(i -> i.stream())
                            .collect(Collectors.toList());
```

Collection::stream

4. Working with streams

- **4.2 Mapping: flapMap**
- Example: Convert array of arrays to single array

```
String[][] dataArray = new String[][]{{"a", "b"}, {"c", "d"}, {"e", "f"}, {"g", "h"}};

String[] listOfAllChars = Arrays.stream(dataArray)
                           .flatMap(d -> Arrays.stream(d))
                           .toArray(String[]::new);
```

Arrays::stream

4. Working with streams

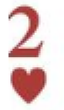
- **4.2 Mapping: flapMap**
- Example: Convert array of arrays to single list

```java
String[][] dataArray = new String[][]{{"a", "b"}, {"c", "d"}, {"e", "f"}, {"g", "h"}};

List<String> listOfAllChars = Arrays.stream(dataArray)
                              .flatMap(d -> Arrays.stream(d))
                              .collect(Collectors.toList());
```

Arrays::stream

4. Working with streams

- **4.2 Mapping: flapMap**
- Exercise: List<List<String>>: String => Cards of players (remaining players after the 1^{st} win)
- Look for the two of hearts left 

⇒ Stream<Object[]> => Stream<Object> ⇔ flatMap(Arrays::stream)
⇒ Stream<Collection<T>> => Stream<T> ⇔ flapMap(Collection::stream)

4. Working with streams

- **4.3 Finding and matching:**
- Another common data processing idiom is finding whether some elements in a set of data match a given property. The Streams API provides such facilities through the **allMatch**, **anyMatch**, **noneMatch**, **findFirst**, and **findAny** methods of a stream
- Parameter: Function<T, R>

- **allMatch:** whether all the elements of the stream match the given predicate
- **anyMatch:** whether is there an element in the stream matching the given predicate
- **noneMatch:** ensures that no elements in the stream match the given predicate
- **findFirst:** returns the first element of the current stream, combine with filter, map …
- **findAny:** returns an arbitrary element of the current stream, combine with filter …

4. Working with streams

- **4.3 Finding and matching:**
- Example: Give a list of numbers: 1, 2, 1, 4, 5, 8, 10, 4, 12
- 1st: is there any element that is divisible by 10
- 2nd: find the first element that is divisible by 4
- 3nd: find the element that is divisible by 5 in the list
- 4th: does all elements less than 100
- 5th: does no elements greater than 0

4. Working with streams

- **4.3 Finding and matching:**
- **Optional in a nutshell**

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>. We won't go into the details of Optional here, because we show in detail in chapter 10 how your code can benefit from using Optional to avoid bugs related to null checking. But for now, it's good to know that there are a few methods available in Optional that force you 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. We introduced the Consumer functional interface in chapter 3; it lets you pass a lambda that takes an argument of type T and returns void.
- T get() returns the value if present; otherwise it throws a NoSuchElement-Exception.
- T orElse(T other) returns the value if present; otherwise it returns a default value.

4. Working with streams

- **4.4 Reducing:**
- So far, the terminal operations you've seen return a boolean (allMatch and so on), void (forEach), int (count) or an Optional object (findAny and so on). You've also been using collect to combine all elements in a stream into a List.

- In this section, you'll see how you can combine elements of a stream to express more complicated queries such as "Calculate the sum of all calories in the menu," or "What is the highest calorie dish in the menu?" using the reduce operation

4. Working with streams

- **4.4 Reducing: Summing of the elements**
- Have a look in the usual summing way first

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5);
int sum = 0;
for (Integer number: numbers) {
    sum += number.intValue();
}
```

