Supervision 1

Exercise 1

(a) In functional languages, functions are the fundamental elements while in imperative languages data(variables) are. (b) In functional languages, we mostly use recursive function instead of loops. (c) In functional languages, function can act as an argument of other functions (and even itself), but in imperative languages, the argument of function should be data.

Exercise 2

(a) `int a;`

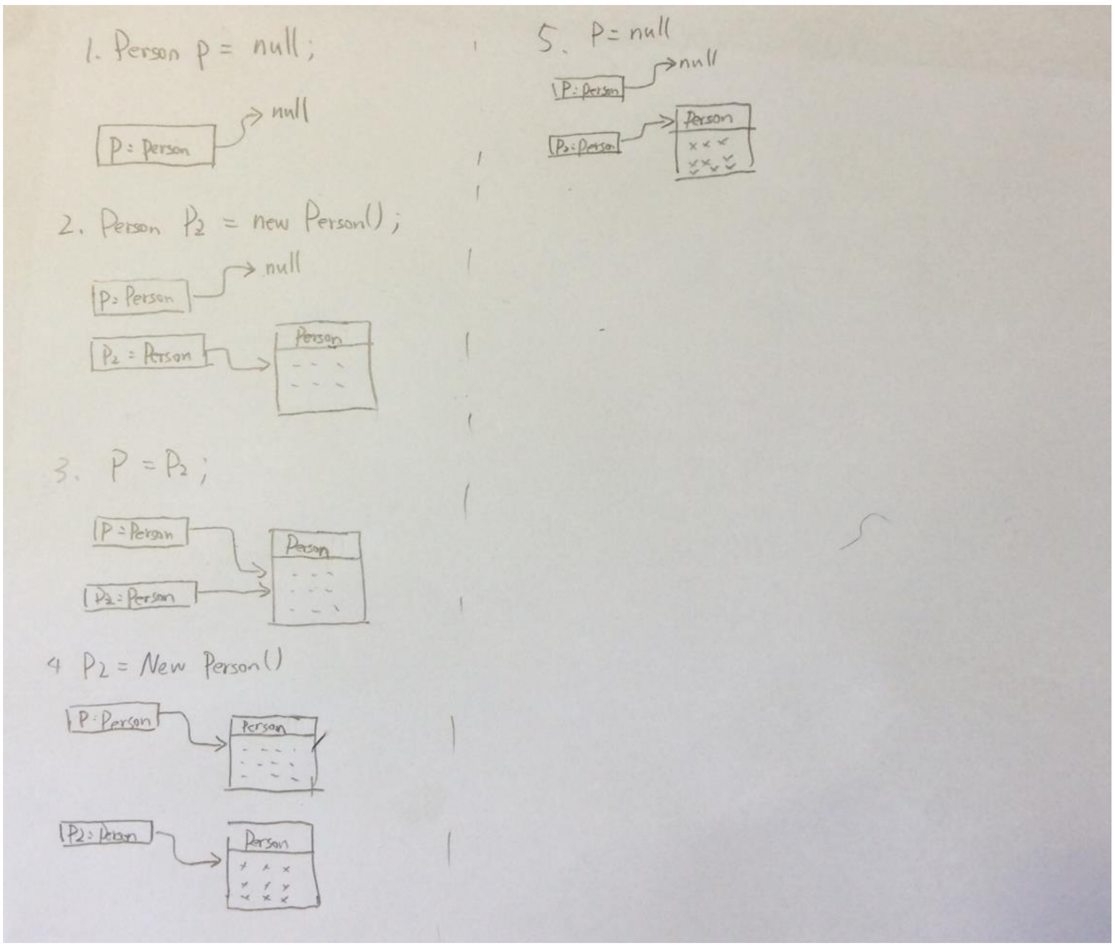
(b) `int[] b;`

(c)

```
class AClass{
    String msg = "This is a class"
    public print(){
        System.out.println(msg);
    }
}
```

(d) AClass mClass = new AClass();

Exercise 3



Exercise 4

```
int sum(int[] a){
    for (int num : a) sum += num;
}
```

```
int[] cumsum(int[] a){
    int ans[] = new int[a.length];
    ans[0] = a[0];
    for(int i = 1; i < a.length; i++){
        ans[i] = ans[i - 1] + a[i];
    return ans;
}
```

```
int positives(int[] a){
    int cnt = 0;
    for(int num : a){
        if(num > 0) cnt++;
    }
    int[] res = new int[cnt];
    int i = 0;
    for(int num : a){
        if(num > 0) res[i++] = num;
    }
    return res;
}
```

Exercise 5

```
float[][] matrix = new float[n][n];
```

Exercise 6

Because in the method vectorAdd, the variables x and y are primitives. It just create a copy of a and b, and do some calculation, but it won't change the value of a and b.

