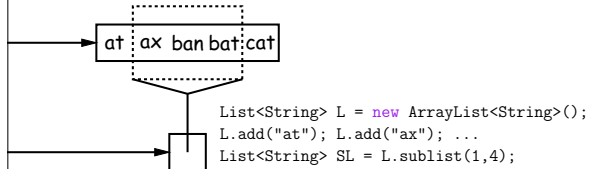


Views

A **view** is an alternative presentation of (interface to) a collection.

For example, the `sublist` method is supposed to yield a "view of" the original list:



After `L.set(2, "bag")`, value of `SL.get(1)` is "bag", and value of `L.get(2)` is "bad".

After `SL.clear()`, `L` will contain only "at" and "cat".

Question: "How do they do that?!"

Map Views

```
interface Map<Key,Value> { // Continuation
    ...
    Views of Maps */
    Set<Key> keySet(); // Set of all keys.
    Set<Value> values(); // Set of all values that can be returned by get.
    Set<Key,Value> entrySet(); // Set of all (key, value) pairs.
}
```

Simple Banking I: Accounts

Implement a simple banking system. Can look up accounts by name, deposit or withdraw, print.

```
Account {
    name: String, number: int, balance: double
}

Account(name, number, init) {
    name; this.number = number;
    balance = init;
}

Account(name, number) {
    name; number;
}

Account(name, number, balance) {
    name; number; balance;
}

Account(name, number, balance, interestRate) {
    name; number; balance; interestRate;
}

Account(name, number, balance, interestRate, ...) {
    name; number; balance; interestRate; ...
}
```

CS61B Lecture #18: Assorted Topics

Implementations

Tradeoffs

Sequences: stacks, queues, deques

Priority queues

Priority stacks

Maps

```
interface Map<Key,Value> {
    Value get(Key key); // Value at KEY.
    Set<Value> values(); // Set get(KEY) -> VALUE
}

Map<String,String> f = new TreeMap<String,String>();
f.put("George", "Martin");
f.put("John", "Paul");
f.get("George").equals("Martin");
f.get("John").equals("Paul");
f.get("Tom") == null
```

View Examples

From a previous slide:

```
Map<String,String> f = new TreeMap<String,String>();
f.put("George", "Martin");
f.put("John", "Paul");
```

Various views of f:

```
f.keySet().iterator().hasNext(); // true
f.keySet().iterator().next(); // Dana, George, Paul
f.keySet().iterator().next(); // John, Martin, George
f.values().iterator().hasNext(); // true
f.values().iterator().next(); // Martin, Paul, George
f.entrySet().iterator().hasNext(); // true
f.entrySet().iterator().next(); // (Dana,John), (George,Martin), (Paul,George)
f.get("Dana"); // null
```

Banks (continued): Iterating

Account Data

```
Print accounts sorted by number on STR. */
void print(PrintStream str) {
    // values() is the set of mapped-to values. Its
    // iterator produces elements in order of the corresponding keys.
    Iterator<Account> accounts = accounts.values().iterator();
    while (accounts.hasNext())
        print(str, accounts.next());
}
```

```
Print bank accounts sorted by name on STR. */
```

```
void print(PrintStream str) {
    Iterator<Account> accounts = names.values().iterator();
    while (accounts.hasNext())
        print(str, accounts.next());
}
```

Question: What would be an appropriate representation for the set of all transactions (deposits and withdrawals) against a given account?

5:18 2018

CS61B: Lecture #17 8

The java.util.AbstractList helper class

```
public abstract class AbstractList<Item> implements List<Item> {
    // inherited from List */
    public abstract int size();
    public abstract Item get(int k);
    public boolean contains(Object x) {
        for (int i = 0; i < size(); i += 1) {
            if (get(i) == null && x == null) ||
                (get(i) != null && x.equals(get(i)))
                return true;
        }
        return false;
    }
    // Throws exception; override to do more. */
    public void add(int k, Item x) {
        throw new UnsupportedOperationException();
    }
    // remove, set
}
```

5:18 2018

CS61B: Lecture #17 10

Example: Another way to do AListIterator

Example to make the nested class non-static:

```
public class AbstractList<Item> {
    private List<Item> list;
    private int where = 0;

    public Iterator<Item> iterator() { return new AListIterator(); }
    public Iterator<Item> listIterator() { return this.new AListIterator(); }

    private class AListIterator implements ListIterator<Item> {
        // position in our list. */
        private int where = 0;

        public boolean hasNext() { return where < list.size(); }
        public Item next() { where += 1; return list.get(where-1); }
        public void add(Item x) { list.add(where, x); where += 1; }
        // remove, set, etc.
    }
}
```

AbstractList.this means "the AbstractList I am attached to".
new AListIterator means "create a new AListIterator attached to X."

you can abbreviate this.new as new and can leave off AbstractList.this parts, since meaning is unambiguous.

