IMPORTS

**import** java.util.stream.Collectors;

**import** java.util.function. {Predicate, Function, Consumer, IntPredicate, Supplier, doublesupplier etc)

**import** java.util.Comparator;

import static java.util.stream.**Collectors**.\*;

**import** java.util.Optional;

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

Simple if then else in one line:

String characteristic = apple.getWeight() > 150 ? "heavy" : "light";

Comparator.comparing

inventory.sort(Comparator.*comparing*(Apple::getWeight).reversed());

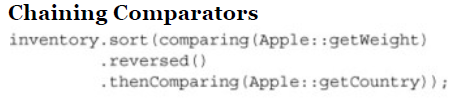
Comparator<Apple> byWeight = (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

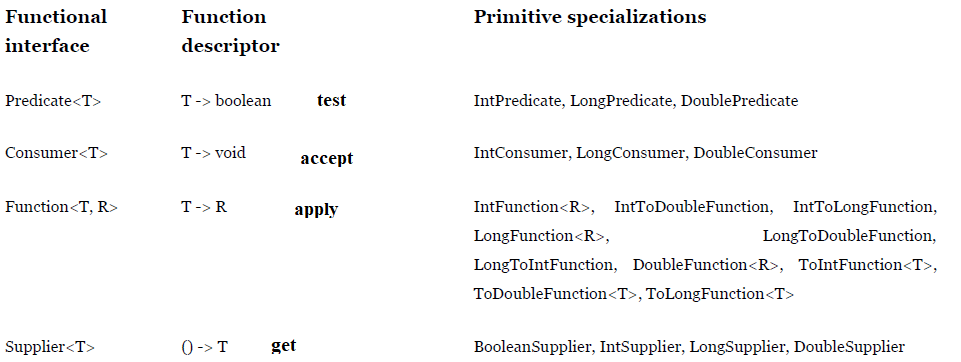
Comparator<Apple> byWeight1 = Comparator.*comparing*(Apple::getColor);



Or

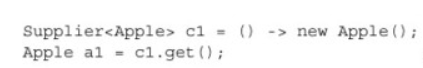


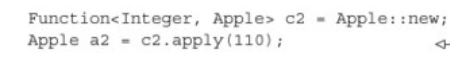




Constructor

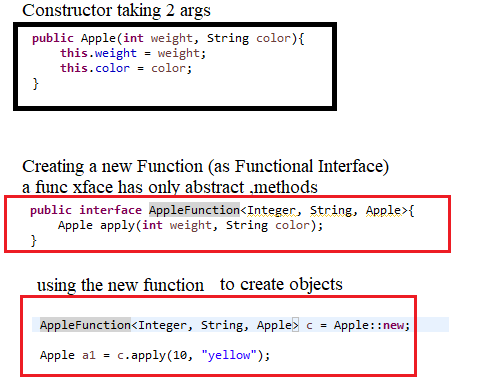


Its same as:



Same as

How to create such a function that can work with a constructor which takes more than one argument?



Below is a better way to do this:

|  |
| --- |
| **public** **interface** AppleFunction<T, S, R>{  R apply (T t, S s); } |

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

Java doc

Keyboard shortcut..

Adding java doc:

Alt+shift+j

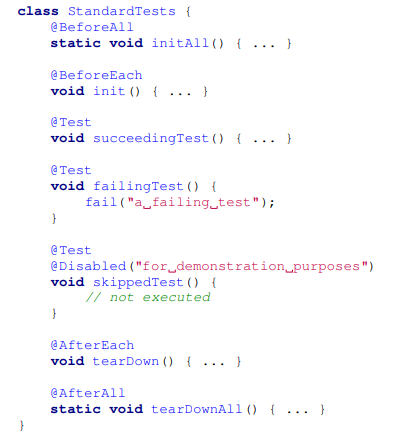
|  |
| --- |
| /\*\*  \* <h1>Add Two Numbers!</h1>  \* The AddNum program implements an application that  \* simply adds two given integer numbers and Prints  \* the output on the screen.  \* <p>  \* <b>Note:</b> Giving proper comments in your program makes it more  \* user friendly and it is assumed as a high quality code.  \*  \* @author Zara Ali  \* @version 1.0  \* @since 2014-03-31  \*/  public class AddNum {  /\*\*  \* This method is used to add two integers. This is  \* a the simplest form of a class method, just to  \* show the usage of various javadoc Tags.  \* @param numA This is the first paramter to addNum method  \* @param numB This is the second parameter to addNum method  \* @return int This returns sum of numA and numB.  \*/  public int addNum(int numA, int numB) {  return numA + numB;  }  /\*\*  \* This is the main method which makes use of addNum method.  \* @param args Unused.  \* @return Nothing.  \* @exception IOException On input error.  \* @see IOException  \*/  public static void main(String args[]) throws IOException {  AddNum obj = new AddNum();  int sum = obj.addNum(10, 20);  System.out.println("Sum of 10 and 20 is :" + sum);  }  } |

