**Lab 11**

1. Consider the following code fragments. For each, if there is a compiler error, identify where it occurs.  
     
   a. First fragment:

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

ints.add(1);

ints.add(2);

List<Number> nums = ints;

nums.add(3.14);  
   
 b. Second fragment:

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

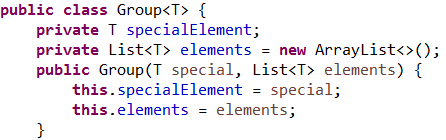
ints.add(1);

ints.add(2);

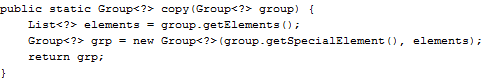
List<? extends Number> nums = ints;

nums.add(3.14);

1. A *group* is a collection of elements having one special element. An example of a group is the set of integers {. . . -2, -1, 0, 1, 2, . . . }, with special element 0.  
     
   Here is a representation of a group as a Java class:



}

The following static method attempts to make a copy of a given instance of a Group, reproducing the state of the group in the copy.  
  
 

The code does not compile. Fix the code by capturing the wildcard with a helper method. Startup code is provided in the directory for this lab problem. Use the main method provided there to test your implementation. Note that the Group class has a toString method that will help in your test.

1. Draw a class diagram showing the inheritance relationships among the following types:  
     
   List<Integer>, List<Number>, List<? extends Integer>,   
   List<? extends Number>, List<? super Integer>, List<? super Number>, List<?>, List<Object>
2. Recall the definition of sum given in the slides:  
    public static double sum(Collection<? extends Number> nums {

double s = 0.0;

for(Number num : nums) s += num.doubleValue();

return s;

}

* 1. Is there a compiler error in the following lines of code? If so, where?

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

ints.add(1);

ints.add(2);

List<? extends Number> nums = ints;

double dbl = sum(nums);

nums.add(3.14);

* 1. Is there a compiler error in the following lines of code? If so, where?

List<Object> objs = new ArrayList<>();

objs.add(1);

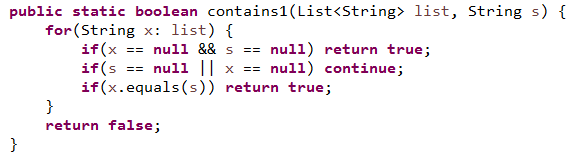
objs.add("two");

List<? super Integer> ints = objs;

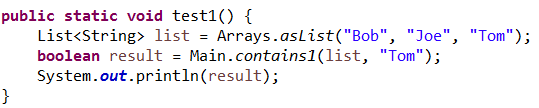
ints.add(3);

double dbl = sum(ints);

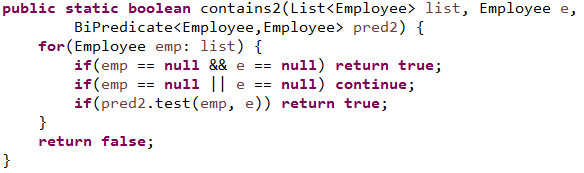
1. Create a generic programming solution to the problem of finding the second smallest element in a list. In other words, devise a public static method secondSmallest so that it can handle the biggest possible range of types.
2. Generalize the contains method for a List in the following way. First consider a simple implementation for a List of Strings:



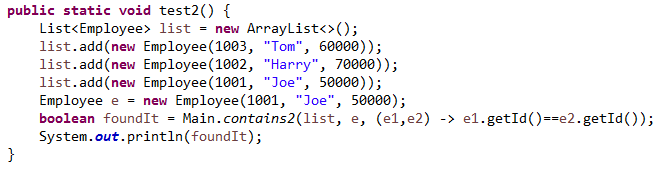
This contains method is tested in the following test method:



In more general lists, the objects in the List may not have overridden the equals method. This could be handled by introducing a BiPredicate, as in the following:



The BiPredicate can be used to represent an equals method for Employees. The following test uses it in this way, declaring two Employees to be equal if their id's are equal:



Now we want to generalize from Employee to a type variable T. Write the code for the most general possible contains method. Test your method using each of the following test() methods. You should be able to define a single contains method that will give correct results for each of these tests. Startup code is in the code folder for this lab.

**public** **static** **void** test2() {

List<Employee> list = **new** ArrayList<>();

list.add(**new** Employee(1003, "Tom", 60000));

list.add(**new** Employee(1002, "Harry", 70000));

list.add(**new** Employee(1001, "Joe", 50000));

Employee e = **new** Employee(1001, "Joe", 50000);

**boolean** foundIt = Main.*contains2*(list, e,   
 (e1,e2) -> e1.getId()==e2.getId());

System.***out***.println(foundIt);

}

**public** **static** **void** test3() {

List<Manager> list = **new** ArrayList<>();

list.add(**new** Manager(1003, "Tom", 60000, 700));

list.add(**new** Manager(1002, "Harry", 70000, 400));

list.add(**new** Manager(1001, "Joe", 50000, 500));

Manager m = **new** Manager(1001, "Joe", 50000, 500);

**boolean** foundIt = Main.*contains3*(list, m,   
 (Employee e1, Employee e2) -> e1.getId()==e2.getId());

System.***out***.println(foundIt);

}

**public** **static** **void** test4() {

List<Manager> list = **new** ArrayList<>();

list.add(**new** Manager(1003, "Tom", 60000, 700));

list.add(**new** Manager(1002, "Harry", 70000, 400));

list.add(**new** Manager(1001, "Joe", 50000, 500));

Manager m = **new** Manager(1001, "Joe", 50000, 500);

**boolean** foundIt = Main.*contains3*(list, m,   
 (Employee e, Person p) -> e.getName().equals(p.getName()));

System.***out***.println(foundIt);

}

**public** **static** **void** test5() {

List<CheckingAccount> list = **new** ArrayList<>();

list.add(**new** CheckingAccount(1001, 25.00));

list.add(**new** CheckingAccount(1002, 35.00));

list.add(**new** CheckingAccount(1003, 125.00));

Account a = **new** CheckingAccount(1002, 35.00);

**boolean** foundIt = Main.*contains4*(list, a,

(Account a1, Account a2) -> a1.getAcctId()==a2.getAcctId());

System.***out***.println(foundIt);

}

1. In the code folder for this lab, there is a mystream package containing a MyStream class (package: lesson11.labs.prob7.mystream\_lab) and two test classes, Good and TestFlatMap. Your task is fully implement MyStream so that it behaves like Java's Stream implementation – it should support filter, map, flatMap and some other operations that belong to Java's Stream. The MyStream class that has been provided is a skeleton – you must implement the methods. You must do it well enough so that the methods in the Good and the TestFlatMap classes (in the same package) produce correct output. Note: the code in Good and TestFlatMap should not be modified.   
     
   Important: Do NOT use Java's Stream type in any way! You are creating your own version of streams, so you cannot rely on Java's implementation.