Exercise 9

Objectives

The aim of this exercise is to explore the concept of binding. In addition, you will learn design patterns that encourage class reuse.

A: Sorting Machine [3 pt]

The following SortingApplication class is an application that sorts *N* randomly generated integers in ascending order. The Judge class generates the data and verifies the validity of your program.

The SortingMachine class sorts the given data. However, the SortingMachine can be configured with a constructor or setStrategy method to specify the sorting algorithm before the sort is executed.

```
class SortingApplication{
    private static int N = 1000;

    public SortingApplication(){
        Judge judge = new Judge(N);
        int[] data = new int[N];

        SortingMachine machine = new SortingMachine(new SelectionSort());
        judge.setData(data);
        machine.sort(data);
        judge.validate(data);

        machine.setStrategy(new BubbleSort());
        judge.setData(data);
        machine.sort(data);
        judge.validate(data);
    }

    public static void main(String[] args){
        new SortingApplication();
    }
}
```

The following is the SortingMachine class. When sort is executed, it outputs information about this sorting object (its name) and then executes the sort method associated with the strategy.

```
class SortingMachine{
    protected Strategy strategy;

    public SortingMachine(Strategy s){ strategy = s; }

    public void setStrategy(Strategy s){ strategy = s; }

    public void sort(int[] data){
        System.out.println(strategy.getClass().getName());
        strategy.sort(data);
    }
}
```

The following is the Judge class. You must use this class as is.

```
import java.util.Random;
import java.util.Arrays;

class Judge{
    private int N;
    private int[] in, out;

    public Judge(int N){
        this.N = N;
        Random random = new Random();
        in = new int[N];
        for ( int i = 0; i < N; i++ )
            in[i] = random.nextInt(1000000000);
        out = in.clone();
        Arrays.sort(out);
    }

    public void setData(int[] data){
        for ( int i = 0; i < N; i++ ) data[i] = in[i];
    }

    public void validate(int[] data){
        String res = "Yes";
        for ( int i = 0; i < N; i++ )
            if ( data[i] != out[i] ) res = "No";
        System.out.println(res);
    }
}
```

Your task is to create SelectionSort and BubbleSort classes that implement Selection Sort and Bubble Sort, respectively, to complete this application. If you get the following output from the execution of SortingApplication, your program is considered correct. Note that you should not modify the contents of the above three classes.

Sample Output
SelectionSort Yes BubbleSort Yes

Submission Files	Types
SortingMachine.java	Java Class
SortingApplication.java	Java Class
Judge.java	Java Class
SelectionSort.java	Java Class
BubbleSort.java	Java Class
Strategy.java	Java Interface

B: Sorting Machine+ [4 pt]

Extend the application created in Problem A to develop the following SortingApplicationPlus class.

```
import java.util.Scanner;

class SortingApplicationPlus{
   private int N;

   public SortingApplicationPlus(){
      Scanner sc = new Scanner(System.in);
      N = sc.nextInt();
      String algorithm = sc.next();

      SortingMachine machine = new AdvancedSortingMachine(new SelectionSort());

      if ( algorithm.equals("bubble") ){
         machine.setStrategy(new BubbleSort());
      } else if ( algorithm.equals("merge") ){
         machine.setStrategy(new MergeSort());
      }

      int[] data = new int[N];
      Judge judge = new Judge(N);

      judge.setData(data);
      machine.sort(data);
      judge.validate(data);
   }

   public static void main(String[] args){
      new SortingApplicationPlus();
   }
}
```

This application is extended from two perspectives

- It extends the SoringMachine with the AdvancedSortingMachine
- The addition of the MergeSort class broadens and extends the choice of algorithms.

Your task is to develop the AdvancedSortingMachine class and the MergeSort class. As a new function the sort method of the AdvancedSortingMachine class must measure and output the execution time required for sorting. To do so you can use System.currentTimeMillis() at before and after the sorting operation.

Check the following input/output examples for verification.

