COSC 1P03 – LAB 5 – Generics

You couldn’t find a (easy-to-find-item) without a Map!  
…Maps. We’re going to make a simple Map as a means of better understanding Generics.

# The heck is a Map?

Yeah, this is one of those annoying things: we take a word that already has a meaning, and then apply it to a *completely* different meaning. Not at all confusing! If helps, there’s also a different name for these things: *dictionaries*!

…yes, that is indeed the same problem.

But both terms actually do make sense!

* In Math, a *map* (or mapping) is an association or function correlating entries between two domains
  + That’s a fancy way of saying you could pick some entry from one domain, and use that to identify a corresponding entry from another
  + In a sense, you could say Java arrays nearly fit this description: the subscript is just an identifier within an integer domain, and by using it to index the array, you get a corresponding entry wherever the array’s data is actually stored
* Similarly, we often colloquially use the term *dictionary* when we have some term (e.g. word in English), and want to look up its corresponding definition (some other record)

For both cases, it’s using ‘one label to identify/retrieve’ an entry. The only thing that makes it special is we generally use text (or occasionally other options) as the identifier.

# Deciding on the types

Before we start working on how to *make* a Map, we’d best understand the problem itself:

* We want to associate *key→record* as a pair

In 2P03, you’ll cover *efficient* means of creating these mappings (in *several* ways). For now, we can go simple.  
But we need to start with identifying precisely ‘what’ maps to ‘what’.

Technically, we could map *anything* to *anything*. The latter ‘anything’ is nice, but the former is overkill. *Well* over 95% of the time, our key is just a String. That said, we *don’t* want to restrict our stored record type to being only Strings, so that means our best choice is *parametric polymorphism* (or *Generics*, if you don’t like fancy-talk).

Basically, we’ll say that the Strings *do* map to some specific type, but that said type may be different for each instance of the collection. And… that’s it! We’re done a major step!

# The specification

We could decide to create an interface, or not. Let’s say yes?  
First, the operations:

* Of course, we need to be able to add records. We need both the key and the entry
  + We need to decide how to handle duplicates: perhaps we *clobber* the old entry?
  + We should account for the possibility of *overflow* if we run out of space
* We need to be able to get an *entry*, based on its key
  + If the requested entry doesn’t exist, this represents an *underflow*
* We might want to be able to delete an entry, based on its key
  + If the entry existed, it’s debatable whether we should expect to receive it back. Let’s say no?
  + If the entry didn’t exist, should that be an error condition? Let’s say no. We just want to ensure ‘no records exist for that key’
* Maybe we’d like to know a count of how many entries there are?
* It wouldn’t be unusual to get a collection of some sort representing the existing keys in it
* It’s conceivable we’d want to be able to readily **traverse** every entry within it. This one will be… interesting

So maybe we formalize this into an interface? I’ll call mine Associative, after the equivalent type in PHP:

package storage;

/\*\*

\* Mapping from key->entry pairs.

\* @see OverflowException which y'all can figure out on your own

\* @see UnderflowException (ditto)

\*/

public interface AssociativeArray {

/\*\*

\* Adds an entry to the collection. Replaces duplicates by key.

\* @param key

\* @param entry

\* @throws OverflowException if there's no more space

\*/

public void add(String key, Object entry);

/\*\*

\* Retrieves and returns the requested record.

\* @param key

\* @return

\* @throws UnderflowException if the key doesn't exist

\*/

public Object get(String key);

/\*\*

\* Deletes the associated entry (if it exists).

\* @param key

\*/

public void delete(String key);

/\*\*

\* Sucks your blood?

\* @return

\*/

public int count();

/\*\*

\* Jingles to amuse small children

\* @return

\*/

public String[] keys();

}

I guess it’s time to address the elephant in the room, eh?

# A very specific lack of specificity

So far, we’re using Object as the entry type. As mentioned in lecture, that *was* the ‘correct’ way to do this once.  
Note: *was*. Now, we have access to Generics! First, we add a *type parameter* to the class’s header:

public interface AssociativeArray<E> {

And then we replace *every* instance of Object with E! (It’s literally just *two* words. I have faith in you)

Next, we’ll get started on the actual implementation!