5:18 2018

CS61B: Lecture #17 12

Simple Banking II: Banks

```
Accounts maintain mappings of String -> Account. They keep
keys (Strings) in "compareTo" order, and the set of
deposits (Amounts) is ordered according to the corresponding keys. */
Map<String, Account> accounts = new TreeMap<String, Account>();
Map<String, Account> names = new TreeMap<String, Account>();
```

```
Account(String name, int initBalance) {
    // choose a number
    int number = chooseNumber();
    add(acc.number, acc);
    add(name, acc);
}
```

```
Account(String number, int amount) {
    // get account
    Account acc = accounts.get(number);
    if (acc == null) ERROR(...);
    acc.deposit(amount);
}
```

```
Account withdraw(String name, int amount) {
    Account acc = accounts.get(name);
    if (acc == null) ERROR(...);
    acc.withdraw(amount);
}
```

5:18 2018

CS61B: Lecture #17 7

Partial Implementations

Interfaces (like List) and concrete types (like LinkedList),
provides abstract classes such as AbstractList.

One advantage of the fact that operations are related to
a single interface is that you can implement only a few of
the methods and still have a usable list.

For example, if you know how to do get(k) and size() for an imple-
mentation of List, you can implement all the other methods needed
for a List (and its iterators).

For example, if you know how to do add(k,x) and you have all you need for the additional
methods of a growable list.

For example, if you know how to do remove(k) and you can implement everything else.

5:18 2018

CS61B: Lecture #17 9

Example, continued: AListIterator

```
public abstract class AbstractList<Item> {
    private List<Item> list;
    private int where = 0;

    public Iterator<Item> iterator() { return new AListIterator(); }
    public Iterator<Item> listIterator() { return this.new AListIterator(); }

    private class AListIterator implements ListIterator<Item> {
        private List<Item> myList;

        AListIterator(AbstractList<Item> L) { myList = L; }

        // position in our list. */
        private int where = 0;

        public boolean hasNext() { return where < myList.size(); }
        public Item next() { where += 1; return myList.get(where-1); }
        public void add(Item x) { myList.add(where, x); where += 1; }
        // remove, set, etc.
    }
}
```

5:18 2018

CS61B: Lecture #17 11

Getting a View: Sublists

Sublist(start, end) is a List that gives a view of part of the original list. Changes in one must affect the other. How?

Implementation of class AbstractList. Error checks not shown.

```

Sublist(int start, int end) {
    this.start = start;
    this.end = end;
}

Sublist(AbstractList list, int start, int end) {
    this.list = list;
    this.start = start;
    this.end = end;
}

size() { return end - start; }
get(int k) { return list.get(start + k); }
add(int k, Item x) { list.add(start + k, x); end += 1; }

```

5:18 2018

CS61B: Lecture #17 14

Example: Using AbstractList

How to create a *reversed view* of an existing List (same size, reverse order). Operations on the original list affect the reversed view.

```

ReverseList<Item> extends AbstractList<Item> {
    List<Item> L;

    ReverseList(List<Item> L) { this.L = L; }

    size() { return L.size(); }
    get(int k) { return L.get(L.size() - k - 1); }
    add(int k, Item x) { L.add(L.size() - k, x); }
    set(int k, Item x) { return L.set(L.size() - k - 1, x); }
    remove(int k) { return L.remove(L.size() - k - 1); }
}

```

5:18 2018

CS61B: Lecture #17 13

Arrays and Links

How to represent a sequence: array and linked list. Comparison: ArrayList and Vector vs. LinkedList.

Arrays: compact, fast ($\Theta(1)$) random access (indexing). Operations: insertion, deletion can be slow ($\Theta(N)$)

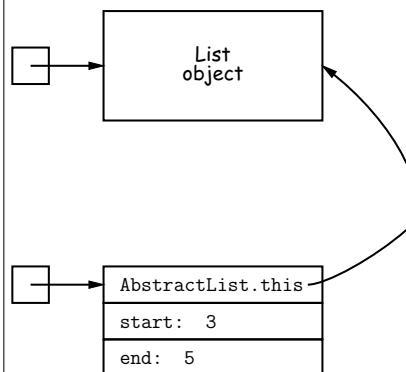
LinkedLists: insertion, deletion fast once position found. Operations: space (link overhead), random access slow.

