Java 8 Update Part 2

Functional Programming and Concurrency

Agenda

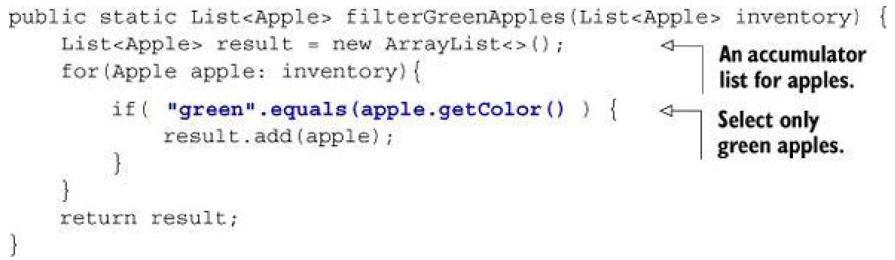
- Introduction to Functional Aspects in Java
- Lambda Expressions and Functional Interfaces
- Stream API
- Parallel Data Processing
- Updates to Interfaces; Default methods
- Functional Programming Techniques; Blending OOP and FP
- Quiz

Why Functional Programming?

- Behavioral Parametrization is a good example of how Functional Programming can be put to use
- It means taking a block of code and making it available without executing it
- E.g on a collection:
    - Can do “something” for every element of a list
    - Can do “something else” when you finish processing the list
    - Can do “yet something else” if you encounter an error
- The idea is turning *functions* into a passable value to methods

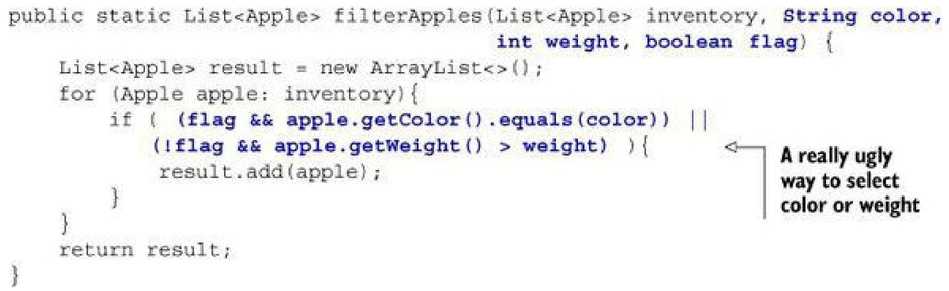
An Apple a Day...

- Image you want to filter green apples in your collection:



```
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;
}
```

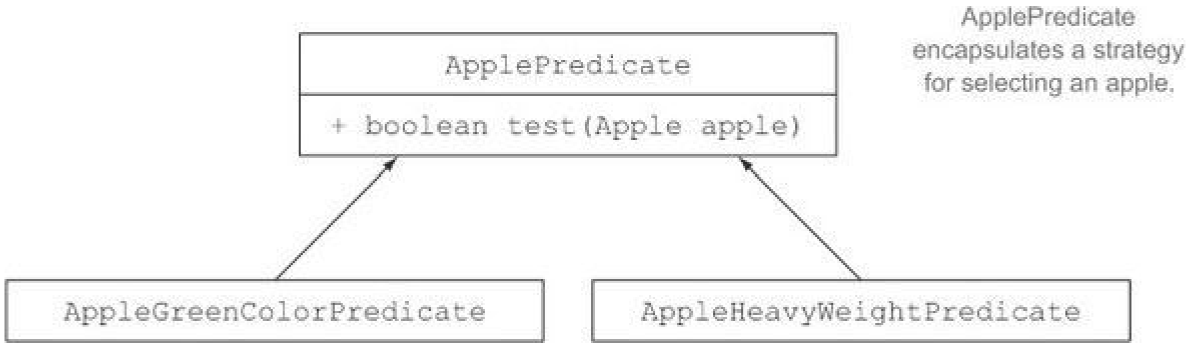
- Let’s say you want to filter by parametric color and weight:



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

Parametrize filter behavior using Strategy

- A design solution is to apply the Strategy pattern, creating an interface to encapsulate the filter behavior


ApplePredicate encapsulates a strategy for selecting an apple.
ApplePredicate
+ boolean test(Apple apple)
AppleGreenColorPredicate
AppleHeavyWeightPredicate

```
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;
}
```

The predicate object encapsulates the condition to test on an apple.

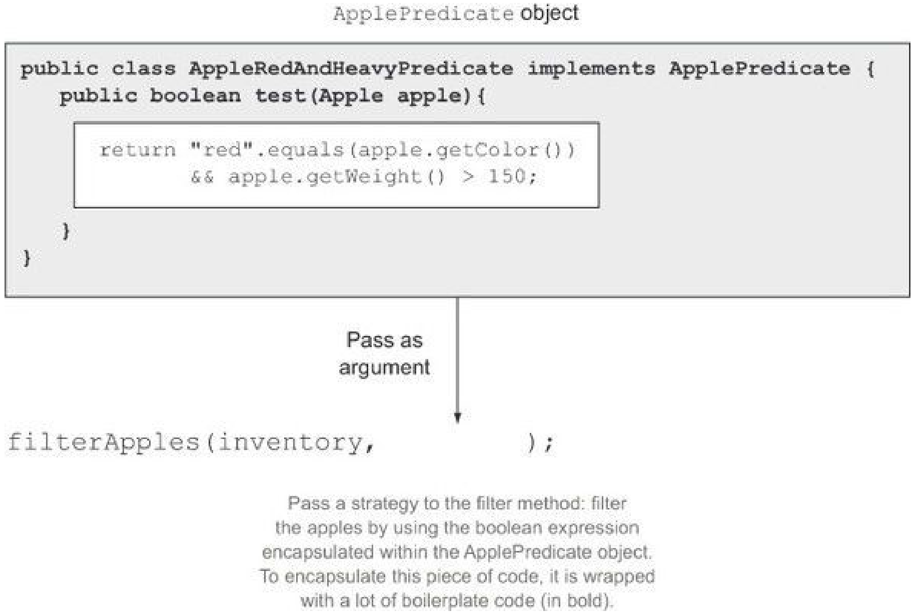
```
public class AppleHeavyWeightPredicate implements ApplePredicate{
    public boolean test(Apple apple){
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate{
    public boolean test(Apple apple){
        return "green".equals(apple.getColor());
    }
}
```

Select only heavy apples.

Select only green apples.

Passing dynamic code as Strategy

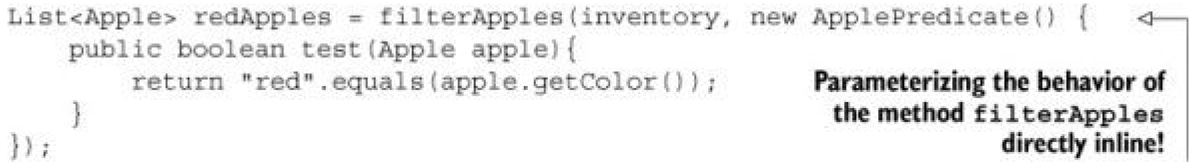
- We can use anonymous classes to avoid creating a specific method for each behaviour



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

```
filterApples(inventory,          );
```

Pass a strategy to the filter method: filter the apples by using the boolean expression encapsulated within the ApplePredicate object. To encapsulate this piece of code, it is wrapped with a lot of boilerplate code (in bold).



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

Parameterizing the behavior of the method `filterApples` directly inline!

Exercise: Write an Apple Formatter

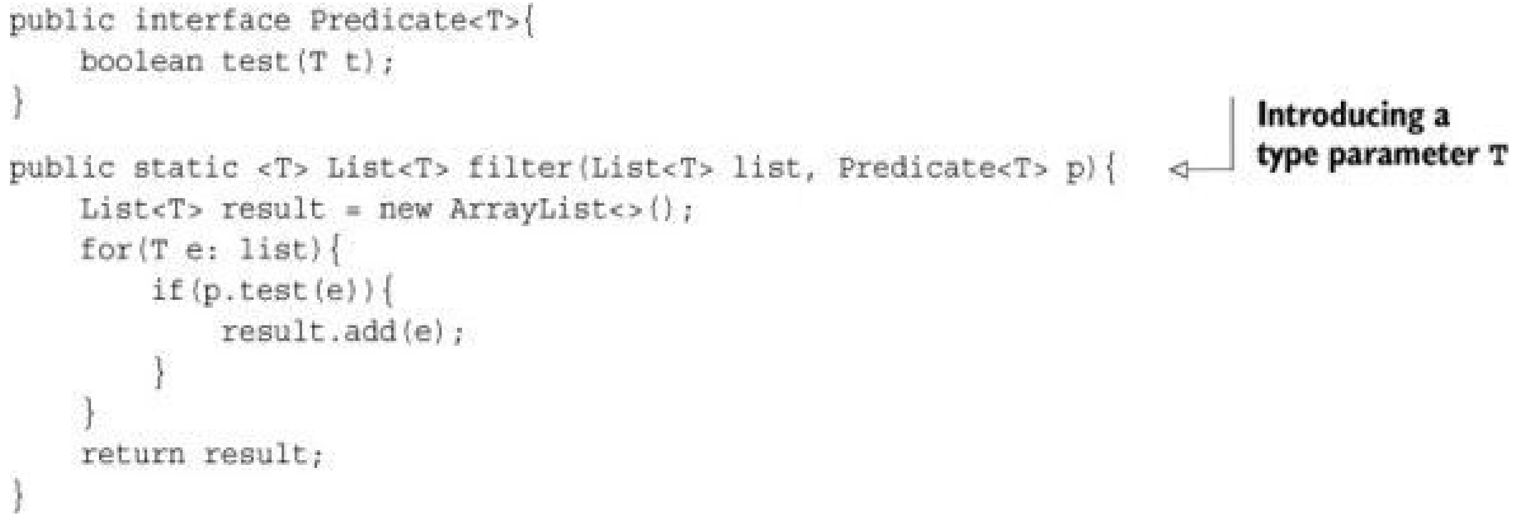
Write a behaviorally generic print formatter for Apples

You can start from this code:

```
public static void prettyPrintApple(List<Apple> inventory, ???){

for(Apple apple: inventory) {

        String output = ???.???(apple);

        System.out.println(output);

} }
```

Extending the filter interfaces with generics

- We can further generalize from Apples by using generics to define the functional interface

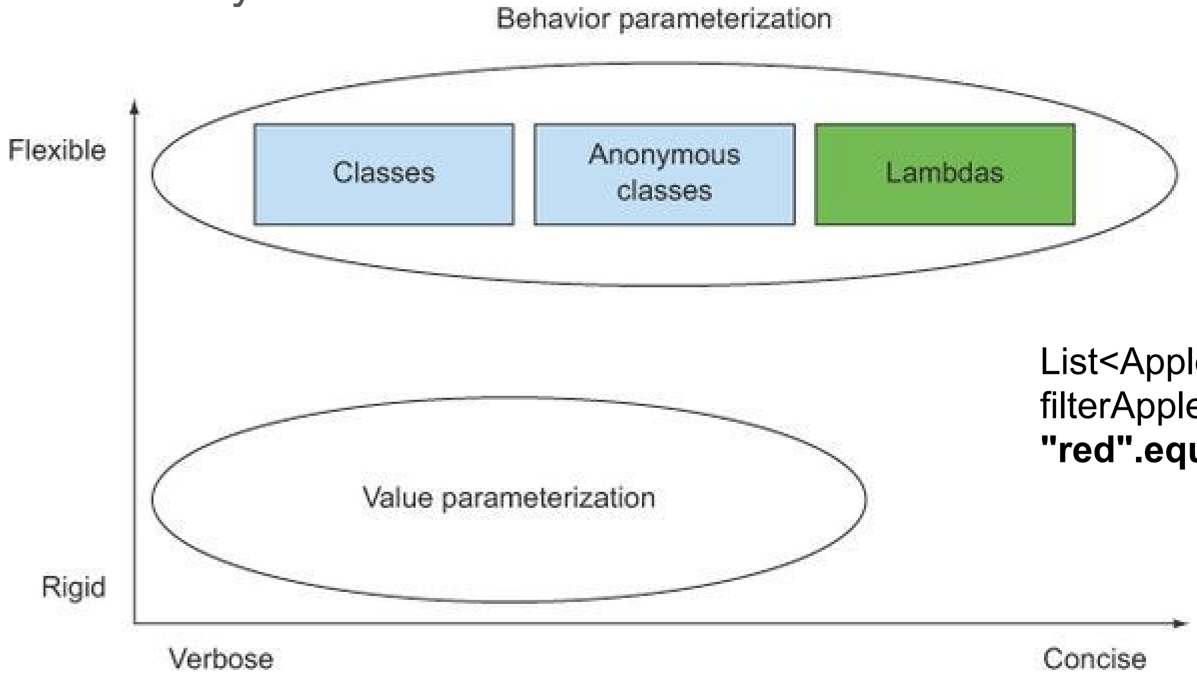


```
public interface Predicate<T>{
    boolean test(T t);
}

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;
}
```

Introducing a type parameter T

Using lambdas instead of anon classes

- Lambda function capture this idea to avoid complexity and scope problems in anonymous classes



List<Apple> result = filterApples(inventory, **(Apple apple) -> "red".equals(apple.getColor())**);

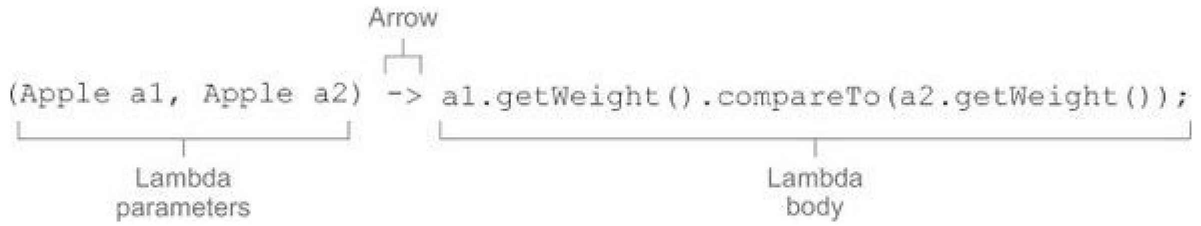
Lambda expressions

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.

