Question 1. [34 MARKS]

Consider the following version of the class `Rational` from lecture. The writer noticed, just as we did, that updating `n` requires the original value of `d`. Unfortunately there is still an error, when an object is added to itself.

```
class Rational implements Addable {

  private int n;
  private int d;

  public Rational(int n, int d) {
    this.n = n;
    this.d = d;
  }

  // Requires r to be Rational
  public void add(Addable r) {
    Rational r1 = (Rational) r;
    // Remember original d for when we update n
    int d = this.d;
    this.d = d * r1.d;
    n = n * r1.d + d * r1.n;
  }

  public boolean equals(Object o) {
    return o instanceof Rational
      && n * ((Rational) o).d == d * ((Rational) o).n;
  }

  public String toString() {
    return n + ``/'' + d;
  }

}
```

Part (a) [8 MARKS]

Show the state of memory when the following code is executed, just before `add` returns (in other words, after `add`'s last line is executed but it and its local variables still exist).

```
class C {
  public static void main(String[] args) {
    Rational a = new Rational(2, 3);
    Rational b = a;
    a.add(b);
  }
}
```

Solution:

TO COME (after I learn to draw :)#).

Part (b) [2 MARKS]

What would then be returned by `a.toString()`?

Solution: "24/9"

Part (c) [8 MARKS]

The writer of `Rational` finds out that many users need a rational equal to 1 and decides to save them memory by providing a method called `one`. This method keeps returning the same object when called, except it starts using a new one each time the current one is no longer 1 (because someone called `add` on it). Even though this method is not actually a good idea, write it.

Solution:

We're letting the user call `one` instead of `new Rational(1, 1)`, so that we don't automatically make a new object each time. We make the method static since the user shouldn't need some arbitrary `Rational` just to get 1.

```
private static Rational one;

public static Rational one() {
  if (one == null || one.n != one.d) {
    one = new Rational(1, 1);
  }
  return one;
}
```

Initializing `one` instead of testing for `null` would also have been okay since one `Rational` is miniscule compared to everything else.

Part (d) [6 MARKS]

Suppose now that `Rational` gets fixed and works properly.

Recall the following code from lecture:

```
class Add {
  // Add all the elements of l to its first element
  //  and return that element.
  // Requires l to be non-empty and contain only Addables.
  public static Addable addUp(LinkedList l) {
    Addable sum = (Addable)(l.get(0));
    Iterator i = l.iterator();
    i.next(); // skip first element since already accounted for
    while (i.hasNext()) {
      sum.add((Addable)(i.next()));
    }
    return sum;
  }
}
```

Unfortunately, `addUp` also has an error, similar to the error in (a). Explain the error and write a single JUnit test case that fails because of it.

Solution:

The problem in (a) was updating something that could be needed again, inadvertently if one object was referred to in two different ways. Here the problem can occur if the same object occurs twice in the list. If it's the first element then during the summation when it is encountered again it no longer has its original value.

```
public class StringTest extends junit.framework.TestCase {

  public void testAddUp() {

    LinkedList l = new LinkedList();
    Rational r = new Rational(1, 1);
    l.add(r);
    l.add(new Rational(1, 1));
    l.add(r);
    assertTrue(Add.addUp(l).equals(new Rational(3, 1)));

  }
}
```

The test will fail because 1 is added to `r`, which is then 2, and then `r` is added to itself to get 4. We could have used `r` for the second element of the list, but the above tests the minimum needed to cause a problem.

Part (e) [5 MARKS]

Many kinds of addable objects have a "zero": an object that doesn't change objects it's added to. Show the modifications to `Addable` and `Rational` to allow users access to a zero object. Note: you will be using this in (f). And don't worry about consistency with your answer to (c); in particular always give users a new zero.

Solution:

If we make a static method for zero then users need to know what type of `Addable` they have in order to call the method. But in (f) we won't know the type. So we want to be able to ask an `Addable` for a corresponding zero, via an instance method.

```
interface Addable {

  void add(Addable a);

  Addable zero();

}
```

In `Rational`:

```
  public Addable zero() {
    return new Rational(0, 1);
  }
```

Part (f) [5 MARKS]

Assuming the modifications from part (e), write a version of `addUp` that doesn't modify the first element, and that takes an `Iterator` instead of a `LinkedList` (and that works correctly).

Solution:

```
class Add {
  // Return the sum of the elements from i.
  // Requires i has at least one element and returns only Addables of the same type.
  public static Addable addUp(Iterator i) {
    Addable first = (Addable)(i.next());
    Addable sum = first.zero();
    sum.add(first);
    while (i.hasNext()) {
      sum.add((Addable)(i.next()));
    }
    return sum;
  }
}
```

Question 2. [8 MARKS]

Consider the following classes:

```
class C {
  public String m() {
    return "Hi";
  }
}
class D extends C {
  public String m() {
    return "Bye";
  }
}
```

Beside each of the following statements, executed in the Interactions Pane of DrJava, write either "Compile error", "Exception" or the return value, as appropriate.

Solution:

```
C c = new C();

c.m() // ``Hi''

((D)c).m() // Exception: ((D)c) claimed that at run time c would refer to a D here.

c = new D(); // Okay, like referring to Product or Consulting as Billable.

D d = new D();

d.m() // ``Bye'': standard overriding

((C)d).m() // ``Bye'': no access to super directly from outside class
           // Similar to: c.m() after c = new D()

d = new C(); // Compile error: declared type of right not D-like.
```

Total Marks = 42