Outline

- Java review
- Interfaces
- Class Hierarchy, Abstract Classes, Inheritance
- Selection sort, Bubble sort, Insertion sort
- Invariants
- Lists
- ArrayList
- LinkedList
- runtime analysis
- Iterators
- Java references

Exam Preparation

- all the material in the lectures
- all the material in book in Chapters 1 and 2, and Section 8.5 and the Java review in appendix A
- all the material in the assignments
- all the quizzes
- invariants are not in the book: see the links instead
- review all the code posted on the course web page. Understand this code well enough to be able to code similar programs. Actual question may ask for something similar, but may be the same or different.
- understand what works in Java and what doesn't:
  - declare variables before using them
  - don't pretend that a method exists when you don't know whether it does
  - return values of the correct type from methods that have a non-void return type,
  - etc.

Java Review: Objects

- know when to create a new object
- static methods are called independently of an object, whereas instance methods (non-static methods) can only be called from an object, and only if the object is not `null`
- examples:
  - `String.length()` is an instance method. So if I have a `String s = ...`, I can call `s.length()`
  - but if `String s == null`, `s.length()` will throw a `NullPointerException`
  - `Arrays.copyOf()` is a static method. So I can call it without having an object of type `Arrays`
- can only use whatever methods the class provides, including methods provided by the superclasses
- `toString()`, `equals` are available for all objects
- should provide at least one constructor for each object, though Java provides a default construcor if you don't have any constructors

Java Review: General

- initialize all variables
- parameters must match, and return values must be returned
- parameters types must match, and return types must match, so for example

```
public static int foo() {
   return true;
}
```

  is incorrect
- array indexing and modulo (what is `27 % 8`?)
- private classes

Java Review: Exceptions

- exceptions can be thrown and caught
- exceptions that are not runtime exceptions must be declared in the method header

```
public static int foo(int n) throws java.io.IOException {
  if (n < 0)
    throw new java.io.IOException();
  else if (n == 0)
    throw new java.util.NoSuchElementException();
  else
    return n - 1;
}
```

Abstract Data Types

- a collection of attributes and behaviors makes up an abstract data type
- similar to the notion of a data type in programming: an `int` type has an attribute (the value) and several behaviors (addition, subtraction, etc) but not others (.length)
- but an ADT can be much more general than a data type
- ADTs are useful when designing programs
  - often a class definition is inspired by an ADT
- there are many intuitive ADTs for real-world objects
- ADTs for programs become more intuitive with practice
  - What is the difference between an Abstract Data Type and an Abstract Class?

Interfaces

- an interface is a list of methods, with their types
- interfaces may also include constants
- the names of method parameters are significant to the reader, but not to the compiler
- any class that implements an interface must provide the corresponding methods
- example: Iterable

```
public interface Iterable<E> {
   java.util.Iterator<E> iterator();
}
```

- for the exam, you need to know the syntax of interfaces, i.e. how to write an interface

Interfaces as Types

- interfaces can be used as Java types:
- any object that implements the interface can be assigned to a variable whose type is the interface

  for example:

```
LinkedList<String> list = new LinkedList<String>();
Iterable<String> iterable = list;
```

- Later, could have

```
Iterator<String> iter = iterable.iterator();
```

- **but**: we cannot call `iterable.add("foo")`, even though `iterable` and `list` refer to the same object, and `list.add("foo")` is perfectly legal

Class Hierarchy, Abstract Classes, Inheritance

- each class except `Object` has a superclass
- each class `extends` exactly one superclass (if not explicitly stated, it extends `Object`)
- `this` and `super` references can be used within any instance method
- every method defined in the superclass is inherited by the subclass
- a class that redefines a superclass's methods is overriding them
- a variable declared to have an interface type can refer to any object that satisfies (`implements`) that interface
- an abstract class does not have constructors or objects. Instead, other classes may inherit from the abstract class and use the methods it does define
  - and must implement any methods that are abstract in the superclass

Bubble Sort, Selection Sort, Insertion Sort

- bubble sort: repeatedly compare adjacent elements, swap them if they are unsorted. Repeat until the entire list has no elements to swap
  - takes linear time on a sorted array or sorted linked list
- selection sort: repeatedly select the smallest element from the unsorted part of the list, add it to the sorted part
  - always takes quadratic time, because finding the minimum takes linear time
- insertion sort: repeatedly take the next element from the unsorted part of the list, insert it in the right position in the sorted part
  - takes linear time on a sorted array or sorted doubly-linked list
  - also fast on a list where most elements are near their sorted position
- all these algorithms are $O(n^2)$
  - because in the worst case, the inner loop takes linear time, and is executed a linear number of times
  - later in the semester we will study O(n log n) sorting algorithms

Invariants

- an invariant is something the programmer believes to be true about relationships among the program's variables
- in an object-oriented language such as Java, the most important relationships are among an object's instance variables (class data fields)
- if a public method is called with the invariant being true, the invariant should still be true at the end of the method call
- an invariant should always be true at the end of a public constructor

