

Massachusetts Institute of Technology
6.005: Elements of Software Construction
Fall 2011

Quiz 1

October 14, 2011

Name: Michael Plasencia

Athena User Name: theplaz

Instructions

This quiz is 50 minutes long. It contains 8 pages (including this page) for a total of 100 points. The quiz is closed-book, closed-notes.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name on the top of every page. Please write neatly. No credit will be given if we cannot read what you write. Good luck!

Question Name	Page	Maximum Points	Points Given
Regexes & grammars	2	12	12
State machines & testing	3	15	14
Specifications	4	18	18
Abstract data types (a)	5	5	5
Abstract data types (b)	6	18	13
Recursive data types (a)	7	15	10
Recursive data types (b)	8	15	5

79

Name: Plasmeier

Regular Expressions and Grammars

Consider the following grammar:

$F ::= B? E N^* M$

$B ::= >$

$E ::= : | ; | 8$

$N ::= - | ^$

$M ::= D | O | P$

$? = 0 \text{ or } 1$
 $* = 0 \text{ or more}$

(a) [8 pts] Which of the following strings could be legally and entirely recognized by the grammar? (circle all that apply)

☒ ;P
☐ :-^
☒ >8O
☐ :^OD
☐ B:^D
☐ <3
☐ O^:
☐ PP
☒ :---D

8/8

(b) [4 pts] Write a regular expression for this grammar. You can use any operators from common regular expression syntax. Quoting or escaping is unnecessary if your meaning is clear.

$[>]? [[:|;|8]] [-|^]* [D|O|P]$

4/4

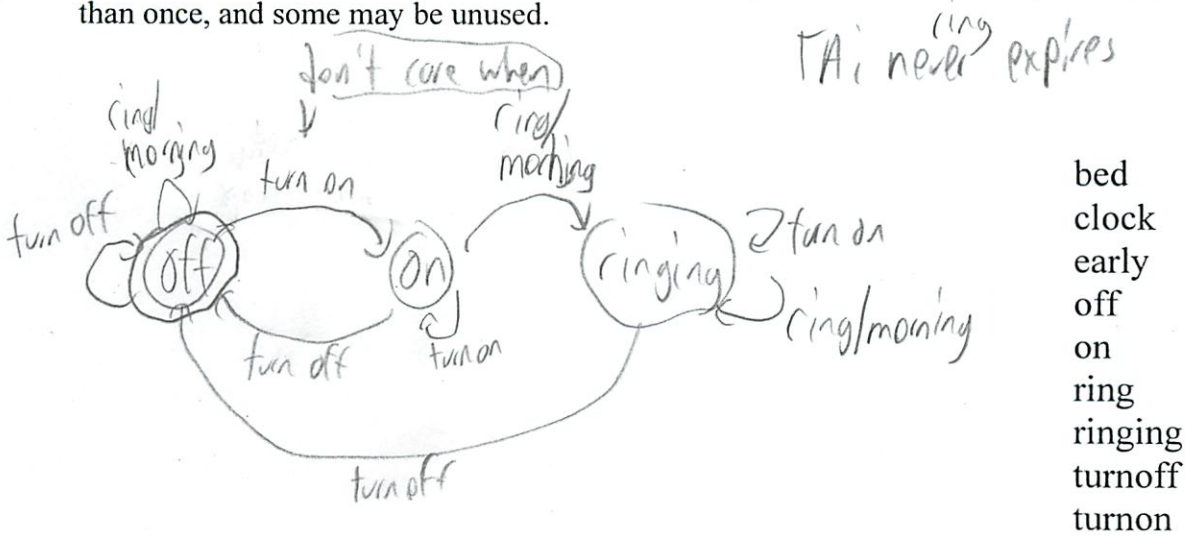
only need one of []'s or |'s.

Name:

State Machines and Testing

Before going to bed every night, Ben Bitdiddle turns on his alarm clock. It rings in the morning to wake him up, and he turns it off. Sometimes – not often – he wakes up early and turns the alarm off before it has a chance to ring.

(a) [12 pts] Draw a state machine for the alarm clock below. Label the states and the transitions, **using only the labels shown at the right**. Some labels may be used more than once, and some may be unused.

 $+12$

(b) [3 pts] Devise one or more test cases that together achieve all-transitions coverage for this state machine. Write each test case below, not as Java code but as an event trace -- a sequence of event labels from the state machine above.

off {	normal {	early {	don't off {
turn off	assert not Ringing	turn on	turn on
morning/ring	turn on	//go to bed	morning/ring
assert never Ring	assert not Ringing	//get up early	assert always Ring
	//go to bed	turn off	turn on
	morning/ring	assert never Ring	assert still Ring
	early assert is Ringing	3	Ring/morning
	turn off		assert still Ringing
	assert not Ringing		

Name: Plasmier

Specifications

Write good specifications for the following methods. Do not change the parameter types or return type of the method, but you may change other parts of the method signature if you feel it's necessary to write a good spec.

/** Compute the square root of a number, truncated to int.

✓ @param int x to compute sq rt of
must be > 0, < Max Int, != null

✓ @return int \sqrt{x} truncated to int

*/
public static int squareRoot(int x);

// IntSet represents a set of integers.

public class IntSet {

... // other fields and methods here

/** Find the smallest element in the set.

✓ @returns int that is smallest in set

✓ @requires IntSet to be non empty and sorted
└ must be made up of integers

*/ Does not modify int set

public int smallest();

}

/** Double every number.

Returns a new list of integers

✓ @param List<Integer> lst which is non 0 and all ints

✓ @returns List<Integer> every integer doubled - each Integer must be < MaxInt/2 not null

*/

public static List<Integer> doubleAll(List<Integer> lst);

Name: Plasnele

Abstract Data Types

Consider the following code.

```
1 /** Text is an immutable data type representing English text. */
2 public class Text {
3     private final String text;
4     private final String[] words;
5
6     // Rep invariant:
7     //   text != null; words != null;
8     //   concatenation of words (words[0]+words[1]+...+words[words.length-1])
9     //   is the same as text with spaces and punctuation removed
10    // Abstraction function:
11    //   represents the English text in the string variable text
12
13    /**
14     * Make a Text object.
15     * @param sentence a sentence in English. Requires sentence != null.
16     */
17    public Text(String sentence) {
18        this.text = sentence;
19        this.words = sentence.split(" ");
20    }
21
22    /** @return the words in the sentence */
23    public String[] getWords() {
24        return words;
25    }
26
27    /** @return the sentence as a string */
28    public String toString() {
29        return text;
30    }
31
32    /** concatenates this Text to that Text. Requires that != null. */
33    public Text add(Text that) {
34        return new Text(this.text + that.text);
35    }
36
37    /** @return true if and only if the word w is in the sentence.
38     * Requires w != null */
39    public boolean contains(String w) {
40        for (String v : words) { if (w.equals(v)) { return true; } }
41        return false;
42    }
43 }
```

(a) [5 pts] For each constructor and method above, write in the box next to it:

C for creator

P for producer ← forget this one

O for observer

M for mutator ← immutable ?

Name: Plasma

(b) [18 pts] The code above was code-reviewed, producing the comments below. Circle AGREE or DISAGREE depending on whether the comment is correct or incorrect, and add your own **one-sentence comment** explaining your answer. The right explanation is worth more than the right circle.

line 23: Rep exposure threatens the rep invariant!

AGREE

~~DISAGREE~~

for the specific line!
If underlying data type is mutable - problem
But String is ~~immutable~~

line 23 reply: No it doesn't, words is a final variable.

AGREE

~~DISAGREE~~

Not enough to guarantee, see above

line 18: Constructor doesn't establish the rep invariant.

~~AGREE~~

DISAGREE

spaces + punctuation need to
be removed

line 17: Rep exposure! Need to make a copy of text before storing it in your rep.

AGREE

~~DISAGREE~~

String is ~~immutable~~
and Text (recursively)

line 33: add() changes this.text, you shouldn't do that in an immutable type.

AGREE

~~DISAGREE~~

No it returns a new text object

line 39: the compiler's static checking will throw an exception here if w == null

AGREE

~~DISAGREE~~

I don't believe that is statically checked by compiler
- but ~~I~~ don't know
Pre condition covers that case anyway

Name: Plasma

Recursive Data Types

In this problem you will implement a recursive data type representing sets of words and intersections of those sets. The datatype definition is:

WordSet = Base(t:Text) + Intersect(left:WordSet, right:WordSet)

(a) [15pts] Write Java code below that implements this datatype. Include the reps (fields) and creators (constructors), but no other methods. **You don't need to write specs for your methods in this problem.** Note that you will need to use the Text datatype defined in the previous problem; assume that all its implementation bugs have been fixed so that it behaves according to its spec.

```
interface WordSet {  
    public WordSet (Base base, Intersect intersect) {  
        this.base = base;  
        this.intersect = intersect;  
    }  
}  
  
class Base implements WordSet {  
    public Base (Text t) {  
        this.t = t;  
    }  
}  
  
class Intersect implements WordSet {  
    public Intersect (WordSet left, WordSet right) {  
        this.left = left;  
        this.right = right;  
    }  
}
```

Name: Plasma 101

(b) [15pts] Define a function over your datatype:

contains: WordSet ws, String w => boolean

// requires: w is a word, with no spaces or punctuation

// returns: true if and only if w is an element of the

// set of words represented by ws

column by types

Implement the function using the **Interpreter pattern**. Write your Java code below. Again, you don't need to write specs for this problem. You also don't need to repeat the code you wrote above, but if you need to insert methods into classes you wrote above, just put a brief outline around it with the name of the class, e.g.

- did not look at much
- notes has 2 sections crossed out

```
ClassName {  
    public void myNewMethod() {  
        ...  
    }  
}
```

not strictly
interpreter ->
but
should work:

```
Text {  
    static public boolean checkSet (WordSet ws, String w) {  
        Iterator<String> it = ws.words.iterator();  
        while (it.hasNext()) {  
            if (w == it.next()) {  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

Iterator (String) it = ws.words.iterator();
this doesn't have an iterator... what is words?
w == it.next() {
return true;
} == does not work for String

// could do some instance of checking

see back
↙

END OF QUIZ

—Dr—

WordSet {

public checkSet (String w) {

Iterator<String> it = this.base.t.words.iterator();
etc same

3



Massachusetts Institute of Technology
6.005: Elements of Software Construction

Fall 2011

Quiz 1

October 14, 2011

Name: SOLUTIONS

Athena User Name: _____

Instructions

This quiz is 50 minutes long. It contains 1 pages (including this page) for a total of 100 points. The quiz is closed-book, closed-notes.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name on the top of every page. Please write neatly. No credit will be given if we cannot read what you write. Good luck!

Question Name	Page	Maximum Points	Points Given
Regexes & grammars	2	12	
State machines & testing	3	15	
Specifications	4	18	
Abstract data types (a)	5	5	
Abstract data types (b)	6	18	
Recursive data types (a)	7	15	
Recursive data types (b)	8	15	

Name: _____

Regular Expressions and Grammars

Consider the following grammar:

$$F ::= B? E N^* M$$
$$B ::= >$$
$$E ::= : | ; | 8$$
$$N ::= - | ^$$
$$M ::= D | O | P$$

(a) [8 pts] Which of the following strings could be legally and entirely recognized by the grammar? (circle all that apply)

☒ ;P

☐ :-^[

☒ >8O

☐ :^OD

☐ B:^D

☐ <3

☐ O^:

☒ :---D

(b) [4 pts] Write a regular expression for this grammar. You can use any operators from common regular expression syntax. Quoting or escaping is unnecessary if your meaning is clear.

$$>?[::8][-^]*[DOP]$$

or

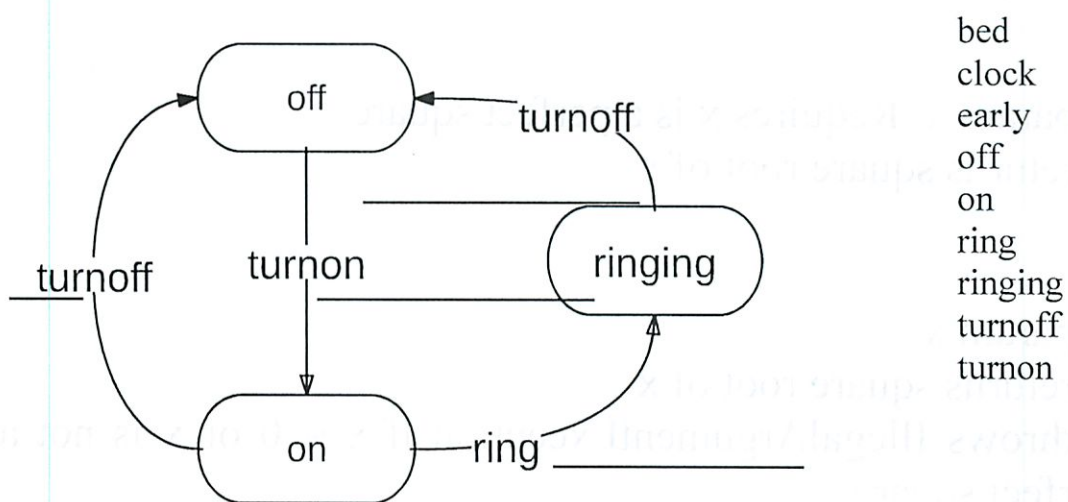
$$>?([:;|8)(-|^)*(D|O|P)$$

Name: _____

State Machines and Testing

Before going to bed every night, Ben Bitdiddle turns on his alarm clock. It rings in the morning to wake him up, and he turns it off. Sometimes – not often – he wakes up early and turns the alarm off before it has a chance to ring.

(a) [12 pts] Draw a state machine for the alarm clock below. Label the states and the transitions, **using only the labels shown at the right**. Some labels may be used more than once, and some may be unused.



(b) [3 pts] Devise one or more test cases that together achieve all-transitions coverage for this state machine. Write each test case below, not as Java code but as an event trace -- a sequence of event labels from the state machine above.

start in off state, then
turnon, ring, turnoff, turnon, turnoff

or

start in off state, then turnon, ring, turnoff
start in off state, then turnon, turnoff

Name: _____

Specifications

Write good specifications for the following methods. Do not change the parameter types or return type of the method, but you may change other parts of the method signature if you feel it's necessary to write a good spec.

`/** Compute the square root of a number.`

`@param x` Requires $x \geq 0$

`@returns` largest integer n such that $n * n \leq x$

or

`@param x` Requires x is a perfect square

`@returns` square root of x

or

`@param x`

`@returns` square root of x

`@throws` `IllegalArgumentException` if $x < 0$ or x is not a perfect square

`*/`

`public static int squareRoot(int x);`

`// IntSet represents a set of integers.`

`public class IntSet {`

`... // other fields and methods here`

`/** Find the smallest element in the set.`

Requires this set is nonempty.

`@returns` smallest x such that x is in this set

or

`@returns` smallest x such that x is in this set

`@throws` `EmptyException` if set is empty

Name: _____

or

@returns smallest x such that x is in this set, or Integer.MIN_INT if set is empty

```
*/  
public int smallest();  
}
```

/** Double every number.

Modifies nothing / Doesn't modify lst.

@returns new list lst2, of same length as lst, such that for all i ($0 \leq i < \text{lst.length}$), $\text{lst2}[i] = \text{lst}[i] * 2$

or

Modifies lst such that $\text{lst}[i]$ after the call is twice as large as $\text{lst}[i]$ before the call.

@returns lst

```
*/  
public static List<Integer> doubleAll(List<Integer> lst);
```


Name: _____

Abstract Data Types

Consider the following code.

```
1 /** Text is an immutable data type representing English text. */
2 public class Text {
3     private final String text;
4     private final String[] words;
5
6     // Rep invariant:
7     //   text != null; words != null;
8     //   concatenation of words (words[0]+words[1]+...+words[words.length-1])
9     //   is the same as text with spaces and punctuation removed
10    // Abstraction function:
11    //   represents the English text in the string variable text
12
13    /**
14     * Make a Text object.
15     * @param sentence a sentence in English. Requires sentence != null.
16     */
17    public Text(String sentence) {
18        this.text = sentence;
19        this.words = sentence.split(" ");
20    }
21
22    /** @return the words in the sentence */
23    public String[] getWords() {
24        return words;
25    }
26
27    /** @return the sentence as a string */
28    public String toString() {
29        return text;
30    }
31
32    /** concatenates this Text to that Text. Requires that != null. */
33    public Text add(Text that) {
34        return new Text(this.text + that.text);
35    }
36
37    /** @return true if and only if the word w is in the sentence.
38     * Requires w != null */
39    public boolean contains(String w) {
40        for (String v : words) { if (w.equals(v)) { return true; } }
41        return false;
42    }
43 }
```

C

O

O

P

O

(a) [5 pts] For each constructor and method above, write in the box next to it:

C for creator
P for producer
O for observer
M for mutator

(b) [18 pts] The code above was code-reviewed, producing the comments below. Circle AGREE or DISAGREE depending on whether the comment is correct or incorrect, and add your own **one-sentence comment** explaining your answer. The right explanation is worth more than the right circle.

line 23: Rep exposure threatens the rep invariant!

AGREE

DISAGREE

Name: _____

Yes, the caller could mutate the words array, making it no longer match sentence (and Text also no longer immutable).

line 23 reply: No it doesn't, words is a final variable.

AGREE DISAGREE

final just makes the words reference unchangeable; it doesn't prevent the words array from being mutated.

line 18: Constructor doesn't establish the rep invariant.

AGREE DISAGREE

Needs to look at punctuation too.

line 17: Rep exposure! Need to make a copy of text before storing it in your rep.

AGREE DISAGREE

String is immutable, so it's safe to share it between rep and caller.

line 33: add() changes this.text, you shouldn't do that in an immutable type.

AGREE DISAGREE

add is a producer, not a mutator; it doesn't modify this.text, and can't anyway because this.text is a final reference to an immutable object.

line 39: the compiler's static checking will throw an exception here if `w == null`

AGREE DISAGREE

an exception will be thrown, but not by the compiler and not by static checking; exceptions are thrown at runtime.

Recursive Data Types

In this problem you will implement a recursive data type representing sets of words and intersections of those sets. The datatype definition is:

Name: _____

WordSet = Base(t:Text) + Intersect(left:WordSet, right:WordSet)

(a) [15pts] Write Java code below that implements this datatype. Include the reps (fields) and creators (constructors), but no other methods. **You don't need to write specs for your methods in this problem.** Note that you will need to use the Text datatype defined in the previous problem; assume that all its implementation bugs have been fixed so that it behaves according to its spec.

```
public interface WordSet {  
}  
  
public class Base implements WordSet {  
    private final Text t;  
    public Base(Text t) {  
        this.t = t;  
    }  
}  
  
public class Intersect implements WordSet {  
    private final WordSet left, right;  
  
    public Intersect(WordSet left, WordSet right) {  
        this.left = left;  
        this.right = right;  
    }  
}
```

Name: _____

- (b) [15pts] Define a function over your datatype:
contains: WordSet ws, String w => boolean
// requires: w is a word, with no spaces or punctuation
// returns: true if and only if w is an element of the
// set of words represented by ws

Implement the function using the **Interpreter pattern**. Write your Java code below. Again, you don't need to write specs for this problem. You also don't need to repeat the code you wrote above, but if you need to insert methods into classes you wrote above, just put a brief outline around it with the name of the class, e.g.

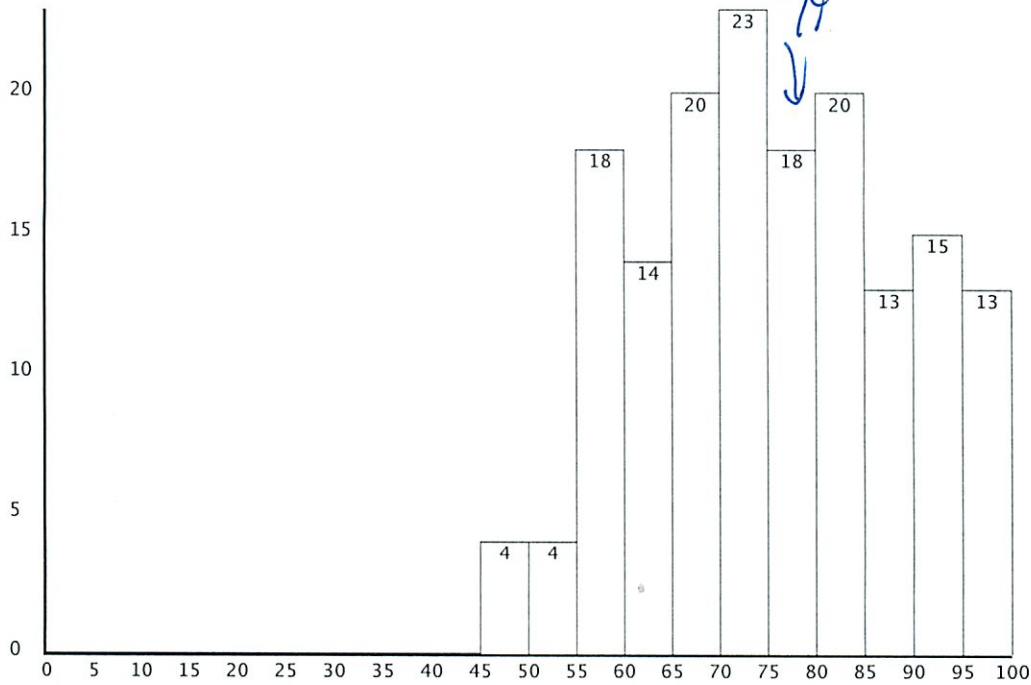
```
ClassName {  
    public void myNewMethod() {  
        ...  
    }  
}  
  
WordSet {  
    public boolean contains(String w);  
}  
  
Base {  
    public boolean contains(String w) {  
        return t.contains(w);  
    }  
}  
  
Intersect {  
    public boolean contains(String w) {  
        return left.contains(w) || right.contains(w);  
    }  
}
```

END OF QUIZ

6.005 Software Construction

Dashboard Students Assignments

Grading Summary for Quiz 1



Number of Scores: 162
 Average: 74.55
 Standard Deviation: 13.48

ewoot! better than avg!

61005 Debreit

10/19

Messed up Interpreter

- stupid anyway

Think test went very well

- considering studied 40 min

- and in group

Forgot a term as well

Project 1: An ABC Music Player

Friday, October 14, 2011

analysis, etc). Finally, the project will give you further practice in software engineering fundamentals, such as (7) clarifying a problem statement; (8) inventing clear and simple interfaces to minimize coupling, and identifying and resolving undesirable dependences; (9) structuring a program to make it easily testable, organizing, executing and evaluating test suites; and (10) working collaboratively in a team.

References

Before reading the problem specification, you should keep in mind that this document is **NOT** meant to provide you with comprehensive information on the abc notation. Instead, you should consult the following list of sites during your project:

- abc standard v1.6: Current official standard for abc.
- Required abc subset: a description of the required abc subset, including the grammar in an EBNF. For your interest, the full EBNF can be found here, but remember, you are required to implement **ONLY** our subset.
- Chris Walshaw's abc site: An informative web site by the inventor of the language. Among other things, the site contains a set of examples and a tutorial, which should help get you up to speed with abc.
- John Chambers' abc site: Another comprehensive site on abc. A great feature on this site is the abc tune finder, which lets you search through thousands of abc files around the web.
- Wikipedia article on abc.
- Wikipedia article on modern musical symbols: A fairly comprehensive overview of the Western musical notation.

Specification

Note: You are **NOT** allowed to use any code taken from an existing abc player as a part of your implementation in this project. *our old player*

Required Subset of abc. The subset of abc that you are required to implement in this project is described in a separate document, the abc subset for 6.005.

Tasks

You should perform the following tasks:

- **Team Contracts.** Please see this page for details.
- **Warmup.** There are some warmup exercises that you need to do, listed below. They will help you learn about how the abc music player works.
- **Grammars and datatypes.** Write out the grammar of the abc music player. Then specify which datatype definitions you would like to use for the different parts of the project.
- **Snapshot Diagram.** Write out diagrams of 3 distinct example ABC expressions. It should show what happens in each part of the code in the middle of running the program.
- **Implementation.** Based on how your snapshot diagram, implement your code in Java. You may find that you want to make changes to your design, and thus your snapshot diagram. You are free to do this, but should record the changes so it is clear how and why you diverged from the original. Remember to do test first programming. You will need to write unit tests.
- **Test.** Test your entire system on the staff sample inputs. Create at least **three** additional test cases (e.g. your own abc files) to demonstrate that your player is able to correctly parse and play various musical constructs, and also detect any semantic errors in an abc file.
- **Reflection.** Write a brief commentary saying what you learned from this experience. What was easy? What was hard? What was unexpected? Briefly evaluate your solution, pointing out its key merits and deficiencies. This is an individual activity.

Infrastructure

Java MIDI Sequencer

Before you start the project, you should learn about the Java MIDI Sequencer. The Sequencer allows you

to schedule a series of notes to be played at certain time intervals.

Look at the package `sound` under `src`, and study the provided class, `SequencePlayer`. For this project, you will not need to modify this class, but you should become comfortable using its constructor and the two methods, `addNote` and `play`.

`addNote(int note, int startTick, int numTicks)`: This method schedules a note to be played at `startTick` for the duration of `numTicks`. Here, a "tick" is similar to a time step. At the beginning of a musical piece, the global tick is initialized to 0, and as the music progresses through the notes, the global tick is incremented by some number.

The first parameter `note` is a MIDI note value that corresponds to the pitch of a note. The provided class `Pitch` contains a number of useful methods for working with pitches. The method `toMidiNote` returns the MIDI note value of the particular note, and `transpose` can be used to transpose the note some number of semitones up or down.

`SequencePlayer(int beatsPerMinute, int ticksPerQuarterNote)`: The constructor for `SequencePlayer` takes two parameters:

Songwriting class in 4/5!

- The first, `beatsPerMinute`, is the tempo of a musical piece. It is expressed in the number of beats per minute (BPM), where each beat is equal to the duration of **one quarter note**. The BPM to be used for a particular piece depends on the value of the optional tempo field ('Q') in the input abc file. When this field is absent, the default tempo is 100 BPM, where each beat is equal in duration to the **default note length** (indicated by the field 'L').
- The second parameter, `ticksPerQuarterNote`, corresponds to the number of ticks per quarter note. Note that ticks used by the sequencer are based on discrete time. Think about how large this number needs to be in order to play notes of different durations in an abc piece. For example, if `ticksPerQuarterNote` had a value of 2, then an eighth note would be played for the duration of one tick, but you would not be able to schedule a sixteenth note for the correct duration.

`play()`: After all of the notes have been scheduled, you can invoke `play` to tell the sequencer to begin playing the music.

Run the main method in `SequencePlayer`, which shows an example of using a sequencer to play up and down a C major scale. In this example, all of the notes in the scale have been hard-coded. In your abc player, you will be walking over your data structures that represent a musical piece and automatically schedule the notes at appropriate ticks.

We are also providing some example abc files that you can use to test your abc player (included in the SVN directory `sample_abc`):

- A simple scale (scale.abc)
- A Little Night Music by W. A. Mozart (little_night_music.abc)
- Paddy O'Rafferty, an Irish tune (paddy.abc)
- Invention by J. S. Bach (invention.abc)
- Prelude by J. S. Bach (prelude.abc)
- Fur Elise by L. v. Beethoven (fur_elise.abc)

You can find many more examples online, including [here](#).

Warmup

abc Music Notation

Task 1: Transcribe each of the following small pieces of music into an abc file. Name your files as **piece1.abc** and **piece2.abc**, respectively, and commit them under the directory `sample_abc` in your team's SVN repository.

Piece No.1: A simple, 4/4 meter piece with triplets. As a starter, the header and the first two bars are already provided. You should complete the rest of the piece by transcribing the last two bars.

Piece No.2: A more complex piece, with chords, accidentals, and rests. **Set its tempo to 200, with the default note length of 1/4.**

Task 2: Write JUnit tests that play these pieces using the sequencer, similar to the main method in the `SequencePlayer` class. Store them in a separate class called `SequencePlayerTest`.

Deliverables and Grading

- Team contracts, which should be in a file called TeamContract.pdf;
- Your results to the tasks in Warmup;
- Grammars, datatypes, state machines, and snapshot diagrams of the three given example ABC expressions, in one file called Design.pdf.

- the revised design in Design.pdf;
- the implementation;
- the tests;

- Reflections on Team. How did you feel the group did? How did your team work? How was the coding? How did you split the work?
- Reflections on Individual. How do you think you did? What did you do in the project? How do you feel about it?

Reflections should be committed to abc-reflections, which you will pull as if it's a pset.

Your code should be committed in the repository you share with your teammates by the deadline. All other parts of the project should be stored in your repository as two separate PDF documents, one for each deadline, as mentioned above. Each commit to the repository should have a commit comment saying what you changed, as well as who worked on it. Your TA/Mentor will be receiving your commit emails.

↑
dan

Grades will be allotted according to the following breakdown:

- Team Contract -- 5%;
- Design -- 25%;
- Implementation -- 50%;
- Testing -- 15%;
- Reflection -- 5%

Hints

Start early! This project is more work than it seems. Starting early on the project will give you more time to sort out any issues and ask the staff questions that may arise, especially if you have trouble with transcribing music.

Lexing and parsing: Given the grammar for abc, you will need to build a lexer that breaks the input string into tokens and a parser that groups the tokens into a valid syntactic construct and produces an abstract syntax tree (AST).

Designing datatypes: For the design of your abc player, you should think carefully about datatypes that you need to represent the musical constructs in abc. Start with simple constructs, such as notes and rests, and think about how you would build up on these primitive objects to create more complex structures. How would you represent a triplet? A chord? What does a bar consist of? How would you represent multiple voices? Sort out answers to these questions with your team members during the design stage.

Why do there need to be certain datatypes?

Evaluating expressions: Once you parse an abc file and create your own internal representation of the music as an abstract syntax tree (AST), you will need to perform various computations that involve traversing the tree, possibly multiple times. Consider carefully the patterns you learned for performing these traversals: you may want to use the visitor pattern, though an interpreter pattern might also work.

Parsing and pattern matching: We recommend that you think carefully about your approach to parsing the abc file; consider how to make it easiest to write a single parser that can handle the entire file, with complexities such as bars split across lines. You may find the Java pattern matching libraries such as `java.util.regex.Pattern` and `java.util.regex.Matcher` helpful, but you should use them judiciously.

Multiple voices: A particular challenge you should think about is how you will represent multiple voices, and how you will merge them into a single sequence of midi events.

Use classtime wisely: Lecture and Recitation are cancelled for team/TA meetings for the duration of the project. You should use the time to work on the project, or meet up with your TA/mentor. This mentor is randomly assigned, so may not be the one who teaches your recitation.

An ABC Subset for 6.005

This document provides a high-level, but not necessarily comprehensive, overview of language constructs that your abc player will need to handle. For information on the exact syntax of abc, you should consult the grammar in the [EBNF](#).

An abc file consists of two parts: (1) the header, which contains various information about the musical piece, such as its title, composer, tempo, meter, and note length, and (2) the body, which contains a sequence of notes that make up the musical piece.

1 Header

Each line in the header corresponds to a *field*, and begins with a single uppercase letter and a colon (':'), followed by the content of the field. The following shows an example of what a header may look like:

```
X: 3
T: Turkish March
C: W. Mozart
M: 2/4
L: 1/8
K: Am
```

Need to learn music terms

According to this header, the title of the piece contained in this file is "Turkish March", and was composed by a person named "W. Mozart". The piece is in A minor ('K: Am'), and the default length of each note or beat in the music is one eighth ('L: 1/8'). Each bar contains two quarter beats ('M: 2/4').

The abc standard specifies a large number of fields. You are required to handle only the following subset:

- **C:** Name of the composer.
- **K:** Key, which determines the *key signature* for the piece.
- **L:** Default length or duration of a note. *it's per qtr note*
- **M:** Meter. It determines the sum of the durations of all notes within a bar.
- **Q:** Tempo. It represents the number of default-length notes per minute. *beats per minute*
- **T:** Title of the piece.
- **X:** Index number, similar to the track number in a recording. In this project, this field does not carry any meaning, as you are required to parse only one abc file at a time. However, the official standard designates this field to be mandatory in every abc file, and therefore, your parser must be able at least to read it (and may then discard it).

- not quite - smallest unit want to play

There are several additional requirements on the header:

- The first field in the header must be the index number ('X').
- The second field in the header must be the title ('T').
- The last field in the header must be the key ('K').
- Each field in the header occurs on a separate line.

All fields other than 'X', 'T', and 'K' are optional, and may appear in any order. When omitted, the default values are 1/8 for the note length, 4/4 for the meter, and 100 beats per minute for the tempo. When the field 'C' is omitted, any reasonable string will suffice (e.g. "Unknown").

More about key signature: A key signature is a set of sharps or flats associated with particular notes in the scale. A *sharp* causes every occurrence of that note to be played one semitone **higher**, and a *flat* causes every occurrence of that note to be played one semitone **lower**. For example, the key signature in the diagram shown on the right contains three sharps (placed on C, F, and G). Throughout the piece, every C, F or G (irrespective of the octave) should be played one semitone higher. Your abc player should handle every major and minor key; the list of keys and their signatures can be found [here](#). You should also note that the effect of the key signature can be temporarily overridden



*C, F, G
one tone
higher*

2 Body

Journal of Management Inquiry 18(6)

Classification of

...the consists of

On this occasion,

← all at once

at once
next highest

the next higher

sequences of notes th

b

Wickham

the two octaves up

also fill

the three octaves u

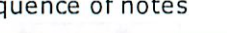
(faint handwritten notes)

— example, the fall

ated by

1. A. B. C. D. E. F. G.

... has the long



32 32 34 36 38

Note that either the denominator or the numerator (or both) in a fractional multiplicative factor can be omitted. An absent numerator should be treated as 1, and an absent denominator as 2. In addition, any apostrophes or commas that indicate the deviation from the middle octave must precede the multiplicative factor.

2.3 Accidentals

The effect of the key signature as discussed in Section 1 ('K' field) can be temporarily overridden using accidentals. An accidental can be a sharp (denoted by '^' in abc), a flat ('_'), or a natural ('='). A sharp causes a note to be played one semitone higher, a flat one semitone lower, and a natural causes the note to be played as if the key were in C without any accidental. Double sharps ('^^') and flats ('__') are also allowed.

The effect of an accidental on a note lasts only up to the end of the current bar, and can be overridden by another accidental that occurs in the same bar. The effect of the key signature is restored at the beginning of the next bar.

2.4 Rests

A rest is denoted by

z

and has the same default length as a note. Multiplicative factors have the same effect on rests as they do on notes, but accidentals cannot be applied to rests.

2.5 Chords

One or more notes may be played simultaneously in a chord. In abc, a chord is denoted by a group of notes between square brackets '[' and '']':

[CEG]

no this is all at once

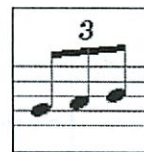
Notes within a chord can be embellished with an accidental or a multiplicative factor.

2.6 Tuplets

A tuplet is a consecutive group of notes that are to be played for a duration that is either greater or less than the sum of the individual notes within that group. In abc, a tuplet is denoted by an opening round bracket '(', the tuplet number, and the actual notes in the tuplet. For example,

(3GAB

is a tuplet that consists of three notes, and called triplet.



*← 150
60 up
support this?*

In this project, we require you to handle **ONLY** duplets (a group of two notes), triplets, and quadruplet (a group of four notes). The duration of each type of the tuplets are as follows:

- Duplet: 2 notes in the time of 3 notes
- Triplet: 3 notes in the time of 2 notes
- Quadruplet: 4 notes in the time of 3 notes

For example, a triplet that contains three eighth notes is equal in duration to one quarter note; therefore, each eighth note in the triplet should be played $\frac{2}{3}$ the duration of a standard eighth note.

2.7 Repeats

A section of music that is enclosed within ':' and ':' is to be repeated once. For example, in the following fragment of abc,

|: C D E F | G A B c :|

the two bars are repeated, so the sequence of notes that your abc player should produce as output is:

C D E F G A B c C D E F G A B c

The begin repeat bar ':' may be omitted; in this case, the repeat is from the beginning of a major section (i.e. the bar that immediately follows '[]'), or the beginning of the musical piece.

A repeated section may have a different ending when it is played the second time. In abc, alternate endings are indicated using '[1]' and '[2]'. For example,

```
|: C D E F |[1 G A B c :|[2 F E D C |
```

should be played as

```
C D E F G A B c C D E F F E D C
```

Note that when the repeat is played the second time, the bar that begins with '[1]' is entirely skipped over.

3 Non-standard Extension: Multiple Voices

All of the abc constructs that we have discussed so far allow us to play only a single melodic line. Many pieces of music (e.g. orchestral or band music), however, involve multiple instruments or voices that are played simultaneously. Multiple voices do not belong to the official abc language definition, but have nevertheless become a part of the *de facto* standard among abc users.

Different definitions for multiple voices can be found online. Your abc player should be able to parse and play multiple voices as defined as follows. Your abc player is **NOT** required to be able to play different instruments. *kinda basic*

Voices are listed in the header of an abc file using one or more field lines that begin with 'V'. The content of each voice field is the identifier for a particular voice, and can be an arbitrary string. For example, the header

```
X:0
T: Prelude No. 1
C: J. S. Bach
M:4/4
L:1/16
Q:100
V: upper
V: middle
V: lower
K:C
```

says that this piece of music contains three different voices, labeled "upper", "middle", and "lower". There is no limit on the number of voices in a piece.

A 'V' field line may re-appear in the middle of the body to indicate that the following sequence of notes belongs to a particular voice, until another voice field with a different identifier appears. For increased readability, middle-of-body voice fields are often placed between small sequences of bars for different voices in alternating fashion (as it is normally done in sheet music):