Exercise 7 and 8

LinkedList.java

```
public class LinkedList {
    protected Element head;

    public LinkedList(int x) {
        head = new Element(x);
    }
    public void add(int index, int x) throws LinkedListException {
        if (index < 0) throw new LinkedListException("out of
bound!");
        if (index == 0) {
            Element tmp = new Element(x);
            tmp.setNext(head);
        } else {
            Element now = head;
            for (int i = 1; i < index; i++) {
                now = now.getNext();
            }
            Element tmp = new Element(x);
            if(!now.isTail())tmp.setNext(now.getNext());
            now.setNext(tmp);
        }
    }

    public void remove(int index) throws LinkedListException {
        if (index < 0) throw new LinkedListException("out of
bound!");
        if (index == 0) {
            if (head.isTail()) throw new LinkedListException("List is
not Exist!");
            else head = head.getNext();
        } else {
            Element now = head;
            for (int i = 1; i < index; i++) {
                now = now.getNext();
            }
            if (now.getNext().isTail()) now.cutTail();

            else now.setNext(now.getNext().getNext());
        }
    }

    public int getHead() {
        return head.getVal();
    }

    public int getNth(int n) throws LinkedListException {
        if (n < 1) throw new LinkedListException("out of bound!");

        n--;
        Element now = head;
```

```java
        while (n != 0) {
            now = now.getNext();
            n--;
        }
        return now.getVal();
    }

    public int length() throws LinkedListException {
        if(hasCycle()) throw new LinkedListException("there is a
Cycle!");
        int cnt = 1;
        Element now = head;
        while (!now.isTail()) {
            now = now.getNext();
            cnt++;
        }
        return cnt;
    }

    public boolean hasCycle() throws LinkedListException {
        if (head.isTail()) return false;
        Element i = head.getNext();
        if (i.isTail()) return false;
        Element j = i.getNext();
        while (i != j) {
            if(j.isTail()) return false;
            j=j.getNext();
            if(j.isTail()) return false;
            j=j.getNext();
            i=i.getNext();
        }
        return true;
    }
    public static void main(String args[]){
        LinkedList linkedList = new LinkedList(1);
        try{
            linkedList.add(1,2);
            linkedList.add(1,3);
            linkedList.add(2,4);
            System.out.println(linkedList.hasCycle());
            System.out.println(linkedList.length());
            System.out.println(linkedList.getNth(2));
            linkedList.remove(1);
            linkedList.remove(2);
            System.out.println(linkedList.getNth(2));

        } catch(LinkedListException e){
            System.out.println(e.getMessage());
        }
    }
}
```

Element.java

```
Public class Element {
    private int val;
    private Element next;

    public Element(int x) {
        val = x;
        next = null;
    }

    public Element getNext() throws LinkedListException {
        if (next == null) throw new LinkedListException("out of 
bound!");
        return next;
    }

    public void setNext(Element x) throws LinkedListException {
        if (x == null) throw new LinkedListException("element doesn't 
exist!");
        next = x;
    }

    public void cutTail() {
        next = null;
    }

    public int getVal() {
        return val;
    }

    public boolean isTail() {
        if (next == null) return true;
        return false;
    }
}
```

LinkedListException.java

```java
public class LinkedListException extends Exception {
    public LinkedListException(String msg) {
        super(msg);
    }
}
```

Exercise 9

The stack is a First-In-Last-Out data structure, we can only access the data at the top of the stack. A typical stack has three methods: push(x) puts x at the top of the stack, top() returns the value of top element in the stack and pop() delete the top element.

Exercise 10

StackException.java

```java
public class StackException extends LinkedListException {
    public StackException(String msg) {
        super(msg);
    }
}
```

StackInt.java

```java
public interface StackInt {
    void pop() throws StackException;

    void push(int x) throws StackException;

    int top();
}
```

Stack.java

```java

public class Stack extends LinkedList implements StackInt{
    public Stack(int x) {
        super(x);
    }

    @Override
    public void pop() throws StackException {
        try {
            head = head.getNext();
        } catch (LinkedListException e) {
            throw new StackException("Empty Stack!");
        }
    }
    @Override
    public void push(int x) throws StackException {
        Element tmp = new Element(x);
        try {
            tmp.setNext(head);
            head = tmp;
        } catch (LinkedListException e) {
            throw new StackException("Nothing pushed");
        }

    }

    @Override
    public int top() {
        return head.getVal();
    }
}
```