6.005 Software Construction
Fall 2011
Prof. Rob Miller

L4: State Machines

Today

- Mutable objects
- Aliasing
- State machines

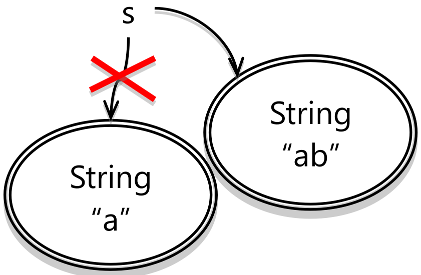
Required reading (from the Java Tutorial)

- Classes and Objects, including:
  - Classes
  - Objects
  - More on Classes
  - Nested Classes
  - Enum Types
- the static keyword
- the final keyword

Mutable objects

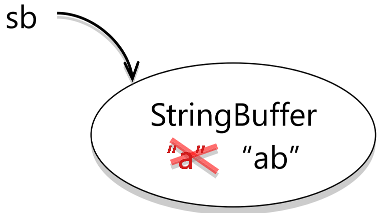
String is immutable: once created, a String object always has the same value. To add something to the end of a String, you have to create a new String object

```
String s = "a";
s = s.concat("b");    /// s = s + "b" is syntactic sugar for this
```



StringBuilder (another builtin Java class) is a **mutable** object that represents a string of characters. It has operations that change the value of the object, rather than just returning new values:

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def");
```

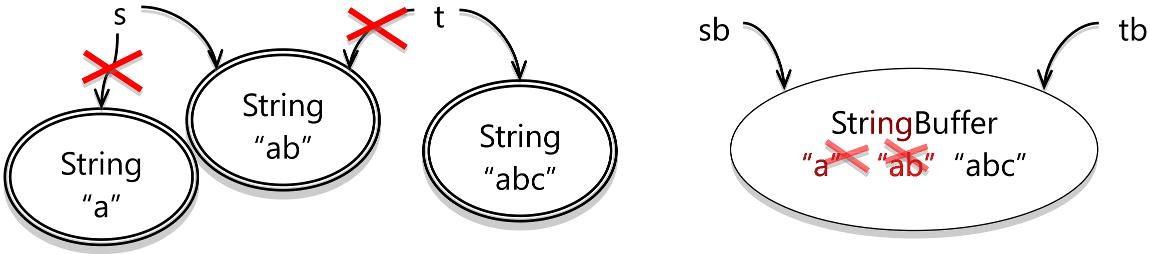


These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors.

StringBuilder has other **mutator** operations as well, for deleting parts of the string, inserting in the middle, or changing individual characters.

So what? In both cases, you end up with s and sb referring to the string of characters abcdef. The difference between mutability and immutability doesn't matter much when there's only one reference to the object. But there are big differences in how they behave when there are *other* references to the object, like t and tb introduced below:

```
String t = s;
t = t + "c";

StringBuilder tb = sb;
tb.append("c");
```



Why do we need the mutable StringBuilder in programming? A common use for it is to concatenate a large number of strings together, like this:

```
String s = "";
for (int i = 0; i < n; ++i) {
       s = s + n;
}
```

Using immutable Strings, this makes a lot of temporary copies – the first number of the string ("0") is actually copied *n* times in the course of building up the final string, the second number is copied *n-1* times, and so on. It actually costs O(n^2) time just to do all that copying, even though we only concatenated *n* elements.

StringBuilder is designed to minimize this copying. It uses a simple but clever internal data structure to avoid doing any copying at all until the very end, when you ask for the final String:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; ++i) {
       sb.append(String.valueOf(n));
}
String s = sb.toString();
```

Getting good performance is one reason why we use mutable objects. Another is convenient sharing: two parts of your program can communicate more conveniently by sharing a common mutable data structure.

But using mutable data comes with big risks – it becomes harder to understand what your program is doing, and much harder to enforce contracts. We'll look at an example of that next.