- Using reducing method

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

Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

Integer::*sum*

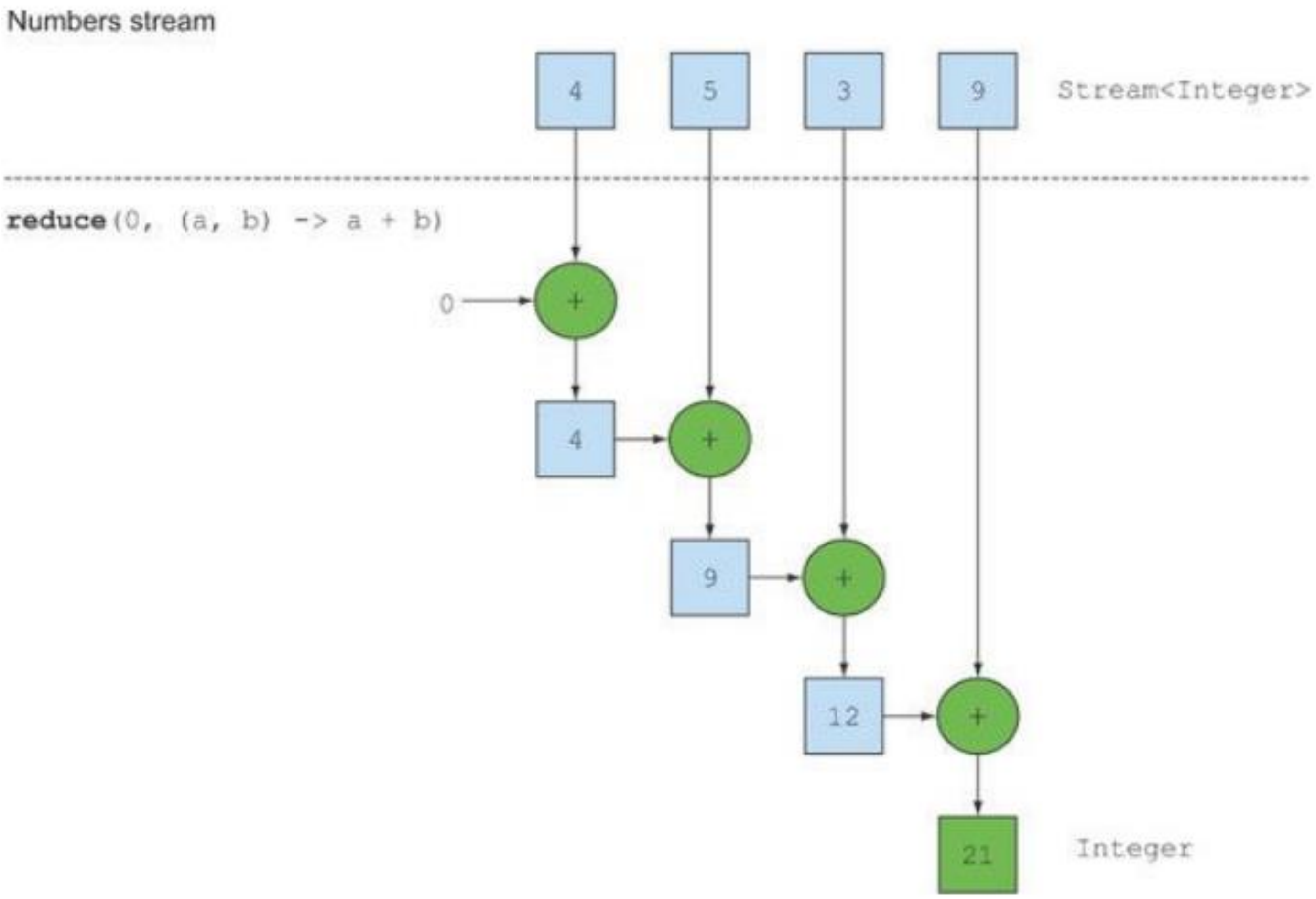
- Reduce takes two arguments
- A initial value, here 0
- A BinaryOperator<T> to combine two elements and produce a new value, here you use the lambda (a + b) -> a + b

