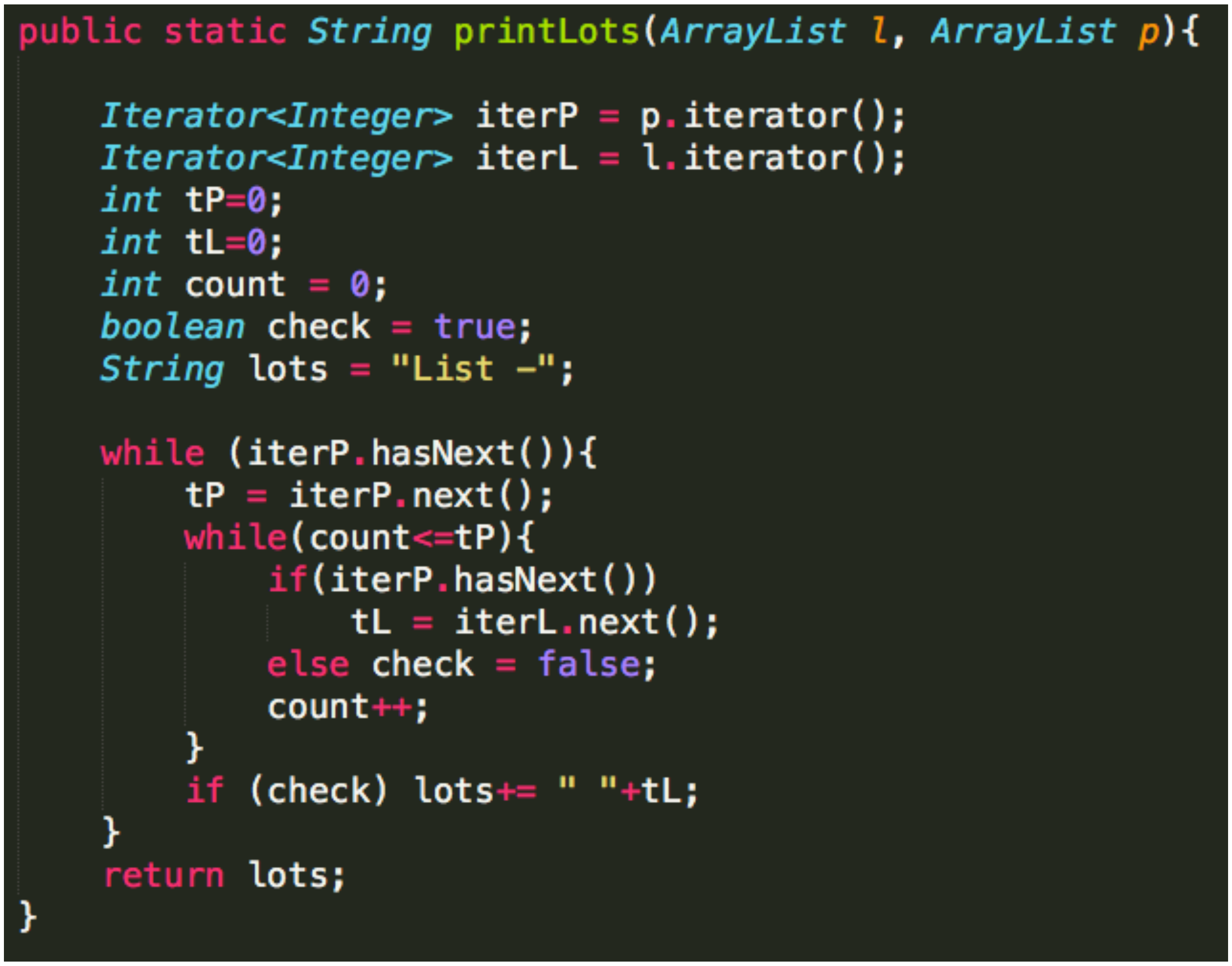
Ben Arbib (ba2490)