Lke a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.

A lambda expression can be passed as argument to a method or stored in a variable. It is concise — You don’t need to write a lot of boilerplate like you do for anonymous classes.

Anatomy of a Lambda Expression

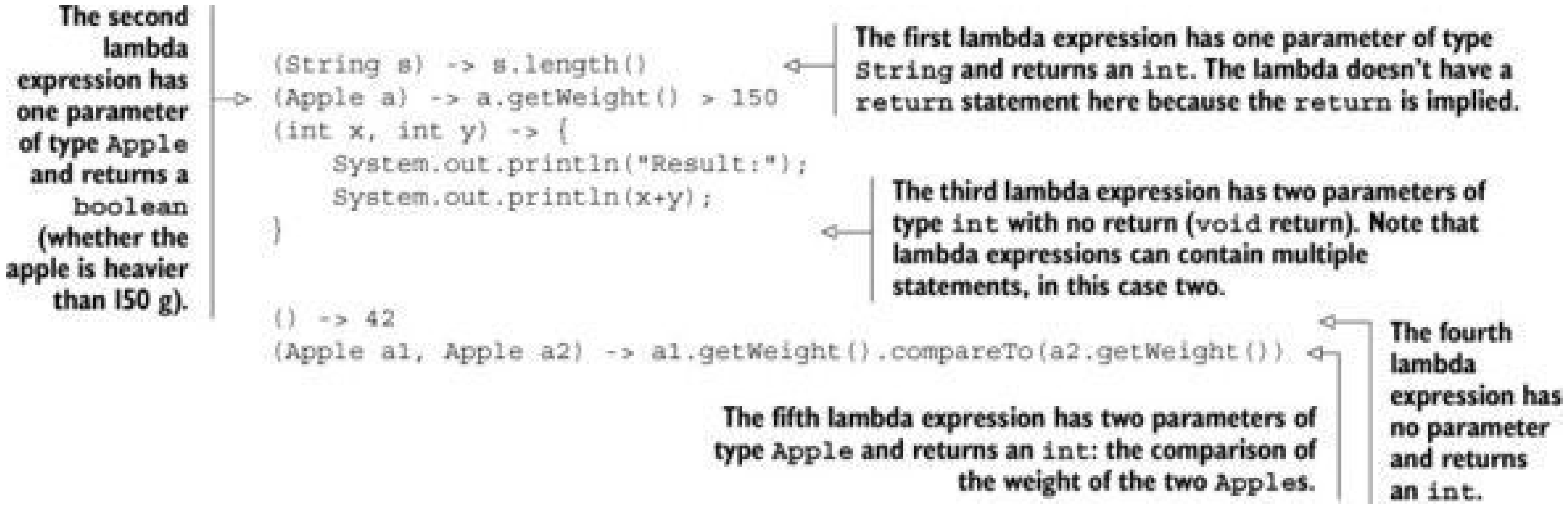

Arrow
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
Lambda parameters
Lambda body

A list of parameters— In this case it mirrors the parameters of the compare method of a Comparator—two Apples.

An arrow— The arrow -> separates the list of parameters from the body of the lambda.

The body of the lambda— Compare two Apples using their weights. The expression is considered the lambda’s return value.

Examples



```
(String s) -> s.length()
(Apple a) -> a.getWeight() > 150
(int x, int y) -> {
    System.out.println("Result:");
    System.out.println(x+y);
}

() -> 42
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
```

The first lambda expression has one parameter of type `String` and returns an `int`. The lambda doesn't have a `return` statement here because the `return` is implied.

The second lambda expression has one parameter of type `Apple` and returns a `boolean` (whether the apple is heavier than 150 g).

The third lambda expression has two parameters of type `int` with no return (`void` return). Note that lambda expressions can contain multiple statements, in this case two.

The fourth lambda expression has no parameter and returns an `int`.

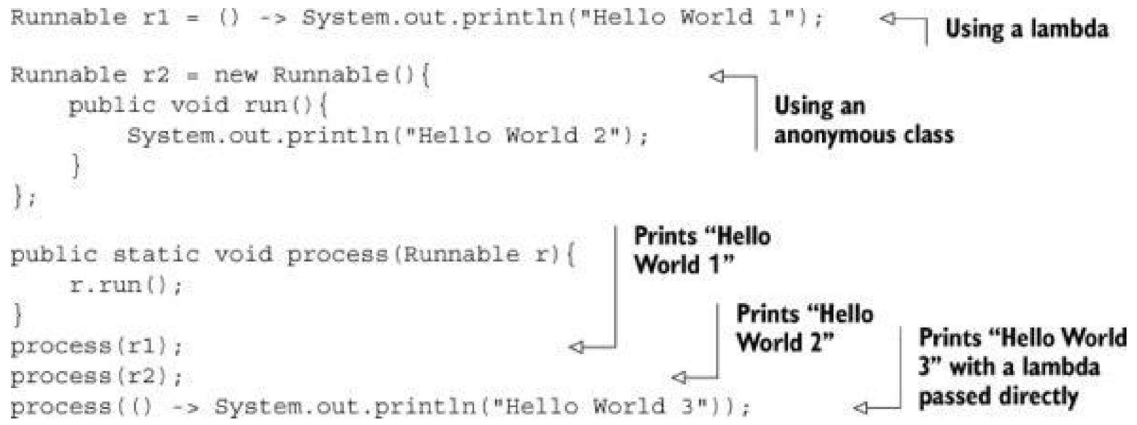
The fifth lambda expression has two parameters of type `Apple` and returns an `int`: the comparison of the weight of the two `Apples`.

Functional Interface

A functional interface is an interface that specifies *exactly one* abstract method.

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

Some pre-Java 8 interfaces are functional, e.g. *Runnable*



```
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"));
```

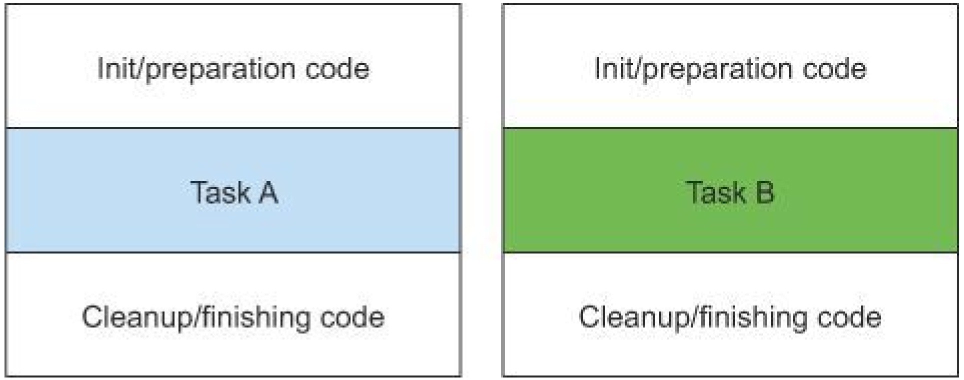
@FunctionalInterface

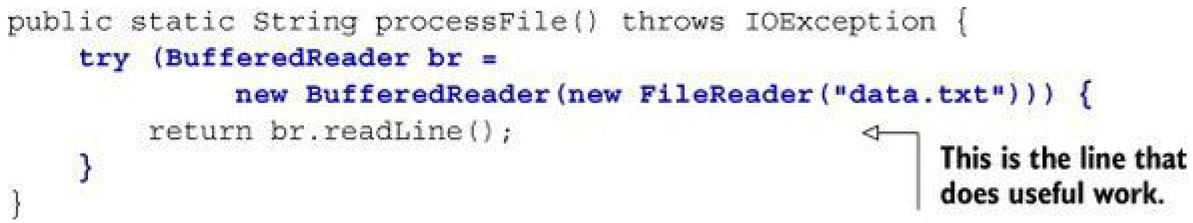
You can create specific interfaces with a single method

Use the @FunctionalInterface to clarify that the intent is creating a FI (not compulsory)

```
@FunctionalInterface
public interface BufferedReaderProcessor {
 String process(BufferedReader b) throws IOException;
}
```

Example: Execute Around



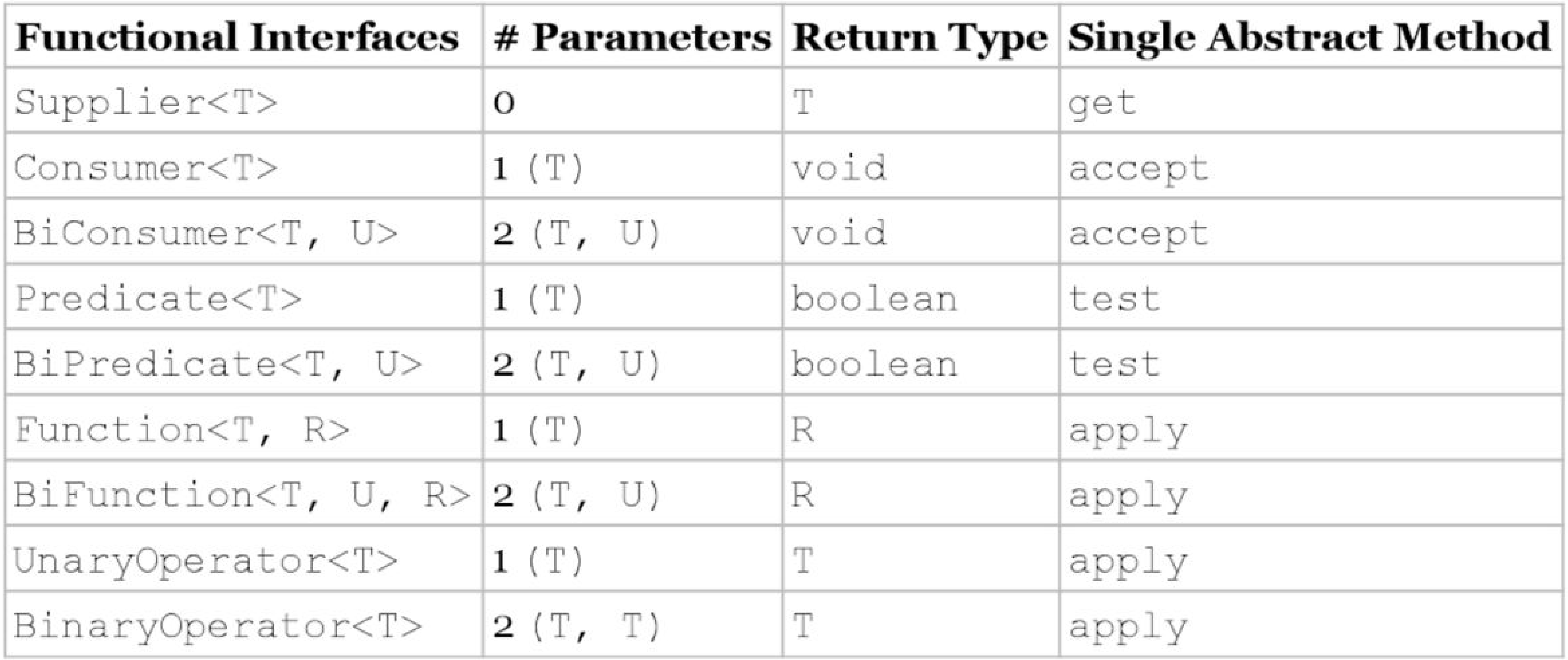


```
public static String processFile() throws IOException {
    try (BufferedReader br =
             new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();
    }
}
```

This is the line that does useful work.