Iterating over collections

The next mutable object we're going to look at is an iterator – an object that steps through a collection of elements and returns the elements one by one. Iterators are used under the covers in Java when you're using a for loop to step through a List or array. This code:

```
List<String> l = ...;
for (String s : l) {
       System.out.println(s);
```

```
        }
```

actually unpacks into something like this:

```
        List<String> l = ...;
        Iterator iter = l.iterator();
        while (l.hasNext()) {
                String s = iter.next();
                System.out.println(s);
        }
```

An iterator has two methods: next() returns the next element in the collection; and hasNext() tests whether the iterator has reached the end of the collection. Note that the next() method is a **mutator**, not only returning an element but also advancing the iterator so that the subsequent call to next() will return a different element.

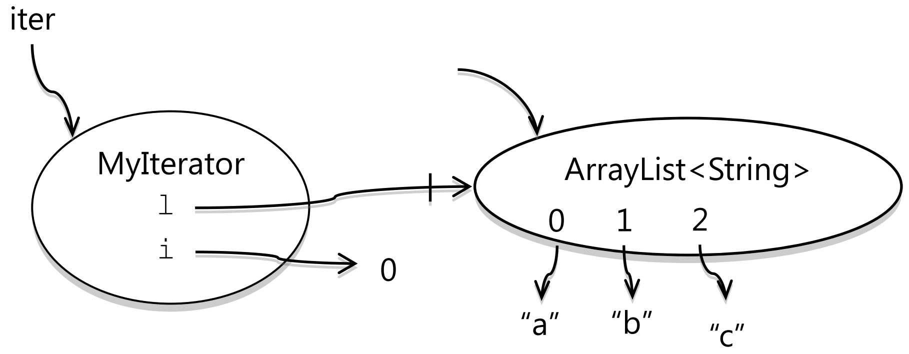
To better understand how an iterator works, here's a simple implementation of an iterator for ArrayList<String>:

```
/**
 * A MyIterator is a mutable object that iterates over
 * the elements of an ArrayList<String>, from first to last.
 * This is just an example to show how an iterator works.
 * In practice, you should use the ArrayList's own iterator object,
 * returned by its iterator() method.
 */
public class MyIterator {

    private final ArrayList<String> l;
    private int i;
    // l[i] is the next element that will be returned by next();
    // i == l.size() means no more elements to return

    /**
     * Make an iterator.
     * @param l list to iterate over
     */
    public MyIterator(ArrayList<String> l) {
        this.l = l;
        this.i = 0;
    }

    /**
     * Test whether the iterator has more elements to return.
     * @return true if next() will return another element,
     *         false if all elements have been returned.
     */
    public boolean hasNext() {
        return i < l.size();
    }

    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String s = l.get(i);
        ++i;
        return s;
        }
}
```

MyIterator makes use of a few Java language features that are different from the classes we've been writing to this point. Make sure you read the Java Tutorial sections required for this lecture so that you understand them:

- *instance variables*, also called fields in Java. Instance variables differ from method parameters and local variables; the instance variables are stored in the object instance and persist for longer than a method call. What are the instance variables of MyIterator?
- a *constructor*, which makes a new object instance and initializes its instance variable. Where is the constructor of MyIterator?
- the *static* keyword is missing from MyIterator's methods, which means they are instance methods that must be called on an instance of the object, e.g. iter.next().
- the *this* keyword is used at one point to refer to the instance object, in particular to refer to an instance variable (*this.l*). This was done to disambiguate two different variables named *l* (an instance variable and a constructor parameter). Most of MyIterator's code refers to instance variables without an explicit *this*, but this is just a convenient shorthand that Java supports -- e.g., *i* actually means *this.i.*
- *private* is used for the object's internal state and internal helper methods, while *public* indicates methods and constructors that are intended for clients of the class.
- *final* is used to indicate which parts of the object's internal state can change and which can't. *i* is allowed to change (next() updates it as it steps through the list), but *l* cannot (the iterator has to keep pointing at the same list for its entire life – if you want to iterate through another list, you're expected to create another iterator object).

Here's a snapshot diagram showing a typical state for a MyIterator object in action:



Note that we drew a slash across the arrow from *l*, to indicate that it's *final.* That means that the arrow can't change once it's drawn. But the ArrayList object it points to is mutable – elements can be changed within it -- and declaring *l* as final has no effect on that.