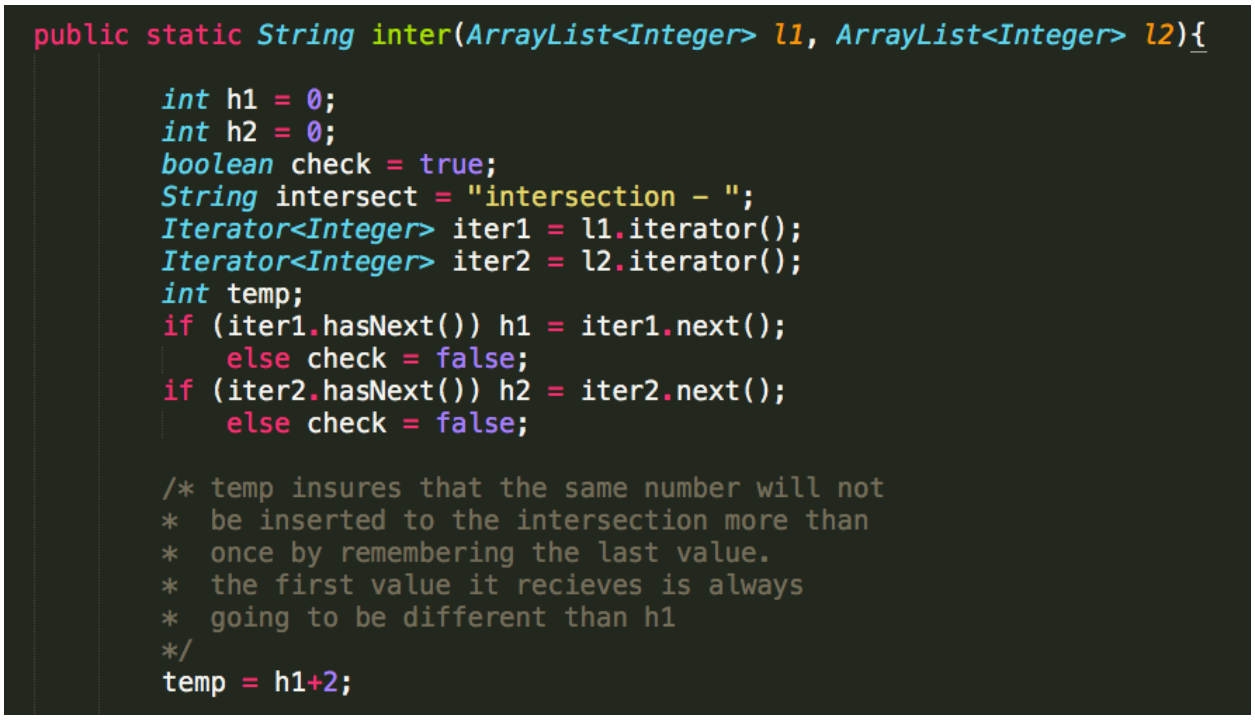
HW 2

1. (Screenshot of my printLots method) – in order to avoid runtime errors I used hasNext on both lists as a condition to continue. In case hasNext returns false the methods prints only the numbers it got so far. Also, if one of the hasNext returns false it: (1) means that P has reached its last spot or (2) the location in L does not exist. There is no reason to continue since both lists are in ascending order – therefore no numbers will be found after that. Although there are two while loops **<u>the efficiency is O(n)</u>** since it goes over each list only once.



