Outline

- the Java Collection interface
- testing
- errors
- reasoning about programs

Collection Interface

- A collection holds values of a different type E
- the number of values can change over time
  - unless the collection is unmodifiable
- collections typically have at least two constructors
  - one with no arguments
  - one with a collection as an argument, makes a copy of the collection
- other methods:
  - size
  - search: contains, containsAll (why not containsAny?)
  - remove is also a search operation: remove, removeAll, removeIf, retainAll
  - add, addAll
  - toArray, iterator, spliterator, stream, parallelStream
- notice that Collection does not specify an ordering of elements

Collection Hierarchy

- List and Set are sub-interfaces of Collection
  - Lists are ordered
  - Sets are unordered and cannot have duplicates
- Many classes implement collection, including ArrayList, Vector, LinkedList, Stack, and at least 5 Set classes
- Because there are many possible constraints on adding values, add returns false if the element cannot be added
  - specifically if adding to a set that already has the value
  - if refusing to add for any other reason, e.g. if adding null to a collection that will not store null values, add throws an exception

Abstract collection classes

- to create a Collection:
  - extend AbstractCollection and implement add, size, and iterator
  - or extend AbstractList, and implement add, get, remove, set, and size
  - or extend AbstractSequentialList, and implement listIterator and size
- these abstract classes let you create a collection with a minimum of work

Testing

- unless code is written correctly from the start, errors are found by testing
- but writing correct code from the start is hard, so most programmers use testing to make their code as close to correct as possible
- test routines can be included in the `main` method of any class that doesn't already have one, or in a separate test program. This is a *unit test*. This main method is a *driver program*. The driver program can also be defined separately.
- the unit test should call all the methods of the class, with as many combinations of parameters as possible
- if the writer of the test code doesn't study the code under test, this is *black box testing*
- at the very least, the goal of testing is *full coverage*: making sure that every path through the code has been used at least once, and has produced an acceptable result
- to produce full coverage, the programmer of the test program must study the code being tested: this is *white box testing*

A few common types of errors

- off-by-one (fencepost): how many fence posts are needed for a fence that is 20 feet long and has a post every 2 feet?
- not initializing data correctly. Sometimes this causes null pointer access
- using different variables as if they were one, or using one variable as if it were two variables
- assumptions that don't turn out to be true (misconceptions), not establishing and maintaining invariants
- not checking things that should be checked, e.g.

```
if (x == null)
```

strategies for testing

- print/show all method invocations and their parameters and return values
- write code to check that the invariants are established and maintained
- write test cases to not only provide full coverage, but also check all boundary conditions, where the result should change (make sure it changes where it should)
- some common special cases:
  - less than 0, 0, 1, greater than 1
  - first and last elements of an array, collection, linked list
  - elements and values that are null
  - desired element is not in the collection, or is in the collection more than once
  - collection has size 0, 1, or larger
- for example, when testing adding on a linked list, can test adding at the beginning of a linked list, at the end, in the middle, and adding into an empty linked list, and in both positions of a 1-element linked list. Also, adding an element that has the value `null` (is the behavior of your program defined in that case? Should it be?)
- if code to be tested needs to call a method that is not yet implemented, a *stub* of that method can do only what is needed for the test

reasoning about programs

- a **precondition** must be true before a method is called
- the code in the method is designed assuming that the precondition is true
- the caller of a method must guarantee (be sure) that the precondition holds
- a **postcondition** will be true after a method is called
- the code in the method must guarantee that the postcondition is true
- preconditions and postconditions are a little bit like a contract or any other agreement: if the caller provides the preconditions, the method will provide the postconditions
- preconditions and postconditions are documented in Javadoc
- invariants are postconditions of every method, **including** the constructors
- invariants are preconditions of every method **except** the constructors
- invariants are usually documented for the entire class rather than for each method

proof of program correctness

- given a mathematical specification
- it might be possible to prove that a program implements that specification
- specification: `add1` adds 1 and returns the result

```
private static int add1(int parameter) {
  return parameter + 1;
}
```

- seems obvious
- but fails if parameter is `Integer.MAX_VALUE`