Recreation

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

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

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

CS61B Lecture #11: Examples: Comparable & Reader + Some Features Supporting Abstraction

Comparable

- Java library provides an interface to describe Objects that have a *natural order* on them, such as String, Integer, BigInteger and BigDecimal:

```
public interface Comparable { // For now, the Java 1.4 version
  /** Returns value <0, == 0, or > 0 depending on whether THIS is
   *  <, ==, or > OBJ.  Exception if OBJ not of compatible type. */
  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 Strings, 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 not an IntSequence
       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.myCount
    }
}
```

Implementing Comparable II

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

```
class ComparableIntSequence extends IntSequence implements Comparable {

}
```

- Java would then "match up" the compareTo in IntSequence with that in Comparable.

Java Generics (I)

- We've shown you the old Java 1.4 Comparable. The current version 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 that its "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 abstract class
  /** 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-stream. */
  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 are related.
- Can express this with a *partial implementation,* which leaves key methods unimplemented and provides default bodies for others.
- 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.length); }

  public int read() { return (read(buf1) == -1) ? -1 : buf1[0]; }

  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

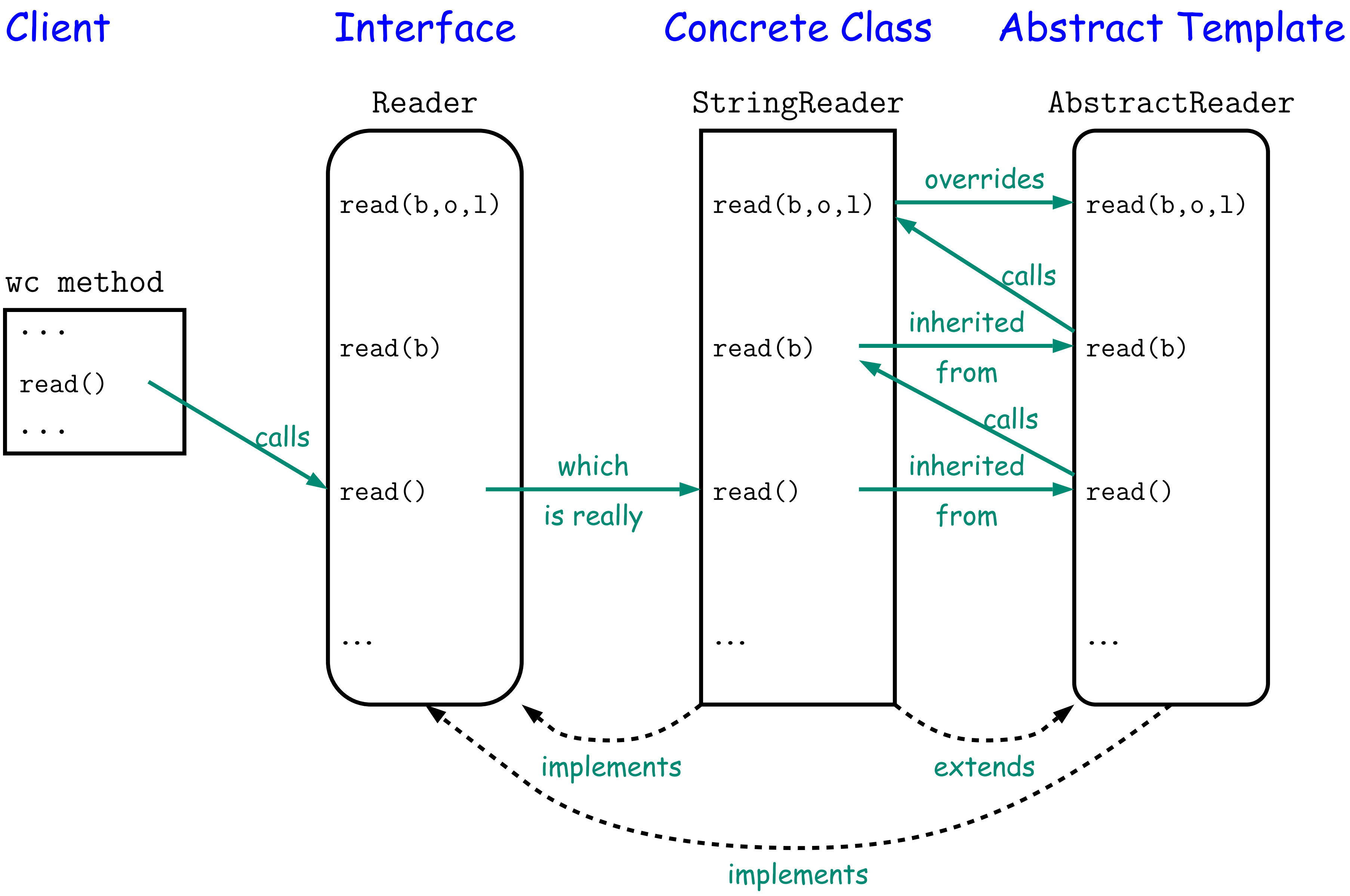
Consider this method, which counts words:

```
/** 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 someText
wc(new InputStreamReader(System.in))    // # words in standard input
wc(new FileReader("foo.txt"))           // # words in file foo.txt.
```