5:18 2018

CS61B: Lecture #17 16

What Does a Sublist Look Like?

```
ReverseList L = L.sublist(3, 5);
```



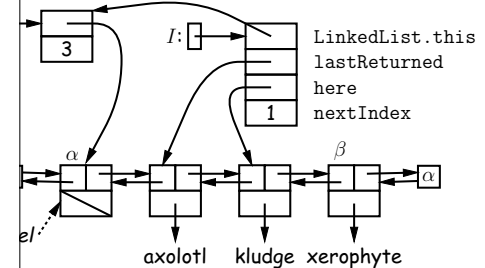
5:18 2018

CS61B: Lecture #17 15

Linking

Why linking should now be familiar

Implementation of a LinkedList. One possible representation for linked list iterator object over it:



```

LinkedList L = new LinkedList<>();
L.add("axolotl");
L.add("kludge");
L.add("xerophyte");

LinkedListIterator I = L.listIterator();
I.next();

```

5:18 2018

CS61B: Lecture #17 18

Implementing with Arrays

Problem with implementing with arrays is insertion/deletion in the middle of a long list (moving a lot of things over).

Operations from ends can be made fast:

Array size to grow; amortized cost constant (Lecture #15).

Operations at one end really easy; classical stack implementation:

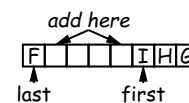
```

push("A");
push("B");
push("C");

```

The diagram shows a stack implemented with an array. The array contains the elements 'A', 'B', and 'C'. The size of the stack is 3. An arrow points to the end of the array with the label 'add here', indicating where new elements are added.

Operations at both ends, use *circular buffering*:



Access still fast.

5:18 2018

CS61B: Lecture #17 17

Specialization

Special cases of general list:

add and delete from one end (LIFO).

add at end, delete from front (FIFO).

Add or delete at either end.

Not easily representable by either array (with circular buffer or deque) or linked list.

Use List types, which can act like any of these (although with additional names for some of the operations).

java.util.Stack, a subtype of List, which gives traditional ("push", "pop") to its operations. There is, however, no face.

Stacks and Recursion

Used to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance gain).

Remember "push current variables and parameters, set parameter values, and loop."

Remember "pop to restore variables and parameters."

```

    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            if isExit(start)
                FOUND
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    )
    mb(start))
    t start;
    re, x,
    start:
    start,x) && !isCrumb(x)
    t(x)
    )

```



Stacks and Recursion

Used to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance gain).

Remember "push current variables and parameters, set parameter values, and loop."

Remember "pop to restore variables and parameters."

```

    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            if isExit(start)
                FOUND
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    )
    mb(start))
    t start;
    re, x,
    start:
    start,x) && !isCrumb(x)
    t(x)
    )

```



Clever trick: Sentinels

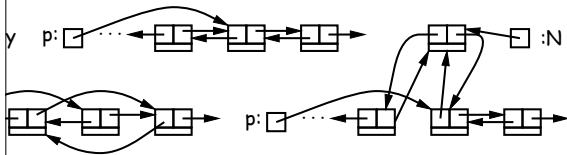
Use a dummy object containing no useful data except links. This eliminates special cases and to provide a fixed object to refer to access a data structure.

Use special cases ('if' statements) by ensuring that the first and last nodes in a list always have (non-null) nodes—possibly sentinels—before and after them:

```

// List node at p: // To add new node N before p:
p.prev;           N.prev = p.prev; N.next = p;
p.next;           p.prev.next = N;
                  p.prev = N;

```



Stacks and Recursion

Used to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance gain).

Remember "push current variables and parameters, set parameter values, and loop."

Remember "pop to restore variables and parameters."

```

    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            if isExit(start)
                FOUND
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    )
    mb(start))
    t start;
    re, x,
    start:
    start,x) && !isCrumb(x)
    t(x)
    )

```



Stacks and Recursion

Used to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance gain).

Remember "push current variables and parameters, set parameter values, and loop."

Remember "pop to restore variables and parameters."

```

    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            if isExit(start)
                FOUND
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    )
    mb(start))
    t start;
    re, x,
    start:
    start,x) && !isCrumb(x)
    t(x)
    )

```



Stacks and Recursion

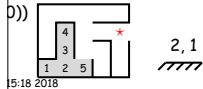
ed to recursion. In fact, can convert any recursive al-
 tack-based (however, generally no great performance

me "push current variables and parameters, set param-
 ew values, and loop."

comes "pop to restore variables and parameters."

```

    )
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
            start:
            if isExit(start)
                FOUND
            start,x) && !isCrumb(x)
            t(x)
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    
```