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

4. Working with streams

- **4.4 Reducing: Summing of the elements**

Figure 5.7. Using reduce to sum the numbers in a stream



4. Working with streams

- 4.4 Reducing: Summing of the elements

- Quiz 4.4: Reducing
  - How could you count the number of dishes in a stream using the map and reducing method
  - How could you calculate sum of the calories in the menu

- So far you saw reduction examples that produced an Integer: the sum of a stream, the maximum of a stream, or the number of elements in a stream.
- You'll see in section 5.6 – numeric stream that built-in methods such as sum and max are available as well to help you **write slightly more concise code for common reduction patterns**

4. Working with streams

- **4.4 Reducing: Summing of the elements**
- Similar with calculate max and min

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

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

```
(a, b) -> a > b ? a : b
```

- Don't forget that min, max, sum, subtract could use with the single type only (Integer, String, Double, Float, Long)

4. Working with streams

- **Stream operations: stateless vs stateful**
- Stateless: filter, map, flapMap, match, find, collect, count
- Operations like map and filter take each element from the input stream and produce zero or one result in the output stream. These operations are thus in general stateless: they don't have an internal state

- Stateful: distinct, skip, limit, sorted, reduce
- Operations like reduce, sum, and max need to have internal state to accumulate the result. In this case the internal state is small. In our example it consisted of an int or double
- By contrast, some operations such as sorted or distinct seem at first to behave like filter or map—all take a stream and produce another stream (an intermediate operation), but there's a crucial difference. Both **sorting** and **removing** duplicates from a stream require knowing the **previous history** to do their job

4. Working with streams

Table 5.1. Intermediate and terminal operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
distinct	Intermediate (stateful-unbounded)	Stream<T>		
skip	Intermediate (stateful-bounded)	Stream<T>	long	
limit	Intermediate (stateful-bounded)	Stream<T>	long	
map	Intermediate	Stream<R>	Function<T, R>	T -> R
flatMap	Intermediate	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	Intermediate (stateful-unbounded)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	Terminal	boolean	Predicate<T>	T -> boolean
noneMatch	Terminal	boolean	Predicate<T>	T -> boolean
allMatch	Terminal	boolean	Predicate<T>	T -> boolean
findAny	Terminal	Optional<T>		
findFirst	Terminal	Optional<T>		
forEach	Terminal	void	Consumer<T>	T -> void
collect	terminal	R	Collector<T, A, R>	
reduce	Terminal (stateful-bounded)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	Terminal	long		

4. Working with streams

4.4 Numeric streams

You saw earlier that you could use the reduce method to calculate the sum of the elements of a stream. For example, you can calculate the number of calories in the menu as follows:

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .reduce(0, Integer::sum);
```

The problem with this code is that there's an insidious boxing cost. Behind the scenes each Integer needs to be unboxed to a primitive before performing the summation. In addition, wouldn't it be nicer if you could call a sum method directly as follows?

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

But this isn't possible. The problem is that the method map generates a Stream<T>. Even though the elements of the stream are of type Integer, the Streams interface doesn't define a sum method. Why not? Say you had only a Stream<Dish> like the menu; it wouldn't make any sense to be able to sum dishes. But don't worry; the Streams API also supplies *primitive stream specializations* that support specialized methods to work with streams of numbers.

4. Working with streams

- **4.4 Numeric streams : Primitive stream specializations**
- Java 8 introduces three primitive specialized stream interfaces to tackle this issue, **IntStream**, **DoubleStream**, and **LongStream**, that respectively specialize the elements of a stream to be int, long, and double—and there by avoid hidden boxing costs

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

IntStream

Stream<Integer> - IntStream : Unboxed

- Converting back to stream of objects

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> calories = intStream.boxed();
```

4. Working with streams

- **4.4 Numeric streams : Numeric ranges**
- A common use case when dealing with numbers is working with ranges of numeric values. For example, suppose you'd like to generate all numbers between 1 and 100.
- Java 8 introduces two static methods available on IntStream and LongStream to help generate such ranges: range and rangeClosed. Both methods take the starting value of the range as the first parameter and the end value of the range as the second parameter. But range is exclusive, whereas rangeClosed is inclusive

Represents the range [1, 100].

```
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
                                 .filter(n -> n % 2 == 0);
System.out.println(evenNumbers.count());
```

A stream of even numbers from 1 to 100.