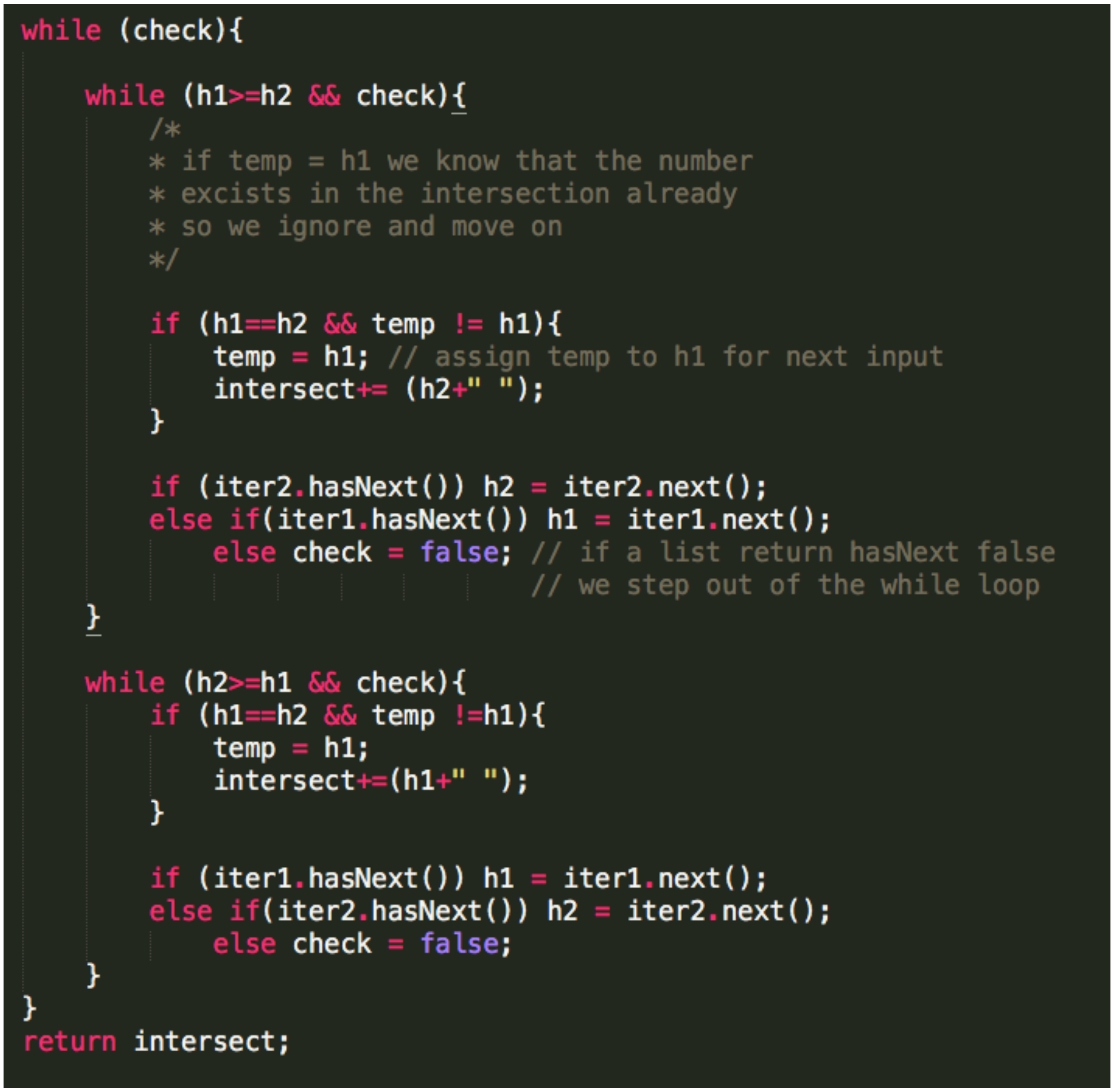
```java
public static String printLots(ArrayList l, ArrayList p){

    Iterator<Integer> iterP = p.iterator();
    Iterator<Integer> iterL = l.iterator();
    int tP=0;
    int tL=0;
    int count = 0;
    boolean check = true;
    String lots = "List -";

    while (iterP.hasNext()){
        tP = iterP.next();
        while(count<=tP){
            if(iterP.hasNext())
                tL = iterL.next();
            else check = false;
            count++;
        }
        if (check) lots+= " "+tL;
    }
    return lots;
}
```

2. (Screenshot of Weiss 3.4) – intersection of two lists. This is a method that receives two ArrayLists and returns a string with all integers that exist in both lists. **<u>the efficiency is O(n)</u>**. Each iterator goes over its list once.
    a. first part – receive ArrayLists, assign iterators, and first values



```
public static String inter(ArrayList<Integer> l1, ArrayList<Integer> l2){

        int h1 = 0;
        int h2 = 0;
        boolean check = true;
        String intersect = "intersection - ";
        Iterator<Integer> iter1 = l1.iterator();
        Iterator<Integer> iter2 = l2.iterator();
        int temp;
        if (iter1.hasNext()) h1 = iter1.next();
            else check = false;
        if (iter2.hasNext()) h2 = iter2.next();
            else check = false;

        /* temp insures that the same number will not
        *  be inserted to the intersection more than
        *  once by remembering the last value.
        *  the first value it recieves is always
        *  going to be different than h1
        */
        temp = h1+2;
```