Why do iterators exist? There are many kinds of collection data structures (linked lists, maps, hash tables) with different kinds of internal representations. The iterator concept allows a single uniform way to access them all, so that client code is simpler and the collection implementation can change without changing the client code. Most modern languages (including Python, C#, and Ruby) use the notion of an iterator. It's an effective **design pattern** (a well-tested solution to a common design problem). We'll see many other design patterns as we move through the course.

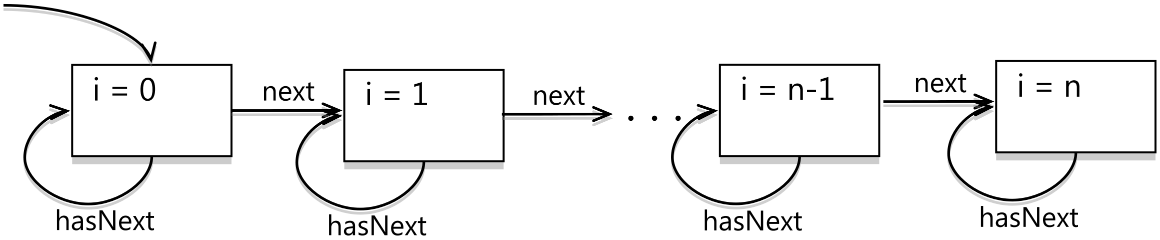
Regarding a mutable object as a state machine

One useful perspective for regarding mutable is as a *state machine*. A state machine is a set of *states* that a system can be in (drawn as boxes) with the possible transitions between them (drawn as edges). The transitions are called *events*.

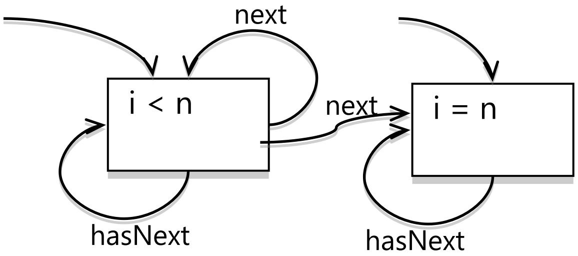
For a mutable object, the state is represented by the **instance variables** of the object – particularly the ones that can be mutated over the course of the object's lifetime. The state of a MyIterator object is represented by *i*, the index of its current position in the list.

The events are the public operations that can be performed on the object -- the methods that can be called. These represent events arriving from the outside world (from a client using the object) that cause transitions in the object's internal state. For a MyIterator, the methods are hasNext and next. Of these two methods, next is a **mutator**, however, so it can cause transitions to a new state. hasNext is an **observer** – it returns information about the current state, but never causes a transition to a new state.

Here's a very explicit state machine diagram for a MyIterator object over a list of length *n* (*n > 0*), showing all the possible states that the object can experience:



In practice we will find it more useful to draw more abstract state diagrams that combine multiple states together:



Here we've combined all the states where i<n into a single state, and combined their transitions as well. Note that we now have a *nondeterministic* state machine – the *next* event (method call) can transition to either i<n (if we still have more elements to return) or to i=n (if we've run to the end of the list). In this case, the nondeterminism was introduced by our decision to abstract away from the particular value of i. The behavior of the MyIterator object is still completely deterministic; it knows which particular value of i it has. We've just omitted it from this picture. Other state machines might be genuinely nondeterministic, in the sense that the transition chosen in response to an event might be random.

This state machine also correctly handles the case where *n* (the length of the list is zero), by having two initial-state transitions. Either i<n or i=n may be the starting state of the machine.

The risk of mutation

Let’s try using our iterator for a simple job. Suppose we have a list of tokens extracted from a web page, so the tokens include both words (“hello” and “world”) and HTML tags (“<a>”, “<br>”, etc.). We want a method *stripTags* that will delete the HTML tags from the list, leaving the words behind. Following good practices, we first write the spec:

```
/**
  * Remove html tags from a list.
  * Modifies l by removing elements of the form "<*>".
  * @param l list of words and html tags.
  */
 public static void stripTags(ArrayList<String> l) {...}
```

Note that stripTags has a frame condition (*modifies* clause) in its contract, warning the client that its list argument will be mutated.