```
V: upper
z2 Gc eGce z2 Gc eGce | z2 Ad fAdf z2 Ad fAdf |
V: middle
z E7 z E7 | z D7 z D7 |
V: lower
C8 C8 | C8 C8 |
V: upper
z2 Gd fGdf z2 Gd fGdf | z2 Ae aAea z2 Ae aAea ||
V: middle
z D7 z D7 | z E7 z E7 ||
V: lower
B,8 B,8 | C8 C8 ||
```

However, your abc player should not make any assumptions about the order or frequency of middle-of-

body voice fields. For example, instead of interleaving voices as in the previous example, an abc author could write each voice in its entirety under a single middle-of-body voice field:

V: upper

z2 Gc eGce z2 Gc eGce | z2 Ad fAdf z2 Ad fAdf | z2 Gd fGdf z2 Gd fGdf | z2 Ae aAea z2 Ae aAea |]

V: middle

z E7 z E7 | z D7 z D7 | z D7 z D7 | z E7 z E7 |]

V: lower

C8 C8 | C8 C8 | B,8 B,8 | C8 C8 |]

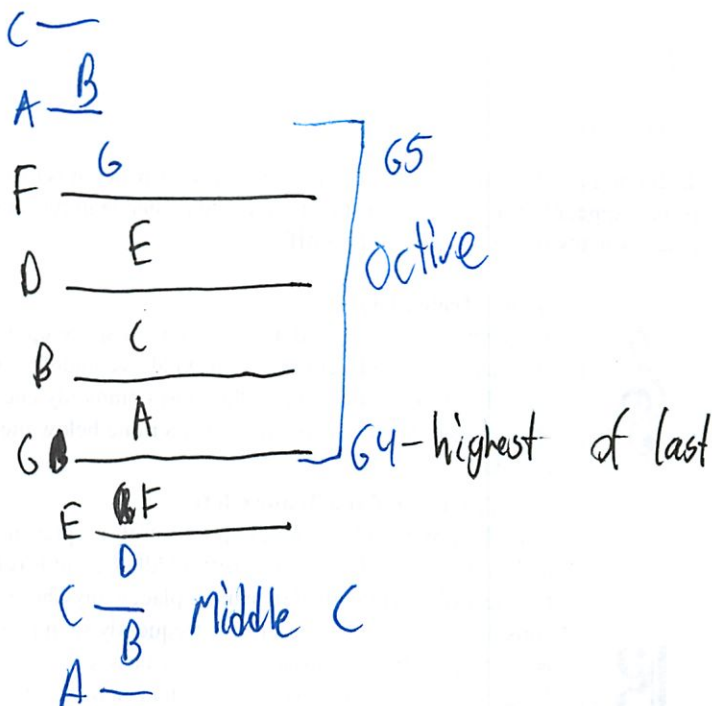
List of musical symbols

From Wikipedia, the free encyclopedia

Modern musical symbols are the marks and symbols that are widely used in musical scores of all styles and instruments today. This is intended to be a comprehensive guide to the various symbols encountered in modern musical notation.

Contents

- 1 Lines
- 2 Clefs
- 3 Notes and rests
- 4 Breaks
- 5 Accidentals and key signatures
 - 5.1 Common accidentals
 - 5.2 Key signatures
 - 5.3 Quarter tones
- 6 Time signatures
- 7 Note relationships
- 8 Dynamics
- 9 Articulation marks
- 10 Ornaments
- 11 Octave signs
- 12 Repetition and codas
- 13 Instrument-specific notation
 - 13.1 Guitar
 - 13.2 Piano
 - 13.2.1 Pedal marks
 - 13.2.2 Other piano notation
- 14 See also
- 15 References
- 16 External links



Lines

Staff

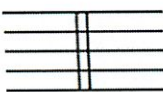
The fundamental latticework of music notation, upon which symbols are placed. The five stave lines and four intervening spaces correspond to pitches of the diatonic scale - which pitch is meant by a given line or space is defined by the clef. With treble clef, the bottom staff line is assigned to E above middle C (E4 in note-octave notation); the space above it is F4, and so on. The grand staff combines bass and treble staves into one system joined by a brace. It is used for keyboard and harp music. The lines on a basic five-line staff are designated a number from one to five, the bottom line being the first one and the top line being the fifth. The spaces between the lines are, in the same fashion, numbered from one to four. In music education, for the Treble Clef, the mnemonic "Every Good Boy Does Fine" (or "Every Good Boy Deserves Fudge") is used to remember the value of each line from bottom to top. The interstitial spaces are often remembered as spelling the word "face" (notes F-A-C-E).

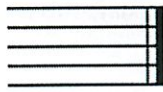
Ledger or leger lines

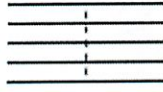
Used to extend the staff to pitches that fall above or below it. Such ledger lines are placed behind the note heads, and extend a small distance to each side. Multiple ledger lines may be used when necessary to notate pitches even farther above or below the staff.


Bar line

Used to separate measures (see time signatures below for an explanation of measures). Bar lines are extended to connect the upper and lower staves of a grand staff.

**Double bar line, Double barline**
Used to separate two sections of music. Also used at changes in key signature, time signature or major changes in style or tempo.

**Bold double bar line, Bold double barline**
Used to indicate the conclusion of a movement or an entire composition.


**Dotted bar line, Dotted barline**
Subdivides long measures of complex meter into shorter segments for ease of reading, usually according to natural rhythmic subdivisions.


**Accolade, brace**
Connects two or more lines of music that are played simultaneously.^[1] Depending on the instruments playing, the brace, or accolade, will vary in designs and styles.

Clefs

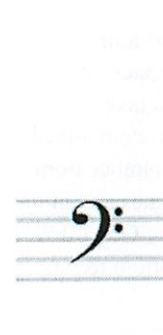
Main article: *Clef*

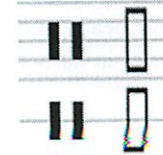
Clefs define the pitch range, or tessitura, of the staff on which it is placed. A clef is usually the *leftmost* symbol on a staff. Additional clefs may appear in the middle of a staff to indicate a change in register for instruments with a wide range. In early music, clefs could be placed on any of several lines on a staff.


**G clef (Treble Clef)**
The centre of the spiral defines the line or space upon which it rests as the pitch *G above middle C*, or approximately 392 Hz. Positioned here, it assigns G above middle C to the *second line from the bottom* of the staff, and is referred to as the "treble clef." This is the most commonly encountered clef in modern notation, and is used for most modern vocal music. Middle-C is the 1st ledger line below the stave here. The shape of the clef comes from a stylised upper-case-G.

**C clef (Alto Clef and Tenor Clef)**
This clef points to the line (or space, rarely) representing middle C, or approximately 262 Hz. Positioned here, it makes the *center line on the staff* middle C, and is referred to as the "alto clef." This clef is used in modern notation for the viola. While all clefs can be placed anywhere on the staff to indicate various tessitura, the C clef is most often considered a "movable" clef: it is frequently seen pointing instead to the fourth line and called a "tenor clef". This clef is used very often in music written for bassoon, cello, and trombone; it replaces the bass clef when the number of ledger lines above the bass staff hinders easy reading.

C clefs were used in vocal music of the classical era and earlier; however, their usage in vocal music has been supplanted by the universal use of the treble and bass clefs. Modern editions of music from such periods generally transpose the original C-clef parts to either treble (female voices), octave treble (tenors), or bass clef (tenors and basses).

**F clef (Bass Clef)**
The line or space between the dots in this clef denotes F below middle C, or approximately 175 Hz. Positioned here, it makes the *second line from the top* of the staff F below middle C, and is called a "bass clef." This clef appears nearly as often as the treble clef, especially in choral music, where it represents the bass and baritone voices. Middle C is the 1st ledger line above the stave here. The shape of the clef comes from a stylised upper-case-F (which used to be written the reverse of the modern F)

**Neutral clef**
Used for pitchless instruments, such as some of those used for percussion. Each line can represent a specific percussion instrument within a set, such as in a drum set. Two different styles of neutral clefs are pictured here. It *may also* be drawn with a separate single-line staff for each untuned percussion instrument.

**Octave Clef**
Treble and bass clefs can also be modified by octave numbers. An eight or fifteen above a clef raises the intended pitch range by one or two octaves respectively. Similarly, an eight or fifteen below a clef lowers the pitch range by one or two octaves respectively. A treble clef with an eight below is the most commonly used, typically used instead of a C clef for tenor lines in choral scores. Even if the eight is not present, tenor parts in the treble clef are understood

to be sung an octave lower than written.




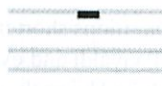



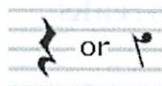





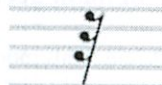


Tablature

For guitars and other plucked instruments it is possible to notate tablature in place of ordinary notes. In this case, a TAB-sign is often written instead of a clef. The number of lines of the staff is not necessarily five: one line is used for each string of the instrument (so, for standard 6-stringed guitars, six lines would be used). Numbers on the lines show on which fret the string should be played. This Tab-sign, like the Percussion clef, is not a clef in the true sense, but rather a symbol employed instead of a clef. The interstitial spaces on a tablature are never used.

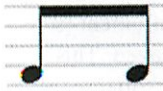
Notes and rests

Main article: Note value

Note and rest values are not absolutely defined, but are proportional in duration to all other note and rest values. The whole note is the reference value, and the other notes are named (in American) in comparison; i.e. a quarter note is a quarter the length of a whole note.

Note	British name / American name	Rest
	Breve / Double whole note	
	Semibreve / Whole note	
	Minim / Half note	
	Crotchet / Quarter note	
	Quaver / Eighth note	
For notes of this length and shorter, the note has the same number of flags (or hooks) as the rest has branches.		
	Semiquaver / Sixteenth note	
	Demisemiquaver / Thirty-second note	
	Hemidemisemiquaver / Sixty-fourth note	

Beamed notes



Beams connect eighth notes (quavers) and notes of shorter value, and are equivalent in value to flags. In metered music, beams reflect the rhythmic grouping of notes. They may also be used to group short phrases of notes of the same value, regardless of the meter; this is more common in ametrical passages. In older printings of vocal music, beams are often only used when several notes are to be sung to one beat; modern notation encourages the use of beaming in a consistent manner with instrumental engraving, and the presence of beams or flags no longer informs the singer. Today, due to the body of music in which traditional metric states are not always assumed, beaming is at the discretion of the composer or arranger and irregular beams are often used to place emphasis on a particular rhythmic pattern.

Dotted note



Placing dots to the right of the corresponding notehead lengthens the note's duration, e.g. one dot by one-half, two dots by three-quarters, three dots by seven-eighths, and so on. Rests can be dotted in the same manner as notes. For example, if a quarter note had one dot alongside itself, it would get one and a half beats. Therefore n dots lengthen the note's or rest's original d duration to $d \times (2 - 2^{-n})$.

10



Multi-measure rest

Indicates the number of measures in a resting part without a change in meter, used to conserve space and to simplify notation. Also called "gathered rest" or "multi-bar rest".

Durations shorter than the 64th are rare but not unknown. 128th notes are used by Mozart and Beethoven; 256th notes occur in works of Vivaldi and even Beethoven. An extreme case is the Toccata Grande Cromatica by early-19th-century American composer Anthony Philip Heinrich, which uses note values as short as 2,048ths; however, the context shows clearly that these notes have one beam more than intended, so they should really be 1,024th notes.

The name of very short notes can be found with this formula: $Name = 2^{(\text{number of flags on note} + 2)}_{th}$ note.

Breaks



Breath mark

In a score, this symbol tells the performer or singer to take a breath (or make a slight pause for non-wind instruments). This pause usually does not affect the overall tempo. For bowed instruments, it indicates to lift the bow and play the next note with a downward (or upward, if marked) bow.



Caesura

Indicates a brief, silent pause, during which time is not counted. In ensemble playing, time resumes when conductor or leader indicates.

Accidentals and key signatures

Main articles: Accidental (music) and Key signature

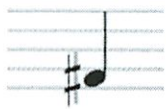
Common accidentals

Accidentals modify the pitch of the notes that follow them on the same staff position within a measure, unless cancelled by an additional accidental.

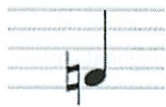


Flat

Lowers the pitch of a note by one semitone.

**Sharp**

Raises the pitch of a note by one semitone.

**Natural**

Cancels a previous accidental, or modifies the pitch of a sharp or flat as defined by the prevailing key signature (such as F-sharp in the key of G major, for example).

**Double flat**

Lowers the pitch of a note by two chromatic semitones. Usually used when the note to be modified is already flatted by the key signature.

**Double sharp**

Raises the pitch of a note by two chromatic semitones. Usually used when the note to be modified is already sharped by the key signature.

Key signatures

Key signatures define the prevailing key of the music that follows, thus avoiding the use of accidentals for many notes. If no key signature appears, the key is assumed to be C major/A minor, but can also signify a neutral key, employing individual accidentals as required for each note. The key signature examples shown here are described as they would appear on a *treble* staff.

**Flat key signature**

Lowers by a semitone the pitch of notes on the corresponding line or space, and all octaves thereof, thus defining the prevailing major or minor key. Different keys are defined by the number of flats in the key signature, starting with the leftmost, i.e., B ♭, and proceeding to the right; for example, if only the first two flats are used, the key is B ♭ major/G minor, and all B's and E's are "flattened", i.e. lowered to B ♭ and E ♭.

**Sharp key signature**

Raises by a semitone the pitch of notes on the corresponding line or space, and all octaves thereof, thus defining the prevailing major or minor key. Different keys are defined by the number of sharps in the key signature, also proceeding from left to right; for example, if only the first four sharps are used, the key is E major/C♯ minor, and the corresponding pitches are raised.

Quarter tones

Quarter-tone notation in Western music is not standardized. A common notation involves writing the fraction $1/4$ next to an arrow pointing up or down. Below are examples of an alternative notation:

**Demiflat**

Lowers the pitch of a note by one quarter tone. (Another notation for the demiflat is a flat with a diagonal slash through its stem. In systems where pitches are divided into intervals smaller than a quarter tone, the slashed flat represents a lower note than the reversed flat.)

**Flat-and-a-half (sesquiflat)**

Lowers the pitch of a note by three quarter tones.

**Demisharp**

Raises the pitch of a note by one quarter tone.

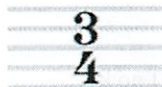
**Sharp-and-a-half**

Raises the pitch of a note by three quarter tones. Occasionally represented with two vertical and three diagonal bars instead.

Time signatures

Main article: Time signature

Time signatures define the meter of the music. Music is "marked off" in uniform sections called bars or measures, and time signatures establish the number of beats in each. This is not necessarily intended to indicate which beats are emphasized, however. A time signature that conveys information about the way the piece actually sounds is thus chosen. Time signatures tend to suggest, but only *suggest*, prevailing groupings of beats or pulses.

Specific time

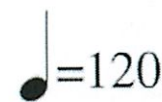
The bottom number represents the note value of the basic pulse of the music (in this case the 4 represents the crotchet or quarter-note). The top number indicates how many of these note values appear in each measure. This example announces that each measure is the equivalent length of three crotchets (quarter-notes). You would pronounce this as "Three Four Time", and was referred to as a "perfect" time.

Common time

This symbol is a throwback to sixteenth century rhythmic notation, when it represented 2/4, or "imperfect time". Today it represents 4/4.

Alla breve or Cut time

This symbol represents 2/2 time, indicating two minim (or half-note) beats per measure. Here, a crotchet (or quarter note) would get half a beat.

Metronome mark

Written at the start of a score, and at any significant change of tempo, this symbol precisely defines the tempo of the music by assigning absolute durations to all note values within the score. In this particular example, the performer is told that 120 crotchets, or quarter notes, fit into one minute of time. Many publishers precede the marking with letters "M.M.", referring to Maelzel's Metronome.

Note relationships

Tie

Indicates that the two (or more) notes joined together are to be played as one note with the time values added together. To be a tie, the notes must be identical; that is, they must be on the same line or the same space; otherwise, it is a slur (see below).

Slur

Indicates that two or more notes are to be played in one physical stroke, one uninterrupted breath, or (on instruments with neither breath nor bow) connected into a phrase as if played in a single breath. In certain contexts, a slur may only indicate that the notes are to be played legato; in this case, rearticulation is permitted.

Slurs and ties are similar in appearance. A tie is distinguishable because it always joins exactly two immediately adjacent notes of the same pitch, whereas a slur may join any number of notes of varying pitches.



A *phrase mark* (or less commonly, *ligature*) is a mark that is visually identical to a slur, but connects a passage of music over several measures. A phrase mark indicates a musical phrase and may not necessarily require that the music be slurred. In vocal music, a phrase mark usually shows how each syllable in the lyrics is to be sung.



Glissando or Portamento

A continuous, unbroken glide from one note to the next that includes the pitches between. Some instruments, such as the trombone, timpani, non-fretted string instruments, electronic instruments, and the human voice can make this glide continuously (portamento), while other instruments such as the piano or mallet instruments will blur the discrete pitches between the start and end notes to mimic a continuous slide (glissando).



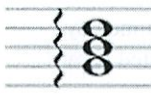
Tuplet

A number of notes of irregular duration are performed within the duration of a given number of notes of regular time value; e.g., five notes played in the normal duration of four notes; seven notes played in the normal duration of two; three notes played in the normal duration of four. Tuplets are named according to the number of irregular notes; e.g., duplets, triplets, quadruplets, etc.



Chord

Several notes sounded simultaneously ("solid" or "block"), or in succession ("broken"). Two-note chords are called **dyad**; three-note chords are called **triads**. A chord may contain any number of notes.



Arpeggiated chord

A chord with notes played in rapid succession, usually ascending, each note being sustained as the others are played. Also called a "broken chord".

Dynamics

Main article: Dynamics (music)

Dynamics are indicators of the relative intensity or volume of a musical line.

ppp

Pianississimo

Extremely soft. Very infrequently does one see softer dynamics than this, which are specified with additional *ps*.

pp

Pianissimo

Very soft. Usually the softest indication in a piece of music, though softer dynamics are often specified with additional *ps*.

p

Piano

Soft. Usually the most often used indication.

mp

Mezzo piano

Literally, half as soft as *piano*.

mf

Mezzo forte

Similarly, half as loud as *forte*. If no dynamic appears, *mezzo-forte* is assumed to be the prevailing dynamic level.

f

Forte

Loud. Used as often as *piano* to indicate contrast.

ff

Fortissimo

Very loud. Usually the loudest indication in a piece, though louder dynamics are often specified with additional *fs* (such as fortississimo - seen below).

fff

Fortississimo

Extremely loud. Very infrequently does one see louder dynamics than this, which are specified with additional *fs*.

sfz

Sforzando

Literally "forced", denotes an abrupt, fierce accent on a single sound or chord. When written out in full, it applies to the sequence of sounds or chords under or over which it is placed.

**Crescendo**

A gradual increase in volume.

Can be extended under many notes to indicate that the volume steadily increases during the passage.

**Diminuendo**

Also **decrescendo**

A gradual decrease in volume. Can be extended in the same manner as crescendo.

Other commonly used dynamics build upon these values. For example "piano-pianissimo" (represented as '**ppp**' meaning so softly as to be almost inaudible, and forte-fortissimo, ('**fff**') meaning extremely loud. In some European countries, use of this dynamic has been virtually outlawed as endangering the hearing of the performers.^[2] A small "s" in front of the dynamic notations means "subito", and means that the dynamic is to be changed to the new notation rapidly. Subito is commonly used with sforzandos, but all other notations, most commonly as "sff" (subitofortissimo) or "spp" (subitopianissimo).

**Forte-piano**

A section of music in which the music should initially be played loudly (forte), then immediately softly (piano).

Another value that rarely appears is *niente*, which means 'nothing'. This may be used at the end of a diminuendo to indicate 'fade out to nothing'.

Articulation marks

Articulations (or accents) specify how individual notes are to be performed within a phrase or passage. They can be fine-tuned by combining more than one such symbol over or under a note. They may also appear in conjunction with phrasing marks listed above.

**Staccato**

This indicates that the note is to be played shorter than notated, usually half the value, the rest of the metric value is then silent.

Staccato marks may appear on notes of any value, shortening their performed duration without speeding the music itself.

**Staccatissimo**

Indicates a longer silence after the note (as described above), making the note very short. Usually applied to quarter notes or shorter. (In the past, this marking's meaning was more ambiguous: it sometimes was used interchangeably with staccato, and sometimes indicated an accent and not staccato. These usages are now almost defunct, but still appear in some scores.)

**Accent**

The note is played louder or with a harder attack than surrounding unaccented notes. May appear on notes of any duration.

**Tenuto**

This symbol has several meanings: It may indicate that a note be played for its full value, or slightly longer; it may indicate a slight dynamic emphasis; or it may indicate a separate attack on a note. It may be combined with a staccato dot to indicate a slight detachment ("portato" or "mezzo staccato").

**Marcato**

The note is played somewhat louder or more forcefully than a note with a regular accent mark (open horizontal wedge).



Left-hand pizzicato or Stopped note

A note on a stringed instrument where the string is plucked with the left hand (the hand that usually stops the strings) rather than bowed. On the horn, this accent indicates a "stopped note" (a note played with the stopping hand shoved further into the bell of the horn). |-



Snap pizzicato

On a stringed instrument, a note played by stretching a string away from the frame of the instrument and letting it go, making it "snap" against the frame. Also known as a Bartók pizzicato.



Natural harmonic or Open note

On a stringed instrument, denotes that a natural harmonic (also called *flageolet*) is to be played. On a valved brass instrument, denotes that the note is to be played "open" (without lowering any valve, or without mute). In organ music, this denotes that a pedal note is to be played with the heel.



Fermata (Pause)

An indefinitely-sustained note, chord, or rest. Usually appears over all parts at the same metrical location in a piece, to show a halt in tempo. It can be placed above or below the note.



Up bow or Sull'arco

On a bowed string instrument, the note is played while drawing the bow upward. On a plucked string instrument played with a plectrum or pick (such as a guitar played pickstyle or a mandolin), the note is played with an upstroke. In organ notation, this marking indicates to play the pedal note with the toe.



Down bow or Giù arco

Like *sull'arco*, except the bow is drawn downward. On a plucked string instrument played with a plectrum or pick (such as a guitar played pickstyle or a mandolin), the note is played with a downstroke. Also note in organ notation, this marking indicates to play the pedal note with the heel.

Ornaments

Ornaments modify the pitch pattern of individual notes.



Trill

A rapid alternation between the specified note and the next higher note (according to key signature) within its duration. Also called a "shake." When followed by a wavy horizontal line, this symbol indicates an extended, or running, trill. Trills can begin on either the specified root note or the upper auxiliary note, though the latter is more prevalent in modern performances.



Mordent

Rapidly play the principal note, the next higher note (according to key signature) then return to the principal note for the remaining duration. In most music, the mordent begins on the auxiliary note, and the alternation between the two notes may be extended.



Mordent (lower)

Rapidly play the principal note, the note below it, then return to the principal note for the remaining duration. In much music, the mordent begins on the auxiliary note, and the alternation between the two notes may be extended.



Turn

When placed directly above the note, the turn (also known as a *gruppetto*) indicates a sequence of upper auxiliary note, principal note, lower auxiliary note, and a return to the principal note. When placed to the right of the note, the principal note is played first, followed by the above pattern. By either placing a vertical line through the turn symbol or inverting it, it indicates the order of the auxiliary notes is to be reversed.



Appoggiatura

The first half of the principal note's duration has the pitch of the grace note (the first two-thirds if the principal note is a dotted note).



Acciaccatura

The acciaccatura is of very brief duration, as though brushed on the way to the principal note, which receives virtually all of its notated duration.

Octave signs



Ottava

The *8va* sign is placed *above* the staff (as shown) to indicate the passage is to be played one octave *higher*.

(The *8va* sign is placed *below* the staff to indicate the passage is to be played one octave *lower*.^{[3][4]})



Quindicesima

The *15ma* sign is placed *above* the staff (as shown) to indicate the passage is to be played two octaves *higher*.

(The *15ma* sign is placed *below* the staff to indicate the passage is to be played two octaves *lower*.)

8va and *15ma* are sometimes abbreviated further to *8* and *15*. When they appear below the staff, the word *bassa* is sometimes added.

Repetition and codas

Tremolo

A rapidly-repeated note. If the tremolo is between two notes, then they are played in rapid alternation. The number of slashes through the stem (or number of diagonal bars between two notes) indicates the frequency at which the note is to be repeated (or alternated). As shown here, the note is to be repeated at a demisemiquaver (thirty-second note) rate.



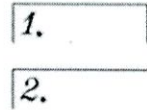
In percussion notation, tremolos are used to indicate rolls, diddles, and drags. Typically, a **single tremolo** line on a sufficiently short note (such as a sixteenth) is played as a drag, and a combination of three stem and tremolo lines indicates a double-stroke roll (or a single-stroke roll, in the case of timpani, mallet percussions and some untuned percussion instrument such as triangle and bass drum) for a period equivalent to the duration of the note. In other cases, the interpretation of tremolos is highly variable, and should be examined by the director and performers.



Repeat signs
Enclose a passage that is to be played more than once. If there is no left repeat sign, the right repeat sign sends the performer back to the start of the piece or the nearest double bar.



Simile marks
Denote that preceding groups of beats or measures are to be repeated. In the examples here, the first usually means to repeat the previous measure, and the second usually means to repeat the previous two measures.



Volta brackets (1st and 2nd endings, or 1st and 2nd time bars)
Denote that a repeated passage is to be played in different ways on different playings. (Can also have more than two endings (1st,2nd,3rd...n'th endings) by changing the number inside the bracket to the number repeated. (Note: More than two or three endings is very uncommon but can be found in some musical arrangements.))

D.C.

Da capo
Tells the performer to repeat playing of the music from its beginning. This is followed by *al fine*, which means to repeat to the word *fine* and stop, or *al coda*, which means repeat to the coda sign and then jump forward.

D.S.

Dal segno
Tells the performer to repeat playing of the music starting at the nearest *segno*. This is followed by *al fine* or *al coda* just as with *da capo*.



Segno
Mark used with *dal segno*.



Coda
Indicates a forward jump in the music to its ending passage, marked with the same sign. Only used after playing through a *D.S. al coda* or *D.C. al coda*.

Instrument-specific notation

Guitar

The guitar has a right-hand fingering notation system derived from the names of the fingers in Spanish or Latin. They are written above, below, or beside the note to which they are attached. (The little finger is rarely used in classical music.) They read as follows:

Symbol	Spanish	Latin	English
p	pulgar	pollex	thumb
i	índice	index	index
m	medio	media	middle
a	anular	anularis	ring
c, x, e, q, a	meñique	minimus	little

Piano

Pedal marks

These pedal marks appear in music for instruments with sustain pedals, such as the piano, vibraphone and chimes.



Engage pedal
Tells the player to put the sustain pedal down.



Release pedal
Tells the player to let the sustain pedal up.

Variable pedal mark



More accurately indicates the precise use of the sustain pedal. The extended lower line tells the player to keep the sustain pedal depressed for all notes below which it appears. The inverted "V" shape (Λ) indicates the pedal is to be momentarily released, then depressed again.

Other piano notation

m.d. / MD / **mano destra** (Italian)
r.H. / r.h. / **main droite** (French)
RH **rechte Hand** (German)
right hand (English)

m.s. / MS / **mano sinistra** (Italian)
m.g. / MG / **main gauche** (French)
l.H. / l.h. / LH **linke Hand** (German)
left hand (English)

Finger identifications:

1 = thumb
 2 = index
 3 = middle
 4 = ring
 5 = little

See also

- Graphic notation
- Music theory
- Glossary of musical terminology
- Eye movement in music reading
- Shape note

References

- ↑ U+007B left curly bracket (<http://www.decodeunicode.org/u+007B>) at decodeunicode.org; retrieved on May 3, 2009
- ↑ "No Fortissimo? Symphony Told to Keep It Down" (<http://www.nytimes.com/2008/04/20/arts/music/20noise.html>) by Sarah Lyall, *The New York Times* (20 April 2008)
- ↑ George Heussenstamm, *The Norton Manual of Music Notation* (New York and London: W. W. Norton & Company), p.16
- ↑ Anthony Donato, *Preparing Music Manuscript* (Englewood Cliffs, New Jersey: Prentice-Hall, Inc.), pp. 42-43

External links

- Comprehensive list of music symbols fonts (<http://www.music-notation.info/en/compmus/musicfonts.html>)
- Music theory & history (<http://www.dolmetsch.com/theoryintro.htm>) (Dolmetsch Online)
- Dictionary of musical symbols (<http://www.dolmetsch.com/musicalsymbols.htm>) (Dolmetsch Online)
- Sight reading tutorial with symbol variations (<http://www.music-mind.com/Music/indexlrm.HTM>) Amy Appleby

Retrieved from "http://en.wikipedia.org/w/index.php?title=List_of_musical_symbols&oldid=455630260"

Categories: Musical notation

- This page was last modified on 15 October 2011 at 02:30.
 - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
- Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

ABC notation

From Wikipedia, the free encyclopedia

(Redirected from Abc notation)

ABC notation is a shorthand form of musical notation that has been in use since at least the 19th century. In basic form it uses the letters A through G to represent the given notes, with other elements used to place added value on these - sharp, flat, the length of the note, key, ornamentation. Later, with computers becoming a major means of communication, others saw the possibilities of using this form of notation as an ASCII code that could facilitate the sharing of music online, also adding a new and simple language for software developers. In this later form it remains a language for notating music using the ASCII character set. The earlier ABC notation was built on, standardized and changed to better fit the keyboard and an ASCII character set by Chris Walshaw, with the help and input of others. Although now re-packaged in this form, the original ease of writing and reading, for memory jogs and for sharing tunes with others on a scrap of paper or a beer coaster remains, a simple and accessible form of music notation, not unlike others, such as tablature and Solfeggio. Originally designed for use with folk and traditional tunes of Western European origin (e.g. English, Irish, Scottish) which are typically single-voice melodies which can be written on a single staff in standard notation, the work of Chris Walshaw and others has opened this up with an increased list of characters and headers in a syntax that can also support metadata for each tune.^[1]

ABC Notation being ASCII-based, any text editor can be used to edit the music. Even so, there are now many ABC Notation software packages available that offer a wide variety of features, including the ability to read and process abc notation, including into midi files and as standard 'dotted' notation. Such software is readily available for most computer systems including Microsoft Windows, Unix/Linux, Macintosh, PalmOS, and web-based.^[2]

Later 3rd-party software packages have provided direct output (bypassing the TeX typesetter),^[3] and have extended the syntax to support lyrics aligned with notes,^[4] multi-voice and multi-staff notation,^[5] tablature,^[6] and MIDI.^[7]

toh can
use it w/ TeX

Contents

- 1 History of ASCII ABC Notation
 - 1.1 Standardization
- 2 Example
- 3 Collaborative abc
- 4 See also
- 5 References
- 6 External links
 - 6.1 Documentation
 - 6.2 Software
 - 6.3 ABC Search Engines
 - 6.4 ABC Tune Collections

History of ASCII ABC Notation

In the 1980s Chris Walshaw began writing out fragments of folk/traditional tunes using letters to represent the notes before he learned standard Western music notation. Later he began using MusicTeX to notate French bagpipe music. To reduce the tedium of writing the MusicTeX code, he wrote a front-end for generating the TeX commands, which by 1993 evolved into the abc2mtex program.^[8] For more details see Chris' short history of abc (<http://abcnotation.com/history.html>), and John Chambers' chronology of ABC notation and software (http://trillian.mit.edu/~jc/music/abc/doc/ABC_tut_History.html).

Standardization

The closest thing to an official standard is the (officially labelled "draft") "ABC Music standard 2.0".^[9] It is a textual description of abc syntax and was grown from the 1996 user guide of version 1.6 of Chris Walshaw's original abc2mtex (<http://abcnotation.com/abc2mtex/>) program. In 1997, Henrik Norbeck published a BNF description of the abc v1.6 standard (<http://www.norbeck.nu/abc/abcbnf.htm>).^[10]

In 1997, Steve Allen registered the text/vnd.abc MIME media type with the IANA.^[11] But registration as a top level MIME type would require a full-blown RFC.^[12] In 2006 Phil Taylor reported that quite a few websites still serve abc files as text/plain.^[13]

Who cares
 In 1999, Chris Walshaw started work on a new version of the abc specification to standardize the extensions that had been developed in various 3rd-party tools. After much discussion on the ABC users mailing list, a draft standard - version 1.7.6] was eventually produced in August 2000, but was never officially released.^[14] At that point Chris stepped away for several years from actively developing abc.^[15]

Guido Gonzato later compiled a new version of the specification and published a draft of version 2.0. This specification is now maintained by Irwin Oppenheim. Henrik Norbeck has also published a corresponding BNF specification.^[16]

After a surge of renewed interest in clarifying some ambiguities in the 2.0 draft and suggestions for new features, serious discussion of a new (and official) standard resumed in 2011 and is currently (July 2011) ongoing on the abcusers mailing list. Chris Walshaw has gotten involved again and is coordinating this effort. The changes currently being discussed are reflected at <http://abcnotation.com/wiki/abc:standard:v2.1>

Example

The following is an example of the use of abc notation

```
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:G
GFG BAB | gfg gab | GFG BAB | d2A AFD |
GFG BAB | gfg gab | age edB |1 dBA AFD :|2 dBA ABd |
efe edB | dBA ABd | efe edB | gdB ABd |
efe edB | d2d def | gfe edB |1 dBA ABd :|2 dBA AFD |]
```

Lines in the first part of the tune notation, beginning with a letter followed by a colon, indicate various aspects of the tune such as the index, when there are more than one tune in a file (X:), the title (T:), the time signature (M:), the

default note length (L:), the type of tune (R:) and the key (K:). Lines following the key designation represent the tune. This example can be translated into traditional music notation using one of the abc conversion tools. For example, abcm2ps software produces output that looks like the following image:

More examples can be found on Chris Walshaw's abc examples page (<http://abcnotation.com/examples.html>) which displays almost extensively abc basic features, except rest that are noted "z".

Collaborative abc

Recently abc has been implemented as a means of composing and editing music collections in collaborative environments. Several examples of Wiki environments that have been adapted to use abc are:

- AbcMusic (<http://www.mediawiki.org/wiki/Extension:AbcMusic>) , plugin for MediaWiki. Note: This implementation uses GNU LilyPond as the underlying rendering engine. LilyPond comes packaged with a script, abc2ly, that converts ABC notation to LilyPond. The extension calls abc2ly then LilyPond.
- MusicWiki (<http://www.soe.ucsc.edu/cgi-bin/cgiwrap/nwhitehe/moin.cgi/MusicWiki>) , a Python plugin implementation for MoinMoin
- AbcMusic (<http://www.pmwiki.org/wiki/Cookbook/AbcMusic>) for displaying abc notation in PmWiki
- Traditional Music Wiki (<http://music.gordfisch.net/tradmusic/>) A collaborative source for traditional music using a tailored version of the AbcMusic (<http://www.pmwiki.org/wiki/Cookbook/AbcMusic>) plugin
- abc plugin (<http://wiki.splitbrain.org/plugin:abc>) for displaying abc notation in DokuWiki. This plugin uses Jef Moine's abcm2ps (<http://moinejf.free.fr/>) package as the rendering engine. It optionally uses abc2midi (available from the ABC Plus Project (<http://abcplus.sourceforge.net/>)) to produce midi audio output.
- abcjs plugin (<http://code.google.com/p/abcjs>) for displaying abc notation on any web page. This allows abc to be stored as text on the server and rendered client-side.
- Zap's abc (<http://www.appbrain.com/app/se.petersson.abc>) an Android application combining abcm2ps, abc2midi and a bit of abc4j into a tool for composing in your pocket. Online help

Team Contract

A team contract is an agreement between you and your teammates about how your team will operate -- a set of conventions that you plan to abide by. The questions below will help you consider what might go into your team contract. You should also think back to good or bad aspects of team project experiences you've already had.

Your contract doesn't have to answer all the questions below, but must answer the boldfaced questions. Focus on the issues that your team considers most important.

Goals

- What are the goals of the team?
- What are your personal goals for this assignment?
- What kind of obstacles might you encounter in reaching your goals?
- What happens if all of you decide you want to get an A grade, but because of time constraints, one person decides that a B will be acceptable?
- Is it acceptable for one or two team members to do more work than the others in order to get the team an A?

sure!

Meeting Norms

- Do you have a preference for when meetings will be held? Do you have a preference for where they should be held?
- **How will you use the in class time?**
- How often do you think the team will need to meet outside of class? How long do you anticipate meetings will be?
- Will it be okay for team members to eat during meetings?
- How will you record and distribute the minutes and action lists produced by each meeting?

silly...

Work Norms

- How much time per week do you anticipate it will take to make the project successful?
- How will work be distributed?
- How will deadlines be set?
- How will you decide who should do which tasks?
- **Where will you record who is responsible for which tasks?**
- What will happen if someone does not follow through on a commitment (e.g., missing a deadline, not showing up to meetings)?
- How will the work be reviewed?
- What happens if people have different opinions on the quality of the work?
- What will you do if one or more team members are not doing their share of the work?
- How will you deal with different work habits of individual team members (e.g., some people like to get assignments done as early as possible; others like to work under the pressure of a deadline)?

Decision Making

- Do you need consensus (100% approval of all team members) before making a decision?
- What will you do if one of you fixates on a particular idea?

10/16

6.005
Project ABC
Meeting 1

Plaz

- do the project
- do to best of ability w/o hindering other classes
- stuff takes too long
 - unclear instructions
 - buggy frame code

Michael: has not used SVN before

- ~~Don't~~ Don't commit code that breaks compiling
- If you really want an A - put in the effort

Can meet in W20

Have skype open - skype room

Google doc what actively working on

In class time: discuss what worked on and then work
Out of class if needed

If all 3 of us are here - no minutes!

②

hrs per week: as much as needed

Distributed:

- Just talk for Tue
- no good idea

~~Wk~~ For Tue

- Warmup
- Individual: No

- transcribe

- best for parser person JWang

- Plaz: Junit tests

Michael - formal stuff Tue at 5'

- weakness

Plaz - will review

Record task at meeting

Miss commitment.

Take time before deadline to review

- leave 3 days before - so on 24th 1st draft of code

③

They fix it

All Plaz + Jon get done early

- ~~Michael~~
Mike } = wish get done early

- so we will him w/ soft deadlines

Think it would make another angry - then ask
have a discussion

~~Highest priority; written~~

Focus on getting assignment done

Do team Contact list

LATEX

w/ digital signature :)

What is ticks per min

I'm bad at musical notation!

First piece has $\frac{1}{16}$

↳ so need 4 ticks per qt note






Hard to write test first!

So the line w/ a dot is half longer

Getting faster.

having to add lines is annoying

Where does an octave start/end?


Oh middle C is     

It defaults to middle C!

Tuplets:

- ~~main~~ here is confusing...
description

(2)

So  is 3 notes in half note time

So $\frac{2}{3}$ of normal duration

So for me 8 is normal
— divide into 3

grrr...

So these are 3 eighth notes

So $\frac{1}{4}$ is 4

$\frac{4}{3}$ five slots each

So can only do ints

— hmmm...

Do later

— lots unanswered q on piazza

How put a rest in?

So just leave fire I guess

(Songwriting by PC finally handy!)

Return test

I asked TA - have ~~quarter~~ divisible #'s for triplets
like 12 ticks per quarter note

Group members not here!

So does this make sense

Now I have 4 ~~notes~~ ticks per quarter
- So down to 16th notes

I could multiply everything by 3

Then could I divide 2 eighth notes in 3

2 eighth notes is now 4

would be $4 \cdot 3 = 12$

Then divide by 3 = 4

This seems annoying to have to do

Other one has 3 quarter notes triplet

Can do less than 12?

Currently 2 quarter notes = 8

• 2

= $16 / 3 = 5.33 \text{ } \emptyset$

• 3

= $24 / 3 = 8 \checkmark$ do this too

② So multiply everything by 3

So a 2 qt notes was 8

$$8 \cdot 3 = 24 \text{ now}$$

2 eight notes = 4 \cdot 3 12 total ✓ so is right

(class path not working on laptop, grr!

6.005 Mtg

10/18

Lexer

Parser built by Juany

- he tested w/ main method

- I need to do JUnit

- Class path Eclipse

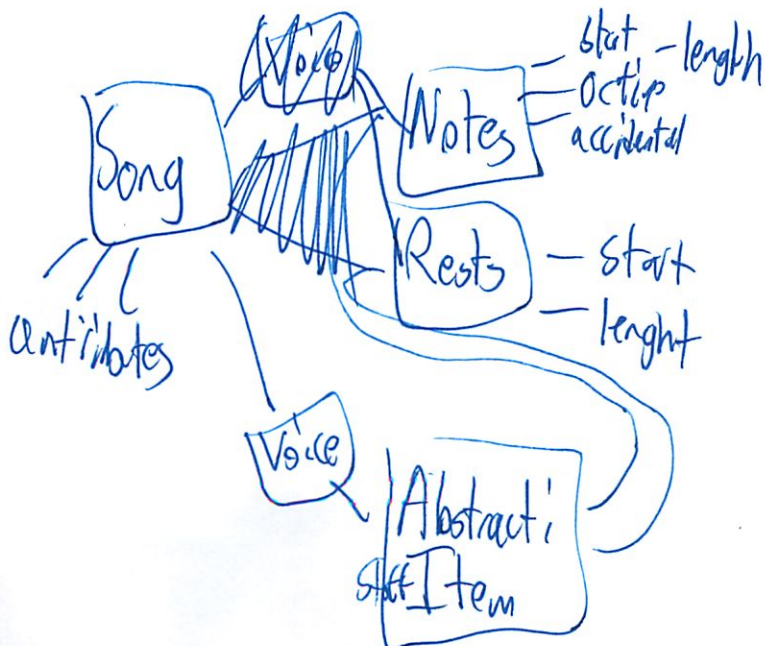
Diagrams

- Song class

- Meter $4 \leftarrow \# \text{ notes per measure}$
 $4 \leftarrow \text{inv of length of a note}$

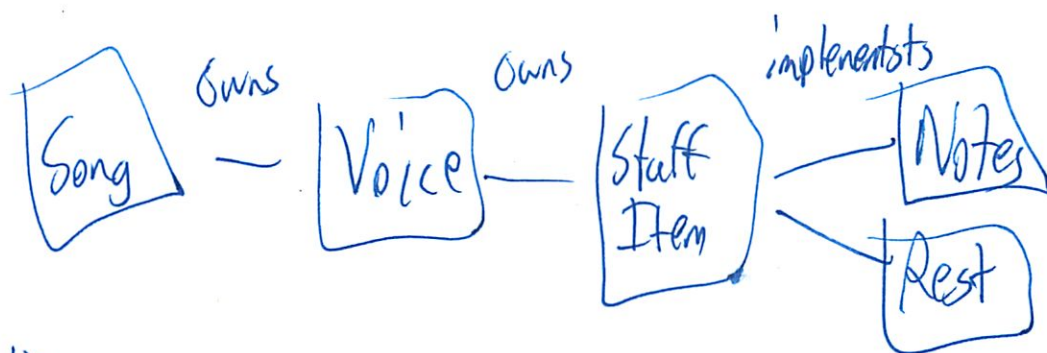
Data types

- need to use visitor



\leftarrow can store as 2

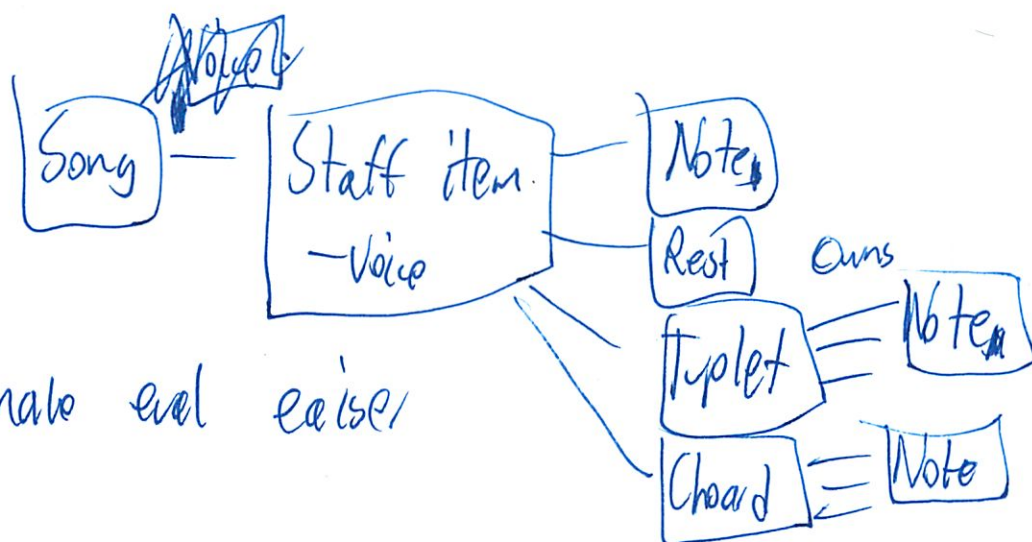
②



Evaluator assigns ticks

Should evaluator check each voice

Or



To make eval easier

Should have seq of inputs

- eval will calc the ticks

Can voices start in the middle?

I say no

Lets put voice ^{back} in



③

Should Tuple be different?

Verbose proper way

Mutable?

State Machine

~~Parser~~ Evaluator? Parser?

Order

Current token
position

- what voice you are on

Review them

Mid header is crazy
i put parts in quote

Mult_i is multiplication factor

Pitch is w/ commas + apostrophes

Emailed to ash

Added staff item diagram

Do I need anything else?

10/19

^{6.005}
I add as you go to voices

Can choose mutability

- but mutability is scary
- getters + setters should be used
- no rep exposure

Parsing is hardest part

Forgot repeat

Jon - Building parser

Plaz - will do tests

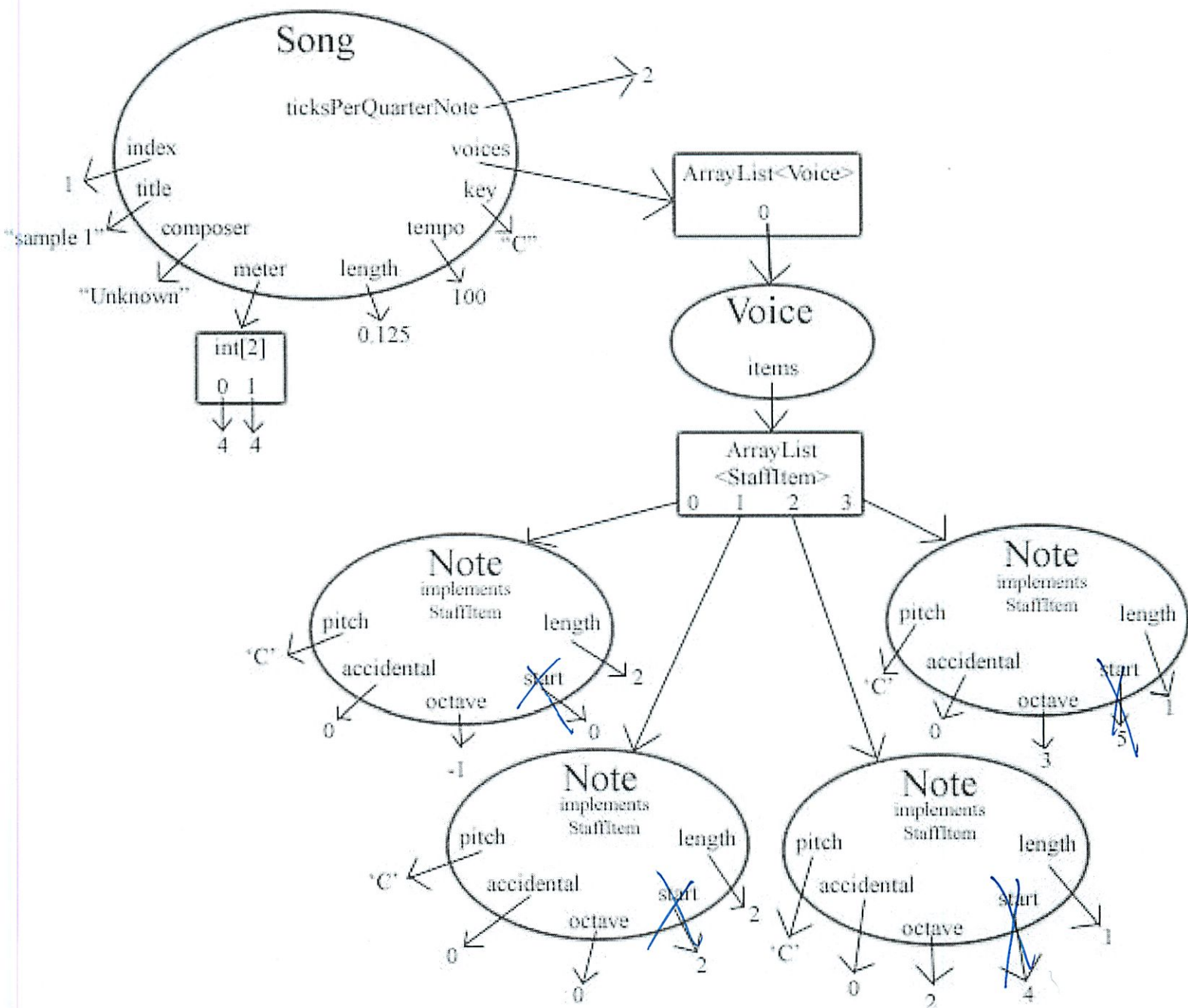
Mike - off

- no time to work on weekend
-

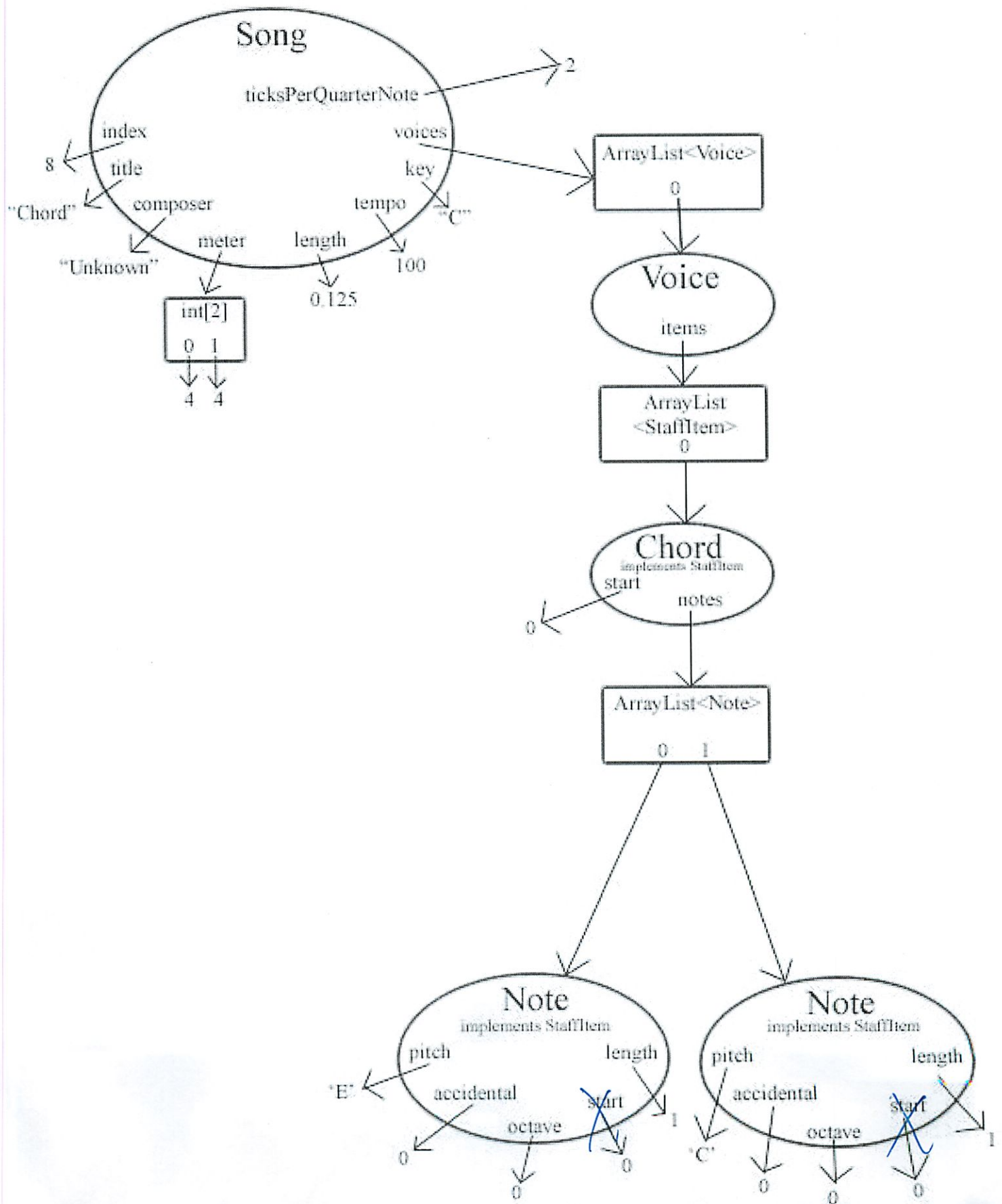
Next meeting Mon @ Lecture time, room

Snapshot Diagrams

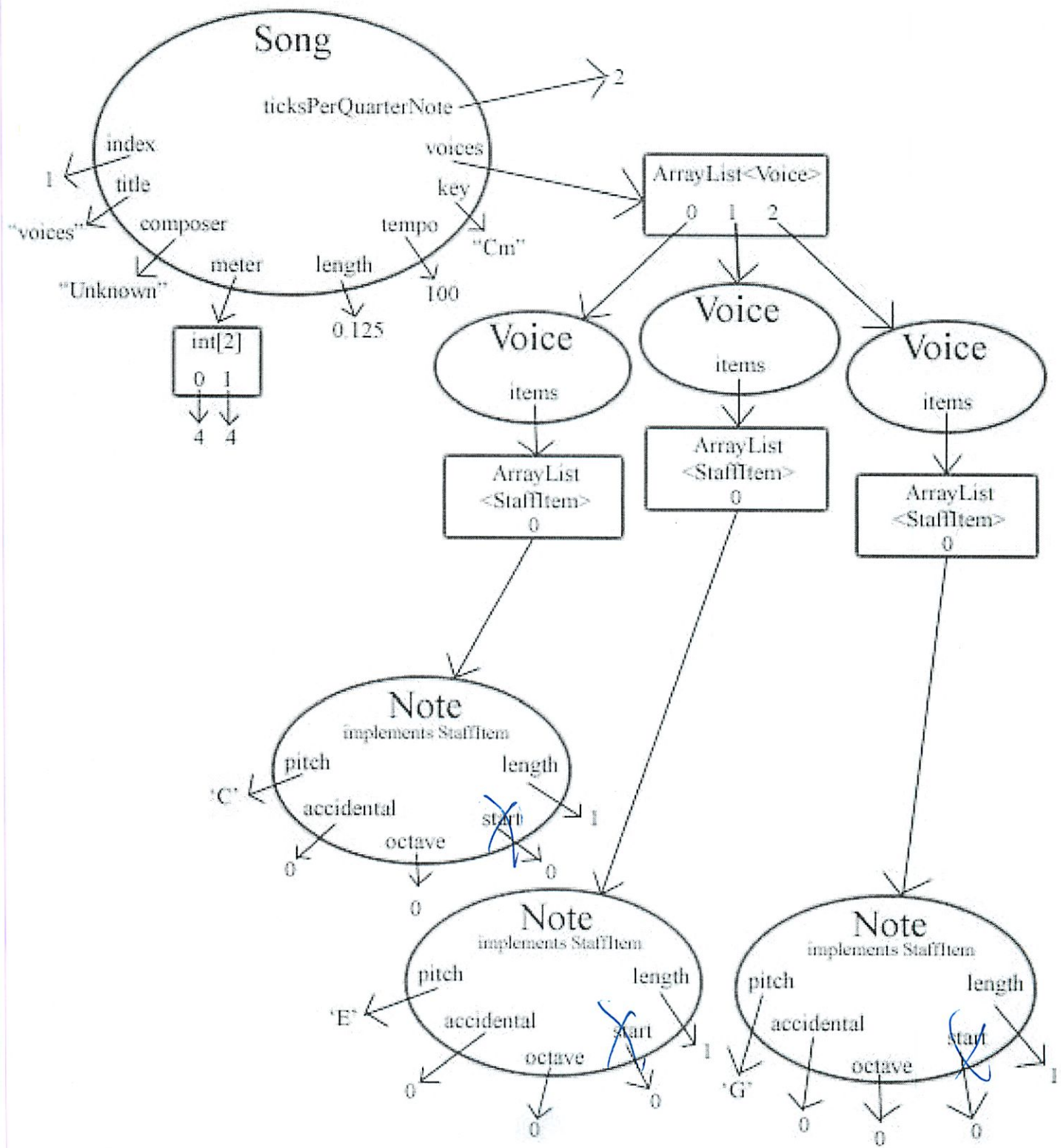
Example 1



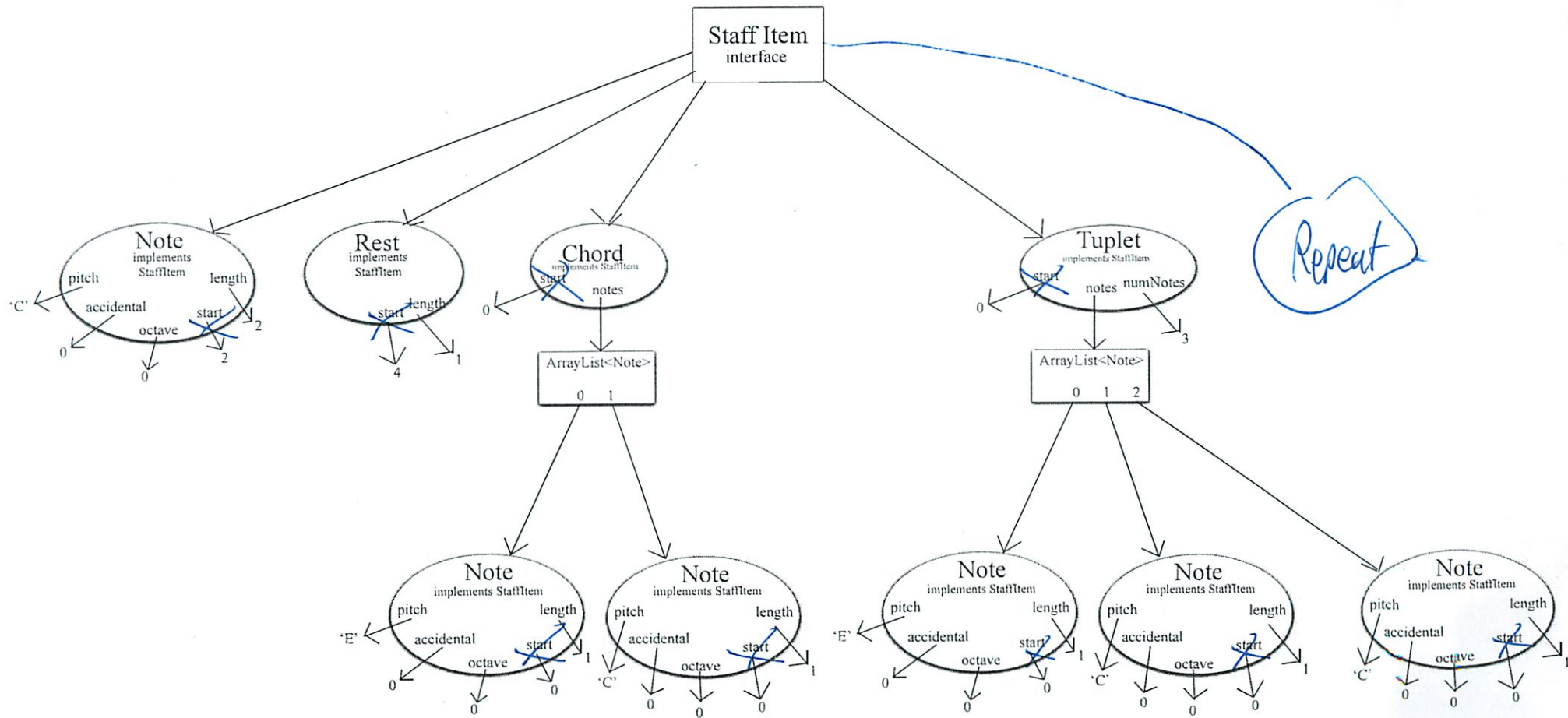
Example 2



Example 3



Data Types




```

Song ::= Header (Voice* | Items)
Header ::= X T MidHeader K
MidHeader ::= (C (L (M Q?|Q M?))?
               | M (L Q?|Q L?))?
               | Q (L M?|M L?))?
               )?
               | L (C (M Q?|Q M?))?
               | M (C Q?|Q C?))?
               | Q (C M?|M C?))?
               )?
               | M (C (L Q?|Q L?))?
               | L (C Q?|Q C?))?
               | Q (C L?|L C?))?
               )?
               | Q (C (L M?|M L?))?
               | L (C M?|M C?))?
               | M (C L?|L C?))?
               )?
               )?
//MidHeader = C?, L?, M? and Q? in any order