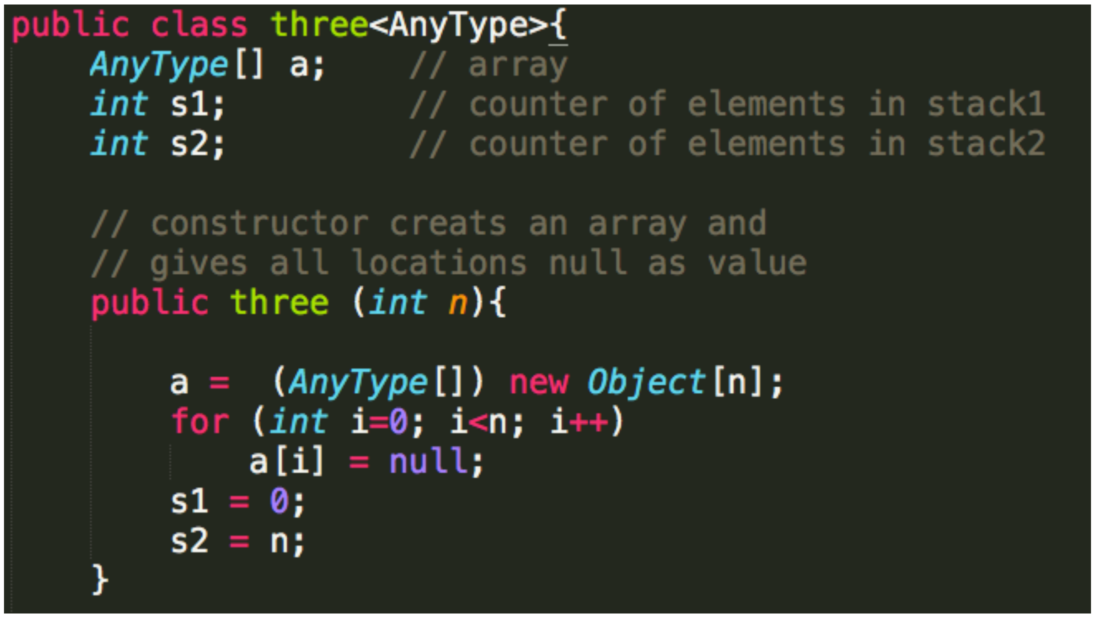
b. second part – run a loop until one of the lists reaches last element (we know that there aren't any similarities after that) when h1 has a greater value than h2 we give h2 the next element until they are equal or h2 is greater than h1. From this point h1 receives the next element. When they are equal the value is added to intersection. If the number exists already, indicated by temp, we move on.



```
while (check){

    while (h1>=h2 && check){
        /*
        * if temp = h1 we know that the number
        * excists in the intersection already
        * so we ignore and move on
        */

        if (h1==h2 && temp != h1){
            temp = h1; // assign temp to h1 for next input
            intersect+= (h2+" ");
        }

        if (iter2.hasNext()) h2 = iter2.next();
        else if(iter1.hasNext()) h1 = iter1.next();
            else check = false; // if a list return hasNext false
                                // we step out of the while loop
    }

    while (h2>=h1 && check){
        if (h1==h2 && temp !=h1){
            temp = h1;
            intersect+=(h1+" ");
        }

        if (iter1.hasNext()) h1 = iter1.next();
        else if(iter2.hasNext()) h2 = iter2.next();
            else check = false;
    }
}
return intersect;
```

