Recall: Designing and implementing a new data structure

The interface describes what a data structure can do (e.g., its operations). The interface is a promise from the provider of the data structure to the user of the data structure.

The implementation describes how the data structure works. It makes good on the promise of the interface. The implementation includes the data structure’s encoding, which describes how information in the structure is stored using existing, simpler data structures. The implementation also includes the algorithms that operate over the encoding, to provide the promised operations.

It should be possible for the provider of the data structure to replace the implementation, without modifying the interface. That way, the client of the data structure can continue using the data structure (and its presumably improved implementation), without changing their code.

Space vs Time

When designing / implementing a new data structure, we can often “trade space for time”. That is to say: when an operation costs too much, we can expand the encoding so that the data structure stores more information, which the structure can then look up instead of compute. Although storing information might save time for a particular operation, there are other costs: higher storage overhead, higher time for other operations, and more complex code.

Designing and implementing a new data structure in Java

Every common programming language defines features that allows the programmer to describe the interface and implementation of a new data structure. In Java:

- The interface corresponds to public methods and fields.  
- If there is more than one way to implement a data structure, then we’ll often define a Java interface, which a particular class can implement (to provide the implementation).  
- The encoding typically corresponds to private fields. If the type of a field is itself a new data structure, we’ll sometimes define a private, inner class to describe that field’s type.  
- The implementation consists of the bodies of the public methods, along with any necessary private, helper methods.

Java Lists

The Java library defines an interface List<T>, where T is a type parameter that describes the type of values stored in the list.

Type parameters

A type parameter must be an Object, not a primitive. Fortunately, there are Object versions of primitive types, e.g., Integer.
**List implementations**

To use Java Lists, we need to import them from the `java.util` package.

The Java library defines classes `LinkedList<T>` and `ArrayList<T>`, which are particular implementations of the `List<T>` interface.

*Good programming practice*: whenever possible, declare the type of your variables to be interfaces and use classes only after the keyword `new`. For example:

```java
List<Integer> linkedValues = new LinkedList<Integer>();
List<Integer> arrayValues = new ArrayList<Integer>();
```

**Using the “foreach” loop to iterate over values**

Java has some nice syntax for iterating over collections (including Lists). For example:

```java
for (int i : linkedValues) {
    System.out.println(i);
}
```

**Java Maps**

A Java Map is like a Python dictionary.

The Java library defines an interface `Map<K, V>`, where `K` is a type parameter that describes the type of keys used in the map and `V` describes the type of values stored in the map.

To use Java Maps, we need to import them from the `java.util` package.

The Java library defines classes `HashMap<K, V>` (which we often use to store unordered data) and `TreeMap<K, V>` (which we often use to store ordered data).

**equals and hashCode**

So that we can store user-defined types in Maps, we follow some programming conventions:

- If two objects are `.equals`, their `.hashCode` methods should return the same value.
- Therefore, if we define an `.equals` method, we should define a `.hashCode` method.
- Often, we let Eclipse define both these methods for us.

**Java graphs**

The Java library doesn’t define graphs; but we can define them ourselves!

**Recall: representing graphs**

*Edge list*: store a list of edges (where an edge is stored as a triple of source, destination, weight).

*Adjacency list*: for each source node, store a list of adjacencies (where an adjacency is stored as a pair of destination, weight).

*Adjacency matrix*: a table that, for each pair of nodes, stores the cost of an edge between those nodes.

*Next time: bonus topics!*