5:18 2018 CS61B: Lecture #17 26

Stacks and Recursion

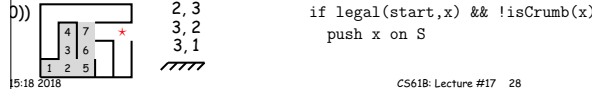
ed to recursion. In fact, can convert any recursive al-
 tack-based (however, generally no great performance

me "push current variables and parameters, set param-
 ew values, and loop."

comes "pop to restore variables and parameters."

```

    )
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
            start:
            if isExit(start)
                FOUND
            start,x) && !isCrumb(x)
            t(x)
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    
```



5:18 2018 CS61B: Lecture #17 28

Stacks and Recursion

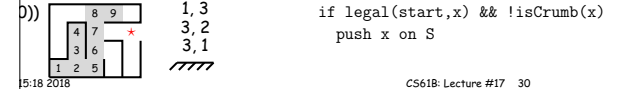
ed to recursion. In fact, can convert any recursive al-
 tack-based (however, generally no great performance

me "push current variables and parameters, set param-
 ew values, and loop."

comes "pop to restore variables and parameters."

```

    )
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
            start:
            if isExit(start)
                FOUND
            start,x) && !isCrumb(x)
            t(x)
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    
```



5:18 2018 CS61B: Lecture #17 30

Stacks and Recursion

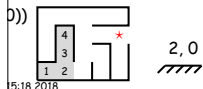
ed to recursion. In fact, can convert any recursive al-
 tack-based (however, generally no great performance

me "push current variables and parameters, set param-
 ew values, and loop."

comes "pop to restore variables and parameters."

```

    )
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
            start:
            if isExit(start)
                FOUND
            start,x) && !isCrumb(x)
            t(x)
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    
```



5:18 2018 CS61B: Lecture #17 25

Stacks and Recursion

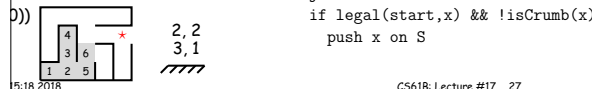
ed to recursion. In fact, can convert any recursive al-
 tack-based (however, generally no great performance

me "push current variables and parameters, set param-
 ew values, and loop."

comes "pop to restore variables and parameters."

```

    )
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
            start:
            if isExit(start)
                FOUND
            start,x) && !isCrumb(x)
            t(x)
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    
```



5:18 2018 CS61B: Lecture #17 27

Stacks and Recursion

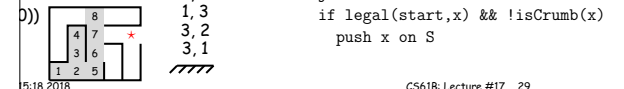
ed to recursion. In fact, can convert any recursive al-
 tack-based (however, generally no great performance

me "push current variables and parameters, set param-
 ew values, and loop."

comes "pop to restore variables and parameters."

```

    )
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
            start:
            if isExit(start)
                FOUND
            start,x) && !isCrumb(x)
            t(x)
            else if (!isCrumb(start))
                leave crumb at start;
                for each square, x,
                    adjacent to start (in reverse):
                        if legal(start,x) && !isCrumb(x)
                            push x on S
    
```



5:18 2018 CS61B: Lecture #17 29

Stacks and Recursion

ed to recursion. In fact, can convert any recursive al-
tack-based (however, generally no great performance

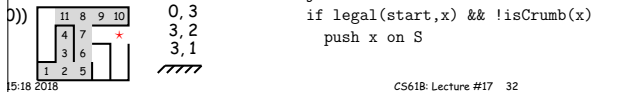
me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```

)
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
                if isExit(start)
                    start:
                    start,x) && !isCrumb(x)
                    t(x)
                FOUND
                else if (!isCrumb(start))
                    leave crumb at start;
                    for each square, x,
                        adjacent to start (in reverse):
                            if legal(start,x) && !isCrumb(x)
                                push x on S

```



5:18 2018

CS61B: Lecture #17 32

Stacks and Recursion

ed to recursion. In fact, can convert any recursive al-
tack-based (however, generally no great performance

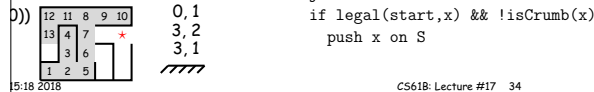
me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```

)
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
                if isExit(start)
                    start:
                    start,x) && !isCrumb(x)
                    t(x)
                FOUND
                else if (!isCrumb(start))
                    leave crumb at start;
                    for each square, x,
                        adjacent to start (in reverse):
                            if legal(start,x) && !isCrumb(x)
                                push x on S