# The concrete class, featuring generic arrays!

Make a class to implement the interface, say AssociativeArray. Here’s the header:

public class AssociativeArray<E> implements Associative<E> {

So. Instance variables! Obviously we need to keep track of the count, but what do we *store* the entries in?  
Let’s go with an array, say arr?

private E[] arr; //Is this even possible?

We’ll get to the ‘possible’ part in a moment. First, add a default constructor, to *chain* with a capacity of 100.  
More interestingly:

@SuppressWarnings("unchecked")

public AssociativeArray(int capacity) {

count=0;

arr = (E[])new Object[capacity];

}

There’s a bit to unpack here:

* Recall from lecture that we can have a ‘generic array reference’, but can’t actually *create* a generic array. We get around this by creating an array of Object references, knowing that it can still hold references of whatever other type (since those ‘whatevers’ will still be of subtype Object)
  + There are still odd limitations here and there. For example, it would be very tedious to have a function creating and returning an E[] array, since E[] is *not* a subtype of Object[]
* We’re doing something that would *normally* be sketchy (the cast), but we can tell the compiler to stop warning

We’re far from ready to continue though, since we’re still missing something major: a means of holding all of the keys!

* Since we have a finite upper-bound on the *number of entries*, we can do the same for *keys*
  + Basically, we *also* need a String array, with the same length. I’ll call mine keys?

We’ll look at individual operations next, but let’s do them in a different order?

## Counting the entries

This one’s trivial. Despite having no get prefix, this is just an *accessor*:

public int count() {

return count;

}

That’s it.

## Getting a requested entry

One of two things are true: the requested key exists in the keys array, or it does not. Let’s find out:

public E get(String key) {

for ( int i=0; i<count; i++ ) {

if ( keys[i].equalsIgnoreCase(key) )

return arr[i];

}

throw new UnderflowException(); //If it doesn’t exist... oops!

}

## Getting the keys

The only thing notable here is we need to create an array sized to how many entries we actually have:

public String[] keys() {

String[] k=new String[count]; //Safe, even for zero entries!

for ( int i=0; i<count; i++ )

k[i] = keys[i];

return k;

}

## Adding a key:entry pair

This is where things start getting interesting, but the biggest concern is simply being *careful*.  
Specifically, how do we address *overflow*?

* If we have a capacity of 100, and have 100 entries stored, will attempting an add trigger an overflow?
  + **Maybe!** 
    - And yes, that *is* the most correct answer!

Why is that?  
Because remember the special case: if we try adding a pair with a key that already exists, we *replace* that entry!

So all we need to do is to traverse the array, looking for the key. If we stop on a legal index, then that’s where the corresponding entry goes (irrespective of whether it’s a replacement). We then *might* increment the count:

public void add(String key, E entry) {

int index=0;

while ( index<count && !keys[index].equals(key) )

index++;

if ( index==keys.length ) throw new OverflowException();

arr[index] = entry;

keys[index] = key; //Might be redundant, but always safe!

if ( index==count ) count++; //If it's a new key

}

Can you see what each of the terms in the loop’s condition are for? Can you see the reasoning behind the ‘short-circuit and’? Can you see why we aren’t explicitly checking against the array’s capacity?

Anyhoo, all that’s left is deletion, so let’s get to that!

## Delete! Delete! Delete!

Since we’re talking about contiguous storage, besides removing the specified entry, we also need to *shift* everything that followed it up by one:

public void delete(String key) {

int index = 0;

while ( index<count && !keys[index].equals(key) ) index++;

if ( index<count ) {

for ( int i=index; i<count-1; i++ ) { //Shift other records up 1

keys[i] = keys[i+1];

arr[i] = arr[i+1];

}

count--;

keys[count]=null;

arr[count]=null; //Garbage collector-friendly!

}

}

And that’s kinda it! We have a Map now!

Except… it still feels like something is missing, right? But what could that possibly be? Maybe traversal?

# Traversal!

This one’s interesting because technically we’re already covered: in our client code, we could always just request all the keys, and then get each of those keys, right? To be clear: that is *entirely* valid!

