Java 8

Chap 1

The main benefit of Java 8 to a programmer is that it provides more programming tools andconcepts to solve new or existing programming problems more quickly or, more importantly, ina more concise, more easily maintainable way.

* 1. **Stream processing**

cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3

1. Unix the commands (cat, tr, sort, and tail) are executed concurrently, so that sort can be processing the first few lines before cat or tr has finished.
2. A more mechanical analogy is a car-manufacturing assembly line where a stream of cars is queued between processing stations that each take a car, modify it, and pass it on to the next station for further processing; processing at separate stations is typically concurrent even though the assembly line is physically a sequence
3. Java 8 adds a Streams API (note the uppercase *S*) in java.util.stream based on this idea; Stream<T> is a sequence of items of type T.
4. The key motivation for this is that you can now program in Java 8 at a higher level of abstraction, structuring your thoughts of turning a stream of this into a stream of that.
5. Another advantage is that Java 8 can transparently run your pipeline of Stream operations on several CPU cores on disjoint parts of the input—this is parallelism *almost for free* instead of hard work using Threads.

**Passing code to methods with behavior parameterization**

**Passing code with behaviour parameterization**

behavior parameterization : Changing the behaviour with parameters to different behaviour for different members of collection

Eg. A collection of apple, have a different behaviour for red apples and different for green apples

*Behavior parameterization* is a software development pattern that lets you handle frequent requirement changes.

Managing changing requirement :

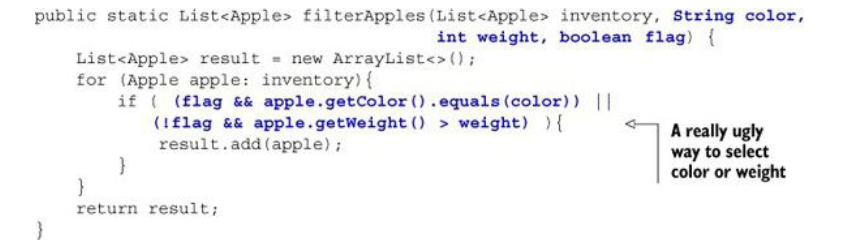
Out of collection it is possible that requirement is of filtering out list of red apple initially, green apple later, apple with weight greater then 150 later on.

Possible solutions

1. Have different method : filerredApple, filterGreeenApple
2. Pass color as parameter, filterApplesByColor(inventory, "green");

Issue : DRY – redundant code

1. Filtering with every possible attribute u ca think of.



Issue : Don’t u think its a really bad design

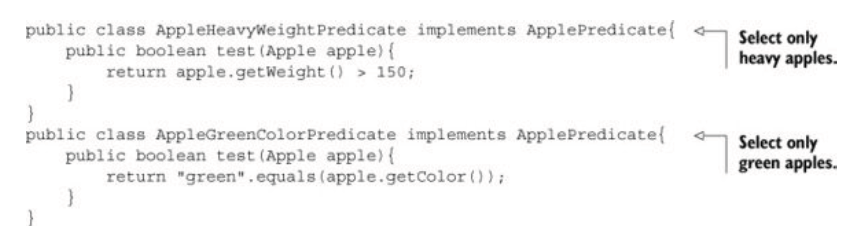
Whatif new parameter were added : size , shape, origin

**Behavior parameterization**

public interface ApplePredicate{

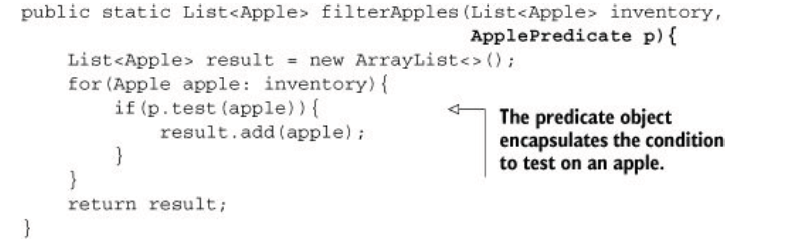
boolean test (Apple apple);

}



Above is example of strategy design pattern and is like compartor.