Voice ::= V: String Items
Items ::= (Note | Rest | Chord | Tuplet)*
X ::= X: Int
T ::= T: String
C ::= C: String
L ::= L: Int (/ Int)?
M ::= M: Int / Int
Q ::= Q: Int
K ::= K: [^_]? [ABCDEFG] m?
Note ::= (^ ^? | = | _ _)? [ABCDEFGGabcdeg] (,* | '*') Mult?
//Note = Accidental (if any), letter, octave shift (if any), multiplicative factor (if
any)
Rest ::= z Mult? check if 0
Chord ::= '[' Note* ']'
Tuplet ::= '(' Int Note*
Mult ::= Int? (/ Int)?
Int ::= [0123456789]+
String ::= []* //any number of any character

```

Song: Info about a song and the music required to play it

- >index: the index number of the song
- >title: the title of the song
- >composer: the name of the person who composed the song
- >meter: the time signature of the song, as an array containing the numerator at index 0 and the denominator at index 1
- >length: the length of a default note, as a fraction of a measure
- >tempo: the number of beats per minute in the song
- >key: the key of the song
- >ticksPerQuarterNote: the number of ticks that represent a quarter note in the sequence player
- >voices: a list of all the voices that play during the song, as well as the music they play

Voice: a voice that plays during the song, as well as the music it plays

- >items: the list of notes, rests, chords, and tuples that the voice plays

StaffItem: Interface

Note (implements StaffItem): A single note within a voice

- >pitch: the letter representing the pitch of the note within an octave
- >accidental: the sharp or flat applied to the note, if any (-1 for flat, 0 for no accidental, 1 for sharp)
- >octave: the octave in which the note is located (represented as a number of octaves above the middle octave- a negative number if the octave is lower than the middle octave)
- >start: the time at which the note begins, in ticks since the beginning of the song
- >length: the number of ticks for which the note lasts

Rest (implements StaffItem): A single rest (pause in playing) within a voice

- >start: the time at which the rest begins, in ticks since the beginning of the song
- >length: the number of ticks for which the rest lasts

Chord (implements StaffItem): A series of notes played simultaneously by the same voice

- >start: the time at which the chord begins, in ticks since the beginning of the song
- >notes: the list of notes played in the chord

Tuplet (implements StaffItem): A series of notes played sequentially for different lengths from what their notation would otherwise imply

- >start: the time at which the tuple begins, in ticks since the beginning of the song
- >numNotes: the number of notes in the tuple
- >notes: the list of notes played in the tuple



-Add

10/21

6.005 Work Building Test Cases

So test Lexer, Parser, Evaluator

and each part

↳ check Rep

do that later

do this first

Lexer

input string
output tokens

empty

all the types

header

Voices

note

note length

accidental

rest

chord

triplat

Double

Bar line

bar line

Repeat

white space

②

Why do we need a enum for this
↳ some rule for something

Ah he did it with nice rules

For each line

for each position

for each rule

if it matches

~~move~~ add token

Then set position at end

So tuple is only a part one

Reset resets from start

has Next

next

So big thing is trying each type

Get a basic one

- need to see what to call

③

How to get Lexer on normal ~~file~~ text
- load in other file?

feed it file handler?

Write temp file?

Got lot (empty) test set up!

Made a bunch more

I still think Lexer stupid

Oh tokens not asserting Equals right

- always ~~the~~ ~~the~~ returns true

- No it fails sometimes

Urg Jon changed it!

Oct are must come first!

Lexer done

4

Parser

Same as Lexer

but run through parser

TODO so handy!

∴ test parse w/ To String
- or check each object!

I think I can

- Juang built! yeah

Writing tests by checking output...

(I should review Visitor to make sure I understand)

Did one myself :-)

New lines + spaces arbitrary
too bad

5

Evaluator

How to test?

SP, Player to String?

It does add to send player

Every so often file locked... " grr

Oh Jwang put in a break point

Done except test 3

Jwang says that is working too

Not seeming to work - restart Eclipse

Still no

- Oh do it later...

6.005
Project

10/24

Jwang finished implementation

— even went above + beyond w/ UI

I need to relook at tests

— Michael will help

Michael will update Design files

Due Thur night @ midnight

Reflections due Sat @ midnight

— private

— will do Fri

Michael will do drafts by Wed 9:30

— Then we review

6-005
Project

10/27

Finish 3 Todos

6.005
ULO Concurrency

10/26

Lectures restart today

- ☐ Shared memory + message passing
- ☐ processes + threads
- ☐ time slicing
- ☐ ~~new~~ race conditions
- ☐ deadlocks

Proj 1 due Thur

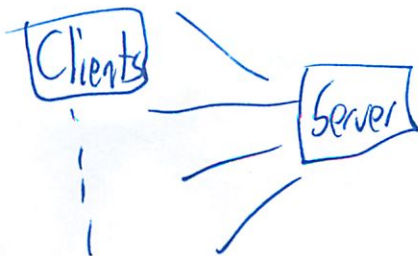
PS 5 out Sat, due next Thur

Recitations tomorrow

When this was 6.170 it wasn't in here

Concurrency is hard

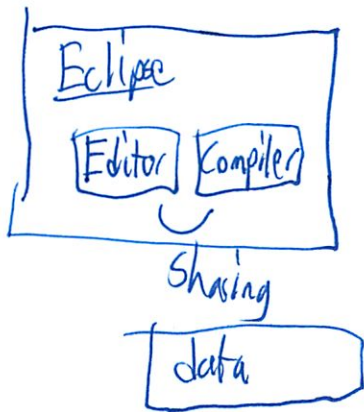
Multiple processes at same time that must interact
Fact of modern programming



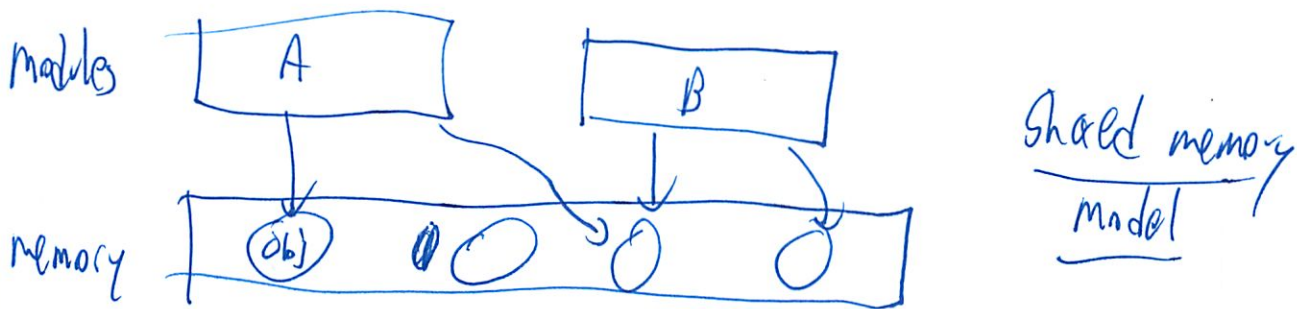
(2)

Many new systems are being built this way
All web apps - need to think about concurrency

Or 2 apps/modules working at same time



So have some shared memory
- modules are trying to access + change

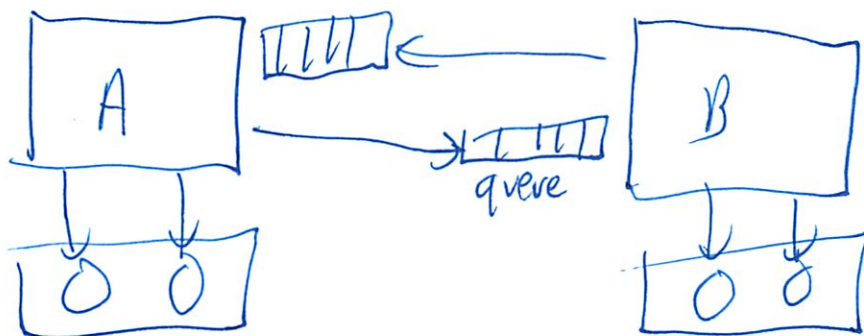


if immutable - would not care

So key problem is when objects are mutable

(3)

Also message passing model



Each has
own memory

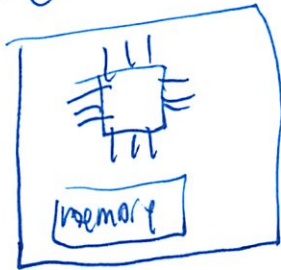
- how webserver / browsers work

Examples

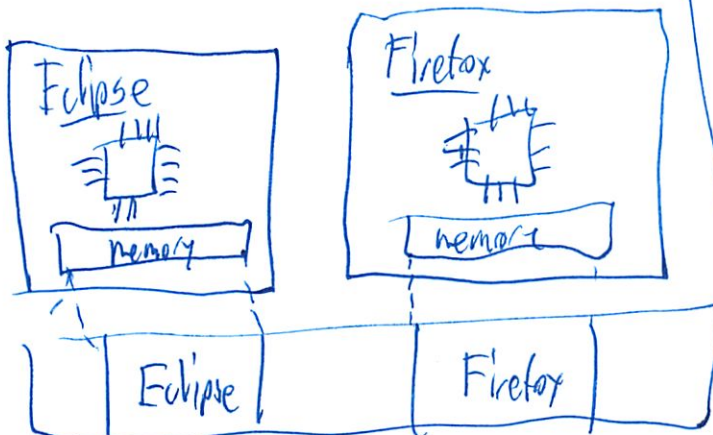
- Dropbox - shared memory
- SVN - message passing
- Emailing others - message passing
- Email Act - shared memory

Process

- Like a virtual computer
- gets simulation of being alone



- if looked at clock would realize not always running
- sharing time



each has own piece



↑
Suspended and resume
program doesn't know it was frozen
↳ Unless it looks at clock

Thread

next pg

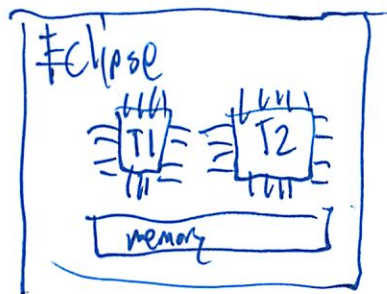
5

Newer PCs have multicore processors

(ah the days when doing an AV scan slowed your PC)
(~ well still kinda w/ HDD access)

Thread

Like a virtual processor within virtual computer / process



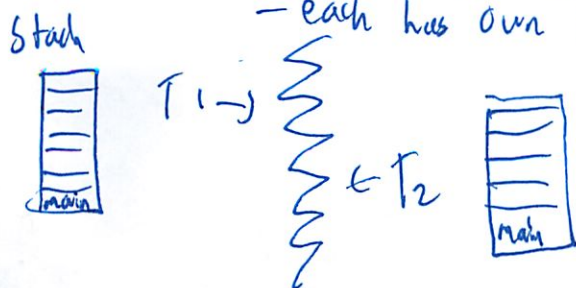
T_1, T_2 can be executing diff code
But ~~some~~ share same memory

Are features that let you break this

- can share memory b/w processors
- or give threads their own memory
- create sockets (message passing pipes) b/w processes

Thread = locus of control

- place where execution is
- each has own stack



6

Can stack overflow

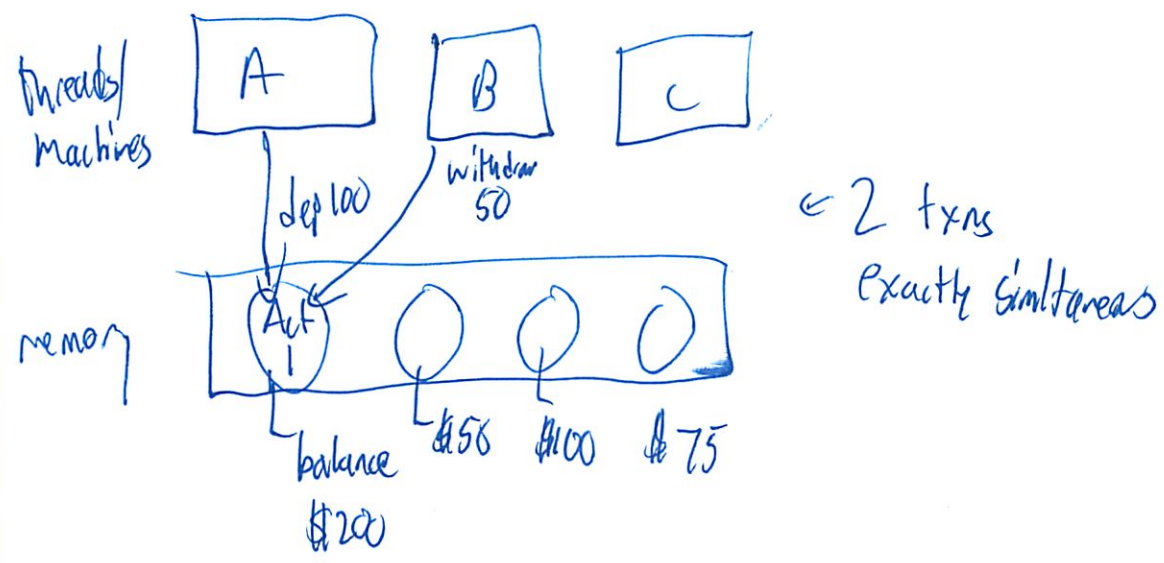
↳ usually because of ∞ loop

Can see stack in Eclipse

Problems (today's goal: scare us)

example Cash machines w/ shared memory

- often use bank examples



How do the operations break down inside computer?

Want ans to be \$150

But primitive ops

$$\begin{cases} t = \text{act1.balance} \\ t = t + 100 \\ \text{act1.balance} = t \end{cases}$$

②

withdraw

$$\left[\begin{array}{l} t = \text{actl.bal} \\ t = t - 50 \\ \text{actl.bal} = t \end{array} \right.$$

All 3 of 1 operation must happen together

If they break up - bad!

Thread 1
 $t = \text{actl.bal}$
($t = 100$)

$t = t + 100$
($t = 200$)

$\text{actl.bal} = t$
($\text{bal} = 200$)

Thread 2

$t = \text{actl.bal}$
($t = 100$)

~~$t =$~~ 50
(~~$t =$~~ 50)

$\text{actl.bal} = t$
($\text{bal} = 50$)

↑ called a race condition

②

Race condition - correctness depends on timing

The problem w/ this is that it does not happen always

- makes it ~~thicker~~ tricky to debug

- external environment matter

 - OS, processor, time of day, cosmic rays

Create threads in Java - next week

- ↳ don't do in PS 5. Use message passing

Example → he is trying to break it on purpose

Often don't see bugs when just you testing

- ↳ need some more users

You can't count on order items will occur

Processors might reorder commands

- ↳ Since order does not matter when running sequentially

Put threads, yield in to help debug
forces concurrency

⑨ Putting in prints / breakpoints change timing
So it will usually work

So 1. hard to ~~reproduce~~ reproduce

2. hard to put debug code in

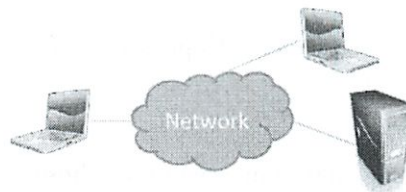
L10: Concurrency

Today

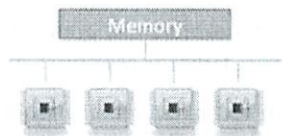
- Processes & threads
- Time slicing
- Message passing & shared memory
- Race conditions
- Deadlocks

Concurrency

- Concurrency is everywhere, whether we like it or not
 - Multiple computers in a network



- Multiple applications running on one computer
- Multiple processors in a computer (today, often multiple **processor cores** on a single chip)

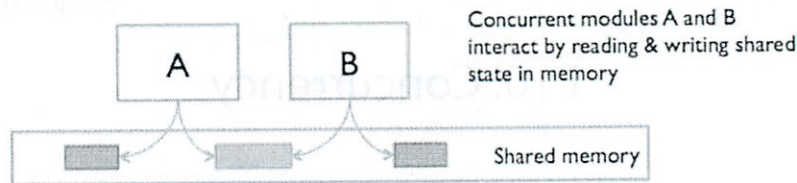


- Concurrency is essential in modern programming
 - Web sites must handle multiple simultaneous users
 - Mobile apps need to do some of their processing on servers (“in the cloud”)
 - Graphical user interfaces often require background work (e.g. Eclipse compiling your Java code while you’re editing it)
 - Processor clock speeds are no longer increasing – instead we’re getting more cores on each new generation of chips. So in the future, we’ll have to split up a computation into concurrent pieces in order to get it to run faster.

Two Models for Concurrent Programming

Shared Memory

- Analogy: two processors in a computer, sharing the same physical memory



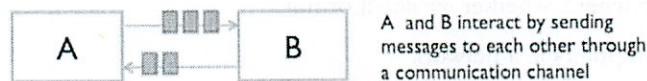
Other ways to think about this:

A and B might be two *threads* in a Java program (we'll explain what a thread is later)

A and B might be two programs running on a computer, sharing a common filesystem

Message Passing

- Analogy: two computers in a network, communicating by network connections



A and B might be a web browser and a web server – A opens a connection to B, asks for a web page, and B sends the web page data back to A.

A and B might be an IM client and server.

A and B might be two programs running on the same computer whose input and output have been connected by a pipe (like `ls | grep`).

Threads & Processes

Process

- A **process** is an instance of a running program that is isolated from other processes on the same machine (particularly for resources like memory)
- Tries to make the program feel like it has the whole machine to itself – like a **fresh computer** has been created, with fresh memory
- By default, processes have no shared memory (needs special effort)
- Automatically ready for message passing (standard input & output streams)

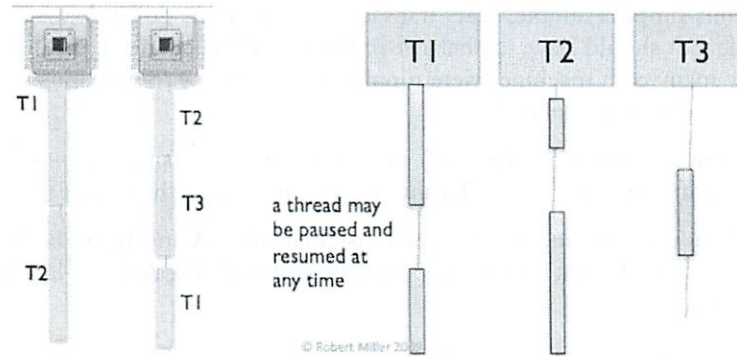
Thread

- A **thread** is a locus of control inside a running program (i.e. position in the code + stack, representing the current point in the computation)
- Simulates making a **fresh processor** inside the computer, running the same program and sharing the same memory as other threads in process
- Automatically ready for shared memory, because threads share all the memory in the process (needs special effort to get “thread-local” memory that’s private to the thread)
- Must set up message passing explicitly (e.g. by creating queues)

Time-slicing

How can I have many concurrent threads with only one or two processors in my computer?

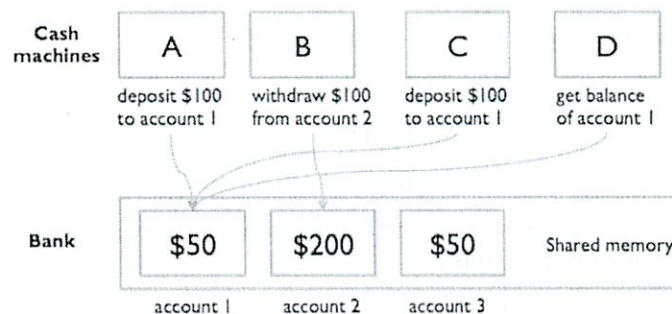
- When there are more threads than processors, concurrency is simulated by **time slicing** (processor switches between threads)
- Time slicing happens unpredictably and nondeterministically



A Shared Memory Example

Four customers using cash machines simultaneously

- Shared memory model – each cash machine reads and writes the account balance directly



```
// all the cash machines share a single bank account
private static int balance = 0;
```

```
private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

```
// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
```

```

private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
        deposit(); // put a dollar in
        withdraw(); // take it back out
    }
}

```

Throughout the day, each cash machine in our network is running `cashMachine()`, processing transactions. In this simple example, every transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged. So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed, we expect the account balance to still be 0.

But it's not. If more than one `cashMachine()` call is running at the same time – say, on separate processors in the same computer – then balance may *not* be zero at the end of the day. Why not?

Here's one thing that can happen. Suppose two cash machines, A and B, are both working on a deposit at the same time. Here's how the `deposit()` step typically break down into low-level processor instructions:

get balance	0
add 1	1
write back the result	1

When A and B are running concurrently, these low-level instructions **interleave** with each other (some might even be simultaneous in some sense, but let's just worry about interleaving for now):

A get balance	0	
A add 1	1	
A write back the result	1	
		B get balance
		1
		B add 1
		2
		B write back the result
		2

This interleaving is fine – we end up with balance 2, so both A and B successfully put in a dollar. But what if the interleaving looked like this:

A get balance	0	
		B get balance
		0
A add 1	1	
		B add 1
		1
A write back the result	1	
		B write back the result
		1

The balance is now 1 – A's dollar was lost! A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other's deposit into account.

This is an example of a race condition

- A **race condition** means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations
 - “A is in a race with B”
- Some interleavings of events may be OK (in the sense that they are consistent with what a single, nonconcurrent process would produce), but other interleavings produce wrong answers – violating postconditions or invariants

Tweaking the Code Won't Help

All these versions of the code exhibit the same race condition:

```
// version 1
private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}

// version 2
private static void deposit() {
    balance += 1;
}
private static void withdraw() {
    balance -= 1;
}

// version 3
private static void deposit() {
    ++balance;
}
private static void withdraw() {
    --balance;
}
```

You can't tell just from looking at Java code how the processor is going to execute it. You can't tell what the indivisible operations – the *atomic* operations – will be. It isn't atomic just because it's one line of Java. It doesn't touch `balance` only once just because the `balance` identifier occurs only once in the line. The Java compiler, and in fact the processor itself, makes no commitments about what low-level operations it will generate from your code. In fact, a typical Java compiler produces exactly the same code for all three of these versions!

The key lesson is that **you can't tell by looking at an expression whether it will be safe from race conditions.**

Reordering

It's even worse than that, in fact. The race condition on the bank account balance can be explained in terms of different interleavings of sequential operations on different processors. But *in fact, when*

you're using multiple variables and multiple processors, you can't even count on changes to those variables appearing in the same order.

Here's an example:

```
private boolean ready = false;
private int answer = 0;

// computeAnswer runs in one thread
private void computeAnswer() {
    answer = 42;
    ready = true;
}

// useAnswer runs in a different thread
private void useAnswer() {
    while (!ready) {
        Thread.yield();
    }
    System.out.println(answer);
}
```

We have two methods that are being run in different threads. `computeAnswer` does a long calculation, finally coming up with the answer 42, which it puts in the `answer` variable. Then it sets the `ready` variable to true, in order to signal to the method running in the other thread, `useAnswer`, that the answer is ready for it to use. Looking at the code, `answer` is set *before* `ready` is set, so once `useAnswer` sees `ready` as true, then it seems reasonable that it can assume that the answer will be 42, right? Not so.

The problem is that modern compilers and processors do a lot of things to make the code fast. One of those things is making temporary copies of variables like `answer` and `ready` in faster storage (registers or caches on a processor), and working with them temporarily before eventually storing them back to their official location in memory. The storeback may occur in a *different order* than the variables were manipulated in your code. Here's what might be going on under the covers (but expressed in Java syntax to make it clear). The processor is effectively creating two temporary variables, `tmpr` and `tmpa`, to manipulate the fields `ready` and `answer`:

```
private void computeAnswer() {
    boolean tmpr = ready;
    int tmpa = answer;

    tmpa = 42;
    tmpr = true;

    ready = tmpr;
    // <== what happens if useAnswer() interleaves here?
    //      ready is set, but answer isn't
    answer = tmpa;
}
```

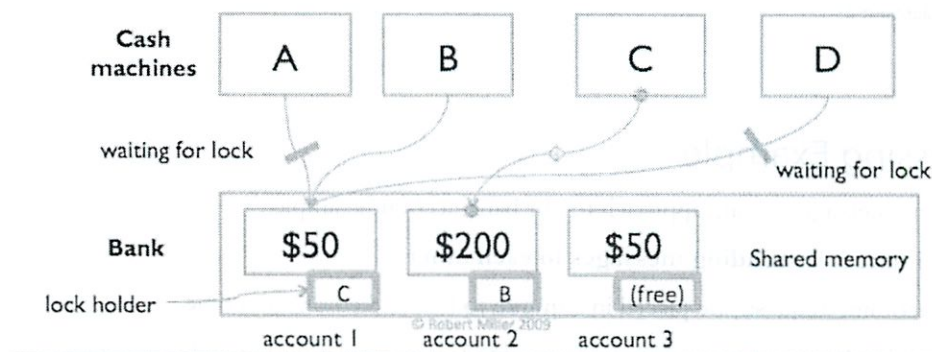
Synchronization

The correctness of a concurrent program should not depend on accidents of timing

Race conditions are nasty bugs -- may be rarely observed, hard to reproduce, hard to debug, but may have very serious effects.

To avoid race conditions, concurrent modules that share memory need to **synchronize** with each other.

- **Locks** are a common synchronization mechanism
- Holding a lock means "I'm changing this; don't touch it right now"
- Suppose B acquires the lock first; then A must wait to read and write the balance until B finishes and releases the lock
- Ensures that A and B are synchronized, but another cash machine C would be able to run independently on a different account (with a different lock)



- Acquiring or releasing a lock also tells the compiler and processor that you're using shared memory concurrently, so that registers and caches will be flushed out to the shared storage (which solves the reordering problem)

Deadlock

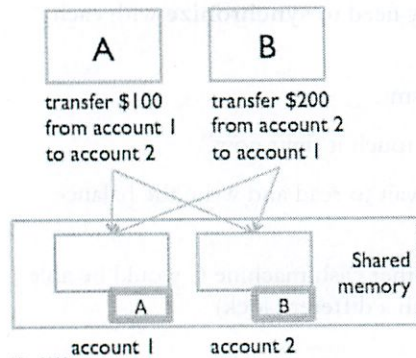
Suppose A and B are making simultaneous transfers

- A transfer between accounts needs to lock both accounts, so that money can't disappear from the system
- A and B each acquire the lock on the "from" account
- Now each must wait for the other to give up the lock on the "to" account
- Stalemate! A and B are frozen, and the accounts are locked up.

"Deadly embrace"

- **Deadlock** occurs when concurrent modules are stuck waiting for each other to do something

- A deadlock may involve more than two modules (e.g., a cycle of transfers among N accounts)
- You can have deadlock without using locks – example later

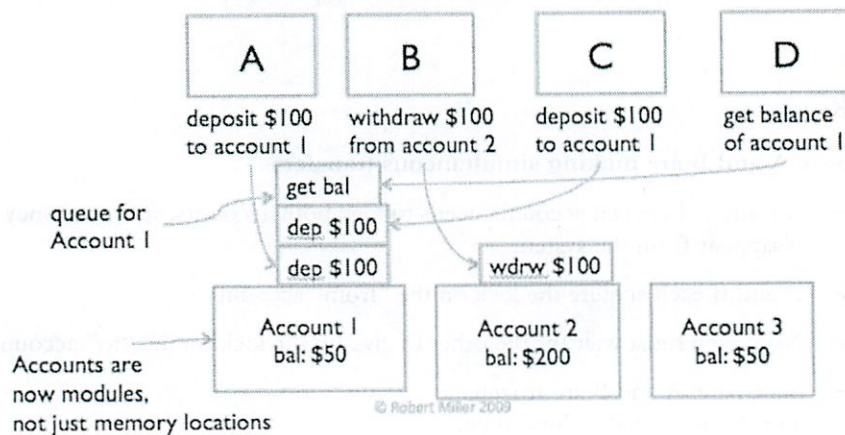


Message Passing Example

Now let's look at the message-passing approach to our bank account example.

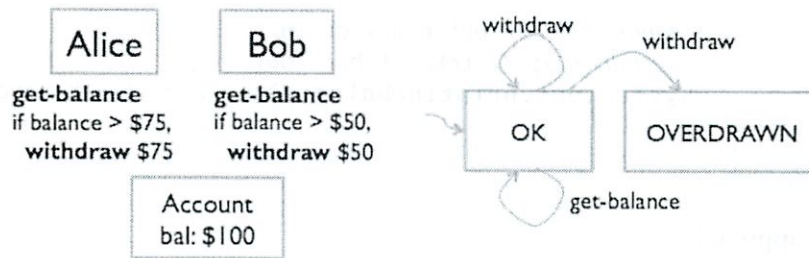
Modules interact by sending messages to each other

- Incoming requests are placed in a **queue** to be handled one at a time
- Sender doesn't stop working while waiting for an answer to its request; it handles more requests from its own queue
- Reply eventually comes back as another message



Message passing doesn't eliminate race conditions

- Suppose the account state machine supports **get-balance** and **withdraw** operations (with corresponding messages)
- Can Alice and Bob always stay out of the **OVERDRAWN** state?



- Lesson: need to carefully choose the **atomic** (indivisible) operations of the state machine – `withdraw-if-sufficient-funds` would be better

Message-passing can have deadlocks too

- Particularly when using finite queues that can fill up

Concurrency is Hard to Test and Debug

If we haven't persuaded you that concurrency is tricky, here's the worst of it. It's very hard to discover these kinds of concurrency bugs (race conditions and deadlocks) using testing. And even once a test has found a bug, it may be very hard to localize it to the part of the program causing it.

Poor coverage

- Recall our notions of coverage
 - all states, all transitions, or all paths through a state machine
- Given two concurrent state machines (with N states and M states), the combined system has $N \times M$ states (and many more transitions and paths)
- As concurrency increases, the state space explodes, and achieving sufficient coverage becomes infeasible

Poor reproducibility

- Transitions are **nondeterministic**, depending on relative timing of events that are strongly influenced by the environment
 - Delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc.
- Test driver can't possibly control all these factors
- So even if state coverage were feasible, the test driver can't reliably reproduce particular paths through the combined state machine

heisenbugs

- a "heisenbug" is nondeterministic, hard to reproduce (as opposed to a "bohrbug", which shows up repeatedly whenever you look at it – almost all bugs in sequential programming are bohrbugs)
- a heisenbug may even disappear when you try to look at it with `println` or debugger!

```
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
```

```

        deposit(); // put a dollar in
        withdraw(); // take it back out
        System.out.println(balance); // makes the bug
unreproducible!
    }
}

```

one approach

- build a lightweight event log (circular buffer)
- log events during execution of program as it runs at speed
- when you detect the error, stop program and examine logs

Summary

Concurrency

- Multiple computations running simultaneously

Shared-memory & message-passing paradigms

- Shared memory needs a synchronization mechanism, like locks
- Message passing synchronizes on communication channels, like streams or queues

Processes & threads

- Process is like a virtual computer; thread is like a virtual processor

Race conditions

- When correctness of result (postconditions and invariants) depends on relative timing of events

Deadlock

- When concurrent modules get stuck waiting for each other

Projects due today

No automated grading

Specify behavior you chose
document

Concurrency

Lecture summary

Is it faster?

- ~~code~~ can be slower
- perform different tasks

If 1 CPU you will go slower

- Thread overhead

→ CPU can parallelize

* Must be specific type of problem

Concurrency can make things faster

Need for UI

Or web servers

Processes

- Different apps
- No ~~all~~ memory sharing b/w processes by default
- So can do message passing

Threads

- ~~part~~ different parts of same app
- have a shared memory b/w ~~process~~ threads
- can have its own memory too
- Could also use message passing

Threads dangerous - must be careful
Processes much less danger

Clock Processor assigns ~~same~~ time slices to each processes
One wins

③

```
thread = new Thread (Runnable)  
thread.start();  
i++;
```

```
public void run()  
{  
    i = "hello";  
}
```

Ya don't know which is 1st!

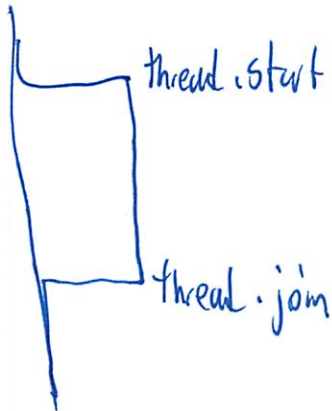
It can happen either way

No promises

4

thread.join()

- wait for thread to finish executing
- then join it back into the main program



6.005
Lecture 11
Processes + Sockets

☐ client/server

☐ network sockets

☐ blocking

☐ wire protocols

☐ deadlock

PS 5 out, due Thur night

3 more Pscts 5, 6, 7

Then final project

PS 5: processes + networking

PS 6: threads + locks

PS 7: GUI

Project: Combines all 3: IM client/server system

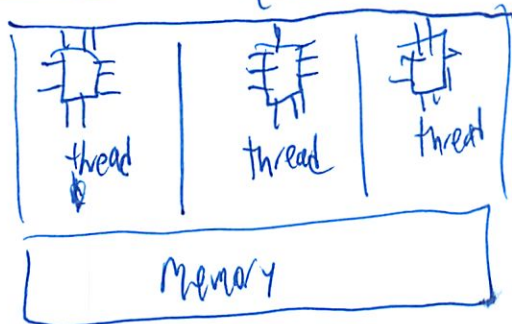
Some lectures only on 'functional programming

- immutability
- passing higher order procedures
 - take procedures as arguments
- Map Reduce

②

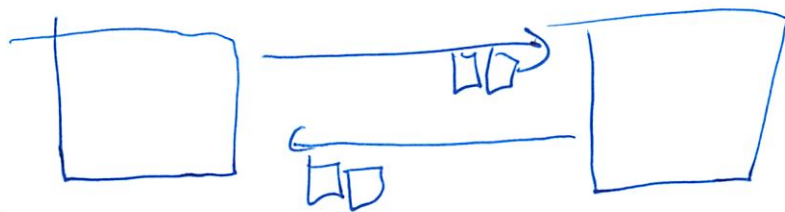
Concurrency

- Shared memory model



- problems when threads both change something at same time - race condition
 - answer depends on specific timing

- message passing

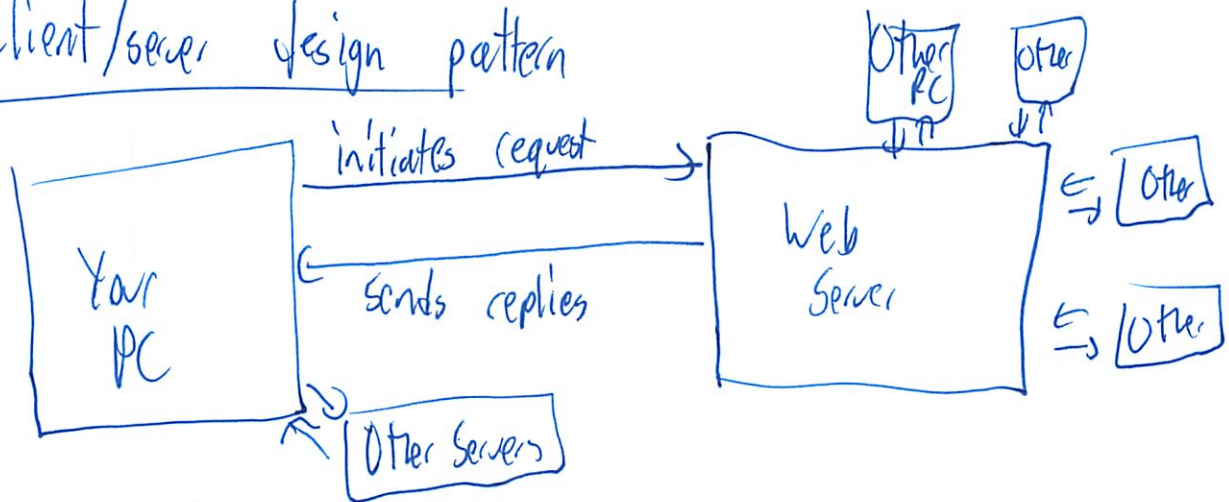


- talk about today
- deadlock
 - big issue in both cases
 - mostly in message passing
 - but can also arise in shared memory when trying to ~~kill~~^{prevent} race conditions

3

In Java very easy to set up shared memory
- so easy ya don't realize how dangerous it is

Client/server design pattern



Our PS & servers make other clients wait

Look at via network sockets or processes

- Processes

like virtual computer

own CPU (it looks like)

own memory



Input Streams

Output Streams

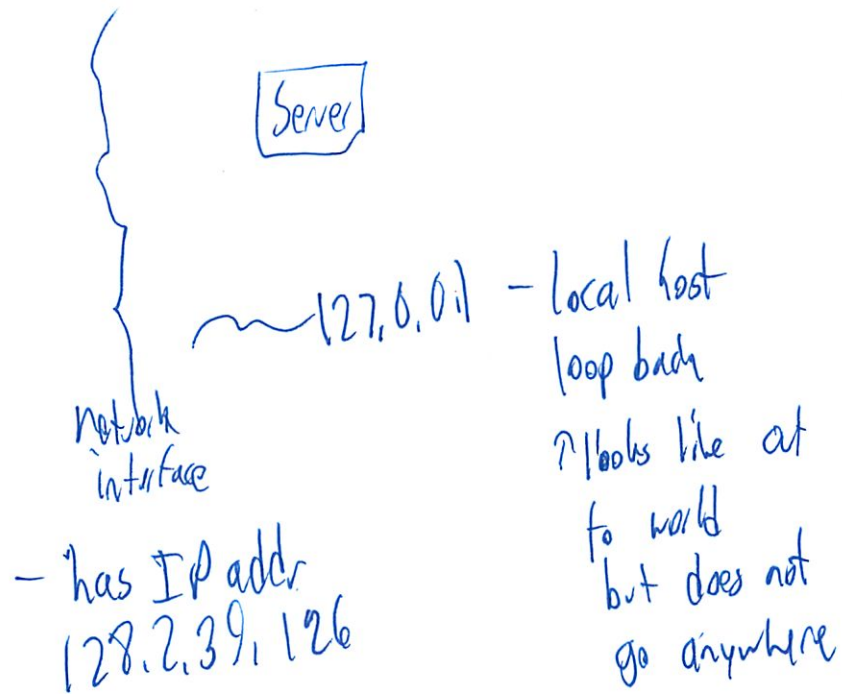
↑ like a file's contents

(4)

Client/server can be on same machine as diff processes

↳ still uses networking stack

- but does not go out online



Each network interface has a range of ports

0 → 65535

(for each interface)

two way

like mit.edu: 80
port 80

Web server = port 80

↳ listen on port 80

- creates socket obj on port 80

5

Port #s different if currently open

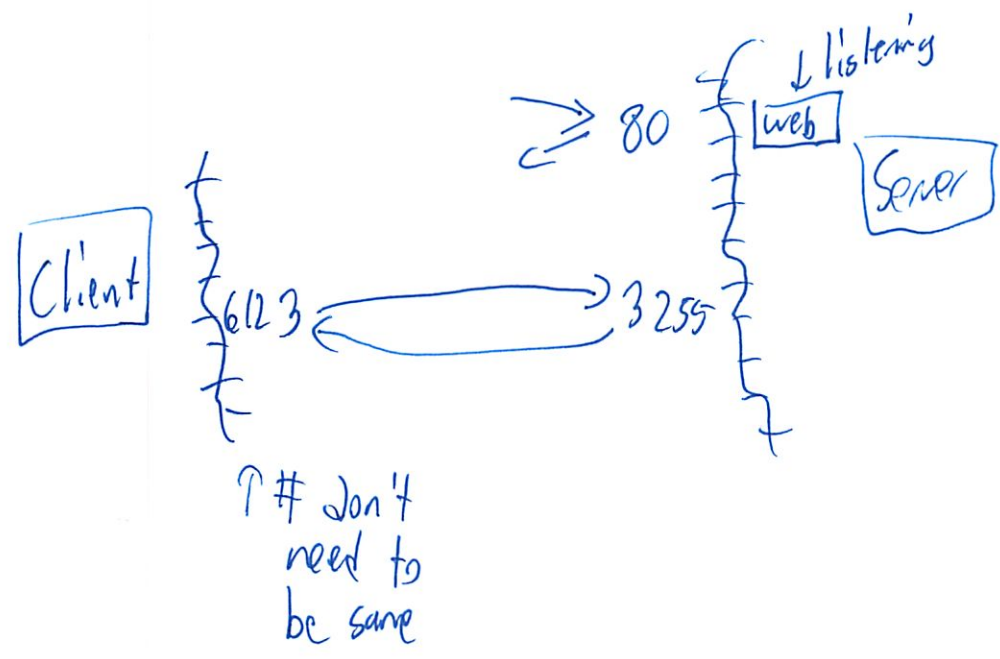
Server Socket - listens for new connections on fixed port

~~Client socket connects~~

When client connection arrives - allocate fresh port

L Socket

↑ "Berkeley Sockets" model



We view network as black box - TCP

But we can't count on timing

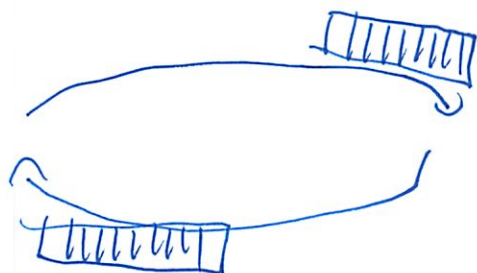
↓ brings up blocking

⑥ Blocking

Read() returns data if available
waits if not available

Write() writes data if it can
waits if all buffers are full

Since we have a finite-sized buffer on connection



Same Parties send other party message if buffer is full

6.033 covers in more detail

What does data on wire actually look like?

Protocol

Set of messages \rightarrow requests by client
 \rightarrow replies from server

Sequences of chars / bytes

grammar to specify our "wire protocol"

⑦

So for a server that takes stream of ints x, y, z
and yields their squares x^2, y^2, z^2

Client requests

Client $i := \text{Request}$

Request $i := \text{Num} \backslash n$

Num $i := i[0-9]^+$

Server replies

Server $i := \text{Reply}$

Reply $i := \text{Num} \backslash n$



Web servers use protocols like this

↳ ASCII based

Can use telnet to open web pages
Very basic program

- good for debugging

Good protocol design

- ~~Not~~ ready for change

- version #

- extensibility

- So can support sth like IPv6

②

Common names would let it support other things

GET, PUT, DELETE

Platform independence

- client could be written in other lang
 - like Java object serialization
 - but other languages don't understand
 - so use XML instead
 - JSON
 - generic enough so anything can read
-

Started 2 different processes

L11: Processes & Sockets

Today

- Client/server
- Network sockets
- Blocking
- Wire protocols
- Deadlocks

Required reading (from the Java Tutorial)

- [I/O Streams](#) (up to I/O from the Command Line)
- [Network Sockets](#)

Review

Shared memory vs. message passing

Race conditions caused by shared memory access

Today: dig deeper into message passing, and see our first example of deadlock

Client/Server Design Pattern

in today's lecture (and in PS5) we're going to use a well-established design pattern for message passing called **client/server**.

This pattern has multiple processes communicating by message passing. There are two kinds of processes: clients and servers. A client initiates the communication by connecting to a server. The client sends requests to the server, and the server sends replies back. Finally the client disconnects.

Many Internet applications work this way: web browsers are clients for web servers, an email program like Thunderbird or Outlook is a client for a mail server, etc.

On the Internet, client and server processes are often running on different machines connected by the network, but it doesn't have to be -- the server can be a process running on the same machine as the client.

Network Sockets

a network interface is identified by an IP address (or a hostname, which translates into an IP address; so there may be many synonyms)

examples: 127.0.0.1, localhost; web.mit.edu

an interface has 65536 ports, numbered from 0 to 65535

a server process binds to a port (the listening port). clients have to know which number it's binding to. Some numbers are well-known (port 80 is the standard web server port, port 22 is the SSH port, port 25 is the standard SMTP email server port). When it's not a standard port for the kind of server, you just treat it as part of the address (you may have seen URLs like <http://128.2.39.10:9000?> The 9000 is the port number to connect to on the computer at IP address 128.2.39.10.

the listening port is just used to accept incoming client connections. Once the connection is accepted, the server creates a new socket for the actual connection, with a fresh port number (unrelated to the listening port number). Both the client and server sockets have port numbers.

Buffers

Data is sent over a network in chunks. Rarely just byte-sized chunks (though they may be). The sending side typically writes a big chunk (maybe a whole string like “Hello, world!”, or maybe 20 megabytes worth of video data all at once). The network chops that chunk up into packets, which are routed separately over the network. And the receiver reassembles the packets together to a stream of bytes.

The result is a bursty kind of data transmission – the data may be there when you want to read it, or you may have to wait for it.

When data arrives, it is put into a **buffer**, which is simply an array in memory that is holding it until you read it.

Streams

Stream abstraction: a sequence of bytes

Most modern languages support Unicode, in which characters are 16 bits. But file storage and network transmission uses bytes, which are only 8 bits long.

character set (Unicode) vs. character encoding (Latin-1, UTF-8, UTF-16, Windows horrible)

InputStream/OutputStream vs. Reader/Writers

Blocking

Blocking means a thread waits (doing nothing) until an event occurs. It’s usually used to refer to a method call: when a method call **blocks**, it delays returning to its caller until the event occurs.

Socket streams exhibit blocking behavior:

- When an incoming socket’s buffer is empty, `read()` blocks.
- When the destination socket’s buffer is full, `write()` blocks.

Blocking is very convenient from a programmer’s point of view, because the programmer can write code as if the `read()` call will always succeed, no matter what the timing of data arrival. The operating system takes care of the details of delaying your thread until `read()` *can* succeed.

Blocking happens throughout concurrent programming, not just in I/O. Concurrent modules don’t work in lockstep, like sequential programs do, so they typically have to wait for each other to catch up.

We’ll see, though, that all this waiting causes the second major kind of bug in concurrent programming: **deadlocks**.

Wire Protocols

Now that we’ve got our client and our server and they’re connected up with sockets, what do they pass back and forth over those sockets?

a **protocol** is a set of messages that can be exchanged by two communicating parties. A **wire protocol**, in particular, is a set of messages represented as byte sequences, like “hello world” and “bye”.

most Internet applications use simple ASCII-based wire protocols. You can even use a Telnet program to check them out. For example:

HTTP:

```
telnet web.mit.edu 80
GET /
```

The GET command gets a web page; the / is the path of the page you want on the web.mit.edu server. So this command effectively fetches the page at `http://web.mit.edu:80/`

Internet protocols are defined by RFC specifications (RFC stands for “request for comment”).

Designing a Wire Protocol

similar to defining operations for an abstract data type: small, coherent, adequate

the equivalent of representation independence is platform-independence

ready for change – e.g., version number that client and server can announce to each other. GET / HTTP/1.0

Data Serialization

Java object serialization

XML

JSON

Deadlock

When buffers fill up, message passing systems can experience deadlock.

Deadlock: two concurrent modules are both blocked waiting for each other to do something. Since they’re blocked, neither will be able to make it happen, and neither will break the deadlock.

In general, in a system of multiple concurrent modules communicating with each other, we can imagine drawing a graph in which the nodes are the modules and there’s an edge from A to B if A is blocked waiting for B to do something. The system is deadlocked if at some point in time, there’s a cycle in this graph. The simplest case is the two-node deadlock, $A \rightarrow B$ and $B \rightarrow A$, but more complex systems can have larger deadlocks.

Deadlocked systems appear to simply hang. They’re not done, there’s still work to be done, they just can’t make any progress.

One solution to deadlock is to design the system so that there is no possibility of a cycle – so that if it’s possible for A to wait for B, then it’s never possible for B to wait for A.

Another approach to deadlock is *timeouts* – if a module has blocked for too long (maybe 100 milliseconds? maybe 10 seconds? it depends on the application and how long you need to wait), then you stop blocking and throw an exception. Then the problem becomes what do you do when that **exception** gets thrown.

We'll come back to deadlocks again when we talk about locking (which is what gave deadlock its name).

Can have arbitrary # of sockets on a port
Buffered reader should be used

```
Socket s = new Socket("18.111.2.101", 9444);  
BufferedReader in = new BufferedReader(new  
new InputStreamReader(s.getInputStream()));  
System.out.println(in.readLine());
```

~~Ans~~

PrintWriter p = ...

p.println("Pisces");

p.flush();

↳ auto flushes after
every print line

L12: Thread Safety

Today

- Confinement
- Threadsafe datatypes

Required reading

- [Concurrency](#)
- [Wrapper Collections](#)

Optional reading

The material in this lecture and the next lecture is inspired by an excellent book:

- Brian Goetz et al. *Java Concurrency in Practice*. Addison-Wesley, 2006.

Review

Recall race conditions: multiple threads sharing the same mutable variable without coordinating what they're doing. This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.

There are basically four ways to make variable access safe in shared-memory concurrency:

- **don't share** the variable between threads. This idea is called *confinement*, and we'll explore it today.
- make the shared data **immutable**. We've talked a lot about immutability already, but there are some additional constraints for concurrent programming that we'll talk about in this lecture.
- encapsulate the shared data in a **threadsafe datatype** that does the coordination for you. We'll talk about that today.
- use **synchronization** to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own threadsafe datatype. We'll talk about that next time.

Threads

A look at how to create threads in Java. For more details, see the Java tutorial link above.

```
public void serve() throws IOException {
    ServerSocket serverSocket = new ServerSocket(PORT);

    while (true) {
        // block until a client connects
        final Socket socket = serverSocket.accept();

        // start a new thread to handle the connection
        Thread thread = new Thread(new Runnable() {
            public void run() {
                // the client socket object is now owned by this thread,
                // and mustn't be touched again in the main thread
                handle(socket);
            }
        });
        thread.start(); // IMPORTANT! easy to forget
        // when does thread.start() return?
        // when will the thread stop?
    }
}
```

Thread Confinement

Thread confinement is a simple idea: you avoid races on mutable data by keeping that data confined to a single thread. Don't give any other threads the ability to read or write the data directly.

Local variables are always thread confined! A local variable is stored in the stack, and each thread has its own stack. There may be multiple invocations of a method running at a time (in different threads or even at different levels of a single thread's stack, if the method is recursive), but each of those invocations has its own private copy of the variable, so the variable itself is confined.

But be careful – the *variable* is thread confined, but if it's an object reference, you also need to check the *object* it points to. If the object is mutable, then we want to check that the object is confined as well – there can't be references to it that are reachable from any other thread.

