JAVA PROGRAMMING LANGUAGE - 36B
BERKELEY CITY COLLEGE
SPRING 2019

LAB B

A. Concurrent Programming

Write a program to calculate the distance between two arrays points p and q of size 10,000,000 points each.

Create a class called Point that contains the coordinates x, y, and z. Create two arrays of points p and q of size 10,000,000 each. Use random double to generate the coordinates of the points. Write a program that calculates the distance between the points using the following formula.

Distance = $\text{sqrt}((x1-x2)^2 + (y1-y2)^2 + (z1-z2)^2)$

First, calculate the computation time using a single thread.

Then write a multi-threaded program that recursively divides the calculations into smaller sections of arrays of size N calculations using the Fork/Join framework. Use different values for N and see if there is any difference in the time it takes to complete the calculations. Try for N = 100, 1000, 10000, 100000, and 1000000. You can use System.nanoTime() to get the system time before and after the computations.

B. Generics

Write a set of classes that are needed to manage the library items and the checkout cart for a library. The library has a collection of books and DVDs that can be borrowed.

Start with a class LibraryItem that contains the following fields:

- Unique ID for all library items
- void setDescription(String s)
- String getDescription()

interface Borrowable{

- void setBorrowerID(String s);
- void setBorrowDate(Date d);
- void setReturnDate(Date d);

}

Then create subclass BorrowableItem that implements the above Borrowable interface. Next, derive the subclasses Book and DVD from the BorrowableItem. Add a field called book title for the Book and movie title for the DVD. Books and DVDs implement the Borrowable interface that contain the following methods:

Create the class LibCollection<E> that contains the following methods:

- boolean add(E o) // Add the element E
- boolean addAll(Collection<? extends E> c) // Add the contents of c to
  // LibCollection
- boolean remove (Object o) // Remove the element o
- void clear() // Remove all elements
- boolean removeAll(LibCollection<? Super E> c)// Remove the elements found in
  // LibColletction from list c
- retainAll(LibCollection<?> c) // Remove the elements not in c
- boolean contains(Object o) // true if element is present
- boolean containsAll(LibCollection<?> c) // True if all elements of c are present
- boolean is Empty() // True if no elements are present
- int size() // Number of elements in collection
- Iterator<E> getIterator() // Get an iterator over the elements
- Object[] toArray() // Copy the contents to an Object{}
- <T> T[] toArray(T[] a) // copy contents to a T[]

The Iterator is an interface with the following methods:

```
Interface Iterator<E> {

        E getNext();                    // Get the next element, null if none.
        boolean hasNext()               // True if more elements, false if no more
        boolean isEmpty                 // True if no elements are present, false otherwise
}
```

Write a tester class and test your implementation.

C. Lambda Expressions

Complete the following ListProcessor class. The Predicate, Consumer, and Function are three functional interfaces as shown below.

```
Class ListProcesser  {

        //filter method takes a List<T> of items and returns a List<T> of items that satisfies
        //Predicate<T>
        public static<T> List<T>  filter(List<T>, Predicate<T> p) {

                //Complete the method

        }
        //forEach method takes a List<T> of items and executes the Consumer<T> method for
        //every item in the List.
        public static<T> void  forEach(List<T>, Consumer<T> c) {

                //Complete the method

        }
        //map method takes a List<T> of items and the Function<T, R>, runs the Function for
        //every item in the List<T>, and returns a List<R> that contain the returns value from the
        //Function<T, R>
        public static<T, R> List<R> map(List<T> list, Function <T, R> f) {

                //Complete the method
        }
}

@functionalInterface
public interface Predicate<T>{
        boolean test(T t)
}

@functionalInterface
public interface Consumer<T>{
        void accept(T t)
}


@functionalInterface
public interface Function<T, R>{
        R apply(T t)
}
```

A) Create a IntegerListProcessor class that creates ArrayList of 25 random Integers between 0 and 1000.

   Using the IntegerListProcessor and appropriate (i) Methods that implement the functional interface and (ii) lambda expressions do the following:

- Print the list of all the numbers.
- Print the sum of all numbers.
- Print all the prime numbers
- Print the squares of all numbers in the list.
- Print the list of numbers between any two numbers x and y.

B) Create DateListProcessor that creates an ArrayList of 10 Date objects. Use a range of -50 to +50 for the coordinates x and y. Use the GregorianCalendar to create random dates. You can random numbers to generate the dates.