And filter method will look like:

****

public class AppleRedAndHeavyPredicate implements ApplePredicate{

public boolean test(Apple apple){

return "red".equals(apple.getColor())

&& apple.getWeight() > 150;

}

}

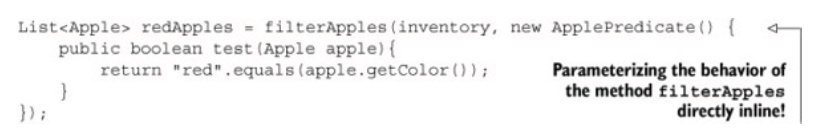
List<Apple> redAndHeavyApples =

filter(inventory, new AppleRedAndHeavyPredicate());

**2.3 Tackling verbosity**

At the moment, when you want to pass new behavior to your filterApples method, you’re forced to declare several classes that implement the ApplePredicate interface and then instantiate several ApplePredicate objects that you allocate only once.

How to improve it : **Anonymous classes**

****

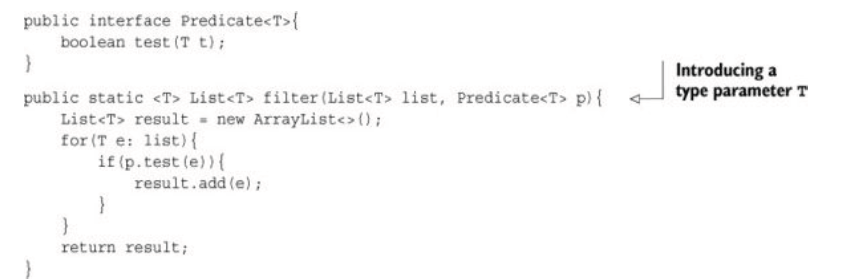
But still things can improved

**Introducing Lambda**

List<Apple> result =

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

Abstracting over List



You can now use the method filter with a List of bananas, oranges, Integers, or Strings! Here’s

an example, using lambda expressions:

List<Apple> redApples =

filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));

List<String> evenNumbers =

filter(numbers, (Integer i) -> i % 2 == 0);

Summary

This pattern lets you encapsulate a behavior (a piece of code) and parameterize

the behavior of methods by passing and using these behaviors you create (for example, different predicates for an Apple)

**Lambda Expression**

using anonymous classes to represent different behaviors is unsatisfying: it’s

verbose, which doesn’t encourage programmers to use behavior parameterization in practice

**A lambda expression is composed of parameters, an arrow, and a 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

(parameters) -> expression

or (note the curly braces for statements)

(parameters) -> { statements; }

To check if its a valid lamada or not in expression lamada types add a return in expression

For example :

(Integer i) -> return "Alan" + i;

To make it valid

(Integer i) -> {return "Alan" + i;}.

(String s) -> {"Iron Man";}

To make it valid either :

(String s) -> "Iron Man". Or

(String s) -> { return"Iron Man";}

**Functional interface**

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

Comparator, Runnable, Callable are example of functional interface

Predicate is also an example of functional interface

public interface Predicate<T>{

boolean test (T t);

}

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* (more technically speaking,

An instance of a *concrete implementation* of the functional interface). You can achieve the same thing with an anonymous inner class, although it’s clumsier:

public void process(Runnable r){

r.run();

}

process(() -> System.out.println("This is awesome!!"));

**@FunctionalInterface?**

If you explore the new Java API, you’ll notice that functional interfaces are annotated with @FunctionalInterface

**Putting lambdas into practice: the execute around pattern**

String result = processFile((BufferedReader br) ->

br.readLine() + br.readLine());

public static String processFile(BufferedReaderProcessor p) throws IOException {

...