But it’s not the only way we’ve seen for ‘traversing each member of a collection’, is it?  
In fact, even back in 1P02, we saw a very clean mechanism that afforded us the ability to *not even care* about *how* the collection was organized!

For-each loops! … can we use those here?

public Demo() {

Associative<String> arr = new AssociativeArray<>(10); //import the package!

for ( int i=0; i<10; i++ ) {

arr.add(Integer.toString(i+1),Character.toString('A'+i));

}

arr.add("10","Joe"); //testing an update

for (String k:arr.keys()) System.out.print("["+k+":"+arr.get(k)+"]");

System.out.println(); //The keys()-based seems to work!

for (String entry:arr) System.out.println("> "+entry); //...why won't this compile?

}

The “this won’t even compile” aspect probably means this won’t work. But why not?  
Recall that even in 1P02 you were sort of told how for-each loops work:

* An iterator — a single-use class specialized to know how to traverse the collection — is instantiated
* A loop checks the iterator for whether or not there are remaining entries left to be considered
* If so, the next member is retrieved (and any internal counter/pointer is incremented/advanced)

So that means we need:

1. To write an *Iterator*; an expert in traversing *our* collection
2. To somehow mark the collection as guaranteeing the ability to grant an Iterator

Let’s deal with #1 first.

Iterator is an interface under java.util, so import it and write a class to implement it:

class Associator<E> implements Iterator<E> {

Notice the visibility: *package-private*! A client will never *directly* instantiate it (we’ll see how to instantiate it later).  
For the instance variables, we need three things:

* A reference to the array of entries
* A count of how many entries there are total (since we’re not traversing the full array)
* A count of how far we are so far for *this* traversal

So *one* possible constructor:

Associator( E[] entries, int count ) {

this.entries=entries;

this.count=count;

index=0; //Always start at 'the start'

}

Any iterator must only answer two questions: are there more entries? And (if so) which one is next?

public boolean hasNext() {

return index<count;

}

and:

public E next() {

return entries[index++];

}

But how do we *use* that iterator? And how does that tie into the for-each loop?  
We may use a for-each loop for any collection that *is-A* Iterable:

public interface Associative<E> extends Iterable<E> { //Adjusting the interface!

This promises we’ll include an iterator() function:

public Iterator<E> iterator() {

return new Associator<>(arr,count);

}

(Of course, this goes into the implementation class!)

So wait, what does this tell us about for-each loops?  
They’re a good example of *syntactic sugar*: an extra feature in the language that doesn’t change the underlying functionality at all, but does make writing for it a bit more ‘palatable’. Autoboxing/autounboxing is another example.

So the following two code snippets are functionally identical:

for (String entry:arr)

System.out.println("> "+entry);

and:

java.util.Iterator<String> iter = arr.iterator();

while (iter.hasNext()) {

String entry = iter.next();

System.out.println("> "+entry);

}

Of course, this also means as far back as 1P02, we *could* have been using Generics had we wanted to:

Picture pic = new Picture();

Iterator<Pixel> iter = pic.iterator();

while ( iter.hasNext() ) {

Pixel p = iter.next();

p.setGreen( p.getGreen()/2 );

}

Anyhoo, so long as you’ve been following along (and *not* skimming), this is objectively the simplest lab.  
So, one final task (or pair of tasks?) for you…

Caveat emptor! As these are for the underlying mechanism powering for-each loops, they have (as they’re the source of) the same limitation: besides not needing to know the underlying organization, the client also doesn’t *get* to.  
In this case, by using this iterator, the client will never see the keys (and so e.g. couldn’t *modify* its existing entries via a for-each loop).

Show that you understand this!

Above, you have all the code you need to completely implement a Map/Dictionary/Associative Array, including *two* mechanisms for traversing! (One based on the keys, and one via iterator)

There’s also enough sample code provided that writing a trivial client to populate with some mock data should be easy.

Here’s what you should have no problem doing:

* After populating the collection, display the contents of it according to the ‘normal’ sequence defined by where the keys were added (i.e. changing nothing from the corresponding example above)
* Rewrite your iterator to display the entries in *reverse* sequence