package edu.neu.csye6200;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

/\*\*

\* Final Exam Questions:

\* 1. What is this: 'System.out::println' ?

\* (Answer: Method Reference [Fill In M\_\_\_\_\_ R\_\_\_] or Lambda Expression or Lambda [Fill In L\_\_\_])

\* 2. What is the number 1 restriction when using a Generic class in Java?

\* (Answer: must use reference type, can't use primitive type)

\* 3. How can you AUTOMATICALLY use a primitive type with a generic class?

\* (Answer: AutoBoxing, i.e. wrapping int in Integer wrapper class)

\* 4. What Interface must the Lambda supplied to the forEach() method implement?

\* (Answer: Consumer Function Interface.)

\* 5. If class Student DOES NOT extends Dog and d is type Dog and s is type Student,

\* Why is d = s NOT allowed in?

\* (Answer: because Java is a Strongly typed language)

\* 6. If class Student extends Parent and p is type Parent and s is type Student,

\* Why is p = s allowed in?

\* (Answer: s IS-A Person because of Inheritance)

\* 7. Therefore Student s3 = (Student) p; // down casting is allowed

\* SO Why is down casting bad?

\* (Answer: because if violates S.O.L.I.D. design principles)

\* 8. Which Solid design principles does Legal down casting violate?

\* (Answer: O.L.D. of S.O.L.I.D. design principles)

\* 9. Why can you ONLY update a Swing GUI control

\* e.g. textField.setText()

\* from code executing on the Event Dispatch Thread (EDT)?

\* (Answer: because Java Swing is NOT thread safe)

\*/

/\*\*

\* Generic class ShowState, a generic class to demonstrate

\* generic programming. Formal Type < E > is replaced by

\* actual type upon use of this generic class.

\*

\* Single data member is a Sequential container of Generic Type < E > elements.

\*

\* Show the object state by using generic methods.

\*

\* @author dpeters

\*

\* @param <E> Formal Type E (pithy naming)

\*/

public class ShowState<E> {

/\*\*

\* Storage for objects of type <E>

\*/

private List<E> objects = null;

public enum Color { RED, WHITE, BLUE };

public enum ExplosionEnum { GUNSHOT, GRENADE, ABOMB };

public enum StockEnum { EBAY, IBM, GOOGLE, YAHOO, ATT };

/\*\*

\* Enumeration for counting

\* BUT only allow from

\* ONE, which is a value of 1

\* TWO, which is a value of 2

\* THREE, which is a value of 3

\* @author dpeters

\*

\*/

public enum CountEnum {

ONE(1), TWO(2), THREE(3); // assign specific values

private Integer countNumber;

private CountEnum(Integer n) {

countNumber = n;

}

public Integer getCountNumber() {

return countNumber;

}

}

/\*\*

\* Object Instance initializer

\*/

{

objects = new ArrayList<E>();

}

/\*\*

\* add an element object to list

\* @param obj

\*/

public void addOBject(E obj) {

this.objects.add(obj);

}

/\*\*

\* a person

\*

\* @author dpeters

\*

\*/

private class MyPerson {

private int age;

private String name;

/\*\*

\* Object Instance Initializer List

\* Each object instantiated from this class

\* will be initialized according to this block of code.

\* (the following block of code is executed with every

\* class constructor.

\*/

{

age = 17;

name = "Dan";

}

public MyPerson() {

super();

// TODO Auto-generated constructor stub

}

public MyPerson(int age, String name) {

super();

this.age = age;

this.name = name;

}

@Override

public String toString() {

return name + ",age " + age;

}

}

/\*\*

\* output the state of each object in list

\* @param title

\*/

public void showObjects1(String title) {

System.out.println(title);

this.objects.stream()

.forEach(System.out::println);

System.out.println(objects.size() + " objects.[1]");

}

/\*\*

\* output the state of each object in list

\* @param title

\*/

public void showObjects2(String title) {

System.out.println(title);

this.objects.stream()

.forEach(o -> System.out.println(o));

System.out.println(objects.size() + " objects.[2]");

}

public void demoEnum() {

System.out.println("\n\t" + ShowState.class.getName() + ".demoEnum()...");

StockEnum stockName = StockEnum.EBAY;

// StockEnum stockName = 1; // COMPILER ERROR: ONLY ASSIGNABLE BY ENUM STOCK TYPE

System.out.println("\n\t" + ShowState.class.getName() + ".demoEnum()... done");

}

/\*\*

\* demonstrate how to use Arrays.asList() to initialize a sequential container class.

