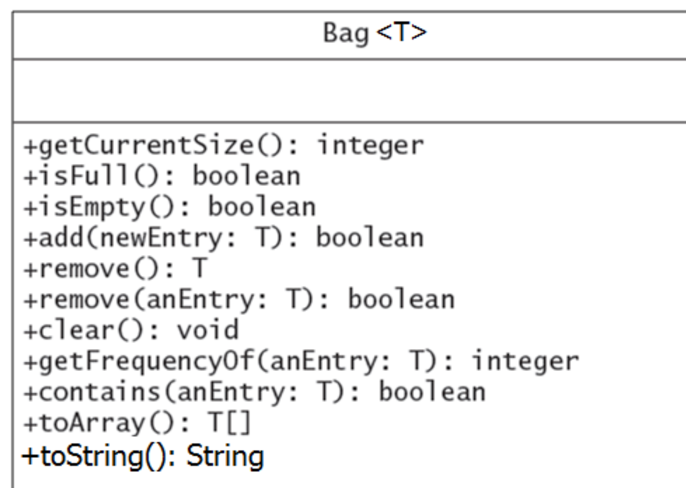
# CSIS 10B		Lab 2	Bags and Stacks

Part A Bags and Inheritance

In this part of the lab we will be exploring the use of the Bag ADT to manage quantities of data of a certain generic type (listed as T in the UML diagram below). We will also get to work with two classes that are related through inheritance, and two classes that are related through composition (adapter classes)

The Bag class was discussed in lecture. Here is the UML diagram for all the methods you can use in the Bag class:



You will want to keep this diagram handy as you work on these exercises.

Demo 1 Defining a Bag object and why generic type specifiers are good

We are given some code that compiles but gives us a warning:

Lab2.java uses unchecked or unsafe operations.

This warning is telling us something about how we are using the Bag object b. The Bag class is designed to take a specifier for the type of information your Bag object will contain. If you don't specify a type, Java still compiles the program, but it then lets you add any class of object you like, from String to Double to Person objects!

Try the code and see how it runs.

This is considered unsafe because it prevents the compiler from checking your code for incorrect use of your Bag object. In general we want our Bags to contain objects of only one type. If you want to allow anything to be added, and prevent the warning, you could declare a Bag<Object>, but this is not recommended.

Now fix the program by specifying b is a Bag<String> both in front of the variable declaration b and after the new operator. Now when you compile notice the error message alerting you of a type mismatch error. Now modify turn all the items added into String and run the program again. It should run without error.

Problem 1 Working with the Bag class

Now work with your partner to complete problem 1, trying out all the methods asked for in each part, and printing the result. In most cases you can type

```
System.out.println( "some message " + myBag.methodToCheck() );
```

where *methodToCheck()* is the method you are trying to verify, such as contains().

Problem 2 Using a bag to represent a Club of Persons

a) Now you can use the generic specifier <Person> to create a Bag of Person objects called CS_club, and add three persons to the Bag.

b) In order to be able to see if the club contains a certain Person, such as Joe Frank, you would use the statement `CS_club.contains(new Person("Joe Frank"));` First, try printing out whether the club contains Joe Frank. You should get the value “false.” Which is not correct!

   To understand why, it helps to realize that the `contains` method needs to traverse the bag’s internal array to see if the argument (such as `Person("Joe Frank")`) EQUALS one of the objects stored in the array, and return `true` if a match is found. But If we don’t properly define an `equals` method for the Person class, the `contains` method won’t work for a Bag of `Person` objects.

c) To get the correct result for b), please define an equals method for the Person class using the pattern we often used in CSIS10A, as shown for the Item class in the lecture slides. Try running b) again and verify that the CS_club’s contains method now shows Joe Frank as being in the club.

d) Wait, it STILL doesn’t work? OK, now implement the equals method so it takes a parameter of type Object, as shown for the Item class in the slide after the CSIS10A version. NOW you should finally get the proper behavior when you check to see if the CS_club contains “Joe Frank”!

Problem 3 Using inheritance to derive a Student class from the Person class

We can derive a Student class from Person because a Student IS A Person. Fill in the parts of the class indicated with "***" comments. Then test the Student class by activating the driver found in the main method at the bottom of the Student class file.

Add a toString method that invokes toString from the super class and appends to it the ID number.

Add an equals method for the Student class, using the equals method for the super class and also verifying the ID fields are the same.

Verify your methods work by running the code already given in Problem 3. Then try to add a couple Student objects to your club of Persons. Does it work? It should! Inheritence lets you add a Student to a Club of Persons since a Student IS A Person. (however if the club was defined to only work with Student, a Person could not join the club because a Person is NOT (always) a Student

Problem 4 Making a Club adapter class (with a Bag inside to hold data)

Perhaps it seems clumsy to create a   Bag<Person> CS_club .   After all, a club is NOT a Bag!! Suppose we wanted to make a Club class using a Bag to hold all of the members of the club.

Such a class would be considered an Adapter class (it uses Composition) and would start out like the following:

```
public class Club
{
        private  Bag<Person>  members;  // the set of club members

        ...
```

Here is a UML diagram of how the Club class might look:

```
Club
-----------------------------------------------
-members: Bag<Person>
-----------------------------------------------
+Club()
+getCurrentSize(): integer
+isFull(): boolean
+isEmpty(): boolean
+add(newEntry: Person): boolean
+remove(): Person
+remove(anEntry: Person): boolean
+clear(): void
+getFrequencyOf(anEntry: Person): integer
+contains(anEntry: Person): boolean
+toString(): String
```

The constructor would simply assign members = new Bag<Person>() ;

The other methods would essentially “Pass the buck” to the members Bag. In other words,

```
public boolean isFull()
{
        return members.isFull();
}
```

Create a new Club class now as an adapter for Bag and define a few of the methods above. Then modify your solution to problem 2 to use your Club class instead of a Bag<Person>. Note that you won’t say Club<Person> since members is “hard coded” to be a Bag<Person>, no generic type specifier is needed.