<https://www.tutorialspoint.com/java/java_documentation.htm>

<https://github.com/java8/Java8InAction/tree/master/src/main/java/lambdasinaction>

-----------

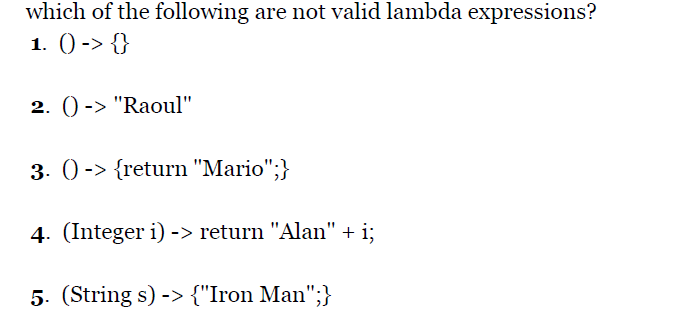
Unit tests



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

To avoid auto boxing:

Use IntPredicate in place of Predicate



1,2,3 r right… If u wanna return smthing have return statement. 4 is wrong. As u need return keyword with curly brackets… 5 is wrong as it returns a string, u shud have return keyword with curly brackets if u wanna return.



**Func Interface may have any methods but just ONE ABSTRACT METHOD**

In Java they are annotated with **@FunctionalInterface**

**Predicate**- which returns a Boolean.

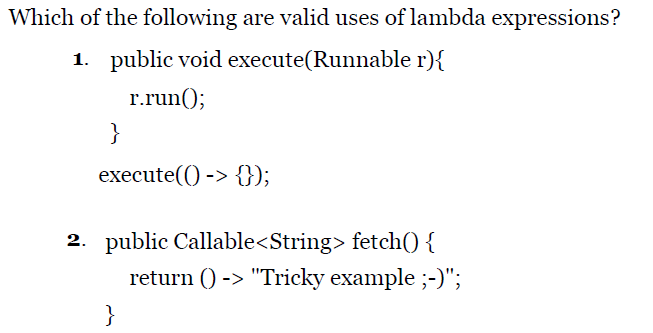
**Consumer**: Takes an argument <T> and returns a void. Eg. **forEach function** that prints..

****

**Function** takes **one (or more)** and returns one.. function<T,R>

public interface Function<T,R> {

R apply(T t); }

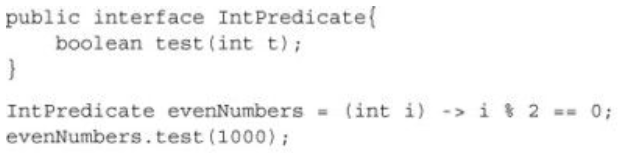


1. The first example is valid because the lambda () -> {} has the signature () -> void, which matches the signature of the abstract method run defined in Runnable.