   Using the IntegerListProcessor and appropriate (i) Methods that implement the functional interface and (ii) lambda expressions do the following:

- Print all the Date objects.
- Sort and print the Date objects.
- Print the Date objects that are after a given date.
- Print all the dates within a given range of dates.

- Print all the dates non in a given range of dates.

D. Use the Reflection class to inspect the ConcreteClass given below.

BaseInterface.java

```
public interface BaseInterface {

    public int interfaceInt=0;

    void method1();

    int method2(String str);
}
```

BaseClass.java

```
public class BaseClass {

    public int baseInt;

    private static void method3(){
        System.out.println("Method3");
    }

    public int method4(){
        System.out.println("Method4");
        return 0;
    }

    public static int method5(){
        System.out.println("Method5");
        return 0;
    }

    void method6(){
        System.out.println("Method6");
    }

    // inner public class
    public class BaseClassInnerClass{}

    //member public enum
    public enum BaseClassMemberEnum{}
}
```

ConcreteClass.java

```java
@Deprecated
public class ConcreteClass extends BaseClass implements BaseInterface {

    public int publicInt;
    private String privateString="private string";
    protected boolean protectedBoolean;
    Object defaultObject;

    public ConcreteClass(int i){
        this.publicInt=i;
    }

    @Override
    public void method1() {
        System.out.println("Method1 impl.");
    }

    @Override
    public int method2(String str) {
        System.out.println("Method2 impl.");
        return 0;
    }

    @Override
    public int method4(){
        System.out.println("Method4 overridden.");
        return 0;
    }

    public int method5(int i){
        System.out.println("Method5 overridden.");
        return 0;
    }

    // inner classes
    public class ConcreteClassPublicClass{}
    private class ConcreteClassPrivateClass{}
    protected class ConcreteClassProtectedClass{}
    class ConcreteClassDefaultClass{}

    //member enum
    enum ConcreteClassDefaultEnum{}
    public enum ConcreteClassPublicEnum{}

    //member interface
    public interface ConcreteClassPublicInterface{}
}
```

Use Java Reflection class on the ConcreteClass and do the following:

- Get the canonical name of the class
- Get the super classes
- Get the public interface members inherited from super classes and the public class and interface members declared by the class.
- Get all the classes and interfaces declared as members of the class (not inherited)
- Get the class modifiers
- Get the implemented interfaces
- Get the class annotations
- Get the public constructors and use the Constructor class to create a new instance
- Get the public fields and use the Field class to get and set the public fields.
- Get the public methods and use the Method class to invoke public methods.

E. Write a UnitTestDriver program that will allow you to create and test the Calculator program. Your program should contain the following classes:

- class Calculator: In addition to the constructor, the Calculator class should contain the following methods: add, subtract, multiply, and divide methods which take two doubles and return a double.

- class CalculatorUnitTester that contains the following. In addition to the constructor, the CalculatorUnitTester should contain the following methods:
  - An initialize method that is run before all tests (annotated by “@initialize”)
  - A cleanup method that is run after all tests (annotated by “@cleanup”)
  - A set up method that is run before every test (annotated by “@before”)
  - A finish test method that is run after each test (annotated by “@after”)
  - A suite of test methods that will set the calculator methods that look like the following:

    ```
    @test
    void testAdd(){
        Calculator c = new Calculator();
        double expected = 30;
        double result = c.multiply(5, 6);
        if (Math.abs(expected – result) < 0.000000001)
                System.out.println(“TestAdd succeeded”)
       else
                Sytem.out.println(“TestAdd failed”)
    }
    ```

    The test methods are annotated by either @test or @ignore.

- class UnitTestDriver that creates an instance CalculatorUnitTester, inspects the CalculatorUnitTester methods and run the tests in the following sequence:

1) Run the constructor
2) Run all the methods that is annotated with “@initialize”
3) Run the series of test methods annotated with “@test”. Each test is run in the following sequence:
    a. Run the method annotated with “@before”
    b. Run the method annotated with “@test”
    c. Run the method annotated with “@after”
4) Run the method that is annotated with “@cleanup”

To make life easy, the initialize, cleanup, setup test, and finish test methods can simply dummy System.out.println(“*Whatever you want to say……*”) code.