}

@FunctionalInterface

public interface BufferedReaderProcessor {

String process(BufferedReader b) throws IOException;

}

@FunctionalInterface

public interface Predicate<T>{

boolean test(T t);

}

Predicate<T>, Consumer<T>, and Function<T, R> are example of function interface

**3.5. Type checking, type inference, and restrictions**

List<Apple> heavierThan150g =

filter(inventory, ***(Apple a) -> a.getWeight() > 150***);

The type-checking process is deconstructed as follows:

 First, you look up the declaration of the filter method.

 Second, it expects as the second formal parameter an object of type Predicate-<Apple> (the target type).

 Third, Predicate<Apple> is a functional interface defining a single abstract method called test.

 Fourth, the method test describes a function descriptor that accepts an Apple and returns a

boolean.

Finally, any actual argument to the filter method needs to match this requirement.

**Same lambda, different functional interfaces**

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

Callable<Integer> c = () -> 42;

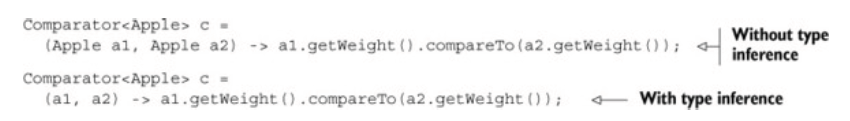
PrivilegedAction<Integer> p = () -> 42;

Type **inference**

The Java compiler deduces what functional interface to associate with a lambda expression from its surrounding context (the target type), meaning it can also deduce an appropriate signature for the lambda because the function descriptor is available through the target type.

Matlab this valid :





Variable reference :

Like anonymous class :

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. In other words, lambda expressions can capture local variables that are assigned to them only once

**Method references**

Method references let you reuse existing method definitions and pass them just like lambdas

inventory.sort((Apple a1, Apple a2)

-> a1.getWeight().compareTo(a2.getWeight()));

After (using a method reference and java.util.Comparator.comparing):



Visit : <http://www.tutorialspoint.com/java8/java8_method_references.htm>

**List**<**String**> names = **new** **LinkedList**<>();

*// ... add some names to the collection*

names.forEach(name -> **System**.**out**.println(name));

or

names.forEach(System.**out**::println);

Method reference can be used in place of lamadas. Method reference seems more complicated.

A method reference can be used to point the following types of methods

* Static methods
* Instance methods (String::length) ?
* Constructors using new operator (TreeSet::new) ?

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.

Abt 1st

Function<String, Integer> stringToInteger =

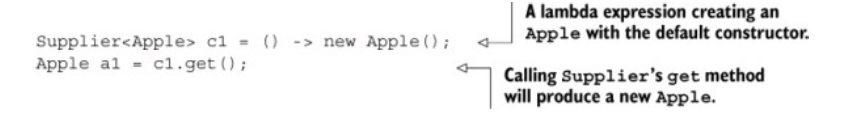
(String s) -> Integer.parseInt(s);

Function<String, Integer> stringToInteger = Integer::parseInt;

More abt 2n d

(String s) -> s.toUpperCase() can be rewritten as String::toUpperCase

Example of 3rd



To



3.7 Lamads in practice

The final solution of sorting apple problem :

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

Sort method in the list interface

void sort(Comparator<? super E> c)

Java8 workspace

AppleSorting

**Stream** :

*Streams* are an update to the Java API that lets you manipulate collections of data in a declarative way

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

the Streams API in Java 8 lets you write code that’s

 ***Declarative*—** More concise and readable

 ***Composable*—** Greater flexibility

 ***Parallelizable*—** Better performance

***Sequence of elements*—**Like a collection, a stream provides an interface to a sequenced set of

values of a specific element type

**Source** - Streams consume from a data-providing source such as collections, arrays, or I/O resources. Note that generating a stream from an ordered collection preserves the ordering

***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.

---------------------------------------------------------------------------------------

***The two other properties of streams***

***Internal iteration*—** In contrast to collections, which are iterated explicitly using an iterator,

stream operations do the iteration behind the scenes for you.

