Recreation

An integer is divided by 9 when a certain one of its digits is c
and the resulting number is again divisible by 9.

a. Prove that actually dividing the resulting number by 9 re
   deleting another digit.

b. Find all integers satisfying the conditions of this problem.

Announcements

- Project 0 autograder has been running. Check the Scores
  results.
- Yes, you can resubmit. See the Course Info tab.
- In particular, *many* people need to do style fixes! Use `mak`
  or `style61b signpost/*.java` to check before submission.

Project Ethics

Basic Rules:

1. By You Alone: All major submitted non-skeleton code should
   ten by you alone.
2. Do Not Possess or Share Code: Before a project deadline, yo
   never be in possession of solution code that you did not wr
   distribute your own code to others in the class.
3. Cite Your Sources: When you receive significant assistan
   project from someone else (other than the staff), cite tho
   tance somewhere in your source code.

Ethical Collaboration

Unproblematic

- Discussion of approaches for solving a problem.
- Giving away or receiving significant ideas towards a problem s
  if cited.
- Discussion of specific syntax issues and bugs in your code.
- Using small snippets of code that you find online for solv
  problems (e.g. googling "uppercase string java" may lead you
  sample code that you copy and paste. Cite these.

Requiring Great Caution:

- Looking at someone else's project code to assist with debug
- Looking at someone else's project code to understand a pa
  idea or part of a project. Generally unwise though, due to the
  of plagiarism.

Unethical Collaborations

- Possessing another student's project code in any form befor deadline, or distributing your own.
- Possessing project solution code that you did not write your fore a final deadline (e.g., from github, or from staff soluti found somewhere). Likewise, distributing such code.

CS61B Lecture #11: Examples: Comparable & Re

Comparable

- Java library provides an interface to describe Objects th a *natural order* on them, such as `String`, `Integer`, `BigInte` `BigDecimal`:

```
public interface Comparable { // For now, the Java 1.4 vers
  /** Returns value <0, == 0, or > 0 depending on whether T
   *  <, ==, or > OBJ.  Exception if OBJ not of compatible
  int compareTo(Object obj);
}
```

- Might use in a general-purpose max function:

```
/** The largest value in array A, or null if A empty. */
public static Comparable max(Comparable[] A) {
  if (A.length == 0) return null;
  Comparable result; result = A[0];
  for (int i = 1; i < A.length; i += 1)
    if (result.compareTo(A[i]) < 0) result = A[i];
  return result;
}
```

- Now `max(S)` will return maximum value in `S` if `S` is an array of or any other kind of Object that implements `Comparable`.

Examples: Implementing Comparable

```
/** A class representing a sequence of ints. */
class IntSequence implements Comparable {
    private int[] myValues;
    private int myCount;
    ...
    public int get(int k) { return myValues[k]; }

    @Override
    public int compareTo(Object obj) {
       IntSequence x = (IntSequence) obj; // Blows up if obj no
       for (int i = 0; i < myCount && i < x.myCount; i += 1) {
           if (myValues[i] < x.myValues[i]) {
              return -1;
           } else if (myValues[i] > x.myValues[i]) {
              return 1;
       }
       return myCount - x.myCount;  // <0 iff myCount < x.myCou
    }
}
```

Implementing Comparable II

- Also possible to add an interface retroactively.
- If `IntSequence` did not implement `Comparable`, but did im
  `compareTo` (without `@Override`), we could write

```
class ComparableIntSequence extends IntSequence implements

}
```

- Java would then "match up" the `compareTo` in `IntSequence` w
  in `Comparable`.

Java Generics (I)

- We've shown you the old Java 1.4 `Comparable`. The current uses a newer feature: Java generic types:

```
public interface Comparable<T> {
    int compareTo(T x);
}
```

- Here, `T` is like a formal parameter in a method, except "value" is a *type*.
- Revised `IntSequence` (no casting needed):

```
class IntSequence implements Comparable<IntSequence> {
    ...
    @Override
    public int compareTo(IntSequence x) {
       for (int i = 0; i < myCount && i < x.myCount; i += 1
           if (myValues[i] < x.myValues[i]) ...

       return myCount - x.myCount;
    }
}
```

Example: Readers

- Java class `java.io.Reader` abstracts *sources of characters*
- Here, we present a revisionist version (not the real thing):

```
public interface Reader {  // Real java.io.Reader is abstra
  /** Release this stream: further reads are illegal */
  void close();

  /** Read as many characters as possible, up to LEN,
   *  into BUF[OFF], BUF[OFF+1],..., and return the
   *  number read, or -1 if at end-of-stream. */
  int read(char[] buf, int off, int len);

  /** Short for read(BUF, 0, BUF.length). */
  int read(char[] buf);

  /** Read and return single character, or -1 at end-of-str
  int read();
}
```

- Can't write `new Reader()`; it's abstract. So what good is it?

