CONTAINERS AND VARIABLES
CONTAINERS FOR HOLDING OBJECTS

what sort of **object** is:
  a bag of rocks?
  a horde of monsters?
  the stash of weapons a player is carrying?
  the suite of rooms in a game?

does a bag of rocks extend Rock?
does a horde of monsters extend Monster?
does a stash of weapons extend Weapon?
does a suite of rooms extend Room?

think of bags, hordes, stashes, and suites as objects in their own right.

collection, or container, objects

what operations would you perform on such an object?

what would be its responsibilities?
A VARIETY OF CONTAINERS

ordered (sequenced)
  linear, tree, graph
  fixed vs. dynamic order
unordered (random or don't care)
homogeneous (all instances of same type)
  e.g. arrays
heterogeneous
  of different, but related, sub-type
  of arbitrary type (junk drawer)
constructed with initial contents, or not
methods for:
  adding/removing elements
  delivering an element for "scrutiny"
  count of contents
  accepting visitors

why put objects in containers?
  to organize by type
  to organize by use to be made of them
  to implement notion of ownership
  to filter, catalog, ...
you realize you don't have any more clean socks, so all elements of DL (socks, shirts, ...) are accumulated in one container (the LB) ...

the DL elements are then sorted into two piles according to whether they are color versus white then each pile is washed separately the piles are then placed back into the LB, and etc.

operations: adding, removing, sorting, filtering, ...
one-at-a-time examination of each item (“iteration”)
if (!socksDrawer.more()) {
    LB lb = new LB("Rubbermaid");
    DL item;
    while (bedroomFloor.hasNext()) {
        item = bedroomFloor.get();
        bedroomFloor.remove();
        lb.add(item);
    }
    Pile whitePile = new Pile();
    Pile colorPile = new Pile();
    While (lb.hasNext()) {
        item = lb.get();
        lb.remove(); // take it out of the lb
        if (item instanceof WhiteL) {
            whitePile.add(item);
        } else {
            colorPile.add(item);
        }
    }
    Washer washer = new Washer("LG");
    While (whitePile.hasNext()) {
        washer.add(whitePile.get());
        whitePile.remove();
    }
}
some containers are ordered:
column of soldier ants
row of ducks
some containers are not (and do not allow duplicates):
set, bag, hord

for ordered collections of elements it makes sense to have methods
get() ... which gets the "next"
getFirst()
getLast() ... and etc.
and to iterate through the elements, perhaps using
while (c.hasNext()) // for some collection c
    item = c.get();
but that could apply for both ordered and unordered collections
ADDING AND REMOVING OBJECTS FROM

A collection might be created with initial contents ... or empty

And add objects to a collection

add(<some type of> Item)

Or remove an object from a collection

remove(<some type of> Item)

Wherein there's the question of identity, and how to deal with multiple instances

remove() removes the `current' item

How do you deal with a set (like the change in your pocket?)

while (pocket.hasNext())

    coin = (Coin)pocket.get();

And what happens to the collection in the pocket?
ITERATION OVER (UNORDERED) SETS

a useful class of container would be one that holds elements in no particular order, and which allows you to access to them one-at-a-time, without removing them.

internally, the items are inevitably in some order.

simulating randomness through the enumeration process each element accessed once and only once different order of enumeration each time

how to interleave add/remove/enumeration of elements?
ArrayC, AN ORDERED COLLECTION

and let's define an ordered Collection, ArrayC

note that an array in Java already has an order but you have to do all the management:
  of manipulating indices
  of not overflowing the array
  of adding, removing, etc.
each time you need to use it!

we want to add ability to iterate down to the “end” of the collection “in order”

    start();
    while there is more to enumerate
        get next item

this too is a pattern
ArrayC, AN ORDERED COLLECTION

ArrayC is a “wrapper” around an array which encapsulates operations on an array.

It implements CollectionI

```java
    boolean add(Item i);
    Item get();
    void start();
    boolean step();
    boolean hasNext();
    boolean remove();
    void removeAll();
    void markPlace();
    void restartFromMarker();
    int getCount();
```

but the order of Items returned by get() reflects the underlying order of the array.

