**0.SCI**

Transcendental Consciousness is the field of pure awareness, beyond the active thinking level, that is the birthright and essential nature of everyone.Everyone “inherits” from pure consciousness,Wholeness moving within itself: In Unity Consciousness, there is an even deeper realization:The only data and behavior that exist in the universe is that which is “inherited from” pure consciousness – everything in that state is seen as the play of one’s own consciousness.

**1.(lesson7)inner class**

public class Static {

private String name = "Joe";

private Pair p = new Pair();

{

p.first = 4;

p.second = 5;

System.out.println(p);

}

private void printHello() {

System.out.println("Hello" + name);

}

static class Pair {

static int first;

int second;

Pair() {

//no access//printHello();

}

public String toString() {

return "(" + first + ", " + second+ ")";

}

}

public static void main(String[] args) {

(new Static()).printHello();

Pair.first = 1;

}}

**2.(lesson7)In Java 8, Enums Can “inherit”**

interface MyInterface {

default usefulThings() {. . .}}

enum MySingleton implements MyInterface {

INSTANCE;

@Override

public void behavior() {

. . .

usefulThings();}}

**3.(lesson7)New (Java 8) in the Iterable interface is a default method:**

Consumer<String> consumer = new Consumer<String>() {

@Override

public void accept(String s) {

System.out.println(s);

}};

System.out.println("------using new forEach method------");

l.forEach(consumer);

**4.(lesson7)guidelines for create an immutable class**

1)all fields should be private and final,

2)provide getters but no setters for all fields

3)make the class final

4)make sure that getters do not return mutable objects

**5.(lesson8)function programing**

public class FunctionExample {

public static void main(String[] args) {

class MyBiFunction implements BiFunction<Integer, Integer, Integer> {

public Integer apply(Integer x, Integer y) {

return 2 \* x.intValue() - y.intValue();

}}

MyBiFunction f = new MyBiFunction();

System.out.println(f.apply(2, 3)); // output 1

}}

**6.Method References**

public class Examples {

//type: Class::instanceMethod

BiConsumer<Employee, String> setName1 = (e,s) -> e.setName(s);

BiConsumer<Employee, String> setName2 = Employee::setName;

//type: Class::instanceMethod

Comparator<String> strComp1 = (s1,s2) -> s1.compareTo(s2);

Comparator<String> strComp2 = String::compareTo;

//type: Class::staticMethod

Function<String, Integer> parser1 = x -> Integer.parseInt(x);

Function<String, Integer> parser2 = Integer::parseInt;

//type: obj::instanceMethod

EmployeeNameComparator comp = new EmployeeNameComparator();

Comparator<Employee> empComp1 = (e1, e2) -> comp.compare(e1,e2);

Comparator<Employee> empComp2 = comp::compare;

public void evaluator() {

Employee testEmp = new Employee("John", 100000);

setName2.accept(testEmp,"Ralph");

System.out.println(strComp2.compare("a", "b"));

System.out.println(parser2.apply("15"));

System.out.println(empComp2.compare(testEmp, new Employee("John", 100000)));

}

**7.(lesson8)Using Lambdas and Streams (Java 8)**

public List<String> findStartsWithLetter(List<String> list, String letter) {

return list.stream() //convert list to stream

.filter(name -> name.startsWith(letter)) //returns filtered stream

.map(name -> name.toUpperCase()) //maps each string to upper case string

.collect(Collectors.toList()); //organizes into a list

}

public class Advanced {

final BiFunction<List<String>, String, List<String>> listStartsWith

= (list, letter) -> list.stream()

.filter(name -> name.startsWith(letter)).collect(Collectors.toList());

final List<String> friendsStartN = listStartsWith.apply(Folks.friends, "N");

public static void main(String[] args) {

Advanced adv = new Advanced();

System.out.println("Friends with names that start" + " with 'N': "+ adv.friendsStartN);

}}

**8.(lesson9)Get a Stream**

int[] arrOfInt = {1, 3, 5, 7};

