4a) (i) Primitive types in Java contain only the binary representation for their value (with possible exceptions dependent on implementation). Objects in Java may contain primitive types, or references to other objects, and methods.

(ii) Primitive types are stored on the stack, objects are stored on the heap, but their references are stored on the stack.

(iii) Primitive types don't interact with references. Objects are referred to using references when they are stored in a variable or passed to a function. A reference points to the address where an object is stored on the heap.

b) Auto-boxing and unboxing are performed by the java compiler to convert between the primitive types and their respective wrapper classes. Auto-boxing is conversion to a wrapper class, and auto-unboxing is conversion to a primitive type. They occur when variables are assigned or passed to functions with arguments of the opposite type.

A null pointer exception may be thrown if a wrapper class variable is a `null` reference, since there is no content to be converted into a primitive variable. For example:

```
void example(int i) {
    return;
}
Integer i = null;
example(i);
```

c) Java uses pass-by-value for functions, meaning a copy of the value is passed to the function, so `t1` will be unchanged for any primitive type, since they are copied. Immutable objects such as the wrapper classes for primitives will also be unchanged (if they have no self-altering methods). Other objects may be changed, since only the reference to the object is copied, but it points to the same memory address where the object is.

d) Java generics are implemented using type erasure, so all instances of `T` will be replaced by the native Java `Object`. Since primitive types are not subclasses of `Object`, they cannot be used as `Object`s in the same ways.

e) (i) This makes sense since the element-accessing methods involved in an immutable list are simply expanded with the modifying methods for mutability. It doesn't make much sense on a conceptual level though.

(ii) This doesn't make much sense, since the modifying methods required for a `MutableList` are not allowed in an `ImmutableList`. On top of that, this hierarchy makes no sense in concept either. Not good.

(iii) This is a mediocre solution, since it will require rewriting of the accessing methods for each class, but it isn't as bad as (ii), and at least can be argued to be true in concept;

(iv) This would ultimately be a waste of a class, since the methods common to both `MutableList` and `ImmutableList` are just all the methods from the latter, so `CommonList` wouldn't introduce any

new insights. However, this makes the most sense in concept, because it mirrors the way we think about both as lists.