Next, following test-first programming, we devise a testing strategy that partitions the input space, and choose test cases to cover that partition:

```
// Testing strategy:
//   l.size: 0, 1, n
//   contents: no tags, one tag, all tags
//   position: tag at start, tag in middle, tag at end
//   kind of element: <foo>, </foo>, word, empty string

   // Test cases:
   // [] => []
   // [“a”] => [“a”]
   // ["a", “b”, “c”] => ["a", “b”, “c”]
   // [“a”, “<b>”, “c”] => [“a”, “c”]
   // [“<a>”, “<b>”, “<c>”] => []
```

Finally we implement it:

```
public static void stripTags(ArrayList<String> l) {
    MyIterator iter = new MyIterator(l);
    while (iter.hasNext()) {
        String s = iter.next();
        if (isTag(s)) {
            l.remove(s);
        }
    }
}

// returns true iff s is an html tag of the form "<*>" for any *
private static boolean isTag(String s) {
    return s.startsWith("<") && s.endsWith(">");
}
```

(Note that we pulled out the test for whether a token is HTML into a separate method, isTag. This improves the readability of stripTags, and allows us to make isTag more complicated if necessary, since HTML can be very complicated, and test it independently of stripTags.)

Now we run our test cases, and they work! ... almost. The last test case fails:

```
// stripTags([“<a>”, “<b>”, “<c>”])
//    expected [], actual [“<b>”]
```

We got the wrong answer: stripTags left a tag behind in the list. Why? Trace through what happens. It will help to use a snapshot diagram showing the MyIterator object and the ArrayList object and update it while you work through the code.

Note that this isn't just a bug in our MyIterator. The built-in iterator in ArrayList suffers from the same problem, and so does the for() loop that's syntactic sugar for it. The problem just has a different symptom. If you used this code instead:

```
for (String s : l) {
    if (isTag(s)) {
        l.remove(s);
    }
}
```

then you'll get a ConcurrentModificationException. The builtin iterator detects that you're changing the list under its feet, and cries foul. (How do you think it does that?)

How can you fix this problem? One way is to use the remove() method of Iterator, so that the iterator adjusts its index appropriately:

```
Iterator iter = l.iterator();
while (iter.hasNext()) {
    String s = iter.next();
    if (isTag(s)) {
        iter.remove(s);
    }
}
```

(This is actually more efficient as well, it turns out, because iter.remove() already knows where the element it should remove is, while l.remove() had to search for it again.)

But this doesn't fix the whole problem. What if there are *other* Iterators currently active over the same list? They won't all be informed!

Aliasing

This is a fundamental issue with mutable data structures. Multiple references to the same mutable object (also called *aliases* for the object) may mean that multiple places in your program – possibly widely separated – are relying on that object to remain consistent.

To put it in terms of specifications, contracts can't be enforced in just one place anymore, e.g. between the client of a class and the implementer of a class. Contracts involving mutable objects now depend on the good behavior of everyone who has a reference to the mutable object. (As a symptom of this non-local contract phenomenon, consider the Java collections classes, which are normally documented with very clear contracts on the client and implementer of a class. Try to find where it documents this particular requirement on the client – that you can't modify a collection while you're iterating over it. Who takes responsibility for it? Iterator? List? Collection?)

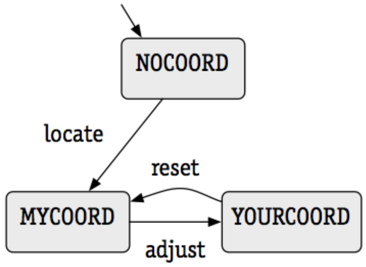
The need to reason about global properties like this make it much harder to understand, and be confident in the correctness of, programs with mutable data structures. We still have to do it – for performance and convenience – but we pay an big cost in bug safety for doing so.