How It Fits Together



Client

Interface

Concrete Class

Abstract Template

wc method

...

read()

...

Reader

read(b,o,l)

read(b)

read()

...

StringReader

read(b,o,l)

read(b)

read()

...

AbstractReader

read(b,o,l)

read(b)

read()

...

Lessons

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

More OOP Features Supporting Abstraction

Parent Constructors

- In lecture notes #7, talked about how Java allows implementer of a class to control all manipulation of objects of that class.
- In particular, this means that Java gives the constructor of a class the first shot at each new object.
- When one class extends another, there are two constructors—one for the parent type and one for the new (child) type.
- In this case, Java guarantees that *one of the parent's constructors is called first.* In effect, there is a call to a parent constructor at the beginning of every one of the child's constructors.
- You can call the parent's constructor yourself explicitly.

```
class Figure {
  public Figure(int sides) {
      ...
  }...
}
```

```
class Rectangle extends Figure {
   public Rectangle() {
       super(4);
   }...
}
```

Default Constructors

- By default, Java calls the “default” (parameterless) constructor if there is no explicit constructor called.

```
  /* This... */
class Thingy extends Rectangle {
    public Thingy() {
       setThingsUp();
    }
}
```

```
  /* Is equivalent to... */
class Thingy extends Rectangle {
    public Thingy() {
        super();
        setThingsUp();
    }
}
```

- And it creates a default constructor for a class if no other constructor is defined for the class.

```
  /* This... */
class Crate {
}
```

```
/* Is equivalent to... */
class Crate {
    public Crate() {
    }
}
```

```
/* And thus to... */
class Crate {
    public Crate() {
        super();
    }
}
```

What Happens Here?

```
class Figure {
  public Figure(int sides) {
      ...
  }
}
```

```
class Rectangle extends Figure {
}
```

What Happens Here?

```
class Figure {                          class Rectangle extends Figure {
  public Figure(int sides) {            }
     ...
  }
}
```

Answer: Compiler error. `Rectangle` has an implicit constructor that tries to call the default construvtor in `Figure`, but there isn't one.

Using an Overridden Method

- Suppose that you wish to *add* to the action defined by a superclass's method, rather than to completely override it.
- The overriding method can refer to overridden methods by using the special prefix super.
- For example, you have a class with expensive functions, and you'd like a memoizing version of the class.

```
class ComputeHard {
  int cogitate(String x, int y) { ... }
}

class ComputeLazily extends ComputeHard {
  int cogitate(String x, int y) {
    if (don't already have answer for this x and y) {
      int result = super.cogitate(x, y);  // <<< Calls overridden function
      memoize (save) result;
      return result;
    }
     return memoized result;
  }
}
```