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

the data source.

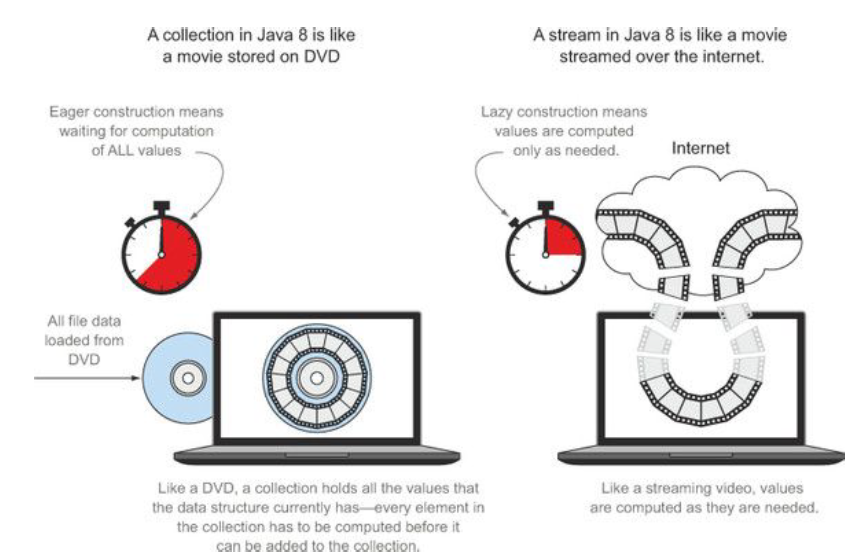
--------------------------------------------------------------------------------------------------------

Stream Vs Collections

1. stream is like a lazily constructed collection: values are computed when they’re solicited by a consumer (in management speak this is demand-driven, or even just-in-time, manufacturing)
2. In contrast, a collection is eagerly constructed (supplier-driven: fill your warehouse before you start selling, like a Christmas novelty that has a limited life).
3. 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

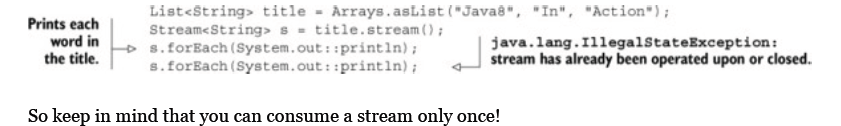
By contrast, a stream is a conceptually fixed data structure (you can’t add or remove elements from it) whose elements are *computed on demand*.



**Traversable only once**

similarly to iterators, a stream can be traversed only once. After that a stream is said

to be consumed.



**External vs. internal iteration**



Vs



**Intermediate vs. terminal operations**

Intermediate operations such as filter or sorted return another stream as the return type.

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.

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

**Working with streams**

Filtering, slicing, and matching

Filtering

List<Dish> vegetarianDishes =menu.stream()

.filter(Dish::isVegetarian)

.collect(toList());

Truncating

List<Dish> dishes = menu.stream()

.filter(d -> d.getCalories() > 300)

***.limit(3)***

.collect(toList());

Skipping

List<Dish> dishes = menu.stream()

.filter(d -> d.getCalories() > 300)

***.skip(2)***

.collect(toList());

**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.

Mapping is used for extracting the specific values from the object It has no relation with util Map function.

Its extracting data.

List<String> dishNames = menu.stream()

.map(Dish::getName)

.collect(toList());

Given a list of words, you’d like to return a list of the number of characters for each word.

List<String> words = Arrays.asList("Java8", "Lambdas", "In", "Action");

List<Integer> wordLengths = words.stream()

.map(String::length)

.collect(toList());

What if you wanted to find out the length of the name of each dish?