What about local variables that are initially shared with a thread's Runnable when it starts, like *socket* in the code above? Java requires those variables to be final! Immutable references are okay to use from multiple threads, because you don't get races with immutability.

Avoid Global Variables

Unlike local variables, global variables (called "static" in Java) are *not* automatically thread confined.

If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly. Better, you should eliminate the static variables entirely.

Here's an example that we looked at in a previous lecture:

```
// This class has a race condition in it.
public class Midi {
```



```

private static Midi midi = null;
// invariant: there should never be more than one Midi object created

private Midi() {
    System.out.println("created a Midi object");
}

// factory method that returns the sole Midi object, creating it if it
doesn't exist
public static Midi getInstance() {
    if (midi == null) {
        midi = new Midi();
    }
    return midi;
}
}

```

This class has a race in the `getInstance()` method – two threads could call it at the same time and end up creating two copies of the `Midi` object, which we don't want.

To fix this race using the thread confinement approach, you would specify that only a certain thread (maybe the “midi playing thread”) is allowed to call `Midi.getInstance()`. The risk here is that Java won't help you guarantee this.

In general, static variables are very risky for concurrency. They might be hiding behind an innocuous function that seems to have no side-effects or mutations. Consider this example:

```

// is this method threadsafe?
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if and only if x is prime
 */
public static boolean isPrime(int x) {
    if (cache.containsKey(x)) return cache.get(x);
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
    cache.put(x, answer);
    return answer;
}

private static Map<Integer, Boolean> cache = new
HashMap<Integer, Boolean>();

```

This method is not safe to call from multiple threads, and its clients may not even realize it.

Threadsafe

A datatype is *threadsafe* if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code.

- “behaves correctly” means satisfying its specification and preserving its rep invariant
- “how threads are executed” means threads might be on multiple processors or timesliced on the same processor
- “without additional coordination” means that the datatype can't put preconditions on its caller related to timing, like “you can't call `get()` while `set()` is in progress.”

Remember Iterator? It's not threadsafe. Iterator's specification says that you can't modify a collection at the same time as you're iterating over it. That's a precondition put on the caller, and Iterator makes no guarantee to behave correctly if you violate it. So it's not threadsafe.

Immutability

Final variables are constants, so the variable itself is threadsafe.

Immutable objects are generally also threadsafe. We say generally because our current definition of immutability is too loose for concurrent programming. We've said that a type is immutable if an object of the type always represents the same abstract value for its entire lifetime. But that actually allows the type the freedom to mutate its rep, as long as those mutations are invisible to clients. We saw an example of this notion, called *benevolent* or *beneficient mutation*, when we looked at an immutable list that cached its length in a mutable field the first time the length was requested by a client. Caching is a typical kind of beneficent mutation.

For concurrency, though, this kind of hidden mutation is a no-go. An immutable datatype that uses beneficent mutation will have to make itself threadsafe using locks (the same technique required of mutable datatypes), which we'll talk about next lecture.

So in order to be confident that an immutable datatype is threadsafe *without* locks, we need stronger rules:

- no mutator methods
- all fields are private and final
- no mutation whatsoever of mutable objects in the rep -- not even beneficent mutation
- no rep exposure (see the ADT lecture to review what rep exposure is)

If you follow these rules, then you can be confident that your immutable type will also be threadsafe.

Threadsafe Collections

The collection interfaces in Java – List, Set, Map – have basic implementations which are *not* threadsafe. The implementations of these that you've been used to using, namely ArrayList, HashMap, and HashSet, cannot be used safely from more than one thread.

Fortunately, just like the Collections API provides wrapper methods that make collections immutable, it provides another set of wrapper methods to make collections threadsafe, while still mutable.

These wrappers effectively make each method of the collection **atomic**. An atomic action effectively happens all at once – it doesn't interleave its internal operations with other threads, and none of the effects of the action are visible to other threads until the entire action is complete, so it never looks partially done.

Now we see a way to fix that `isPrime()` method we had earlier in the lecture:

```
private static Map<Integer, Boolean> cache = Collections.synchronizedMap(  
new HashMap<Integer, Boolean>());
```

A few points here.

First, make sure to throw away references to the underlying non-threadsafe collection, and access it only through the synchronized wrapper. That happens automatically in the line of code above, since the new HashMap is passed only to `synchronizedMap()` and never stored anywhere else.

Second, even though method calls on the collection itself (`get()`, `put()`, `add()`, etc.) are now threadsafe, iterators created from the collection are still *not* threadsafe. So you can't use `iterator()`, or the `for` loop syntax:

```
for (String s: lst) { .... } // not threadsafe, even if lst is a synchronized list wrapper
```

The solution to this problem will be to acquire the collection's lock when you need to iterate over it, which we'll talk about next time.

Finally, the way that you *use* the synchronized collection can still have a race condition! Consider this code, which checks whether a list has at least one element and then gets that element:

```
if (! lst.isEmpty()) { String s = lst.get(0); ... }
```

Even if you make `lst` into a synchronized list, this code still may have a race condition, because another thread may remove the element between the `isEmpty()` call and the `get()` call.

Even the `isPrime()` method still has potential races:

```
if (cache.containsKey(x)) return cache.get(x);
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
cache.put(x, answer);
```

The synchronized map ensures that `containsKey()`, `get()`, and `put()` are now atomic, so using them from multiple threads won't damage the rep invariant of the map. But those three operations can now interleave in arbitrary ways with each other which might break the invariant that `isPrime` needs from the cache: if the cache maps an integer `x` to a value `f`, then `x` is prime if and only if `f` is true. If the cache ever fails this invariant, then we might return the wrong result.

So we have to argue that the races between `containsKey()`, `get()`, and `put()` don't threaten this invariant. First, the race between `containsKey()` and `get()` is not harmful because we never remove items from the cache – once it contains a result for `x`, it will continue to do so. Second, there's a race between `containsKey()` and `put()`. As a result, it may end up that two threads will both test the primeness of the same `x` at the same time, and both will race to call `put()` with the answer. But both of them should call `put()` with the same answer, so the race will be harmless.

The need to make these kinds of careful arguments about safety – even when you're using threadsafe datatypes – is the main reason that concurrency is hard.

Goals of Concurrent Program Design

Now is a good time to pop up a level and look at what we're doing. Recall that the primary goals of this course are to learn how to create software that is (1) safe from bugs, (2) easy to understand, and (3) ready for change. There are other properties of software that are important, like performance, usability, security, etc., but we're deferring those properties for the sake of this course.

Building concurrent software has these same overall goals, but they break down more specifically into some common classes. In particular, when we ask whether a concurrent program is safe from bugs, we care about two properties:

- **Safety.** Does the concurrent program satisfy its invariants and its specifications? Races in accessing mutable data threaten safety. Another way to put this is, can you prove that **nothing bad ever happens**?
- **Liveness.** Does the program keep running and eventually do what you want, or does it get stuck waiting forever for events that will never happen? Can you prove that **something good eventually happens**? Deadlocks threaten liveness. Liveness may also require *fairness*, which means that concurrent modules are able to make progress in their computations

when they are actually able to run. If Eclipse's editor module hogs the only processor in the system, so that the compiler module never gets a chance to run, then you won't ever get your program compiled – a liveness failure. Fairness is mostly a matter for the operating system's thread scheduler, which decides how to timeslice threads, but you can influence the scheduler's decisions with mechanisms like thread priorities, so it's possible for a system design to threaten fairness.

Concurrent programs also usually worry about **performance**, i.e. the speed or resource usage of the program, since that is often the main reason for introducing concurrency into the system in the first place (making the program work faster or respond more quickly). We've largely been postponing issues of performance in 6.005. 6.172 Performance Engineering is strongly recommended for learning about this, and it covers performance of concurrent programs in great detail. But here are some high-level comments about getting good performance with threads.

Create only a few threads. Threads cost resources – memory for a stack, processor time to switch threads, operating system resources. So don't create them as freely as you create data objects. There are typically two reasons why you create threads: to do I/O (e.g. network, MIDI device, graphical user interface), or to handle computation. Each has some rules of thumb:

- For I/O: create at most one thread per stream (often one for reading and one for writing), so that it can block without preventing other streams from making progress.
- For computation: the sweet spot is slightly more threads than you have processors (`Runtime.getRuntime().availableProcessors()`). If you start 100 threads but you have only 4 processors for them to run on, then you're just wasting memory and time, and you won't finish the job any faster than 4 threads would. Java has an interface called `ExecutorService` that manages a pool of threads for a queue of tasks, so you can chop your computation up into bits and use as many threads as make sense to do them.

Don't move work between threads unnecessarily. Sending requests to other threads is expensive; switching threads on a processor is expensive; moving data between threads is expensive; synchronizing and coordinating between threads is expensive. So if thread A has a piece of work that needs to get done, and it has all the data it needs to do the work and can do it safely (without races), then A should just do the work itself, rather than handing it off to another thread B. If an I/O thread in a server receives a request that doesn't require accessing mutable data that is confined to another thread, it should just handle the request itself, rather than sending it to another thread.

How to Make a Safety Argument

We've seen that concurrency is hard to test and debug. So if you want to convince yourself and others that your concurrent program is correct, the best approach is to **make an explicit argument** that it's free from races and deadlocks.

We're going to focus for now on the safety question. Your argument needs to catalog all the threads that exist in your module or program, and the data that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: confinement, immutability, thread-safe datatypes, or synchronization. (When you use the last two, you also need to argue that all accesses to the data are appropriately atomic – that is, that the invariants you depend on are not threatened by interleaving. We gave one of those arguments for `isPrime` above.)

The `SocialServer` code with this lecture has an example of a safety argument:

```
//  
// Thread safety argument  
// -----  
// The threads in the system are:  
// - main thread accepting new connections
```

```
// - one thread per connected client, handling just that client
//
// The serverSocket object is confined to the main thread.
//
// The Socket object for a client is confined to that client's thread;
// the main thread loses its reference to the object right after starting
// the client thread.
//
// The friendsOf map and all the lists inside it are confined to the main
// thread
// during creation and then immutable after creation.
//
// System.err is used by all threads for displaying error messages.
// No other shared mutable data.
//
```

6.005 Lecture 12

Thread Safety

11/2

☐ Confinement

☐ Thread safe datatypes

☐ producer / consumer

☐ blocking queues

PS 5 due Thur night Lcode review

PS 6 due Sat Sun, due next Thur
L multithreadable

Mem: Sync + locks

review

race condition

- big risk in problems w/ shared memory
- multiple threads sharing mutable data w/o coordinating
- ie the bank account problem
- can cause correctness bugs

today

how to do this safely

4 strategies

1. Don't share b/w threads

↳ Confinement

- don't make things accessible from other threads

2. Make data immutable

3. Use "thread safe" data types

↳ Coordination done in data type

- some collections, blocking queues

2

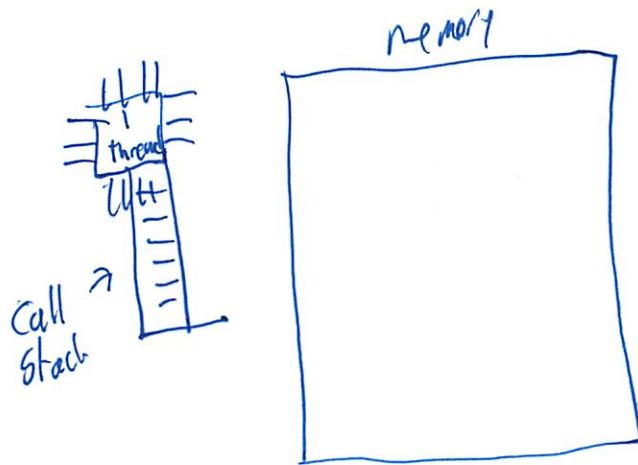
4. Synchronization

- umbrella word for coordinating timing

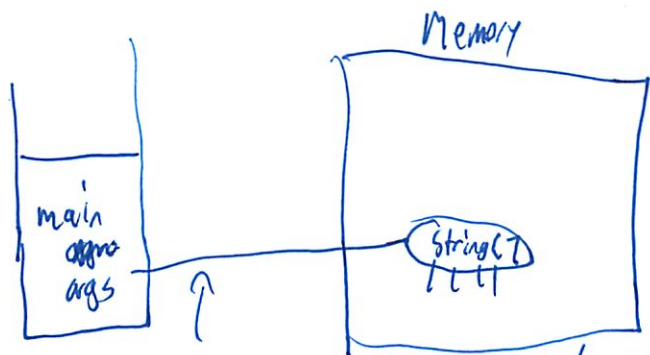
- locks on Monday

Confinement

- we will build a multi-threaded server



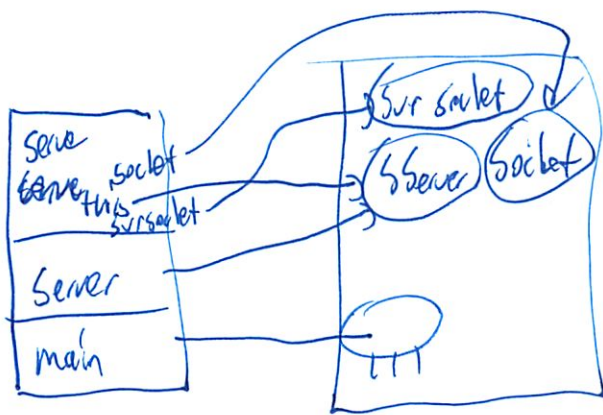
- start w/ 1 thread. First call is to main method



only visible to this thread
(confinement)

No other thread can touch it

3



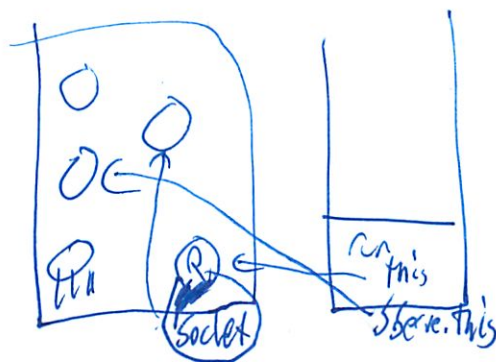
Then original/main thread listens for more sockets

Constructs a new Thread

↳ gives it a starting point: Runnable

- ↳ like a function ya can pass around
- like a lambda in PY
- like sort w/ comparison
- specifies entry point for thread

Need to know local variables inside run



Don't actually get pointer to main thread's stack

We can only reference it if final

- Java will make copy inside runnable

④

Could make unable out to new class for even less access
- So does not get access to SocialServer (SSrv)

Local variables (the arrow) are always confined

but Object itself can be changed

Static variable not confined

↳ don't need object

- if public can access

~~if protected~~

- get Instance is dangerous too

- both can construct at same time

- ~~if~~ instead you could only make 1 object and
accessible by a particular thread

Trace references through object to see if have confinement
if you do its like being in single thread

Static values dangerous

- can sync

5
Example is Prime w/ caching

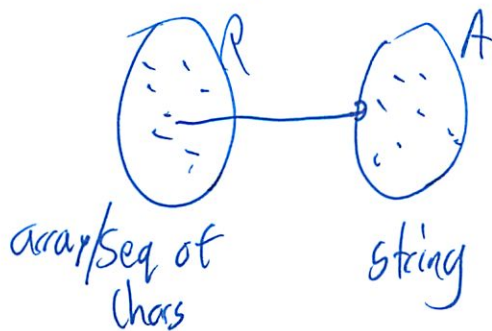
- two primes can race to write to cache
- used to crash
- been hardened
- since internally in data structure ~~was~~ writing and reading at same time bad
 - ↳ breaking invariants for short time when modifying it

Going to protect data structure

2. Immutability - talked about before

~~W/ threads~~

- what it represents never changes



Can have multiple Rs to 1 A



(6)

So under cover ~~bits~~, can actually be changing bits
in the rep - but still represents same value

old abstract value never changes

tweak for ~~rep~~ ^{thread} safe rep value is never mutated

- fields should be final

- important signal to compiler

- if fields point to mutable obj's, those
obj's are not mutated

- no rep exposure

3. Thread Safe Datatype

defn: behaves correctly when used for multiple
threads without add'l coordination

- ~~known~~ preserves invar^{ant}

- obeys spec

- precondition \rightarrow post condition

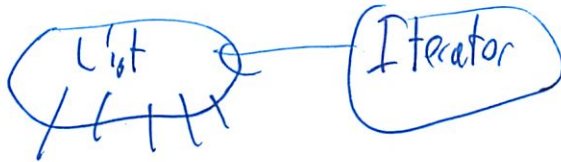
- no preconditions whatsoever about timing

- can't say "you can't call multiple times"

⑦

Must preserve post conditions + rep invariants whenever

- Iterator - can't mutate Collection while Iterating



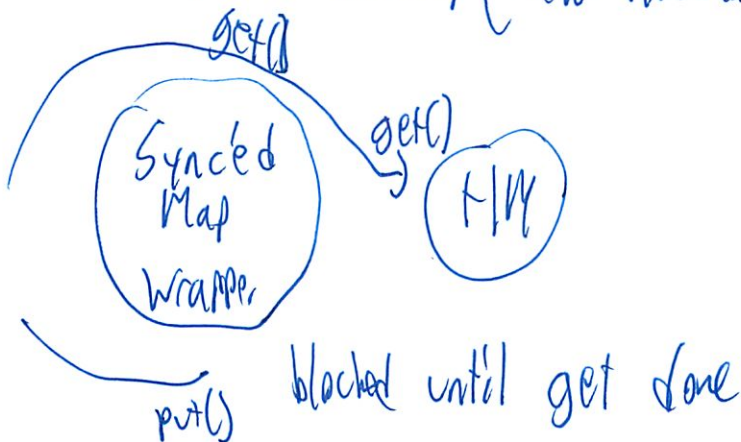
Sometimes throws ConcurrentModificationException
but not promised

So Iterator is not threadsafe

~~ArrayList~~ ArrayList, Hash Set, Hash Map not threadsafe

- Need to add a Synchronized wrapper
↳ like for immutability

Collections.synchronizedMap(new HashMap<Integer, Boolean>());



like a gate - only 1 through at a time

8

We want to make large steps atomic
- can't be interrupted

With sync we don't worry what is going on underneath
- abstracted away

But still race condition

- could argue so could say one does not matter
- if both threads call 'isPrime(5)'
 - will both calculate
 - will both try to insert in ~~cache~~ cache
- must still think about

Properties

- Safety - does ~~something~~ ^{nothing} bad ever happen?
 - races
- Liveness - does something good eventually happen
 - deadlock
- Performance - does it do both of these quickly?
 - more in 6c/72

L12: Thread Safety

Today

- Confinement
- Threadsafe datatypes

Required reading

- Concurrency
- Wrapper Collections

Optional reading

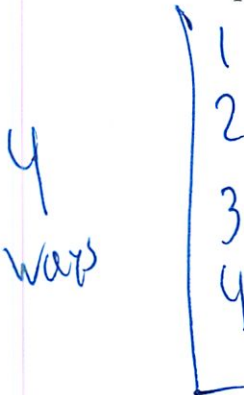
The material in this lecture and the next lecture is inspired by an excellent book:

- Brian Goetz et al. *Java Concurrency in Practice*. Addison-Wesley, 2006.

Review

Recall race conditions: multiple threads sharing the same mutable variable without coordinating what they're doing. This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.

There are basically four ways to make variable access safe in shared-memory concurrency:

- 
- 1 • don't share the variable between threads. This idea is called *confinement*, and we'll explore it today.
 - 2 • make the shared data **immutable**. We've talked a lot about immutability already, but there are some additional constraints for concurrent programming that we'll talk about in this lecture.
 - 3 • encapsulate the shared data in a **threadsafe datatype** that does the coordination for you. We'll talk about that today.
 - 4 • use **synchronization** to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own threadsafe datatype. We'll talk about that next time.

Threads

A look at how to create threads in Java. For more details, see the Java tutorial link above.

```
public void serve() throws IOException {
    ServerSocket serverSocket = new ServerSocket(PORT);

    while (true) {
        // block until a client connects
        final Socket socket = serverSocket.accept();

        // start a new thread to handle the connection
        Thread thread = new Thread(new Runnable() {
            public void run() {
                // the client socket object is now owned by this thread,
                // and mustn't be touched again in the main thread
                handle(socket);
            }
        });
        thread.start(); // IMPORTANT! easy to forget
        // when does thread.start() return?
        // when will the thread stop?
    }
}
```

Thread Confinement

Thread confinement is a simple idea: you avoid races on mutable data by keeping that data confined to a single thread. Don't give any other threads the ability to read or write the data directly.

Local variables are always thread confined! A local variable is stored in the stack, and each thread has its own stack. There may be multiple invocations of a method running at a time (in different threads or even at different levels of a single thread's stack, if the method is recursive), but each of those invocations has its own private copy of the variable, so the variable itself is confined.

But be careful – the *variable* is thread confined, but if it's an object reference, you also need to check the *object* it points to. If the object is mutable, then we want to check that the object is confined as well – there can't be references to it that are reachable from any other thread.

What about local variables that are initially shared with a thread's Runnable when it starts, like *socket* in the code above? Java requires those variables to be final! Immutable references are okay to use from multiple threads, because you don't get races with immutability.

Avoid Global Variables

Unlike local variables, global variables (called "static" in Java) are *not* automatically thread confined.

If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly. Better, you should eliminate the static variables entirely.

Here's an example that we looked at in a previous lecture:

```
// This class has a race condition in it.
public class Midi {
```



```

private static Midi midi = null;
// invariant: there should never be more than one Midi object created

private Midi() {
    System.out.println("created a Midi object");
}

// factory method that returns the sole Midi object, creating it if it
doesn't exist
public static Midi getInstance() {
    if (midi == null) {
        midi = new Midi();
    }
    return midi;
}
}

```

This class has a race in the `getInstance()` method – two threads could call it at the same time and end up creating two copies of the `Midi` object, which we don't want.

To fix this race using the thread confinement approach, you would specify that only a certain thread (maybe the “midi playing thread”) is allowed to call `Midi.getInstance()`. The risk here is that Java won't help you guarantee this.

In general, static variables are very risky for concurrency. They might be hiding behind an innocuous function that seems to have no side-effects or mutations. Consider this example:

```

// is this method threadsafe?
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if and only if x is prime
 */
public static boolean isPrime(int x) {
    if (cache.containsKey(x)) return cache.get(x);
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
    cache.put(x, answer);
    return answer;
}

private static Map<Integer, Boolean> cache = new
HashMap<Integer, Boolean>();

```

This method is not safe to call from multiple threads, and its clients may not even realize it.

Threadsafe

A datatype is *threadsafe* if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code.

- “behaves correctly” means satisfying its specification and preserving its rep invariant
- “how threads are executed” means threads might be on multiple processors or timesliced on the same processor
- “without additional coordination” means that the datatype can't put preconditions on its caller related to timing, like “you can't call `get()` while `set()` is in progress.”

Remember Iterator? It's not threadsafe. Iterator's specification says that you can't modify a collection at the same time as you're iterating over it. That's a precondition put on the caller, and Iterator makes no guarantee to behave correctly if you violate it. So it's not threadsafe.

Immutability

Final variables are constants, so the variable itself is threadsafe.

Immutable objects are generally also threadsafe. We say generally because our current definition of immutability is too loose for concurrent programming. We've said that a type is immutable if an object of the type always represents the same abstract value for its entire lifetime. But that actually allows the type the freedom to mutate its rep, as long as those mutations are invisible to clients. We saw an example of this notion, called *benevolent* or *beneficent mutation*, when we looked at an immutable list that cached its length in a mutable field the first time the length was requested by a client. Caching is a typical kind of beneficent mutation.

For concurrency, though, this kind of hidden mutation is a no-go. An immutable datatype that uses beneficent mutation will have to make itself threadsafe using locks (the same technique required of mutable datatypes), which we'll talk about next lecture.

So in order to be confident that an immutable datatype is threadsafe *without* locks, we need stronger rules:

- no mutator methods
- all fields are private and final
- no mutation whatsoever of mutable objects in the rep -- not even beneficent mutation
- no rep exposure (see the ADT lecture to review what rep exposure is)

If you follow these rules, then you can be confident that your immutable type will also be threadsafe.

Threadsafe Collections

The collection interfaces in Java – List, Set, Map – have basic implementations which are *not* threadsafe. The implementations of these that you've been used to using, namely ArrayList, HashMap, and HashSet, cannot be used safely from more than one thread.

Fortunately, just like the Collections API provides wrapper methods that make collections immutable, it provides another set of wrapper methods to make collections threadsafe, while still mutable.

These wrappers effectively make each method of the collection **atomic**. An atomic action effectively happens all at once – it doesn't interleave its internal operations with other threads, and none of the effects of the action are visible to other threads until the entire action is complete, so it never looks partially done.

Now we see a way to fix that `isPrime()` method we had earlier in the lecture:

```
private static Map<Integer, Boolean> cache = Collections.synchronizedMap(  
new HashMap<Integer, Boolean>());
```

A few points here.

First, make sure to throw away references to the underlying non-threadsafe collection, and access it only through the synchronized wrapper. That happens automatically in the line of code above, since the new HashMap is passed only to `synchronizedMap()` and never stored anywhere else.

Second, even though method calls on the collection itself (`get()`, `put()`, `add()`, etc.) are now threadsafe, iterators created from the collection are still *not* threadsafe. So you can't use `iterator()`, or the `for` loop syntax:

```
for (String s: lst) { .... } // not threadsafe, even if lst is a synchronized list wrapper
```

The solution to this problem will be to acquire the collection's lock when you need to iterate over it, which we'll talk about next time.

Finally, the way that you *use* the synchronized collection can still have a race condition! Consider this code, which checks whether a list has at least one element and then gets that element:

```
if (! lst.isEmpty()) { String s = lst.get(0); ... }
```

Even if you make `lst` into a synchronized list, this code still may have a race condition, because another thread may remove the element between the `isEmpty()` call and the `get()` call.

Even the `isPrime()` method still has potential races:

```
if (cache.containsKey(x)) return cache.get(x);
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
cache.put(x, answer);
```

The synchronized map ensures that `containsKey()`, `get()`, and `put()` are now atomic, so using them from multiple threads won't damage the rep invariant of the map. But those three operations can now interleave in arbitrary ways with each other which might break the invariant that `isPrime` needs from the cache: if the cache maps an integer `x` to a value `f`, then `x` is prime if and only if `f` is true. If the cache ever fails this invariant, then we might return the wrong result.

So we have to argue that the races between `containsKey()`, `get()`, and `put()` don't threaten this invariant. First, the race between `containsKey()` and `get()` is not harmful because we never remove items from the cache – once it contains a result for `x`, it will continue to do so. Second, there's a race between `containsKey()` and `put()`. As a result, it may end up that two threads will both test the primeness of the same `x` at the same time, and both will race to call `put()` with the answer. But both of them should call `put()` with the same answer, so the race will be harmless.

The need to make these kinds of careful arguments about safety – even when you're using threadsafe datatypes – is the main reason that concurrency is hard.

Goals of Concurrent Program Design

Now is a good time to pop up a level and look at what we're doing. Recall that the primary goals of this course are to learn how to create software that is (1) safe from bugs, (2) easy to understand, and (3) ready for change. There are other properties of software that are important, like performance, usability, security, etc., but we're deferring those properties for the sake of this course.

Building concurrent software has these same overall goals, but they break down more specifically into some common classes. In particular, when we ask whether a concurrent program is safe from bugs, we care about two properties:

- **Safety.** Does the concurrent program satisfy its invariants and its specifications? Races in accessing mutable data threaten safety. Another way to put this is, can you prove that **nothing bad ever happens**?
- **Liveness.** Does the program keep running and eventually do what you want, or does it get stuck waiting forever for events that will never happen? Can you prove that **something good eventually happens**? Deadlocks threaten liveness. Liveness may also require *fairness*, which means that concurrent modules are able to make progress in their computations

when they are actually able to run. If Eclipse's editor module hogs the only processor in the system, so that the compiler module never gets a chance to run, then you won't ever get your program compiled – a liveness failure. Fairness is mostly a matter for the operating system's thread scheduler, which decides how to timeslice threads, but you can influence the scheduler's decisions with mechanisms like thread priorities, so it's possible for a system design to threaten fairness.

Concurrent programs also usually worry about **performance**, i.e. the speed or resource usage of the program, since that is often the main reason for introducing concurrency into the system in the first place (making the program work faster or respond more quickly). We've largely been postponing issues of performance in 6.005. 6.172 Performance Engineering is strongly recommended for learning about this, and it covers performance of concurrent programs in great detail. But here are some high-level comments about getting good performance with threads.

Create only a few threads. Threads cost resources – memory for a stack, processor time to switch threads, operating system resources. So don't create them as freely as you create data objects. There are typically two reasons why you create threads: to do I/O (e.g. network, MIDI device, graphical user interface), or to handle computation. Each has some rules of thumb:

- For I/O: create at most one thread per stream (often one for reading and one for writing), so that it can block without preventing other streams from making progress.
- For computation: the sweet spot is slightly more threads than you have processors (`Runtime.getRuntime().availableProcessors()`). If you start 100 threads but you have only 4 processors for them to run on, then you're just wasting memory and time, and you won't finish the job any faster than 4 threads would. Java has an interface called `ExecutorService` that manages a pool of threads for a queue of tasks, so you can chop your computation up into bits and use as many threads as make sense to do them.

Don't move work between threads unnecessarily. Sending requests to other threads is expensive; switching threads on a processor is expensive; moving data between threads is expensive; synchronizing and coordinating between threads is expensive. So if thread A has a piece of work that needs to get done, and it has all the data it needs to do the work and can do it safely (without races), then A should just do the work itself, rather than handing it off to another thread B. If an I/O thread in a server receives a request that doesn't require accessing mutable data that is confined to another thread, it should just handle the request itself, rather than sending it to another thread.

How to Make a Safety Argument

We've seen that concurrency is hard to test and debug. So if you want to convince yourself and others that your concurrent program is correct, the best approach is to **make an explicit argument** that it's free from races and deadlocks.

We're going to focus for now on the safety question. Your argument needs to catalog all the threads that exist in your module or program, and the data that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: confinement, immutability, thread-safe datatypes, or synchronization. (When you use the last two, you also need to argue that all accesses to the data are appropriately atomic – that is, that the invariants you depend on are not threatened by interleaving. We gave one of those arguments for `isPrime` above.)

The `SocialServer` code with this lecture has an example of a safety argument:

```
//  
// Thread safety argument  
// -----  
// The threads in the system are:  
// - main thread accepting new connections
```

```
// - one thread per connected client, handling just that client
//
// The serverSocket object is confined to the main thread.
//
// The Socket object for a client is confined to that client's thread;
// the main thread loses its reference to the object right after starting
// the client thread.
//
// The friendsOf map and all the lists inside it are confined to the main
thread
// during creation and then immutable after creation.
//
// System.err is used by all threads for displaying error messages.
// No other shared mutable data.
//
```

-1 if not present

Should be $i+1$ if digit there

↑ what square is -1

actual	c	2	34	computer →	0	1	2	3
code "	b	2	34		-1	1	2	3

I am # as is

Am using -1

Will take off 3 points for the from file error

Will take off for the internal ref -10

Sample 2 null pointer - fails

$(a \wedge b) \vee (\sim a \wedge \sim b \wedge \sim c \wedge \sim d)$

2 diff paths one or other

②

have 2 Is in some samples
one can

internal cep

- internal solving + formulas

Can take a look at

Grade 72/100 ~~the~~ all

Can do return in

Overall: doing ^{very} well

projects matter most

Producer/Consumer Pattern

- Producer producing lots of objects
- Consumer using
- Producers put stuff on queue
- Consumers take them off
- Will use separate threads for consumers + producers

- queue

- ~~put~~ add() ← producer

- ~~take~~ remove() ← consumer



②

If Consumer is faster than producer

- Consumer can wait - called blocking

→ only blocking itself

- Consumer could ~~wait~~ return

If queue is full

- producer could block (itself)

- Or give up → return

Blocking Queue' Object

- has regular add() remove() methods

- also other methods put() take()

- blocking

- will wait till queue becomes operable

- also offer() poll() will same but will return true if it returns
or the object itself

3

What is advantage of put / take ?

Consumer

① $o = q.take()$

or

② while (true)

$o = poll()$

if ($o \neq null$)

break;

but not good at polling values of null

① waits till something is on queue

- called blocking - it waits itself

② keeps checking

- uses CPU cycles

- works w/ interrupts

- thread waits

- stops thread that line is in

(4)

So this is an example of message passing
Queue is only way threads communicate

My question to TA

Local variables in a thread only for that thread

- if your only reference outside variables in will
it have ~~that~~ access to that
- like in a Thread constructor

So more specifically - ~~it~~ can create new class
that implements Runnable

Then in its constructor can specify
Everything else is local

Mint P55 @ BeforeClass @Before for JUnit tests

6.005 Elements of Software Construction | Fall 2011

Problem Set 5: Finding Prime Factors with Networking

Due: Thursday, November 3 2011, 11:59 PM

The purpose of this problem set is to introduce you to aspects of Java's I/O and networking API, to help you get started building network applications.

You have substantial design freedom on this problem set. However, in order for your solution to be graded, your solution must not change the name, method signature, class name, package name, or specification of the following:

- `factors.client.PrimeFactorsClient.main()`
- `factors.server.PrimeFactorsServer.main()`
- `echo.client.EchoClient.main()`
- `echo.server.EchoServer.main()`

To get started, pull out the problem set code from [SVN Admin](#).

Background

Passing sensitive information over a network is a tricky operation, especially if listeners are ready to intercept your messages. Encryption has become central in protecting your important information. Modern cryptography systems encrypt your data in such a manner so that listeners would have to solve intractable problems, in particular the prime factorization of the product of two very large primes. Finding an efficient solution to factor this number makes it possible to crack the underlying message, exposing the encrypted sensitive data. In this problem set, we will create a solution that will perform brute-force prime factor searching, but distributed over multiple machines.

The **client-server** architecture models a distributed method of computing characterized by two parts, *client systems* and *server systems*. Both systems communicate with one another either over a network or perhaps even on the same system. Accordingly, this architecture is an example of a *distributed system* whereby we use multiple computers to solve a problem.

message passing

Client systems often initiate and make the requests to the servers. Generally, clients are seeking some service or have to rely on the resources of a server to handle their needs. These may include access to files, peripherals, or processing power. In this architecture, clients do not share resources with one another (this characteristic is more in tune with the *peer-to-peer* architecture, which lacks a centralized service provider).

Server systems, on the other hand, hosts server programs whose resources are shared with all connected clients. Servers will generally wait for client requests prior to serving out resources.

Common systems that follow the client/server model include email exchanges, web access, and database access.

You may find the Wikipedia article of the [Client/Server Model](#) helpful in understanding more of the details.

Before You Begin ...

Before starting on this problem set, please ensure that you have Telnet installed. *nix operating systems should have telnet installed by default.

Windows users should first check if Telnet is installed by running the command "Telnet" in command line. If you do not have it, you can install it via Control Panel ---> Programs and Features ---> Turn windows features on/off ---> Telnet client.

You can have Telnet connect to a host/port (for example, "localhost:4444") from the command line with "telnet localhost 4444".

Alternatively you can open the Telnet program and connect from there with the command "open localhost 4444".

Overview

We will be creating two different Client/Server systems, namely an Echo system and a Prime Factors search system.

With the Echo system, our goal is to learn how users, clients, and servers interact with one another under the Java networking framework. The system will be very simple, taking in User input from Standard input, navigating through the entire network system, and finally returning the original User input back to the User via Standard output.

With the Prime Factors system, one of the focuses is to find the prime factors of a given input (which can be potentially very large). We will attempt to do this through a distributed system in the following fashion:

1. Start several servers dedicated to searching for prime factors in a given range of numbers.
2. Start a client that will send requests to these servers.
3. Accumulate the server responses in some meaningful manner.

You should take note that performance is not a criterion on this problem set. Brute-force search for factors is fine. The client is given multiple servers to communicate with and you should use as many servers as given to your program.

Do NOT use multithreading for this problem set. Each Server instance will handle at most one client at a single time. Though we will have multiple Servers running asynchronously, none of them will be run on the same thread.

We will be using the Java networking library for communication between our client and our servers. Please read about Streams (specifically about Buffered Streams), and Sockets before proceeding.

Echo System Specifications

The *Echo system messages* are defined below. Underlined terms are considered terminals in our grammar.

- User-to-Client messages define what the User can input on the standard input stream, consequently what the client reads in.

```
User-to-Client Echo Message Protocol
Valid-Input := String NewLine
String      := [^NewLine]+
NewLine     := \n
```

- Client-to-User messages define what the client outputs on the standard output stream, consequently what the user can read in console.

```
Client-to-User Echo Message Protocol
Valid-Input := Prefix Space String NewLine
Prefix      := >>>
String      := [^NewLine]+
Space       := " " // " " is a single space character
NewLine     := \n
```

- Client-to-Server messages define all messages that the client will send to the server for processing.

Server-to-Client messages define the processed output for the respective Client-to-Server message.

There are no special protocols for Client-to-Server or Server-to-Client messages for the Echo system.

```
Client-to-Server and Server-to-Client Echo Message Protocol
Valid-Input := String NewLine
String      := [^NewLine]+
NewLine     := \n
```

The *Echo Server* component is specified below.

- The server takes in at most one integer Program Argument, the port incoming clients will connect through.
 - If there is no Program Argument, default server to listen to port 4444
 - If server fails to listen on the assigned port, terminate the server.
 - In Eclipse, you can specify a Program Argument by creating a new Run Configuration --> Arguments Tab --> Write some port number in Program Arguments (example: 4444).
- Server should only handle ONE client at any given time. **Do not use any multithreading in your solution.**
- When client disconnects, server will listen for new incoming client connections on the same assigned port.
- When receiving one Client-to-Server message, the exact contents of this message are sent back to client in exactly one Server-to-Client message.

The *Echo Client* component is specified below.

- The client takes in exactly one Program Argument, the address of the server.
 - If there is no Program Argument, output a helpful error statement and terminate client.
- Client will read messages passed in by User from the standard input stream (`System.in`).
 - If user's standard input stream is closed, terminate the client.
- Client will push messages back out to user through the standard output stream (`System.out`).
- The message passing and processing are specified as follows:
 1. User-to-Client messages are read in from the standard input stream.
 2. Client passes these messages directly to the Server as Client-to-Server messages.
 3. Client listens for exactly one Server-to-Client messages.
 4. When this message is received, the contents of the message are sent as a Client-to-User message onto the standard output stream.

Problem 1: Setting up an Echo Server

We will first set up a server instance that accepts an incoming client connection and echoes all incoming statements from the user.

In your `EchoServer` class, there is a `main()` method where all Server setup logic should go. As with the example provided in the Java tutorial, you should set up a ServerSocket to listen for incoming client connections. Your `EchoServer` must follow the specifications indicated above.

When a client connects, your server should listen to input messages from the connected client and send it right back to the client over the connection.

Remember, we only require your server to handle one client at a time (NO Multithreading!). However, if the connected client disconnects, your server should go back to listening for any clients that want to connect.

You can manually check your server is working correctly as follows:

1. Run your `EchoServer` with a single program argument for some port number.
2. Open up telnet and connect to `localhost:[port number]` (example: "telnet localhost 4444", or "open localhost 4444")
3. If this connection fails, telnet will notify you.

4. If it succeeds, try typing some input. You should see the exact same input spit back at you.
5. Close telnet, and then use it again to reconnect to the server. The server should work for the second connection as well.

a. [15 points] Implement and test the EchoServer, which reads incoming messages and outputs the same message on the output stream. It should follow all the specifications defined in the earlier section.

Problem 2: Setting up an Echo Client

We will now write a client that can connect to our EchoServer.

In your EchoClient class, there is a main() method where all Client setup logic should go. As with the example provided in the Java tutorial, you should set up a Socket to connect and communicate to a Server.

You will be able to manually check your client is working by having a Server instance running in a process and attempting to connect to it with a new client instance.

The following is an example conversation between your User and EchoServer through the EchoClient, displayed through console:

```
Hello, I am EchoClient!
>>> Hello, I am EchoClient!
No, I am EchoClient!
>>> No, I am EchoClient!
You are exhausting to talk to.
>>> You are exhausting to talk to.
```

Note that all the statements preceded with ">>>" are a result of the message passing through the EchoServer back through the Client-to-User protocol.

a. [15 points] Implement and test the EchoClient. It should follow all the specifications defined in the earlier section.

Problem 3: Finding Prime Factors

We will now implement a Java function to find prime factors.

A rough pseudocode for solving this problem is provided. You may not modify the specifications defined below. However, you can improve on this algorithm however you choose.

```
@requires BigInteger N. such that 2 <= N
@requires BigInteger low, hi. such that 1 <= low <= hi
@effects finds all prime BigIntegers x
        such that low <= x <= hi AND x divides N evenly.
        Repeated prime factors will be found multiple times.

0. Given BigInteger N, BigInteger low, BigInteger hi:
1.   for x from lo to hi:
2.     if x is prime then
3.       while x divides evenly into N:
4.         add x to the collection of prime factors.
5.         N = N/x
```

You may use the BigInteger function isProbablePrime to approximate if a number is prime and remainder to check if one BigInteger fully divides another. See [BigInteger](#) for further details.

The following are a few example input/outputs for your function:

- (N= 85=5*17, lo= 2, hi=17) -> [5,17]

- (N= 85=5*17, lo= 2, hi=16) -> [5]
- (N= 85=5*17, lo= 2, hi=4) -> []
- (N= 264=2*2*2*3*11, lo= 2, hi=17) -> [2,2,2,3,11]

Note that your outputs do not have to be in any specific order.

a. [10 points] Implement and test the function to find all prime factors of a number, given the range of values to search through.

Prime Factors System Specifications

The *Prime Factors system messages* are defined below. As before, Underlined terms are considered terminals in our grammar.

- User-to-Client Echo Message Protocol
 - Valid-Input := Space N Space NewLine
 - N := [0-9]⁺
 - Space := (" ")* // Inner " " is a single space character
 - NewLine := \n

You can make the assumption that User input is valid only for N >= 2.

- Client-to-User Echo Message Protocol
 - Valid-Input := Prefix Space N Equals Factor (Mult Factor)* NewLine
 - | Invalid
 - Prefix := >>>
 - N := Number
 - Factor := Number
 - Invalid := invalid
 - Equals := =
 - Mult := *
 - Number := [0-9]⁺
 - Space := " " // " " is a single space character
 - NewLine := \n

The following are a few example Client-to-User messages:

- >>> 85=5*17
- >>> 1332425524=2*2*17*19594493

So print "invalid"

- Client-to-Server Message Protocol
 - Message := Factor Space N Space LowBound Space HighBound NewLine
 - Factor := factor
 - N := Number
 - LowBound := Number
 - HighBound := Number
 - Number := [0-9]⁺
 - Space := " " // " " is a single space character
 - NewLine := \n

You can make the assumption that the message is valid only for N >= 2.

The following are a few example Client-to-Server messages:

- factor 85 2 17
means find all prime factors of 85 between 2 and 17.
- factor 1332425524 2 16
means find all prime factors of 1332425524 between 2 and 16.

o factor 1 1 100

is NOT valid under this context because $N=1$.

- Server-to-Client Message Protocol

```

Protocol := Message*
Message  := Found Space N Space Factor NewLine
           | Done Space N Space LowBound Space HighBound NewLine
           | Invalid NewLine

Found    := found
Done     := done
Invalid  := invalid
N        := Number
Factor   := Number
LowBound := Number
HighBound := Number
Number   := [0-9]+
Space    := " " // " " is a single space character
NewLine  := \n
  
```

The following are a few example Server-to-Client messages:

- o found 85 5
means the server found 5 as a prime factor of 85.
- o found 85 17
means the server found 17 as a prime factor of 85.
- o done 85 2 17
means the server in charge of finding factors of 85 between 2 and 17 is complete.

The *Prime Factors Server* component is defined below.

- Server takes in at most one integer Program Argument, the port incoming clients will connect through.
 - o If there is no Program Argument, default server to listen to port 4444
 - o If server fails to listen on the assigned port, terminate the server.
- Server should only handle ONE client at any given time. **Do not use any multithreading in your solution.**
- When client disconnects, server listens for new incoming client connections on the same assigned port.
- Its functionality will be defined as:
 - o Server will listen for Client-to-Server messages. If it does not fit the protocol defined above, send a Server-to-Client "invalid" message.
 - o Server will send a Server-to-Client "found" message for each prime factors x of N such that $\text{LowBound} \leq x \leq \text{HighBound}$.
 - o When server has found all such x , it sends the Server-to-Client "done" message.

The *Prime Factors Client* component is defined below.

- Client will take in at least one Program Argument, the addresses of the PrimeFactorsServers your client will query.
 - o If there is no provided Program Argument, output a helpful error statement and terminate the client.
- Your client will read messages passed in by your User from the standard input stream (`System.in`).
 - o If the User's standard input stream is closed, you should terminate the Client.
- Your client will push messages back out to your User through the standard output stream (`System.out`).
- The entire procedure of message processing is defined as:
 1. User-to-Client messages are read in from the standard input stream, indicating the number to factor.
 2. Client passes Client-to-Server messages, one for each Server. Each message to the Server

- specifies a range of numbers, the sum of which covers all Integers from 2 through \sqrt{N} .
3. Client listens for Server-to-Client messages, each indicating a prime factor x .
 4. Client aggregates prime factors and sends the appropriate Client-to-User message. If the client receives malformed messages or data, it will send the invalid message.

Problem 4: Making a Server to do Prime Factor Searching

The intention of this problem set is to solve this problem in a distributed fashion. We can do so by breaking the problem down into smaller subproblems, as alluded to in the previous problem.

We will create a server that takes in messages indicating three numbers N , LowBound , HighBound .

It will respond with a message for every x where x is a prime factor of N and $\text{LowBound} \leq x \leq \text{HighBound}$.

The Client-to-Server and Server-to-Client Message protocols are defined in the Specifications section above.

You can first check that this works by communicating with your server through telnet. A sample conversation may be as follows:

```
factor 264 2 5
found 264 2
found 264 2
found 264 2
found 264 3
done 264 2 5
factor 1 2 5
invalid
what? why??
invalid
```

(The italics may not be rendered in your Telnet. It is simply an indication of output from the Server.)

Your Server does NOT need to respond with factors in any specific order.

a. [30 points] Implement and test the PrimeFactorsServer to listen for incoming Client connections. You should follow all of PrimeFactorsServer specifications defined in the previous section. You should use the function you defined in Problem 3.

Problem 5: Integrating your Client with Multiple Servers

We will now create a client that will make requests to servers it is connected to, to get back all prime factors.

We have to first define our search space. In fact, we only have to look for prime factors between 2 and \sqrt{N} , as there is guaranteed to be *at most one* prime factor greater than \sqrt{N} . Accordingly, to extend the algorithm presented in Problem 3, you can do the following:

```
1.   for x from 2 to sqrt(N):
2.       if x is prime then
3.           while x divides evenly into N:
4.               x is a prime factor!
5.               N = N/x
6.   if N != 1 then
7.       N is a prime factor!
```

You can find a BigInteger `sqrt` function in the provided `util.BigMath` class.

You should partition your search space in some manner across all servers that your client is connected to.

For example, if your client is connected to three servers, you can simply divide the range $[2, \sqrt{N}]$ in

three even parts, $[2, \sqrt{N}/3]$, $[\sqrt{N}/3 + 1, 2*\sqrt{N}/3]$, $[2*\sqrt{N}/3 + 1, \sqrt{N}]$

When your client has accumulated all of the prime factors, it should display them according to the Client-to-User protocol.

A sample conversation is shown:

```
264
>>> 264=2*2*11*3*2
      1332425524
>>> 1332425524=2*2*17*19594493
cool!
>>> invalid
```

a. [30 points] Implement and test the PrimeFactorsClient. It should obey the above User-to-Client, Client-to-User protocols when interacting with Standard input/output. It should also obey the Server/Client protocols to send and obtain necessary information.

Telnet: virtual terminal

1969

Command line interface

SSL used instead

2 systems: echo and prime factor

No multithreading

Do server lot

- copy from example code

Weird rep invariant \neq null ...

Uses Java Server Socket

While true so goes back to listening afterwards

Client

need to write terminal code

That was easy

②

Now to get it to test

- Take input in Eclipse

Oh it works lot try

but correct reset at end

'Should server disconnect'

- not normally

finally - when try exits

- always executes even if exception occurs

Need it to loop

'Should we keep client open

- its not very carefully

I think its done

But ~~chris~~ didn't do tests and such

- do later

(3)

Prime Factors

need to calc

{ if x is prime {

 Oh use is Probable prime

" x divides evenly into N "

{ $\frac{N}{x}$ {

{ Use same message system {

 Or copy + paste

 - think copy + paste.

~~Should~~ Should write some tests

(working very slowly - Fri night)

Oh here is reverse ^{^^} - distributed system

No its same

diff to implement client w/ multiple servers

Then need to do their weird grammar

(not hard - time consuming)

(4)

11/5

Where was I?

Adding server components

Sends ~~low~~ back found for each

User enters N

Client sends "factor" N Low High

How does client pick?

2 through \sqrt{N}

Need to parse on server

- split w/ spaces

Go out is per line result?

- Sends found line for each

- silly ...

i must send in real line?

Oh saw I've Java's for each

5

Now test

So its only going to 10 for 85
- but that is $\sqrt{85}$

? So this is right ?

Oh client is supposed to aggregate results

I think \sqrt{N} is wrong...

L no posts on piazza implies its correct

~~How to aggregate results~~

$$\text{PrimeFactor}(85) = 5 \times 17$$

^larger than 10

$$\text{factor}(264) = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 11$$
$$\sqrt{264} = 16$$

Ok do multi server now

L oh only 1 prime factor greater than \sqrt{N}

Lahhh thats why

find last one on client??