how to remove that predictability?
class ArrayC implements CollectionI {
    private Item[] items;
    private int currentIndex;
    private int count;
    public ArrayC() { /* <what to do???> */ }
    public ArrayC(int length) {
        items = new Item[length];
        count = 0;
        start();
    }
    public ArrayC(Item[] contents) {
        items = new Item[contents.length];
        count = contents.length;
        for (int i = 0; i < count; i++)
            items[i] = contents[i];
        start();
    }
    public void start() { currentIndex = 0; }
    public Item get() {
        if (currentIndex < count)
            return items[currentIndex++];
        else
            /* <what to do???> */
    }
    public boolean hasNext() { return (currentIndex < count); } ...
}
RandomC, AN UNORDERED COLLECTION

class RandomC implements CollectionI {

need to have get() return a randomly chosen Item
and the order of the internal array will be hidden from
the user
... computer science idea: randomness is difficult
to create from order

make a wrapper around Java's Random class (import
java.util.Random;) could be used in many applications:

class RandIndex {
    private Random r;

    public RandIndex() { r = new Random(); }

    public int get(int range) {
        return r.nextInt(range);
    }
}
class RandomC implements CollectionI {
    private RandIndex r = new RandIndex();
    private int top, count;
    private Item[] items;
    public RandomC(Item[] contents) {
        items = new Item[contents.length];
        count = items.length;
        for (int i = 0; i < count; i++)
            items[i] = contents[i];
    }
    public void start() { top = count - 1; }
    public Item get() {
        if (topIndex >= 0) {
            Item temp = items[top];
            int i = r.get(top);
            Item item = items[i];
            items[top] = item;
            items[i] = temp;
            top--;
            return item;
        } else
            <what to do???>
    }
    public boolean hasNext() { return (top >= 0); }
    ...
    and etc.
you are accustomed to iteration using "for" and "while" loops

    LB lb = new LB("Rubbermaid");
    while (bedroomFloor.hasNext())
        lb.add(floor.remove());

and:

    Pile whitePile = new Pile();
    Pile colorPile = new Pile();
    DL item;
    while (lb.hasNext()) {
        item = lb.remove();
        if (item instanceof WhiteL)
            whitePile.add(item);
        else
            colorPile.add(item);
    }

etc., we now need to look more broadly at the computational basis of iteration, which is a limited form of recursion
ITERATION

consider the following loops:

    for (int i = 0; i <= 0; i--)
        System.out.println("i = " + i);

and

    while (true) { ... }

these might be regarded as buggy, because they don't terminate
but some loops are useful even if you cannot predict when they will terminate:

    while (!session.over()) { ... }

or

    while (!customer.satisfied()) { ... }

we want to consider processes that achieve some goal by incrementally getting closer to completion ... after each iteration less remains to be done
DIVIDE AND CONQUER

take a problem
if problem immediately solvable
    solve problem;
else {
    divide into smaller problems;
    solve each subproblem;
}

Or

take a problem
if problem immediately solvable
    solve problem;
else {
    take off bite-sized subproblem;
    solve that subproblem;
    solve the remaining problem
}
EXAMPLES OF RECURSION

eating grapes?

while (bunch.moreAvailable()) {
    grape = bunch.getNext();
    eat(grape);
}

how to implement the getNext method? a bunch of grapes is either:
   a twig containing a single grape
or
   two (or more) sub-bunches

so starting at the top of the bunch of grapes,
   if bunch is a twig return attached grape (if not null)
else
   apply eatingGrapes to the next sub-bunch
FACTORIAL

if you haven't already see this... welcome to Computer Science!
factorial(n) = product of all numbers from 1 up to n

static int fact(int i) {
    int v = 1;
    for (int n = i; n > 1; n--)
        v *= n;
    return v;
}

... or:

static int fact(int i) {
    int v = 1;
    int j = 1;
    while (true) {
        v *= j++;
        if (j > i)
            break;
    }
    return v;
}
define
    factorial(n) = 1                case n = 0
    factorial(n) = n*factorial(n-1) otherwise

for example, if you have fact(4) to solve
    fact(4) = 4*fact(3)
which gets you closer to the base case:
    fact(4) = 4*fact(3)
    = 4*(3*fact(2))
    = 4*(3*(2*fact(1)))
    = 4*(3*(2*1))

static int fact(int i) {
    return ((i == 1) ? 1 : i*fact(i - 1));
}

    or ...
static int fact(int i) {
    if (i == 0)
        return 1;
    else
        return i*fact(i - 1);
}
how about computing \( x \) raised to the \( y \) power?

```java
class Test {
    static int power(int x, int y) {
        int p = 1;
        while (y-- > 0)
            p *= x;
        return p;
    }

    public static void main(String[] args) {
        for (int i = 0; i < 4; i++)
            System.out.println("2 to the " + i + " = " + power(2, i));
    }
}
```