Lists

- variable sized collections of objects in a particular sequence
- parametrized on the type of the objects that are stored in the list
- duplicate and null elements are permitted
- the List interface defines required operations, and the AbstractList class helps in implementing new list classes
- some particularly useful methods:
  - `boolean add(E e)` add at the end of a list
  - `void add(int index, E e)` add within the list
  - `Iterator<E> iterator()` create a new iterator
  - `E remove(int index)` remove and return the object at the given position
  - `int size()`

Array Lists

- a list in which the elements are stored in an array
- the array may have more elements than the list, so an instance variable (`size`) is needed to keep track of the size of the list
  - `capacity()` is the size of the underlying array
- as the list grows, may need a new array, so adding at the end of the array is O(n)
  - but is amortized constant time as long as the array only grows by doubling the capacity
- adding and removing at the beginning or in the middle of the array is always O(n) -- make sure you understand why

Linked Lists

- a list in which each element is stored in a node, and nodes are linked to each other
- the number of nodes is exactly the same as the number of elements
- each node has a reference to the value stored (`item`, `data`, `value`, or a similar name) and a reference to the next node in the list, if any (always called `next`)
- the end of the list is reached when the `next` field has the value `null`
- a linked list must store the head of the list
- adding at the end of the linked list is faster if a tail pointer is kept -- O(1) instead of O(n)
- adding and removing at the front of a linked list is always O(1)

Circular and Doubly-Linked Lists

- in a circular list, the last `next` field refers (back) to the head of the list
- in a doubly-linked list, each node has a reference to the node before it (`prev` or `previous`)
  - the previous node of the head is null, and the next node of the tail is null
- in a doubly-linked circular list, the tail is the previous node for the head, and the head is the next node for the tail

runtime analysis

- need to meaningfully be able to compare algorithms, independently of which computer they run on
- so ignore constant factors of speed difference
- if two algorithms have different growth rates, e.g. n^2 and n^3, then for large enough n, the n^3 will take longer even if the constant factor is very small
- big-O notation: only consider the fastest growing term in the equation, so $n^2 + n \log n + 5n + 2$ is $O(n^2)$
- in general, $O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n^2) < O(n^3) < O(2^n)$
- this also means that $O(n) < O(n \log n) < O(n \sqrt{n}) < O(n^2)$
- if in doubt, review the loops program (and run it yourself!) and make sure you understand the growth rates and where they come from
- only consider the worst-case running time
- on the exam, you may be asked to do runtime analysis of simple programs or code snippets, or of algorithms studied in this class

iterators

- an iterator is an object that keeps track of the state of a traversal of a collection
- an iterator is like a bookmark for a book: there can be many bookmarks for a given book
- calling the `iterator()` method of a collection class is one way to create a new iterator
- once the iterator exists, calls to `hasNext()` and `next()` may continue until there are no more elements
- or, calling `next()` until it throws an exception
- for example:

```
List<E> list = ...
Iterator<E> iter = list.iterator();
while (iter.hasNext()) {
  E variable = iter.next();
  // can use "variable" in the loop
}
```

iterators: foreach

- as an alternative to explicitly calling the iterator methods, use the foreach style:

```
for (E variable: Iterable<E>) {
   // can use "variable" in the loop
}
```

- for the exam, must understand and be able to implement at least the hasNext() and next() methods of these two iterators: LinkedListIterator.java, and ArrayListIterator.java

Java references and equality

- every Java variable (that is not of a basic type) is a reference to an object, and may be `null`
- two objects can be equal in different ways:
  - `a == b` if and only if both references are to the same object, or both are `null`
  - `a.equals(b)` if `a` is not `null`, and `a`'s `equals` method returns `true` when given `b` as a parameter
  - usually (e.g. for strings), `a.equals(b)` if the contents of `a` match the contents of `b`
    - but only if provided by a .equals method in the class itself
- for example:

```
String a = new String("foo");
String b = new String("foo");
if (a == b) {
  System.out.println("two different objects are equal!  Something is wrong");
}
if (a.equals(b)) {
  System.out.println("foo is equal to foo.  Life is good");
}
```

Java references: parameters

- if I pass an `int` as parameter to a method, and the method changes the value of the int, my value does not change. For example:

```
int x = 3;
m(x);
```

  the value of x is still 3 after the call, no matter what m does.

- if I pass an object as parameter to a method, and the method changes what object the parameter refers to, my value does not change. For example:

```
LinkedList<Integer> x = new LinkedList<Integer>();
...
m(x);
...
void m(LinkedList<Integer> parameter) {
     parameter = new LinkedList<Integer>();
}
```

  x still refers to the original linked list (not the new list) after the call.

Java references: mutable objects

- if I pass an object as parameter to a method, and the method changes the values stored in the object, my value is changed as well. For example:

```
LinkedList<Integer> x = new LinkedList<Integer>();
...
m(x);
...
void m(LinkedList<Integer> parameter) {
    parameter.add(13);
}
```

- after this call, x refers to the same list, which now has a new element

Outline

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

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`