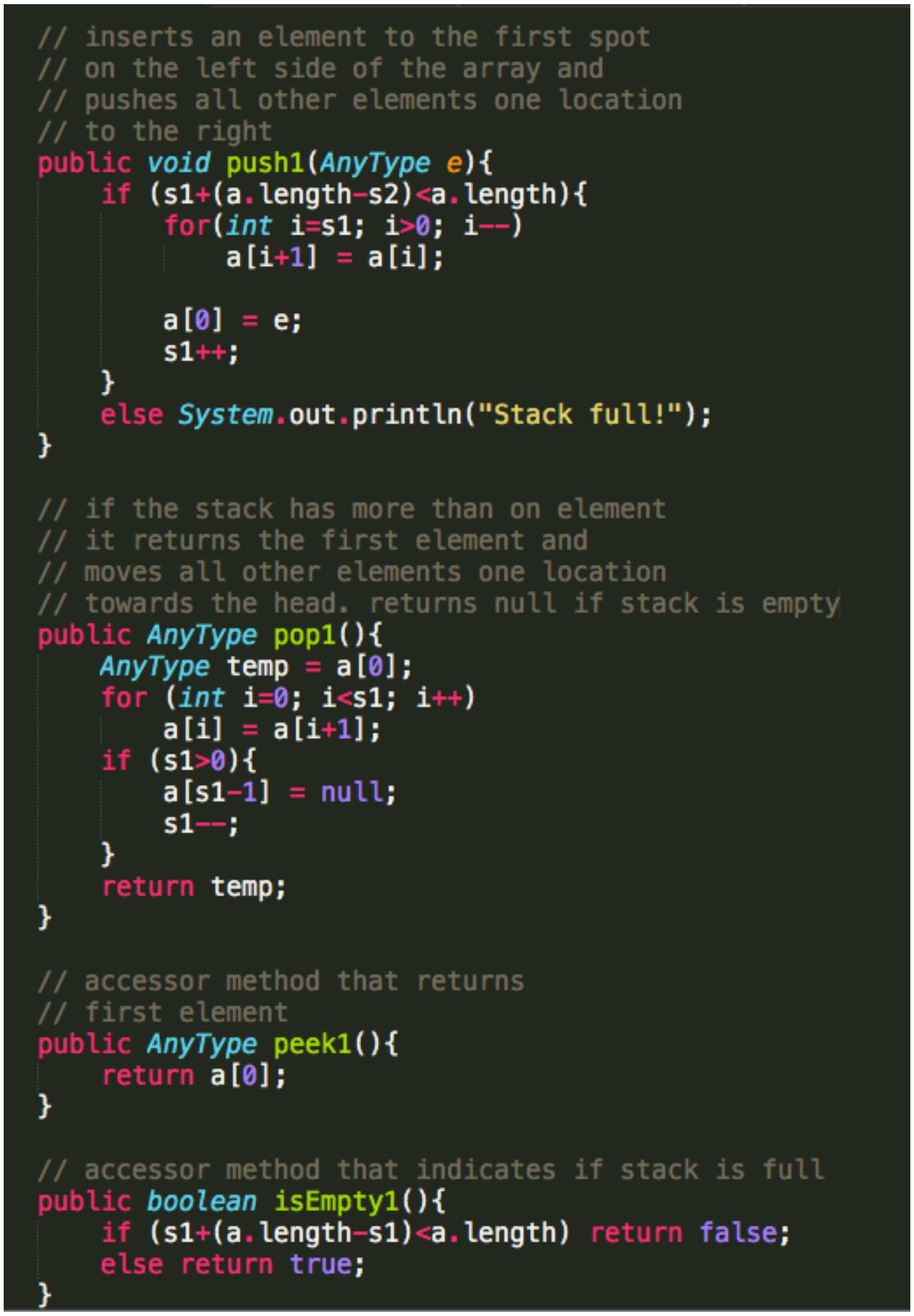
3. (screenshot) – class three uses an array to accommodate two stacks. push1, pop1, peek1, and isEmpty1 use the left side of the array (starting at location 0). push2, pop2, peek2, and isEmpty2 use the right side of the array (starting at location n-1).

    a. top part – constructor and class name. assigns null to all elements and uses s1 and s2 as indicators of elements in each stack.