\*/

public void useAraysAsList() {

System.out.println("\n\t" + ShowState.class.getName() + ".useAraysAsList()...");

/\*\*

\* Examples of using Arrays.asList() for initializing sequential container classes

\*/

String[] aNames = { "anne","bill" }; // Array of String objects

List<String> names = new ArrayList<>(Arrays.asList(aNames));

// List<String> names = new ArrayList<>(Arrays.asList("anne","bill"));

List<MyPerson> persons = new ArrayList<>(Arrays.asList(new MyPerson(), new MyPerson(18, "Jessica")));

// Following will work for Integer objects, NOT String or MyPerson objects

List<Integer> numbers = new ArrayList<>(Arrays.asList(1,2,3));

/\*\*

\* Use Functional Style,

\* Declarative programming style

\*

\* foreEach()

\*

\* method of List interface

\*

\* Use a method reference (a.k.a. Lambda Expression)

\* in forEach() to implement the Consumer Functional Interface

\*

\* List.forEach() [inherited from Iterable Interface]

\* NOTE: NO STREAM EXISTS HERE,

\* NOT USING STREAM API, STREAM IS NOT CREATED HERE

\*/

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

/\*\*

\* Use Functional Style,

\* Declarative programming style

\*

\* foreEach()

\*

\* method of Stream interface

\*

\* Use a method reference (a.k.a. Lambda Expression)

\* in forEach() to implement the Consumer Functional Interface

\*

\* stream.forEach()

\* NOTE: A STREAM EXISTS HERE,

\* USING STEAM API as STREAM IS CREATED HERE with numbers.stream()

\*/

numbers.stream()

.forEach(System.out::println); // Stream.forEach() [inherited from Iterable Interface]

System.out.println("\n\t" + ShowState.class.getName() + ".useAraysAsList()... done!");

}

/\*\*

\* demonstrate how to use Arrays.asList() GENERICALLY

\* to initialize a sequential container class.

\*/

public void useAraysAsListGenerically(List<E> list) {

System.out.println("\n\t" + ShowState.class.getName() + ".useAraysAsList()...");

/\*\*

\* Use Functional Style,

\* Declarative programming style

\*

\* foreEach()

\*

\* method of List interface

\*

\* Use a method reference (a.k.a. Lambda Expression)

\* in forEach() to implement the Consumer Functional Interface

\*

\* List.forEach() [inherited from Iterable Interface]

\* NOTE: NO STREAM EXISTS HERE,

\* NOT USING STREAM API, STREAM IS NOT CREATED HERE

\*/

System.out.println("Using List.forEach()");

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

/\*\*

\* Use Functional Style,

\* Declarative programming style

\*

\* foreEach()

\*

\* method of Stream interface

\*

\* Use a method reference (a.k.a. Lambda Expression)

\* in forEach() to implement the Consumer Functional Interface

\*

\* stream.forEach()

\* NOTE: A STREAM DOES EXISTS HERE,

\* USING STEAM API as STREAM IS CREATED HERE with list.stream()

\*/

System.out.println("Using List.forEach()");

list.stream()

.forEach(System.out::println); // Stream.forEach() [inherited from Iterable Interface]

System.out.println("\n\t" + ShowState.class.getName() + ".useAraysAsList()... done!");

}

/\*\*

\* Demonstrate the use of casting,

\* specifically down casting with the use of "instanceOf" for added safety.