Using state machines for analysis and design

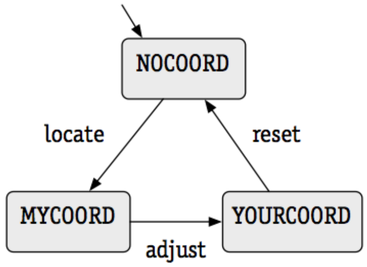
State machine notation gives us a way to think about the behavior of a class, module, or entire system, so that we can understand it, communicate it, and analyze it.

Here's an example. In Afghanistan, December 2001: a US soldier uses a "plugger" (PLGR: precision lightweight global-positioning satellite receiver) to mark a Taliban position for an air-strike. The normal operation of this device starts with a **locate** operation that determins the user's own coordinates (from GPS satellites), and then an **adjust** operation that adjusts by a distance and direction (obtained by sighting from the user's position to the target) to obtain the target's coordinates. In this incident, the soldier noticed a battery-low warning, so he replaced the battery, and then radioed in the coordinates he read from the screen. The resulting airstrike hit his own position, killing him and two comrades and wounding 20 others.

What happened? Replacing the battery reset the device to displaying the *user's* position, forgetting the adjustment to the target. The state machine for the device's display looked like this:



whereas a much safer user interface would have done this instead:



Vernon Loeb. "Friendly Fire Deaths Traced to Dead Battery; Taliban Targeted, but U.S. Forces Killed." *Washington Post.* March 24, 2002

State machines are a simple notation for describing behaviors which is succinct and abstract – but not vague! We can use it as a basis for analysis and description of problems like these, and also for implementation. We'll see later in this lecture how state machines are useful for choosing test cases.

An example: a MIDI piano

State machines are often used for input and output. Java's InputStream and OutputStream objects are simple state machines that read and write streams of characters (such as files or network connections).

Let's look at two other kinds of input and output: your keyboard, and a MIDI music synthesizer.

A single keyboard key has two states

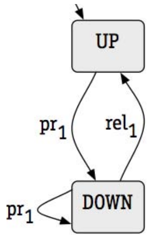
UP: the key is up (initial state)

DOWN: the key is held down

There are two kinds of input events, with these designations:

pr: the keyboard driver reports a key press

rel: the keyboard driver reports a release



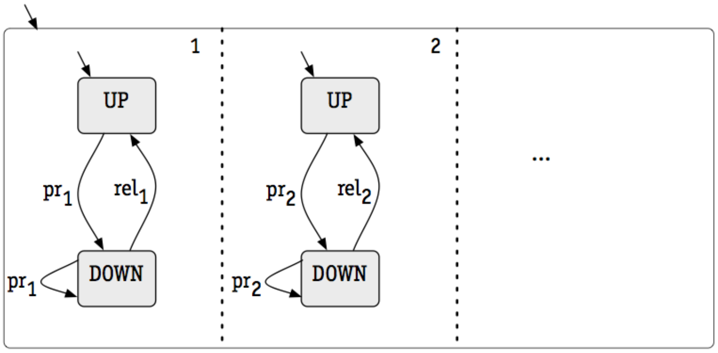
(This keyboard driver reports multiple press events while the key is held down, so we have a pr self-transition in the DOWN state.)

The meaning of a state machine is the **set of traces** that it accepts, where a trace is a sequence of events. The traces of this machine are:

<> (empty trace)
<pr>
<pr, rel>
<pr, pr, rel>

...

An entire keyboard can be represented as a parallel combination of state machines, which we denote graphically as follows:



Each key's machine can take steps independently. As a general rule, **shared** events (transitions with the same label) must be synchronized, but there is no sharing here, because each state machine's events are labeled by the key (1, 2, ...).

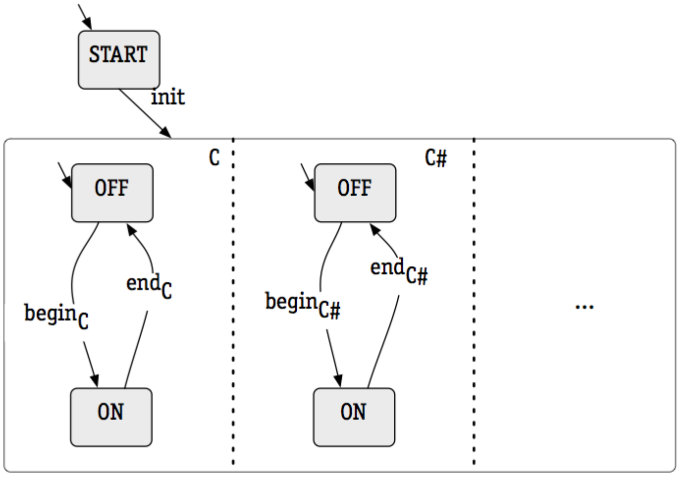
The traces of this machine include:

<>
<pr1>
<pr1, rel1>
<pr1, rel1, pr>
<pr1, pr2, pr1>

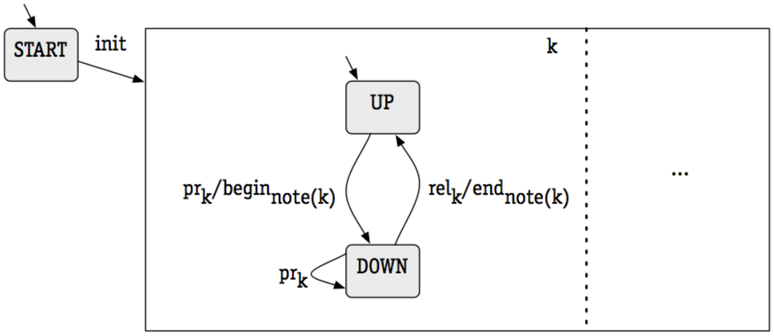
...

This model is actually an *over-approximation* -- it allows more traces than can really happen to a keyboard. One such trace is <pr1, pr2, pr1> -- try it and see. But this is OK; a design that handles cases that can never arise isn't a problem; it's a design that fails to handle cases that's a problem.

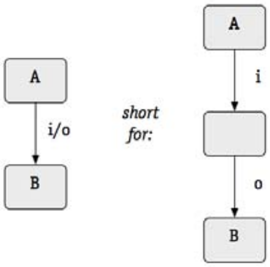
We've looked at the input side of the example. Now let's turn to the output side: the MIDI synthesizer. Unlike a physical keyboard, this is a software system that needs explicit intialization, so we indicate that with an *init* event. Then, we just have a set of parallel state machines, one for each note the synthesizer can play, and events to turn them on or off:



Finally, let's connect them together. We'll draw another state machine that connects the keyboard to the MIDI device:



This machine is using a shorthand on its edges that takes an input event (such as pr$_k$) and emits an output event (begin$_{note(k)}$). This is just a shorthand for two edges with an intervening state, i.e.:



There's a subtlety here, in that the output *o* need not follow the input *i* immediately. Another event might intervene. This will become a critical issue when we design concurrently-running state machines, which is a topic for a future lecture.

Our final picture shows how presses and releases of the keyboard are related to starting and stopping notes in the MIDI synthesizer, as well as the detail that additional presses in the DOWN state should be ignored. It can be translated more or less directly to code.

State machine semantics

Formally, a state machine consists of:

- a set of states State
    - e.g. State = { UP, DOWN, PRESSED, RELEASED }
- a set of initial states Init
    - e.g. Init = { UP }
- a set of events Event
    - e.g Event = { pr, rel, begin, end }
- a transition relation *trans* ⊆ State × Event × State
    - e.g. *trans* ⊆ State × Event × State
      = { (UP, pr, PRESSED), (PRESSED, begin, DOWN), (DOWN, pr, DOWN), (DOWN, rel, RELEASED), (RELEASED, end, UP) }
- a trace set *traces* (derived from trans and Init)
    - e.g. traces = { <>, <pr>, <pr, begin>, <pr, begin, pr>, <pr, begin, rel>, ...}

Common misconceptions about state machine modeling

A state machine is not a flow chart. There is no behavior or logic inside a state. It doesn't make sense to write a label like "Is the key pressed?" inside a state.