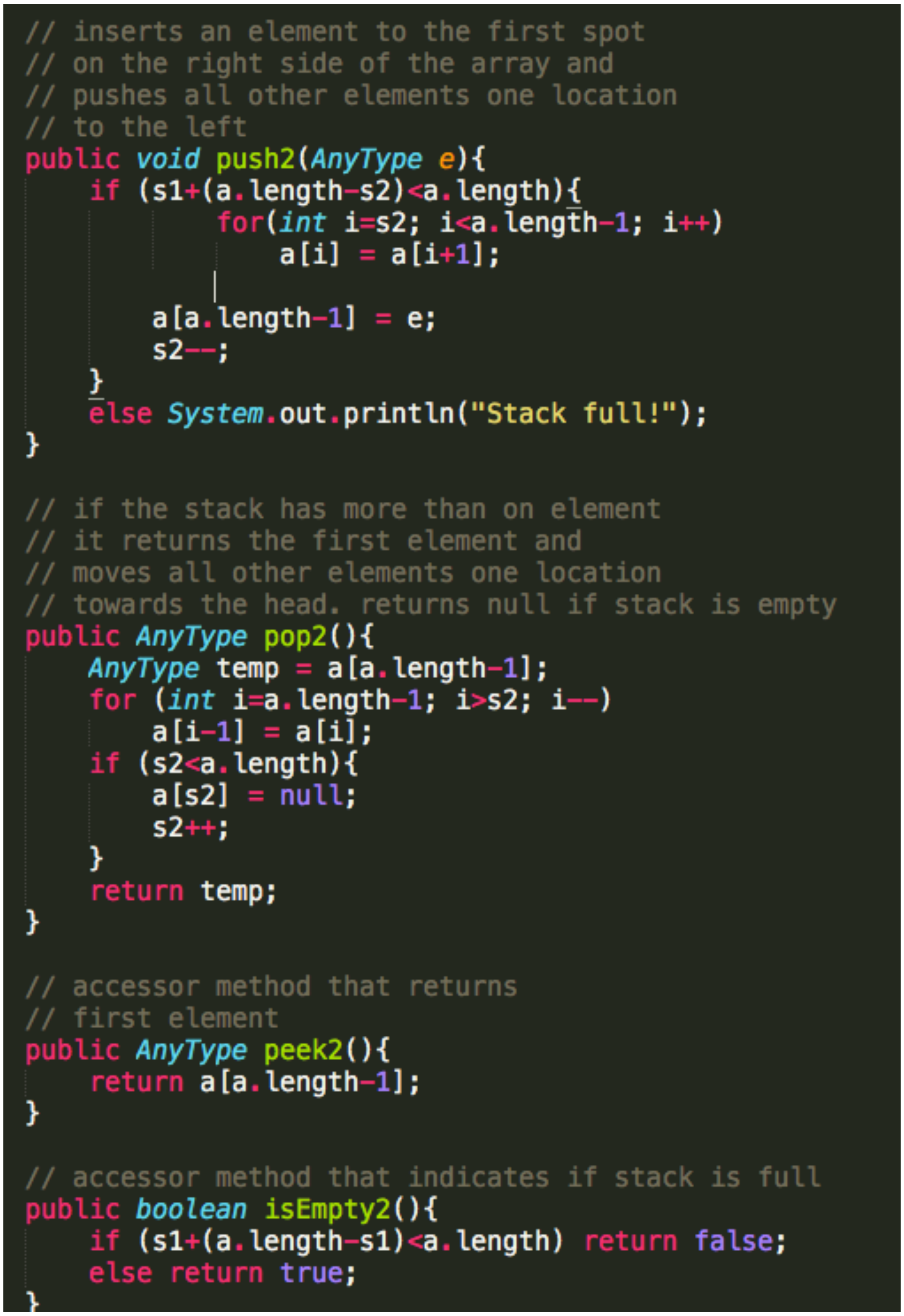
```java
public class three<AnyType>{
    AnyType[] a;    // array
    int s1;         // counter of elements in stack1
    int s2;         // counter of elements in stack2

    // constructor creats an array and
    // gives all locations null as value
    public three (int n){

        a =  (AnyType[]) new Object[n];
        for (int i=0; i<n; i++)
            a[i] = null;
        s1 = 0;
        s2 = n;
    }
```