Stream<Integer> strOfInt = Stream.of(arrOfInt);

Stream<String> song = Stream.of(“gently”, “down”, “the”, “stream”);

Stream<String> echoes = Stream.generate(() -> “Echo”);

Stream<Double> randoms = Stream.generate(Math::random);

Stream<BigInteger> naturalNums  = Stream.iterate(BigInteger.ONE, n -> n.add(BigInteger.ONE))

Stream<Integer> stream2 = Stream.iterate(1, n -> n + 1));

IntStream ints = IntStream.of(1, 2, 4, 8);

IntStream ones = IntStream.generate(() -> 1);

IntStream naturalNums = IntStream.iterate(1, n -> n+1);

IntStream zeroToNinetyNine = IntStream.range(0, 100);

IntStream zeroToHundred = IntStream.rangeClosed(0, 100);

Stream<Integer> integers = IntStream.range(0, 100).boxed();

IntStream lengths = words.mapToInt(String::length);

public static void infStream() {

List<int[]> intArrs = Stream.iterate(BigInteger.ONE, n -> n.add(BigInteger.ONE)).limit(20)

.map(x -> x.intValue()).map(int[]::new).collect(Collectors.toList());

List<String> intArrsStr = intArrs.stream().map(Arrays::toString).collect(Collectors.toList());

System.out.println(intArrsStr);

}

**9.(lesson9) api of stream**

Stream<Double> randoms = Stream.generate(Math::random).limit(100);

Stream<Character> combined =Stream.concat(characterStream("Hello"),characterStream("World"));

public static Stream<Character> characterStream(String s) {

List<Character> result = new ArrayList<>();

for (char c : s.toCharArray()) result.add(c);

return result.stream();}

words.stream().filter(name -> name.contains(""+c)).filter(name -> !name.contains(""+d))

.filter(name -> name.length()==len).count();

List<String> strings = list.stream() .map(x -> x.toString()) .collect(Collectors.toList())

“Flattening” this Stream means putting all elements together in a single list. This is accomplished using flatMap in place of map:

List<String> list = Arrays.asList(“Joe”, “Tom”, “Abe”);

Stream<Stream<Character>> result = list.stream().map(s -> characterStream(s))

Stream<Character> flatResult = list.stream().flatMap(s -> characterStream(s))

Stream<String> uniqueWords= Stream.of("merrily", "merrily", "merrily", "gently").distinct();

//output: ["merrily", "gently"]

//sort by decreasing lengths of words

List<String> words = Arrays.asList("Tom", "Joseph", "Richard");

Stream<String> longestFirst= words.stream().sorted((String x, String y) ->

(new Integer(y.length()).compareTo(new Integer(x.length()))));

System.out.println(longestFirst.collect(Collectors.toList()));//output: Richard, Joseph, Tom

Stream<String> longestFirst

= words.stream().sorted(Comparator.comparing(String::length).reversed());

Function<String, Integer> byLength = x -> x.length(); //same as String::length

Stream<String> longestFirst= words.stream().sorted(Comparator.comparing(byLength).reversed())

List<String> words = //…

int numLongWords = words.stream().filter(w -> w.length() > 12).count();

Optional<String> largest = words.stream().max(String::compareToIgnoreCase);

if (largest.isPresent()) System.out.println("largest: " + largest.get());

Optional<String> startsWithQ= words.stream().filter(s -> s.startsWith("Q")).findFirst();

Optional<String> startsWithQ= words.parallelStream() .filter(s -> s.startsWith("Q")) .findAny();

public static void pickName(List<String> names, String startingLetter) {

final Optional<String> foundName =names.stream().filter(name ->name.startsWith(startingLetter))

.findFirst();

foundName.ifPresent(name -> System.out.println("Hello " + name));

}

int prod1 = list.stream().reduce(1, (a,b) -> a \* b);

System.out.println("prod1 = " + prod1);

public class Reuse {

private static final BiFunction<List<String>, String, Stream<String>> listStartsWith

