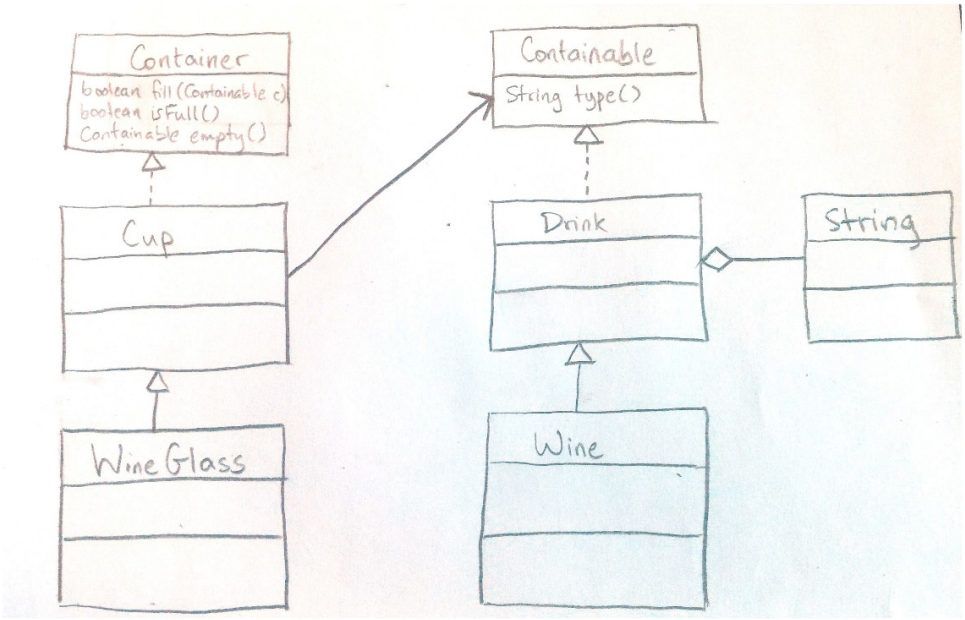
CSE 115 Review Packet

The code given below is correct; it compiles without errors.
Use it as a reference for questions 1-5 in this review packet.

```
  1
public interface Container {
      public boolean fill(Containable c);
      public boolean isFull();
  2   public Containable empty();
}

public interface Containable {
      public String type();
}

public class Drink implements Containable {
  3   private String _type;

      public Drink() {
            _type = new String("drink");
      }

      @Override public String type() { return _type; }
}

public class Wine extends Drink {
      public Wine() {
            super();
      }

      @Override public String type() {
            return super.type() + ": wine";8
      }
}
```

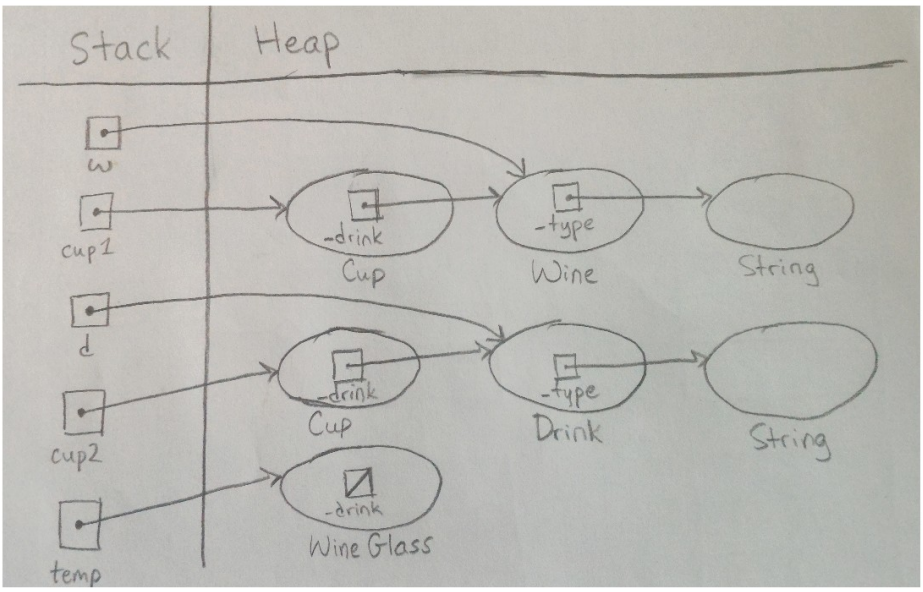
```
public class Cup implements Container {
      private Containable _drink;

      public Cup(Containable c) {
            if (!fill(c)) { // _drink = c, if c is valid
                  _drink = null; //_drink = null, if c is invalid
            }
      }

      @Override public boolean fill(Containable c) {
            if (isFull() || c == null || !c.type().startsWith("drink")) {
               5  return false;
            }
            _drink = c;
            return true;
      }
                                                              6
      @Override public boolean isFull() { return _drink != null; }

      @Override public Containable empty() {
            Containable result = _drink;
            _drink = null;
            return result;
      }
}

public class WineGlass extends Cup {
      public WineGlass(Containable c) {
            super(c);          7
      }

      @Override public boolean fill(Containable c) {
            if (!c.type().contains("wine")) {
        10        return false;
            }
            return super.fill(c);
      }
}
```