There are 50 even numbers from 1 to 100.

4. Working with streams

- **4.4 Numeric streams : Arithmetic Operations**
- Let's start with a few interesting methods used arithmetic operations such as min, max, sum, and average:

```java
int[] integers = new int[] {12, 82, 20, 4};
int min = Arrays.stream(integers)
  .min()
  .getAsInt(); // returns 4
```

```java
double avg = IntStream.of(10, 20, 40, 16, 14)
          .average()
          .getAsDouble(); // returns 20
```

```java
int max = IntStream.of(20, 98, 12, 7, 35)
            .max()
            .getAsInt(); // returns 98
```

```java
double[] doubles = { 20d, 98d, 12d, 7d, 35d };
Arrays.stream(doubles).max();

System.out.println(Stream.of(doubles)
        .flatMapToDouble(row -> Arrays.stream(row)).max());

int[] ints = { 20, 98, 12, 7, 35 };
System.out.println(Stream.of(ints)
        .flatMapToInt(row -> Arrays.stream(row)).average().getAsDouble());
```

4. Working with streams

- **4.5 Building streams**:
- Hopefully by now you're convinced that streams are very powerful and useful to express data processing queries. So far, you were able to get a stream from a collection, array using the stream, Arrays::stream method. In addition, we showed you how to create numerical streams from a range of numbers.
- But you can create streams in many more ways! This section shows how you can create a stream from a sequence of values, from an array, from a file, and even from a generative function to create infinite streams!

4. Working with streams

- **4.5 Building streams:**
- **Streams from values**

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

You can get an empty stream using the empty method as follows:

```
Stream<String> emptyStream = Stream.empty();
```

- **Streams from arrays**

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```

The sum is 41.

- **Streams from collections**

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5);
Stream<Integer> stream = numbers.stream();
```

4. Working with streams

- **4.5 Building streams:**
- **Streams from function**: creating infinite streams!
- The Streams API provides two static methods to generate a stream from a function: **Stream.iterate** and **Stream.generate**.

- These two operations let you create what we call an infinite stream: a stream that doesn't have a fixed size like when you create a stream from a fixed collection. Streams produced by iterate and generate create values on demand given a function and can therefore calculate values forever! It's generally sensible to use limit(n) on such streams to avoid printing an infinite number of values

4. Working with streams

- **4.5 Building streams:**
- **Streams from function**: creating infinite streams!
- **Stream.iterate**

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

Fibonacci