Generic Partial Implementation

- According to their specifications, some of Reader's methods lated.
- Can express this with a *partial implementation,* which lea methods unimplemented and provides default bodies for oth
- Result still abstract: can't use **new** on it.

```
/** A partial implementation of Reader. Concrete
 *  implementations MUST override close and read(,,).
 *  They MAY override the other read methods for speed. */
public abstract class AbstractReader implements Reader {
  // Next two lines are redundant.
  public abstract void close();
  public abstract int read(char[] buf, int off, int len);

  public int read(char[] buf) { return read(buf,0,buf.lengt

  public int read() { return (read(buf1) == -1) ? -1 : buf1

  private char[] buf1 = new char[1];
}
```

Implementation of Reader: StringReader

The class StringReader reads characters from a String:

```
public class StringReader extends AbstractReader {
  private String str;
  private int k;
  /** A Reader that delivers the characters in STR. */
  public StringReader(String s) {
      str = s; k = 0;
  }

  public void close() {
      str = null;
  }

  public int read(char[] buf, int off, int len) {
      if (k == str.length())
          return -1;
      len = Math.min(len, str.length() - k);
      str.getChars(k, k+len, buf, off);
      k += len;
      return len;
  }
}
```

Using Reader

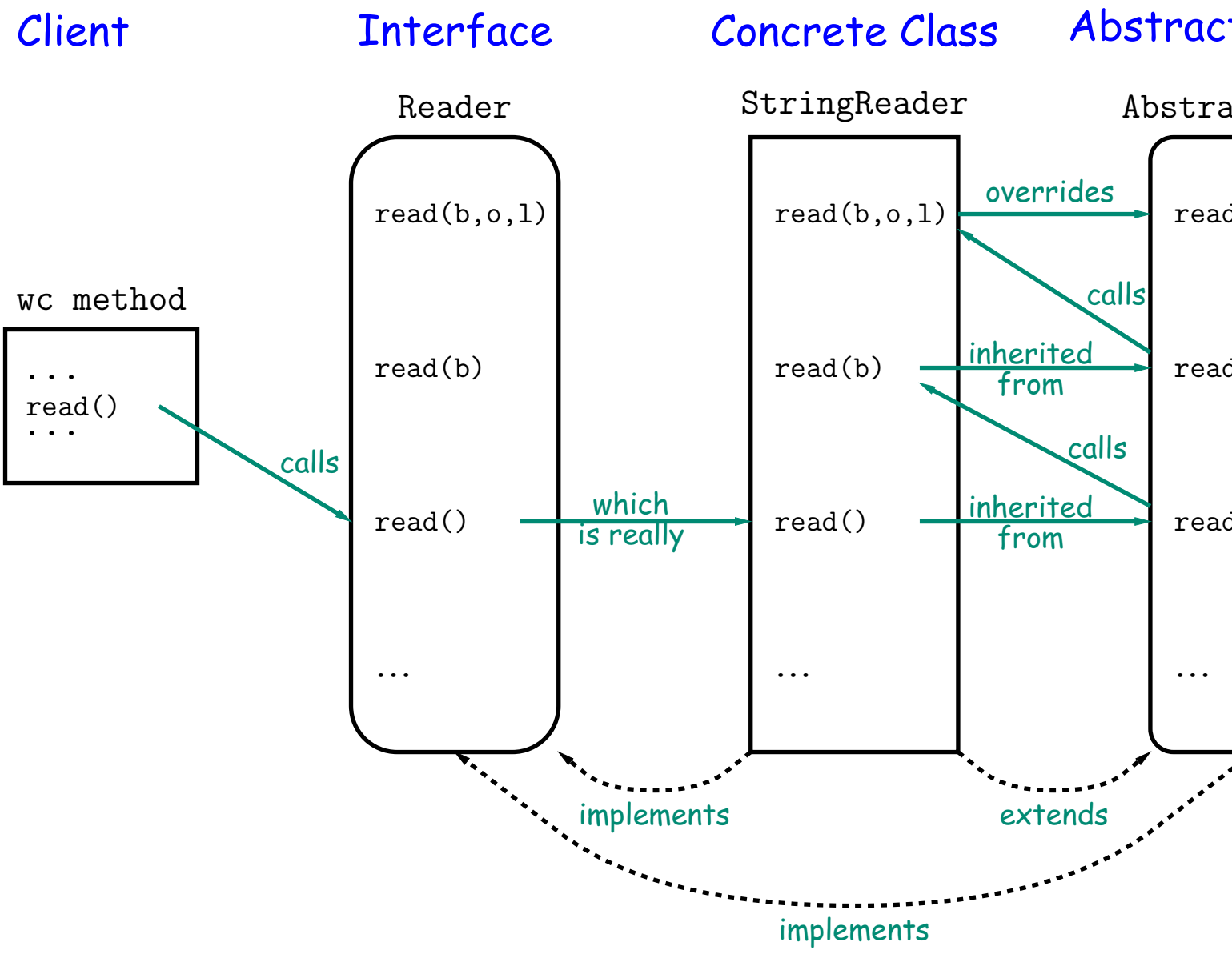
Consider this method, which counts words:

```
/** The total number of words in R, where a "word" is
 *  a maximal sequence of non-whitespace characters. */
int wc(Reader r) {
  int c0, count;
  c0 = ' '; count = 0;
  while (true) {
      int c = r.read();
      if (c == -1) return count;
      if (Character.isWhitespace((char) c0)
          && !Character.isWhitespace((char) c))
          count += 1;
      c0 = c;
  }
}
```

This method works for *any* `Reader`:

```
wc(new StringReader(someText))          // # words in so
wc(new InputStreamReader(System.in))    // # words in st
wc(new FileReader("foo.txt"))           // # words in fi
```

How It Fits Together



Lessons

- The `Reader` interface class served as a *specification* for a w
of readers.
- Ideally, most client methods that deal with `Readers`, like
specify type `Reader` for the formal parameters, not a speci
of `Reader`, thus assuming as little as possible.
- And only when a client creates a new `Reader` will it get specif
what subtype of `Reader` it needs.
- That way, client's methods are as *widely applicable* as possib
- Finally, `AbstractReader` is a tool for implementors of non-a
`Reader` classes, and not used by clients.
- Alas, Java library is not pure. E.g., `AbstractReader` is rea
called `Reader` and there is no interface. In this example,
what they *should* have done!
- The `Comparable` interface allows definition of functions t
pend only on a limited subset of the properties (methods)
arguments (such as "must have a `compareTo` method").