= (list, letter) -> list.stream().filter(name -> name.startsWith(letter));

public List<String> listEditorsNamesStartingWithNUpperCase() {

return listStartsWith.apply(Folks.editors, "N").map(name -> name.toUpperCase())

.collect(Collectors.toList());

}

public static void main(String[] args) {

Reuse r = new Reuse();

System.out.println(r.listEditorsNamesStartingWithNUpperCase());

}

}

public class LambdaLibrary {

public static final BiFunction<List<Customer>, String, List<String>> NAMES\_IN\_CITY

= (list, searchStr)-> list.stream().filter(cust -> cust.getCity().startsWith(searchStr))

.map(cust -> cust.getName()).sorted().collect(Collectors.toList());

}

String[] result = words.toArray(String[]::new);

List<String> result = stream.collect(Collectors.toList());

Set<String> result = stream.collect(Collectors.toSet());

TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new));

String result = stream.collect(Collectors.joining());

String result = stream.collect(Collectors.joining(", "));

String result = stream.map(Object::toString).collect(Collectors.joining(","));

Map<Integer, String> idToName

= people.collect(Collectors.toMap(Person::getId, Person::getName));

Map<Integer, Person> idToPerson

= people.collect(Collectors.toMap(Person::getId, Function.identity()));

List<String> listStr = list.stream()

.filter(cust -> cust.getCity().startsWith("Ma"))

.map(cust -> cust.getName())

.sorted()

.collect(Collectors.toList());

These suggest using a BiFunction as follows:

public static final BiFunction<List<Customer>, String, List<String>> NAMES\_IN\_CITY

= (list, searchStr) -> list.stream()

.filter(cust -> cust.getCity().startsWith(searchStr))

.map(cust -> cust.getName())

.sorted()

.collect(Collectors.toList());

10.(lesson11)Implementing a Generic Interface

public class SimplePair<K,V> {

private K key;

private V value;

public SimplePair(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() { return key; }

public V getValue() { return value; }

}

public static <K, V> boolean compare(SimplePair<K, V> p1, SimplePair<K, V> p2) {

return (Boolean)(p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue()));

}

public static void main(String[] args) {

//without type value

SimplePair<Integer, String> q1 = new SimplePair<>(1, "apple");

SimplePair<Integer, String> q2 = new SimplePair<>(2, "pear");

boolean areTheySame2 = Util.compare(q1, q2);

System.out.println("Result2="+areTheySame2);

}

11.(lesson11)Create a generic subclass

public class MapOperation {

public static void main(String[] args) {

List<Double> someDoubles = Arrays.asList(2.3, 3.5, 6.8);

List<String> words = Arrays.asList("dog", "elephant", "peacock");

List<Manager> mans = Arrays.asList(

new Manager("John", 100000, 2000, 10, 15),

new Manager("Steve", 120000, 1998, 2, 17));

List<Number> numbers =

//here, type R is Number and word.length() is of type Integer

words.stream().map(word -> word.length())

.collect(Collectors.toList());

numbers.addAll(someDoubles);

//here, type T is Manager, and Employee is supertype

numbers.addAll(mans.stream().map((Employee e) -> e.getSalary())

.collect(Collectors.toList()));

System.out.println(numbers);

}

}

public class BoundedTypeVariable {

public static void main(String[] args) {

List<Integer> ints = new ArrayList<>();

ints.add(3);ints.add(4);ints.add(-1);

System.out.println(max0(ints));

List<String> strs = new ArrayList<>();

strs.add("Bob"); strs.add("Steve"); strs.add("Tom");

System.out.println(max1(ints));

System.out.println(max1(strs));

}

public static Integer max0(List<Integer> list) {

Integer max = list.get(0);

for(Integer i : list) {

if(i.compareTo(max) > 0) {

max = i;

}

}

return max;

}

public static <T extends Comparable<T>> T max1(List<T> list) {

T max = list.get(0);

for(T i : list) {

if(i.compareTo(max) > 0) {

max = i;