6

Wait is other one N?

and are more

So $\frac{85}{5} = 17$ gets it?

but does it while again

Jon

- for last one (the missing one)

on client

take every result

~~consider~~ $\frac{N}{\text{result}} \neq 1$

add to list

the division

does order matter don't think so

See if Piazza says same thing

(P-set is easy - just annoying to figure out what they want)

7

Now divide up by servers
 L_k

$$\left[2, \frac{\sqrt{N}}{k} \right] \left[\frac{\sqrt{N}}{k} + 1, 2 \frac{\sqrt{N}}{k} \right] \left(\frac{2\sqrt{N}}{k} + 1, \frac{3\sqrt{N}}{k} + 1 \right) \dots$$

How to create list of servers?

Define arg

- or fake it in main method?

Oh some ints can be Big - need to fix

So : start several servers?

like w/ arguments?

write test code to start a bunch

- but no real constructor

- ~~but~~ but don't change main's purpose

so just have main call constructor!

↳ no already have constructor

- can ignore main method

⑧ Its the 5 in Factors I always leave at
But might have to move some server logic out of client's main
So one client can talk to multiple servers

Like I'm not sure if doing Exceptions right
and no way to know!

~~Some~~ So many things - don't know if doing right!

Actually trying more stuff now

Can have multiple catch blocks

Type casting problem grr

Urr just change to it ...

So have 1 client - or multiple clients?

Oh can take std in here

Oh test needs multiple threads?

Need to read up on that

⑨

On all this stuff needs to be Big Int

So no multi threading - so one after the other?

Need to aggregate multiple server replies and

For loop (servers)

while true (replies)

Ok seems to be back working

Big Integer so annoying!

Now done not running right....

Fixed

Testing

Need to get the multiple servers working

- can we use threads?

Asked Jon - can do threads

Trouble to get working!

I did diff stuff than he did - - -

10

Still waiting somewhere...

Jwang is telling me to do it his way
annoying

- Just using main

(backwards to what we normally do!

Oh I converted too much! to err not at

Was never starting servers

thought it was weird no debug code!

Now requests ^{low, high} all wrong!

↳ But finally in relm of can figure it out

Oh rounding problems

I guess it gets rounded up

Oh so first one does not do much

Since algatm does not account for 2 missing

Order of opp error:

Had + on wrong one

①

Low can = high

Now array out of bounds exception

Oh blank request - why?

Oh newline being sent ...

Ok fixed

Fixed null pointer

Now why is it wrong!

- breaking at wrong time

Oh the delete line

⊙ Ahh right :) working

But some random character here

Works!

Now what else - make sure grammar nice
and some more tests

? Can't do > 1 test?

? Any way to see open threads?

(12)

One thing not connecting right

So $2^{64}/2 = 132$ — but not a prime factor

‘Need to keep on it’

‘test afterwards’

Put $\frac{N}{x}$ test back on server?

and \neq original N

But getting some duplicates now?

Should do check at end

Dividing N each time

Now concurrent mod issues

— but should be no other threads

Just means can be bad

Still weird

‘Only add on last one’

(13)

Why does large int ore die?

regular ore does not return error

Oh that is final val

Or not returning no results properly

- no

Stop val too small

- Nope works here -..

Oh was not waiting enough

✓ works

Now why can't it run more than 1 test at once

in is not available -..

So server should close on null

Console is never over - blocks - never null

Ok fixed

- Done now need to make sure it all works .

- And tests - more

Jon did more unit testing - I concentrated on system

Ah unicode works!

Echo's reply does not work!

(14)

Needs a return char?

Oh it was actually never received back

Then why in server

Now seems to be working in real - but not in test...

I was setting up client twice

Now need to quit on null

Ok done

Now some tests writing to wrong out

Oh need to replace

Do I care blank lines don't echo?

+ means 0 or 1?

or more than 1? 

So should be no result

I think I am done whole P-set!

6.005
CS6.Synchronization

11/17

☐ locks

☐ monitor pattern

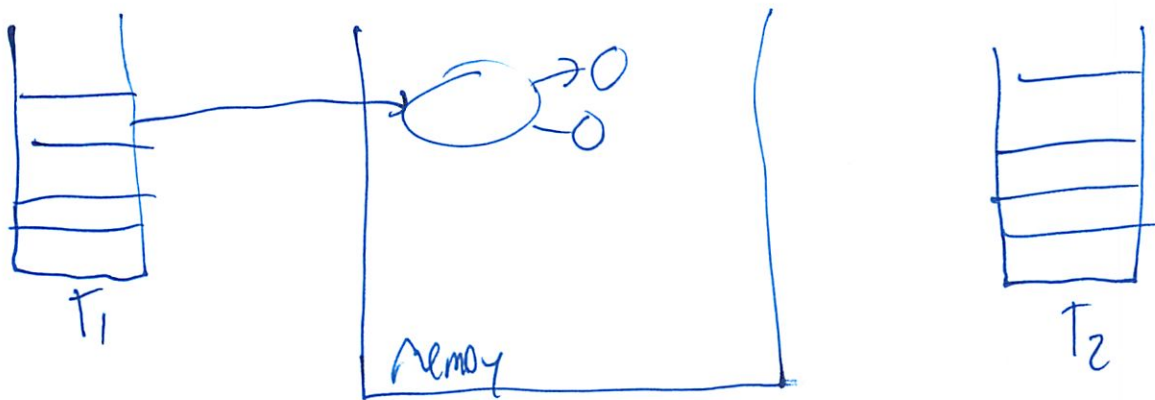
☐ deadlock

☐ locking disciplines

PS6 at, due Thur

Use what we talk about today

- make shared mutable data type safe



A few ways to make safe

1. Confinement

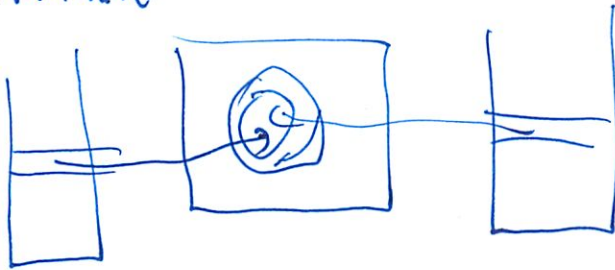
- only safe if no references in other thread

- or ~~new~~ kinda safe: other party agrees not to change

- static - (w/ public methods) everyone has access

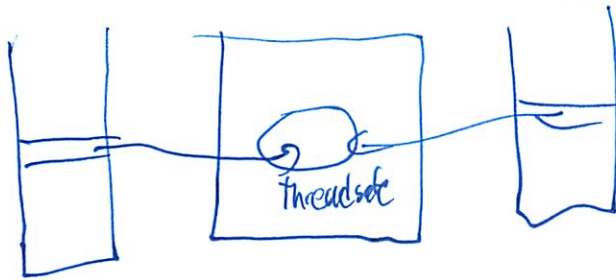
②

2. Immutable



all you can do is read!
no ~~can~~ race conditions possible

3. Thread-safe mutable data types



Synchronized wrappers

Blocking Queues

'Today': make your own data type safe

If building Google Docs

- need mutable, multiuser docs

Create ADT Data Buffer
 _{Abstract}

③

Can

~~Atk~~ - insert
- see
- data) in datatype

1. Specify Edit Buffer interface
- write methods sigs

2. Test
- diff type of strings
- diff states
- get small set datatypes that cover

3. Implement
- constructors (couldn't put in interface due to Java)
- ~~rep~~ choose rep
- could be brute force way to do
- try that first so can test ADT
- is it feasible
- do test cases work
- ~~test team~~ team can use in int'ion
- here String Builder
- but not thread safe
- then do more complex rep

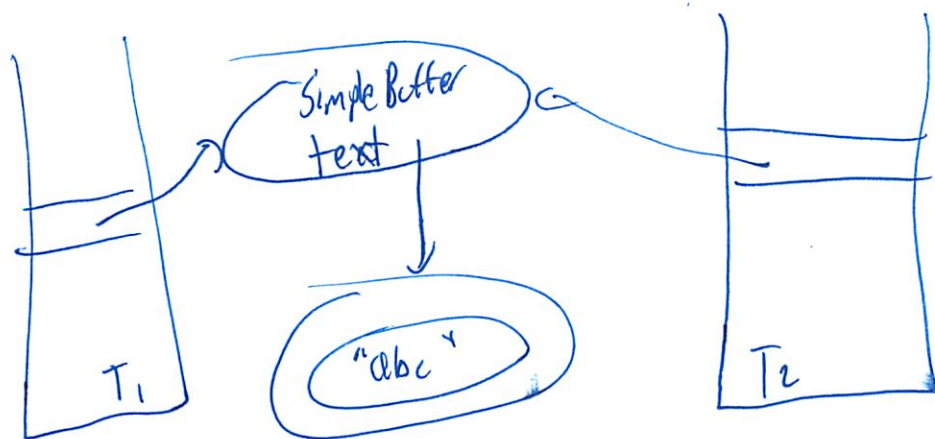
9

Single Threaded list

- keep life simple while debugging
- multi-threaded debugging "nightmare"
- it logic does not work in single threaded - it won't work in multithreaded

The picked String as rep

~~Can~~ insert, construct new string from old parts



Is it ThreadSafe?

is using 'immutable' objects

But the Simple Buffer is mutable!

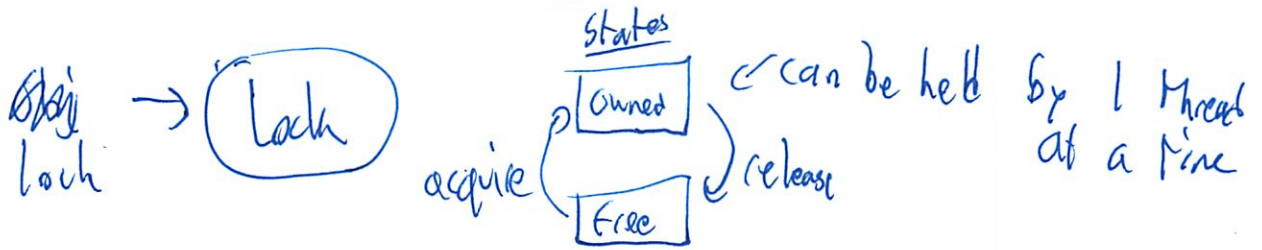
- can't mark it final
- CPU want to mutate it

Same problem as w/ cash machine

⑤ "Text" gets reassigned

So how to fix?

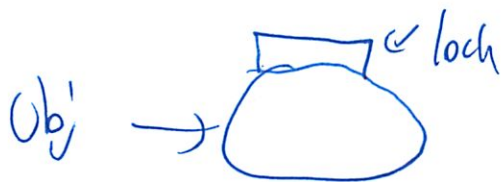
In many ~~cases~~ ^{scenarios} → locking



acquire blocks if another thread owns lock permanently

In Java

- locks are automatically attached to every object



- proof: they prob regret this now
- ya still need discipline to treat lock in certain way
- allows us to achieve temp. confinement
 - only thread currently holding lock can touch it

6

So we `Synchronized(obj) {`
↳ we use that lock 

Arbitrary Java code
`balance = balance + 1;`
`foo();`
↳ can call methods
(which are still synced)

When thread tries to enter sync it tries to
acquire lock

When T2 tries to enter sync (in use) T2 blocks
until lock is acquired

So we have temp confinement

So for example - we'll use lock on Simple Buffer
↳ locks all stuff below

`Synchronized(this)`

⑦

Must do both to mutators and observers

We want atomic operations

↳ low level stuff happens as a unit
- indivisible

[Jump to - hold CTRL + Click]

A thread ~~can~~^{will} reacquire a lock it already has

- will just pass through w/o problems

- exiting won't release

↳ its counting # of syncs

Must be disciplined to put lock everywhere

Can also put "synchronized" before method title

↳ will do same thing as sync block

- but can't do it on constructors

- Object should be created as you are building

- you have not shared references

- but not always true

- you can still sync on constructor, but not w/ keyword

8) Can still have problems on client
find Replace (Edit Buffer, s, t)

- all ops would happen correctly
 - atomic w/ each other
 - but no protection across multiple operations in same fn
 - must make entire thing atomic
 - buffer can offer its lock
- ```
Synchronize (buffer) {
 ;
}
```

- but made mistake using start, end
  - would want ~~buffer~~ cursor object
  - if people change other text will it change what you are highlighting

---

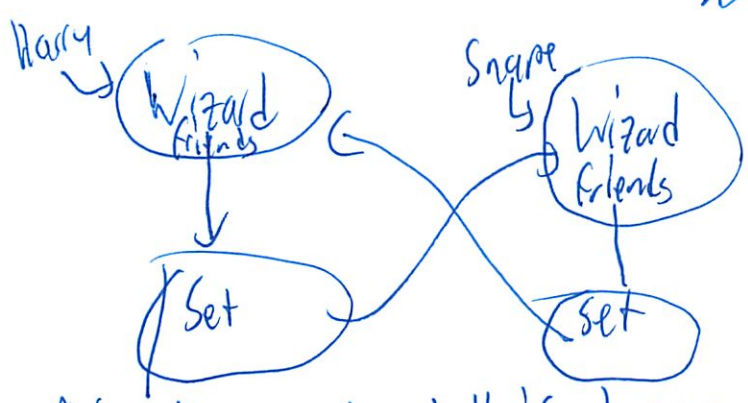
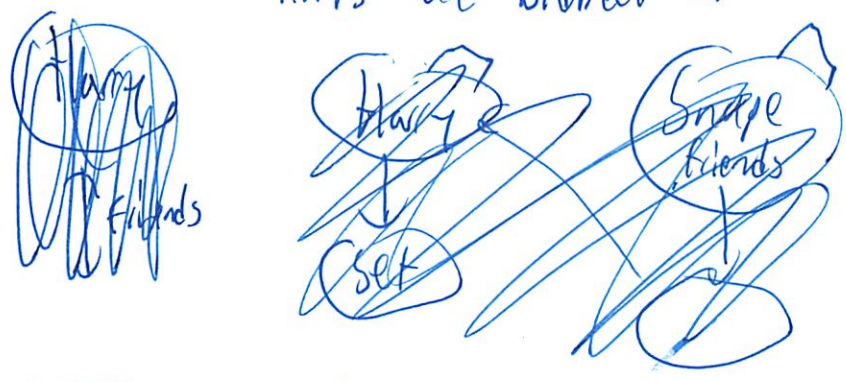
But we've created A thread lock

- threads are now blocking
  - threads can be waiting on each other
- thread A waiting on B  
          B                  A

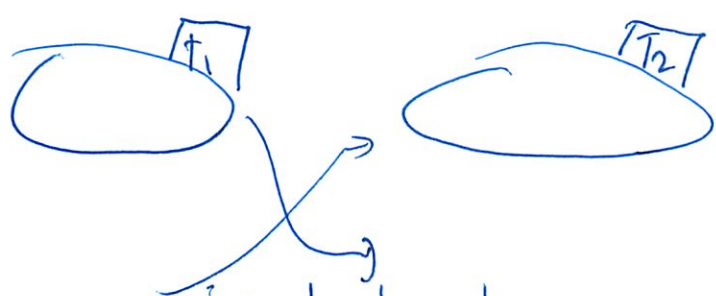
9

# Wizard example

- links are bidirectional



Normally works well if 1 person friends the other - updates both list of friends  
If Harry and Snape both try to friend each other at same time -> deadlock



both trying to acquire the other's lock  
neither will ever exit + release  
happens when reciprocal updates



## ⑩ Solutions

1. One big lock

- ~~A~~ Very safe
- but no parallelism
- Why bother having threads ~~at~~ either

2. Lock ordering

- difficult
- Force locks to be acquired in a certain order  
ie by alpha order
- but you generally don't know which locks to acquire
- used inside OS kernels

## L13: Synchronization

### Today

- Making a datatype threadsafe
- Locks
- Monitor pattern
- Deadlock
- Locking disciplines

### Required reading (from the Java Tutorial)

- [Synchronization](#)

### Optional reading

The material in this lecture is inspired by an excellent book:

- Brian Goetz et al. *Java Concurrency in Practice*. Addison-Wesley, 2006.

### Review

Recall the four ways to make data access safe in shared-memory concurrency:

- **Confinement:** don't share the data between threads. Ensure that only one thread has access to it.
- **Immutability:** make the shared data immutable.
- **Threadsafe datatype:** use a datatype that does the coordination for you, like a `BlockingQueue` or a synchronized collection wrapper.
- **Synchronization:** when the data has to be shared between threads, keep two threads from accessing it at the same time.

We talked about the first three in the last lecture. Synchronization is our topic for today. We'll look at it in the context of designing a threadsafe abstract datatype.

### Developing a Datatype for a Multiuser Editor

Suppose we're building a multi-user editor, like Google Docs, that allows multiple people to connect to it and edit it at the same time. We'll need a mutable datatype to represent the text in the document. Here's the interface; basically it represents a string with insert and delete operations.

```
/** An EditBuffer represents a threadsafe mutable string of characters in a
text editor. */
public interface EditBuffer {
 /**
 * Modifies this by inserting a string.
 * @param i position to insert (requires 0 <= pos <= current buffer length)
 * @param s string to insert
 */
 public void insert(int pos, String s);
}
```

```

/**
 * Modifies this by deleting a substring
 * @param pos start of substring to delete
 * (requires $0 \leq \text{pos} \leq \text{current buffer length}$)
 * @param len length of substring to delete
 * (requires $0 \leq \text{len} \leq \text{current buffer length} - \text{pos}$)
 */
public void delete(int pos, int len);

/**
 * @return length of text sequence in this edit buffer
 */
public int length();

/**
 * @return content of this edit buffer
 */
public String toString();
}

```

A very simple rep for this datatype would just be a string:

```

public class SimpleBuffer implements EditBuffer {
 private String text;
 // Rep invariant:
 // text != null
 // Abstraction function:
 // represents the sequence text[0], ..., text[text.length()-1]
}

```

The downside of this rep is that every time we do an insert or delete, we have to copy the entire string into a new string. That gets expensive. Another rep we could use would be a char array, with space at the end. That's fine if the user is just typing new text at the end of the document (we don't have to copy anything), but if the user is typing at the beginning of the document, then we're copying the entire document with every keystroke.

A more interesting rep, which is used by many text editors in practice, is called a **gap buffer**. It's basically a char array with extra space in it, but instead of having all the extra space at the end, the extra space is a **gap** that can appear anywhere in the buffer. Whenever an insert or delete operation needs to be done, the datatype first moves the gap to the location of the operation, and then does the insert or delete. If the gap is already there, then nothing needs to be copied – an insert just consumes part of the gap, and a delete just enlarges the gap! Gap buffers are particularly well-suited to representing a string that is being edited by a user with a cursor, since inserts and deletes tend to be focused around the cursor, so the gap rarely moves.

```

/** GapBuffer is a nonthreadsafe EditBuffer that is optimized for editing with
 * a cursor, which tends to make a sequence of inserts and deletes at the same
 * place in the buffer. */
public class GapBuffer implements EditBuffer {
 private char[] a;
 private int gapStart;
 private int gapLength;
 // Rep invariant:
 // a != null
 // $0 \leq \text{gapStart} \leq \text{a.length}$
 // $0 \leq \text{gapLength} \leq \text{a.length} - \text{gapStart}$
 // Abstraction function:
}

```



```
// represents the sequence a[0], ..., a[gapStart-1],
// a[gapStart+gapLength], ..., a[length-1]
```

In a multiuser scenario, we'd want multiple gaps, one for each user's cursor, but we'll use a single gap for now.

## Steps to Developing the Datatype

Recall our recipe for designing and implementing an ADT:

1. **Specify.** Define the operations (method signatures and specs). We did that in the EditBuffer interface.
2. **Test.** Develop test cases for the operations. See EditBufferTest in the provided code. The test suite includes a testing strategy based on partitioning the parameter space of the operations.
3. **Rep.** Choose a rep. We chose two of them for EditBuffer, and this is often a good idea:
  - a. **Implement a simple, brute-force rep first.** It's easier to write, you're more likely to get it right, and it will validate your test cases and your specification so you can fix problems in them before you move on to the harder implementation. This is why we implemented SimpleBuffer before moving on to GapBuffer. Don't throw away your simple version, either – keep it around so that you have something to test and compare against in case things go wrong with the more complex one.
  - b. **Write down the rep invariant and abstraction function, and implement checkRep().** checkRep() asserts the rep invariant at the end of every constructor, producer, and mutator method. (It's typically not necessary to call it at the end of an observer, since the rep hasn't changed.) In fact, assertions can be very useful for testing complex implementations, so it's not a bad idea to also assert the postcondition at the end of a complex method. You'll see an example of this in GapBuffer.moveGap() in the code with this lecture.

In all these steps, we're working entirely single-threaded at first. Multithreaded clients should be in the back of our minds at all times while we're writing specs and choosing reps (we'll see later that careful choice of operations may be necessary to avoid race conditions in the *clients* of your datatype). But get it working, and thoroughly tested, in a sequential, single-threaded environment first.

Now we're ready for the next step:

4. **Synchronize.** Make an argument that your rep is threadsafe. Write it down explicitly as a comment in your class, right by the rep invariant, so that a maintainer knows how you designed thread safety into the class.

This lecture is about how to do step 4.

## Examples of Thread Safety Arguments

Let's see some examples of how to make thread safety arguments for a datatype. Remember our four approaches to thread safety: confinement, immutability, threadsafe datatypes, and synchronization.

**Confinement** is not usually an option when we're making an argument just about a datatype, because you have to know what threads exist in the system and what objects they've been given access to. If the datatype creates its own set of threads (like the Crawler datatype we used last lecture), then you can talk about confinement with respect to those threads. Otherwise, the threads are coming in from the outside, carrying client calls, and the datatype may have no guarantees about which threads have references to what. So confinement isn't a useful argument in that case. Usually we use confinement at a higher level, talking about the system as a whole and arguing why we don't

need thread safety for some of our modules or datatypes, because they won't be shared across threads by design.

Immutability is often a useful argument:

```
public class String {
 private final char[] a;
 // thread safety argument:
 // This class is threadsafe because it's immutable:
 // - a is final
 // - a points to a mutable char array, but that array is encapsulated
 // in this object, not shared with any other object or exposed
 // to a client.
```

Here's another rep for String that requires a little more care in the argument:

```
public class String {
 private final char[] a;
 private final int start;
 private final int len;
 // rep invariant:
 // a != null, 0 <= start <= a.length, 0 <= len <= a.length - start
 // abstraction function:
 // represents the string of characters
 // a[start], ..., a[start + len - 1]
 // thread safety argument:
 // This class is threadsafe because it's immutable:
 // - a, start, and len are final
 // - a points to a mutable char array, which may be shared with
 // other String objects, but they never mutate it.
 // The array is never exposed to a client.
```

Note that since this String rep was designed for sharing the array between multiple String objects, we have to ensure that the sharing doesn't threaten its thread safety. As long as it doesn't threaten the String's immutability, however, we can be confident that it won't threaten the thread safety.

We also have to avoid rep exposure. Rep exposure is bad for any datatype, since it threatens the datatype's rep invariant. It's also fatal to thread safety.

## Bad Arguments

Here are some incorrect arguments for thread safety:

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
 private String text;
 // Rep invariant:
 // text != null
 // Abstraction function:
 // represents the sequence text[0], ..., text[text.length() - 1]
 // Thread safety argument:
 // text is an immutable (and hence threadsafe) String,
 // so this object is also threadsafe
```

Why doesn't this argument work? String is indeed immutable and threadsafe; but the rep pointing to that string, specifically the text variable, is **not** immutable. Text is not a final variable, and in fact it



can't be in this datatype, because we need the datatype to support insertion and deletion operations. So reads and writes of the text variable itself are not threadsafe. This argument is false.

Here's another broken argument:

```
public class Graph {
 private final Set<Node> nodes =
 Collections.synchronizedSet(new HashSet<Node>());
 private final Map<Node,Node> edges =
 Collections.synchronizedMap(new HashMap<Node,Node>());
 // rep invariant:
 // nodes, edges != null
 // if edges maps x -> y, then nodes contains x and y
 // abstraction function:
 // represents a directed graph whose nodes are the set of nodes
 // and edges are the x->y pairs in edges
 // thread safety argument:
 // - nodes and edges are final, so those variables are immutable
 // and threadsafe.
 // - nodes and edges point to threadsafe set and map datatypes
```

This is a graph datatype, which stores its nodes in a set and its edges in a map. (Quick quiz: is Graph a mutable or immutable datatype? What do the final keywords have to do with its mutability?)

Graph relies on other threadsafe datatypes to help it implement its rep – specifically the threadsafe Set and Map wrappers that we talked about last lecture. That helps a lot to prevent race conditions, but it doesn't provide a sufficient guarantee, because the graph's rep invariant includes a relationship *between* the node set and the edge map: all nodes that appear in the edge map also have to appear in the node set. So there may be code like this:

```
public void addEdge(Node from, Node to) {
 edges.put(from, to);
 nodes.add(from);
 nodes.add(to);
}
```

which would produce a race condition – a moment when the rep invariant is violated. Even though the threadsafe set and map datatypes guarantee that their own add() and put() methods are atomic and noninterfering, they can't extend that guarantee to interactions *between* the two data structures. So the rep invariant of the Graph is not safe from race conditions. Just using immutable and threadsafe-mutable datatypes is not sufficient when the rep invariant depends on relationships between objects in the rep.

## Locking

So we need a way for a datatype to protect itself from interfering access by multiple threads. A popular synchronization technique, so common that Java provides it as a built-in language feature, is **locking**.

A lock is an abstraction that allows at most one thread to own it at a time. Locks have two operations: **acquire** (to own the lock for a while) and **release** (to allow another thread to own it). If a thread tries to acquire a lock currently owned by another thread, then it blocks until the other thread releases the lock and the thread successfully acquires it.

In Java, every object has a lock implicitly associated with it – a String, an array, an ArrayList, and every class you create, all their object instances have a lock. Even a humble Object has a lock, so bare Objects are often used for explicit locking:



```
Object lock = new Object();
```

You can't call acquire and release on that lock directly, however. Instead you use the **synchronized** statement to acquire a lock for the duration of a statement block.

```
synchronized (lock) { // thread blocks here until lock is free
 // now this thread has the lock
 balance = balance + 1;
 // exiting the block releases the lock
}
```

Synchronized regions like this provide **mutual exclusion**: only one thread at a time can be in a synchronized region guarded by a given object's lock. In other words, you are back in sequential programming world, with only one thread running at a time, at least with respect to other synchronized regions that refer to the same object.

Locks are used to **guard** a shared data variable, like the account balance shown here. If all accesses to a data variable are guarded (surrounded by a synchronized block) by the same lock object, then those accesses will be guaranteed to be atomic – uninterrupted by other threads.

## Monitor Pattern

When you are writing methods of a class, the most convenient lock is the object instance itself, i.e. **this**. As a simple approach, we can guard the entire rep of a class by wrapping all accesses to the rep inside `synchronized(this)`.

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
 private String text;
 ...
 public SimpleBuffer() {
 synchronized (this) {
 text = "";
 checkRep();
 }
 }

 public void insert(int pos, String s) {
 synchronized (this) {
 text = text.substring(0, pos) + s + text.substring(pos);
 checkRep();
 }
 }

 public void delete(int pos, int len) {
 synchronized (this) {
 text = text.substring(0, pos) + text.substring(pos+len);
 checkRep();
 }
 }

 public int length() {
 synchronized (this) {
 return text.length();
 }
 }
}
```

```

 public String toString() {
 synchronized (this) {
 return text;
 }
 }
 }
}

```

Note the very careful discipline here. *Every* method that touches the rep must be guarded with the lock – even apparently small and trivial ones like `length()` and `toString()`. This is because reads must be guarded as well as writes – if reads are left unguarded, then they may be able to see the rep in a partially-modified state.

This approach is called the **monitor pattern**. A monitor is a class whose methods are mutually exclusive, so that only one thread can be inside the class at a time.

The monitor pattern is so useful in Java that Java provides some syntactic sugar for it. If you add the keyword `synchronized` to the method signature, then Java will act as if you wrote `synchronized(this)` around its method body. So the code below is an equivalent way to implement the `synchronized` `SimpleBuffer`:

```

/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
 private String text;
 ...
 public SimpleBuffer() {
 text = "";
 checkRep();
 }

 public synchronized void insert(int pos, String s) {
 text = text.substring(0, pos) + s + text.substring(pos);
 checkRep();
 }

 public synchronized void delete(int pos, int len) {
 text = text.substring(0, pos) + text.substring(pos+len);
 checkRep();
 }

 public synchronized int length() {
 return text.length();
 }

 /** @see EditBuffer#toString */
 public synchronized String toString() {
 return text;
 }
}

```

Notice that the `SimpleBuffer` constructor doesn't have a `synchronized` keyword. Java actually forbids it, syntactically, because an object under construction should be confined to a single thread until it has returned from its constructor. So synchronizing constructors should be unnecessary.

```

/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {

```

## Thread Safety Argument with Synchronization

Now that we're protecting SimpleBuffer's rep with a lock, we can write a better thread safety argument:

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
 private String text;
 // Rep invariant:
 // text != null
 // Abstraction function:
 // represents the sequence text[0],...,text[text.length()-1]
 // Thread safety argument:
 // all accesses to text happen within SimpleBuffer methods,
 // which are all guarded by SimpleBuffer's lock
}
```

The same argument works for GapBuffer, if we use the monitor pattern to synchronize all its methods.

Note that the encapsulation of the class, the absence of rep exposure, is very important for making this argument. If text were public:

```
public String text;
```

then clients outside SimpleBuffer would be able to read and write it *without* knowing that they should first acquire the lock, and SimpleBuffer would no longer be threadsafe.

## Locking Discipline

1. Every shared mutable variable must be guarded by some lock. The data may not be read or written except inside a synchronized block that acquires that lock.
2. If an invariant involves multiple shared mutable variables (which might be in different objects), then all the variables involved must be guarded by the **same** lock. The invariant must be satisfied before releasing the lock.

The monitor pattern satisfies both rules. All the shared mutable data in the rep – which the rep invariant depends on -- is guarded by the same lock.

## Giving Clients Access to a Lock

It's sometimes useful to make your datatype's lock available to clients, so that they can use it to implement higher-level atomic operations using your datatype. For example, consider a find and replace operation on the EditBuffer datatype:

```
/* Modifies buf by replacing the first occurrence of s with t.
 * If s not found in buf, then has no effect.
 * @returns true if and only if a replacement was made
 */
public static boolean findReplace(EditBuffer buf, String s, String t) {
 int i = buf.toString().indexOf(s);
 if (i == -1) {
 return false;
 }
 buf.delete(i, s.length());
 buf.insert(i, t);
 return true;
}
```



```
}
```

This method makes three different calls to `buf` – to convert it to a string in order to search for `s`, to delete it, and then to insert `t` in its place. Even though each of these calls individually is atomic, the `findReplace` method as a whole is not threadsafe, because other threads might mutate the buffer while `findReplace()` is working, causing it to delete the wrong region or put the replacement back in the wrong place.

To prevent this, `findReplace()` needs to synchronize with all other clients of `buf`. One approach is to simply document that clients can use the `EditBuffer`'s lock to synchronize with each other:

```
/** An EditBuffer represents a threadsafe mutable string of characters
 in a text editor. Clients may synchronize with each other using
 the EditBuffer object itself. */
public interface EditBuffer {
 ...
}
```

And then `findReplace()` can synchronize on `buf`:

```
public static boolean findReplace(EditBuffer buf, String s, String t) {
 synchronized (buf) {
 int i = buf.toString().indexOf(s);
 if (i == -1) {
 return false;
 }
 buf.delete(i, s.length());
 buf.insert(i, t);
 return true;
 }
}
```

The effect of this is to enlarge the synchronization region that the monitor pattern already put around the individual `toString()`, `delete()`, and `insert()` methods, into a single atomic region that ensures that all three methods are executed without interference from other threads.

## Sprinkling Synchronized Everywhere?

So is thread safety simply a matter of putting the `synchronized` keyword on every method in your program? Unfortunately not.

First, you actually don't want to synchronize methods willy-nilly. Synchronization imposes a large cost on your program. Making a synchronized method call may take significantly longer, because of the need to acquire a lock (and flush caches and communicate with other processors). Java leaves many of its mutable datatypes unsynchronized by default exactly for these performance reasons. When you don't need synchronization, don't use it.

Second, it's not sufficient. Dropping *synchronized* onto a method without thinking means that you're acquiring a lock without thinking about which lock it is, or about whether it's the *right* lock for guarding the shared data access you're about to do. Suppose we had tried to solve `findReplace()`'s synchronization problem simply by dropping `synchronized` onto its method:

```
public static synchronized boolean findReplace(EditBuffer buf, ...) {
```

This wouldn't do what we want. It would indeed acquire a lock -- because `findReplace` is a static method, it would acquire a static lock for the whole class that `findReplace` happens to be in, rather than an instance object lock. As a result, only one thread could call `findReplace()` at a time -- even if other threads want to operate on *different* buffers, which should be safe, they'd still be blocked *until*

the single lock was free. So we'd suffer a significant loss in performance, because only one user of our massive multiuser editor would be allowed to do a find-and-replace at a time, even if they're all editing different documents. Worse, however, it wouldn't provide much protection, because other code that touches the document probably wouldn't be acquiring the same lock.

The synchronized keyword is not a panacea. Thread safety requires a discipline – using confinement, immutability, or locks to protect shared data. That discipline needs to be written down, or maintainers won't know what it is.

## Designing a Datatype For Concurrency

findReplace()'s problem can be interpreted another way: that the EditBuffer interface really isn't that friendly to multiple simultaneous clients. It relies on integer indexes to specify insert and delete locations, which are extremely brittle to other mutations. If somebody else inserts or deletes before the index position, then the index becomes invalid.

So if we're designing a datatype specifically for use in a concurrent system, we need to think about providing operations that have better-defined semantics when they are interleaved. For example, it might be better to pair EditBuffer with a Position datatype representing a cursor position in the buffer, or even a Selection datatype representing a selected range. Once obtained, a Position could hold its location in the text against the wash of insertions and deletions around it, until the client was ready to use that Position. If some other thread deleted all the text around the Position, then the Position would be able to inform a subsequent client about what had happened (perhaps with an exception), and allow the client to decide what to do. These kinds of considerations come into play when designing a datatype for concurrency.

As another example, consider the ConcurrentMap interface in Java. This interface extends the existing Map interface, adding a few key methods that are commonly needed as atomic operations on a shared mutable map, e.g.:

```
map.putIfAbsent(key,value) is an atomic version of
 if (!map.containsKey(key)) map.put(key, value);

map.replace(key, value) is an atomic version of
 if (map.containsKey(key)) map.put(key, value);
```

## Deadlock Rears its Ugly Head

The locking approach to thread safety is powerful, but (unlike confinement and immutability) it introduces blocking into the program. Threads must sometimes wait for other threads to get out of synchronized regions before they can proceed. And blocking raises the possibility of deadlock – a very real risk, and frankly far more common in this setting than in message passing with blocking I/O (where we first saw it last week).

With locking, deadlock happens when threads acquire multiple locks at the same time, and two threads end up blocked while holding locks that they are each waiting for the other to release. The monitor pattern unfortunately makes this fairly easy to do. Here's an example. Suppose we're modeling the social network of a series of books:

```
public class Wizard {
 private final String name;
 private final Set<Wizard> friends;
 // rep invariant:
 // name, friends != null
 // friend links are bidirectional:
```



```

// for all f in friends, f.friends contains this
// concurrency argument:
// threadsafe by monitor pattern: all accesses to rep
// are guarded by this object's lock

public Wizard(String name) {
 this.name = name;
 this.friends = new HashSet<Wizard>();
}

public synchronized boolean isFriendsWith(Wizard that) {
 return this.friends.contains(that);
}

public synchronized void friend(Wizard that) {
 if (friends.add(that)) {
 that.friend(this);
 }
}

public synchronized void defriend(Wizard that) {
 if (friends.remove(that)) {
 that.defriend(this);
 }
}
}

```

Like Facebook, this social network is bidirectional: if x is friends with y, then y is friends with x. The friend() and defriend() methods enforce that invariant by modifying the reps of both objects, which because they use the monitor pattern means acquiring the locks to both objects as well.

Let's create a couple of wizards:

```

Wizard harry = new Wizard("Harry Potter");
Wizard snape = new Wizard("Severus Snape");

```

And then think about what happens when two independent threads are repeatedly running:

```

// thread A // thread B
harry.friend(snape); snape.friend(harry);
harry.defriend(snape); snape.defriend(harry);

```

We will deadlock very rapidly. Here's why. Suppose thread A is about to execute harry.friend(snape), and thread B is about to execute snape.friend(harry). Thread A acquires the lock on harry (because the friend method is synchronized). Then thread B acquires the lock on snape (for the same reason). They both update their individual reps independently, and then try to call friend() on the other object – which requires them to acquire the lock on the other object. So A is holding Harry and waiting for Snape, and B is holding Snape and waiting for Harry. Both threads are stuck in friend(), so neither one will ever manage to exit the synchronized region and release the lock to the other. This is a classic deadly embrace. The program simply stops.

The essence of the problem is acquiring multiple locks, and holding some of the locks while waiting for another lock to become free.



## One Solution to Deadlock: Lock Ordering

One way to prevent deadlock is to put an ordering on the locks that need to be acquired simultaneously, and ensuring that all code acquires the locks in that order. This is a

In our social network example, we might always acquire the locks on the Wizard objects in alphabetical order by the wizard's name. Since thread A and thread B are both going to need the locks for Harry and Snape, they would both acquire them in that order: Harry's lock first, then Snape's. If thread A gets Harry's lock before B does, it will also get Snape's lock before B does, because B can't proceed until A releases Harry's lock again. The ordering on the locks forces an ordering on the threads acquiring them, so there's no way to produce a cycle in the waiting-for graph.

Here's what the code might look like.

```
public void friend(Wizard that) {
 Wizard first, second;
 if (this.name.compareTo(that.name) < 0) {
 first = this; second = that;
 } else {
 first = that; second = this;
 }

 synchronized (first) {
 synchronized (second) {
 if (friends.add(that)) {
 that.friend(this);
 }
 }
 }
}
```

(Note that the decision to order the locks alphabetically by the person's name would work fine for this book, but it wouldn't work in a real life social network. Why not? What would be better to use for lock ordering than the name?)

Although lock ordering is useful (particularly in code like operating system kernels), it has a number of drawbacks in practice. First, it's not modular – the code has to know about all the locks in the system, or at least in its subsystem. Second, it may be difficult or impossible for the code to know exactly which of those locks it will need before it even acquires the first one. It may need to do some computation to figure it out. Think about doing a depth-first search on the social network graph, for example – how would you know which nodes need to be locked, before you've even started looking for them?

## Another Approach: Coarse-Grained Locking

A more common approach than lock ordering, particularly for application programming (as opposed to operating system or device driver programming), is to use coarser locking – use a single lock to guard many object instances, or even a whole subsystem of a program.

For example, we might have a single lock for an entire social network, and have all the operations on any of its constituent parts synchronize on that lock. In the code below, all Wizards belong to a Castle, and we just use that Castle object's lock to synchronize:

```
public class Wizard {
 private final Castle castle;
 private final String name;
```

```

private final Set<Wizard> friends;
...
public void friend(Wizard that) {
 synchronized (castle) {
 if (this.friends.add(that)) {
 that.friend(this);
 }
 }
}
}

```

Coarse-grained locks can have a significant performance penalty. If you guard a large pile of mutable data with a single lock, then you're giving up the ability to access any of that data concurrently. In the worst case, having a single lock protecting everything, your program might be essentially sequential – only one thread is allowed to make progress at a time.

## Concurrency in Practice

What strategies are typically followed in real programs?

- **Library data structures** either use no synchronization (to offer performance to single-threaded clients, while leaving it to multithreaded clients to add locking on top) or the monitor pattern.
- **Mutable data structures with many parts** typically use either coarse-grained locking or thread confinement. Most graphical user interface toolkits follow one of these approaches, because a graphical user interface is basically a big mutable tree of mutable objects. Java Swing, the toolkit we'll be looking at in the next lecture, uses thread confinement. Only a single dedicated thread is allowed to access Swing's tree. Other threads have to pass messages to that dedicated thread in order to access the tree.
- **Search** often uses immutable datatypes. Our Sudoku SAT solver would be easy to make multithreaded, because all the datatypes involved were immutable. There would be no risk of either races or deadlocks.
- **Operating systems** often use fine-grained locks in order to get high performance, and use lock ordering to deal with deadlock problems.

We've omitted one important approach to mutable shared data because it's outside the scope of this course, but it's worth mentioning: a **database**. Database systems are widely used for distributed client/server systems like web applications. Databases avoid race conditions using *transactions*, which are similar to synchronized regions in that their effects are atomic, but they don't have to acquire locks, though a transaction may fail and be rolled back if it turns out that a race occurred. Databases can also manage locks, and handle locking order automatically. For more about how to use databases in system design, 6.170 Software Studio is strongly recommended; for more about how databases work on the inside, take 6.814 Database Systems.

## Summary

- Make thread safety arguments about your datatypes, and document them in the code
- Acquiring a lock allows a thread to have exclusive access to the data guarded by that lock, forcing other threads to block
- The monitor pattern guards the rep of a datatype with a single lock that is acquired by every method
- Blocking caused by acquiring multiple locks creates the possibility of deadlock, which can be solved by lock ordering or coarse-grained locking



## Threads

- everyone has access to plan
- in same class - inline kind
- pulling in same reference
- only one instance of  
we define
- Runnable class right there  
abstract
- ~~a subclass~~ like always  
nested class
- could also create separate MyRunnable
- plan would not be in scope
- would have to add to constructor

Subclass - is a  
not nested  
class

What if instantiate new instance inside a thread  
TravelPlan plan 2 = new TravelPlan()  
new Runnable() {

void run() {

TravelPlan plan 2 = new TravelPlan(); ← only this  
plan 1.update() ← same instance as main class } Thread has access

2 3



②

Race conditions possible in at Traveler code

Destination list won't have right things always

Since <sup>lots of</sup> threads running ~~one~~ at different times

Concurrent Modification Exception  $\neq$  w/ Iterator  
if someone tries to concurrently modify

Can add synchronize method to every instance method

What methods do we need to add to?

- getters and setters

? don't want to display old or <sup>during</sup> bad rep code  
depends on <sup>what</sup> halfway state is

- do you care if get midway

If over synchronizing - lose benefits of threads

3

So need to weigh pros + cons

Can only sync some operations on object

Synchronized ( ) {

}

So if have bar and baz

- each needs to be in sync with itself
- but they don't need to be updated w/ regards to each other

~~update~~

bar.update() ← sync this  
baz.update() ← sync this  
                    ↑  
                    its update method

} but not the method that calls it

Or

Synchronized(bar) { bar.update();

Synchronized(baz) { baz.update(); }

4

Can have Collections - synchronized()

Its very hard to make sure you are syncing everything that needs it

---

What if wanted to transfer method

- instance method
- removes from old travel plan
- adds to yours

Deadlock possibility

↳ two things waiting forever

could acquire locks in predetermined order

ordered by hash id - same unique int

not A(original) calling B(~~other~~ argument)

Could place global lock

- On parent object all objects are from



5

Wrong way

```
Synchronized (this) {
 Synchronized (other) {
```

```
 }
}
```

Want to do something w/ global lock order

↳ does not matter which is this or other

```
if (this.lockOrder() < other.lockOrder()) {
```

```
 lock 1 = this
 lock 2 = other
```

```
} else {
```

```
 lock 1 = other
```

```
 lock 2 = this
```

```
}
```

```
 Synchronized (lock 1) {
```

```
 Synchronized (lock 2) {
```

```
 }
 }
}
```

⑥ Synchronized keyword is equal to

Synchronized (this) {

3

```
package beforeclass;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
/**
 * SocialServer simulates a social networking web server
 * in which each user has a list of friends.
 * Clients make requests of the form:
 * Request ::= Person\n
 * Person ::= [^\n]+
 * and for each request, the server sends a reply listing the
 * person's friends, one per line, terminated by a blank line:
 * Reply ::= (Person\n)*\n
 * An unknown person returns an empty list of friends.
 */
```

```
//
// Thread safety argument
// -----
// The threads in the system are:
// - main thread accepting new connections
// - one thread per connected client, handling just that client
//
// The serverSocket object is confined to the main thread.
//
// The Socket object for a client is confined to that client's thread;
// the main thread loses its reference to the object right after starting
// the client thread.
//
// The friendsOf map and all the lists inside it are confined to the main thread
// during creation and then immutable after creation.
//
// System.err is used by all threads for displaying error messages.
// No other shared mutable data.
//
```

```
public class SocialServer {
 // default port number where the server listens for connections
 public static final int PORT = 3003;

 private final Map<String, List<String>> friendsOf = makeFriendsGraph();
```

) for Server

/ change  
for Nineserver



```

// rep invariant:
// serverSocket, friendsOf, and all references in friendsOf != null
// all strings in friendsOf contain no \n character

/**
 * Make a SocialServer.
 */
public SocialServer() {

}

/**
 * @return immutable graph of person -> list of friends
 */
private static Map<String, List<String>> makeFriendsGraph() {
 Map<String, List<String>> graph = new HashMap<String, List<String>>();
 add(graph, "Harry Potter", "Ron Weasley", "Hermione Granger", "Sirius Black");
 add(graph, "Ron Weasley", "Harry Potter", "Hermione Granger");
 add(graph, "Hermione Granger", "Ron Weasley", "Harry Potter");
 return Collections.unmodifiableMap(graph);
}

/**
 * modifies graph by adding a mapping from person to an immutable list of friends
 */
private static void add(Map<String, List<String>> graph, String person, String...
friends) {
 graph.put(person, Collections.unmodifiableList(Arrays.asList(friends)));
}

/**
 * Run the server, listening for client connections and handling them.
 * Never returns unless an exception is thrown.
 * @throws IOException if the main server socket is broken
 * (IOExceptions from individual clients do *not* terminate serve()).
 */
public void serve() throws IOException {
 ServerSocket serverSocket = new ServerSocket(PORT);

 while (true) {
 // block until a client connects
 final Socket socket = serverSocket.accept();

 // start a new thread to handle the connection
 Thread thread = new Thread(new Runnable() {
 public void run() {
 // the client socket object is now owned by this thread,
 // and mustn't be touched again in the main thread
 handle(socket);
 }
 });
 thread.start(); // IMPORTANT! easy to forget
 }
}

```

*kinda weak  
I don't know  
why need this...*

*main thread listen*

*each dispatcher  
to thread*

```

 // when does thread.start() return?
 // when will the thread stop?
 }
}

/**
 * Handle a single client connection. Returns when client disconnects.
 * @param socket socket where client is connected
 * @throws IOException if connection has an error or terminates unexpectedly
 */
private void handle(Socket socket) {
 try {
 System.err.println("client connected");

 // get the socket's input stream, and wrap converters around it
 // that convert it from a byte stream to a character stream,
 // and that buffer it so that we can read a line at a time
 BufferedReader in = new BufferedReader(new InputStreamReader(socket.
getInputStream()));

 // similarly, wrap character=>byte stream converter around the socket output
stream,
 // and wrap a PrintWriter around that so that we have more convenient ways to
write
 // Java primitive types to it.
 PrintWriter out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream
()));

 // in and out are thread-confined

 try {
 // each request from the client is a single line containing a person's name
 for (String person = in.readLine(); person != null; person = in.readLine()) {
 System.err.println("request: " + person);

 // look up the person's friends
 List<String> friends = friendsOf.get(person);
 if (friends != null) {
 for (String friend : friends) {
 System.err.println("reply: " + friend);
 out.print(friend + "\n");
 }
 }

 // send the terminating blank line
 out.print("\n");

 // VERY IMPORTANT! flush our buffer so the client gets the reply
 out.flush();
 }
 } finally {
 out.close();
 }
 }
}

```

*What we do w/ a connection*

*Actual interesting part*

*- wild reply*

```
 in.close();
 socket.close();
 }
} catch (IOException e) {
 e.printStackTrace();
}
}

/**
 * Start a SquareServer running on the default port.
 */
public static void main(String[] args) {
 try {
 SocialServer server = new SocialServer();
 server.serve();
 } catch (IOException e) {
 e.printStackTrace();
 }
}
}
```

set it up



## 6.005 Elements of Software Construction | Fall 2011

### Problem Set 6: Multiplayer Minesweeper

#### Due: Thursday, November 3 2011, 11:59 PM

---

The purpose of this problem set is to explore multithreaded programming with a shared mutable datatype, which you should protect using synchronization.

**You have substantial design freedom on this problem set.** However, in order for your solution to be graded, your solution must not change the name, method signature, class name, package name, or specification of the following:

- `minesweeper.server.MinesweeperServer.main()`

To get started, pull out the problem set code from [SVN Admin](#).

### Overview

---

You will start with some minimal server code and implement a server and thread-safe data structure for playing a multiplayer variant of the classic computer game "Minesweeper"

You can review the traditional/single-player minesweeper concept / rules here: [http://en.wikipedia.org/wiki/Minesweeper\\_\(video\\_game\)](http://en.wikipedia.org/wiki/Minesweeper_(video_game))

You can try playing traditional/single-player minesweeper here: <http://www.chezpeor.com/minesweeper/minesweeper.html> [1]

the final product will consist of a server and no client; it should be fully playable using telnet.

### Notes

---

We will refer to the board as an  $N \times N$  grid where each square has a state which can be 'flagged', 'dug', or 'untouched', and each square either has a bomb or does not have a bomb.

Our variant works very similarly to real minesweeper but with multiple players for one board. the main functional difference is that when one player blows up a bomb in single player, they just lose. when one player blows up a bomb in our version, they still lose, [i.e. server ends their connection] but the other players may continue playing. The square where the bomb was blown up is now a dug square with no bomb. (The player who lost may also reconnect to the same game via telnet.)

Note that there are some tricky cases of user-level concurrency: as a notable example, user A has just modified the game state (i.e. by digging in one or more squares) such that square  $i,j$  obviously has a bomb. Meanwhile, user B has not observed the board state since this update has taken place, so user B goes ahead and digs in square  $i,j$ . Your program should allow the user to dig in that square-- a user of Multiplayer Minesweeper must accept this kind of risk.

We are not specifically defining, or asking you to implement, any kind of "win condition" for the game.

In our version of minesweeper, the board is always square.

### Protocol and Specification

---

You must implement the following protocol for communication between the user and the server, and associated specification.

#### Grammar for messages from the user to the server:

```
User-to-Server Minesweeper Message Protocol
MESSAGE ::= (LOOK | DIG | FLAG | DEFLAG | HELP_REQ | BYE) NEWLINE
LOOK ::= "look"
DIG ::= "dig" SPACE X SPACE Y
FLAG ::= "flag" SPACE X SPACE Y
DEFLAG ::= "deflag" SPACE X SPACE Y
HELP_REQ ::= "help"
BYE ::= "bye"
```

```

NEWLINE ::= "\n"
X ::= INT
Y ::= INT
SPACE ::= " "
INT ::= [0-9]+

```

Server spec for dealing with user input:

#### LOOK message:

Returns a BOARD message, a string representation of the board's state. Does not mutate anything on the server. See SERVER->CLIENT protocol for the grammar of the BOARD message.

#### DIG message:

1. If either x or y is less than 0 or greater than board size, do nothing and return a BOARD message.
2. If square x,y's state is 'untouched', change square x,y's state to 'dug'.
3. If square x,y has a bomb, change it so it has no bomb and send a BOOM message to the user (see SERVER-USER protocol). Then, if the DEBUG flag is not set to 'true' (see Question 4), terminate the user's connection.
4. For any DIG message where a BOOM message is not returned, return a BOARD message.
5. If this operation results in a 'dug' square whose neighbors all have no bomb, perform the DIG operation on all of those squares.

#### FLAG message:

1. If x and y are both greater than or equal to 0, and less than board size, and square x,y is in the 'untouched' state, change it to be in the 'flagged' state.
2. For any FLAG message, return a BOARD message.

#### DEFLAG message:

1. If x and y are both greater than or equal to 0, and less than board size, and square x,y is in the 'flagged' state, change it to be in the 'untouched' state.
2. For any DEFLAG message, return a BOARD message.

#### HELP\_REQ message:

Returns a HELP message as defined in the SERVER-USER protocol. Does not mutate anything on the server.

#### BYE message:

Terminate the connection with this client.

To clarify, for any message which matches the grammar, other than a BYE message, we should always be returning either a BOARD message, a BOOM message, or a HELP message.

For any server input which does not match the USER->SERVER grammar, do nothing.

#### Grammar for messages from the server to the user:

```

MESSAGE ::= BOARD | BOOM | HELP | HELLO
BOARD ::= LINE+
LINE ::= (SQUARE SPACE)* SQUARE NEWLINE
SQUARE ::= "-" | "F" | COUNT | SPACE
SPACE ::= " "
NEWLINE ::= "\n"
COUNT ::= [1-8]
BOOM ::= "BOOM!"
HELP ::= [^NewLine]+
HELLO ::= "Welcome to Minesweeper. " N " people are playing including you. Type 'help' for help."
N ::= INT
INT ::= [0-9]+

```

The **BOARD** message, as the grammar indicates, consists of a series of newline-separated rows of space-separated chars, thereby giving a grid representation of the board's state with exactly one char for each square. The mapping of chars is as follows:

- "-" for squares with state "untouched".
- "F" for squares with state "flagged".
- " " for squares with state "dug" and 0 neighbors who have a bomb.



- integer COUNT in range [1-8] for squares with state "dug" and COUNT neighbors who have a bomb.

Notice that in this representation we reveal every square's state of "untouched", "flagged", or "dug", and we indirectly reveal limited information about whether some squares have bombs.

The **HELP** message should print out a message which indicates all the commands the user can send to the server (the exact design of the message is up to you.)

As the grammar indicates, the **HELLO** message includes N which is the number of users currently connected to the server. This message should be sent to the user only once, immediately after the server connects to the user.

## Problem 1: Setting up a Server to Deal with Multiple Clients

a. [15 points] We have provided you with a single-thread server which can accept connections with one client at a time, and which includes code to parse the input according to the client-server protocol above. Modify the server so it can maintain multiple client connections simultaneously. Each client connection should be maintained by its own thread. You may wish to add another class to do this. You may continue to do nothing with the parsed user input at this time.

## Problem 2: Implementing a Data Structure for Minesweeper

a. [30 points] Specify, implement, and test the minesweeper board data structure (as a Java type, without using sockets or threads). You are encouraged to add additional classes beyond the Board class that we have provided for you.

## Problem 3: Making your Data Structure Thread Safe

a. [15 points] Make the minesweeper board thread-safe using synchronization (again, just using Java method calls, not sockets).

b. [5 points] Near the top of your Board.java source file, include a substantial comment with an argument about why your board is thread-safe.

## Problem 4: Setting up the Server to take Command Line Arguments

We will now add a few command line arguments to MinesweeperServer. Here is the protocol for the arguments:

```

ARGS ::= DEBUG SPACE (SIZE | FILE)?
DEBUG ::= "true" | "false"
SIZE ::= SIZE_FLAG SPACE X
SIZE_FLAG ::= "-s"
X ::= INT
FILE ::= FILE_FLAG SPACE PATH
FILE_FLAG ::= "-f"
PATH ::= .+
INT ::= [0-9]+
SPACE ::= " "
```

To fulfill the server's specification, we will need the server to have an instance of our Board data structure.

For the **DEBUG** argument, the server should set a boolean DEBUG flag with the corresponding value (this will be used in Question 5.)

For a **SIZE** argument: if  $X > 0$ , the server's Board instance should be randomly generated and should have size equal to  $X$  by  $X$ . To randomly generate your board, you should assign each square to have a bomb with probability .25; else no bomb. All squares' states should be set to 'untouched'.

For a **FILE** argument: If a file exists at the given PATH, read the the corresponding file, and if it is properly formatted, deterministically create the Board instance. The file format for input should be:

```

FILE ::= LINE+
LINE ::= (VAL SPACE)* VAL NEWLINE
VAL ::= 0 | 1
SPACE ::= " "
NEWLINE ::= "\n"
```

In a properly formatted file matching the FILE grammar, if there are N LINES, each line must contain N VALs. If the file read



is properly formatted, the Board should be instantiated such that square  $i,j$  has a bomb iff the  $j$ 'th VAL in LINE  $i$  of the input is 1.

In if neither a SIZE or FILE argument is present, the Board should be randomly generated as in the case of a SIZE argument, but with a size of 10 by 10.

For example, if you were running your server from the command line and the executable was called 'server', some command line arguments might look like:

```
./server true
./server false -s 30
./server true -f ../testBoard
```

**a. [10 points]** Modify your server so that it parses these command line arguments according to the grammar, and fulfills our specification for the arguments.

## Problem 5: Putting it all Together

---

**a. [20 points]** Modify your server so that it implements our protocols and specifications, by keeping a shared reference to a single instance of Board. Note that when we send a BOOM message to the user, we should terminate their connection iff the DEBUG flag (set as instructed in problem 4) is set to 'false'.

**b. [5 points]** Near the top of your MinesweeperServer.java source file, include a substantial comment with an argument about why your server is thread-safe.

[1] You may notice that this implementation does something subtle: it ensures that there's never a bomb where you make your first click of the game. You should not implement this for the assignment. (It would be in conflict with giving the option to pass in pre-designed board, for example.)

Doing  
6.005 ~~PS6~~ PS6

11/9

Minesweeper

Except multiple users can play

So when one user making a mod lock minesweeper board  
diff clients talk to diff threads

Step 2

Step 1

No win condition - just play

Step 1

So ~~we~~ <sup>must</sup> have multiple threads

Can have another class

- so a ~~thread~~ <sup>class</sup> which spans threads

So <sup>main</sup> spans a thread which waits for a connection

When main sees that it is taken - spawn a new thread?

We've never done this type of work

- but people doing online prob have

②

Thread Pools - always a fixed # of threads  
- ~~die~~ when one dies one created

- so can limit # of requests  
Seems like a great idea in real life - but not  
for these specs

So when server gets connection

↳ Java spawns new thread

↳ do I care about thread pointer

↳ it does not seem like

- so just spawn them

Yeah never did server before

↳ did port?

No Social Server is example

Can create thread to handle (socket)

I thought I copied that

↳ No I copied Secure Server  
not Social Server



(2)

11/9

Ok lets do this

Copy Social Server

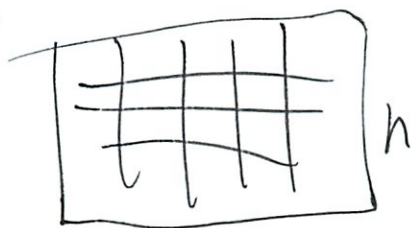
Now need to parse

Oh they did

---

Now Data Structure

Bowd



Can have bomb  
#  
be shown

Cell objects

(first real object creation thing)

- Where we have physical objects

display store data in list

- mutable

= cols

(4)

Can you have sub, sub classes?

↳ try not to

(I actually really like building object from grand p!)

Need to make everything synced

- even constructor?

Getters/setters needed

all the complications of setting up game

↳ Do we need to do <sup>ccc</sup>ccc

What does making inside class private do?

- Only that

Need get methods

- to print I guess

- Over~~an~~ride

Woot prints right on let try!

Set flag

- all on let try! tests too!

5

Ok that might be enough w/ board now

Start thinking about threads

Lock whole board whenever anything happens

Simple Buffer uses new lock object

Sync on that

Or could do on all board methods

So what was it to prevent rep exposure

1. private

2. and pointers 'inside'

And thread safety

lock whole board at board level

On the board obj - this

(could have separate ~~getall()~~ , Release() lock

- don't want to

- not in Java

No - don't just want one 'item'!

Want board to control it



6

Or users have to call start ... end  
unless the boards public methods were all the ones we  
need to call

2 ie only call 1 per term

Wizard looks on Castle

~~know~~ so whole thing looks

And then I guess each method big

---

Ok now try test

We'll take commands

Need to call right things in board

Oh Board output controlled

— F cant space  
↑ ↑ ↑ ↑  
untouched flag > 0 0 neighbors  
neighbors

How create a new board?  
What size?

Oh new messages later

Command line arguments not from MServer  
— when you start

7

Can only have 1 server instance

---

Need to do nearby logic  
later

---

What do they call x?  
Column?

---

So doing the complex logic now

Need to fix some grammar stuff to match  
also need to do arguments  
then size or file

↳ get importing

Need to make .25 bomb

Also need to update nearby counts

File import

- read file in

- so need new board input

↳ 2nd main

⑧

On never logged at Piazza

Row x column y

so 2, 4

2 ↓  $\xrightarrow{4}$  x

And 2 recursive dig calls

Now need to do read in

1 if bomb

actually not that hard

Done - all ~~initiated~~

---

Now need to do can't update

Then need to do debug messages

Where do they say can't update

- Just assumed?

Also recursive dig

[ Also fix grammars



9

This is prob hardest part

So click and it removes all nearby bombs

|   |       |   |
|---|-------|---|
| 1 | 2     | 2 |
|   | click | 1 |
| 2 | 2     | 1 |

If bombs nearby - just show you

Counts appear static

So need

setCounts() - runs at initialization

checkNeighbors()

~~clear~~ digNeighbors()

---

But how to write set counts?

For each bomb increment neighbors?

Oh

Don't care about overshooting - it fixes

✓ done

(10)

Now check neighbors

That was actually not that hard

Now need to test

Do debug mode 1st

---

Oh testing

need to test my new methods

but had to test w/o specific board

need file

would be nice if they provided one

Null pointer error...

Why is it not reading lines straight?

Oh it's not in -return Fine

Oh opps -stupid mistake!

---

Stack overflow

Oh dig recursive - can't do recursive part!

11) What should be where bomb is

- change so no bomb

Ok cant just pts 9 on bomb

- oh when remove bomb must reset counts

- but fix the increment

We'll say 1 on bomb cant itself

Oh need blank column

Need to change get to Object (not Linked List) get()

---

Now why printing wrong?

Oh never activated?

Oh ha - was actually right

---

Now why no line breaks?

New line never called

Oh error w/ this board  
did not need

---

Now removing wrong area

Oh sure cant error



(12)

Oh need if land on bomb recant  
✓ Oh wrote tests for simple cases

---

Oh is it clearing right?

It should clear recursively of available spots

Test larger to see

But what am doing now if no immediate neighbors have  
bomb show out to that ✓ correct

---

I am not clearing boards!

I am just doing nearby - instead of clear

- my mistake in reading

So now goes on only long

- since my fake runs

- seems like big error

Always row 10

- which does not really exist

Same ones over + over again  
(Pain to debug!)

(13)

Oh values were wrong defined

Ahh works much better now!

Very nice - better than I can think on lglang

---

Oh how can do thread count?

I guess 'increment/decrement value - synced!

---

Now need to try w/ command line + telnet

↳ Ah Windows firewall error!

Oh both PCs trying to open telnet created freeze  
for 3+ min so far

Look at tmo on Athena :-)

(14)

Ok on atkney

cons out of memory starting threads

✓ Fixed !!!

- copied to online

- don't know what changed

Now playing!

Jig does not print!

Fixed

When cleared - clear flag

- I guess

- don't know what win does

Multi connect works!

Flagging over #!

Fixed bye

---

After bomb are we clearing correctly?

(no working)  
~~Done~~



6.005 L14  
GVI

1/9

- ☐ view tree
- ☒ listener pattern
- ☐ model-view-controller pattern
- ☒ background threads

PS6 due Thur

PS7 out Sat, due next Thur

↳ last one

returns Nov 30

---

New topic today: GVI

- but concurrency is a problem
- under a bit more control
- but still hard to follow

How most GVIs are structured: View Tree

- all langs
- tree of objects
- bottom: buttons, labels, images, etc
  - names start w/ J

JButton

JTextField

JLabel

JList

(2)

- large libraries of abstract objects
- all grouped together into JPanel
- which is inside JFrame
- like a window in ~~other~~ <sup>the</sup> Windows

- tree corresponds to special hierarchy
- Views have bounding boxes that are nested following the tree
- Every view implements a common interface



↳ JComponent

everything so far implements JComponent's  
(all variants of ADT JComponent)

- two type of views

- Primitive: can't have children
  - just a label, button, etc

- Composite: can contain other views

We saw this before w/ recursive data types  
called composite pattern

- can build trees using them

③

If have Primitive Expressions and Composite Expressions  
have problem

---

3 places view tree is used

- output to draw UI
  - redraws anytime something changes
  - calls paint()
    - ↳ method in JComponent ADT
  - each ~~thin~~ Composite delegates
  - 2 options ADT - just uses OS
    - Swing - actually redraws it itself
    - can choose Copy Mac

- layout

- each object has bounding box
- layout algorithms run over tree
  - each item has constraints
  - bounding boxes recomputed
- called "auto layout"
  - each toolkit does this different
  - Java allows multiple Layout Managers
  - most common GroupLayout



4

- others

Border layout: fixed size on all 4 edges

- center expands to fill

- good for simple apps

Others either too complex or not powerful enough

Group layout: best

produces bounding boxes that look normal

- input

- View tree breaks up window into regions

- these regions filter input events

- mouse: has x, y coords

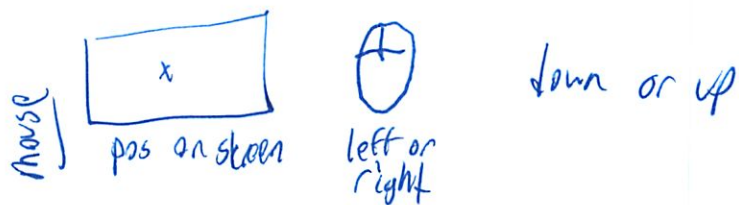
- keyboard

- So events sent to view that is used

- attach Listeners to ~~event~~ objects

- have event source (mouse events)

- state machine → state of input device



(5)

Listeners subscribe to hear events

- register interest

- add <sup>source</sup> foo Listener(listener);

? object that is ready to receive those events

interface Listener {

void eventHappened (Event event);

3

^ pass into  
what type of  
event it was

- which key or mouse button

multiple listeners can subscribe and unsubscribe

decouples source from direct dependence on listeners

- listeners are attaching themselves

---

Constructor of View does a lot of work

- build objects
- put them in tree
- <sup>build layout</sup> attach listeners

Building layout is complicated to define

Attaching listener is kinda functional programming  
higher order

## ⑥ Treating Objects as Functions

People don't just click on buttons

- but can tab over to and hit space bar

- so ~~event~~ <sup>use</sup> Action Performed - triggered  $\Rightarrow$  w/ all our methods

Thru defriend/friend concurrency example

---

VI displays and edits underlying Model object

- data that is actually being displayed

MVC school of thought ~~seem~~ requires separation

Can still write model unit tests

Or swap out VIs - HTML instead of Swing

Or put client/server split

When handling long event

Swing calls event handler

But only single thread by default

So VI is blocked

Appears non-responsive

- Doesn't even show that button has been let go



⑦

Event handlers must complete quickly  
Or put them in background

↳ New thread that actually does that

And that calls `refreshInUIThread()`

But have race condition problem

↳ Whole UI is giant mutable data object

In many GUI toolkits only main thread  
can update it

So pass message back to UI

Blocking Queue for input events built in

So then it will be Thread Safe

---

Mon: Map Reduce

## L14: Graphical User Interfaces

### Today

- View tree
- Listener pattern
- Model-view-controller pattern
- Background threads

### Required reading (from the Java Tutorial)

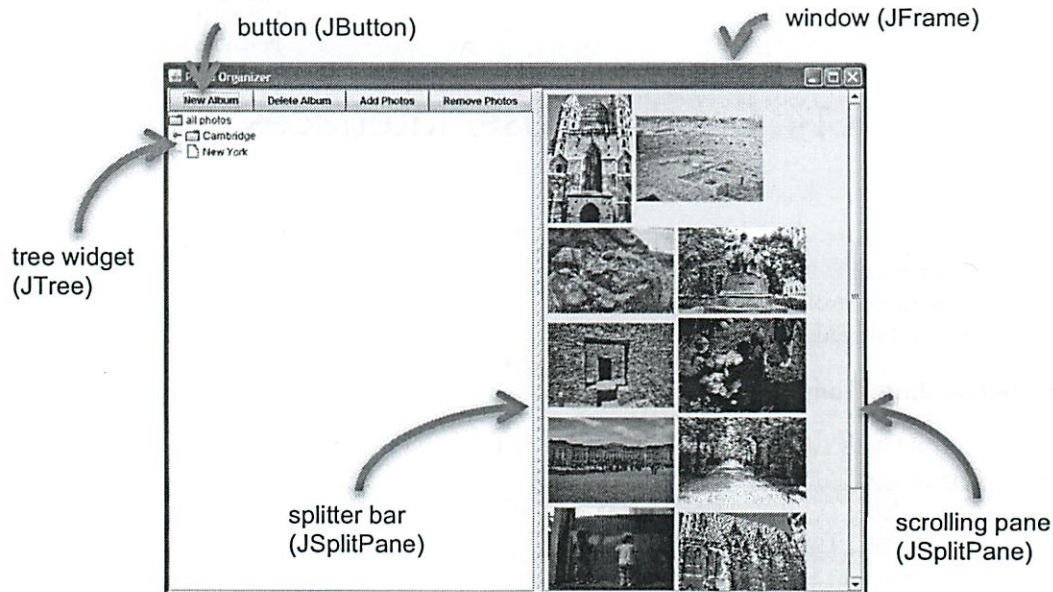
- [A Visual Guide to Swing Components](#)
- [Using Swing Components](#), specifically:
  - [Using Top-Level Containers](#)
  - [Using Text Components](#)
  - [How to Use Buttons](#)
  - [How to Use Lists](#)
  - [How to Make Frames](#)
- [Laying Out Components Within a Container](#), specifically:
  - [How to Use GroupLayout](#)
- [Writing Event Listeners](#), specifically:
  - [Introduction to Event Listeners](#)
- [Concurrency in Swing](#) (up to The Event Dispatch Thread)

Today we'll take a high-level look at the software architecture of GUI software, focusing on the **design patterns** that have proven most useful. Three of the most important patterns are:

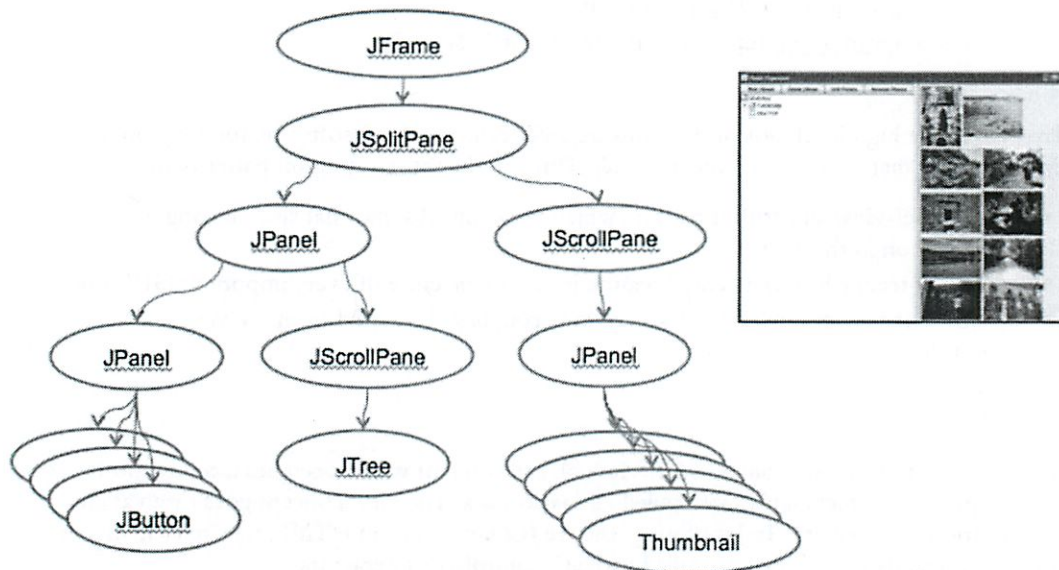
- the **model-view-controller** pattern, which has evolved somewhat since its original formulation in the early 80's;
- the **view tree**, which is a central feature in the architecture of every important GUI toolkit;
- the **listener** pattern, which is essential to decoupling the model from the view and controller.

### View Tree

Graphical user interfaces are composed of **view** objects, each of which occupies a certain portion of the screen, generally a rectangular area called its *bounding box*. The view concept goes by a variety of names in various UI toolkits. In Java Swing, they're JComponents; in HTML, they're elements or nodes; in other toolkits, they may be called widgets, controls, or interactors.



This leads to the first important pattern we'll talk about today: the **view tree**. Views are arranged into a hierarchy of containment, in which some views contain other views. Typical containers are windows, panels, and toolbars. The view tree is not just an arbitrary hierarchy, but is in fact a spatial one: child views are nested inside their parent's bounding box.



View hierarchy is an example of the Composite pattern

- **Primitive views** don't contain other views
  - button, tree widget, textbox, thumbnail, etc.
- **Composite views** are used for grouping or modifying other views
  - JSplitPane displays two views side-by-side with an adjustable splitter
  - JScrollPane displays only part of a view, with adjustable scrollbars



### Key idea

- primitives and composites implement a common interface (JComponent)
- containers can hold any JComponent: both primitives and other containers
- Composite pattern gives rise to a tree, with primitive views at the leaves and containers at the internal nodes

## How the View Tree is Used

Virtually every GUI system has some kind of view tree. The view tree is a powerful structuring idea, which is loaded with responsibilities in a typical GUI:

**Output.** Views are responsible for displaying themselves, and the view hierarchy directs the display process. GUIs change their output by mutating the view tree. For example, e.g., to show a new set of photos, the current Thumbnails are removed from the tree and a new set of Thumbnails is added in their place. A redraw algorithm built into the GUI toolkit automatically redraws the affected parts of the subtree. In Java Swing, this is done with the Interpreter pattern: every view in the tree has a `paint()` method that knows how to draw itself on the screen. The repaint process is driven by calling `paint()` on the root of the tree.

**Input.** Views can have input handlers, and the view tree controls how mouse and keyboard input is processed. More on this in a moment.

**Layout.** The view tree controls how the views are laid out on the screen, i.e. how their bounding boxes are assigned. An automatic layout algorithm automatically calculates positions and sizes of views. Specialized composites (like `JSplitPane`, `JScrollPane`) do layout themselves. More generic composites (`JPanel`, `JFrame`) delegate layout decisions to a **layout manager** (e.g. `GroupLayout`, `BorderLayout`, `BoxLayout`, ...)

## Input Handling

Input is handled somewhat differently in GUIs than we've been handling it in parsers and servers. In those systems, we've seen a single parser that peels apart the input and decides how to direct it to different modules of the program:

```
while (true) {
 read mouse click
 if (clicked on New Album) doNewAlbum();
 else if (clicked on Delete Album) doDeleteAlbum();
 else if (clicked on Add Photos) doAddPhotos();
 ...
 else if (clicked on an album in the tree) doSelectAlbum();
 else if (clicked on +/- button in the tree) doToggleTreeExpansion();

 else if (clicked on a thumbnail) doToggleThumbnailSelection();
 ...
}
```

In a GUI, we don't directly write this kind of method, because it's not modular – it mixes up responsibilities for button panel, album tree, and thumbnails all in one place. Instead, GUIs exploit the spatial separation provided by the view tree to provide functional separation as well. Mouse clicks and keyboard events are distributed around the view tree, depending on *where* they occur.

GUI input event handling is an instance of the Listener pattern (also known as Publish-Subscribe). In the Listener pattern:

- an event source generates a stream of discrete events, which correspond to state transitions in the source.
- one or more listeners register interest (subscribe) to the stream of events, providing a function to be called when a new event occurs.

In this case, the mouse is the event source, and the events are changes in the state of the mouse: its x,y position or the state of its buttons (whether they are pressed or released). Events often include additional information about the transition (such as the x,y position of mouse), which might be bundled into an **event object** or passed as parameters.

When an event occurs, the event source distributes it to all subscribed listeners, by calling their callback methods.

#### Control flow through a graphical user interface

- A top-level **event loop** reads input from mouse and keyboard
- For each input event, it finds the right view in the hierarchy (by looking at the x,y position of the mouse) and sends the event to that view's listeners
- Listener does its thing (e.g. modifying the view hierarchy) and returns immediately to the event loop

#### Higher-level GUI input events

- JButton sends an action event when it is pressed (whether by the mouse or by the keyboard)
- JTree sends a selection event when the selected element changes (whether by mouse or by keyboard)
- JTextbox sends change events when the text inside it changes for any reason

## Separating Frontend from Backend

We've seen how GUI programs are structured around a view tree, and how input events are handled by attaching listeners to views. This is the start of a *separation of concerns* – output handled by views, and input handled by listeners.

But we're still missing the application itself – the backend that represents the data and logic that the user interface is showing and editing. (Why do we want to separate this from the user interface?)

The **model-view-controller** pattern has this separation of concerns as its primary goal. It separates the user interface frontend from the application backend, by putting backend code into the model and frontend code into the view and controller. MVC also separates input from output; the controller is supposed to handle input, and the view is supposed to handle output.

The model is responsible for maintaining application-specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants. OK, all mutable objects do that. But a model must also notify its clients



when there are changes to its data, so that dependent views can update their displays, and dependent controllers can respond appropriately. Models do this notification using the **listener pattern**, in which interested views and controllers register themselves as listeners for change events generated by the model.

View objects are responsible for output. A view usually occupies some chunk of the screen, usually a rectangular area. Basically, the view queries the model for data and draws the data on the screen. It listens for changes from the model so that it can update the screen to reflect those changes.

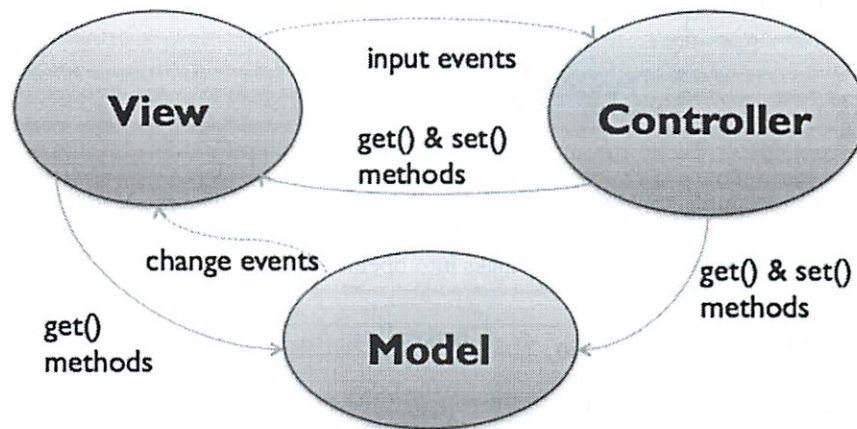
Finally, the controller handles the input. It receives keyboard and mouse events, and instructs the model to change accordingly.

**View handles output**

- gets data from the model to display it
- listens for model changes and updates display

**Controller handles input**

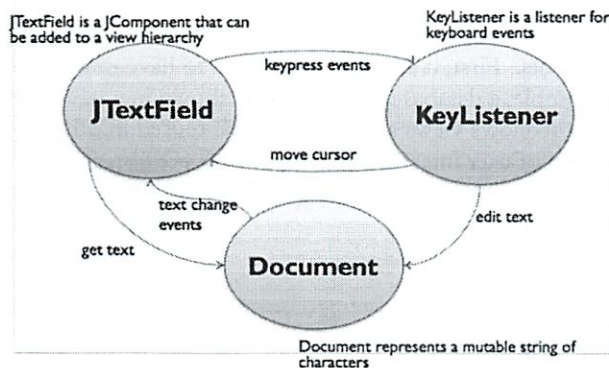
- listens for input events on the view tree
- calls mutators on model or view



**Model maintains application state**

- implements state-changing behavior
- sends change events to views

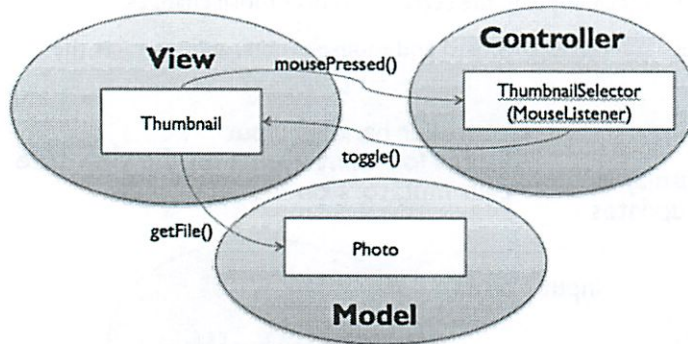
A simple example of the MVC pattern is a text field widget (this is Java Swing's text widget). Its model is a mutable string of characters. The view is an object that draws the text on the screen (usually with a rectangle around it to indicate that it's an editable text field). The controller is an object that receives keystrokes typed by the user and inserts them in the string.



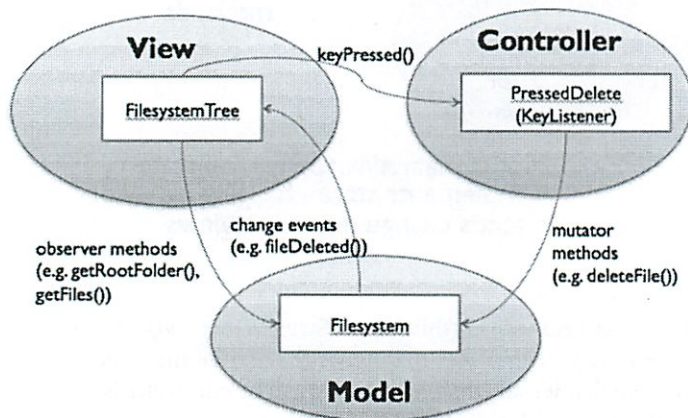


Instances of the MVC pattern appear at many scales in GUI software. At a higher level, this text field might be part of a view (like the address book editor), with a different controller listening to it (for text-changed events), for a different model (like the address book). But when you drill down to a lower level, the text field itself is an instance of MVC.

Here's an example from a photo browsing application:



And here's a larger example, in which the view is a filesystem browser (like the Mac Finder or Windows Explorer), the model is the disk filesystem, and the controller is an input handler that translates the user's keystrokes and mouse clicks into operations on the model and view.



The separation of model and view has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and models to be reused in other applications. The MVC pattern enables the creation of user interface **toolkits**, which are libraries of reusable views. Java Swing is such a toolkit. You can easily reuse view classes from this library (like JButton and JTree) while plugging your own models into them.

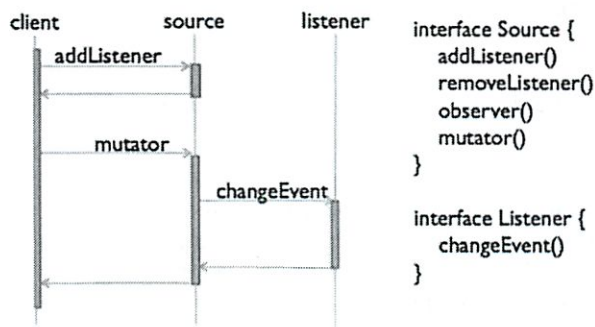
## Risks of Event-Based Programming

Control flow through an event-based program is not simple. You can't follow the control just by studying the source code, because control flow depends on listener relationships established at

runtime, and input events happening nondeterministically. Careful discipline about who listens to what (like the model-view-controller pattern) is essential for limiting the complexity of control flow and understanding how to debug your program.

The hidden control flow leads to some unexpected pitfalls, which is the next thing we'll look at in this lecture.

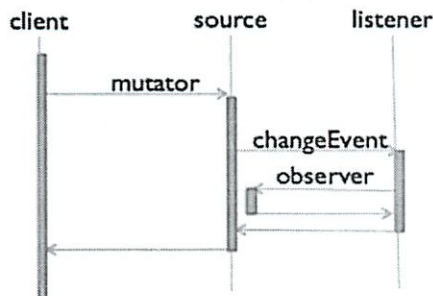
First, a bit of notation. The diagram below is a **sequence diagram**, which is useful for depicting control flow. Time flows downward. Vertical time lines represent objects, such as an event source or a listener. Horizontal arrows show method calls and returns passing control between objects. Finally, dark rectangles show when a method is active (i.e., on the call stack).



Here's the conventional interaction that occurs in the listener pattern. A client uses **addListener** (or a similar method) registers a listener to receive notifications from the event source. Then, when the source changes state (usually due to some other object calling a **mutator** method), it fires an event to all its registered listeners by calling **changeEvent** on them.

## Pitfall #1: Listener Calls Observer Methods

This leads to the first pitfall. The listener often reacts to the change in the model by pulling more data from the source using **observer** method calls. For example, when a textbox gets a change event from its model, it needs to call **getText()** to get the new text and display it. So calls to **observer()** may occur while **mutator()** is still in progress.



Why is this a potential problem? Because the **mutator()** method hasn't returned yet, it's possible that the source data structure is not yet in a consistent state (i.e. not yet satisfying its rep invariant), which might cause the **observer()** method to return garbage (or worse, throw an exception).

When the source calls **changeEvent()** on its listeners, it is giving up control – in much the same way that a method gives up control when it returns to its caller, or that a thread gives up control when its timeslice is over and another thread takes control of the processor. In fact, what we're seeing here is a concurrency problem!

Here's some pseudocode that demonstrates this pitfall:

```
class Filesystem {
 private Map<File, List<File>> cache;
 public List<File> getContents(File folder) {
 ... check for folder in cache, else read it from disk and update
 cache
 }
 public void deleteContents(File folder) {
 for (File f: getContents(folder)) {
 f.delete();
 fireChangeEvent(f, REMOVED); // notify listeners that f was
 deleted
 }
 cache.remove(folder); // cache is no longer valid for this folder
 }
}
```

Filesystem is a model class that represents a disk filesystem. It has an observer method `getContents()`, which returns the files in a folder, and a mutator method `deleteContents()`, which deletes all the files in a folder. To minimize disk traffic, it also maintains a cache mapping folder names to their files, so `getContents()` doesn't always have to hit the disk.

Now suppose a filesystem browser view is showing the files from a folder (which it obtained with `getContents()`) and then `deleteContents()` is invoked. One by one the files are removed from the folder on disk, and change events are sent back to the view – but if the view is calling `getContents()` to update itself, **it will always see the stale copy in the cache**. The cache isn't invalidated until the end of `deleteContents()`, which is too late for the view.

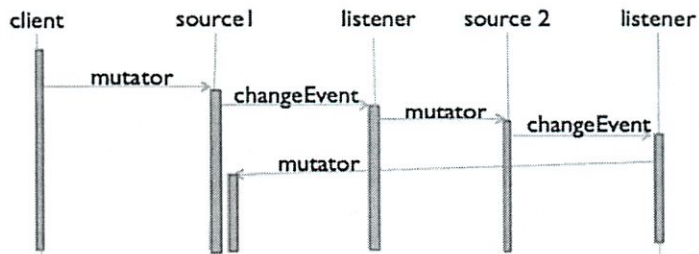
The essential problem is that this class has a rep invariant (that the cache correctly reflects the state of the filesystem) that doesn't hold when the view calls `getContents()`. It's quite normal, even inevitable, for invariants to be *temporarily* unsatisfied while a mutator method is executing. In this case, event-passing has created an opportunity for a client to get control during that period of inconsistency, and see buggy results.

So an event source has to make sure that it's consistent --- i.e., that it has established all of its internal invariants – before it starts issuing notifications to listeners. It's often best to delay firing off events until the end of the method that caused the modification. Don't fire events while you're in the midst of making changes to the model's data structure. This example could be fixed either by batching up events until the end of `deleteContents()`, or by invalidating the cache *before* starting to remove files, so that the invariant is never unsatisfied.

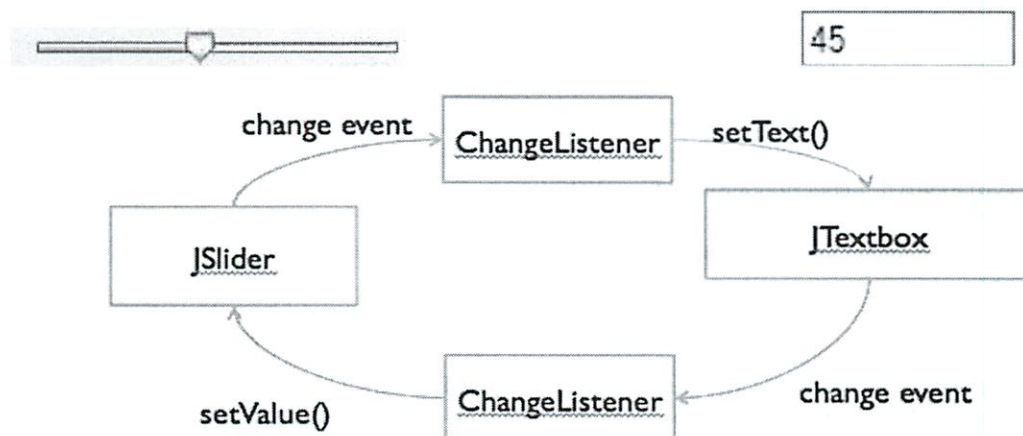


## Pitfall #2: Listener Calls Mutator Methods

Another pitfall occurs when a listener responds to an update message by calling the mutator on the model. Why would it do that? It might, for instance, be trying to keep the model within some legal range. Or two models could be listening to each other in order to keep their state synchronized. So recursive calls to `mutator()` may occur while `mutator()` is still in progress. Obviously, this could lead to infinite regress if you're not careful.



Here's a concrete example of this pitfall: a numeric slider and a textbox, which you want to synchronize with each other so that the user can change the value using either widget:

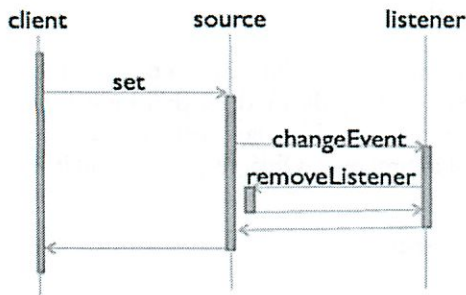


So both widgets end up listening to each other's changes, and an infinite regress might result.

A good practice for models to protect themselves against this regress is to only send update events if a change actually occurs; if a client calls `mutator()` but it has no actual effect on the model, then no events should be sent.

## Pitfall #3: Listener Removes Itself

Another potential pitfall is a listener that unregisters itself with `removeListener`. For example, suppose we have a model of stock market data, and a listener that's watching for a certain stock to reach a certain price. Once the stock hits the target price, the listener does its thing (e.g., popping up a window to notify the user, or executing a trade); but then it's no longer needed, so it unregisters itself from the model.



This is a problem if the model is iterating naively over its collection of listeners, and the collection is allowed to change in the midst of the iteration. Here's some concrete code showing the problem – the listeners are stored as a simple fixed-size array. Trace through and see what happens if the *i*th listener calls `removeListener()` on itself from its `changeEvent()` method:

```

class Source {
 private Listener[] listeners;
 private int size;
 public void removeListener(Listener l) {
 for (int i = 0; i < size; ++i) {
 if (listeners[i] == l) {
 listeners[i] = listeners[size-1]; --size;
 }
 }
 }
 private void fireChangeEvent(...) {
 for (int i = 0; i < size; ++i) {
 listeners[i].changeEvent(...);
 }
 }
}

```

Most Java collections (Lists, Sets) have the same problem. If you're lucky, Java will throw a `ConcurrentModificationException` when you mutate a collection that you're currently iterating. If you're not, your list will simply be quietly corrupted.

It's safer to iterate over a *copy* of the listener list. Since one-shot listeners are not particularly common, however, this imposes an extra cost on every event broadcast. So the ideal solution is to copy the listener list only when necessary – i.e., when a register or unregister occurs in the midst of event dispatch. `javax.swing.EventListenerList` works this way.

## Background Processing in Graphical User Interfaces

The last major topic for today connects back to concurrency.

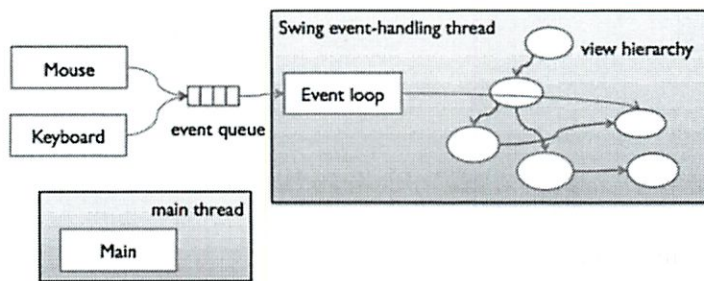
First, some motivation. Why do we need to do background processing in graphical user interfaces? Even though computer systems are steadily getting faster, we're also asking them to do more. Many programs need to do operations that may take some time: retrieving URLs over the network, running database queries, scanning a filesystem, doing complex calculations, etc.

But graphical user interfaces are event-driven programs, which means (generally speaking) **everything is triggered by an input event handler**. For example, in a web browser, clicking a hyperlink starts loading a new web page. But if the click handler is written so that it actually retrieves the web page itself, then the web browser will be very painful to use. Why? Because its interface will

appear to **freeze up** until the click handler finishes retrieving the web page and returns to the event loop. Here's why.

This happens because input handling and screen repainting is all handled from a single thread. That thread (called the **event handling thread**) has a loop that reads an input event from the queue and dispatches it to listeners on the view hierarchy. When there are no input events left to process, it repaints the screen. But if an input handler you've written delays returning to this loop – because it's blocking on a network read, or because it's searching for the solution to a big Sudoku puzzle – then input events stop being handled, and the screen stops updating. So long tasks need to run in the background.

In Java, the event-handling thread is distinct from the main thread of the program (see below). It is started automatically when a user interface object is created. As a result, every Java GUI program is **automatically multithreaded**. Many programmers don't notice, because the main thread typically doesn't do much in a GUI program – it starts creation of the view, and then the main thread just exits, leaving only the event-handling thread to do the main work of the program.



The fact that Swing programs are multithreaded by default creates risks. There's very often a shared mutable datatype in your GUI: the **model**. If you use background threads to modify the model without blocking the event-handling thread, then you have to make sure your data structure is **threadsafe**.

But another important shared mutable datatype in your GUI is the view tree. **Java Swing's view tree is not threadsafe**. In general, you cannot safely call methods on a Swing object from anywhere but the event-handling thread.

#### The view tree is a big meatball of shared state

- And there's no lock protecting it at all
- It's confined (by specification) to the event-handling thread, so it's ok to access view objects from the event-handling thread (i.e., in response to input events)
- But the Swing specification forbids touching – reading *or* writing – any `JComponent` objects from a different thread
  - See "Threads and Swing",  
<http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html>
  - The truth is that Swing's implementation does have *one big lock* (`Component.getTreeLock()`) but only some Swing methods use it (e.g. `layout`)

#### Solution: the event queue is also a message-passing queue

- To access or update Swing objects from a different thread, you can put a message (represented as a `Runnable` object) on the event queue

```
SwingUtilities.invokeLater(new Runnable() {
```

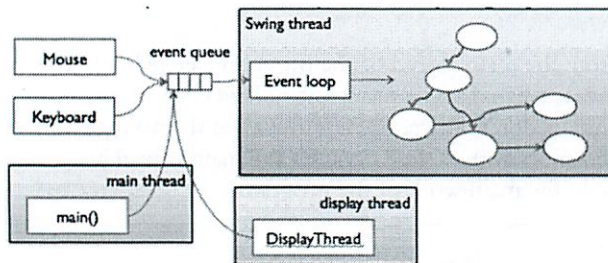