List<Integer> dishNameLengths = menu.stream()

.map(Dish::getName)

***.map(String::length)***

.collect(toList());

More on map vs flat Map latter

*short-circuiting* operations

if(menu.stream().anyMatch(Dish::isVegetarian)){

System.out.println("The menu is (somewhat) vegetarian friendly!!");

}

boolean isHealthy = menu.stream()

.allMatch(d -> d.getCalories() < 1000);

boolean isHealthy = menu.stream()

.noneMatch(d -> d.getCalories() >= 1000);

here’s no need to evaluate the entire expression. This is what

*short-circuiting* refers to.

In relation to streams, certain operations such as allMatch, noneMatch, findFirst, and findAny don’t need to process the whole stream to produce a result. As soon as an element is found, a result can be produced.

Optonal



**Reducing**

Summing the elements

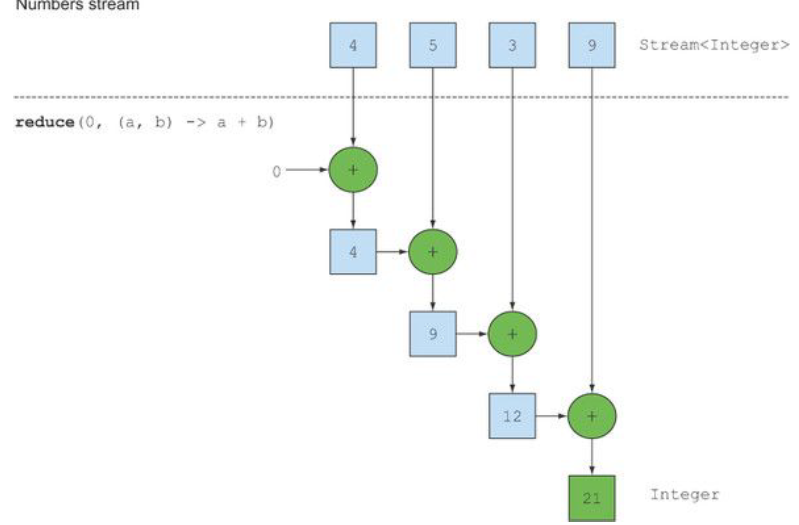
**int** sum = numbers.stream().reduce(0, (a, b) -> a + b);

reduce takes two arguments:

 An 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.



Or

Int sum = numbers.stream().reduce(0, Integer::sum);

How to find max using reduce

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

How would you count the number of dishes in a stream using the map and reduce methods

int count = menu.stream()

.map(d -> 1)

.reduce(0, (a, b) -> a + b);

A chain of map and reduce is commonly known as the map-reduce pattern,

While map is extracting the thing, reduce is summing\combining them up.

**Benefit of the reduce method and parallelism**

The benefit of using reduce compared to the step-by-step iteration summation that you wrote earlier is that the iteration is abstracted using internal iteration, which enables the internal implementation to choose to perform the reduce operation in parallel. The iterative summation example involves shared updates to a sum variable, which doesn’t parallelize gracefully

Parallelizing this computation

requires a different approach: partition the input, sum the partitions, and combine the sums

**Stream operations: stateless vs. Stateful**

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 aninternal state

But operations like reduce, sum, and max need to have internal state to accumulate the result.

The internal state is of *bounded size* no matter how many elements are in the stream being processed.

What should reversing the stream of all prime numbers do? It should return the largest prime number, which mathematics tells us doesn’t exist.) We call these operations *stateful operations*.

**Numeric streams**

**Mapping to a numeric stream**

you can use mapToInt as follows to calculate the sum of calories in the menu:





**Building streams**

**Streams from values**

Stream<String> stream = ***Stream.of("Java 8 ", "Lambdas ", "In ", "Action")***;

Arrays



Files



Chapter 6. Collecting data with streams