\*

\* NOTE: casting is STILL a bad practice as it violates S.O.L.I.D design principles

\*/

public void useCasting() {

System.out.println("\n\t" + ShowState.class.getName() + ".useAraysAsList()...");

Number n = null;

Integer i = new Integer(7);

if (i instanceof Integer) {

System.out.println(i + " IS-A Integer type."); // TRUE

}

n = i; // Integer i IS-A Number (Integer is subclass of Number)

if (n instanceof Integer) {

System.out.println(n + " IS-A Integer type (assigned to a Number variable n)."); // TRUE

}

System.out.println("\n\t" + ShowState.class.getName() + ".useCasting()... done!");

}

/\*\*

\* demonstrate how to use a generic class with an inner class

\* NOTE: an inner class can not be instantiated from a static method.

\*/

public void useGenericClass() {

System.out.println("\n\t" + ShowState.class.getName() + ".useGenericClass()...");

/\*\*

\* Instantiate ShowState generic class for MyPerson objects

\*/

ShowState<MyPerson> myPersonObjectList = new ShowState<>();

/\*\*

\* Add MyPerson objects

\*/

myPersonObjectList.addOBject(new MyPerson());

myPersonObjectList.addOBject(new MyPerson(18, "anne"));

myPersonObjectList.addOBject(new MyPerson(19, "jim"));

myPersonObjectList.showObjects1("ShowState.ShowObjects1() List of MyPerson objects;");

myPersonObjectList.showObjects2("ShowState.ShowObjects2() List of MyPerson objects;");

System.out.println("\n\t" + ShowState.class.getName() + ".useGenericClass()... done!");

}

public static void demo() {

System.out.println("\n\t" + ShowState.class.getName() + ".demo()...");

/\*\*

\* Instantiate ShowState generic class for Integer objects

\*/

ShowState<Integer> integerObjectList = new ShowState<>();

/\*\*

\* Add Integer objects

\*/

integerObjectList.addOBject(new Integer(1));

/\*\*

\* AutoBoxing of Literal int value inside an Integer object

\*/

integerObjectList.addOBject(2);

integerObjectList.addOBject(3);

integerObjectList.showObjects1("ShowState.ShowObjects1() List of Integer objects;");

integerObjectList.showObjects2("ShowState.ShowObjects2() List of Integer objects;");

/\*\*

\* Instantiate ShowState generic class for Double objects

\*/

ShowState<Double> doubleObjectList = new ShowState<>();

/\*\*

\* Add Double objects

\*/

doubleObjectList.addOBject(new Double(1));

/\*\*

\* AutoBoxing of Literal double value inside an Double object

\*/

doubleObjectList.addOBject(2.2);

doubleObjectList.addOBject(3.3);

doubleObjectList.showObjects1("ShowState.ShowObjects1() List of Double objects;");

doubleObjectList.showObjects2("ShowState.ShowObjects2() List of Double objects;");

/\*\*

\* Instantiate ShowState generic class for String objects

\*/

ShowState<String> stringObjectList = new ShowState<>();

/\*\*

\* Add Strings

\*/

stringObjectList.addOBject("One");

stringObjectList.addOBject("Two");

stringObjectList.addOBject("Three");

stringObjectList.showObjects1("ShowState.ShowObjects1() List of String objects;");

stringObjectList.showObjects2("ShowState.ShowObjects2() List of String objects;");

/\*\*

\* This is NOT good,

\* Using a Generic class

\* WITHOUT specifying the parameterized type (Raw Type)

\* and Eclipse will show a warning

\*/

ShowState obj = new ShowState(); // Raw type, missing parameterized type

obj.useAraysAsList();

ShowState<Object> object = new ShowState<>();

object.useGenericClass();

/\*\*

\* Better Generic programming and usage

\*/

integerObjectList.useAraysAsListGenerically(Arrays.asList(1,2));

stringObjectList.useAraysAsListGenerically(Arrays.asList("One","Two"));

integerObjectList.useCasting();

System.out.println("\n\t" + ShowState.class.getName() + ".demo()... done!");

}

}