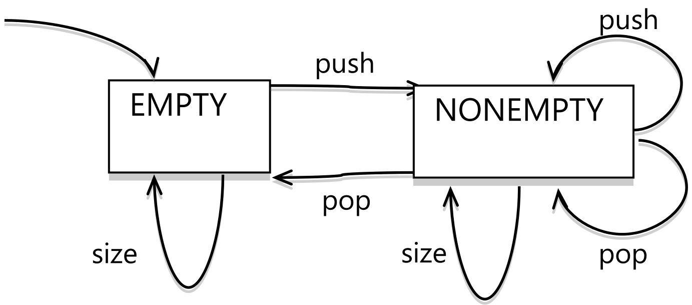
A state machine has no "decision edges," either. Some transitions in the state machine may be nondeterministic (such as the *next* transitions in the Iterator state machine above), and logic in the system may determine which transition to take, but that logic is not expressed in the state machine diagram.

Change doesn't happen within a state. Think of it like a snapshot of the world. Change happens only on the transitions.

Another example: a stack

Let's develop another mutable class and model it as a state machine. A *stack* is a collection of elements that can only be changed at one end: you can *push* an element on top of the stack, and *pop* an element from the top of the stack, but you can't insert or delete elements in the middle or bottom of the stack. Think of a stack like a stack of dinner plates at a buffet. Stacks are useful for keeping track of work in recursive algorithms (in fact, method calling in most programming languages uses a stack to keep track of the methods that are waiting for results from methods they've called). Stacks are also used in "reverse Polish" calculators, like those made by HP.

Let's draw the state machine for a stack. We'll have two states, and three events corresponding to the *push* and *pop* mutators, plus a *size* method that doesn't change state:



Recall that this is an abstraction for more detailed states: NONEMPTY represents a lot of different stacks with different sizes and different contents.

Note also that we've made a design choice here that pop events are not legal in the empty state.

Again we follow spec-first, test-next, implement-last pattern. Here's our spec:

```
/**
 * A Stack is a mutable object representing a last-in-first-out
 * stack of elements (of an arbitrary type E).
 * Elements can be pushed onto the stack, and then popped off
 * in the reverse order that they were pushed.
 * A Stack can hold an arbitrary number of elements.
 */
public class Stack<E> {

    /**
     * Make a Stack, initially empty.
     */
    public Stack() {...}

    /**
     * Modifies this stack by pushing an element onto it.
     * @param e element to push on top
     */
    public void push(E e) {...}

    /**
     * Modifies this stack by popping off the top element.
     * Requires: stack is not empty, i.e. size() > 0.
     * @return element on top of stack
     */
    public E pop() {...}

    /**
     * @return number of elements in the stack
     */
    public int size() {...}
}
```

We'll choose an internal state for this stack, and decide how it maps to the Empty and Nonempty states in our state machine:

```
private final List<E> elems = new ArrayList<E>();
// elems contains the elements in the stack,
// in order from oldest pushed (elems[0]) to
//          to the latest item pushed, and the
//           next to be popped (elems[size-1]).
// If elems.size == 0, then the stack is empty.
```

Can you finish the implementation of Stack, by writing the constructor, push, pop, and size methods?

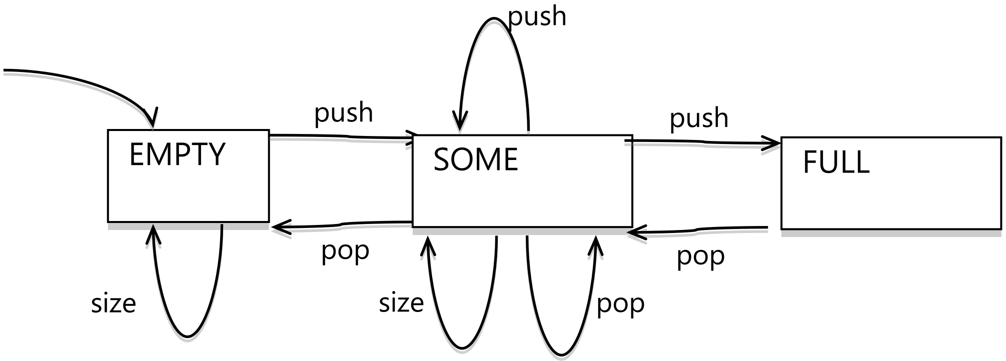
A fixed-size stack

Our stack is flexible, but if we want a truly high-performance stack, it might not be suitable. The ArrayList implementation copies the stack when it gets too big, in order to make room for more elements.