does this do the job?

```
java Test
2 to the 0 = 1
2 to the 1 = 2
2 to the 2 = 4
2 to the 3 = 8
```
OTHER RECURSIVE PROGRAMS

and the recursive version

```java
class Test {
    static int power(int x, int y) {
        if (y == 0)
            return 1;
        else
            return x * power(x, y - 1);
    }

    public static void main (String[] args) {
        for (int i = 0; i < 4; i++)
            System.out.println("2 to the " + i + " = " + + power(2, i));
    }
}
```
A LIST CLASS
AN EXAMPLE OF A RECURSIVE DATA TYPE

provide a means to signify that a given object may be part of a list:

```java
interface Item {
}
```

provide a minimum set of requirements for any List implementation:

```java
interface ListI {
    abstract public int getLength();
    abstract public void setItem(Item c);
    abstract public void setNext(List l);
    abstract public Item getItem();
    abstract public List getNext();
}
```
IMPLEMENTATION OF LIST USING NULL (bleah)

by the usual definition:
a List is either: null (representing the empty list), or a "node" containing a reference to an Item plus a link to the rest of the List

class List implements ListI {
    Item item;
    List next;

    public List(Item c, List l) {
        setItem(c);
        setNext(l);
    }

    public int getLength() {
        return (next == null) ? 1 : next.getLength() + 1;
    }

    public void setItem(Item c) { item = c; }
    public void setNext(List l) { next = l; }
    public Item getItem() { return item; }
    public List getNext() { return next; }
}
here's how the Collection would be implemented using a private list hiding inside

class Collection implements CollectionI {
    private List list;
    private List current;
    public Collection() {
        list = null;
        current = list;
    }

    public Item get() {
        Item contents = current.getItem();
        current = current.getNext();
        return contents;
    }

    public boolean add(Item c) {
        list = new List(c, list);
        System.out.println("adding; count now " + getCount());
        return true;
    }

    public int getCount() { return list.getLength(); }
    public void start() { current = list; }
    public boolean more () { return (current instanceof List); }
}
and an application program using the collection

```
class Test {
    public static void main (String[] args) {
        Collection c = new Collection();
        System.out.println("loading ");
        for (int i = 0; i < 3; i++)
            c.add(new Thing(i));
        c.start();
        while (c.more()) {
            Thing t = (Thing)c.get();
            System.out.println("thing's value = " + t.getValue());
        }
    }
}
>>java Test
Loading
adding; count now 0
adding; count now 1
adding; count now 2
thing's value = 2
thing's value = 1
thing's value = 0
```

note the order in which the things are held in the collection ... the reverse of how they were added
AN OBJECT-ORIENTED IMPLEMENTATION OF LIST

using the same interface

interface ListI {
    abstract public int getLength();
    abstract public void setItem(Item c);
    abstract public void setNext(List l);
    abstract public Item getItem();
    abstract public List getNext();
}

let's make an abstract class to represent the recursive definition of a list

abstract class List implements ListI {}

and start the recursive definition of a list with a class representing the empty list, which has no length, whose value is null, and whose next is the empty list

class EmptyList extends List {
    public int getLength() { return 0; }
    public void setItem(Item c) {}
    public void setNext(List l) {}
    public Item getItem() { return null; }
    public List getNext() { return this; }
}
AN OBJECT-ORIENTED IMPLEMENTATION OF LIST

and now we complete the recursive definition of a list with a class representing the non-empty list, which is a Node which references an Item and a List

class Node extends List {
    Item item;
    List next;

    public Node(Item c, List l) {
        setItem(c);
        setNext(l);
    }

    public void setItem(Item c) { Item = c; }
    public void setNext(List l) { next = l; }
    public Item getItem()       { return Item; }
    public List getNext()       { return next; }
    public int  getLength() {
        return next.getLength() + 1;
    }
}
AN OBJECT-ORIENTED IMPLEMENTATION OF LIST

and here's how the Collection can be implemented using this OOD version of List

class Collection implements CollectionI {
    private List list;
    private List current;

    public Collection() {
        list      = new EmptyList();
        current   = list;
        start();
    }

    public Item getNext() {
        Item contents = current.getItem();
        current      = current.getNext();
        return contents;
    }

    public boolean add(Item c) {
        list      = new Node(c, list);
        return true;
    }

    public int     getCount() { return list.getLength(); }
    public void    start()    { current = list; }
    public boolean more ()    { return (current instanceof Node); }
}
TOWARDS AN INDUSTRIAL-STRENGTH LIST CLASS