Is this beneficial? It does bring more logic to the code, and we could also add attributes to Club that are unique to clubs. For example, we could add a field that holds the contact information for the Club, or the name of the Secretary, or the location where the club meetings take place. All these fields would then be managed by additional Club methods, but each club would still use a Bag to store the list of members.

Problem 5 Static Method to Find the Intersection of Two Bags (EXTRA CREDIT)

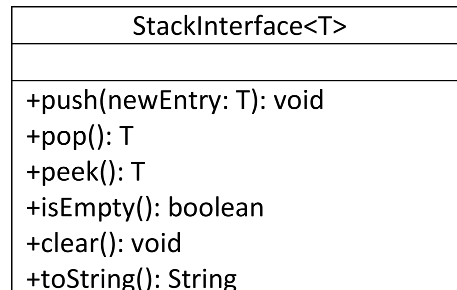
The *intersection* of two collections is a new collection of the entries that occur in both collections. That is, it contains the overlapping entries. Define a static (client) method that takes two Bag objects and returns a third bag containing the items in common to both bags.

Be sure to verify that when your method returns, that bagA and bagB have their original items (it’s ok if their order is different). Use the following header already provided inside the Lab2 class file:

```
public static <T> Bag<T> intersection( Bag<T> bagA, Bag<T> bagB)
```

Part B Stacks and Interfaces

We will now turn to the Stack, a real workhorse data structure in computing. As discussed in lecture, the UML for StackInterface is



You will probably need to refer to this during the lab.

Problem 1 Using StackInterface to define Stack variables

We'll begin this part of the lab in the Lab2B project folder, and in the Lab2B program file. Create a StackInterface variable and assign it either a new ArrayStack or new LinkedStack.

Create another of the other variety. Notice how a StackInterface variable can refer to either an ArrayStack or a LinkedStack.

This is because either class is guaranteed to have all the methods needed for a Stack, because they both said "public class XXXXStack implements StackInterface" and the compiler forced them to have the exact same methods as the interface specified.
 Problem 2 Examine and remove items

Write code to use peek to print the top item of the Stack without removing it, then write a while loop that removes all the items as long as the stack is not empty.

Problems 2-4 see Lecture Slides or Video

Problem 5 Postfix Processor

Step 1. Run the postfix solution and watch it evaluate postfix expressions like: 53+   or   67*6-3/
Step 2. Run the program at problem 5

Then, uncomment out problem 5 in Lab2B. This code sets up all the variables you will need to process a postfix expression (as discussed in lecture) and compute its numeric result using a Stack of numbers.

Since inputting data is a bit tricky, we're reading a complete postfix expression as a single String (MAKE SURE YOU DO NOT ENTER SPACES in your postfix input) such as 34+5*   which means (3+4)*5 or 60.

Each character is either a digit or an operator (+,-,*,/). We then use a traversal (for) loop so we can process the expression one character at a time using `mathExpression.charAt(k)` to retrieve the kth symbol in the expression.

As given, the program currently reads an expression and prints it to the screen one letter at a time

Step 3. Process digit tokens

To convert each digit token to a number we can use methods in the Character class. First, google java Character class and read through the javadocs for Character. Look for a method that tells you if a character is a digit.

`static boolean`	`isDigit(char ch)` Determines if the specified character is a digit.

also look for a method that will convert a character digit to a numeric value

`static int`	`getNumericValue(char ch)` Returns the `int` value that the specified Unicode character represents.

Now, in the for loop, after the line that prints `symbol`, add an if statement that tests if `symbol` is a digit, and if so, write another statement that converts `symbol` to an integer (use getNumericValue above), and store it in the double variable `value`. Finally, push `value` onto the stack. This marks the end of the if block.

Also, add a statement that prints out the current stack contents at the bottom of the while block. Then run your program and it should show the numeric tokens being pushed onto the stack.

Step 4. Process operators

Now we will add an else block to our previous if statement. This is to handle when the token is not a digit, i.e. it is either +,-,*,/

This else block should:

a. pop an item from the stack, and store in `op2` (the first item off is the last item in the operation)
b. do it again and store in op1
c. Then construct four more if statements inside this else block so that if `symbol` is '+' you will `push` onto `numStack` the sum of `op1+op2`,
d. and so on for the other operations.

Finally, after the whole traversal loop finishes processing the expression, pop off the last item from the stack (it will be the final answer) and print it out.

Try it out on a few expressions.

When finished, call the instructor over to get signed off.

EXTRA POINTS (3 points) STACK SORT

Consider the following algorithm to sort the entries in a stack S_1. First create two empty stacks, S_2 and S_3. At any given time, stack S_2 will hold the entries in sorted order, with the smallest at the top of the stack. Move the top entry of S_1 to S_2. Pop and consider the top entry *t* of S_1. Pop entries of stack S_2 and push them onto stack S_3 until you reach the correct place to put *t*. Then push *t* onto S_2. Next move all the entries from S_3 to S_2.

a. Write a Java implementation of this algorithm.

b. Consider the following revision of this algorithm. After moving the top entry of S_1 to S_2, compare the new top entry *t* of S_1 with the top entry of S_2 and the top entry of S_3. Then either move entries from S_2 to S_3 or from S_3 to S_2 until you locate the correct position for *t*. Push *t* onto S_2. Continue until S_1 is empty. Finally, move any entries remaining in S_3 to S_2. Implement this revised algorithm.