Do not begin the exam until you are told to do so.

You have 50 minutes to complete the exam.

There are 100 total points.

There are 9 pages in this exam.

Make sure your name and Pennkey (a.k.a. username) is on the top of this page.

Be sure to allow enough time for all the problems—skim the entire exam first to get a sense of what there is to do.
1. Java vs. OCaml (22 points)

   a. In OCaml, the proper way to check whether two string values \( s \) and \( t \) are structurally equal is:
      - \( s == t \)
      - \( s = t \)
      - \( s \text{.equals}(t) \)
      - \( s \text{ := } t \)

   b. In Java, the proper way to check whether non-null String objects \( s \) and \( t \) are structurally equal is:
      - \( s == t \)
      - \( s = t \)
      - \( s \text{.equals}(t) \)
      - \( \text{String.equals}(s, t) \)

   c. Every Java type is a(n) __________ of class Object.
      - supertype
      - subtype
      - instance
      - extension

   d. In Java, object values are stored in the __________ of the Abstract Stack Machine.
      - stack
      - workspace
      - heap
      - class table

   e. If you were to port the OCaml GUI project (HW06) to Java, it would be natural to make \texttt{Gctx} (graphics contexts) a class that is a subtype of \texttt{Widget}.
      - true
      - false

   f. In simple inheritance, the subclass adds new fields or methods without overriding any of the parent class’s members.
      - true
      - false
g. Invariants (like the ones used in queue programming HW05 or in the resizable array example from lecture) are properties of a datastructure or relationships among values that hold both before and after a method/function runs.

- true
- false

h. Encapsulation of state to preserve invariants can be enforced in OCaml using:

- first-class option types
- local let declarations or module interfaces
- recursion and lists
- mutable record fields

i. Encapsulation of state to preserve invariants can be enforced in Java using:

- private fields and interfaces
- static methods and null
- loops and arrays
- mutable fields

j. In Java, a static method dispatch C.m() implicitly pushes the this reference onto the stack.

- true
- false

k. Which OCaml construct is closest to a Java object?

- a record of closures
- a record of mutable option fields
- an anonymous function
- a module with just one type

l. Which Java construct is closest to an OCaml anonymous function?

- an interface with just one method called apply
- an object with just one method called apply
- a class with just one method called apply
- a static method called apply
2. Abstract Stack Machine (20 points)

Consider the following OCaml program that uses the queue types seen in Lecture and HW05:

(* Mutable queues, as defined in class. *)

type 'a queuenode = { v: 'a;
                  mutable next: 'a queuenode option}

    type 'a queue = { mutable head : 'a queuenode option;
                     mutable tail : 'a queuenode option }

    let qn1 : int queuenode = {v = 1; next = None;}
    let qn2 : int queuenode = {v = 2; next = None;}
    let qn3 : int queuenode = qn1
    ;; qn2.next <- Some qn2
    ;; qn3.next <- Some qn2
    let qn4 : int queuenode = {v = 4; next = qn1.next}
    (* HERE *)

Complete the diagram below of the state of the stack and heap parts of the ASM when the program reaches the point marked (* HERE *) in the program above. Note:

- you might need to allocate new heap objects,
- you may need to add “Some bubbles” in the appropriate places,
- if you are simulating the execution of the program, you might have to erase pointers at times (or, if using ink, mark the erased pointers clearly with an X)
- should show only the final state!
- the Appendix of the exam contains an example of the stack and heap diagram for a similar OCaml program.

![Diagram](image-url)
3. Subtyping, Interfaces, and Inheritance (16 points)

Consider the following Java interface and class definitions:

```java
interface X {
    int getX();
}

interface Y extends X {
    int getY();
}

class C implements X {
    public int getX() {
        return 1;
    }
    public int getY() {
        return getX() + getX();
    }
}

class D extends C implements X, Y {
    public int getX() {
        return 2;
    }
}

class E extends C {
    public int getY() {
        return 3;
    }
}
```

For each code snippet below, write the integer value that will be printed out, or write “ill typed” if the compiler would flag a type error (i.e. Eclipse would underline something in red).

- `X x1 = new C();
  System.out.println(x1.getX()); ____________________`

- `X x2 = new D();
  System.out.println(x2.getY()); ____________________`

- `X x3 = new E();
  System.out.println(x3.getX()); ____________________`

- `Y y1 = new C();
  System.out.println(y1.getY()); ____________________`

- `C c1 = new D();
  System.out.println(c1.getX()); ____________________`

- `C c2 = new C();
  System.out.println(c2.getY()); ____________________`

- `D d1 = new D();
  System.out.println(d1.getY()); ____________________`

- `D d2 = new E();
  System.out.println(d2.getY()); ____________________`
4. **Java Programming** (20 points total)

The following Java class `ATree` implements a tree datastructure in which each node has an integer value `v` and an arbitrary (but finite) number of children stored in an array.

```java
class ATree {
    int v;
    ATree[] children;

    public ATree(int v, ATree[] c) {
        this.v = v;
        this.children = c;
    }
}
```

Consider the problem of writing a method called `sum` such that `a.sum()` returns the result of adding up all of the values in the tree `a`.