```

public void run() {
 content.add(thumbnail);
 ...
}
});

```

- The event loop handles one of these pseudo-events by calling run()



## Summary

### View hierarchy

- Organizes the screen into a tree of nested rectangles
- Used for dispatching input as well as displaying output
- Uses the Composite pattern: compound views (windows, panels) can be treated just like primitive views (buttons, labels)

### Publish-subscribe pattern

- An event source sends a stream of events to registered listeners
- Decouples the source from the identity of the listeners
- Beware of pitfalls

### MVC pattern

- Separation of responsibilities: model=data, view=output, controller=input
- Decouples view from model

### Background threads

- Swing view tree is confined to the event-handling thread
- To read and write the tree from another thread, use the event loop as a message-passing queue

Le 005  
Recitation

11/10

(skipped)

## 6.005 Elements of Software Construction | Fall 2011

### Problem Set 7: Jotto Client GUI

#### Due: Thursday, November 17 2011, 11:59 PM

The purpose of this problem set is to give you experience with GUI programming. In particular you will be using Java Swing. Finally, you will get some more practice with multithreading. *grr*

**You have substantial design freedom on this problem set.** However, in order for your solution to be graded, your solution must include the following objects in your GUI as specified in the problems. They are defined for you in the JottoGUI class (and instantiated with default constructors), but you can instantiate them however you would like. These objects must be added directly to the layout of the JottoGUI JFrame.

- a JButton named "newPuzzleButton"
- a JLabel named "puzzleNumber"
- a JTextField named "guess"
- a JTextField named "newPuzzleNumber"
- a JTable named "guessTable"

*NOTE: You can rename the variables themselves, but their "name" property must be set appropriately using the setName() method.*

To get started, pull out the problem set code from SVN Admin.

### Overview

*Who or what is a Jotto?*

In this problem set, you will implement a simple Jotto playing client that communicates with a server that we have provided for you. If you've never played Jotto, you can find a short description of it here.

In our particular version of the game, the client selects a puzzle ID at random which corresponds to a secret 5-letter word on the server. The user can then submit 5-letter dictionary word guesses to the server, which will respond with the number of letters in common between the two words and the number of letters in the correct position.

You will communicate with the server through scripts.mit.edu. All communication from the client to the server will be through a request of the following form:

*their server*  
**http://6.005.scripts.mit.edu/jotto.py?puzzle=[puzzle #]&guess=[5-letter dictionary word]**

where [puzzle #] will be replaced by an integer generated by the client, and [5-letter dictionary word] is a (you guessed it) 5-letter dictionary word.

The response from the server should be:

**guess [in common] [correct position]**

where [in common] is the number of letters that the secret word, which the server determines by the puzzle #, and the guess have in common. [correct position] is the number of letters in the guess that are in the correct position in the secret word. Thus, [in common] will always be greater than or equal to [correct position].

If the request sent to the server was invalid, the response will be one of the following errors (in **bold**). The unbold text is a description of the error.

**error 0** Ill-formatted request.

**error 1** Non-number puzzle ID.



**error 2** Invalid guess. Length of guess != 5 or guess is not a dictionary word.

For example, if you were to submit

**http://6.005.scripts.mit.edu/jotto.py?puzzle=16952&guess=crazy**

The secret word is "cargo," so the server should respond

**guess 3 1**

But if you submitted

**http://6.005.scripts.mit.edu/jotto.py?puzzle=16952&guess=elephant**

The server would respond with

**error 2**

This problem set requires the use of multithreading. To help in debugging and testing multithreading behavior, the server will provide delayed responses for any guess containing an asterisk ("\*). Ensure that your GUI is still responsive during this delay.

For example, if you were to submit

*so GUI must stay responsive - 2 threads*

**http://6.005.scripts.mit.edu/jotto.py?puzzle=16952&guess=\*bean**

The server will provide a delayed response, waiting for 5 seconds before responding back with:

**guess 1 0**

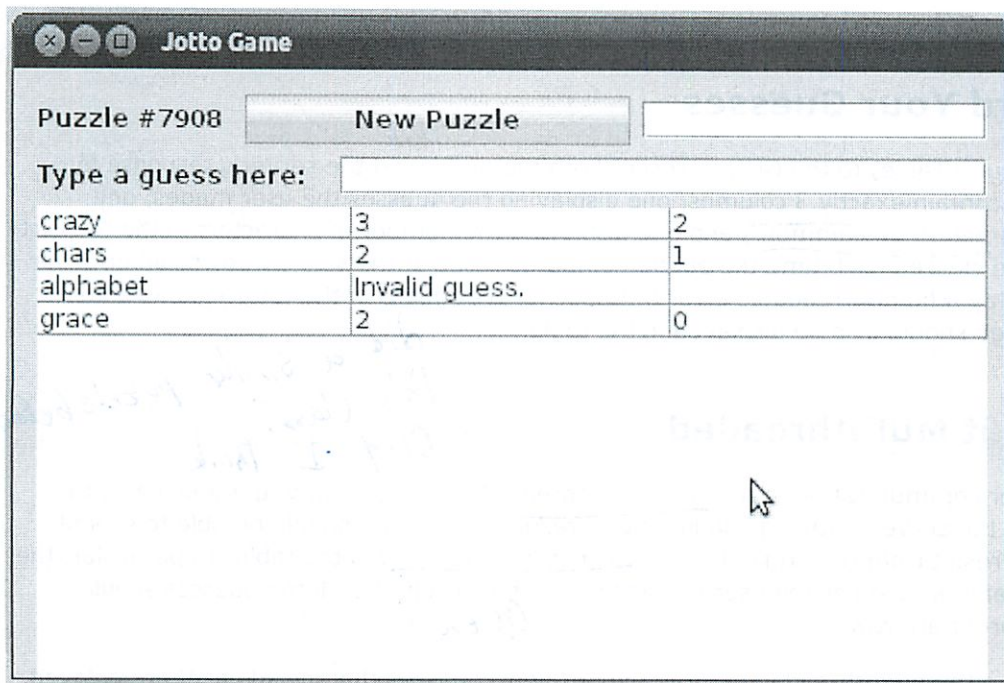
The '\*' will be considered a character in the server, so any guesses of incorrect length including the '\*' returns an error:

**http://6.005.scripts.mit.edu/jotto.py?puzzle=16952&guess=\*bea**

**error 2**

By the end of this problem set you will have built a GUI for your client to interact seamlessly with the server. The interface should eventually look something like this:





Your GUI does not have to look exactly like the window above. It should pass our tests as long as it functions, and the necessary components are there.

## Problem 1: Communicate with the Server

**[20 points]** Write a Java method that uses `java.net.URL` (See the Java Tutorial for more details on how to use `java.net.URL`) to send a guess to the server, read back the reply, and return it. You must handle exceptions and errors from the server appropriately. We've provided a method signature for you in JottoModel called `makeGuess`, but you are free to change anything about it that you like. In fact, its return type is currently void, so you must change that. Make sure to include a spec and test your method thoroughly.

## Problem 2: Set the Puzzle Number

- a. [10 points]** Create a GUI that includes a "New Puzzle" button (`newPuzzleButton`) next to a text field (`newPuzzleNumber`) and a label (`puzzleNumber`) to display the puzzle number. We recommend that you use the GroupLayout layout manager. It should basically look like the top row of the window above.
- b. [10 points]** Next, register an `ActionListener` to set the input from the text field as the new puzzle number, displaying it in the label next to the button. If no number is provided or the input is not a positive integer, pick a random positive integer. Make sure you can generate at least 10,000 different numbers. We've provided a JottoGUI class for you to use, but as stated earlier, you are free to change whatever about this that you like as long as you have the necessary components named correctly. Your program should start with a puzzle number already selected, be it random or always the same.

## Problem 3: Make a Guess

**[20 points]** Add a `JTextField` (`guess`) to the GUI for the user to input a guess. Then use an `ActionListener` to send the guess to the server when the user presses ENTER, and print the result to standard output. Also clear the textbox after every guess, so that it's easy for the user to type the next guess. You should print "You win! The secret word was [secret word]!" when the user guesses the word correctly, ie. the server responds with "guess 5 5." Feel free to be creative with the message, but make sure it contains the phrase "you win."

hdg



## Problem 4: Record Your Guesses

[20 points] Add a `JTable` (`guessTable`) to the GUI, to record both the guess and the server's response for that guess. The table should contain exactly 3 columns, one displaying the guesses the user made, one displaying the number of characters "in common", and one displaying the number of characters in the "correct position". The table should be cleared each time the puzzle number is reset. If the server returns an error for a guess, the GUI should display a human-readable error message for that guess in the table instead of its score. The "You win!" message should also be displayed in the table.

## Problem 5: Make it Multithreaded

[20 points] Move the server communication to a background thread. Make sure that you are still able to interact with the GUI even if the server is taking a long time to respond. The user should be able to submit multiple guesses even if the result from the original query has not yet appeared in the table. In particular, the guess should appear in the table as soon as the user submits it, and the results for all the guesses should eventually appear in the appropriate rows.

Remember, to test and debug the multithreaded nature of your Client, guesses that include a '\*' are delayed. You should make sure that your GUI remains responsive during this delay.

did a similar puzzle before  
this class?  
6.004 I think

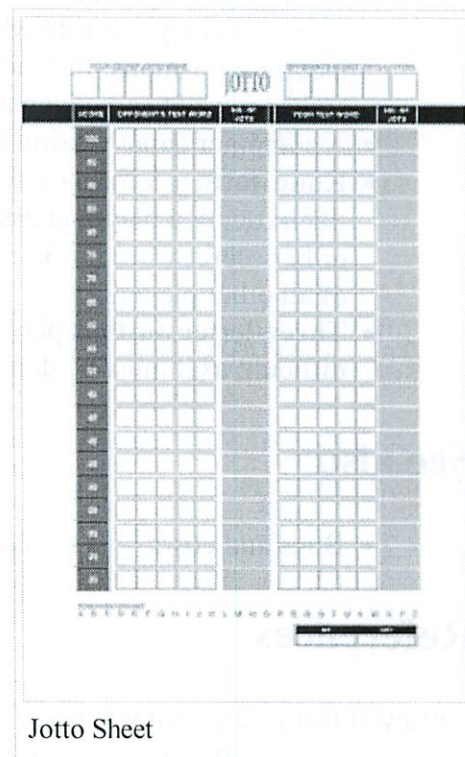
Queue?



# Jotto

From Wikipedia, the free encyclopedia

**Jotto** (sometimes **Giotto Džiotto**) is a logic-oriented word game played with two players, a writing implement, and a piece of paper. Each player picks a secret word of five letters (that is in the dictionary, generally no proper nouns are allowed, and generally consisting of all different letters), and the object of the game is to correctly guess the other player's word first. Players take turns guessing and giving the number of Jots, or the number of letters that are in both the guessed word and the secret word. The positions of the letters don't matter: for example, if the secret word is OTHER and a player guesses PEACH, he gets a reply of 2 (for the E and the H, even though they're in the wrong positions). Using a written-out alphabet, players cross out letters that are eliminated with Jot counts of 0 and other logical deductions.



## Contents

- 1 History
- 2 Variations
- 3 See also
- 4 References

## History

Jotto was invented in 1955 by Morton M. Rosenfeld and marketed by his New York-based Jotto Corp. In the 1970s, copyright passed to the Selchow and Righter Company. It is now made by Endless Games.

## Variations

- The most common variation uses words of four letters instead of five.
- Players should agree before playing if secret words can contain duplicate letters.
- Six-Letter** (sometimes called 'Word Master Mind', though its logical content places it well beyond Master Mind and Jotto). Known to have been played with pencil and paper in UK computer departments at least as far back as 1970. Each player picks a secret word of 6 letters, and they take turns guessing the other's word. Secret words (sometimes called 'targets') and guesses, must all be real words verifiable in a nominated dictionary. When you guess a word, you find out the # of letters which are perfect matches only. So if you guess PEACHY and the secret word was OTHERS, you get a reply of 0 (because the E and H are in the wrong positions). Even more so than Jotto, this game stretches one's skills in combinatorial logic as well as one's command of the dictionary. The name of the same computer game is Sixicon by Island Software, 1979.
- Five-Letter**, like Jotto, requires players to take turns guessing at an opponent's five-letter word. Like Six-Letter, responses only indicate the number of perfect matches. If you guess PEACH and

the secret word is PHIAL, the response would be 1 -- the P is an exact match, but the A and the H are not. A strong strategy for Five-Letter involves attacking the middle three positions (2, 3, and 4) with vowels.

- The television game show Lingo consists largely of contestants playing a variant of Jotto. In Lingo, the player is told which matching letters are in the correct position, and which are in incorrect positions. Instead of being selected by the contestants, the words are chosen by the show and contestants are given a limited number of guesses.
- A computer game version called Josho created by Zachary Stern is available.
- **Kane Jotto** is a form of multi-player Jotto. Each player has a secret word, like the original game. Each round, a player guesses a word and each player goes around stating the number of letters his or her secret word has in common with the guess. The winner correctly guesses the secret words of all other players.
- **Three-letter Jotto** is played without using pencil and paper. In order that the game not be overly difficult, as it must be done entirely in the head, only perfect matches return a count.

## See also

- Bulls and cows — a similar game with numbers

← yeah 6.004

## References

Retrieved from "http://en.wikipedia.org/w/index.php?title=Jotto&oldid=442871384"

Categories: Word games | Selchow and Righter games | Endless Games games | Paper-and-pencil games

- This page was last modified on 3 August 2011 at 17:10.
  - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
- Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.



```

 }
 });
}

/** Sets the model shown by this view. */
public void setWizard(Wizard w) {
 this.model = w;
 refresh();
}

/** Refresh this view to reflect changes in the model. */
public void refresh() {
 name.setText(model.getName());
 picture.setIcon(new ImageIcon(model.getPhoto()));
 friendsModel.clear();
 for (Wizard friend : model.getFriends()) {
 friendsModel.addElement(friend);
 }
}

/** Switch the model to display the friend currently selected in the friends list.
 * If no friend selected, has no effect. */
private void visitSelectedFriend() {
 if (friends.isSelectionEmpty()) return; // no wizard selected, can't switch to it
 int selectedIndex = friends.getMinSelectionIndex();
 setWizard((Wizard) friendsModel.get(selectedIndex));
}

/** Repeatedly friends and unfriends the currently-selected friend N times.
 * If no friend selected, has no effect. */
private void thrashWithSelectedFriend() {
 if (friends.isSelectionEmpty()) return; // no wizard selected, can't switch to it
 int selectedIndex = friends.getMinSelectionIndex();
 Wizard selectedWizard = (Wizard) friendsModel.get(selectedIndex);
 thrashForeground(model, selectedWizard);
}

private static final int NUM_THRASHES = 5;

private void thrashForeground(Wizard source, Wizard target) {
 try {
 for (int j = 0; j < NUM_THRASHES; ++j) {
 // unfriend them
 source.defriend(target);
 refresh();
 Thread.sleep(1000); // wait for a second

 // then refriend them
 source.friend(target);
 refresh();
 Thread.sleep(1000); // wait for a second
 }
 }
}

```



```
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 }
 }

 private void thrashBackground(final Wizard source, final Wizard target) {
 Thread backgroundThread = new Thread(new Runnable() {
 public void run() {
 try {
 for (int j = 0; j < NUM_THRASHES; ++j) {
 // unfriend them
 source.defriend(target);
 refreshInUIThread();
 Thread.sleep(1000); // wait for a second

 // then refriend them
 source.friend(target);
 refreshInUIThread();
 Thread.sleep(1000); // wait for a second
 }
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 }
 }
 });

 private void refreshInUIThread() {
 SwingUtilities.invokeLater(new Runnable() {
 public void run() {
 refresh();
 }
 });
 }

 backgroundThread.start();
 }
}
```

```

//setPreferredSize(new Dimension(400,400));

// get some margins around components by default
layout.setAutoCreateContainerGaps(true);
layout.setAutoCreateGaps(true);

// place the components in the layout (which also adds them
// as children of this view)
layout.setHorizontalGroup(
 layout.createParallelGroup()
 .addGroup(layout.createSequentialGroup()
 .addComponent(picture, 100, 100, 100)
 .addComponent(name))
 .addGroup(layout.createParallelGroup()
 .addComponent(friendsLabel)
 .addComponent(friends, 0, GroupLayout.PREFERRED_SIZE, Integer.
MAX_VALUE)
 .addComponent(thrashButton))
);
layout.setVerticalGroup(
 layout.createSequentialGroup()
 .addGroup(layout.createParallelGroup()
 .addComponent(picture, 100, 100, 100)
 .addComponent(name, GroupLayout.PREFERRED_SIZE, GroupLayout.
PREFERRED_SIZE, GroupLayout.PREFERRED_SIZE))
 .addComponent(friendsLabel)
 .addComponent(friends, 0, 200, Integer.MAX_VALUE)
 .addComponent(thrashButton)
);

// add listeners for user input
friends.addMouseListener(new MouseAdapter() {
 public void mouseClicked(MouseEvent event) {
 // respond only on double-click
 if (event.getClickCount() == 2) {
 visitSelectedFriend();
 }
 }
});

friends.addKeyListener(new KeyAdapter() {
 public void keyPressed(KeyEvent event) {
 if (event.getKeyCode() == KeyEvent.VK_ENTER) {
 visitSelectedFriend();
 }
 }
});

thrashButton.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent event) {
 thrashWithSelectedFriend();
 }
});

```

everything  
needs  
both  
horiz +  
vertical

```
package beforeclass.hogwarts.gui;

import beforeclass.hogwarts.model.Wizard;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.DefaultListModel;
import javax.swing.GroupLayout;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

@SuppressWarnings("serial")
public class WizardView extends JPanel {

 // this view's model
 private Wizard model = null;
 // may be null

 // view objects used to display this view
 private final JLabel picture;
 private final JTextField name;
 private final JList friends;
 private final DefaultListModel friendsModel;
 // rep invariant: all view objects != null

 /** Make a WizardView. */
 public WizardView() {
 // create the components
 picture = new JLabel("[picture]");
 name = new JTextField("Harry Potter");
 friendsModel = new DefaultListModel();
 friends = new JList(friendsModel);

 // create labels and other decorations that we don't need to access later,
 // so we don't save them in the rep
 final JLabel friendsLabel = new JLabel("Friends:");
 final JButton thrashButton = new JButton("Thrash");

 // define layout
 GroupLayout layout = new GroupLayout(this);
 setLayout(layout);
 }
}
```

✓ view stuff here

uses group layout



```
package beforeclass.hogwarts.gui;
```

```
import beforeclass.hogwarts.model.Castle;
import beforeclass.hogwarts.model.Castle.SameNameException;
import beforeclass.hogwarts.model.Wizard;
```

```
import java.awt.BorderLayout;
import java.net.MalformedURLException;
import java.net.URL;
```

```
import javax.swing.JFrame;
```

```
@SuppressWarnings("serial")
```

```
public class MainWindow extends JFrame {
```

```
 private final WizardView wizardView;
```

```
 public MainWindow() {
```

```
 setTitle("Hogwarts");
```

```
 setLayout(new BorderLayout());
```

```
 wizardView = new WizardView();
```

```
 add(wizardView, BorderLayout.CENTER);
```

```
 pack();
```

```
 }
```

```
 public void setStartingWizard(Wizard w) {
```

```
 wizardView.setWizard(w);
```

```
 }
```

```
 private static Castle makeHogwarts() {
```

```
 try {
```

```
 Castle hogwarts = new Castle();
```

```
 URL noPicture = MainWindow.class.getResource("images/no-photo.jpg");
```

```
 hogwarts.add(new Wizard(hogwarts, "Harry Potter", MainWindow.class.getResource(
"images/harry.jpg")));
```

```
 hogwarts.add(new Wizard(hogwarts, "Hermione Granger", MainWindow.class.
getResource("images/hermione.jpg")));
```

```
 hogwarts.add(new Wizard(hogwarts, "Ron Weasley", noPicture));
```

```
 hogwarts.add(new Wizard(hogwarts, "Albus Dumbledore", noPicture));
```

```
 hogwarts.add(new Wizard(hogwarts, "Severus Snape", noPicture));
```

```
 hogwarts.lookup("Harry Potter").friend(hogwarts.lookup("Hermione Granger"));
```

```
 hogwarts.lookup("Harry Potter").friend(hogwarts.lookup("Ron Weasley"));
```

```
 hogwarts.lookup("Hermione Granger").friend(hogwarts.lookup("Ron Weasley"));
```

```
 hogwarts.lookup("Harry Potter").friend(hogwarts.lookup("Albus Dumbledore"));
```

```
 hogwarts.lookup("Severus Snape").friend(hogwarts.lookup("Albus Dumbledore"));
```

```
 return hogwarts;
```

```
 } catch (SameNameException e) {
```

```
 e.printStackTrace();
```

```
 throw new AssertionError("shouldn't happen");
```

```
 }
```

```
 }
```

p-set calls Jotto GUI

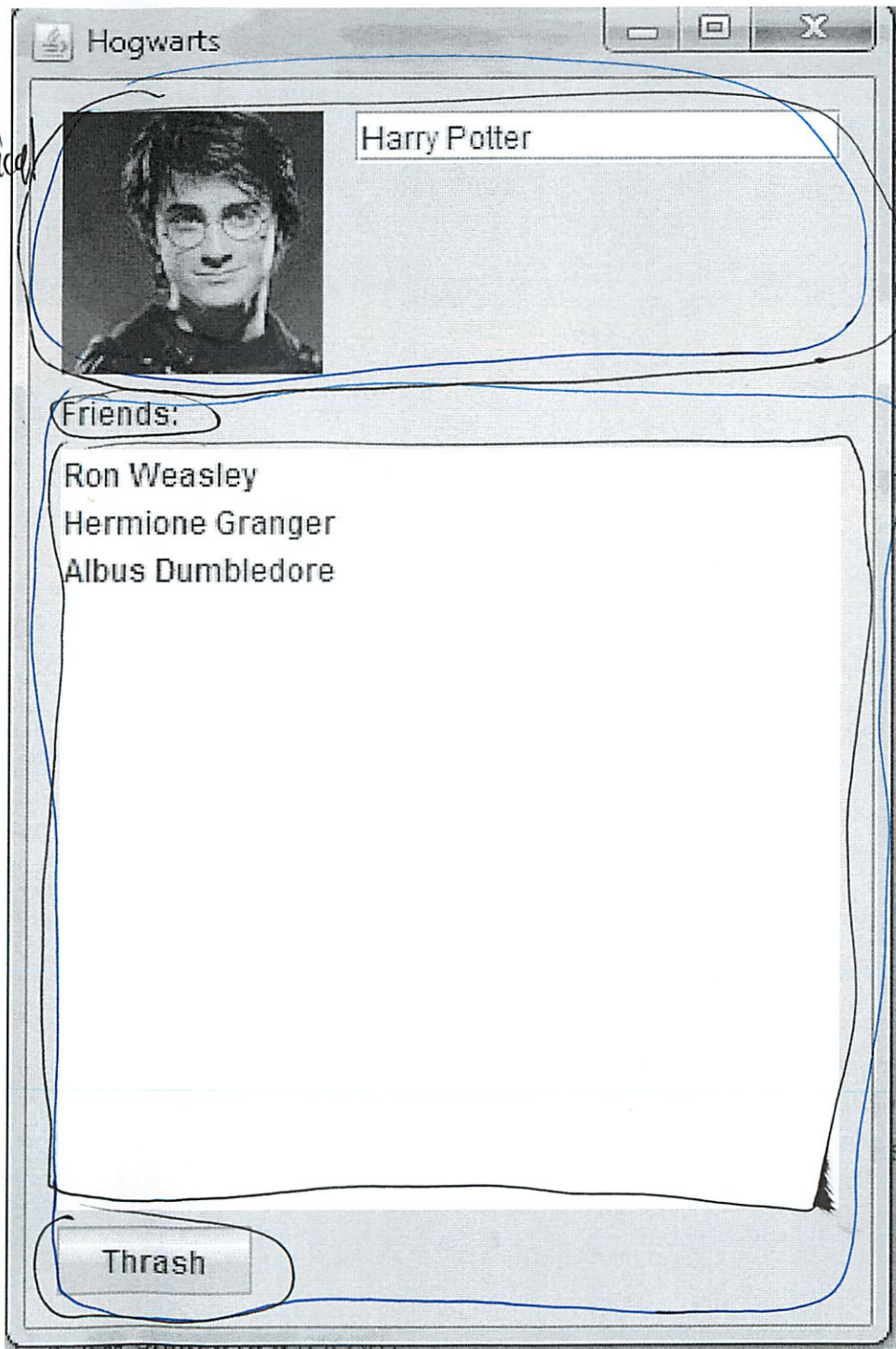
← creator method

← set layout for window!

or

call model/data stuff

```
public static void main(String[] args) {
 Castle hogwarts = makeHogwarts();
 MainWindow win = new MainWindow();
 win.setStartingWizard(hogwarts.lookup("Harry Potter"));
 win.setVisible(true);
}
```





[sun.com](http://java.sun.com)<http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html>

Nov 13, 2011

## Article

# Threads and Swing

### Threads and Swing

*This article about multithreading in Swing was archived in April 1998. A month later, we published another article, ["Using a Swing Worker Thread"](#), expanding on the subject. For a better understanding of how multithreading works in Swing, we recommend that you read both articles.*

**Note:** In September 2000 we changed this article and its example to use an updated version of `SwingWorker` that fixes a subtle threading bug.

By Hans Muller and Kathy Walrath

The Swing API was designed to be powerful, flexible, and easy to use. In particular, we wanted to make it easy for programmers to build new Swing components, whether from scratch or by extending components that we provide.



For this reason, we do not require Swing components to support access from multiple threads. Instead, we make it easy to send requests to a component so that the requests run on a single thread.

This article gives you information about threads and Swing components. The purpose is not only to help you use the Swing API in a thread-safe way, but also to explain why we took the thread approach we did.

Here are the sections of this article:

- The single-thread rule: Swing components can be accessed by only one thread at a time. Generally, this thread is the event-dispatching thread.
- Exceptions to the rule: A few operations are guaranteed to be thread-safe.
- Event dispatching: If you need access to the UI from outside event-handling or drawing code, then you can use the SwingUtilities.invokeLater() or invokeAndWait() method.
- Creating threads: If you need to create a thread -- for example, to handle a job that's computationally expensive or I/O bound -- you can use a thread utility class such as SwingWorker or Timer.
- Why did we implement Swing this way?: This article ends with some background information about Swing's thread safety.

## The single-thread rule

Here's the rule:

***Once a Swing component has been realized, all code that might affect or depend on the state of that component should be executed in the event-dispatching thread.***

This rule might sound scary, but for many simple programs, you don't have to worry about threads. Before we go into detail about how to write Swing code, let's define two terms: realized and event-dispatching thread.

Realized means that the component's `paint()` method has been or might be called. A Swing component that's a top-level window is realized by having one of these methods invoked on it: `setVisible(true)`, `show()`, or (this might surprise you) `pack()`. Once a window is realized, all components that it contains are realized. Another way to realize a component is to add it to a container that's already realized. You'll see examples of realizing components later.

The event-dispatching thread is the thread that executes drawing and event-handling code. For example, the `paint()` and `actionPerformed()` methods are automatically executed in the event-dispatching thread. Another way to execute code in the event-dispatching thread is to use the `SwingUtilities.invokeLater()` method.

## Exceptions to the rule

There are a few exceptions to the rule that all code that might affect a realized Swing component must run in the event-dispatching thread:

- **A few methods are thread-safe:** In the Swing API documentation, thread-safe methods are marked with this text:

*This method is thread safe, although most Swing methods are not.*

*which?*

- **An application's GUI can often be constructed and shown in the main thread:** The following typical code is safe, as long as no components (Swing or otherwise) have been realized:

```
public class MyApplication {
 public static void main(String[] args) {
 JFrame f = new JFrame("Labels");
 // Add components to
 // the frame here...
 f.pack();
 f.show();
 // Don't do any more GUI work here...
 }
}
```

*↑ that's what I have done*

All the code shown above runs on the "main" thread. The `f.pack()` call realizes the components under the `JFrame`. This means that, technically, the `f.show()` call is unsafe and should be executed in the event-dispatching thread. However, as long as the program doesn't already have a visible GUI, it's exceedingly unlikely that the `JFrame` or its contents will receive a `paint()` call before `f.show()` returns. Because there's no GUI code after the `f.show()` call, all GUI work moves from the main thread to the event-dispatching thread, and the preceding code is, in practice, thread-safe.

- **An applet's GUI can be constructed and shown in the `init()` method:** Existing browsers don't draw an applet until after its `init()` and `start()` methods have been called. Thus, constructing the GUI in the applet's `init()` method is safe, as long as you never call `show()` or `setVisible(true)` on the actual applet object.

By the way, applets that use Swing components must be implemented as subclasses of `JApplet`, and components should be added to the `JApplet` content pane, rather than directly to the `JApplet`. As for any applet, you should never perform time-consuming initialization in the `init()` or `start()` method; instead, you should start a thread that performs the time-consuming task.

- **The following `JComponent` methods are safe to call from any thread:** `repaint()`, `revalidate()`, and `invalidate()`. The `repaint()` and `revalidate()` methods queue requests for the event-dispatching thread to call `paint()` and `validate()`, respectively. The `invalidate()` method just marks a component and all of its direct ancestors as requiring validation.
- **Listener lists can be modified from any thread:** It's always safe to call the `addChangeListener()` and `removeChangeListener()` methods. The add/remove operations have no effect on an event dispatch that's under way.

**NOTE:** The important difference between `revalidate()` and the older `validate()` method is that `revalidate()` queues requests that might be coalesced into a single `validate()` call. This is similar to the way that `repaint()` queues paint requests that might be coalesced.



NOTE



## Event dispatching

Most post-initialization GUI work naturally occurs in the event-dispatching thread. Once the GUI is visible, most programs are driven by events such as button actions or mouse clicks, which are always handled in the event-dispatching thread.

However, some programs need to perform non-event-driven GUI work after the GUI is visible. Here are some examples:

- **Programs that must perform a lengthy initialization operation before they can be used:** This kind of program should generally show some GUI while the initialization is occurring, and then update or change the GUI. The initialization should *not* occur in the event-dispatching thread; otherwise, repainting and event dispatch would stop. However, after initialization the GUI update/change *should* occur in the event-dispatching thread, for thread-safety reasons.
- **Programs whose GUI must be updated as the result of non-AWT events:** For example, suppose a server program can get requests from other programs that might be running on different machines. These requests can come at any time, and they result in one of the server's methods being invoked in some possibly unknown thread. How can that method update the GUI? By executing the GUI update code in the event-dispatching thread.

The `SwingUtilities` class provides two methods to help you run code in the event-dispatching thread:

- `invokeLater()`: Requests that some code be executed in the event-dispatching thread. This method returns immediately, without waiting for the code to execute.
- `invokeAndWait()`: Acts like `invokeLater()`, except that this method waits for the code to execute. As a rule, you should use `invokeLater()` instead of this method.

This page gives you some examples of using this API. Also see the [BINGO example](#) in *The Java Tutorial*, especially the following classes: [CardWindow](#), [ControlPane](#), [Player](#), and [OverallStatusPane](#).

### Using the `invokeLater()` Method

You can call `invokeLater()` from any thread to request the event-dispatching thread to run certain code. You must put this code in the `run()` method of a `Runnable` object and specify the `Runnable` object as the argument to

`invokeLater()`. The `invokeLater` method returns immediately, without waiting for the event-dispatching thread to execute the code. Here's an example of using `invokeLater()`:

```
Runnable doWorkRunnable = new Runnable() {
 public void run() { doWork(); }
};
SwingUtilities.invokeLater(doWorkRunnable);
```

*is static?*

*isn't how to do this inside another thread*

*— make sure shared reference...*

### Using the `invokeAndWait()` Method

The `invokeAndWait()` method is just like `invokeLater()`, except that `invokeAndWait()` doesn't return until the event-dispatching thread has executed the specified code. Whenever possible, you should use `invokeLater()` instead of `invokeAndWait()`. If you use `invokeAndWait()`, make sure that the thread that calls `invokeAndWait()` does not hold any locks that other threads might need while the call is occurring.

Here's an example of using `invokeAndWait()`:

```
void showHelloThereDialog()
```



```
throws Exception {
Runnable showModalDialog = new
Runnable() {
 public void run() {
 JOptionPane.showMessageDialog(
 myMainFrame, "Hello There");
 }
};
SwingUtilities.invokeLaterAndWait
(showModalDialog);
}
```

Similarly, a thread that needs access to GUI state, such as the contents of a pair of text fields, might have the following code:

```
void printTextField() throws Exception {
 final String[] myStrings =
 new String[2];

 Runnable getTextFieldText =
 new Runnable() {
 public void run() {
 myStrings[0] =
 textField0.getText();
 myStrings[1] =
 textField1.getText();
 }
 };
 SwingUtilities.invokeLaterAndWait
 (getTextFieldText);

 System.out.println(myStrings[0]
 + " " + myStrings[1]);
}
```

## Creating threads

If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug. In general, they just aren't necessary for strictly GUI work, such as updating component properties.

However, sometimes threads *are* necessary. Here are some typical situations where threads are used:

- To perform a time-consuming task without locking up the event-dispatching thread. Examples include making extensive calculations, doing something that results in many classes being loaded (initialization, for example), and blocking for network or disk I/O.
- To perform an operation repeatedly, usually with some predetermined period of time between operations.
- To wait for messages from clients.

You can use two classes to help you implement threads:

- **SwingWorker**: Creates a background thread to execute time-consuming operations.
- **Timer**: Creates a thread that executes some code one or more times, with a user-specified delay between executions.

## Using the SwingWorker Class



The `SwingWorker` class is implemented in `SwingWorker.java`, which is *not* included in any of our releases, so you must download it separately.

**DOWNLOAD** `SwingWorker` does all the dirty work of implementing a background thread. Although many programs don't need background threads, background threads are sometimes useful for performing time-consuming operations, which can improve the perceived performance of a program.

To use the `SwingWorker` class, you first create a subclass of it. In the subclass, you must implement the `construct()` method so that it contains the code to perform your lengthy operation. When you instantiate your `SwingWorker` subclass, the `SwingWorker` creates a thread but does not start it. You invoke `start()` on your `SwingWorker` object to start the thread, which then calls your `construct()` method. When you need the object returned by the `construct()` method, you call the `SwingWorker`'s `get()` method. Here's an example of using `SwingWorker`:

```
...//in the main method:
final SwingWorker worker =
 new SwingWorker() {
 public Object construct() {
 return new
 expensiveDialogComponent();
 }
 };
worker.start();

...//in an action event handler:
JOptionPane.showMessageDialog
(f, worker.get());
```

When the program's `main()` method invokes the `start()` method, the `SwingWorker` starts a new thread that instantiates `ExpensiveDialogComponent`. The `main()` method also constructs a GUI that consists of a window with a button.



When the user clicks the button, the program blocks, if necessary, until the `ExpensiveDialogComponent` has been created. The program then shows a modal dialog containing the `ExpensiveDialogComponent`. You can find the entire program in `MyApplication.java`.

**DOWNLOAD**

## Using the Timer Class

The `Timer` class works with an `ActionListener` to perform an operation one or more times. When you create a timer, you specify how often the timer should perform the operation, and you specify which object is the listener for the timer's action events. Once you start the timer, the action listener's `actionPerformed()` method is invoked one or more times to perform its operation.



**NOTE**

The `actionPerformed()` method defined in the `Timer`'s action listener is invoked in the event-dispatching thread. That means that you never have to use the `invokeLater()` method in it.

Here's an example of using a `Timer` to implement an animation loop:

```
public class AnimatorApplicationTimer
 extends JFrame implements
 ActionListener {
 ...//where instance variables
```



```
...//are declared:
Timer timer;

public AnimatorApplicationTimer(...) {
 ...
 // Set up a timer that calls this
 // object's action handler.
 timer = new Timer(delay, this);
 timer.setInitialDelay(0);
 timer.setCoalesce(true);
 ...
}

public void startAnimation() {
 if (frozen) {
 // Do nothing. The user has
 // requested that we stop
 // changing the image.
 } else {
 //Start (or restart) animating!
 timer.start();
 }
}

public void stopAnimation() {
 //Stop the animating thread.
 timer.stop();
}

public void actionPerformed
(ActionEvent e) {
 //Advance the animation frame.
 frameNumber++;

 //Display it.
 repaint();
}
...
}
```

## Why did we implement Swing this way?

There are several advantages in executing all of the user interface code in a single thread:

- **Component developers do not have to have an in-depth understanding of threads programming:** Toolkits like ViewPoint and Trestle, in which all components must fully support multithreaded access, can be difficult to extend, particularly for developers who are not expert at threads programming. Many of the toolkits developed more recently, such as SubArctic and IFC, have designs similar to Swing's.
- **Events are dispatched in a predictable order:** The runnable objects enqueued by `invokeLater()` are dispatched from the same event queue as mouse and keyboard events, timer events, and paint requests. In toolkits where components support multithreaded access, component changes are interleaved with event processing at the whim of the thread scheduler. This makes comprehensive testing difficult or impossible.
- **Less overhead:** Toolkits that attempt to carefully lock critical sections can spend a substantial amount of time and space managing locks. Whenever the toolkit calls a method that might be implemented in client code (for example, any public or protected method in a public class), the toolkit must save its state and release all locks so that the client code can grab locks if necessary. When control returns from the method, the toolkit must regrab its locks and restore its state. All



applications bear the cost of this, even though most applications do not require concurrent access to the GUI.

Here's a description, written by the authors of the SubArctic Java Toolkit, of the problem of supporting multithreaded access in a toolkit:

*It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications, particularly those which have a GUI component. Use of threads can be very deceptive. In many cases they appear to greatly simplify programming by allowing design in terms of simple autonomous entities focused on a single task. In fact in some cases they do simplify design and coding. However, in almost all cases they also make debugging, testing, and maintenance vastly more difficult and sometimes impossible. Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism. For example, thorough testing (which is always difficult) becomes nearly impossible when bugs are timing dependent. This is particularly true in Java where one program can run on many different types of machines and OS platforms, and where each program must work under both preemptive or non-preemptive scheduling.*

*As a result of these inherent difficulties, we urge you to think twice about using threads in cases where they are not absolutely necessary. However, in some cases threads are necessary (or are imposed by other software packages) and so subArctic provides a thread-safe access mechanism. This section describes this mechanism and how to use it to safely manipulate the interactor tree from an independent thread.*

The thread-safe mechanism they're referring to is very similar to the `invokeLater()` and `invokeAndWait()` methods provided by the `SwingUtilities` class.

copyright © Sun Microsystems, Inc

6.005  
PS 7

11/12

Last P-Set!

Step 1      Communicate w/ Server

— Jotto Model

Make new thread for each guess  
Or thread pool for performance  $\rightarrow$  better performance

Need a game #

One for each model

Outlined model code

Need to ~~the~~ add error handling

See how stuff returns

Where return back

Actually handling does not need to be different  
— but can be

Runnable can't return type String  
(this stuff I don't really get...)

②

So how to return?

→ Save in a shared location

---

So modify shared data structure  
? the table?

Or a static table?

~~Linked~~ Linked List of guesses

? make new guess object?

---

Must make Guess return a value

- void now
- says should change
- ? but can ~~then~~ just modify underlying data store?

---

Try it out w/ unit tests!

Wait(), Notify(), NotifyAll()

✓ Wait returns error 2

So using shared data model!



(3)

May change later to notify when result returned  
update grid handler:

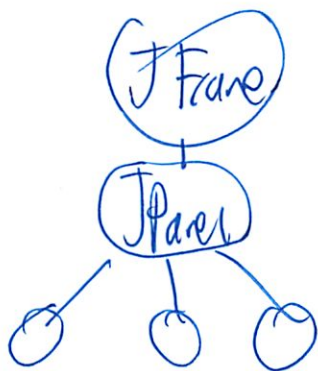
## Step 2 GUI

Start GUI w/ Newpuzzle button #  
+ event listener

Try launching what we have  
Makes empty window

Title - □ x

See example



GraphLayout needs container host

- Jot to GUI

L is a JFrame

Or make JPanel!  
- usually in other file

(4) What goes in which File?

I try to combine in 1 file?

---

Lot of UI code

Sequential Group - one after another - row

~~Parallel~~ Parallel Group - 4 possible ways - column

I see nothing! Why?

Why are we using new layout manager we have not before  
- oh no - have used

I think I need to extend JPanel - not add

Its just weird where they created it

Piazza posted

I will assume ok to move

---

Ok now error when tries to run

on pack()

Ok everything needs both vert + horiz group

⑤ Woot! Something runs - but ugly!!!

Ok replace w/ t/P code

I totally don't get this layout method!

Also need to label buttons in their constructor

✓ Much nicer!

This parallel + sequential thing is weird!

✓ Cool UI looks right

---

Now listeners

Need action

How update table?

✓ Woot new Puzzle works!

✓ Make it do it on window launch

Ok now need to let it take a number

✓ Now be able to hit enter

Should use action performed on button

✓ Done - Getting the hang of this



6

### Step 3 Make a Guess

How throw ~~error~~ error?

Just blank - ? do nothing ...

Have a separate update Guess Table

How to write  $\rightarrow$  set Value At?

Create new JTable each time?

↑ don't know if can do that

Print where on win?

Oh can use fires

So data should be from Abstract Table Model  
↳ you should build one

Ok change new puzzle thing

Still no idea how to update it ...

Did H/P have one?

⑦

Oh calls add element

Oh but that is list - this is table

(Big time sigh...)

✓ Found equivalent  
Does not error anymore

- but no update

Table never displays

~ may be more basic problem...

Ahh I needed to add cols

✓ Working now w/ Default table model

! Add scroll pane to be fancy

Trying to get header to show

Ah got it - needs to be in group

- but how remove margin?

- screw it

8

Now need to pass back message to update table

Always 5 or less

(No grammar! :))

Here throw error

Now need to pass message back

how?

registers a listener?

want it to be decoupled

wait is not right

meh just pass back

i can I do that

wait has other wait to do something

Can repaint every 100 sec?

or make it public method

i guess have position need repaint

i how to have loop checking for?

Need like an Interrupt

- piazza



9

(can I do an object change listener?)

let me try wait()

↳ But that makes it block

Oh well try rest

✓ Oh all that needs now is a sync ~~wait~~

- don't need to finish this weekend

callback ← what I want

This should not be hard!

SwingWorker?

- has a 3rd thread Event Dispatch

---

Students on Piazza recommend SwingUtilities

Read doc

So also need to pass model ref in

Actually need pointer to panel

Yay it works !!! ✓

(sometimes just need break to re-think)

11/13

Event Dispatch  
Thread

(10)

On new game must clear

Let me try w/o 'invoke Later

L works too!

queue is a bit better form

my problem was ~~for~~ bringing a pointer to  
panel into the model - wanted to avoid

✓ Game Finished

---

Review tests

✓ Slow works manually

So pass tests null panel

✓ Fixed and added some

---

Now JFrame - Panel discrepancy

learn that  
and specs todo

RCM

Questions

Thread callback

Access to stuff in thread

Encapsulation callback

~~the~~ JPanel vs JFrame

J Scroll Panel

Debugging Threads

— Visualizing what's running

---

need some call back reference

Thread safety for refresh

— add invokeLater

— event queue

— give it a runnable

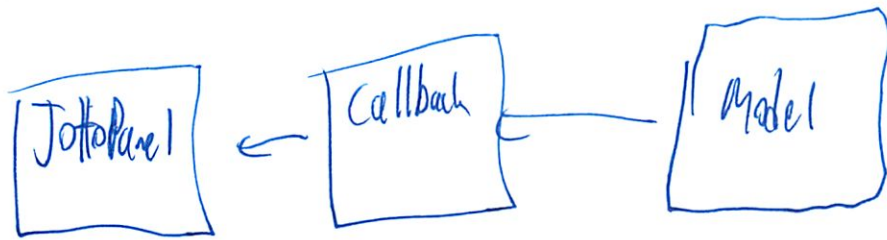
For testing → stub of GUI

— tie

— as assumption that using Swing utilities



② To make clean: interpose interface

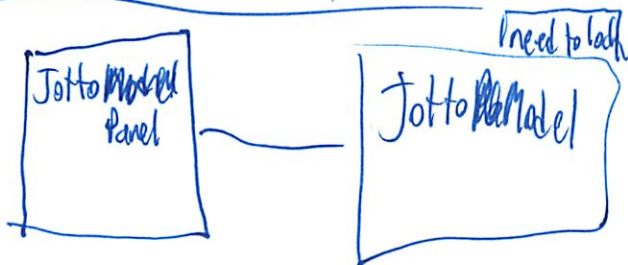


Simple way

Callback's interface `public void done(m)` ↓ could pass results

in JottoPanel - inner class that implements callback

Complex way (Hogwarts)



So does not need to callback<sup>w/</sup> data  
Just callback

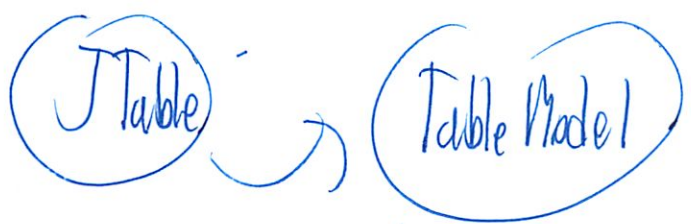
Real way: make Model

add Change Listener (change Listener)

- will have one changed method

Panel adds itself as listener to Models' ↩

3



↳ could tie model - but then tied to swing

Interface - purely methods  
- no fields

Abstract - can have default methods + fields  
- can have unimplemented / abstract methods  
- can't extend more than 1 class at a time

Panel is adding listener to ~~abstract~~ JTable Model data type  
- not directly to list  
add JTable Change listener

---

could do JPanel - would be graded same  
- w/o JPanel could be simpler  
- if don't reuse

9

## Need to add Scroll Pane

Eclipse Debugger will debug threads

- "Bug" button adds debugger
  - not JUnit
- Need to pause to look in
- Can see stack trace
- 1st line of sync block means no lock
- Pause only good for dead lock
- Or can put in Breakpoints
- Step in - go inside constructor path
  - step return to go back
- Step over - skip over ya think working
- Only within current thread



(5)

## Grading

1. Correct
2. Ready for change
3. ~~Something else~~ Easy to understand

Adherence to spec very important  
following every last line  
no file paths

But rep should not have been checked

---

Emailed me: it said rep was in assignment  
I should have read assignment closer.

(I think I am getting much better at double checking paper)

6.005 L15  
Map / Filter / Reduce

11/14

- ☐ list processing
- ☐ lambda expressions
- ☐ functional objects
- ☐ higher-order functions

PS 7 out, due Thur

Quiz 2 next Mon 11-12 PM, Walker

Proj 2 starts next week

- focus on new material
- but includes old material

Sign up for terms this week  
due last day of class

Wed: normal lecture

Thur: recitation: quiz review

No recitations or lectures after Thur

---

~~Abstract functions~~ Functional programming  
lots of verbose code  
But nice in other languages

(Unclear about vocab  
terms here  
- see notes)

Can be a handy/nice pattern  
Map/Filter/Reduce

- no control (if, for) code disappears
- for working on sequences

(2)

Java good at higher order functions

- must write types
  - Confusing - mind blowing
- 

Will do this in Python lot

- Can integrate Python into Java
- 

Want a list of all words in files

- lots of iteration over sequences
- lots of structure (if, for, etc)

filter files by suffix

---

Map/Filter/Reduce lets us abstract away from all this structure

---

Abstract datatype

- won't even have Java interface
- since it covers a wide range of datatypes

$\text{Seq}\langle E \rangle$  is a sequence of elements in type  $E$

including  
Java  $\left[ \begin{array}{ll} \text{List}\langle E \rangle & \text{Iterable}\langle E \rangle \\ E[] & \text{Iterable}\langle E \rangle.\text{iterator}() \end{array} \right.$



③

Python

- list [1,2,3] mutable
- string
- tuple  $\in$  ordered list of objects (1,2,3)  
immutable
- Streams  $\in$  sequence of strings, 1 per line
- iterable - any type can provide those methods

Map

- the body of for loop
- takes a sequence
- and a function to be applied to every sequence

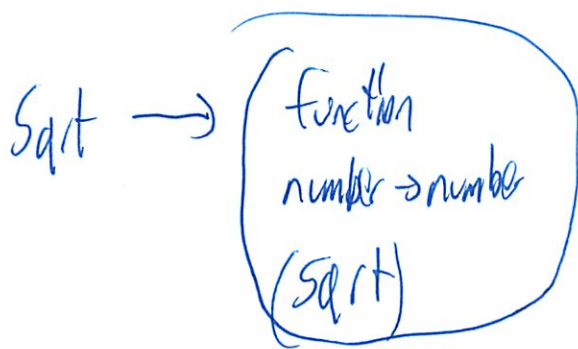
$$\text{map}: (E \rightarrow F) \times \text{Seq} \langle E \rangle \rightarrow \text{Seq} \langle F \rangle$$

$$\text{map}(f, [e_0, \dots, e_{n-1}]) = [f(e_0), \dots, f(e_{n-1})]$$

ie  $\gg \text{map}(\text{sqrt}, [1, 4, 9, 16])$   
 $[1.0, 2.0, 3.0, 4.0]$

Note no parenthesis  $\rightarrow$  refers to function element

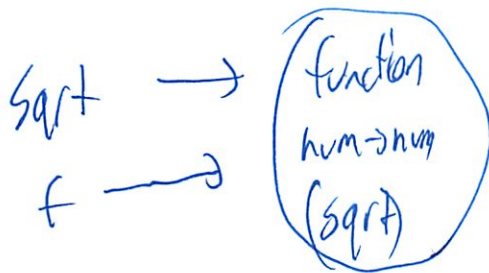
4



Referring to function by name

Can say  
 $f = \text{sqrt}$

Now



So Sqrt is object you can pass ~~around~~ around

functions are 1st class values

- $\rightarrow$  can name
- $\rightarrow$  can pass it as argument
- $\rightarrow$  can return it
- $\rightarrow$  can put it in the lists, tuples, sets, ...

5

public-ness  
generic  
static-ness  
generic-ness  
functions

are not lot  
class in Java

Jython - java interpreter of python

Can import Java functions into Jython

- makes function lot class object

could also do threads `map(Thread.join, threads)`

Lambda expression { will return [None, None, ... None]

Syntactic sugar for map  
creates new anonymous fns on the fly

`[t.join() for t in threads]`

if want `map(2k, [1, 2, 3, 4])`

could write

`def powerOfTwo(k) return 2**k`  
`map(powerOfTwo, [1, 2, 3, 4])`



6

But it won't raise power of Two again

Just say

$\text{map}(\lambda k: 2 \times k, [1, 2, 3, 4])$

$\lambda = \text{lambda}$

Invented by Church 20-30 years before computers

Similar to creating anonymous inner classes in Java

Maps let us iterate easily

Filter

$(E \rightarrow \text{bool}) \times \text{Seq } E \Rightarrow \text{Seq } E$

tests predicate  
like it satisfy  
particular condition

$\text{filter}(P, [e_0, \dots, e_{n-1}]) = [e_i \text{ s.t. } P(e_i)]$   
such that

(7)

So list comprehension

$$\left[ t.join() \text{ for } t \text{ in threads, if } t.isRunning() \right]$$
  
lambda t: t.join()

map(lambda x: x % 2 == 1, [1, 2, 3, 4, 5])

returns [1, 3, 5]

only returns odd #s

filter(str.isalpha, "s4j2\_")

Python treats string as seq of characters

and returns "sj" ← only the alpha characters

## Reduce

- most powerful of the two
- could implement the others w/ this
  - but more abstract

$$(F \times E \rightarrow F) \times \text{Seq}[E] \rightarrow F$$

8

$\text{reduce}(\text{add}, [1, 2, 3], 0)$   
default ~~args~~ this  
to start with

$$(((0 + 1) + 2) + 3) = 6$$

builds up combos of items on list

add does not need to be same type  
ie to turn list of #s into string

Op of  $\text{string} \times \text{num} \rightarrow \text{string}$

$\text{reduce}(\text{lambda } s, e: s + \text{str}(e), [1, 2, 3], "")$   
need 2 arguments  
empty string

$$((( "" + \text{str}(1) ) + \text{str}(2) ) + \text{str}(3)) = "123"$$

could also have fold right

$$(1 \oplus (2 \oplus (3 \oplus 0)))$$

Python does fold left  
→



9

Add does not matter - could do either way

But some things matter

def last term (list):

# list, Seq [Seq [E]]

return reduce (list, concat, list, [])

[ [...], [...], [...]]

Will bottom out when it has no more sub-directories

Lisp can do recursive

## L15: Map, Filter, Reduce

### Today

- Map/filter/reduce
- Lambda expressions
- Functional objects
- Higher-order functions

### Example

Suppose we're given the following problem: write a method that finds the words in the Java files in your project.

Following good practice, we break it down into several simpler steps and write a method for each one:

- find all the files in the project, by scanning recursively from the project's root folder
- restrict them to files with a particular suffix, in this case .java
- open each file and read it in line-by-line
- break each line into words

Writing the individual methods for these substeps, we'll find ourselves writing a lot of low-level iteration code. For example, here's what the recursive traversal of the project folder might look like:

```
/** Find all the files in the filesystem subtree rooted at folder.
 * @param folder root of subtree. Requires folder.isDirectory() == true.
 * @return list of all ordinary files (not folders) that have folder as
their ancestor.
 * @throws IOException if an error occurs while accessing the filesystem
 */
public static List<File> allFilesIn(File folder) throws IOException {
 List<File> files = new ArrayList<File>();
 for (File f: folder.listFiles()) {
 if (f.isDirectory()) {
 files.addAll(allFilesIn(f));
 } else if (f.isFile()) {
 files.add(f);
 }
 }
 return files;
}
```

And here's what the filtering method might look like, which restricts that file list down to just the Java files (imagine calling this like `onlyFilesWithSuffix(files, ".java")`):