```
Stream.iterate(new int[]{0, 1},
               t -> new int[]{t[1], t[0]+t[1]})
      .limit(20)
      .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));
```

The iterate method takes an initial value, here 0, and a lambda (of type Unary-Operator<T>) to apply successively on each new value produced. Here you return the previous element added with 2 using the lambda n -> n + 2. As a result, the iterate method produces a stream of all even numbers: the first element of the stream is the initial value 0. Then it adds 2 to produce the new value 2; it adds 2 again to produce the new value 4 and so on. This iterate operation is fundamentally sequential because the result depends on the previous application. Note that this operation produces an *infinite stream*—the stream doesn't have an end because values are computed on demand and can be computed forever. We say the stream is *unbounded*. As we discussed earlier, this is a key difference between a stream and a collection. You're using the limit method to explicitly limit the size of the stream. Here you select only the first 10 even numbers. You then call the forEach terminal operation to consume the stream and print each element individually.

Stateful

4. Working with streams

- **4.5 Building streams:**
- **Streams from function**: creating infinite streams!
- **Stream.generate:** Similarly to the method iterate, the method generate lets you produce an infinite stream of values computed on demand. But generate doesn't apply successively a function on each new produced value. It takes a lambda of type Supplier<T> to provide new values. Let's look at an example of how to use it

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

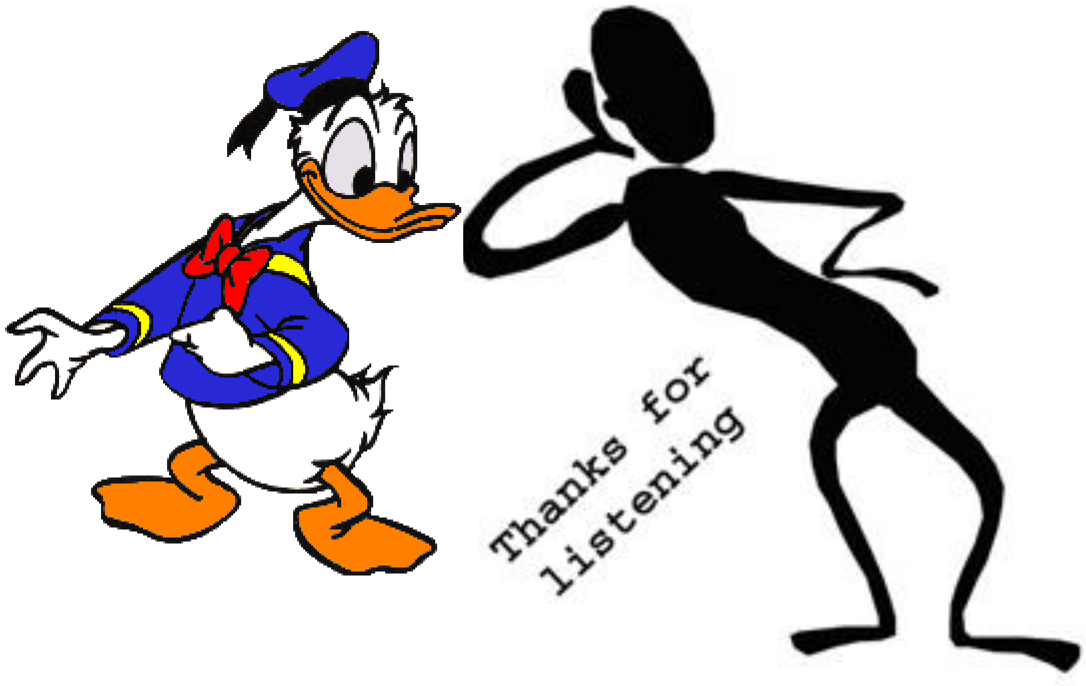
Stateless

4. Working with streams

▪ 4.5 Summary

It's been a long but rewarding chapter! You can now process collections more effectively. Indeed, streams let you express sophisticated data processing queries concisely. In addition, streams can be parallelized transparently. Here are some key concepts to take away from this chapter:

- The Streams API lets you express complex data processing queries. Common stream operations are summarized in table 5.1.
- You can filter and slice a stream using the filter, distinct, skip, and limit methods.
- You can extract or transform elements of a stream using the map and flatMap methods.
- You can find elements in a stream using the findFirst and findAny methods. You can match a given predicate in a stream using the allMatch, noneMatch, and anyMatch methods.
- These methods make use of short-circuiting: a computation stops as soon as a result is found; there's no need to process the whole stream.
- You can 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.
- Some operations such as filter and map are stateless; they don't store any state. Some operations such as reduce store state to calculate a value. Some operations such as sorted and distinct also store state because they need to buffer all the elements of a stream before returning a new stream. Such operations are called *stateful operations*.
- There are three primitive specializations of streams: IntStream, DoubleStream, and LongStream. Their operations are also specialized accordingly.
- Streams can be created not only from a collection but also from values, arrays, files, and specific methods such as iterate and generate.
- An infinite stream is a stream that has no fixed size.



END