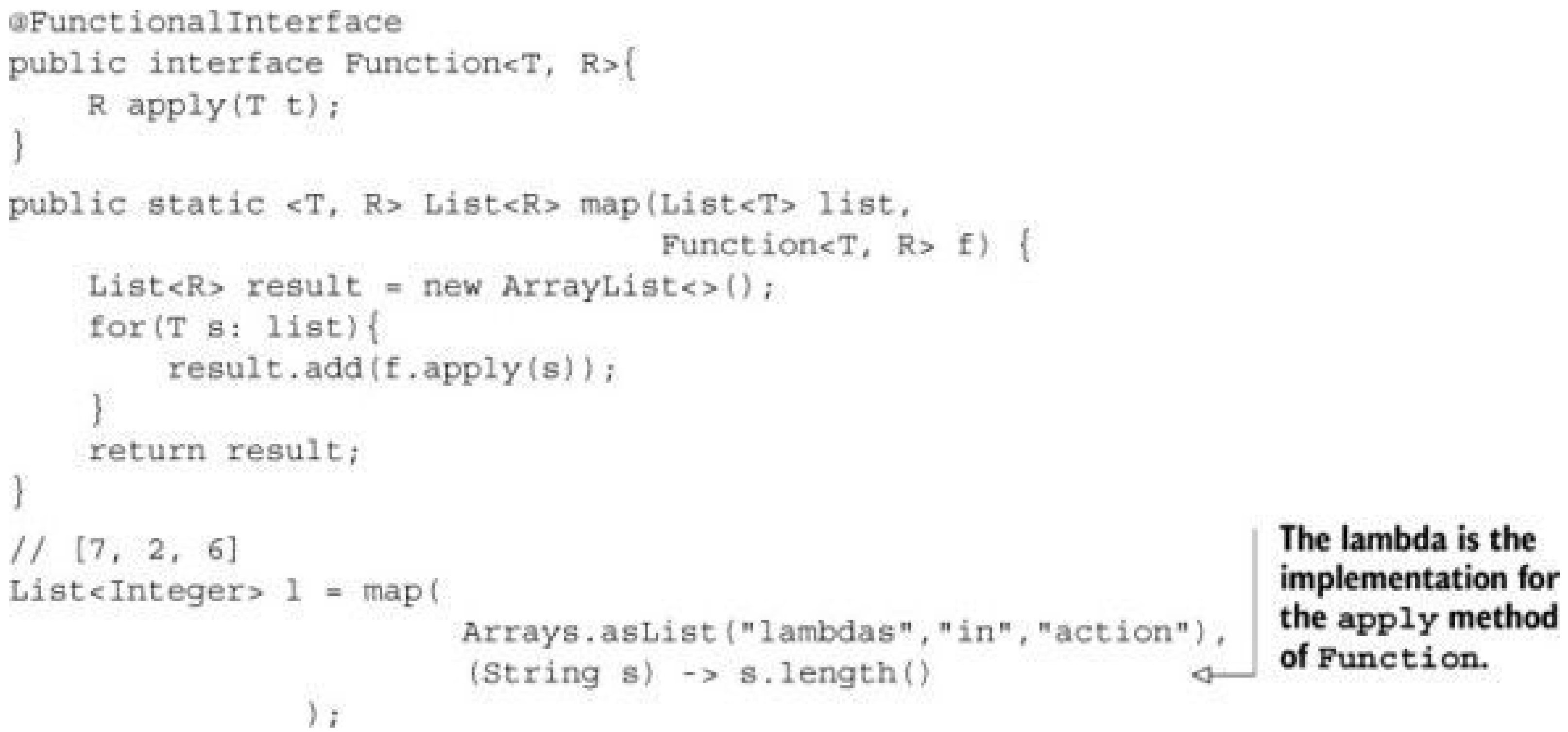
Reuse the processFile method and process files in different ways by passing different lambdas.

Common functional interfaces in Java 8



Functional Interfaces	# Parameters	Return Type	Single Abstract Method
`Supplier<T>`	0	`T`	`get`
`Consumer<T>`	1 (T)	`void`	`accept`
`BiConsumer<T, U>`	2 (T, U)	`void`	`accept`
`Predicate<T>`	1 (T)	`boolean`	`test`
`BiPredicate<T, U>`	2 (T, U)	`boolean`	`test`
`Function<T, R>`	1 (T)	`R`	`apply`
`BiFunction<T, U, R>`	2 (T, U)	`R`	`apply`
`UnaryOperator<T>`	1 (T)	`T`	`apply`
`BinaryOperator<T>`	2 (T, T)	`T`	`apply`

Example of Common FI: Function<T, R>

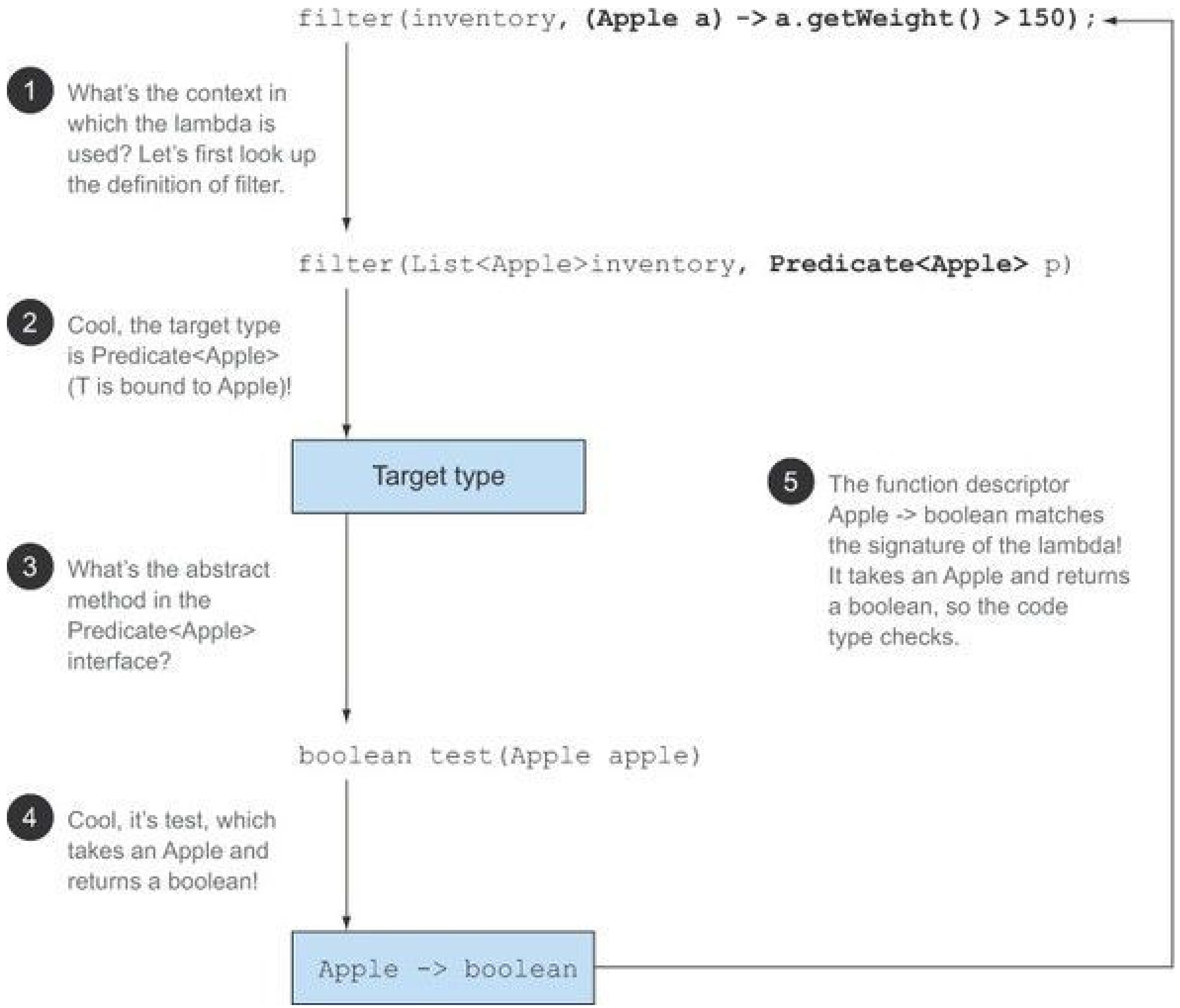


```
@FunctionalInterface
public interface Function<T, R>{
    R apply(T t);
}

public static <T, R> List<R> map(List<T> list,
                                 Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T s: list){
        result.add(f.apply(s));
    }
    return result;
}
// [7, 2, 6]
List<Integer> l = map(
                      Arrays.asList("lambdas","in","action"),
                      (String s) -> s.length()
               );
```

The lambda is the implementation for the `apply` method of `Function`.

Type checking

The type of a lambda is deduced from the context in which the lambda is used. The type expected for the lambda expression inside the context is called the *target type*.


filter(inventory, (Apple a) -> a.getWeight() > 150);
1 What's the context in which the lambda is used? Let's first look up the definition of filter.
filter(List<Apple>inventory, Predicate<Apple> p)
2 Cool, the target type is Predicate<Apple> (T is bound to Apple)!
Target type
3 What's the abstract method in the Predicate<Apple> interface?
boolean test(Apple apple)
4 Cool, it's test, which takes an Apple and returns a boolean!
Apple -> boolean
5 The function descriptor Apple -> boolean matches the signature of the lambda! It takes an Apple and returns a boolean, so the code type checks.

Lambda Target Typing

Because of the idea of target typing, the same lambda expression can be associated with different functional interfaces if they have a compatible abstract method signature.

The same lambda can therefore be used with multiple different functional interfaces:

```
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());

BiFunction<Apple, Apple, Integer> c3 = (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Special void-compatibility rule

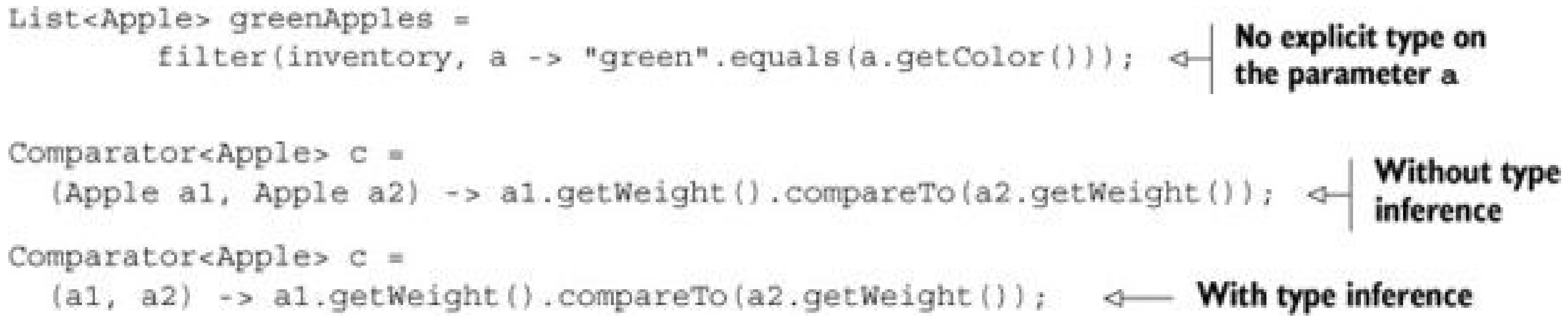
If a lambda has a statement expression as its body, it’s compatible with a function descriptor that returns void (provided the parameter list is compatible too).

```
// Predicate has a boolean return
Predicate<String> p = s -> list.add(s);

// Consumer has a void return
Consumer<String> b = s -> list.add(s);
```

Type inference

The Java compiler deduces what functional interface to associate with a lambda expression from its surrounding context (the target type)

It can also deduce an appropriate signature for the lambda because the function descriptor is available through the target type.



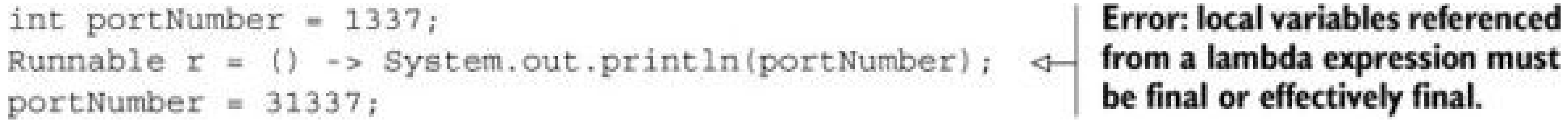
```
List<Apple> greenApples =
        filter(inventory, a -> "green".equals(a.getColor()));

Comparator<Apple> c =
   (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
Comparator<Apple> c =
   (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

No explicit type on the parameter a
Without type inference
With type inference

Using local variables

When lambda expressions contain references to free variables (variables that aren’t the parameters and defined in an outer scope), they’re called capturing lambdas.

Lambdas are allowed to capture (that is, to reference in their bodies) instance variables and static variables without restrictions. But local variables have to be explicitly declared final or are effectively final.



```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
portNumber = 31337;
```

Error: local variables referenced from a lambda expression must be final or effectively final.

Closure

A *closure* is an instance of a function that can reference nonlocal variables of that function with no restrictions. When a closure is passed as argument to another function, it can also access and modify variables defined outside its scope.

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*.

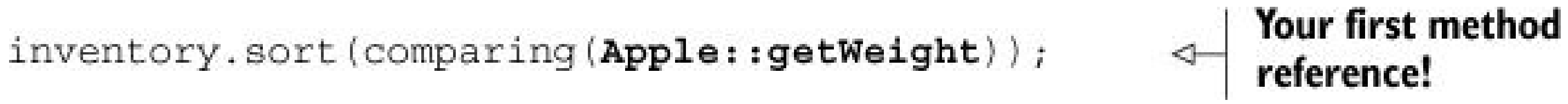
Those variables have to be implicitly final: this restriction exists because local variables live on the stack and are implicitly confined to the thread they’re in. Allowing capture of mutable local variables opens new thread-unsafe possibilities, which are undesirable (instance variables are fine because they live on the heap, which is shared across threads).

