**Group 31**

Brennon Francis - bqfranci - [bqfranci@asu.edu](mailto:bqfranci@asu.edu)

Andronick Martusheff - aamartus - [aamartus@asu.edu](mailto:aamartus@asu.edu)

**HW1**

**3. ArrayLists**

**3.1**

H1\_31.java file submitted separately.

**3.2**

*Initial contents of List:*

*list = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 }*

*for(int i = 1; i < 10; i++) {*

*list.set(i, list.get(i) + list.get(i-1));*

*}*

After the above for loop completes, the new contents of list are:

list = {1, 3, 6, 10, 15, 19, 22, 24, 25, 25}

**3.3**

**//** The method below passes an ArrayList to function “getNumNegative” and returns the number of

// negative numbers within the passed list. If there are none, the function will return 0.

*public static int getNumNegative(ArrayList<Integer> list) {*

*int numNegative = 0;*

*for(Integer num : list) {*

*if(num < 0)*

*numNegative++;*

*}*

*return numNegative;*

*}*

**3.4**

H1\_34.java file submitted separately.

**3.5**

H1\_35.java file submitted separately.

**3.6**

**//** The method below passes String ‘pName’ into ArrayList ‘pList’. pList is sorted in ascending order, and

// pName is inserted by iterating through pList until the proper index is reached.

public static void insertName(ArrayList<String> pList, String pName){

for(int i = 0; i < pList.size(); i++) {

if ((pName.compareToIgnoreCase(pList.get(i))) <= 0) {

pList.add(i, pName);

break;

} else if(i == pList.size() - 1) {

pList.add(pName);

break;

}

}

}

**3.7**

//The method below takes in two parameters: Int ArrayList, Int. Every instance of the int in the second //parameter is removed from the ArrayList specified in the first parameter.

**public static void arrayListRemove(ArrayList<Integer> pList, int pValue) {**

**pList.removeIf(value -> value == pValue);**

**}**

**4. Text File IO**

**4.1**

When trying to read a file that does not exist, an exception is thrown. (‘File not found…’)

**4.2**

When trying to write to a file that does not exist, the file is created.

**4.3**

H1\_43.java file submitted separately.

**5. Exceptions & Exception Handling**

**5.1**

An exception is thrown when something in the code fails. An exception handler will catch the thrown exception, and handle the exception.

**5.2**

Checked exceptions aren’t due to bugs in the code. They are required to be handled, or else your code will not compile. If they are not handled, you will receive syntax errors because your Java program expects you to handle them.

**5.3**

Unchecked exceptions are due to bugs in the code. They are not required to be handled in an exception handler, and you really don’t want them to be anyway. If you leave code related bugs to the JVM, you will be able to locate the bugs by reading up the list of the presented stack trace. Because your code has immediate control of the bug, you want to fix the code within the program.

**5.4**

The type of uncaught exceptions that must be declared in the method header are “Checked Exceptions”. If there is no declaration, then there will be an error in the code and the program will terminate.

**5.5**

You don’t need to declare an Index Out of Bounds exception in your code because it’s favorable to have the program crash, and solve the bug within the code. It’s an internal error.

**5.6**

An exception handler could be written with the focus of catching a specifically expected exception, but it can also be home to multiple catch clauses that will each have a specific way of handling a thrown exception.

**5.7**

The finally clause is added to a try block where the code may throw an exception. Regardless of whether or not the code throws an exception, the code within the finally clause will be run. An example of this is if the try block tries to open a file but throws an exception, a catch clause will run and handle the exception. After the exception is handled, and the file is presumably opened, the file still needs to be closed. The finally clause will contain the code to close the file.

**5.8**

The exceptions that both *next()* and *nextInt()* throw are:

* NoSuchElementException (if no more tokens are available)
* IllegalStateException (if the scanner is closed)

*nextInt()* also throws the:

* InputMismatchException (if the next token does not match the integer regular expression, or is out of range)

which the *next()*  method does **not** throw.

All of these exceptions listed as “Unchecked Exceptions”.

**6. Objects and Classes**

**6.1**

Instance methods require an object of its class to be created before being called. Static methods are methods in Java that can be called without creating an object.