```



5:18 2018

CS61B: Lecture #17 34

Stacks and Recursion

ed to recursion. In fact, can convert any recursive al-
tack-based (however, generally no great performance

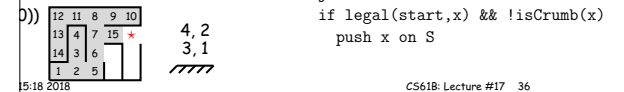
me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```

)
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
                if isExit(start)
                    start:
                    start,x) && !isCrumb(x)
                    t(x)
                FOUND
                else if (!isCrumb(start))
                    leave crumb at start;
                    for each square, x,
                        adjacent to start (in reverse):
                            if legal(start,x) && !isCrumb(x)
                                push x on S

```



5:18 2018

CS61B: Lecture #17 36

Stacks and Recursion

ed to recursion. In fact, can convert any recursive al-
tack-based (however, generally no great performance

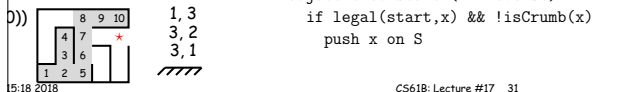
me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```

)
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
                if isExit(start)
                    start:
                    start,x) && !isCrumb(x)
                    t(x)
                FOUND
                else if (!isCrumb(start))
                    leave crumb at start;
                    for each square, x,
                        adjacent to start (in reverse):
                            if legal(start,x) && !isCrumb(x)
                                push x on S

```



5:18 2018

CS61B: Lecture #17 31

Stacks and Recursion

ed to recursion. In fact, can convert any recursive al-
tack-based (however, generally no great performance

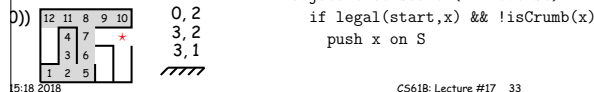
me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```

)
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
                if isExit(start)
                    start:
                    start,x) && !isCrumb(x)
                    t(x)
                FOUND
                else if (!isCrumb(start))
                    leave crumb at start;
                    for each square, x,
                        adjacent to start (in reverse):
                            if legal(start,x) && !isCrumb(x)
                                push x on S

```



5:18 2018

CS61B: Lecture #17 33

Stacks and Recursion

ed to recursion. In fact, can convert any recursive al-
tack-based (however, generally no great performance

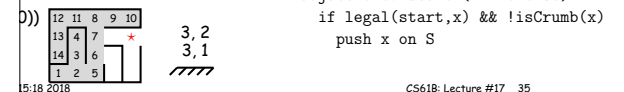
me "push current variables and parameters, set param-
ew values, and loop."

comes "pop to restore variables and parameters."

```

)
    findExit(start):
        S = new empty stack;
        push start on S;
        while S not empty:
            pop S into start;
            re, x,
                if isExit(start)
                    start:
                    start,x) && !isCrumb(x)
                    t(x)
                FOUND
                else if (!isCrumb(start))
                    leave crumb at start;
                    for each square, x,
                        adjacent to start (in reverse):
                            if legal(start,x) && !isCrumb(x)
                                push x on S

```



5:18 2018

CS61B: Lecture #17 35

Topics: Extension, Delegation, Adaptation

Define java.util.Stack type *extends* Vector:

```
> extends Vector<Item> { void push(Item x) { add(x); } ... }
```

Define have *delegated* to a field:

```
Stack<Item> {  
    ArrayList<Item> repl = new ArrayList<Item>();  
    void push(Item x) { repl.add(x); } ...  
}
```

Generalize, and define an *adapter*: a class used to make one kind behave as another:

```
StackAdapter<Item> {  
    Stack<Item> repl;  
    StackAdapter(List<Item> repl) { this.repl = repl; }  
    void push(Item x) { repl.add(x); } ...  
}
```

```
Stack<Item> extends StackAdapter<Item> {  
    Stack() { super(new ArrayList<Item>()); }  
}
```

Stacks and Recursion

Applied to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance improvement)

Push: "push current variables and parameters, set parameter values, and loop."

Pop: "pop to restore variables and parameters."

```
) findExit(start):  
    S = new empty stack;  
    push start on S;  
    while S not empty:  
        pop S into start;  
        if isExit(start)  
            return FOUND  
        else if (!isCrumb(start))  
            leave crumb at start;  
        for each square, x,  
            adjacent to start (in reverse):  
                if legal(start,x) && !isCrumb(x)  
                    push x on S  
mb(start)  
return start;  
return x,  
start:  
start,x) && !isCrumb(x)  
return x)  
))
```

12	11	8	9	10
13	4	7	15	*
14	3	6		
1	2	5		

3,1