1. Draw a UML diagram that shows the relationships between the classes and interfaces in the reference code. Make sure to consider all the types of relationships you've learned, including realization, inheritance, association, and composition. You need only to consider the classes and interfaces explicitly used (e.g. you should use String, but not Object).



2. Draw an object diagram to show the *resulting* program state.

```
Drink d = new Drink();
Cup cup1 = new Cup(d);

Wine w = new Wine();
Cup cup2 = new Cup(w);

WineGlass temp = new WineGlass(null); //temp is empty
temp.fill(cup2.empty());
cup2.fill(cup1.empty());
cup1.fill(temp.empty());
```



3. Write a class called BreadBox that is a Container. Using the Cup code as a reference, BreadBox objects should only be allowed to contain Bread. The code for the Bread class is provided below.

```
public class Bread implements Containable {
	private String _type;

	public Bread() { _type = new String("bread"); }

	@Override public String type() { return _type; }
}
```

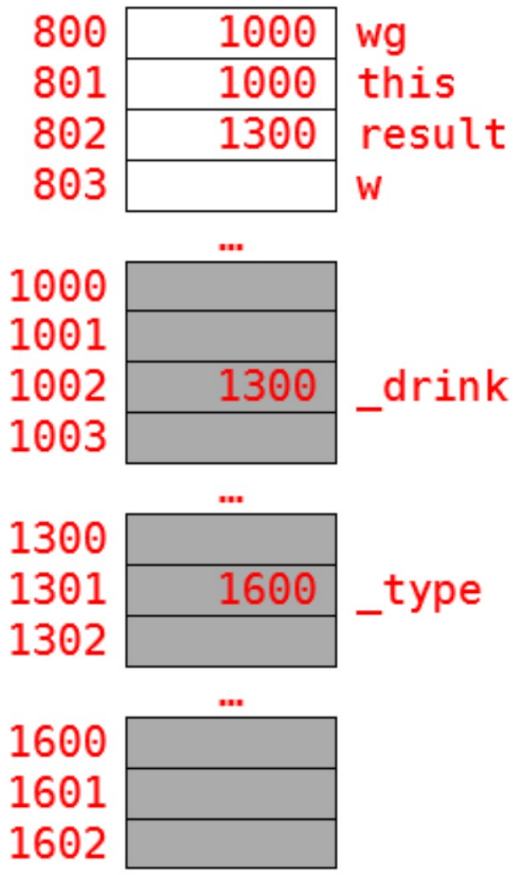
```
public class BreadBox implements Container {
	private Containable _bread;

	public BreadBox(Containable c) {
		if (!fill(c)) {
			_bread = null;
		}
	}

	@Override public boolean fill(Containable c) {
		if (isFull() || c == null || !c.type().startsWith("bread")) {
			return false;
		}
		_bread = c;
		return true;
	}

	@Override public boolean isFull() { return _bread != null; }

	@Override public Containable empty() {
		Containable result = _bread;
		_bread = null;
		return result;
	}
}
```

4. Given that the following code has been run:

```
WineGlass wg = new WineGlass(new Wine());
```

draw a memory diagram showing a possible snapshot of memory during the invocation of the following method:

```
Wine w = wg.empty();
```



5. Circle, and identify by number, *one* ***and only one*** example of each of the following items in the reference code. If you believe no example exists, write “no example” next to that item in the list.

