UM Data Struct 2022/2023 Generics Lab Test (Thursday)

INTRODUCTION

One of the most common use cases of generics is in computation libraries. Computation libraries are collections of code designed to perform mathematical procedures, such as matrix arithmetic and Fourier transforms. These libraries need to be able to run on a variety of hardware devices without forcing users to change their code every time.

To achieve this, computation libraries use generic operations. These operations allow low-level operations, like addition and multiplication, to be performed differently on each device while keeping the same signature for higher-level operations. This means that the same code can be used across different hardware types without much modifications.

Imagine trying to write code for every possible hardware device; it would be a nightmare! Generics make it possible for computation libraries to be versatile and efficient, providing users with the high level tools they need to perform complex mathematical tasks quickly while taking care of the nitty gritty details of interfacing with different devices.

LAB TEST

In this lab test, you will implement a toy example of a library that does vector operations which runs on CPU as well as GPU. Unfortunately, FSKTM computers are not equipped with fancy GPUs. Therefore, we will just print some debug statements to "pretend" the code is being run on different devices.

This lab question consists of 7 pages, and is divided into 4 tasks.**

- In Task 1, you will create the data classes and interface.
- In Task 2, you will begin on the vector utility library with methods to construct and print vectors.
- In Task 3, you will create vector multiplication operations.
- In Task 4, you will create the dot product.

HINTS

- First, you are strongly encourage to read and understand the code below before you start Task 1.
- Second, please use the CHECKLIST in Page 6 and 7 as a step by step guidance to prepare your solution.

Allowed imports and testing code

Copy this code snippet into your IDE. These are the imports that are allowed for this lab as well as the tester code that will be used to evaluate your answer. **The TODO line is the only line that will be modified when your final code is marked** so make sure you stick exactly to the method names used here. You may add your own lines for debugging purposes in your own copy. However, do make sure that your code ultimately produces the correct output on this master copy.

```
import java.util.*;
import java.lang.*;

public class TestDS2023Generics
{
    public static int progress = 1; // TODO: Change this value as you
proceed through parts of the question.
    public static void main (String[] args) throws java.lang.Exception
    {
        testCPU();
        testGPU();
    }
    public static void testCPU () throws java.lang.Exception
    {
        System.out.println("=== CPU Class constructors, conversions and
string representation ===");
        CpuInt a = new CpuInt(3);
        CpuInt b = new CpuInt().fromInteger(5);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.printf("a.toInteger() -> class: %s, value: %d\n",
a.toInteger().getClass(), a.toInteger());
        System.out.println("---");

        if (progress < 2) { return; }

        System.out.println("=== CPU Vector constructors and printing
===");
        ArrayList<CpuInt> aa = VecUtil.arr2vec(CpuInt.class, new int[]
{1,2,3});
        ArrayList<CpuInt> bb = VecUtil.ones(CpuInt.class, 3);
        VecUtil.printVec(aa);
        VecUtil.printVec(bb);
        System.out.println("---");

        if (progress < 3) { return; }

        System.out.println("=== CPU Vector multiplications ===");
        bb = VecUtil.mul(bb, new CpuInt(3)); // Vec * Scalar
        ArrayList<CpuInt> cc = VecUtil.mul(aa, bb); // Vec * Vec
(elementwise multiplication)
        VecUtil.printVec(bb);
        VecUtil.printVec(cc);
        System.out.println("---");

        if (progress < 4) { return; }

        System.out.println("=== CPU Vector Dot Product ===");
        System.out.println(VecUtil.dot(aa, bb));
        System.out.println("---");
    }
    public static void testGPU () throws java.lang.Exception
    {
```

```
        System.out.println("=== GPU Class constructors, conversions and
string representation ===");
        GpuInt a = new GpuInt(3);
        GpuInt b = new GpuInt().fromInteger(5);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.printf("a.toInteger() -> class: %s, value: %d\n",
a.toInteger().getClass(), a.toInteger());
        System.out.println("---");

        if (progress < 2) { return; }

        System.out.println("=== GPU Vector constructors and printing
===");
        ArrayList<GpuInt> aa = VecUtil.arr2vec(GpuInt.class, new int[]
{1,2,3});
        ArrayList<GpuInt> bb = VecUtil.ones(GpuInt.class, 3);
        VecUtil.printVec(aa);
        VecUtil.printVec(bb);
        System.out.println("---");

        if (progress < 3) { return; }

        System.out.println("=== GPU Vector multiplications ===");
        bb = VecUtil.mul(bb, new GpuInt(3)); // Vec * Scalar
        ArrayList<GpuInt> cc = VecUtil.mul(aa, bb); // Vec * Vec
(elementwise multiplication)
        VecUtil.printVec(bb);
        VecUtil.printVec(cc);
        System.out.println("---");

        if (progress < 4) { return; }

        System.out.println("=== GPU Vector Dot Product ===");
        System.out.println(VecUtil.dot(aa, bb));
        System.out.println("---");
    }
}
```

Task 1

In this task, you will be creating integer classes. With the help of the code below,

- (i) create the correct signature for the `NumberInterface` interface, and
- (ii) create two integer classes `CpuInt` and `GpuInt`.