Method references

Method references let you reuse existing method definitions and pass them just like lambdas, when lambdas actually call *one single method*.

Let’s look at the Sorting Apples Example.

Using a method reference and java.util.Comparator.comparing:



```
inventory.sort(comparing(Apple::getWeight));
```

Your first method reference!

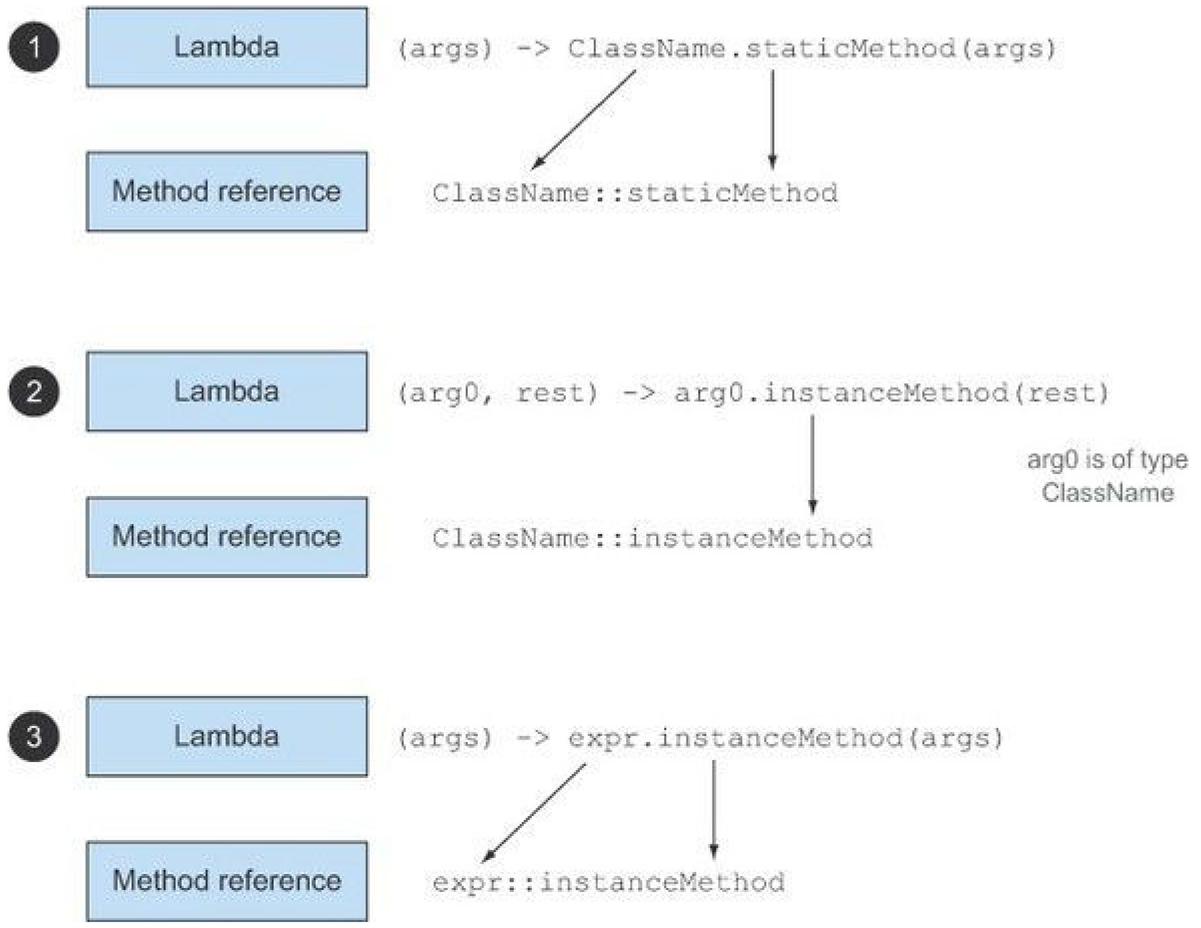
Apple::getWeight is shorthand for the lambda expression (Apple a) -> a.getWeight().

comparing is **<Apple, Integer> Comparator<Apple> java.util.Comparator.comparing(Function<? super Apple, ? extends Integer> keyExtractor)**

Kinds of Method References

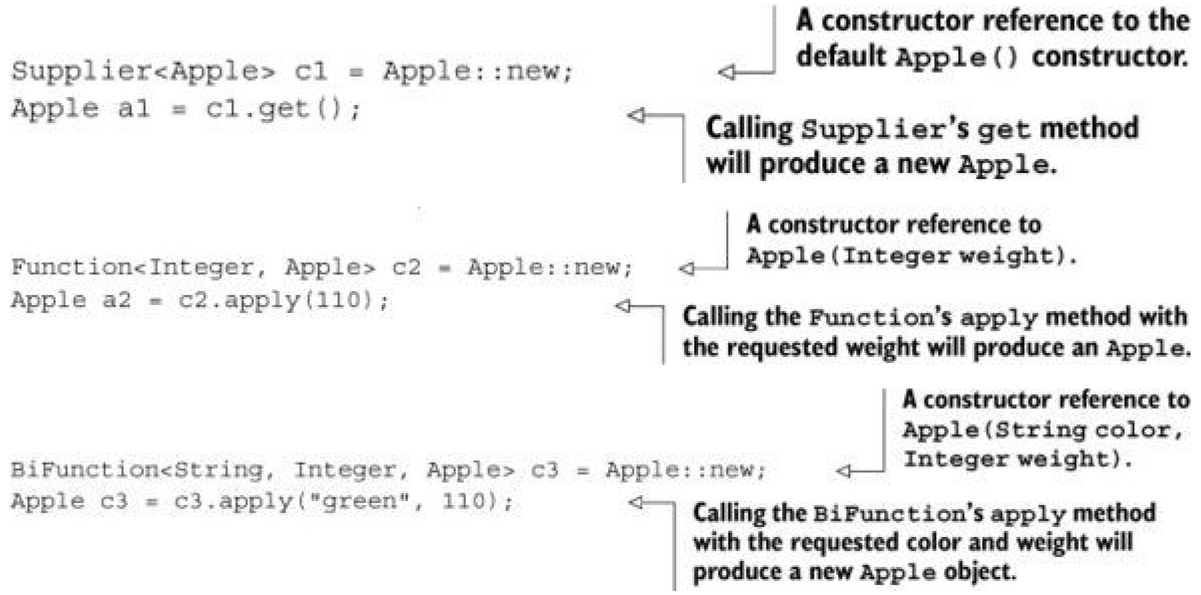
1. A method reference to a **static method** (for example, the method parseInt of Integer, written Integer::parseInt)
2. A method reference to an **instance method of an arbitrary type** (for example, the method length of a String, written String::length)
3. A method reference to an **instance method of an existing object** (for example, suppose you have a local variable anApple that holds an object of type Apple, which supports an instance method getWeight(); you can write anApple::getWeight())

Writing Method References for Lambdas


1
Lambda
(args) -> ClassName.staticMethod(args)
Method reference
ClassName::staticMethod
2
Lambda
(arg0, rest) -> arg0.instanceMethod(rest)
arg0 is of type ClassName
Method reference
ClassName::instanceMethod
3
Lambda
(args) -> expr.instanceMethod(args)
Method reference
expr::instanceMethod

Constructor References

You can create a reference to an existing constructor using its name and the keyword new



```
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`.

```
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`.

```
BiFunction<String, Integer, Apple> c3 = Apple::new;
Apple c3 = c3.apply("green", 110);
```

A constructor reference to `Apple(String color, Integer weight)`.

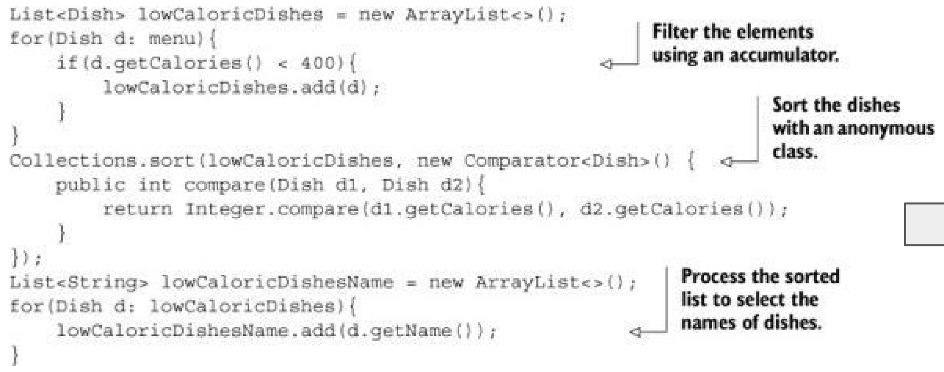
Calling the `BiFunction`'s `apply` method with the requested color and weight will produce a new `Apple` object.

This allows to create “dyamic object factories”

Streams

Streams are an update to the Java API that lets you manipulate collections of data in a declarative way. The focus is not on a collection as a “container”, but a collection as a “processor”. Streams can be processed in parallel transparently, without you having to write any multithreaded code

StreamBasic Example:

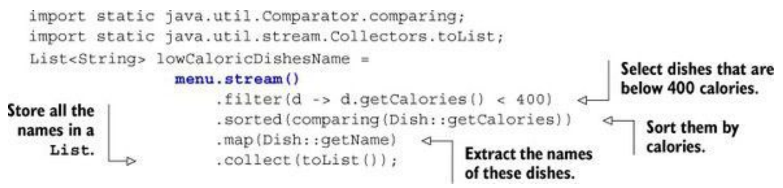


```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: menu){
    if(d.getCalories() < 400){
        lowCaloricDishes.add(d);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(), d2.getCalories());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

Filter the elements using an accumulator.

Sort the dishes with an anonymous class.

Process the sorted list to select the names of dishes.



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

Select dishes that are below 400 calories.

Sort them by calories.

Extract the names of these dishes.

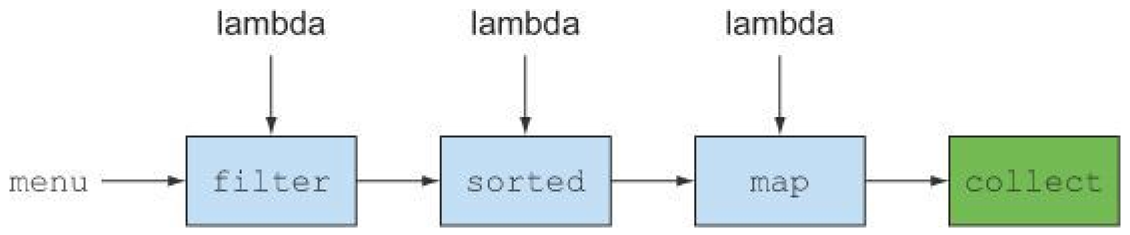
Store all the names in a List.

Benefits of Streams

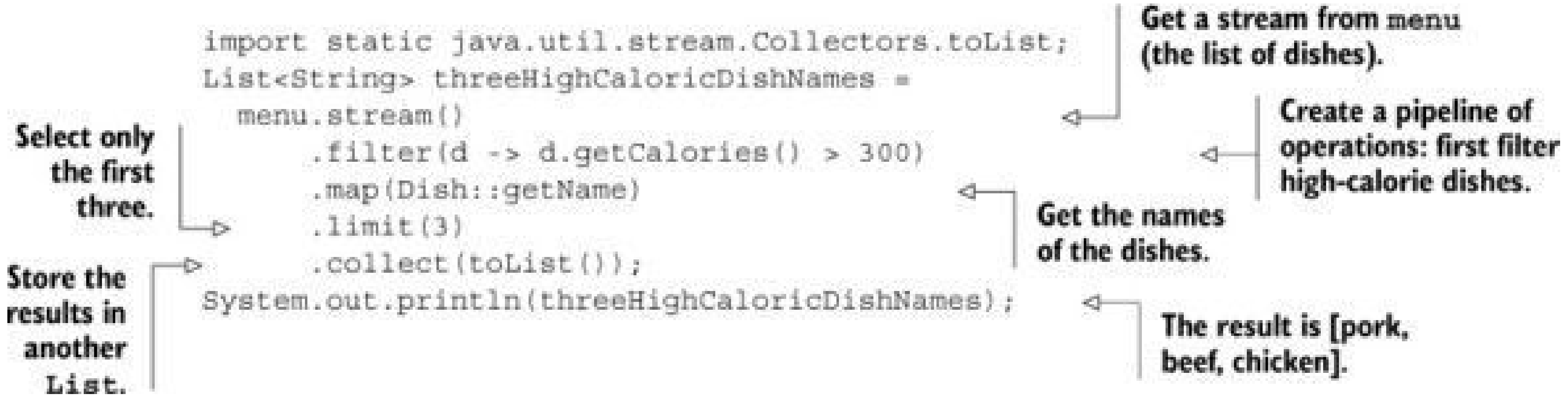
The code is written in a declarative way: you specify what you want to achieve as opposed to specifying how to implement an operation

You chain together several building-block operations to express a complicated data processing pipeline

The processing can be inherently parallelized (given certain conditions)



Recipe for building Streams



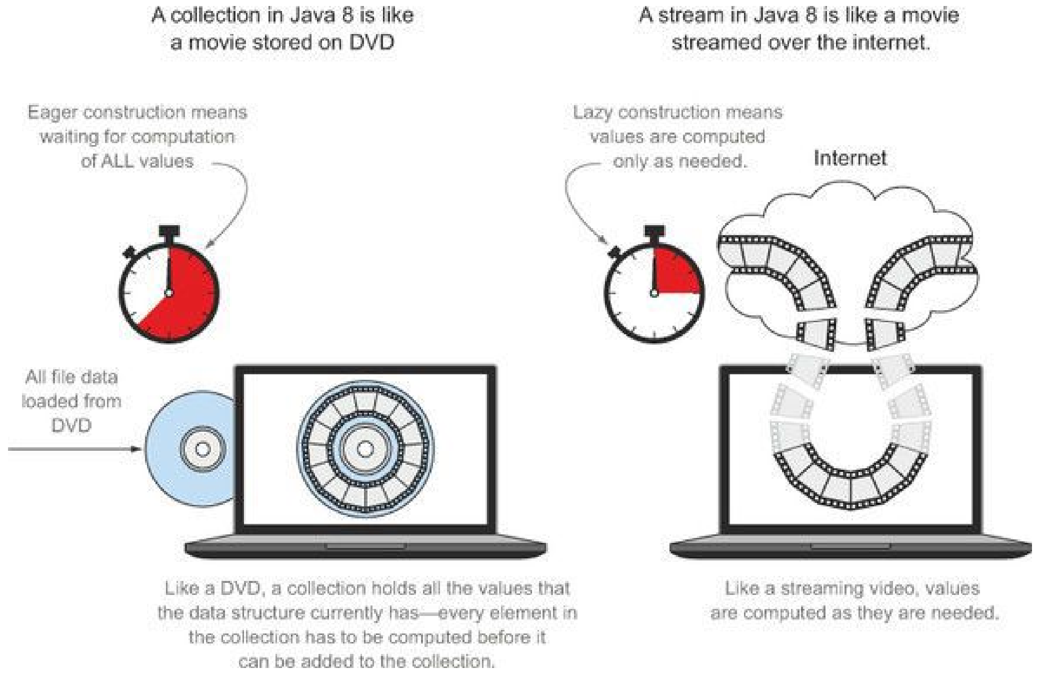
```
import static java.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames =
  menu.stream()
      .filter(d -> d.getCalories() > 300)
      .map(Dish::getName)
      .limit(3)
      .collect(toList());
System.out.println(threeHighCaloricDishNames);
```

Get a stream from menu (the list of dishes).
Create a pipeline of operations: first filter high-calorie dishes.
Get the names of the dishes.
Select only the first three.
Store the results in another List.
The result is [pork, beef, chicken].

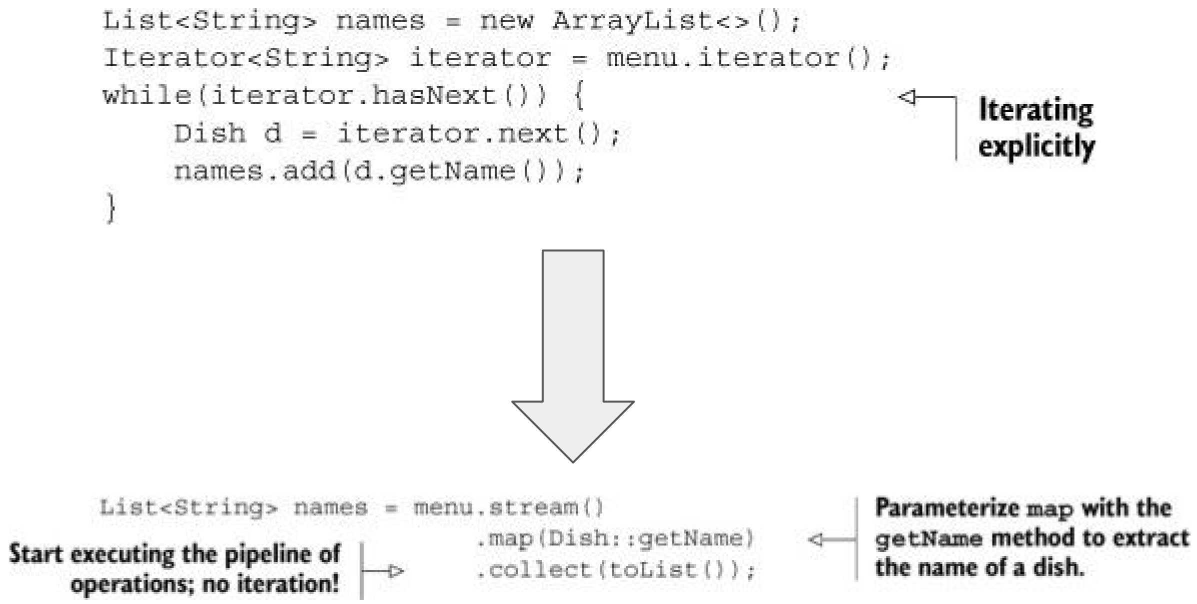
Streams vs Collections

In streams the focus is on “just in time” processing, not acting as a container

A stream is traversable *only once* (Streamvscollection example)


A collection in Java 8 is like a movie stored on DVD
A stream in Java 8 is like a movie streamed over the internet.
Eager construction means waiting for computation of ALL values
Lazy construction means values are computed only as needed.
Internet
All file data loaded from DVD
Like a DVD, a collection holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.
Like a streaming video, values are computed as they are needed.

External vs Internal Iteration



```
List<String> names = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish d = iterator.next();
    names.add(d.getName());
}
```

Iterating explicitly

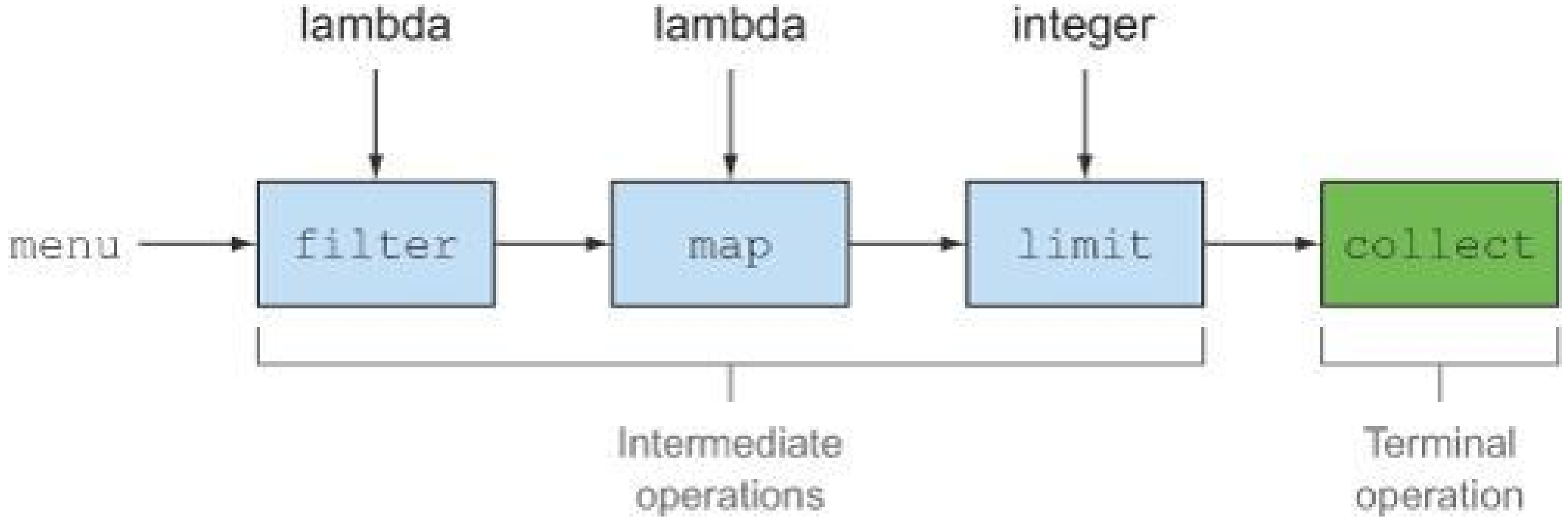
```
List<String> names = menu.stream()
                         .map(Dish::getName)
                         .collect(toList());
```

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

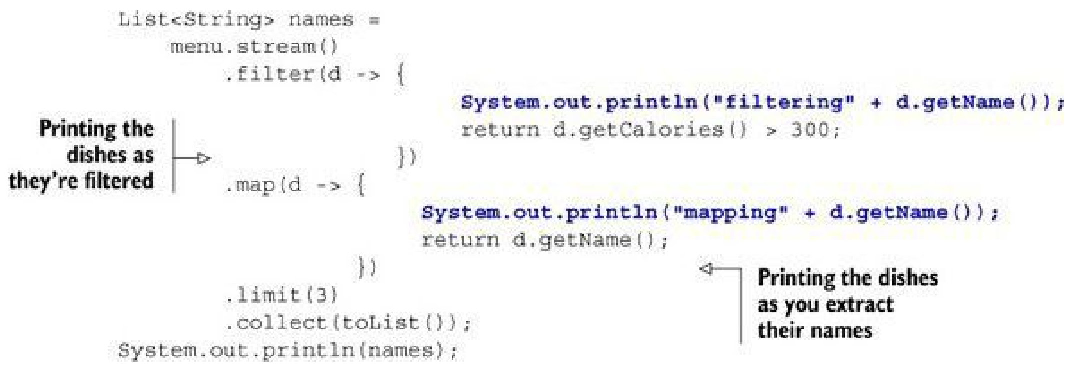
Start executing the pipeline of operations; no iteration!

Stream Operations

Intermediate operations and terminal operations



Stream Structure



```
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

Laziness execution
Short-circuit operation
Loop fusion - what is the output of this code?
Terminal operations produce a result from a stream pipeline

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

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>`		

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.

General Operations with Streams

- Filtering and Slicing (example) - filter, distinct, limit, skip
- Mapping and Flattening (example) - map, flatMap
- Finding and Matching (example) - anyMatch, allMatch, nonMatch, findAny, findFirst
- Reducing (example)

Exercise

Given two lists of numbers, how would you return all pairs of numbers? For example, given a list [1, 2, 3] and a list [3, 4] you should return [(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)].

For simplicity, you can represent a pair as an array with two elements.

Return only pairs whose sum is divisible by 3; for example, (2, 4) and (3, 3) are valid

Parallelism, Stateless Operations, Bounded and Unbounded Streams

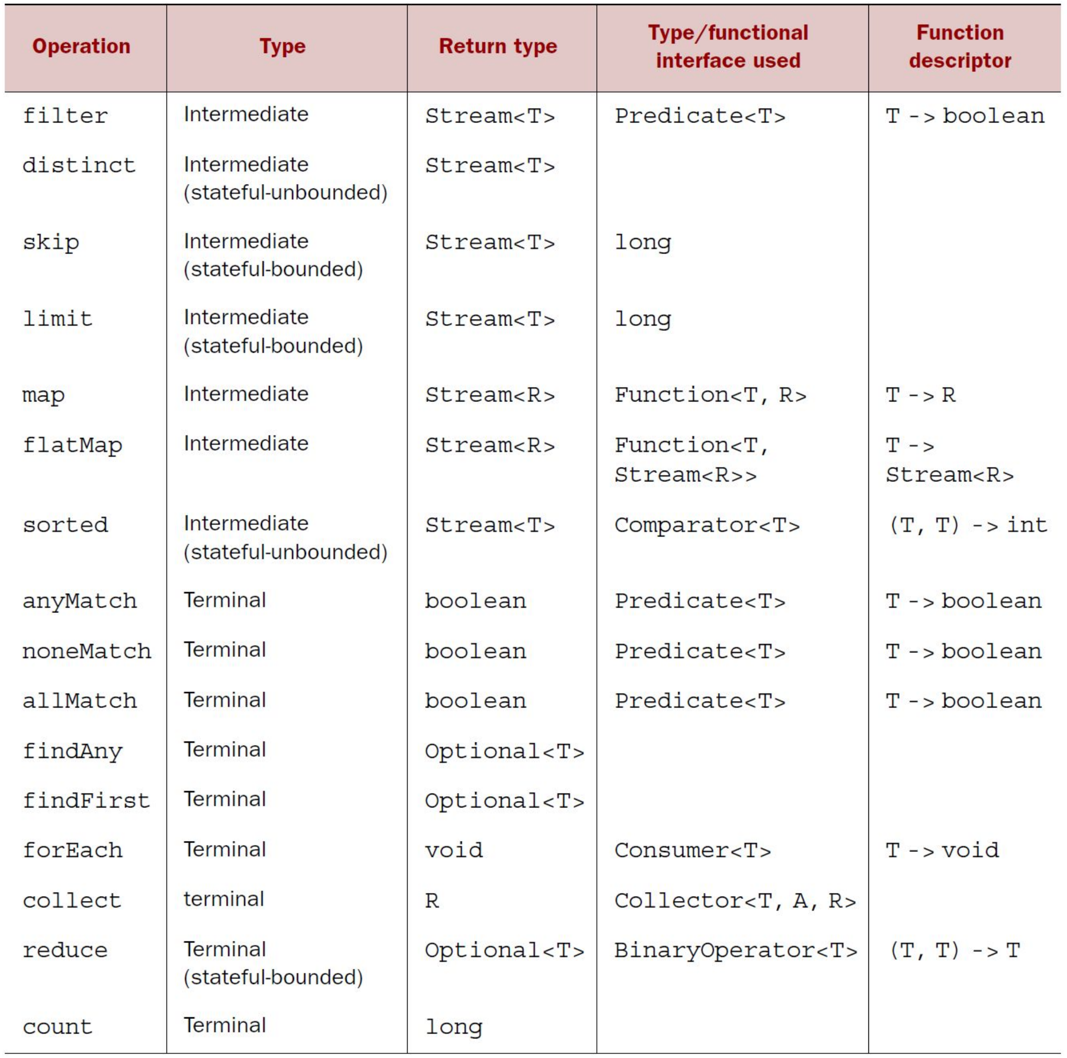
Using internal iteration, the internal implementation can perform the reduce operation in parallel

Iterative summation via external iteration, in contrast, involves shared updates to a sum variable, which doesn't parallelize gracefully. You have to use synchronization, and thread contention prevents any substantial parallelism

To do real parallelism, a new pattern is necessary: partition the input, sum the partitions, and combine the sums. To achieve this, the lambda passed to reduce *can't change state* (stateless operations) (for example, instance variables), and the *operation needs to be associative* so it can be executed in any order.

