Recreation

vided by 9 when a certain one of its digits is deleted, g number is again divisible by 9.

ıctually dividing the resulting number by 9 results in ther digit.

ıers satisfying the conditions of this problem.

re #11: Examples: Comparable & Reader + ıe Features Supporting Abstraction

Comparable

provides an interface to describe Objects that have *der* on them, such as String, Integer, BigInteger and

```
erface Comparable { // For now, the Java 1.4 version
rns value <0, == 0, or > 0 depending on whether THIS is
=, or > OBJ.  Exception if OBJ not of compatible type. */
areTo(Object obj);
```

a general-purpose max function:

```
rgest value in array A, or null if A empty. */
tic Comparable max(Comparable[] A) {
ngth == 0) return null;
le result; result = A[0];
 i = 1; i < A.length; i += 1)
sult.compareTo(A[i]) < 0) result = A[i];
esult;
```

will return maximum value in S if S is an array of Strings, kind of Object that implements Comparable.

amples: Implementing Comparable

```
presenting a sequence of ints. */
nce implements Comparable {
t[] myValues;
t myCount;

 get(int k) { return myValues[k]; }


 compareTo(Object obj) {
ence x = (IntSequence) obj; // Blows up if obj not an IntSequence
t i = 0; i < myCount && i < x.myCount; i += 1) {
(myValues[i] < x.myValues[i]) {
return -1;
lse if (myValues[i] > x.myValues[i]) {
return 1;

myCount - x.myCount;  // <0 iff myCount < x.myCount
```


Implementing Comparable II

: to add an interface retroactively.

nce did not implement Comparable, but did implement vithout @Override), we could write

```
arableIntSequence extends IntSequence implements Comparable {
```

hen "match up" the compareTo in IntSequence with that e.

Java Generics (I)

you the old Java 1.4 Comparable. The current version feature: Java generic types:

```
terface Comparable<T> {
ompareTo(T x);
```

ke a formal parameter in a method, except that its *ype*.

equence (no casting needed):

```
equence implements Comparable<IntSequence> {

ide
 int compareTo(IntSequence x) {
 (int i = 0; i < myCount && i < x.myCount; i += 1) {
 if (myValues[i] < x.myValues[i]) ...

rn myCount - x.myCount;
```

Generic Partial Implementation

 their specifications, some of Reader's methods are re-

 this with a *partial implementation,* which leaves key
nplemented and provides default bodies for others.

bstract: can't use **new** on it.

```
ial implementation of Reader. Concrete
entations MUST override close and read(,,).
AY override the other read methods for speed. */
tract class AbstractReader implements Reader {
two lines are redundant.
bstract void close();
bstract int read(char[] buf, int off, int len);

nt read(char[] buf) { return read(buf,0,buf.length); }

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

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


Using Reader

ethod, which counts words:

```
number of words in R, where a "word" is
sequence of non-whitespace characters. */
r) {
t;
unt = 0;
 {
r.read();
 -1) return count;
acter.isWhitespace((char) c0)
Character.isWhitespace((char) c))
t += 1;
```

rks for *any* Reader:

```
Reader(someText))        // # words in someText
treamReader(System.in))  // # words in standard input
ader("foo.txt"))         // # words in file foo.txt.
```


Lessons

nterface class served as a *specification* for a whole set

t client methods that deal with Readers, like wc, will
 Reader for the formal parameters, not a specific kind
hus assuming as little as possible.

n a client creates a new Reader will it get specific about
e of Reader it needs.

ent's methods are as *widely applicable* as possible.

ractReader is a tool for implementors of non-abstract
es, and not used by clients.

brary is not pure. E.g., AbstractReader is really just
r and there is no interface. In this example, we saw
hould have done!

able interface allows definition of functions that de-
 a limited subset of the properties (methods) of their
uch as "must have a compareTo method").

Example: Readers

ava.io.Reader abstracts *sources of characters.*

sent a revisionist version (not the real thing):

```
erface Reader {  // Real java.io.Reader is abstract class
ase this stream: further reads are illegal */
se();

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

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

 and return single character, or -1 at end-of-stream. */
();
```

ew Reader(); it's abstract. So what good is it?

ementation of Reader: StringReader

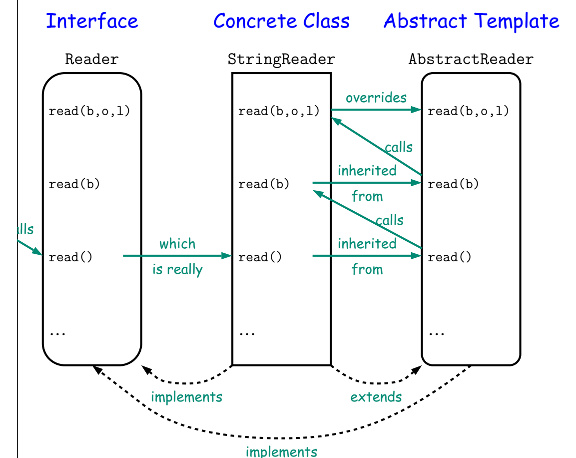
gReader reads characters from a String:

```
tringReader extends AbstractReader {
ng str;
k;
 that delivers the characters in STR. */
gReader(String s) {
 k = 0;

close() {
ll;

ead(char[] buf, int off, int len) {
 str.length())
rn -1;
th.min(len, str.length() - k);
hars(k, k+len, buf, off);
;
en;
```


How It Fits Together



OOP Features Supporting Abstraction

Parent Constructors

ɔtes #7, talked about how Java allows implementer of a
rol all manipulation of objects of that class.

, this means that Java gives the constructor of a class
ɔt at each new object.

ass extends another, there are two constructors—one
nt type and one for the new (child) type.

Java guarantees that *one of the parent's constructors*
t. In effect, there is a call to a parent constructor at
g of every one of the child's constructors.

the parent's constructor yourself explicitly.

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


Default Constructors

Java calls the "default" (parameterless) constructor if
xplicit constructor called.

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

es a default constructor for a class if no other con-
efined for the class.

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


What Happens Here?

```
                class Rectangle extends Figure {
int sides) {    }
```


What Happens Here?

```
                class Rectangle extends Figure {
int sides) {    }
```

iler error. Rectangle has an implicit constructor that
default construvtor in Figure, but there isn't one.

Using an Overridden Method

t you wish to *add* to the action defined by a superclass's
ier than to completely override it.

ng method can refer to overridden methods by using
refix super.

, you have a class with expensive functions, and you'd
zing version of the class.

```
Hard {
e(String x, int y) { ... }

Lazily extends ComputeHard {
e(String x, int y) {
already have answer for this x and y) {
sult = super.cogitate(x, y);  // <<< Calls overridden function
e (save) result;
 result;

nemoized result;
```