Suppose we know there's a maximum size our stacks will grow, and we're willing to live with that limitation in exchange for fast performance. Then we can use a fixed-size array to implement the stack:

```
private final E[] elems;
private int n;
// elems[] contains the elements in the stack.
// elems[0] is the oldest item pushed;
// elems[n-1] is the latest item pushed, and the
//            next to be popped.
// If n == 0, then the stack is empty.
// If n == elems.length, then the stack is full.
```

The state machine for FixedStack looks like this:



Can you modify the specs for the push, pop, and size methods to match this new design, and implement them?

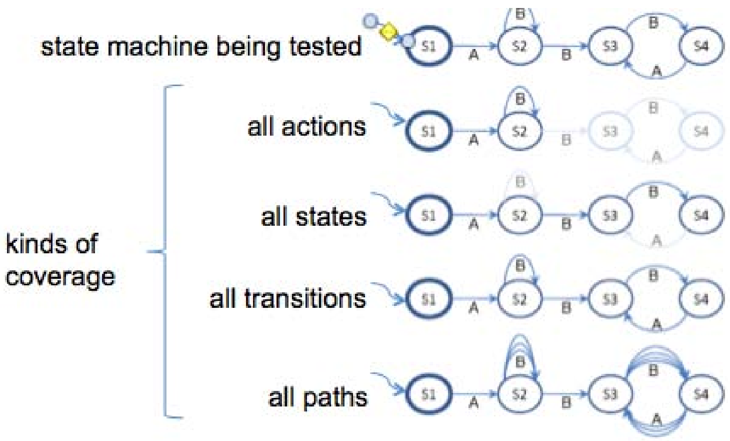
Testing a state machine

For mutable objects, test cases aren't just single inputs – they are *traces*, a sequence of events experienced by the object over its lifetime. For a stack, a single test case would be a sequence of push and pop (and size) method calls.

By drawing the state machine diagram, we can visualize our test cases and see how well they cover the different behaviors of the object. **Coverage** is a measure of test suite quality that refers to the extent to which the tests 'cover' the specification or the implementation.

There are four common kinds of coverage for state machines:

- *all-actions* includes every event in at least some test
- *all-states* visits every state (of the abstract state machine) in at least some test
- *all-transitions* makes every legal transition in the state machine in some test
- *all-paths* explores every possible path of transitions through the state machine.



The four kinds of coverage are related as follows:

- all actions and all-states are weaker than all-transitions
- all-transitions is weaker than all-paths

Consider testing the Iterator state machine below:

i<n   i=n

A test case is a **trace** of events through the state machine, which in this case are method calls like (*hasNext, next, hasNext, next*). Here are some test suites that obtain different levels of coverage of the Iterator state machine:

- all-actions: start with a 2-element list, create a MyIterator, and invoke (*hasNext, next*). This tries every event, so it achieves all-actions coverage, but it's not very convincing as a test suite! Which state of the state machine does this test suite completely miss?
- all states: start with a 1-element list, and then just call (*next*). This touches both states in the diagram, so it achieves all-states coverage, but it misses testing hasNext() entirely.
- all transitions: start with a 2-element list can invoke (*hasNext, next, next, hasNext).* This test crosses every edge in the state machine.

This approach to selecting test cases for a state machine is very similar to the input-space-partitioning technique we've already seen for functions. The state machine partitions the space of object states so that similar states and behaviors are lumped together (this is what we did when we abstracted away from the detailed Iterator state machine to the more abstract one seen here). Then all-transitions coverage selects test cases that cover the state machine.

Summary

This lecture took a look at mutable objects, and how to represent their behavior as state machines. We saw that mutability is useful for performance and convenience, but it also creates risks of bugs by requiring the code that uses the objects to be well-behaved on a global level, greatly complicating the reasoning and testing we have to do to be confident in its correctness. We also looked at how state machines can be used to represent input and output connections, and how they can help the selection of test cases.

6.005 Elements of Software Construction
Fall 2011