**6.2**

If a class named “C” is written without having any constructors implemented, Java will provide a default construction of the Object. Also, if no parameters are given when trying to create an object from class “C”, a default constructor can be used to give the object of class “C” a baseline default value.

**6.3**

1. Instance method is called using an object.
2. Class method is called using the class name.

**6.4**

The output of the code snippet gives a Runtime Exception: NullPointerException because String S was never assigned a value.

**6.5**

H1\_65.java file submitted separately.

**6.6**

H1\_66.java file submitted separately.

**6.7**

In 6.6, class H1\_65 was created, and H1\_66,java held the main method to run the class that was held in a different file. For 6.6, the constructor was set to create cObj1 (using the default constructor). For 6.7, cObj2 was created in a similar way, except with passing the value ‘10’ for pX in the constructor, which ultimately set mX to 10 for the cObj2 object.

**6.8**

The following are snippets of code using the H1\_65 class, followed by a statement of whether or not they work/compile.

*(a) int a1 = H1\_65.mX; - Does not compile. mX is not a class variable.*

*(b) int a2 = H1\_65.mY; - Does not compile. mY is not a class variable.*

*(c) int a3 = H1\_65.A; - Does compile. A is a class variable.*

*(d) int a4 = H1\_65.B; - Does compile. B is a class variable.*

*(e) cObj1.H1\_65(20); - Does not compile. Cannot construct using object.*

*(f) int a5 = cObj1.getX(); - Does compile. Assigns received value to a5.*

*(g) cObj1.setX(20); - Does compile. Appropriately sets value of cObj1.*

*(h) cObj2.setX(cObj1.getX()); - Does compile. Sets value to to received value.*

*(i) int a6 = H1\_65.getX(); - Does not compile. getX is an instance method*

*(j) H1\_65.setX(20); - Does not compile. setX is an instance method.*

*(k) int a7 = cObj1.getY(); - Does compile. Assigns received value to a7.*

*(l) cObj1.setY(20); - Does compile. Sets value of mY appropriately.*

*(m) int a8 = H1\_65.getY(); - Does not compile. getY is an instance method.*

*(n) H1\_65.setY(20); - Does not compile. setY is an instance method.*

**6.9**

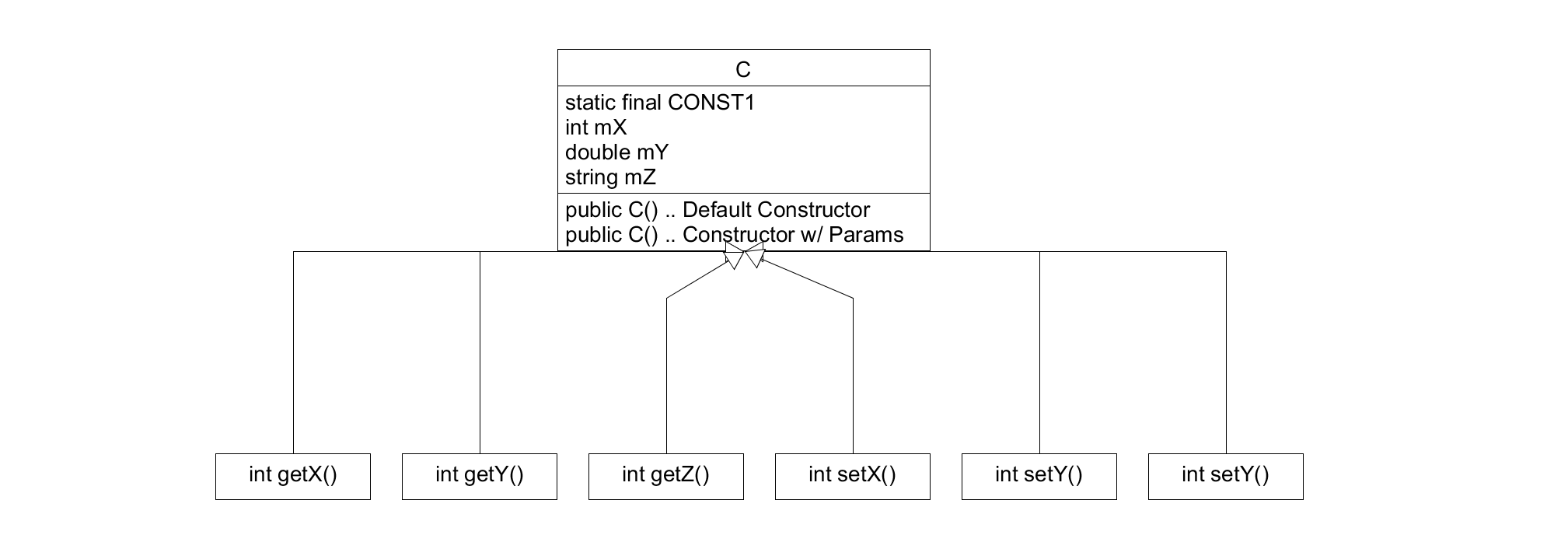
*(a) public void f() {mX = 0; mY = 0;} - Does compile. Variables set/accessed appropriately.*