Sample Input	**Sample Output**

`30000 selection`	`SelectionSort`
	`time: 445`
	`Yes`

`30000 bubble`	`BubbleSort`
	`time: 1408`
	`Yes`

`30000 merge`	`MergeSort`
	`time: 13`
	`Yes`

`100000 selection`	`SelectionSort`
	`time: 4769`
	`Yes`

`100000 merge`	`MergeSort`
	`time: 33`
	`Yes`

Although the measurement time will vary depending on the environment, the execution time of Merge Sort should be significantly faster than the other two sorting algorithms.

Submission Files	**Types**
`AdvancedSortingMachine.java`	`Java Class`
`SortingApplicationPlus.java`	`Java Class`
`Judge.java`	`Java Class`
`SelectionSort.java`	`Java Class`
`BubbleSort.java`	`Java Class`
`MergeSort.java`	`Java Class`
`Strategy.java`	`Java Interface`

C: Adapter [6 pt]

The ContainerApplication shown below is a simulator that creates Deque, Queue, and Stack objects and performs data insertion and deletion for them. Deque is a special queue that can add and delete data from both forward and backward respectively.

```
import java.util.Scanner;

class ContainerApplication{
   public static void main(String[] args){
      new ContainerApplication().run();
   }

   public void run(){
      Scanner sc = new Scanner(System.in);
      Deque deque = new DequeImplByDLL();
      Queue queue = new QueueImplByDeque();
      Stack stack = new StackImplByDeque();

      while(true){
         String to = sc.next();  // target
         if ( to.equals("end") ) break;
         String com = sc.next(); // command
         if ( to.equals("deque") ){
            if ( com.equals("insertFront") ) {
               deque.insertFront(sc.nextInt());
            } else if ( com.equals("insertBack") ){
               deque.insertBack(sc.nextInt());
            } else if ( com.equals("removeFront") ){
               deque.removeFront();
            } else if ( com.equals("removeBack") ){
               deque.removeBack();
            } else if ( com.equals("front") ){
               System.out.println(deque.front() + " from the front of the deque.");
            } else if ( com.equals("back") ){
               System.out.println(deque.back() + " from the back of the deque.");
            }
         } else if ( to.equals("stack") ){
            if ( com.equals("push") ) {
               stack.push(sc.nextInt());
            } else if ( com.equals("pop") ){
               System.out.println(stack.pop() + " from the top of the stack.");
            }
         } else if ( to.equals("queue") ){
            if ( com.equals("enqueue") ) {
               queue.enqueue(sc.nextInt());
            } else if ( com.equals("dequeue") ){
               System.out.println(queue.dequeue() + " from the front of the queue.");
            }
         }
      }
   }
}
```

You can validate this program with the following data, for example.

Sample Input	Sample Output
deque insertFront 3 deque insertFront 2 deque insertFront 1 deque insertBack 4 deque insertBack 5 deque removeFront deque removeFront deque removeBack deque front deque back deque insertFront 8 deque insertFront 9 deque removeBack deque removeBack deque removeBack deque front deque back stack push 1 stack push 5 stack push 3 stack pop stack pop stack push 8 stack push 11 stack pop stack pop stack pop queue enqueue 100 queue enqueue 101 queue enqueue 102 queue dequeue queue enqueue 300 queue dequeue queue enqueue 301 queue enqueue 302 queue dequeue queue dequeue queue dequeue end	3 from the front of the deque. 4 from the back of the deque. 9 from the front of the deque. 9 from the back of the deque. 3 from the top of the stack. 5 from the top of the stack. 11 from the top of the stack. 8 from the top of the stack. 1 from the top of the stack. 100 from the front of the queue. 101 from the front of the queue. 102 from the front of the queue. 300 from the front of the queue. 301 from the front of the queue.