**2. The second example is also valid. Indeed, the return type of the method fetch is Callable<String>.**

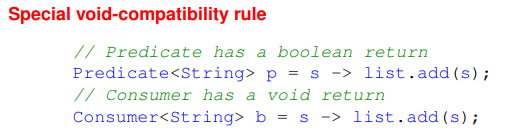
**Callable<String> essentially defines a method with the signature () -> String when T is replaced with String. Because the lambda () -> "Tricky example ;-)" has the signature () -> String, the lambda can be used in this context.**

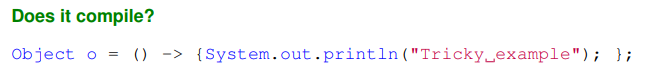
For example here, using intPredicate, no autoboxing happens:

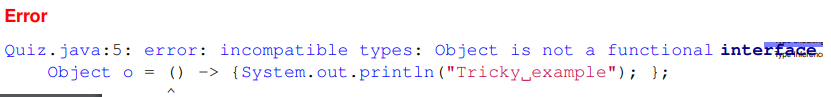


But the below predicate does autoboxing:





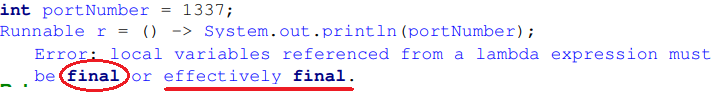




To fix above we can:



Only FINAL variables are allowed….



Convert below to method reference:



Ans:



Suppose you have an Apple class which has a default constructor, you can use a **SUPPLIER**

|  |
| --- |
| **Supplier**<Apple> c1 = () -> new Apple();  Then call supplier’s get method:  Apple a1= c1.**get**(); |

Now with constructor reference:

|  |
| --- |
| Supplier<Apple> c1 = **Apple::new**;  Then call supplier’s get method:  Apple a1= **c1.get();** |

Calling a non-default constructor: Here you need to use a **FUNCTION** and not Supplier

|  |
| --- |
| **Function<Integer, Apple> c2 = (weight) -> new Apple(weight);**  Then call functions apply method:  **Apple a2 = c2.apply(110);** |

Now with constructor reference

|  |
| --- |
| **Function<Integer, Apple> c2 = Apple::new;**  Then call functions apply method:  **Apple a2 = c2.apply(110);** |

We use Supplier’s get method for default constructor and Function’s apply for other.

Suppose we have a 3 parameter constructor, we can define a new function interface:

public interface TrippleFunction < T , U , V , R > {

R apply(T t, U u, V v); }

Now suppose our class is Color which takes 3 integers as constructor input:

TrippleFunction <Integer, Integer, Integer, Color> c1 = Color::new;

c1.apply(100,200,300);

Above is same as: TrippleFunction <Integer, Integer, Integer, Color> c1 = (i, j, k) -> new Color(i,j,k);

c1.apply(100,200,300);

**Composing Comparators**

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

Chaining Comparators

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

.**reversed**()

.**thenComparing**(Apple::getSize());

Composing Predicates

Suppose you have a redApple predicate…

Predicate<Apple> redApple = ((Apple a) -> a.getColor().equals(“red”));

We can now negate this:

Predicate<Apple> notRedApple = redApple**.negate**();

Apples that are red and heavy:

Predicate<Apple> redAndHeavyApple = redApple.**and**(a -> a.getWeight() > 150);

We can also use “or”.

Function<Integer, Integer> f = x -> x + 1;

Function<Integer, Integer> g = x -> x \* 2;

Function<Integer, Integer> h = f.**andThen**(g);

Gives 4 when h.apply(1)..

Function<Integer, Integer> j = f.**compose**(g);

Gives 3 when j.apply(1);

1. For each trader, return the number of *lowercase* letters in the name (hint: look at the chars method on String).
2. Find the city String with the largest number of lowercase letters from all the cities in the transaction list. Experiment with returning an Optional<String> to account for the case of an empty input list.

System.***out***.println(transactions.stream().map(x-> x.getTrader().getName()).distinct().map(x-> x.chars().filter(Character::*isLowerCase*).count()).collect(*toList*()));

transactions.stream().map(x->x.getTrader().getCity()).max(Comparator.*comparing*(o -> o.chars().filter(Character::*isLowerCase*).count())).get();

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

Stream Iterate and Generate function:

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

**Generating fibonnaci**

Stream.*iterate*(**new** **int**[]{0, 1},

t -> **new** **int**[]{t[1],t[0] + t[1]})

.limit(5)

.map(t -> t[0])

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

Fibonnaci tuples:

Stream.*iterate*(**new** **int**[]{0, 1},

t -> **new** **int**[]{t[1], t[0]+t[1]})

.limit(5)

.forEach(t -> System.***out***.println("(" + t[0] + "," + t[1] +")"));

Stream Generate: This code will generate a stream of five random double numbers **from 0 to 1.**

Stream.generate(Math::random)

.limit(5)

.forEach(System.out::println);

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

**Converting arrays to stream**

String[] arrayOfWords = {"Goodbye", "World"};

Stream<String> streamOfwords = Arrays.*stream*(arrayOfWords);

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

Two ways to get distinct characters

String[] arrayOfWords = {"Goodbye", "World"};

System.***out***.println(

Arrays.*stream*(arrayOfWords).reduce((a, b)-> a+b)

.map(x->x.split(""))

.map(Arrays::*stream*).filter(x-> !x.equals(""))

.get().distinct().collect(*toList*())

);

System.***out***.println(

Arrays.*stream*(arrayOfWords).map(x->x.split(""))

.flatMap(Arrays::*stream*)

.distinct()

.collect(*toList*())

);

From a list of people weight, find the one lower than 10.. If none exists return -10..

**int**[] ages = {10,20,30};

System.***out***.println( Arrays.*stream*(ages).filter(x-> x<10).findAny().orElse(-10) );

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

**Streams from values**

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

**Streams from arrays**



**Collect Methods:**

Always remember: you just have to formulate the result you want to obtain the “what” and not the steps you need to perform to obtain it—the “how.”

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

its same as long howManyDishes = menu.stream().count();

**Finding maximum and minimum in a stream of values**

First define a comparator:

Comparator<Dish> dishCaloriesComparator = Comparator.comparingInt(Dish::getCalories);

Optional<Dish> mostCalorieDish = menu.stream().**collect**(**maxBy**(dishCaloriesComparator));

**Summarization**

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

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

**Joining Strings**

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

To make it readable:

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

**Reduction: It takes three arguments:** second argument is the function

int totalCalories = menu.stream().collect(**reducing(0, Dish::getCalories, (i, j) -> i + j)**);

Using the same process we can find highest calorie one:

Optional<Dish> mostCalorie = menu.stream().collect(**reducing((d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2));**

Collect

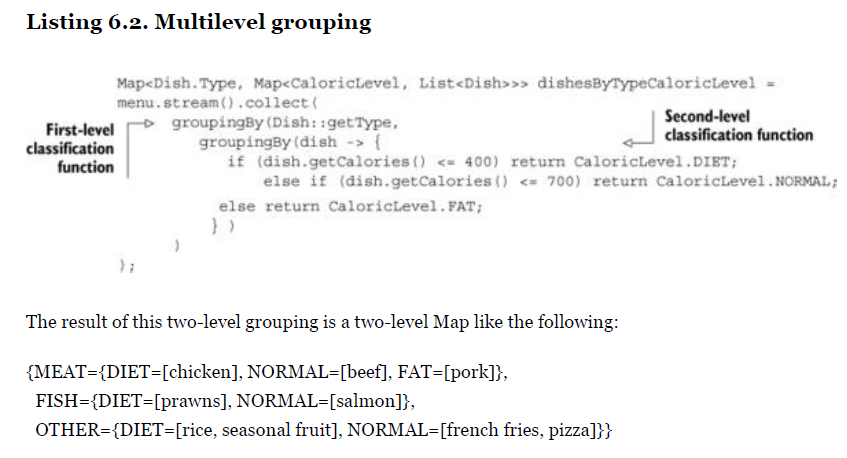


Or, int totalCalories = menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();

Or, int totalCalories = menu.stream().mapToInt(Dish::getCalories).sum();

**Grouping**

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



it’s possible to count the number of Dishes in the menu for each type, by passing the counting collector as a second argument to the groupingBy collector

Map<Dish.Type, Long> typesCount = menu.stream().collect(**groupingBy**(Dish::getType, **counting**()));

The result is the following Map: {MEAT=3, FISH=2, OTHER=4}

Collecting as a Map..