1. access control modifier
2. method header
3. instance variable declaration
4. local variable assignment
5. method call
6. boolean expression
7. parameter declaration
8. String literal
9. type variable `no example`
10. conditional statement

6. The following method is correct except for *one* ***and only one*** line. Indicate which line is incorrect and write a replacement for it.

```
/**
 * Returns the sum of the squares of the all the numbers from 1 to n.
 * If n is less than 1, returns 0.
 * Examples:    n = -1 -> 0
 *              n = 0  -> 0
 *              n = 3  -> 14 (1^2 + 2^2 + 3^2 = 14)
 */
1   public int sumOfSquares(int n) {
2      int result = 0;
3      while (n > 0) {
4         result = n ^ 2;
5         n = n - 1;
6      }
7      return result;
8   }
```

`line 4:  result = result + n * n;`

7. Write a method which takes two Points as parameters (java.awt.Point) and returns a boolean value. The method should return true only if the points are adjacent.

   For example, if the method is named adjacent and is defined in a class named Question7, then

```
new Question7().adjacent(null, new Point(0,0));
```

   must not produce any runtime errors and must return false, and

```
new Question7().adjacent(new Point(0,6), new Point(1,5));
```

   must not produce any runtime errors and must return false, whereas

```
new Question7().adjacent(new Point(1,3), new Point(2,3));
```

   must not produce any runtime errors and must return true.

```
public boolean adjacent(Point p, Point q) {
     if (p == null || q == null) {
          return false;
     }

     int xDiff = p.x - q.x;
     int yDiff = p.y - q.y;

     // take the absolute value
     if (xDiff < 0) {
          xDiff = -xDiff;
     }

     if (yDiff < 0) {
          yDiff = -yDiff;
     }

     return (xDiff + yDiff == 1);
}

//another possible solution

public boolean adjacent(Point p, Point q) {
     return p != null && q != null &&
            Math.abs(p.x - q.x) + Math.abs(p.y - q.y) == 1;
}
```

8. Study the following code:

```
public void question8(char c, int n) {
   for (int i = 0; i < n; i = i + 1) {
      for (int j = 0; j <= i && j < n - i; j = j + 1) {
           System.out.print(c);
      }
      System.out.println();
   }
}
```

Show what is printed by the following method call:

```
question8('#', 4)
```

```
#
##
##
#
```

9. For this question, use an **8-bit wide two's complement** representation for integers.
   a. Convert 23_{10} into two's complement.
   b. Convert 00101010 into decimal, interpreting the bit string as a two's complement number.
   c. Compute $00001110_2 + 00111101_2$. Show your work.
   d. Compute the two's complement of 00000110_2 and write down the result. Also express the result in base 10. Show your work.

a. $23_{10} = 0*64 + 0*32 + 1*16 + 0*8 + 1*4 + 1*2 + 1*1$
$= 00010111_2$

b. $00101010_2 = 0*64 + 1*32 + 0*16 + 1*8 + 0*4 + 1*2 + 0*1$
$= 42_{10}$

c.
```
  1 1 1 1
  00001110
+ 00111101
  01001011
```

d.
```
  00000110
  11111001 one's complement
+ 00000001 add 1
  11111010 two's complement
```

$00000110_2 = 4 + 2 = 6_{10}$, so
$11111001_2 = -6_{10}$.

10. For each vocabulary term, write down the letter of its definition in the box. Note that there are more definitions than terms.

o	this
b	invocation record
a	reference
n	stack
d	lifetime
i	class
g	variable
k	heap
f	method
c	scope

a. The address at which an object is stored in memory
b. An encapsulation of the parameters and local variables used in a method call
c. The part of a program where a variable declaration is in effect
d. The period of time during execution of a program that the variable exists in memory
e. The representation scheme for floating point numbers
f. A behavior than an object is able to execute
g. Holds a value - either a primitive value or a reference to an object
h. A class used to wrap a primitive value into an object
i. A description of the properties and behaviors that objects of this type will have
j. A special method used to create an object by instantiating the class
k. The part of memory where objects, their properties, and their behaviors are stored
l. A data structure that stores an arbitrary number of references to objects
m. The part of memory where static variables and static methods are stored
n. The part of memory where invocation records are stored
o. An implicit variable in a method which holds a reference to the object on which the method was called