*(b) public static void g() { - Does not compile. “this” is not present within the*

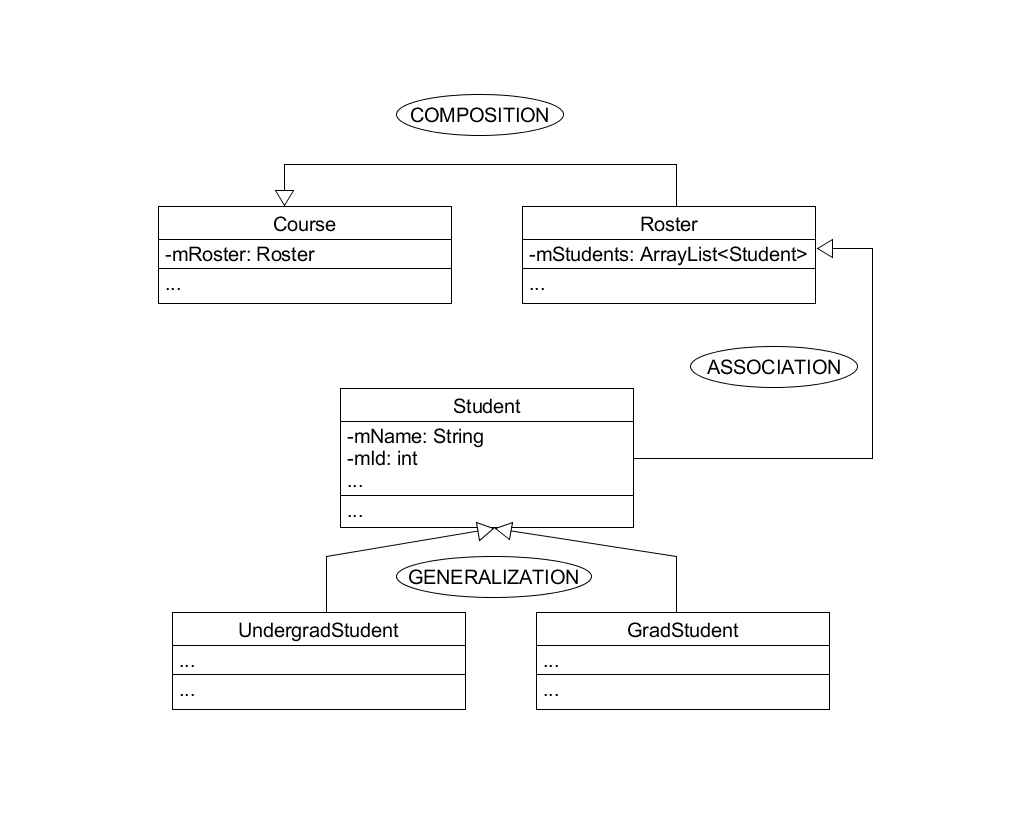
*mX = 0; mY = 0;} static method. It needs it in order to compile.*

**7. Object Oriented Design & UML Class Diagrams**

**7.1**

****

**7.2**

****

1. *Composition. Multiple courses become a roster, so several instances of courses by a student compose a roster.*
2. *Association. The relationship between Student and Roster is association because Student will be in the Roster.*
3. *Generalization. Because UndergradStudent is a type of student, they carry a generalized relationship.*
4. *Generalization. Because GradStudent is a type of student, they carry a generalized relationship.*
5. *None. Although both have a similar relationship to the Student, they are each their own individual type of student class.*