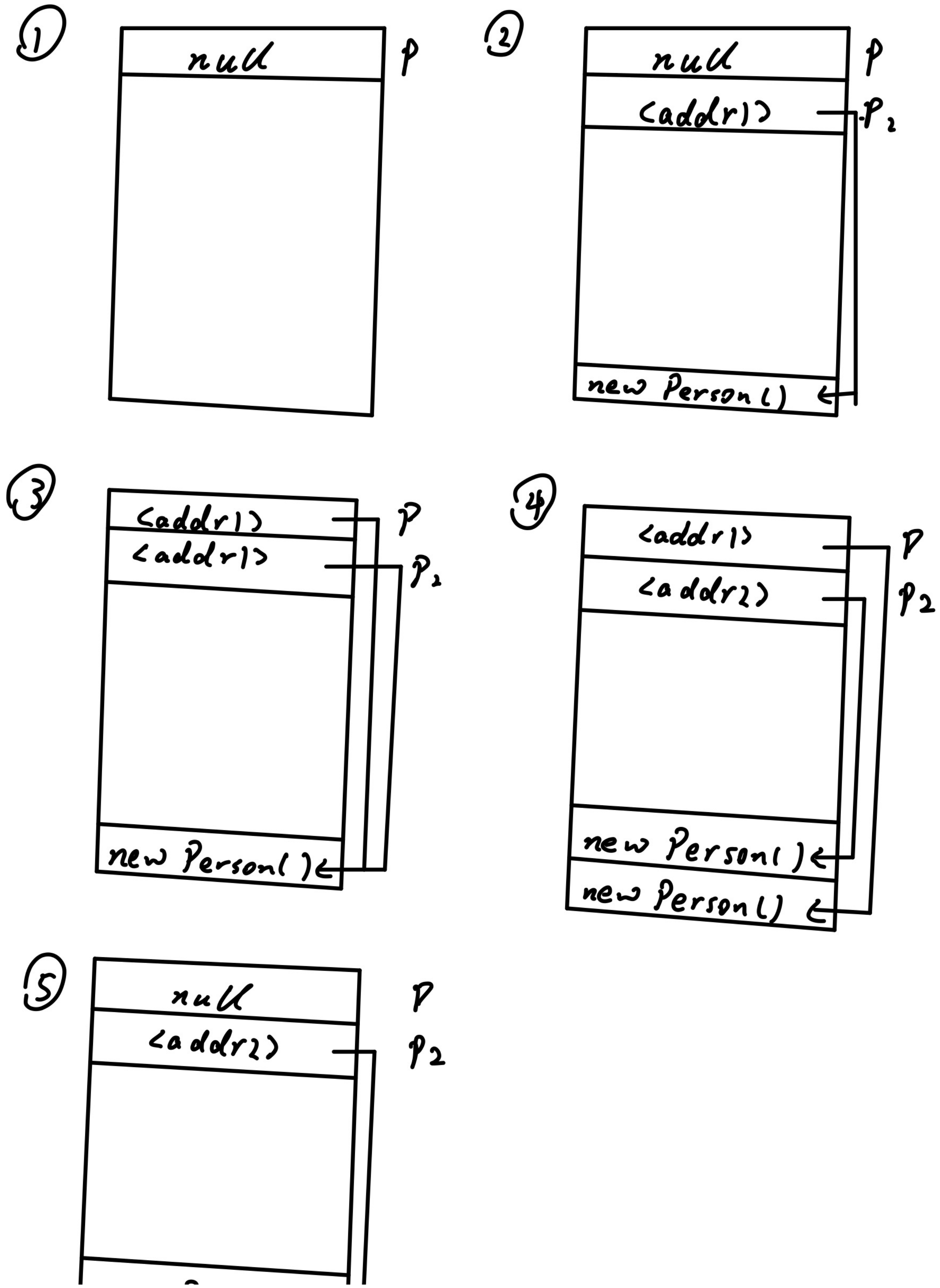
1. (1). In functional programming, everything is an expression, which is not the case in imperative programming.

   (2). Functional programming specifies what to do, not how to do it. Imperative programming specifies both what to do and how to do it.

   (3). Functional programming aims at avoiding or minimizing side effects, shared data, and mutable data while imperative programming can easily change the state of the computer.

   ```
   class Ex2 {
        int a;
        int[] b = new int[] {1};

        public static void main(String[] args) {
             Ex2 o = new Ex2();
        }
   }
   ```



3.

4.

```java
import java.util.ArrayList;

class Ex3 {
    int sum (int[] a){
        int s = 0;
        for (int x : a)
            s += x;
        return s;
    }

    int[] cumsum (int[] a){
        int s = 0;
        int[] b = new int[a.length];
        for (int i = 0; i < a.length; i++)
            b[i] = (s += a[i]);
        return b;
    }

    int[] positives (int[] a){
        ArrayList<Integer> b = new
ArrayList<Integer>();
        for (int x : a)
            if (x > 0)
                b.add(x);
        int i = 0;
        int[] c = new int[b.size()];
```

```
        for (int x : b)
            c[i++] = x;
        return c;
    }

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

```
class Ex5 {
    int n;

    float[][] matrix (int n){
        return new float[n][n];
    }
}
```

```
class Ex6 {

    int x, y;

    Ex6(int x,int y){
        this.x = x;
        this.y = y;
    }


```

```
    public void vectorAdd(int dx, int dy) {
        x=x+dx;
        y=y+dy;
    }

    public static void main(String[] args) {
        Ex6 o = new Ex6(0, 2);
        o.vectorAdd(1, 1);
        System.out.println(o.x+" "+o.y);
        // (a,b) is still (0,2)
    }
}
```

```
class SinglyLinkedList {
    class Element{
        int val;
        Element next;

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

    int length;
```

```
    Element head, tail;

    SinglyLinkedList(){
        length = 0;
        head = tail = null;
    }

    void Add(int x){
        Element node = new Element (x);
        if (head == null)
            head = tail = node;
        else {
            tail.next = node;
            tail = node;
        }
        length++;
    }

    void Remove(){
        head = head.next;
        length--;
    }

    int QueryHead(){
        return head.val;
    }
```

```
    int nth(int n, Element now){
        if(n == 1)
            return now.val;
        return nth(n - 1, now.next);
    }

    int nth(int n){
        return nth(n, head);
    }

    int QueryLength(){
        return length;
    }

    public static void main(String[] args){
        SinglyLinkedList list = new
SinglyLinkedList();
        list.Add(1);
        list.Add(2);
        System.out.println(list.QueryHead());
    }
}
```

```
boolean check(Element a, Element b){
  if (a == b)
      return true;
  return check(a.next, b.next.next);
}

boolean check(){
  return check(head, head.next);
}

//one pointer move one every time, one
pointer move two every time.
//if a cycle exists, one pointer must catch
up with another
```

9. A stack is a sequence such that items can be added or removed from the head only. A stack obeys a Last-In-First-Out (LIFO) discipline: the item next to be removed is the one that has been in the queue for the shortest time.

```
public class Stack<T>{
    class Element{
        T val;
        Element next;
```

```
        Element(T x){
            val = x;
            next = null;
        }
    }

    Element head;

    Stack(){
        head = null;
    }

    void Push(T x){
        Element node = new Element(x);
        node.next = head;
        head = node;
    }

    void Pop(){
        head = head.next;
    }

    T top(){
        return head.val;
    }
```

```
    public static void main(String[] args) {
        Stack<Integer> s = new Stack<Integer>
();
    }
}
```