But 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. The internal state is of *bounded* size no matter how many elements are in the stream being processed. In other cases, the state is unbounded: all the unknown size has to be gathered to perform the operation (e.g. sort, distinct)

Overview of Stream 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`		

The Trading Exercise

Using the Transaction and Trader objects given in the example:

1. Find all transactions in the year 2011 and sort them by value (small to high).
2. What are all the unique cities where the traders work?
3. Find all traders from Cambridge and sort them by name.
4. Return a string of all traders' names sorted alphabetically.
5. Are any traders based in Milan?
6. Print all transactions' values from the traders living in Cambridge.
7. What's the highest value of all the transactions?
8. Find the transaction with the smallest value.

```
Trader raoul = new Trader("Raoul", "Cambridge");
        Trader mario = new Trader("Mario","Milan");
        Trader alan = new Trader("Alan","Cambridge");
        Trader brian = new Trader("Brian","Cambridge");

                List<Transaction> transactions = Arrays.asList(
            new Transaction(brian, 2011, 300),
            new Transaction(raoul, 2012, 1000),
            new Transaction(raoul, 2011, 400),
            new Transaction(mario, 2012, 710),
            new Transaction(mario, 2012, 700),
            new Transaction(alan, 2012, 950)
          );
```

Creating Streams

Streams can be created in various way (see examples):

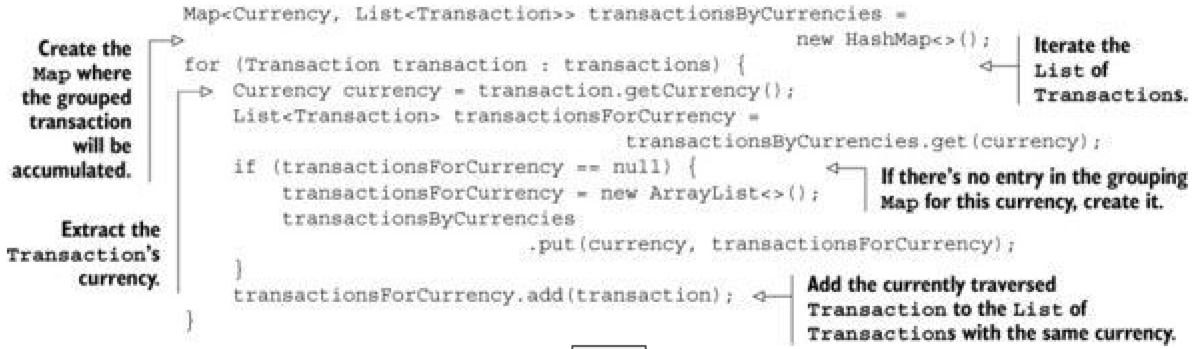
- From values - Stream.of()
- From arrays
- From files
- Stream.iterate
- Stream.generate
- Using a Supplier for a Stream

Collectors

Collectors implement a general collect interface, which allows for better performance and parallelism in the terminal stage (reduce is immutable, collect is mutable). Collectors can, for example:

- Group a list of transactions by currency to obtain the sum of the values of all transactions with that currency (returning a Map<Currency, Integer>)
- Partition a list of transactions into two groups: expensive and not expensive (returning a Map<Boolean, List<Transaction>>)
- Create multilevel groupings such as grouping transactions by cities and then further categorizing by whether they're expensive or not (returning a Map<String, Map<Boolean, List<Transaction>>>)

Collectors - Grouping



```
Map<Currency, List<Transaction>> transactionsByCurrencies =
                                                   new HashMap<>();
for (Transaction transaction : transactions) {
    Currency currency = transaction.getCurrency();
    List<Transaction> transactionsForCurrency =
                               transactionsByCurrencies.get(currency);
    if (transactionsForCurrency == null) {
        transactionsForCurrency = new ArrayList<>();
        transactionsByCurrencies
                          .put(currency, transactionsForCurrency);
    }
    transactionsForCurrency.add(transaction);
}
```

Create the Map where the grouped transaction will be accumulated.

Iterate the List of Transactions.

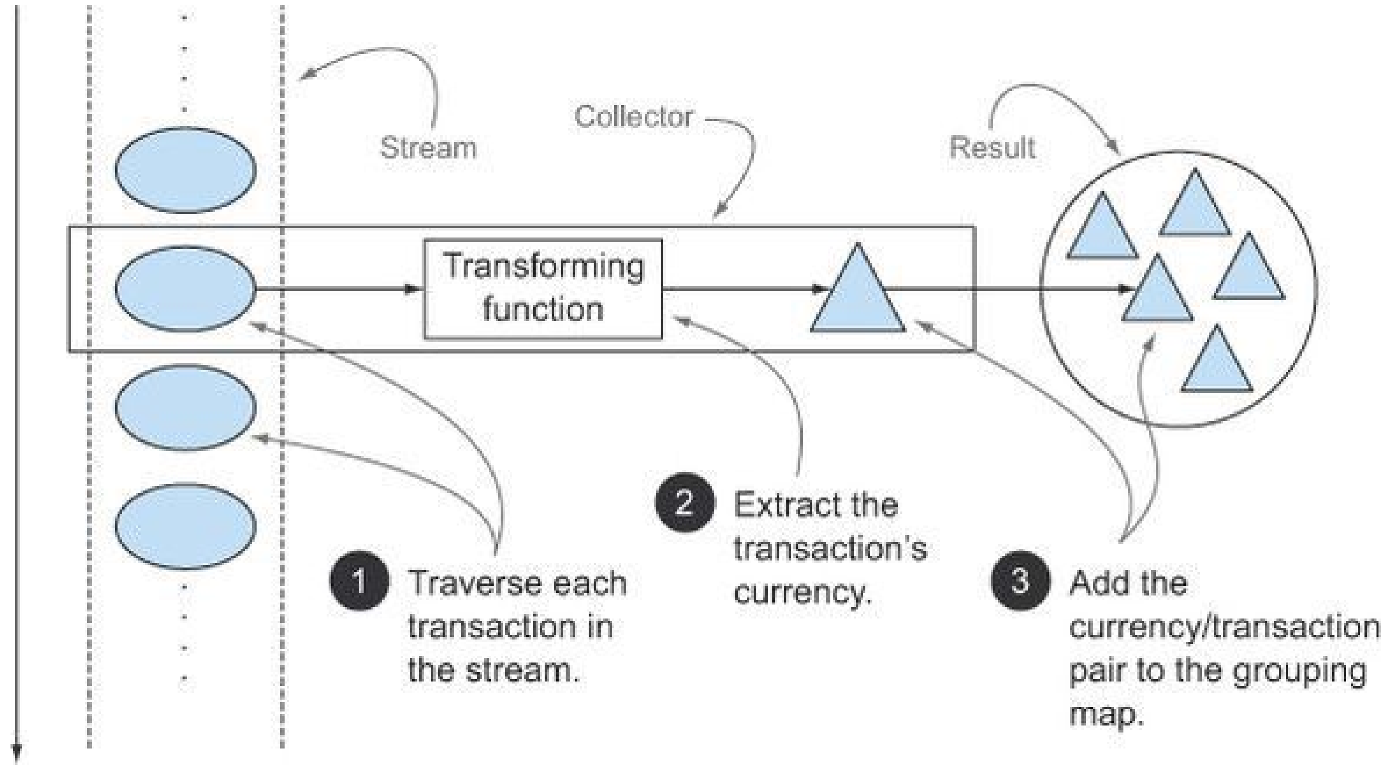
Extract the Transaction's currency.

If there's no entry in the grouping Map for this currency, create it.

Add the currently traversed Transaction to the List of Transactions with the same currency.

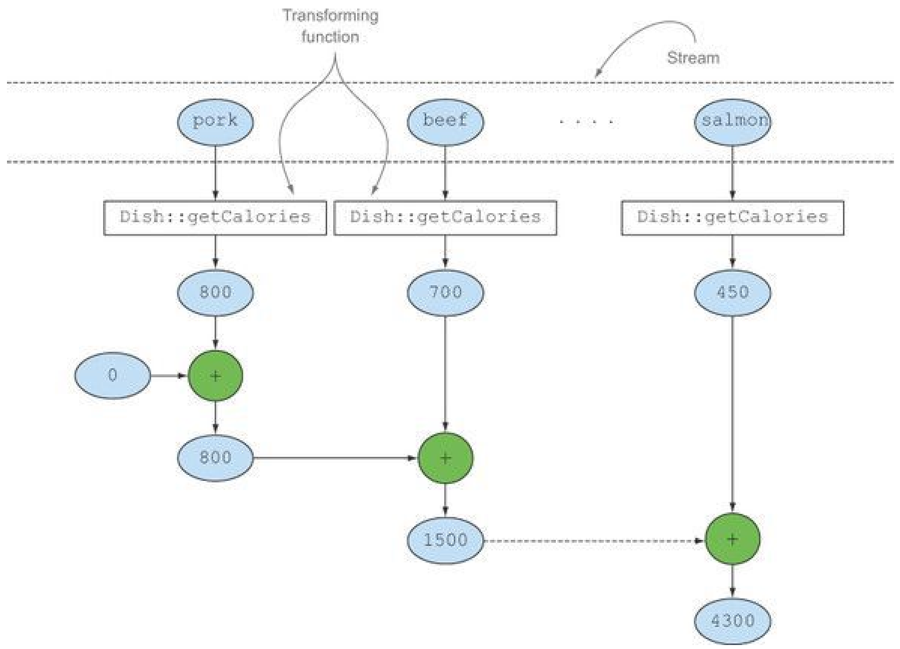
Map<Currency, List<Transaction>> transactionsByCurrencies =
transactions.stream().collect(groupingBy(Transaction::getCurrency));

Collectors as general reduction methods


Stream
Collector
Result
Transforming function
1 Traverse each transaction in the stream.
2 Extract the transaction's currency.
3 Add the currency/transaction pair to the grouping map.