Please note you would be tempted to use:

.collect(Collectors.toMap(…..)… Instead its **easier** to use: **Collectors. groupingBy**

Map<Dish.Type, Long> typesCount

= menu.stream().collect(**Collectors.groupingBy**(Dish::getType, **Collectors.counting**()));

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

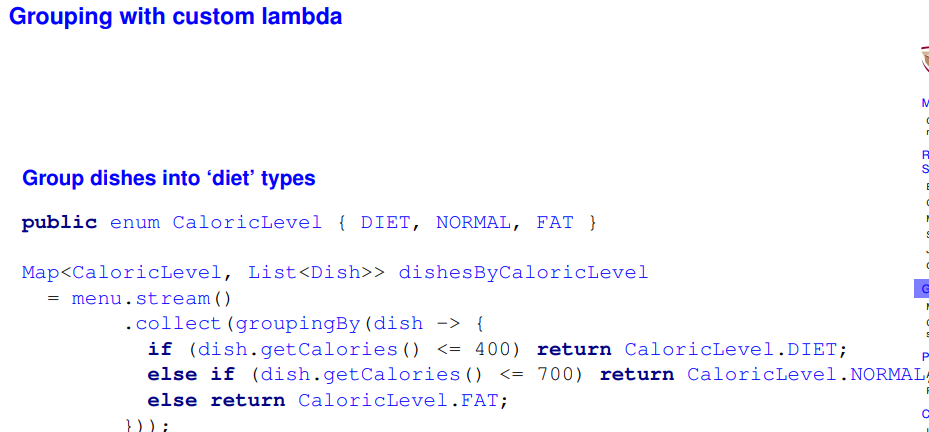
In the above example, if we want to find which dish in each type has highest calories?

Map<Dish.Type, **Optional**<Dish>> mostCaloricByType

= Dish.***menu***.stream().collect(Collectors.*groupingBy*(Dish::getType,Collectors. ***maxBy***(Comparator.*comparing*(Dish::getCalories))) );

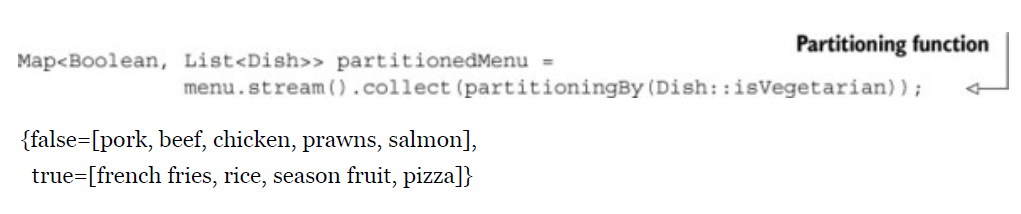
Please note “maxBy” returns an Optional… to be able to get the value out of optional, we need to use: ***collectingAndThen***

Map<Dish.Type, Dish> mostCaloricByType1 = Dish.***menu***.stream().collect(Collectors.*groupingBy*(Dish::getType, Collectors.***collectingAndThen***(Collectors.***maxBy***(Comparator.*comparing*(Dish::getCalories)), Optional::get)));





Partition data into 2 groups:



List<Dish> vegetarianDishes **= partitionedMenu.get(true);**

Remember if the first part is a Boolean, then it’s a partitioned thing

Also, in partitioned by clauses, the first thing is a Boolean as well..

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

Two ways to find maximum length of a String, reduce vs collect

String longestName = Dish.***menu***.stream().collect(Collectors.*maxBy*(Comparator.*comparing*(Dish::getName))).get().getName();

String longestName1 = Dish.***menu***.stream().map(Dish::getName).reduce((a,b) -> a.length() < b.length() ? b:a).get();

**OPTIONAL**

To get hold of an empty optional object

Optional<Car> optCar = Optional.empty();

**Optional from a non-null value**

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

* But if car was null, we will get NullPointerException..

To avoid:

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

In this case, if car was null, then resulting Optional object would be empty.

We can use “isPresent()” method to make sure value is present.

Example: Object value = map.get("key");

will return null if there’s no value in the map

Optional<Object> value = Optional.ofNullable(map.get("key"));