**Step 1:** The first step of the program design process is to **understand the problem**. There is nothing for you to write here, but we need to pay careful attention to the use of `null` in this datatype.

- If `a.children == null` then `a` is a leaf node of the tree.
- If `a` is not `null`, then `a.sum()` should **never** throw a `NullPointerException`.

**Step 2:** The second step is to **define the interface** of the method. For this problem the interface is particularly simple, so we do not ask you do it:

```java
public int sum() { ... }
```

**Step 3:** The next step is to **write test cases**. We have provided the two test cases shown below. Make sure that you understand them!

```java
@Test
public void testLeaf() {
    ATree a = new ATree(1, null); // leaf
    assertTrue(a.sum() == 1);
}

@Test
public void testChild1() {
    ATree a1 = new ATree(1, null); // leaf
    ATree a2 = new ATree(2, null); // leaf
    ATree[] children = { a1, a2 };
    ATree a = new ATree(4, children); // non-empty tree
    assertTrue(a.sum() == 7); // (1 + 2) + 4 == 7
}
```
a. (6 points) Now consider the following similar test code:

```java
@Test
public void testChild2() {
    ATree a1 = new ATree(1, null);
    ATree a2 = __________________; // Fill in here!
    ATree[] children = { a1, a2 };  
    ATree a = new ATree(2, children); 
    assertTrue(a.sum() == 3); // note expected value is 3
}
```

Give two different Java expressions (that evaluate to distinct values) that can be placed in the blank above to create a well-typed program such that the test succeeds.

Answer 1: ____________________________

Answer 2: ____________________________

b. (14 points) Complete the implementation of the `sum` method. For your convenience, we repeat the other code for `ATree` here:

```java
class ATree {
    int v;
    ATree[] children;

    public ATree(int v, ATree[] c) {
        this.v = v;
        this.children = c;
    }

    /∗ Sums the values of all the v’s in the tree ∗/
    public int sum() {
```
Implement in Java a static method called `canBalance`, that, given a non-null and non-empty array, returns `true` if there is a place to split the array so that the sum of the numbers on the left side is equal to the sum of the numbers on the right side (and `false` otherwise).

For example (using a shorthand notation for integer arrays):

- `canBalance({1, 1, 1, 2, 1})` ⇒ `true` because `(1+1+1) == (2+1)`
- `canBalance({2, 1, 1, 2, 1})` ⇒ `false`
- `canBalance({10, 10})` ⇒ `true` because `10 == 10`
- `canBalance({5})` ⇒ `false`
- `canBalance({10, 0, 1, -1, 10})` ⇒ `true` because `10 == (0 + 1 + -1 + 10)`
- `canBalance({1, 1, 1, 3})` ⇒ `true` because `(1+1+1) == 3`

To get you started, we have given you the skeleton of the algorithm:

```java
public static boolean canBalance(int[] nums) {
    if (nums.length < ___________) {
        return false;
    }
    int leftSum = 0;
    int rightSum = 0;
    for (int i = _____________; i < _____________; i++) {
        leftSum = leftSum + _______________; // accumulate left sum
        rightSum = _______________;
    }
    for (int j = _____________; j < ______________; j++) {
        rightSum = rightSum + _______________; // accumulate right sum
    }
    if (________________________________________) {
        return ______________;
    }
    return ______________;
}
```
Appendix

This appendix shows an example of the Stack and Heap components of the OCaml Abstract Stack Machine. Your diagram for Problem 1 should use similar “graphical notation” for Some v and None values.

(* The types of mutable queues. *)

```ocaml
type 'a queuenode = { v : 'a; mutable next : 'a queuenode option}

type 'a queue = { mutable head : 'a queuenode option; mutable tail : 'a queuenode option }

let qn1 : int queuenode = {v = 1; next = None;}
let qn2 : int queuenode = {v = 2; next = Some qn1;}
let q : int queue = {head = Some qn2; tail = Some qn1;}

(* HERE *)
```

The OCaml program above yields the ASM Stack and Heap depicted below when the program execution reaches the point marked (* HERE *).

![Diagram of OCaml program execution](image-url)