interface Item {}

class EmptyListE extends Exception {}

interface ListI {
    abstract public void setItem(Item c) throws EmptyListE;
    abstract public void setNext(List l) throws EmptyListE;
    abstract public Item getItem() throws EmptyListE;
    abstract public List getNext() throws EmptyListE;
    abstract public int getLength();
}
abstract class List implements ListI {}

class EmptyList extends List {

    public void setItem(Item i) throws EmptyListE {
        throw new EmptyListE();
    }

    public void setNext(List l) throws EmptyListE {
        throw new EmptyListE();
    }

    public Item getItem() throws EmptyListE {
        throw new EmptyListE();
    }

    public List getNext() throws EmptyListE {
        throw new EmptyListE();
    }

    public int getLength() { return 0; }
}
TOWARDS AN INDUSTRIAL-STRENGTH LIST CLASS

class Node extends List {
    Item item;
    List next;

    public Node(Item i, List l) {
        setItem(i);
        setNext(l);
    }

    public void setItem(Item i) { item = i; }
    public void setNext(List l) { next = l; }

    public Item getItem() { return item; }
    public List getNext() { return next; }
    public int getLength() { return next.getLength() + 1; }
}
class CollectionE extends Exception {}

and a slightly modified interface for the collection class:

interface CollectionI {
    Item get() throws CollectionE;
    boolean add(Item c);
    int getCount();
    void start();
    boolean more();
}

we now use this new interface to implement ListC
LISTC, A COLLECTION CLASS BASED ON AN UNDERLYING (PRIVATE) LIST

class ListC implements CollectionI {
    private List list;
    private List previous;
    private List current;
    private int count;

    public ListC() {
        list = new EmptyList();
        count = 0;
        start();
    }

    public Item get() throws CollectionE {
        Item item;
        try {
            item = current.getItem();
        } catch (EmptyListE e) {
            throw new CollectionE();
        }
        return item;
    }
}
LISTC, A COLLECTION CLASS BASED ON AN UNDERLYING (PRIVATE) LIST

```java
public boolean add(Item c) {
    list = new Node(c, list);
    count++;
    return true;
}

public int getCount() { return count; }

public void start() {
    current = list;
    previous = list;
}

public boolean more() { return current instanceof Node; }

public boolean step() {
    try {
        current = current.getNext();
    }
    catch (EmptyListE e) { return false; }
    return true;
}
```
LISTC, A COLLECTION CLASS BASED ON AN UNDERLYING (PRIVATE) LIST

usage:

class Thing implements Item {
    private int value;
    public Thing(int x) { value = x; }
    public int getValue() { return value; }
}

class Test {
    public static void main (String[] args) {
        ListC c = new ListC();
        System.out.println("loading ");
        for (int i = 0; i < 4; i++)
            c.addItem(new Thing(i));
        c.start();
        try {
            while (c.more()) {
                Thing t = (Thing)c.get();
                System.out.println("value = " + t.getValue());
                c.step();
            }
        }
        catch (CollectionE e) { System.err.println("oops"); }
    }
}
ListC implements CollectionI

but CollectionI has no ability to remove an item yet

interface CollectionI {
    boolean add(Item i);
    Item    get() throws CollectionE;
    void    start();
    void    start();
    void    start();
    boolean step();
    boolean more();
    Item    remove() throws CollectionE;
    int     getCount();
}

CollectionI, being an interface, leaves unresolved many questions:
e.g. what would be the desired behavior of:
    removing multiple times?
    adding while enumerating?
    removing while enumerating?
    getting just after removing?
    stepping and removing then getting?
DESIGNING CONTAINER CLASSES IS NONTRIVIAL

moral: use the JAVA API's
it's not enough to design a good interface... the semantics (meanings) of the methods also critical

and using mutable recursive data types is also nontrivial

Containers();

List c1 = new List();
List c2;
while (itsSunnyOutside()) {
    c1 = c1.add(new Node(s.getRandomItem(), c1);
    c2 = c1;
} while (timeForAFewMore())
    c1 = c1.add(new Node(s.getRandomItem(), c1);
} c2.setNext(new List());

what happens to c1?
how can you avoid that and other problems?
much more OO Lists

how to add to the tail of a list?
  growing a list using the Decorator Pattern?
how to detect breaks? (Observer Pattern?)
how to detect cycles?
for ordered lists (Visitor Pattern?)
  how to add a new node at the correct position?
  make a list self-sorting?
  reverse order of list?
how to introduce the State Pattern (Empty or Node)?
Enumeration by Visitor?
doubly linked list