}

}

return max;

}

public static <T extends Comparable<? super T>> T max1A(List<T> list) {

T max = list.get(0);

for(T i : list) {

if(i.compareTo(max) > 0) {

max = i;

}

}

return max;

}

public static <T> T max2(List<T> list, Comparator<T> cmp) {

T theMax = list.get(0);

for(T item : list) {

if(cmp.compare(item, theMax) > 0) {

theMax = item;

}

}

return theMax;

}

public static <T> T max3(List<? extends T> list, Comparator<T> cmp) {

T theMax = list.get(0);

for(T item : list) {

if(cmp.compare(item, theMax) > 0) {

theMax = item;

}

}

return theMax;

}

public static <T> T max4(List<? extends T> list, Comparator<? super T> cmp) {

T theMax = list.get(0);

for(T item : list) {

if(cmp.compare(item, theMax) > 0) {

theMax = item;

}

}

return theMax;

}

static class IntegerComparator implements Comparator<Integer> {

@Override

public int compare(Integer o1, Integer o2) {

return o1.compareTo(o2);

}}}

public class Capture {

public void copyFirstToEnd(List<?> items) {

//items.add(items.get(0)); //compiler error

}

public void copyFirstToEnd2(List<?> items) {

copyFirstToEndHelper(items);

}

private <T> void copyFirstToEndHelper(List<T> items) {

T item = items.get(0);

items.add(item);

}}

public static <T> void copy(List<? super T> destination, List<? extends T> source) {  for(int i = 0; i < source.size(); ++i) {  destination.set(i, source.get(i));  } }

List<? super Integer> list = new ArrayList<>(); list.add(1);

list.add(2);

Object ob = list.get(0);

System.out.println(ob.toString()); //output: 1

The map operation on Stream<T> has the following signature.

Stream<R> map(Function<? super T,? extends R> mapper)

This means that the type the map is transforming can be a supertype of the type of the list or collection that is being traversed, and that the type the map sends to can be a subtype of the expected return type.

**12.(lesson10)**

public class GenericWrapper {

public List<String> getCanonicalPaths(String[] dirs) {

return Stream.of(dirs).map(

path -> unchecked((String p) -> new File(p).getCanonicalPath()).apply(path))

.map(Object::toString).collect(Collectors.toList());

}

public static void main(String[] args) {

String[] localDirs = {"/usr", "//usr", "/etc"};

List<String> canonicalPaths

= (new GenericWrapper()).getCanonicalPaths(localDirs);

System.out.println(canonicalPaths);

}

public static <T, R> Function<T,R> unchecked(FunctionWithException<T,R> f) {

return x -> {

try {

return f.apply(x);

} catch(Exception e) {

throw new RuntimeException(e);

}

};}}

Unit-Testing Stream Pipelines:  Simple Expressions

@Test

public void multipleWordsToUppercase() {

List<String> input = Arrays.asList("a", "b", "hello");

List<String> result = Testing.allToUpperCase.apply(input);

assertEquals(Arrays.asList("A", "B", "HELLO"), result);

}

Unit-Testing Stream Pipelines:  ComplexExpressions

public class LibraryCompanion {

static Comparator<Employee> empComp

= Comparator.comparing((Employee e) -> e.getName())

.thenComparing((Employee e) -> e.getSalary(),

Comparator.reverseOrder());

public static int compareEmps(Employee e1, Employee e2) {

return empComp.compare(e1, e2);

}

}

Employee joe1 = new Employee("Joe", 50000);

Employee jim = new Employee("Jim", 70000);

//verify that jim comes before joe

assert(LibraryCompanion.compareEmps(jim, joe1) < 0);

static void printFile(File f) {

try {

BufferedReader reader = new BufferedReader(new FileReader(f));

System.out.println(reader.lines().collect(Collectors.joining(", ")));

reader.close();

} catch(IOException e) {

System.out.println("Error printing file: " + e.getMessage());

}

}