Note that the `NumberInterface` will define what method signatures our classes expose to. This is important because we need the signatures to inform the compiler in our generic methods later.

(Task 1 partial code)

```
interface NumberInterface
// TODO (this signature is incomplete, and your Task 1(i) code start here)
{
    // Metadata
    // TODO

    // Conversions
    public T fromInteger(int value);
    public Integer toInteger();

    // Operations
    // TODO
}

// Object for running integer operations on CPU
class CpuInt
// TODO (this signature is incomplete, and your Task 1(ii) code start
here)
{
    // Constructors
    // TODO

    // Metadata
    @Override
    public String toString() {
        return String.format("%s[%d]" , getDevice(), this.value);
    }
    public String getDevice() {
        return "CPU";
    }

    // Conversions
    // TODO

    // Operations
    public CpuInt add(CpuInt o) {
        System.out.printf("CPU Debug: %d + %d = %d\n", this.value,
o.value, this.value + o.value);
        return new CpuInt(this.value + o.value);
    }
    public CpuInt mul(CpuInt o) {
        System.out.printf("CPU Debug: %d * %d = %d\n", this.value,
o.value, this.value * o.value);
        return new CpuInt(this.value * o.value);
    }
}

// Object for running integer operations on GPU
class GpuInt
// TODO (this signature is incomplete, and your Task 1(ii) code start
here)
{
    // Constructors
```

```
    // TODO

    // Metadata
    @Override
    public String toString() {
        return String.format("%s[%d]" , getDevice(), this.value);
    }
    public String getDevice() {
        return "GPU";
    }

    // Conversions
    // TODO

    // Operations
    public GpuInt add(GpuInt o) {
        System.out.printf("GPU Debug: %d + %d = %d\n", this.value,
o.value, this.value + o.value);
        return new GpuInt(this.value + o.value);
    }
    public GpuInt mul(GpuInt o) {
        System.out.printf("GPU Debug: %d * %d = %d\n", this.value,
o.value, this.value * o.value);
        return new GpuInt(this.value * o.value);
    }
}
```

Task 2

In this task, you will start on the vector utility library where it is assumed that ArrayList of **the integers (CpuInt and GpuInt)** as vectors. Again, with the aid of the code below, create two ways to construct vectors as followed:

- (i) `VecUtil.arr2vec` which takes an array of integers and outputs a vector on the correct device, and
- (ii) `VecUtil.ones` which instantiates a vector of ones with a given length.

Following this, create (iii) a printing utility `VecUtil.printVec` to print out vectors.

(Task 2 partial code)

```
// Contains vector operations that work for CPU and GPU numbers
class VecUtil {
    // Vector Constructors
    // TODO (Your Task 2(i & ii) code start here)

    // Printing Utility
    // TODO (Your Task 2(iii) code start here)
}
```

Task 3

In this task, you are required to create functions in `VecUtil` that able to perform the following mathematical operations:

- (i) multiply vectors with vectors in element-wise, and
- (ii) multiply vectors with scalars (the scalar is multiplied to each element of the vector).

Task 4

Finally, in this task, you are required to create functions in `VecUtil` that able to perform the following mathematical operation:

- (i) vector dot product

APPENDIX - CheckList

If you have implemented all the above tasks (Task 1-4) correctly, you will able to produce this Output

```
=== CPU Class constructors, conversions and string representation ===
a = CPU[3]
b = CPU[5]
a.toInteger() -> class: class java.lang.Integer, value: 3
---
=== CPU Vector constructors and printing ===
CPU[1, 2, 3]
CPU[1, 1, 1]
---
=== CPU Vector multiplications ===
CPU Debug: 1 * 3 = 3
CPU Debug: 1 * 3 = 3
CPU Debug: 1 * 3 = 3
CPU Debug: 1 * 3 = 3
CPU Debug: 2 * 3 = 6
CPU Debug: 3 * 3 = 9
CPU[3, 3, 3]
CPU[3, 6, 9]
---
=== CPU Vector Dot Product ===
CPU Debug: 1 * 3 = 3
CPU Debug: 2 * 3 = 6
CPU Debug: 3 * 3 = 9
CPU Debug: 0 + 3 = 3
CPU Debug: 3 + 6 = 9
CPU Debug: 9 + 9 = 18
CPU[18]
---
=== GPU Class constructors, conversions and string representation ===
a = GPU[3]
b = GPU[5]
a.toInteger() -> class: class java.lang.Integer, value: 3
---
```

```
=== GPU Vector constructors and printing ===
GPU[1, 2, 3]
GPU[1, 1, 1]
---
=== GPU Vector multiplications ===
GPU Debug: 1 * 3 = 3
GPU Debug: 1 * 3 = 3
GPU Debug: 1 * 3 = 3
GPU Debug: 1 * 3 = 3
GPU Debug: 2 * 3 = 6
GPU Debug: 3 * 3 = 9
GPU[3, 3, 3]
GPU[3, 6, 9]
---
=== GPU Vector Dot Product ===
GPU Debug: 1 * 3 = 3
GPU Debug: 2 * 3 = 6
GPU Debug: 3 * 3 = 9
GPU Debug: 0 + 3 = 3
GPU Debug: 3 + 6 = 9
GPU Debug: 9 + 9 = 18
GPU[18]
---
```