Summarizing, Reducing, Joining, Grouping, Partitioning

- counting() -- Summarize example
- summingInt()
- averagingInt()
- summarizingInt()
- joining()
- maxBy(), minBy()
- reducing() -- Reducing example
- groupingBy() -- Grouping example
- collectingAndThen()
- partitioningBy() -- Partitioning example



Collector Interface

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

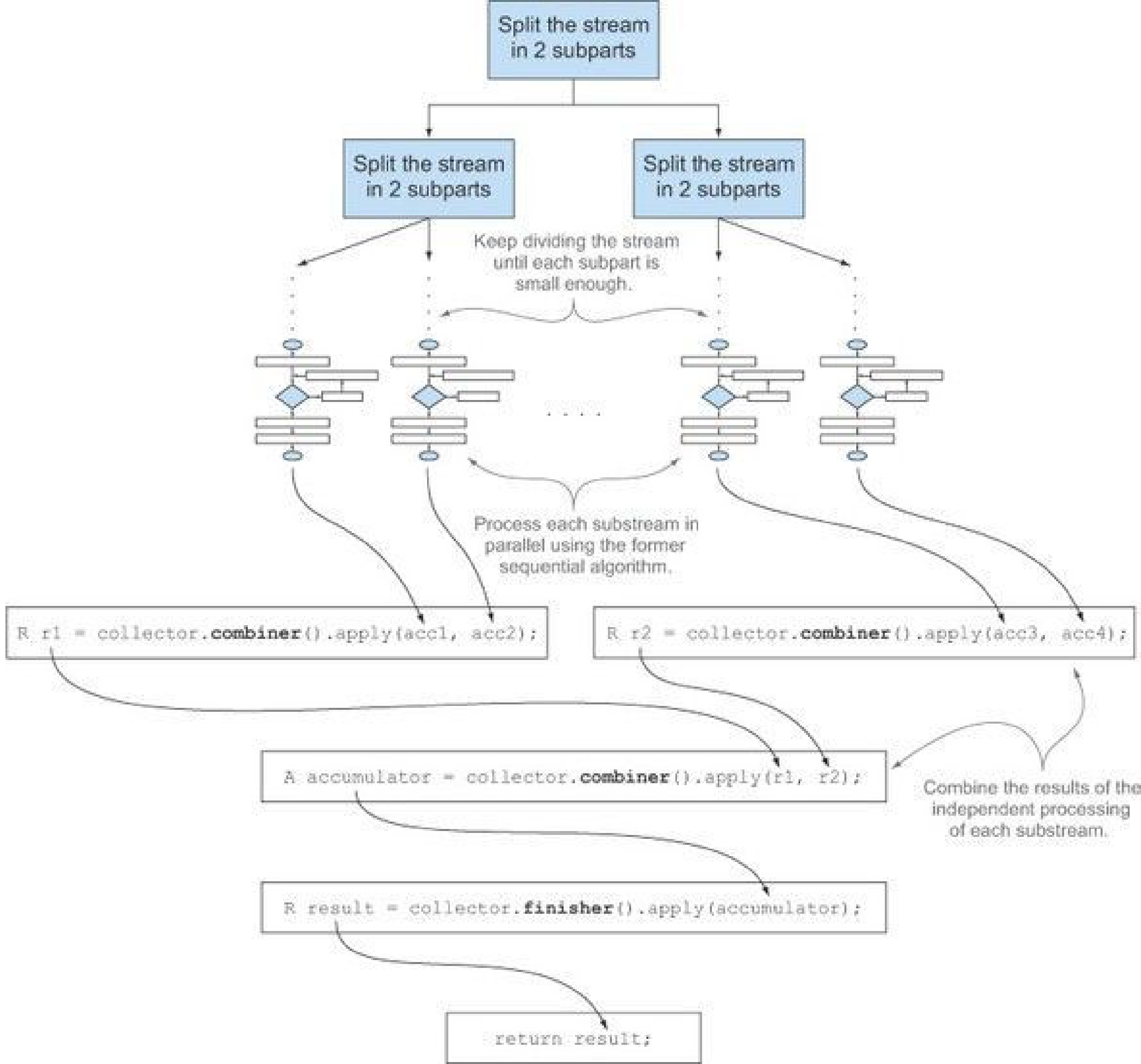
The Collector interface consists of a set of methods that provide a blueprint for how to implement specific reduction operations (Example: ToListCollector)

- T is the generic type of the items in the stream to be collected.
- A is the type of the accumulator, the object on which the partial result will be accumulated during the collection process.
- R is the type of the object (typically, but not always, the collection) resulting from the collect operation.

Components of the Collector Interface

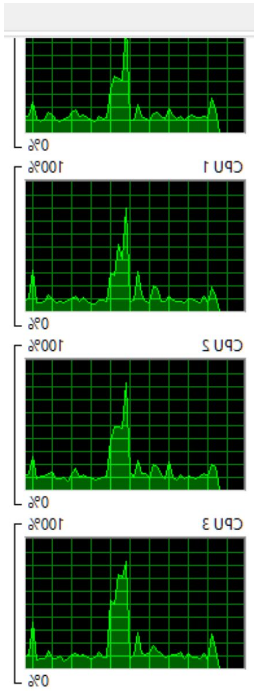
- The supplier method has to return a Supplier of an empty result—a parameterless function that when invoked creates an instance of an empty accumulator used during the collection process.
- The accumulator method returns the function that performs the reduction operation. The function returns void because the accumulator is modified in place.
- The finisher method has to return a function that’s invoked at the end of the accumulation, in order to transform the accumulator object into the final result.
- The combiner method defines how the accumulators resulting from the reduction of different subparts of the stream are combined when the subparts are processed in parallel (allows a parallel reduction of the stream).
- Characteristics, returns an immutable set providing hints about whether the stream can be reduced in parallel and which optimizations are valid when doing so. (UNORDERED CONCURRENT)

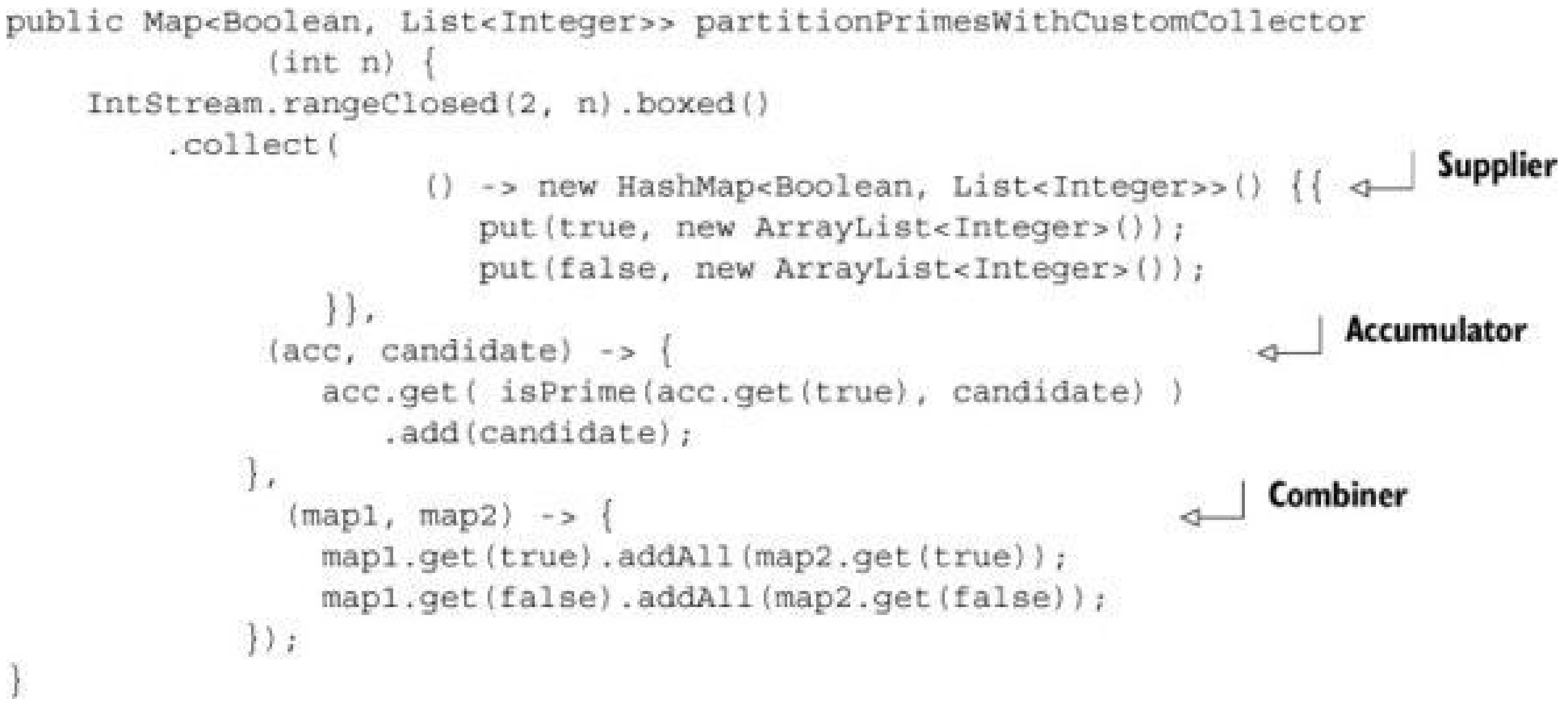
Collector Parallelization



Split the stream in 2 subparts
Split the stream in 2 subparts
Split the stream in 2 subparts
Keep dividing the stream until each subpart is small enough.
Process each substream in parallel using the former sequential algorithm.
R r1 = collector.combiner().apply(acc1, acc2);
R r2 = collector.combiner().apply(acc3, acc4);
A accumulator = collector.combiner().apply(r1, r2);
Combine the results of the independent processing of each substream.
R result = collector.finisher().apply(accumulator);
return result;

Prime Numbers Example





```
public Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector
                 (int n) {
     IntStream.rangeClosed(2, n).boxed()
          .collect(
                    () -> new HashMap<Boolean, List<Integer>>() {{
                        put(true, new ArrayList<Integer>());
                        put(false, new ArrayList<Integer>());
                }},
                 (acc, candidate) -> {
                     acc.get( isPrime(acc.get(true), candidate) )
                         .add(candidate);
                },
                  (map1, map2) -> {
                     map1.get(true).addAll(map2.get(true));
                     map1.get(false).addAll(map2.get(false));
                });
}
```

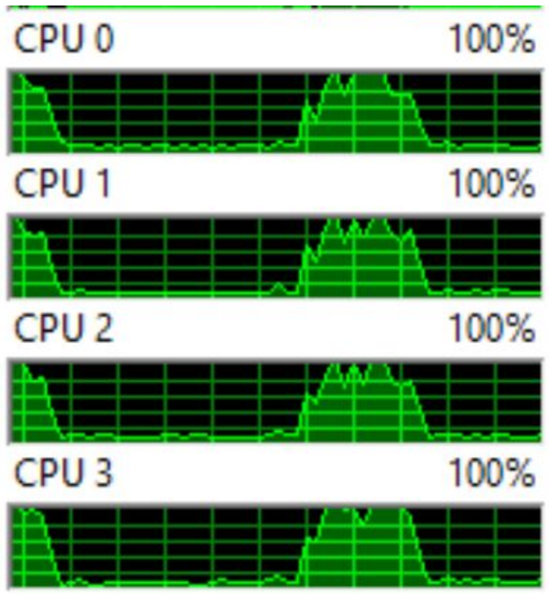
Supplier
Accumulator
Combiner

Parallel Streams

It’s possible to turn a collection into a parallel stream by invoking the method parallel() on the collection source. A parallel stream is a stream thatsplits its elements into multiple chunks, processing each chunk with a different thread. Thus, you can automatically partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy.

Not always parallelization means improvement. In the parallel example, the iterative version using a traditional for loop runs much faster than parallel() because it works at a much lower level and, more important, doesn’t need to perform any boxing or unboxing of the primitive values.

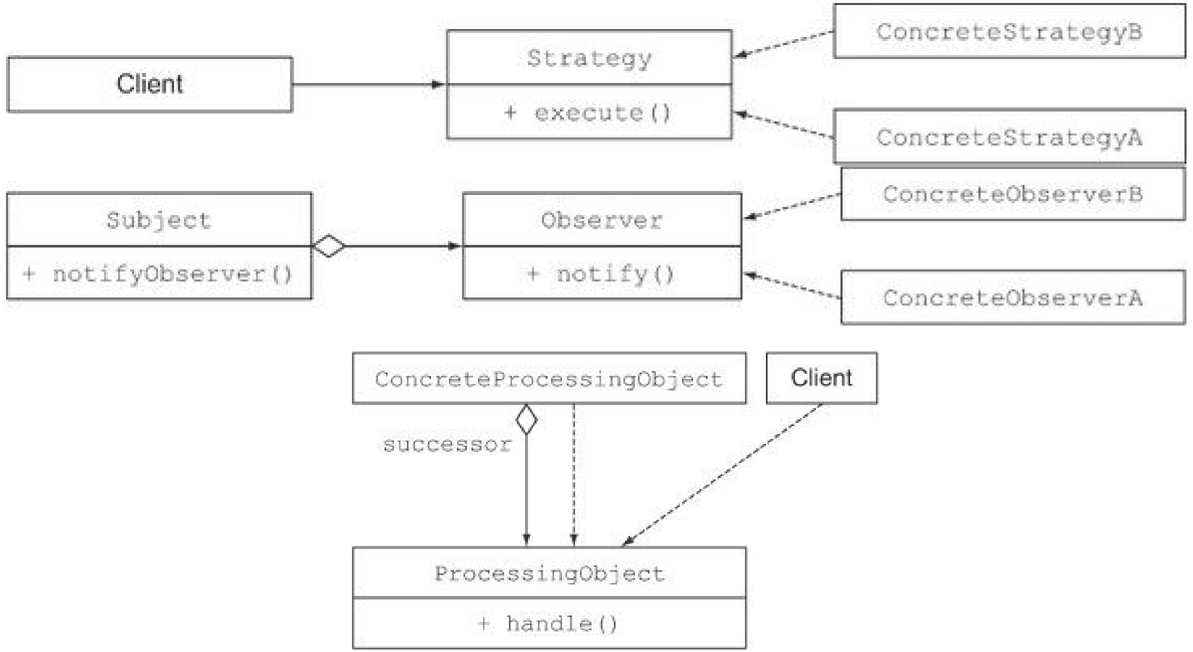
Using a native stream without unboxing yields comparable performance.



