Lecture 10: Designing interfaces

CS 211 Spring 2006

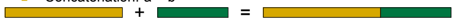
Andrew Myers

Announcements

- A3 due in 6 days
  - Focus: implementation, not documentation
- Special topics section on automatic garbage collection: Hollister 306, 2:30
- Last time:
  - Writing specifications
  - Using Javadoc
  - Programming advice
- Today's topics:
  - ADT Design
  - More programming advice

How to design an ADT

- Example: "Rope"
  - A heavier-weight string
  - Supports efficient **concatenation**
    - Concatenation: a + b



    - On String, takes time proportional to string length (copying)
  - Rope is useful for constructing long strings, e.g. web pages

1. ADT overview
2. Choose operations
3. Specify operations
4. Choose representation
5. Identify invariants
6. Implement operations

ADT overview

/** A Rope is a mutable string of characters. It supports efficient concatenation. */

```
interface Rope {
  …
}
```

Mutable vs. immutable

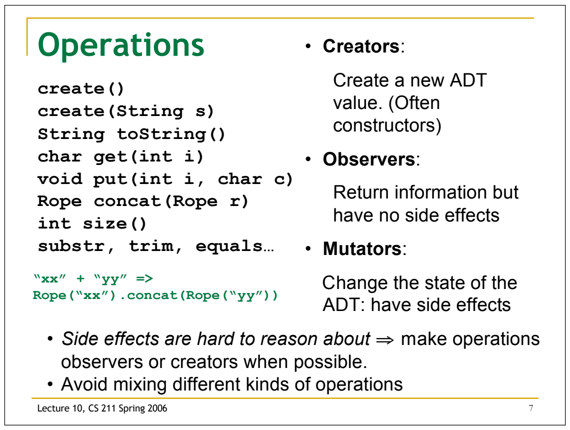
- Mutable abstractions have state that can be updated
- Immutable abstractions can't be changed after creation

- Mutable: arrays, ArrayList
- Immutable: int, String
  - x = 2; updates the variable x, doesn't change "2"
- Rule of thumb: immutable is usually easier to program with correctly

Choosing operations

- Interface should have enough operations for clients to do what they want
  - Efficiently
- Interface should avoid adding operations that few clients need and that are easily implemented.

- **narrow** vs. **wide** interfaces
  - Narrow => simple, client and implementation loosely coupled



Operations

```
create()
create(String s)
String toString()
char get(int i)
void put(int i, char c)
Rope concat(Rope r)
int size()
substr, trim, equals…
```

```
"xx" + "yy" =>
Rope("xx").concat(Rope("yy"))
```

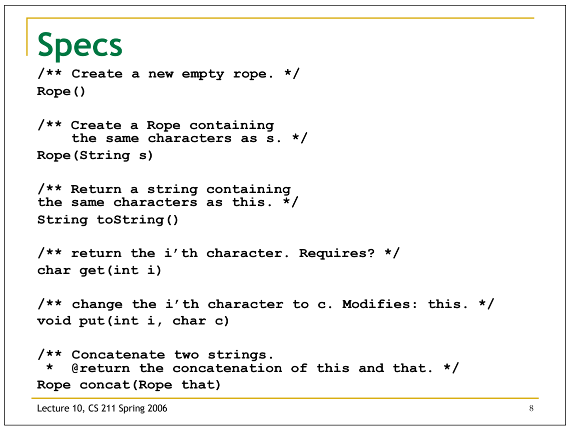
- **Creators**:

  Create a new ADT value. (Often constructors)
- **Observers**:

  Return information but have no side effects
- **Mutators**:

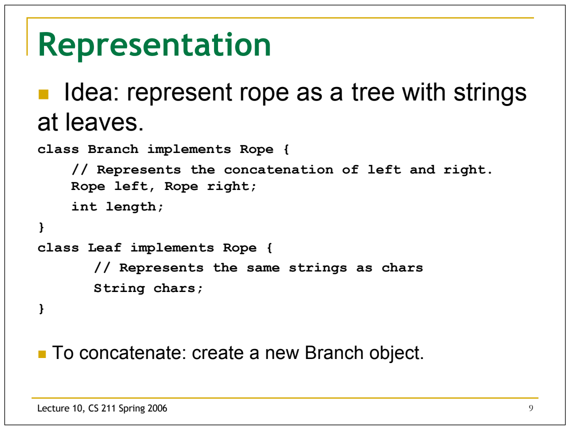
  Change the state of the ADT: have side effects

