# Assignment 2 (Solution)

## Exercise 1

1. Q: Why are method `equals` and method `hashcode` *probably* pure? Under which circumstances are they not pure?

   A: They are *probably* pure because we don't know the implementation `Image.hashcode` which might violate the purity specification. The same would be true for `String.equals` and `String.hashcode` if we didn't know their implementations, but `String` is a builtin Java class whose `equals` and `hashcode` methods actually happen to be pure, and the `String` class is final, meaning that the standard implementations cannot be overridden.

   Q: Is is possible to change the class design such that they are pure under all circumstances?

   A: One possibility would be to provide a pure implementation of method `Image.hashcode`. Note that if we extract it into a separate method we then need to make it final (otherwise subclasses might override it with a non-pure version).

2. One could instrument the following program operations:

   - *object allocations* to keep track of newly allocated objects that can be modified inside pure methods
   - *field writes* to check whether the method is allowed to modify given field
   - *method entry/exit* to update global instrumentation state depending on whether the method is pure or not.

   The following shows a sketch of non-thread safe instrumentation:

```
class PurityChecks
{
  // Contains objects that pure method is allowed to modify
  static IdentityHashSet<object> fresh = new IdentityHashSet<object>();
```

```
 // Denotes whether we are restricting method
 // to modify only freshly created objects.
 // Set to true for all methods annotated as @Pure
 // and all methods called transitively from @Pure methods
 static boolean checking = false;
}
class ImageFile
{
 String file;
 Image image;

 public Image getImage() {
     if (this.image == null) {
    Image tmp = new Image();
    // whenever an object is created we add
    // it to the set of object that can be modified
    + PurityChecks.fresh.insert(tmp);
    ...  load image

    // whenever we modify an object
    // we check if this is allowed
    + assert !PurityChecks.checking || PurityChecks.fresh.contains(this);
    this.image = tmp;
  }

  return image;
 }

 @Pure
 boolean equals(Object o) {
  // Since this method has @Pure annotation we:
  // 1. Save the state of the caller
  + boolean checking = PurityChecks.checking;
  + IdentityHashSet<object> fresh = PurityChecks.fresh;
  // 2. Initialize the PurityChecks
  // fresh set is initially empty as we are allowed to
  // modify only newly created objects
  + PurityChecks.checking = true;
  + PurityChecks.fresh = new IdentityHashSet<object>();

  if( o.getClass() != getClass() ) return false;
  return file.equals( ((ImageFile) o).file );

  // add newly allocated objects to the global set to allow
  // their modification after we return from this method
  + fresh.addAll(PurityCheck.fresh);
  // Restore the state of caller,
  // should be executed before each return statement
  + PurityChecks.checking = checking;
```

```
    + PurityChecks.fresh = fresh;
  }

  @Pure
  int hashcode() {
    + var checking = PurityChecks.checking;
    + var fresh = PurityChecks.fresh;
    + PurityChecks.checking = true;
    + PurityChecks.fresh = new IdentityHashSet<object>();

    if (image == null) {
      return file.hashcode();
    } else {
      return image.hashcode() + file.hashcode();
    }

    + fresh.addAll(PurityCheck.fresh);
    + PurityChecks.checking = checking;
    + PurityChecks.fresh = fresh;
  }
}
```

3. Methods which lazily initialize fields, or which write computed data to a cache, could not be marked as pure; otherwise, the instrumentation would report errors, since the methods modify the state of an object, even though the effects can't be observed by a client. We could weaken the definition of purity to allow such designs.

## Exercise 2

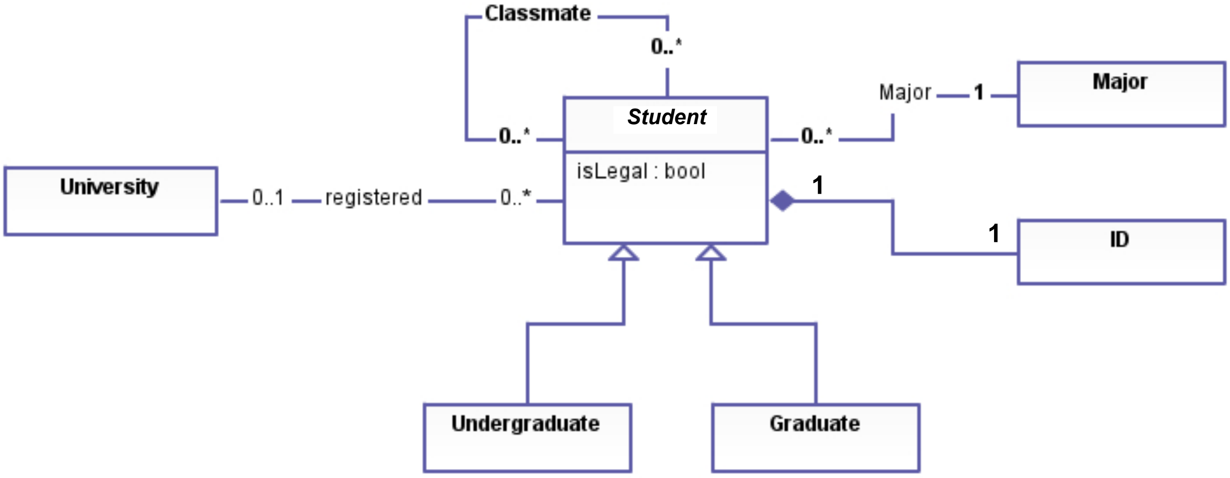
Main insight: `elems.Length` must be strictly greater than 0 so that `Add` works, therefore we get the preconditions `initialElements.length > 0` and `howMany > 0` in the constructors.

```
public class Bag {
    private int[] elems;
    private int count;

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(elems != null);
        Contract.Invariant(0 < elems.Length);
        Contract.Invariant(0 <= count && count <= elems.Length);
    }
    public Bag(int[] initialElements) {
        Contract.Requires(initialElements != null);
        Contract.Requires(0 < initialElements.Length);
        ...
    }
    public Bag(int[] initialElements, int start, int howMany) {
        Contract.Requires(0 <= start);
        Contract.Requires(0 < howMany);
        Contract.Requires(initialElements != null);
        Contract.Requires(start + howMany <= initialElements.Length);
        ....
    }
    [Pure]
    public int Count() {
        ....
    }
    public int RemoveMin() {
        Contract.Requires(0 < Count());
        Contract.Ensures(count == Contract.OldValue(Count()) - 1);
        Contract.Ensures(Contract.Result<int>() ==
                         Contract.OldValue(elems.Take(Count()).Min()));
        ....
    }
    public void Add(int x) {
        Contract.Ensures(Count() == Contract.OldValue(Count()) + 1);
        Contract.Ensures(elems[Contract.OldValue(Count())] == x)
        Contract.Ensures(!Contract.OldValue(Count() == elems.Length) ||
          elems.Length == 2*Contract.OldValue(elems.Length))
        Contract.Ensures(Contract.ForAll(0, Count() - 1, i =>
          elems[i] == Contract.OldValue(elems[i])))
        ....
    }
}
```

## Exercise 3

1. Student Class diagram



2. (a) Classmates have the same major
   (b) A student is legal iff he/she is registered