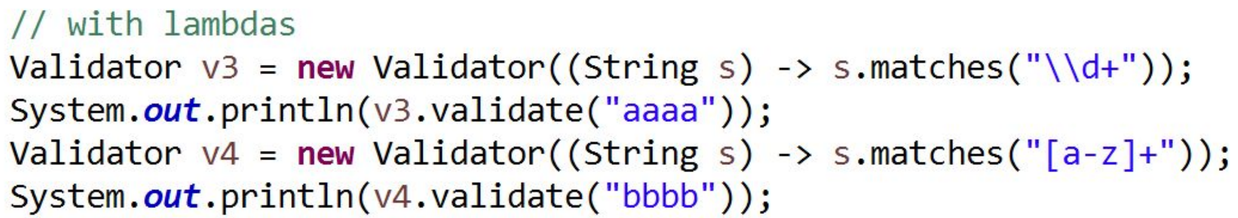
Rethinking OOP Patterns with FP

Many existing object-oriented design patterns can be made redundant or written in a more concise way using lambda expressions. For example these five design patterns:

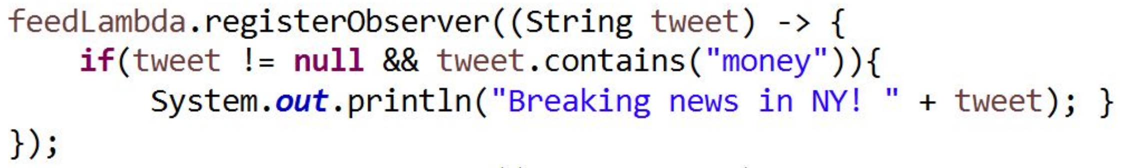
- Strategy
- Template method
- Observer
- Chain of responsibility
- Factory



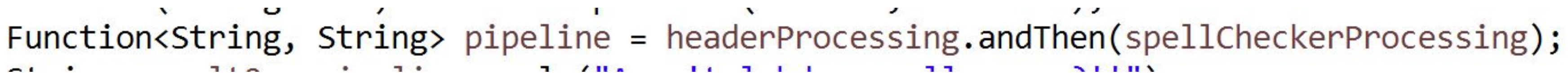
Rethinking OOP Patterns with FP



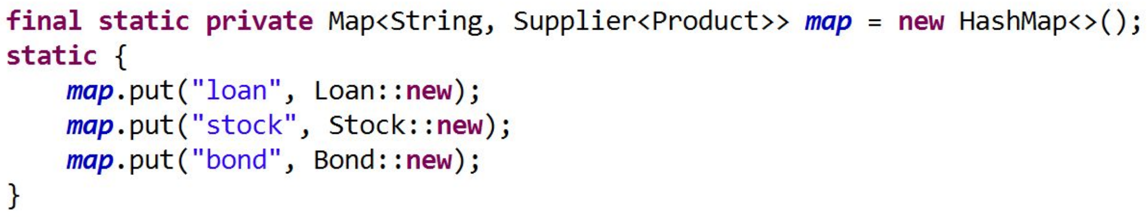
```java
// with lambdas
Validator v3 = new Validator((String s) -> s.matches("\\d+"));
System.out.println(v3.validate("aaaa"));
Validator v4 = new Validator((String s) -> s.matches("[a-z]+"));
System.out.println(v4.validate("bbbb"));
```



```java
feedLambda.registerObserver((String tweet) -> {
    if(tweet != null && tweet.contains("money")){
        System.out.println("Breaking news in NY! " + tweet); }
});
```



```java
Function<String, String> pipeline = headerProcessing.andThen(spellCheckerProcessing);
```



```java
final static private Map<String, Supplier<Product>> map = new HashMap<>();
static {
    map.put("loan", Loan::new);
    map.put("stock", Stock::new);
    map.put("bond", Bond::new);
}
```

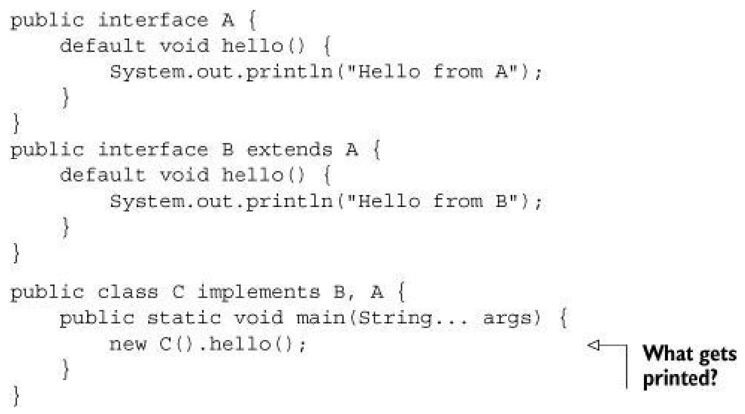
Default Methods

Java interfaces group related methods together into a contract. Any class that implements an interface must provide an implementation for each method defined by the interface or inherit the implementation from a superclass. But this causes a problem when library designers need to update an interface to add a new method.

Interfaces in Java 8 can now declare methods with implementation code; this can happen in two ways:

- First, Java 8 allows static methods inside interfaces.
- Second, Java 8 introduces a new feature called default methods that allows you to provide a default implementation for methods in an interface.

Resolution rules

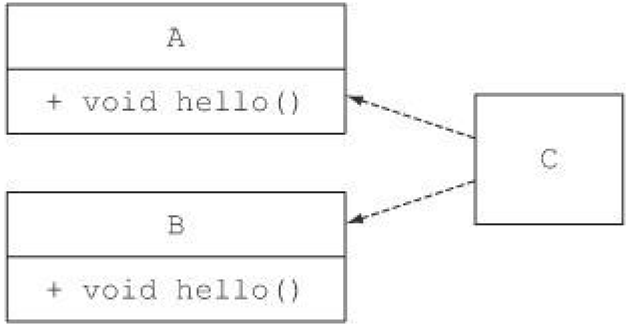


```
public interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}
public interface B extends A {
    default void hello() {
        System.out.println("Hello from B");
    }
}

public class C implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

There are three rules to follow when a class inherits a method with the same signature from multiple places (such as another class or interface):

1. Classes always win. A method declaration in the class or a superclass takes priority over any default method declaration.
2. Otherwise, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected.
3. Finally, if the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly.

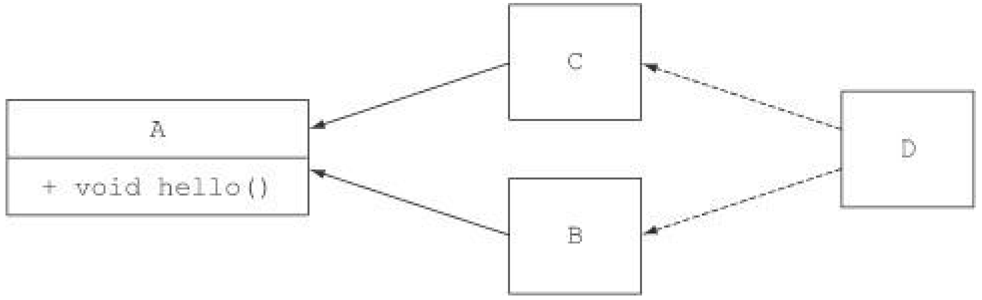
Interface Disambiguation



```
public class C implements B, A {
    void hello(){
        B.super.hello();
    }
}
```

Explicitly choosing to call the method from interface `B`

Diamonds aren't a programmer's best friends



```
public interface A{
    default void hello(){
        System.out.println("Hello from A");
    }
}

public interface B extends A { }

public interface C extends A { }

public class D implements B, C {
    public static void main(String... args) {
        new D().hello();
    }
}
```

What gets printed?

- What if B has a hello() as well?
- What if both B and C have a different hello()?

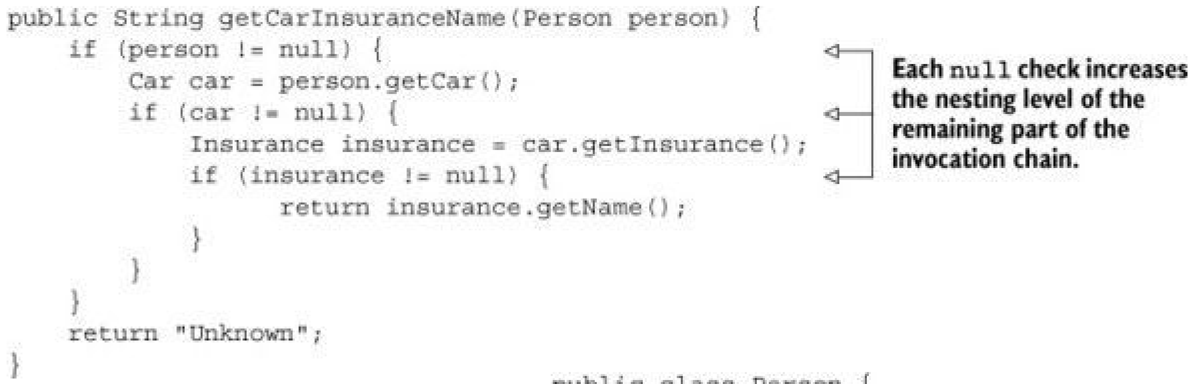
Interfaces vs Abstract Classes

So what’s the difference between an abstract class and an interface? They both can contain abstract methods and methods with a body.

First, a class can extend only from one abstract class, but a class can implement multiple interfaces.

Second, an abstract class can enforce a common state through instance variables (fields). An interface can’t have instance variables.

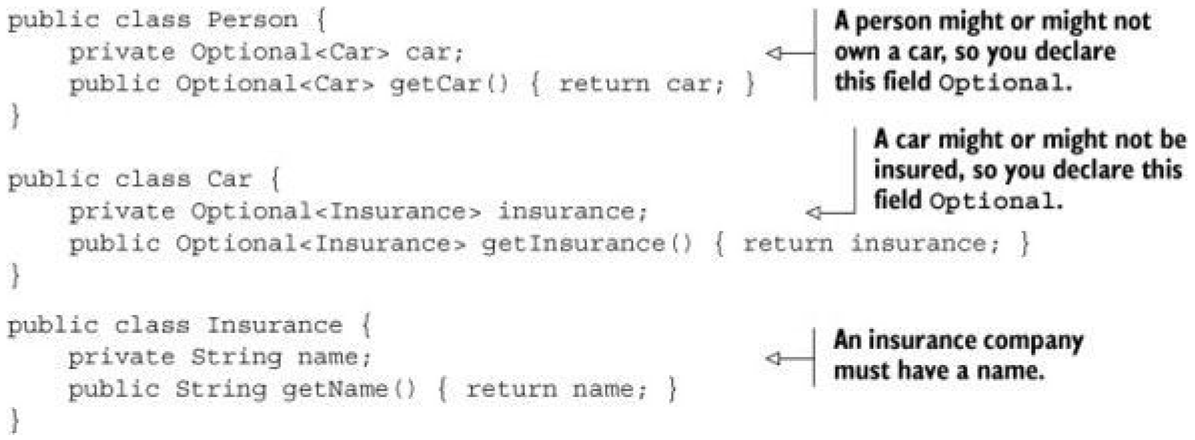
Optionals



```java
public String getCarInsuranceName(Person person) {
    if (person != null) {
        Car car = person.getCar();
        if (car != null) {
            Insurance insurance = car.getInsurance();
            if (insurance != null) {
                  return insurance.getName();
            }
        }
    }
    return "Unknown";
}
```

Each null check increases the nesting level of the remaining part of the invocation chain.

Java 8 introduces a new class called java.util.Optional<T> that encapsulates an optional value.



```java
public class Person {
    private Optional<Car> car;
    public Optional<Car> getCar() { return car; }
}

public class Car {
    private Optional<Insurance> insurance;
    public Optional<Insurance> getInsurance() { return insurance; }
}

public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

A person might or might not own a car, so you declare this field Optional.

A car might or might not be insured, so you declare this field Optional.

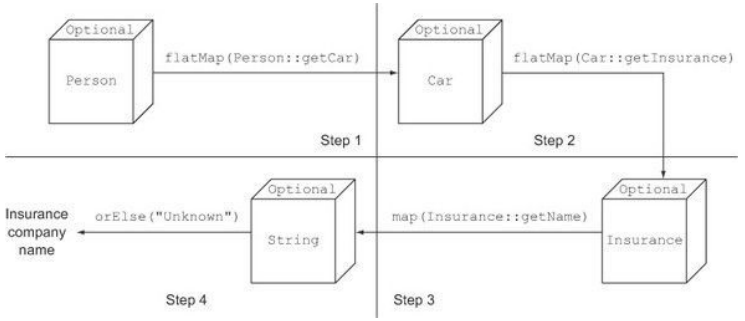
An insurance company must have a name.

Common patterns with Optional

```
Optional<Car> optCar = Optional.of(car);

Optional<Car> optCar = Optional.ofNullable(car);
```

```
String name = null;
if(insurance != null){
name = insurance.getName();
}
```

```
Optional<Insurance> optInsurance = Optional.ofNullable(insurance);
Optional<String> name = optInsurance.map(Insurance::getName);
```



```
public String getCarInsuranceName(Optional<Person> person) {
    return person.flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```