- *Side effects are hard to reason about* ⇒ make operations observers or creators when possible.
- Avoid mixing different kinds of operations



Specs

```
/** Create a new empty rope. */
Rope()

/** Create a Rope containing
    the same characters as s. */
Rope(String s)

/** Return a string containing
the same characters as this. */
String toString()

/** return the i'th character. Requires? */
char get(int i)

/** change the i'th character to c. Modifies: this. */
void put(int i, char c)

/** Concatenate two strings.
 *  @return the concatenation of this and that. */
Rope concat(Rope that)
```

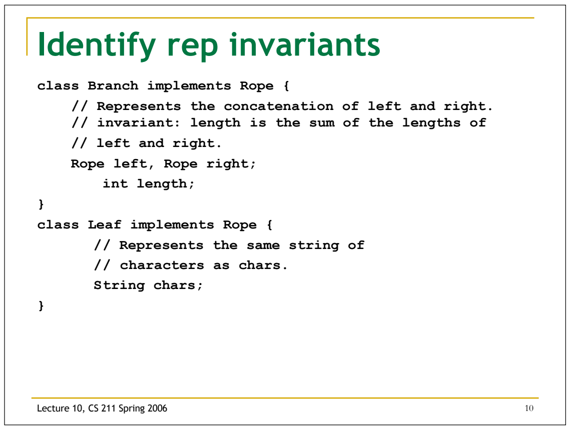


Representation

- Idea: represent rope as a tree with strings at leaves.

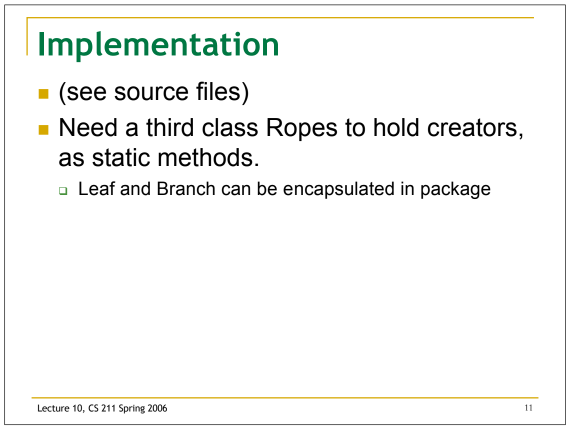
```
class Branch implements Rope {
    // Represents the concatenation of left and right.
    Rope left, Rope right;
    int length;
}
class Leaf implements Rope {
       // Represents the same strings as chars
       String chars;
}
```

- To concatenate: create a new Branch object.



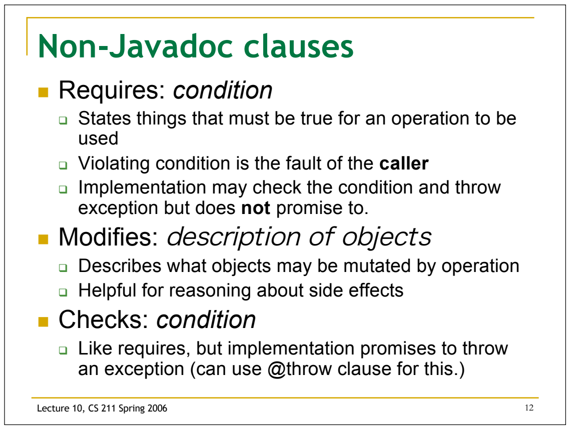
Identify rep invariants

```
class Branch implements Rope {
    // Represents the concatenation of left and right.
    // invariant: length is the sum of the lengths of
    // left and right.
    Rope left, Rope right;
        int length;
}
class Leaf implements Rope {
       // Represents the same string of
       // characters as chars.
       String chars;
}
```



Implementation

- (see source files)
- Need a third class Ropes to hold creators, as static methods.
  - Leaf and Branch can be encapsulated in package



Non-Javadoc clauses

- Requires: *condition*
  - States things that must be true for an operation to be used
  - Violating condition is the fault of the **caller**
  - Implementation may check the condition and throw exception but does **not** promise to.
- Modifies: *description of objects*
  - Describes what objects may be mutated by operation
  - Helpful for reasoning about side effects
- Checks: *condition*
  - Like requires, but implementation promises to throw an exception (can use @throw clause for this.)