Outline

- Java review and Java references
- interfaces, class hierarchy, abstract classes, inheritance
- selection sort, bubble sort, insertion sort
- binary search
- runtime analysis
- recursion

Exam Preparation

- **all the material in the lectures**
- **textbook Chapter 1, Sections 2.4, 5-5.3 and 8-8.4, and the Java review in appendix A**
- **all the material in assignments 1-5 (need to have read and understood and thought about assignment 5)**
- **all the quizzes**
- **review all the code posted on the course web page. Understand this code well enough to be able to code similar programs**
  - the actual question may ask for something similar, and the corresponding code may be the same or different.
- **understand what works in Java:**
  - declare and initialize variables before using them
  - methods that have a non-void return type must return values of the correct type
  - don't pretend that a method exists when you don't know whether it does
  - etc.

Java review: general

- initialize all variables
- parameters must match, and return values must be returned

  so for example:

```
public static int foo() {
  if (n < 0) return;
  System.out.println("hello world");
  return true;
}
```

  is incorrect: wrong return type, undeclared variable n, unhelpful print

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

Java review: objects

- **`new` creates a new object**
- **static methods are called independently of an object, whereas instance methods (non-static methods) can only be called on an object, so only if the object reference is not `null`**
  - `String.length()` is an instance method. So if I have a `String s = ...`, I can call `s.length()`
  - but if `String s` is `null`, calling `s.length()` will throw a `NullPointerException`
  - `Arrays.copyOf()` is a static method. So I can call it without having an object of type `Arrays`
- **only use whatever methods the class actually provides, including methods provided by the superclasses**
  - `toString()` and `equals()` are available for all objects
  - these are instance methods, so cannot be used when the object reference is `null`
- **the code for each class should include at least one constructor, though Java provides a default constructor if your code doesn't have any constructors**

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 hello(int n) throws java.io.IOException {
  if (n < 0)
    throw new java.io.IOException();     // not a RuntimeException
  else if (n == 0)
    throw new java.util.NoSuchElementException(); // a RuntimeException
  else
    return n - 1;
}
```

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`
    - equals method contain Java code, so does whatever the programmer chooses

```
String a = new String("hello");

String b = new String("hello");

if (a == b) {

  System.out.println("two different objects are equal!  Something is wrong");

}

if (a.equals(b)) {

  System.out.println("hello is equal to hello.  Life is good");

}
```

Java references: parameters

- if I pass an int as parameter to a method, and the method changes the value of its int parameter, 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 to its parameter
- this holds for any parameter of a basic type: int, long, float, double, short, byte, char, or boolean

Java references: mutable objects

- **if I pass an object as parameter to a method, and (1) the method changes the values stored in the object, my value is changed as well, but (2) an assignment to the parameter changes the reference without changing the object that the caller refers to.**

```
void m(int[] parameter) {

    parameter[55] = 22;

    parameters = new int[10];

}

...

int[] x = new int[100];

...

m(x);
```

- **after the call to m, x refers to the same 100-element array as before, which now has a new value at location 55**

Interfaces

- **an interface is a list of methods with their return types and parameter types**
  - no constructors, no variables
  - the keyword `public` only in front of the keyword `interface`, not in front of the method headers
- **interfaces may 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: Comparable**

```
public interface Comparable<T> {

  int compareTo(T object);

}
```

- **for the exam, you need to know the syntax of interfaces, i.e. how to write an interface**
- **for the exam, you should be comfortable with the Comparable interface**

Using 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**

```
String hello = new String("hello world");
Comparable<String> comp = hello;
```

- **Later, could have**

```
int comparison = comp.compareTo("yes");
```

- **but: we cannot call `comp.length()`, even though `comp` and `hello` refer to the same object, and `hello.length()` is legal**
  - because `comp` is a reference to a `Comparable<String>`, so does not have access to methods of class `String`
  - so the type of a reference determines what method you may use
  - but the type of the object determines which method is actually called, e.g. when calling .toString

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 implemented in the superclass is inherited by the subclass (except for private methods)**
- **a class that redefines a superclass's methods is overriding them**
  - overloading refers to different methods with the same name, distinguished by parameter types
- **a variable declared to have an interface type can refer to any object that satisfies (`implements`) that interface**
- **an abstract class does not objects, and may have no constructors. Instead, other classes inherit and use methods the abstract class implements, 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 its sorted 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 worst-case $O(n^2)$, quadratic time**
  - because in the worst case, the inner loop takes linear time, and is executed a linear number of times
  - the average time to sort a random array is also $O(n^2)$ / quadratic for all three algorithms
  - later in the semester we will study O(n log n) sorting algorithms

Binary search

- if an array is sorted, search can be much faster than linear search
- start by looking in the middle, and in O(1) – constant time – determining whether the value searched is in the left or right half of the array
- at each loop or recursive call the sub-array – the part of the array we are still looking at – is half the size of the previous loop or call
- so the overall time is O(log n) both in the worst and in the average case
- should be comfortable with the code for binary search, both recursive and looping/iterative

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 multiplied by a very small factor**
- **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, $O(1) < O(\log n) < O(\text{sqrt}(n)) < O(n) < O(n^2) < O(n^3) < O(2^n)$**
  - constant, log, square-root, linear, quadratic, cubic, exponential
- **this also means that $O(n) < O(n \log n) < O(n \text{ sqrt}(n)) < O(n^2)$**
- **unless told otherwise, 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**

recursion

- **recursion is useful when we have a problem that:**
  - has an easy solution for some base cases, and
  - for all other cases, has a solution that can be expressed in terms of solving a problem that is closer to the base case.
- **the problem that is closer to the base case is often a smaller problem**
  - e.g., smaller value of n
  - e.g., smaller subarray in binary search
- **recursion is a way of thinking as well as a programming technique**
  - mathematicians often think recursively

recursion in Java

- **recursion in Java – and in most programming languages – is only available by creating a new method**
- **the parameters of the method encode the problem, and the value of at least one of the parameters must be different on each call**
  - otherwise, infinite recursion is guaranteed
- **on each recursive call, the parameters must come closer to the base case**
  - for an appropriate meaning of “closer”

recursion is easier than loops for:

- **anything that requires reversing the order of something:**
  - in printing a number, the high-order digits should be printed first, but the low-order digits are more easily accessible (using modulo)
- **operation on recursive data structures such as linked lists or trees (not yet studied in this class)**
- **anything that uses multiple recursive calls, for example ackermann, fib, or backtracking / maze traversal**

in many cases, recursion is equivalent to loops

- **repeated operations can be usually implemented as either loops or recursive methods**
- **in most of these cases it doesn't matter much whether the solution is recursive or iterative (except as required on assignments and exams, or by your boss)**
  - operations on arrays
  - binary search
  - arbitrary (terminating) loops
- **any loop can be replaced by calls to a recursive method**

Exam Preparation (repeat)

- **all the material in the lectures**
- **textbook Chapter 1, Sections 2.4, 5-5.3 and 8-8.4, and the Java review in appendix A**
- **all the material in assignments 1-5 (need to have read and understood and thought about assignment 5)**
- **all the quizzes**
- **review all the code posted on the course web page. Understand this code well enough to be able to code similar programs**
  - the actual question may ask for something similar, and the corresponding code may be the same or different.
- **understand what works in Java:**
  - declare and initialize variables before using them
  - methods that have a non-void return type must return values of the correct type
  - don't pretend that a method exists when you don't know whether it does
  - etc.