b. first stack – push adds an element to the first location on the left side of the array and moves all elements one spot towards the middle. Pop removes first element and moves all elements one location towards the head.



```java
// inserts an element to the first spot
// on the left side of the array and
// pushes all other elements one location
// to the right
public void push1(AnyType e){
    if (s1+(a.length-s2)<a.length){
        for(int i=s1; i>0; i--)
            a[i+1] = a[i];

        a[0] = e;
        s1++;
    }
    else System.out.println("Stack full!");
}

// if the stack has more than on element
// it returns the first element and
// moves all other elements one location
// towards the head. returns null if stack is empty
public AnyType pop1(){
    AnyType temp = a[0];
    for (int i=0; i<s1; i++)
        a[i] = a[i+1];
    if (s1>0){
        a[s1-1] = null;
        s1--;
    }
    return temp;
}

// accessor method that returns
// first element
public AnyType peek1(){
    return a[0];
}

// accessor method that indicates if stack is full
public boolean isEmpty1(){
    if (s1+(a.length-s1)<a.length) return false;
    else return true;
}
```

c. second stack – same as stack1, but on the right side. Push to last location, pop from last location



```
// inserts an element to the first spot
// on the right side of the array and
// pushes all other elements one location
// to the left
public void push2(AnyType e){
    if (s1+(a.length-s2)<a.length){
            for(int i=s2; i<a.length-1; i++)
                a[i] = a[i+1];

        a[a.length-1] = e;
        s2--;
    }
    else System.out.println("Stack full!");
}

// if the stack has more than on element
// it returns the first element and
// moves all other elements one location
// towards the head. returns null if stack is empty
public AnyType pop2(){
    AnyType temp = a[a.length-1];
    for (int i=a.length-1; i>s2; i--)
        a[i-1] = a[i];
    if (s2<a.length){
        a[s2] = null;
        s2++;
    }
    return temp;
}

// accessor method that returns
// first element
public AnyType peek2(){
    return a[a.length-1];
}

// accessor method that indicates if stack is full
public boolean isEmpty2(){
    if (s1+(a.length-s1)<a.length) return false;
    else return true;
}
```

4.

a. input track [5, 9, 6, 7, 2, 8, 1, 3, 4]

    i. Move the car at the front of the input track to the top of S_1. S_1 = [4]
    ii. Move the car at the front of the input track to the top of S_1. S_1 = [3, 4]
    iii. Move the car at the front of the input track to the back of the output track. output = [1]
    iv. Move the car at the front of the input track to the top of S_2. S_2 = [8]
    v. Move the car at the front of the input track to the back of the output track. output = [2,1]
    vi. Move the car at the top of S_1 to the back of the output track. Output = [3, 2, 1]. S_1 = [4]
    vii. Move the car at the top of S_1 to the back of the output track. Output = [4, 3, 2, 1]. S_1 = []
    viii. Move the car at the front of the input track to the top of S_2. S_2 = [7, 8]
    ix. Move the car at the front of the input track to the top of S_2.
    S_2 = [6, 7, 8]
    x. Move the car at the front of the input track to the top of S_1. S_1 = [9]
    xi. Move the car at the front of the input track to the back of the output track. output = [5, 4, 3, 2, 1]
    xii. Move the car at the top of S_2 to the back of the output track. Output = [6, 5, 4, 3, 2, 1]. S_2 = [7, 8]
    xiii. Move the car at the top of S_2 to the back of the output track. Output = [7, 6, 5, 4, 3, 2, 1]. S_2 = [8]
    xiv. Move the car at the top of S_2 to the back of the output track. Output = [8, 7, 6, 5, 4, 3, 2, 1]. S_2 = []
    xv. Move the car at the top of S_1 to the back of the output track. Output = [9, 8, 7, 6, 5, 4, 3, 2, 1]. S_1 = []

Output track [9, 8, 7, 6, 5, 4, 3, 2, 1]

b. One example: this will not work with the arrangement [1, 9, 8, 7, 6, 5, 4, 3, 2]. In this case we will have to return some of the cars to the input track in order to achieve the arrangement of cars, but returning a car to the input track is not one of the operations and therefore is not possible. Any combination where 1 is in the back and 2,3,4 are in the front are examples of combinations that don’t work.