Your task is to implement the DequeImplByDLL, QueueImplByDeque, and StackImplByDeque classes used in this simulator. While this may seem like a tough task at first glance, the program you need to write is very limited. First, Stack and Queue can each be implemented using Deque. Furthermore, Deque can be implemented with LinkedList. Even luckier, you have the SimpleDoublyLinkedList class shown below.

```
class SimpleDoublyLinkedList {
    private SimpleNode nil;

    SimpleDoublyLinkedList(){
        nil = new SimpleNode();
        nil.setNext(nil);
        nil.setPrev(nil);
    }

    private void deleteNode(SimpleNode t){
        t.getPrev().setNext(t.getNext());
        t.getNext().setPrev(t.getPrev());
    }

    public void removeFront(){ deleteNode(nil.getNext()); }

    public void removeBack(){ deleteNode(nil.getPrev()); }

    public int front(){ return nil.getNext().getKey(); }

    public int back(){ return nil.getPrev().getKey();}

    public void addFront(int key){
        SimpleNode x = new SimpleNode();
        x.setKey(key);
        x.setNext(nil.getNext());
        nil.getNext().setPrev(x);
        nil.setNext(x);
        x.setPrev(nil);
    }

    public void addBack(int key){
        SimpleNode x = new SimpleNode();
        x.setKey(key);
        x.setPrev(nil.getPrev());
        nil.getPrev().setNext(x);
        nil.setPrev(x);
        x.setNext(nil);
    }
}
```

For the purposes of this exercise, SimpleDoublyLinkedList has been implemented in a very simplified manner, but please consider that this is essentially a very high-functioning, sophisticated and reliable class. In other words, it is reasonable to reuse this class as is. So, let's take advantage of this class to implement the above three data structures.

Although SimpleDoublyLinkedList has all the necessary functions, the interface to access them differs from the standard interface (how to call them) of the three data structures as shown below.

```
public interface Deque {
    void insertFront(int key);
    void insertBack(int key);
    void removeFront();
    void removeBack();
    int front();
    int back();
    int size();
    boolean empty();
}
```

```
public interface Stack{
    void push(int x);
    int pop();
    int top();
    int size();
    boolean empty();
}
```

```
public interface Queue{
    void enqueue(int key);
    int dequeue();
    int front();
    int size();
    boolean empty();
}
```

We cannot change the method names of SimpleDoublyLinkedList just because the interface is different (since we want to use it as it is). Therefore, we implement a concrete class of Deque as follows, embracing SimpleDoublyLinkedList.

```
class DequeImplByDLL implements Deque{
    private SimpleDoublyLinkedList list;
    private int n; // the number of elements in the deque

    DequeImplByDLL(){
        list = new SimpleDoublyLinkedList();
        n = 0;
    }

    /* your codes */
}
```

Furthermore, the interface of Stack is different from the interface of Deque. Therefore, the concrete class of Stack can be implemented as follows, embracing the concrete class of Deque (the same is true for Queue).

```
class StackImplByDeque implements Stack {
    private Deque deque;

    StackImplByDeque(){
        this.deque = new DequeImplByDLL();
    }

    /* your codes */
}
```

The following adapter patterns were employed in the design of this application.

- DequeImplByDLL class is an Adapter to SimpleDoublyLinkedList (Adaptee)
- StackImplByDeque class is an Adapter to DequeImplByDLL (Adaptee)
- QueueImplByDeque class is Adapter to DequeImplByDLL (Adaptee)

Submission Files	Types
ContainerApplication.java	Java Class
SimpleDoublyLinkedList.java	Java Class
SimpleNode.java	Java Class
DequeImplByDLL.java	Java Class
StackImplByDeque.java	Java Class
QueueImplByDeque.java	Java Class
Deque.java	Java Interface
Stack.java	Java Interface
Queue.java	Java Interface

Summary

In this exercise, we first reviewed the mechanism of dynamic bindings. Remember that in SortingApplicationPlus, we do not know which algorithm will be employed until we run the program. This means that object references can refer to different types of dynamically generated objects.

We further deepened our understanding of polymorphism through a variety of applications. The SortingApplicationPlus is very interesting because of its extensible design pattern. In this design, we can extend SortingMachine to enhance its functionally of the application. On the other hand, we can add new algorithms by just creating new classes (capsulized algorithm) in a flexible way (without changing SortingMachine).