```
/** Filter a list of files to those that end with suffix.
 * @param files list of files (all non-null)
 * @param suffix string to test
 * @return a new list consisting of only those files whose names end with
suffix
 */
```

```

 public static List<File> onlyFilesWithSuffix(List<File> files, String
suffix) {
 List<File> result = new ArrayList<File>();
 for (File f : files) {
 if (f.getName().endsWith(suffix)) {
 result.add(f);
 }
 }
 return result;
}

```

Today we're going to talk about map/filter/reduce, a design pattern that substantially simplifies the implementation of functions that operate over sequences of elements. In this example, we'll have lots of sequences – lists of files; input streams that are sequences of lines; lines that are sequences of words; frequency tables that are sequences of (word, count) pairs. Map/filter/reduce will enable us to operate on those sequences with *no* explicit control flow – not a single for loop or if statement.

Along the way, we'll also see an important Big Idea: functions as “first-class” data values, meaning that they can be stored in variables, passed as arguments to functions, and created dynamically like other values. This is unfortunately not well-realized in Java, as we'll see. It's possible to write first-class functions in Java, and useful at times (we've already done it!), but it's verbose, so it won't give us the extra simplicity that we want for map/filter/reduce. So to demonstrate the power of map/filter/reduce, we'll switch back to Python.

## Abstracting Out Control Flow

We've already seen two design patterns that abstract away from the details of iterating over a data structure: Iterator and Visitor.

Iterator gives you a sequence of elements from a data structure, without you having to worry about whether the data structure is a set or a token stream or a list or an array – the Iterator looks the same no matter what the data structure is. Visitor abstracts the details of traversal of a recursive data type.

The map/filter/reduce patterns we'll be looking at today do something similar to Iterator and Visitor, but at an even higher level – they treat the entire sequence of elements as a unit, so that the programmer doesn't have to name and work with the elements individually. In this paradigm, the control statements disappear: specifically, the for statements, the if statements, and the return statements in the code above will be gone. We'll also be able to get rid of most of the temporary names (i.e., the local variables files, f, and result).



## Map

Let's imagine an abstract datatype  $\text{Seq}\langle E \rangle$ , which represents a sequence of elements  $\in E$

e.g.  $[1,2,3,4] \in \text{Seq}\langle \text{Integer} \rangle$

Any datatype that has an Iterator can qualify as a sequence: e.g., array, list, set. A string is also a sequence, of characters, although Java's strings don't offer an Iterator. Python is more consistent in this respect. Not only are lists iterable, but so are strings, tuples (which are immutable lists), and even input streams (which produce a sequence of lines). Python's syntax is also more compact, so we'll be using it for code examples for the next few sections.

We'll have three operations for sequences: map, filter, and reduce. Let's look at each one in turn, and then look at how they work together.

**Map** applies a unary function to each element and returns a new list containing the results, in the same order.

$\text{map} : (E \rightarrow F) \times \text{Seq}\langle E \rangle \rightarrow \text{Seq}\langle F \rangle$

(in Python)

```
from math import sqrt
map(sqrt, [1,4,9,16]) # ==> [1.0, 2.0, 3.0, 4.0]
map(str.lower, ['A', 'b', 'C']) # ==> ['a', 'b', 'c']
```

Map is straightforward to implement in Python:

```
def map(f, seq):
 result = []
 for x in seq:
 result.append(f(x))
 return result
```

This operation captures a common pattern for operating over sequences: doing the same thing to each element of the sequence.

## Functions as Values

Let's pause here for a second, because we're doing something unusual with functions here. The map function takes a reference to a *function* as its first argument – not to the result of that function. When we wrote:

```
map(sqrt, [1,4,9,16])
```

we didn't *call* sqrt (like sqrt(25) is a call); instead we just used its name. In Python, the name of a function is a reference to an object representing that function. You can assign that object to another variable if you like, and it still behaves like sqrt:

```
mysqrt = sqrt
mysqrt(25) # ==> 5.0
```

You can also pass a reference to the function object as a parameter to another function; that's what we're doing with map here. You can use function objects the same way you would use any other data value in Python (like numbers or strings or objects).

Functions are **first-class** in Python, meaning that they can be assigned to variables, passed as parameters, used as return values, and stored in data structures. First-class functions are a very powerful programming idea. The first practical programming language that used them was Lisp, invented by John McCarthy at MIT. But the idea of programming with functions as first-class values

actually predates computers, tracing back to Alonzo Church's lambda calculus. The lambda calculus used the Greek letter  $\lambda$  to define new functions; this term stuck, and you'll see it as a keyword not only in Lisp and its descendants, but also in Python.

We've seen how to use built-in library functions as first-class values; how do we make our own? One way is using a familiar function definition, which gives the function a name:

```
def powerOfTwo(k):
 return 2**k
```

```
map(powerOfTwo, [1,2,3,4]) # ==> [2, 4, 8, 16]
```

When you only need the function in one place, however – which often comes up in programming with functions -- it's more convenient to use a **lambda expression**:

```
lambda k: 2**k
```

This expression represents a function of one argument (called *k*) that returns the value  $2^k$ . You can use it anywhere you would have used `powerOfTwo`:

```
(lambda k: 2**k) (5) # ==> 32
map(lambda k: 2**k, [1,2,3,4]) # ==> [0,1,2,3]
```

Python lambda expressions are unfortunately syntactically limited, to functions that can be written with just a return statement and nothing else (no if statements, no for loops, no local variables). But remember that's our goal with `map`/`filter`/`reduce` anyway, so it won't be a serious obstacle.

Guido Von Rossum, the creator of Python, has written a blog post about the design principle that led not only to first-class functions in Python, but first-class methods as well:

<http://python-history.blogspot.com/2009/02/first-class-everything.html>

## More Ways to Use Map

`Map` is useful even if you don't care about the return value of the function. When you have a sequence of mutable objects, for example, you can map a mutator operation over them:

```
map(IOBase.close, streams) # closes each stream on the list
map(Thread.join, threads) # waits for each thread to finish
```

Some versions of `map` (including Python's builtin `map`) also support mapping functions with multiple arguments. For example, you can add two lists of numbers element-wise:

```
import operator
map(operator.add, [1,2,3], [4,5,6]) # ==> [5, 7, 9]
```

## Filter

Our next important sequence operation is **filter**, which tests each element with a unary predicate. Elements that satisfy predicate are kept; those that don't are removed. A new list is returned; `filter` doesn't modify its input list.

$\text{filter} : (E \rightarrow \text{boolean}) \times \text{Seq}\langle E \rangle \rightarrow \text{Seq}\langle E \rangle$

Python examples:

```
filter(str.isalpha, ['x', 'y', '2', '3', 'a']) # ==> ['x', 'y', 'a']

def isOdd(x): return x % 2 == 1
filter(isOdd, [1,2,3,4]) # ==> [1,3]
```

```
filter(lambda s: len(s)>0, ['abc', '', 'd']) # ==> ['abc', 'd']
```

We can define filter in a straightforward way:

```
def filter(f, seq):
 result = []
 for x in seq:
 if f(x):
 result.append(x)
 return result
```

## Reduce

Our final operator, **reduce** combines the elements of the sequence together, using a binary function. In addition to the function and the list, it also takes an *initial value* that initializes the reduction, and that ends up being the return value if the list is empty.

$$\text{reduce} : (F \times E \rightarrow F) \times \text{Seq}\langle E \rangle \times F \rightarrow F$$

`reduce(f, list, init)` combines the elements of the list from left to right, as follows:

```
result0 = init
result1 = f(result0, list[0])
result2 = f(result1, list[1])
...
resultn = f(resultn-1, list[n-1])
```

result<sub>n</sub> is the final result for an n-element list.

Adding numbers is probably the most straightforward example:

```
reduce(operator.add, [1,2,3], 0) # ==> 6
```

There are two design choices in the reduce operation. First is how to whether to require an initial value. In Python's reduce function, the initial value is optional, and if you omit it, reduce uses the first element of the list as its initial value. So you get behavior like this instead:

```
result0 = undefined (reduce throws an exception if the list is empty)
result1 = list[0]
result2 = f(result1, list[1])
...
resultn = f(resultn-1, list[n-1])
```

This makes it easier to use reducers like *max*, which have no well-defined initial value:

```
reduce(max, [5,8,3,1]) # ==> 8
```

The second design choice is the order in which the elements are accumulated. For associative operators like add and max it makes no difference, but for other operators it can. Python's reduce is also called **fold-left** in other programming languages, because it combines the sequence starting from the left (the first element). **Fold-right** goes in the other direction:



$\text{fold-right} : (E \times F \rightarrow F) \times \text{Seq}\langle E \rangle \times F \rightarrow F$

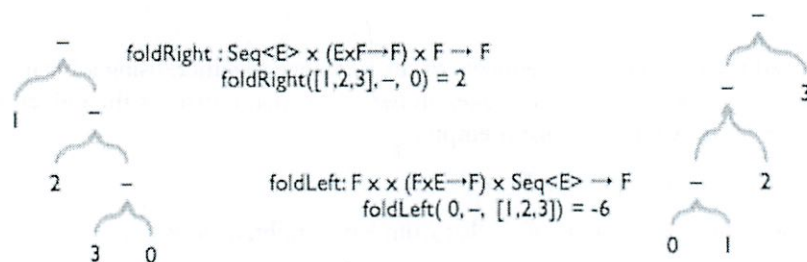
where  $\text{fold-right}(f, \text{list}, \text{init})$  of an  $n$ -element list produces  $\text{result}_n$  from this pattern:

```

result0 = init
result1 = f(list[n-1], result0)
result2 = f(list[n-2], result1)
...
resultn = f(list[0], resultn-1)

```

Two ways to reduce: from the left or the right



The return type of the reduce operation doesn't have to match the type of the list elements. For example, we can use reduce to glue together a sequence into a string:

```
reduce(lambda s, x: s+str(x), [1, 2, 3, 4], '') # ==> '1234'
```

Or to flatten out nested sublists into a single list:

```
reduce(operator.concat, [[1, 2], [3, 4], [], [5]], []) # ==> [1, 2, 3, 4, 5]
```

This is a useful enough sequence operation that we'll define it as **flatten**, although it's just a reduce step inside:

```

def flatten(list):
 return reduce(operator.concat, list, [])

```

## More Examples

Suppose we have a polynomial represented as a list of coefficients,  $a[0], a[1], \dots, a[n-1]$ , where  $a[i]$  is the coefficient of  $x^i$ . Then we can evaluate it using map and reduce:

```

def evaluate(a, x):
 xi = map(lambda i: x**i, range(0, len(a))) # [x^0, x^1, x^2, ..., x^n-1]
 axi = map(operator.mul, a, xi) # [a[0]*x^0, a[1]*x^1, ..., a[n-1]*x^n-1]
 return reduce(operator.add, axi, 0) # sum of axi

```

This code uses the convenient Python generator method `range(a,b)`, which generates a list of integers from  $a$  to  $b-1$ . In map/filter/reduce programming, this kind of method replaces a for loop that indexes from  $a$  to  $b$ .

Now let's look at a typical database query example. Suppose we have a database about digital cameras, in which each object is of type `Camera` with observer methods for its properties (`brand()`, `pixels()`, `cost()`, etc.). The whole database is in a list called `cameras`. Then we can describe queries on this database using `map/filter/reduce`:

```
What's the highest resolution Nikon sells?
reduce(max, map(Camera.pixels, filter(lambda c: c.brand() == "Nikon",
cameras)))
```

Relational databases use the `map/filter/reduce` paradigm (where it's called `project/select/aggregate`). SQL (Structured Query Language) is the de facto standard language for querying relational databases. A typical SQL query looks like this:

```
select max(pixels) from cameras where brand = "Nikon"
```

`cameras` is a **sequence** (a list of rows, where each row has the data for one camera)

where `brand="Nikon"` is a **filter**

`pixels` is a **map** (extracting just the `pixels` field from the row)

`max` is a **reduce**

## Finishing the Example

Going back to the example we started the lecture with, where we want to find the Java files in the project, let's try creating a useful abstraction for filtering:

```
def fileEndsWith(suffix):
 return lambda file: file.getName().endsWith(suffix)
```

`fileEndsWith` returns functions that are useful as filters; it takes a filename suffix like `".java"` and dynamically generates a function that you can use with `filter` to test for that suffix:

```
filter(fileEndsWith(".java"), files)
```

`fileEndsWith` is a different kind of beast than our usual functions. It's a **higher-order function**, meaning that it's a function that takes another function as an argument, or returns another function as its result. Higher-order functions are operations on the datatype of functions; in this case, `fileEndsWith` is a producer of functions.

Now let's use `map`, `filter`, and `flatten` to recursively traverse the folder tree:

```
def allFilesIn(folder):
 children = folder.listFiles()
 descendants = flatten(map(allFilesIn, filter(File.isDirectory, children)))
 return descendants + filter(File.isFile, children)
```

The first line gets all the children of the folder, which might look like this:

```
["src/client", "src/server", "src/Main.java", ...]
```

The second line is the key bit: it filters the children for just the subfolders, and then recursively maps `allFilesIn` against this list of subfolders! The result might look like this:

```
[["src/client/MyClient.java"], ["src/server/MyServer.java"], ...]
```

So we then have to flatten it to remove the nested sublists. Then we add the immediate children which are plain files (not folders), and that's our result.

We can also do the other pieces of the problem with map/filter/reduce. Once we have the list of files we want to extract words from, we're ready to load their contents. We can use map to get their pathnames as strings, open them, and then read in each file as a list of lines:

```
pathnames = map(File.getPath, files)
streams = map(open, pathnames)
lines = map(list, files)
```

This actually looks like a single map() that we want to apply three functions to, so let's pause to create another useful higher-order function: composing functions together.

```
def compose(f, g):
 """Requires that f and g are functions, f:A->B and g:B->C.
 Returns a function A->C by composing f with g."""
 return lambda x: g(f(x))
```

Now we can use a single map:

```
lines = map(compose(compose(File.getPath, open), list), files)
```

Better, since we already have three, let's design a way to compose an arbitrary chain of functions:

```
def chain(funcs):
 """Requires funcs is a list of functions [A->B, B->C, ..., Y->Z].
 Returns a fn A->Z that is the left-to-right composition of funcs."""
 return reduce(compose, funcs)
```

so that the map operation looks more like this:

```
lines = map(chain([File.getPath, open, list]), files)
```

Now we start to see the power of first-class functions – we can put functions into data structures and use operations on those data structures, like map, reduce, and filter, on the functions themselves!

Since this map will produce a list of lists of lines (one for each file), we need to flatten it to get a single line list, ignoring file boundaries:

```
allLines = flatten(map(chain([File.getPath, open, list]), files))
```

Then we split each line into words similarly:

```
words = flatten(map(str.split, lines))
```

And we're done. As promised, the control statements have disappeared.

## Benefits of Abstracting Out Control

Map/filter/reduce can often make code shorter and simpler, and allow the programmer to focus on the heart of the computation rather than on the details of loops, branches, and control flow.

By arranging our program in terms of map, filter, and reduce, and in particular using immutable datatypes and pure functions (i.e. functions that avoid mutating data) as much as possible, we've created more opportunities for safe concurrency. Maps and filters using pure functions over immutable datatypes are instantly parallelizable – invocations of the function on different elements of the sequence can be run in different threads, on different processors, even on different machines, and the result will still be the same.



## First-class Functions in Java

We've seen what first-class functions look like in Python; how does this all work in Java?

In Java, the only first-class values are primitive values (ints, booleans, characters, etc) and object references. But objects can carry functions with them, in the form of methods. So it turns out that the way to implement a first-class function, in an object-oriented programming language like Java that doesn't support first-class functions directly, is to use an object with a method representing the function.

We've actually seen this before several times already:

- The Runnable object that you pass to a Thread is a first-class function, void run().
- The ActionListener object that you register with the graphical user interface toolkit to get keyboard events is a bundle of several functions, keyPressed(KeyEvent), keyReleased(KeyEvent), etc.
- The Visitor object that we created to implement functions over recursive datatypes did indeed represent a function -- with several variants, one for each variant of the datatype.

This design pattern is called a **functional object** or functor, an object whose purpose is to represent a function.

For the sake of implementing map/filter/reduce in Java, let's generalize this notion to a generic unary function interface:

```
/**
 * A Function<T,U> represents a unary function from T to U,
 * i.e. f:T->U.
 */
public interface Function<T,U> {
 /**
 * Apply this function.
 * @param t object to apply this function to
 * @return the result of applying this function to t.
 */
 public U apply(T t);
}
```

Then we can write map() like so:

```
/**
 * Apply a function to every element of a list.
 * @param f function to apply
 * @param list list to iterate over
 * @return [f(list[0]), f(list[1]), ..., f(list[n-1])]
 */
public static <T,U> List<U> map(Function<T,U> f, List<T> list) {
 List<U> result = new ArrayList<U>();
 for (T t : list) {
 result.add(f.apply(t));
 }
 return result;
}
```

And here's an example of using `map()`:

```
// anonymous classes like the one below are effectively lambda expressions
Function<String,String> toLowerCase = new Function<String,String>() {
 public String apply(String s) { return s.toLowerCase(); }
};

map(toLowerCase, Arrays.asList(new String[] {"A", "b", "c"}));
```

Obviously verbose, and Java is not practical for functional programming. But the notion of functors is widely used, and useful, as we've seen from examples like `Runnable` and `Visitor`.

## Higher-Order Functions in Java

`Map/filter/reduce` are obviously higher order functions. But let's look at two others that we introduced in today's lecture: `compose()` and `chain()`.

`Compose()` has a straightforward implementation, and in particular once you get the types of the arguments and return value right, Java's strong typing makes it pretty much impossible to get the method body wrong:

```
/**
 * Compose two functions.
 * @param f function A->B
 * @param g function B->C
 * @return new function A->C formed by composing f with g
 */
public static <A,B,C> Function<A,C> compose(final Function<A,B> f,
 final Function<B,C> g) {
 return new Function<A,C>() {
 public C apply(A t) { return g.apply(f.apply(t)); }
 };
}
```

It turns out that we *can't* write `chain()` in strongly-typed Java, except in a very restricted form in which all the functions in the chain have the identical input and output types. This is because Lists must be homogeneous – `List<Function<A,A>>`.

```
/**
 * Compose a chain of functions.
 * @param list list of functions A->A to compose
 * @return function A->A made by composing list[0] ... list[n-1]
 */
public static <A> Function<A,A> chain(List<Function<A,A>> list) {
 return reduce(
 list,
 new BinOp<Function<A,A>, Function<A,A>, Function<A,A>>() {
 public Function<A, A> apply(Function<A, A> t, Function<A, A> u) {
 return compose(t, u);
 }
 },
 new Function<A,A>() {
 public A apply(A t) { return t; }
 }
);
}
```

```

package mfr;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * The map/filter/reduce operations, and the interfaces they rely on.
 */
public class MapFilterReduce {

 /**
 * Apply a function to every element of a list.
 * @param f function to apply
 * @param list list to iterate over
 * @return [f(list[0]), f(list[1]), ..., f(list[n-1])]
 */
 public static <T, U> List<U> map(Function<T, U> f, List<T> list) {
 List<U> result = new ArrayList<U>();
 for (T t : list) {
 result.add(f.apply(t));
 }
 return result;
 }

 /**
 * Filter a list for elements satisfying a predicate. Doesn't
 * modify the list.
 * @param p predicate to test
 * @param list list to iterate over
 * @return a new list containing all list[i] such that p(list[i]) == true,
 * in the same order they appeared in the original list.
 */
 public static <T> List<T> filter(Predicate<T> p, List<T> list) {
 List<T> result = new ArrayList<T>();
 for (T t : list) {
 if (p.apply(t)) {
 result.add(t);
 }
 }
 return result;
 }

 /**
 * Combine the elements of a list from left to right using a binary operator.
 * @param op binary operator
 * @param list list to iterate over
 * @param init initial value
 * @return (((init op list[0]) op list[1]) ...) op list[n-1]
 */
 public static <T, U> U reduce(List<T> list, BinOp<U, T, U> op, U init) {
 U result = init;

```



```

 for (T t: list) {
 result = op.apply(result, t);
 }
 return result;
}

/**
 * A Function<T,U> represents a unary function f:T->U.
 */
public static interface Function<T, U> {
 /**
 * Apply this function.
 *
 * @param t
 * object to apply this function to
 * @return the result of applying this function to t.
 */
 public U apply(T t);
}

/**
 * Predicate<T> represents a boolean predicate over the type T, i.e. a
 * function p : T -> boolean. Alternatively, Predicate<T> represents the
 * subset of T for which the predicate returns true.
 */
public static interface Predicate<T> {
 /**
 * Test this predicate on an object.
 *
 * @param t
 * object to test
 * @return true iff this predicate is true for t.
 */
 public boolean apply(T t);
}

/**
 * BinOp<T,U,V> represents a binary function r : T x U -> V.
 */
public interface BinOp<T, U, V> {
 /**
 * Apply this binary operation.
 *
 * @param t
 * T value to apply this function to
 * @param u
 * U value to apply this function to
 * @return the result of applying this function to (t,u).
 */
 public V apply(T t, U u);
}

/**
 * Compose two functions.

```

```

* @param f function A->B
* @param g function B->C
* @return new function A->C formed by composing f with g
*/
public static <A,B,C> Function<A,C> compose(final Function<A,B> f, final Function<B,C> g
) {
 return new Function<A,C>() {
 public C apply(A t) { return g.apply(f.apply(t)); }
 };
}

// is the following the same as the chain() we wrote in Python?
/**
 * Compose a chain of functions.
 * @param list list of functions A->A to compose
 * @return function A->A made by composing list[0] ... list[n-1]
 */
public static <A> Function<A,A> chain(List<Function<A,A>> list) {
 return reduce(
 list,
 new BinOp<Function<A,A>, Function<A,A>, Function<A,A>>() {
 public Function<A, A> apply(Function<A, A> t, Function<A, A> u) {
 return compose(t, u);
 }
 },
 new Function<A,A>() {
 public A apply(A t) { return t; }
 }
);
}

// Examples
public static void main(String[] args) {
 // anonymous classes like the one below are effectively lambda expressions
 Function<String,String> toLowerCase = new Function<String,String>() {
 public String apply(String s) { return s.toLowerCase(); }
 };

 map(toLowerCase, Arrays.asList(new String[] { "A", "b", "c" }));
}
}

```

```
The definitions below show how map/filter/reduce can be implemented
in Python. DON'T USE THESE IN PRACTICE. Use Python's builtin
map/filter/reduce functions instead.
```

```
def map(f, seq):
 result = []
 for x in seq:
 result.append(f(x))
 return result

def filter(f, seq):
 result = []
 for x in seq:
 if f(x):
 result.append(x)
 return result

def reduce(op, seq, init):
 result = init
 for x in seq:
 result = op(result, x)
 return result
```

```
map examples
```

```
from math import sqrt
map(sqrt, [1,4,9,16]) # ==> [1.0, 2.0, 3.0, 4.0]
map(str.lower, ["A", "b", "C"]) # ==> ['a', 'b', 'c']
```

```
functions are first-class: you can assign them, pass them, return them, etc.
```

```
mysqrt = sqrt
mysqrt(25) # ==> 5.0
```

```
defining your own functions: either by name (def) or anonymously (lambda)
```

```
def powerOfTwo(k):
 return 2**k
```

```
map(powerOfTwo, [1,2,3,4]) # ==> [2, 4, 8, 16]
```

```
(lambda k: 2**k) (5) # ==> 32
map(lambda k: 2**k, [1,2,3,4]) # ==> [2, 4, 8, 16]
```

```
import io.IOBase
map(IOBase.close, streams) # closes each stream on the list
```

```
import threading.Thread
map(Thread.join, threads) # waits for each thread to finish
```

```
import operator
map(operator.add, [1,2,3], [4,5,6]) # ==> [5, 7, 9]
```



##### filter examples

```
filter(str.isalpha, ['x', 'y', '2', '3', 'a']) # ==> ['x', 'y', 'a']
```

```
def isOdd(x): return x % 2 == 1
```

```
filter(isOdd, [1,2,3,4]) # ==> [1,3]
```

```
filter(lambda s: len(s)>0, ['abc', '', 'd']) # ==> ['abc', 'd']
```

##### reduce examples

```
reduce(operator.add, [1,2,3], 0) # ==> 6
```

```
reduce(max, [5,8,3,1]) # ==> 8
```

```
reduce(lambda s,x: s+str(x), [1,2,3,4], '') # ==> '1234'
```

```
reduce(operator.concat, [[1,2],[3,4],[],[5]], []) # ==> [1,2,3,4,5]
```

```
def flatten(list):
```

```
 return reduce(operator.concat, list, [])
```

##### bigger examples

```
def evaluate(a, x):
```

```
 xi = map(lambda i: x**i, range(0, len(a))) # ==> [x^0, x^1, x^2, ..., x^n-1]
```

```
 axi = map(operator.mul, a, xi) # ==> [a[0]*x^0, a[1]*x^1, a[2]*x^2, ..., a[n-1]*x^n-1]
```

```
 return reduce(operator.add, axi, 0) # ==> sum of axi
```

```
What's the highest resolution Nikon sells?
```

```
reduce(max, map(Camera.pixels, filter(lambda c: c.brand() == "Nikon", cameras)))
```

```
def fileEndsWith(suffix):
```

```
 return lambda file: file.getName().endsWith(suffix)
```

```
filter(fileEndsWith(".java"), files)
```

```

package words;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Words1 {

 /** Find all the files in the filesystem subtree rooted at folder.
 * @param folder root of subtree. Requires folder.isDirectory() == true.
 * @return list of all ordinary files (not folders) that have folder as their ancestor.
 * @throws IOException if an error occurs while accessing the filesystem
 */
 public static List<File> allFilesIn(File folder) throws IOException {
 List<File> files = new ArrayList<File>();
 for (File f: folder.listFiles()) {
 if (f.isDirectory()) {
 files.addAll(allFilesIn(f));
 } else if (f.isFile()) {
 files.add(f);
 }
 }
 return files;
 }

 /** Filter a list of files to those that end with suffix.
 * @param files list of files (all non-null)
 * @param suffix string to test
 * @return a new list consisting of only those files whose names end with suffix
 */
 public static List<File> onlyFilesWithSuffix(List<File> files, String suffix) {
 List<File> result = new ArrayList<File>();
 for (File f : files) {
 if (f.getName().endsWith(suffix)) {
 result.add(f);
 }
 }
 return result;
 }

 /**
 * @param files
 * @return
 * @throws IOException
 */
 public static List<String> getWords(List<File> files) throws IOException {
 List<String> words = new ArrayList<String>();
 for (File f : files) {
 BufferedReader r = new BufferedReader(new FileReader(f));

```

```
String line;
for (line = r.readLine(); line != null; line = r.readLine()) {
 // split on \W (non-word characters, like spaces and punctuation)
 for (String word : line.split("\\W+")) {
 // split can return empty strings, so omit them
 if (!word.isEmpty()) {
 words.add(word);
 }
 }
}
return words;
}

public static void main(String[] args) {
 try {
 List<File> allFiles = allFilesIn(new File("."));
 List<File> javaFiles = onlyFilesWithSuffix(allFiles, ".java");
 List<String> words = getWords(javaFiles);
 for (String s : words) { System.out.println(s); }
 } catch (IOException e) {
 e.printStackTrace();
 }
}
```



```
from java.io import File
from java.lang import String
from operator import concat

def flatten(l):
 return reduce(concat, l, [])

def allFilesIn(folder):
 children = folder.listFiles()
 descendents = flatten(map(allFilesIn, filter(File.isDirectory, children)))
 return descendents + filter(File.isFile, children)

def endsWith(suffix):
 return lambda f: f.getPath().endsWith(suffix)

def compose(f, g):
 """Requires that f and g are functions, f:A->B and g:B->C.
 Returns a function A->C by composing f with g."""
 return lambda x: g(f(x))

def chain(funcs):
 """Requires that funcs is a list of functions [A->B, B->C, ..., Y->Z].
 Returns a function A->Z that is the left-to-right composition of funcs."""
 return reduce(compose, funcs)

readIn = chain([File.getPath, open, list])

def splitIntoWords(s):
 nonempty = lambda s: len(s) > 0
 return filter(nonempty, list(String.split(s, "\\W+")))

the whole program

roots = map(File, ["."])
files = filter(endsWith(".java"), flatten(map(allFilesIn, roots)))
lines = flatten(map(readIn, files))
words = flatten(map(splitIntoWords, lines))
print "\n".join(words)
```

6.005  
Recitation

11/15

Using Python today

Look at URL

Find all lines that don't have token

map: function  $\times$  list  $\rightarrow$  list

example: `map(len, ["alan", "paz", "patricu"])`

$\hookrightarrow$  produces `[4, 4, 7]`

lists do not need to contain same types

list class functions

- Using as object

- can pass it along

filters | function  $\times$  list  $\rightarrow$  list

function  $\rightarrow$  boolean

example `filter(lambda x: x % 2 == 0`

2

Output [4,4]

reduce function  $\times$  list  $\langle E \rangle$  [seed]  $\rightarrow$  single value  
initial argument

function  $\times$  list  $\langle E \rangle$  [seed]  $\rightarrow S$   
 $\times$  initial argument       $\uparrow$  single value

function  $S \times E \rightarrow S$

So  $E \times E \rightarrow E$   
w/o seed  
[E1 E2 E3]

$(E1 + E2) + E3$

w/ seed  
[E1 E2 E3 S]

$((S + E1) + E2) + E3$

like bank account

`reduce(perform, [T1, T2, T3, T4], 'initial-account')`

`def perform(B, T):  
 return T(B)`



3

This is fairgame for exam  
- Conceptual stuff about functions

---

Web cant example

---

Rewrite using list comprehension

Can't do reduce in List comprehension

return filter (lambda x: x.cant(token), .map (  
Strip, Stream, readLines ()))

---

Java

---

↓ want to be able to return any type  
or take anything in

```
public interface Function (A) {
 <A, R>
 public R apply (A arg)
```

3

```
List<A> filter (Function<A, Boolean> f, List<A> list) {
 List<A> result = new
 for
 ↓
```

4

```
for (A a : list) {
 if (f.apply(a)) {
 result.add(a);
 }
}
```

3

3

```
return result;
}
```

back in interface

```
public class MyFunction implements Fun<String, Boolean> {
 private String token;
 public Boolean apply (String args) {
 return arg.contains
 }
}
```

3

```
public MyFunction (String, token) {
 this.token = token;
}
```

3

5

Why is map + filter cool vs reduce?

Can parallelize them

Not dependent on results

Extra exercise



6.005 L16  
Little Languages

11/16

- ☐ code as ~~data~~ data
- ☐ domain-specific languages
- ☐ PCAP

PS 7 due Thur @ midnight

Project 2 Signups by Fri

Quiz 2 Mon 11-12 in Walker gym

Last lecture today

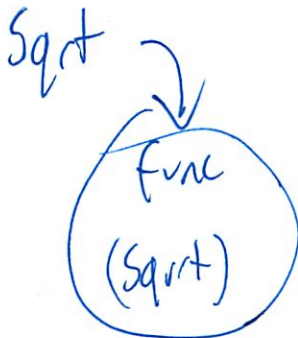
Last recitation tomorrow

Proj 2 work time after

Review

Started looking at code as data on mon w/ 1st Class

↓ - Functions  
Can use them as other data values

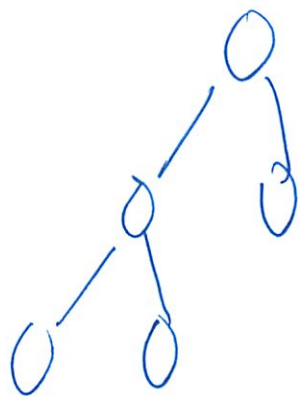


powerful since can do map, filter, reduce

②

② kinda did this before

abs  $\rightarrow$  Parser  $\rightarrow$



$\leftarrow$  expression; ADT

is basically a first class representation of song  
its like code to play song

③ Also the visitor ~~the~~ object



visitor interfaces

interface Visitor {

on (N) { ... }

on (V) { ... }

on (T) { ... }

}

as opposed to interpreter visitor

3

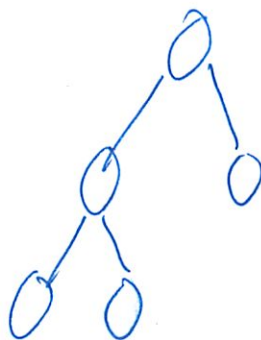
Visitor object 1st class

- passing it around
- but not as manipulable

---

Today: Put stuff together into something that is ready for change

So could have general data type, so can do



And be able to modify this structure

---

Want to be able to do repetitive music

- round - people singing same thing - but come in at certain time
- Canon - singing diff things at same time (✓) (diff people)
- fugue (Bach)



(9)

Want to do this w/ as little input from

programmer/composer

Structure

↳ datatype def for recursive data types

Music = Note (p: Pitch, dur: ~~int~~ double, instr: Instrument)

enum  
type

? # beats  
(can have fraction)

? to make  
voices distinct

OneThing

what midi synthesizer needs to play

(immutable)

+ Rests (dur: double)

+ Concat (m1: Music, m2: Music)

So what type of objects will we see at runtime

L Tree shaped

If wanted lists

+ Cons (f: OneThing, m: Music)

But want tree for more flexibility

But w/ tree there is no way to represent empty list

↳ always think about 0, 1, (2), n  
sometimes

5

Could add

+ Empty()

But to keep # elements small, can do just  
Rest(0)

Operations

creator of data type  
↓ constructor

notes: String x Instrument → Music

"C D E F | F E D C |"

↑ domain specific notation (easy to type in  
for people in field)

creator

observer

duration: Music → double

observer play: Music x Synthesizer → void

↑ sound is representation  
of what's in music

? not generating  
any thing in Java

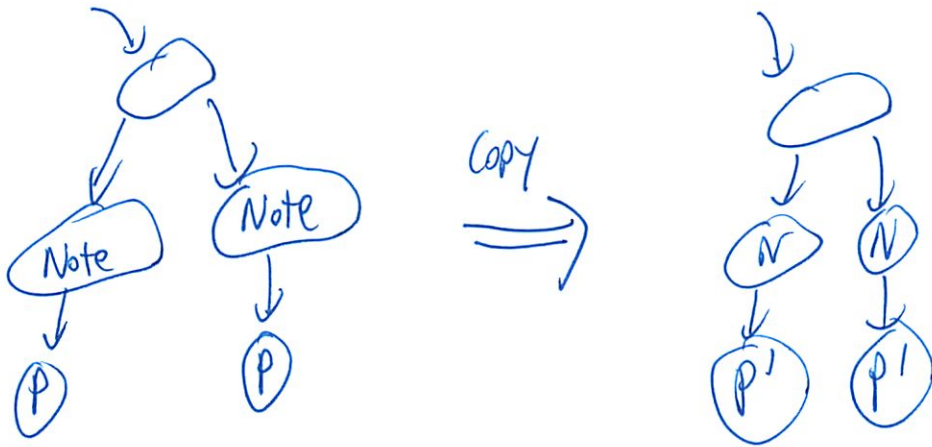
called for side effect

transpose: Music x int → Music

↑ # of semitones up  
(half steps)

Q

So what do we actually do?



Easy to dig inside

Represent data as datatype so can modify

---

Using interpreter pattern

```
interface Music {
 double duration()
```

}

But also Visitor

```
<T> T accept(Visitor<T>)
```

will do crawling over tree

Factory method ~ / notes to create

Empty rest for blank



7

transpose  
↳ builds Visitor object

don't need to transpose much

rests  $\rightarrow$  nothing

Concat  $\rightarrow$  just transpose children

So pattern of "do nothing" visitor

So override w/ behavior ya want

↳ simple way to write what only changes part of tree

Change Tempo

↳ changes Rests and Notes

So for Music datatype - need to add

+ Together ( $m1$ : Music,  $m2$ : Music)

$\nearrow$  like concat - but this is serial  
but Together is — parallel

datatype is also like composite datatype

nodes: Note, Rest

combiners: Concat, Together

many ADTs use composite pattern

8

delay:  $\text{Music} \times \underbrace{\text{table}}_{\substack{\text{\# of beats} \\ \text{delay}}} \rightarrow \text{Music}$   
to insert in beginning

So now we can write a simple and

Can write code in (in Python)

```
play(together([ron Yar Boat, delay(ron Yar Boat, 4)]))
```

```
play(together([ron Yar Boat, delay(transpose(ron Yar Boat;
Octave), 4)]))
```

With just a few operators can transpose/modify music  
we typed in previously

Can define ~~canon~~

Canon:  $\text{Music} \times \underbrace{\text{int}}_{\text{voices}} \times \underbrace{\text{table}}_{\text{delay}} \times \underbrace{(\text{Music} \Rightarrow \text{Music})}_{\text{takes function}} \rightarrow \text{Music}$   
- like: transpose  
- instrument replace  
- etc

9

~~transposer~~ :  $\text{int} \rightarrow (\text{Music} \rightarrow \text{Music})$   
↑ returns function  
you can apply elsewhere

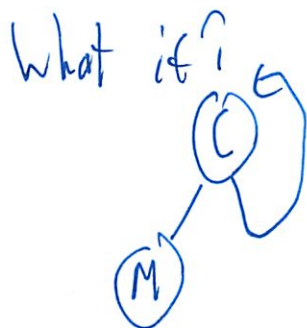
So can represent things easily

Now want stuff to go on forever

~~forever~~ :  $\text{Music} \rightarrow \text{Music}$

Could we do this w/ concat?

↳ might run out of memory eventually



But visitors will do same thing  
and can't even create w/ current  
data type

Must be able  $\text{concat}(m, \dots)$

↑ generally place  
placeholder null

Then mutate  $\text{concat}.m2 = m2$

So we add a new type/variant

Add an  $\infty$  loop node



(10)



Then play can or (forever (my b), 4, 4)  
row VarBuff

Pachelbel Canon has a baseline

So add accompany

repeats baseline as long as needed

---

Using Music essentially as code

Takeaways

1. Code is not fixed

↳ Behavior can be modified/mutated at run time

2. Domain / Content specific languages

↳ can input ABC notation

↳ and operation a user in this language needs

PCAP from b.c.d

↳ primitives (notes, rests) - abstractions (naming + functions in Python)

↳ combinations (concat, canon) - patterns

# 6.005 elements of software construction

## Little Languages

Rob Miller  
Fall 2011

© Rob Miller 2011

## Today's Topics

### Functionals

- Objects representing executable code

### Higher-order functions

- Functions that accept functions as arguments or return them as results

### Domain-specific languages

- PCAP: primitives, combination, abstraction pattern

© Rob Miller 2011

## Representing Code with Data

Consider a datatype representing language syntax

- Formula is the language of propositional logic formulas
- a Formula value represents program code in a data structure; i.e.  
`new And(new Var("x"), new Var("y"))`  
 has the same semantic meaning as the Java code  
`x && y`
- but a Formula value is a **first-class object**
  - first-class: a value that can be passed, returned, stored, manipulated
  - the Java expression `"x && y"` is **not** first-class

© Rob Miller 2011

## Representing Code as Data

Recall the visitor pattern

- A visitor represents a function over a datatype
  - e.g. `new SizeVisitor()` represents `size : List → int`

```
public class SizeVisitor<E> implements ListVisitor<E,Integer> {
 public Integer visit(Empty<E> l) { return 0; }
 public Integer visit(Cons<E> l) { return 1 + l.rest().accept(this); }
}
```

A visitor represents code as a first-class object, too

- A visitor is an **object** that can be passed around, returned, and stored
- But it's also a **function** that can be invoked

Today's lecture will see more examples of code as data

© Rob Miller 2011

## Today's Problem: Music

Interesting music tends to have a lot of repetition

- Let's look at rounds, canons, fugues
- A familiar simple **round** is "Row Row Row Your Boat": one voice starts, other voices enter after a delay  
 Row row row your boat, gently down the stream, merrily merrily ...  
 Row row row your boat, gently down the stream...
- Bach was a master of this kind of music
  - Recommended reading: *Godel Escher Bach*, by Douglas Hofstadter

Recall our MIDI piano from early lectures

- A song could be represented by Java code doing a sequence of calls on a state machine:  
`machine.play(E); machine.play(D); machine.play(C); ...`
- We want to capture the code that operates this kind of machine as **first-class data objects** that we can manipulate, transform, and repeat easily

© Rob Miller 2011

## Music Data Type

Let's start by representing simple tunes

- Music = `Note(duration:double, pitch:Pitch, instr:Instrument)`
  - + `Rest(duration:double)`
  - + `Concat(m1:Music, m2:Music)`
- duration is measured in **beats**
- Pitch represents note frequency (e.g. C, D, E, F, G; essentially the keys on the piano keyboard)
- Instrument represents the instruments available on a MIDI synthesizer
- Design questions**
  - is this a tree or a list? what would it look like defined the other way?
  - what is the "empty" Music object?
    - it's usually good for a data type to be able to represent *nothing*
    - avoid null
  - what are the rep invariants for Note, Rest, Concat?

© Rob Miller 2011

## A Few of Music's Operations

notes : String x Instrument → Music

requires string is in a subset of abc music notation

e.g. notes("E D C D | E E E2 |", PIANO)

1 beat note

2-beat note

abc notation  
can also encode  
sharps & flats,  
higher/lower octaves

duration : Music → double

returns total duration of music in beats

e.g. duration(Concat(m1, m2)) = duration(m1) + duration(m2)

transpose : Music x int → Music

returns music with all notes shifted up or down in pitch by the given number of semitones (i.e., steps on a piano keyboard)

play : Music → void

effects plays the music

all these operations also  
have precondition that  
parameters are non-null

© Robert Martin 2011

## Implementation Choices

**Creators can be constructors or factory methods**

➤ Java constructors are limited: interfaces can't have them, and constructor can't choose which runtime type to return

• new C() must always be an object of type C,

• so we can't have a constructor Music(String, Instrument), whether Music is an interface or an abstract class

**Observers & producers can be methods or visitors**

➤ Methods break up function into many files; visitor is all in one place

➤ Adding a method requires changing source of classes (not always possible)

➤ Visitor keeps dependencies out of data type itself (e.g. MIDI dependence)

➤ Method has direct access to private rep; visitor needs to use observers

**Producers can also be new subclasses of the datatype**

➤ e.g. Music = ... + Transpose(m:Music, semitones:int)

➤ Defers the actual evaluation of the function

➤ Enables more sharing between values

➤ Adding a new subclass requires changing all visitors

## Duality Between Interpreter and Visitor

**Operation using interpreter pattern**

➤ Adding new operation is hard (must add a method to every existing class)

➤ Adding new class is easy (changes only one place: the new class)

**Operation using visitor pattern**

➤ Adding new operation is easy (changes only one place: the new visitor)

➤ Adding new class is hard (must add a method to every existing visitor)

© Robert Martin 2011

## Multiple Voices

**For a round, the parts need to be sung simultaneously**

Music = Note(duration:double, pitch:Pitch, instr:Instrument)

+ Rest(duration:double)

+ Concat(m1:Music, m2:Music)

+ Together(m1:Music, m2:Music)

➤ Here's where our decision to make Concat() tree-like becomes very useful

• Suppose we instead had:

Concat = List<Note + Rest>

Together = List<Concat>

• What kinds of music would we be unable to express?

**Composite pattern**

➤ The composite pattern means that groups of objects (composites) can be treated the same way as single objects (primitives)

Music and Formula are  
composite data types.

➤  $T = C_1(\dots, T) + \dots + C_n(\dots, T) + P_1(\dots) + \dots + P_m(\dots)$

composites

primitives

## Simple Rounds

**We need one more operation:**

delay : Music x double → Music

delay(m, dur) = concat(rest(dur), m)

**And now we can express Row Row Row Your Boat**

rrryb = notes("C C C3/4 D/4 E | E3/4 D/4 E3/4 F/4 G2 | ...", PIANO)

together(rrryb, delay(rrryb, 4))

• Two voices playing together, with the second voice delayed by 4 beats

➤ This pattern is found in all rounds, not just Row Row Row Your Boat

➤ Abstract out the common pattern

canon : Music x double x int → Music

canon(m, dur, n) = m if n == 1

together(m, canon(delay(m, dur), dur, n-1)) if n > 1

➤ The ability to capture a general pattern like canon() is one of the advantages of music as a first-class object rather than merely a sequence of play() calls

© Robert Martin 2011

## Distinguishing Voices

**We want each voice in the canon to be distinguishable**

➤ e.g. an octave higher, or lower, or using a different instrument

➤ So these **operations** over Music also need to be first-class objects that can be passed to canon()

**Extend canon() to apply a function to the repeated melody**

canon : Music x int x double x (Music → Music) → Music

e.g. canon(rrryb, 4, 4, transposer(OCTAVE))

produces 4 voices, each one octave higher than the last

transposer : int → (Music → Music)

transposer(semitones) = lambda m: transpose(m, semitones)

**canon() is a higher-order function**

➤ A higher-order function takes a function as an argument or returns a function as its result

© Robert Martin 2011



## Counterpoint

A canon is a special case of a more general pattern

- Counterpoint is  $n$  voices singing related music, not necessarily delayed  
 $\text{counterpoint} : \text{Music} \times (\text{Music} \rightarrow \text{Music}) \times \text{int} \rightarrow \text{Music}$
- Expressed as counterpoint, a canon applies two functions to the music:  
 delay and transform  
 $\text{canon}(m, d, f, n) = \text{counterpoint}(m, f \circ \text{delayer}(d), n)$   
 $\text{delayer} : \text{int} \rightarrow (\text{Music} \rightarrow \text{Music})$   
 $\text{delayer}(d) = \lambda m. \text{delay}(m, d)$

### Another general pattern

function composition  $\circ : (U \rightarrow V) \times (T \rightarrow U) \rightarrow (T \rightarrow V)$

## Repeating

A line of music can also be repeated by the same voice

$\text{repeat} : \text{Music} \times \text{int} \times (\text{Music} \rightarrow \text{Music}) \rightarrow \text{Music}$   
 e.g.  $\text{repeat}(\text{rrryb}, 2, \text{octaveHigher}) = \text{concat}(\text{rryb}, \text{octaveHigher}(\text{rryb}))$

- Note the similarity to counterpoint():  
 $\text{counterpoint} : m \text{ together } f(m) \text{ together } \dots \text{ together } f^{n-1}(m)$   
 $\text{repetition} : m \text{ concat } f(m) \text{ concat } \dots \text{ concat } f^{n-1}(m)$

➤ And in other domains as well:

sum:  $x + f(x) + \dots + f^{n-1}(m)$

product:  $x \cdot f(x) \cdot \dots \cdot f^{n-1}(m)$

➤ There's a general pattern here, too; let's capture it

series:  $T \times (T \times T \rightarrow T) \times (T \rightarrow T) \times \text{int} \rightarrow T$

initial value    binary op     $f$      $n$   
 $\text{counterpoint}(m, f, n) = \text{series}(m, \text{together}, f, n)$   
 $\text{repeat}(m, f, n) = \text{series}(m, \text{concat}, f, n)$

## Repeating Forever

Music that repeats forever is useful for canons

$\text{forever} : \text{Music} \rightarrow \text{Music}$

$\text{play}(\text{forever}(m))$  plays  $m$  repeatedly, forever

$\text{duration}(\text{forever}(m)) = +\infty$

double actually has a value for this:  
 Double.POSITIVE\_INFINITY

$\text{Music} = \text{Note}(\text{duration}:\text{double}, \text{pitch}:\text{Pitch}, \text{instr}:\text{Instrument})$   
 +  $\text{Rest}(\text{duration}:\text{double})$   
 +  $\text{Concat}(m1:\text{Music}, m2:\text{Music})$   
 +  $\text{Together}(m1:\text{Music}, m2:\text{Music})$   
 + **Forever( $m:\text{Music}$ )**

why can't we implement forever() using repeat(), or any of the existing Music subtypes?

- Here's the Row Row Row Your Boat round, forever:  
 $\text{canon}(\text{forever}(\text{rrryb}), 4, 4, \text{octaveHigher})$

## Accompaniment

$\text{accompany} : \text{Music} \times \text{Music} \rightarrow \text{Music}$

repeats second piece until its length matches the first piece

melody line  
 -----  
 bass line or drum line,  
 repeated to match melody's length

$\text{accompany}(m, b) =$

$\text{together}(m, \text{repeat}(b, \text{identity}, \text{duration}(m)/\text{duration}(b)))$  if  $\text{duration}(m)$  finite  
 $\text{together}(m, \text{forever}(b))$  if  $\text{duration}(m)$  infinite

## Pachelbel's Canon

(well, the first part of it, anyway...)

$\text{pachelbelBass} = \text{notes}(\text{"D,2 A,,2 | B,,2 ^F,, | ... |"}, \text{CELLO})$

$\text{pachelbelMelody} = \text{notes}(\text{"^A^F' 2 E' 2 | D' 2 ^A^C' 2 | ... | ... | ... |"}, \text{VIOLIN})$

$\text{pachelbelCanon} = \text{canon}(\text{forever}(\text{pachelbelMelody}), 3, 16)$

$\text{pachelbel} = \text{concat}(\text{pachelbelBass}, \text{accompany}(\text{pachelbelCanon}, \text{pachelbelBass}))$

## Little Languages

We've built a new language embedded in Java

- Music data type and its operations constitute a **language** for describing music generation
- Instead of just solving one problem (like playing Row Row Row Your Boat), build a language or toolbox that can solve a range of related problems (e.g. Pachelbel's canon)
- This approach gives you more flexibility if your original problem turns out to be the wrong one to solve (which is not uncommon in practice!)
- Capture common patterns as reusable abstractions

**Formula was an embedded language too**

- Formula combined with SAT solver is a powerful tool that solves a wide range of problems

## Embedded Languages

Useful languages have three critical elements

|                      | Java                              | Formula language            | Music language                                   |
|----------------------|-----------------------------------|-----------------------------|--------------------------------------------------|
| Primitives           | 3, false                          | Var, Bool                   | notes, rest                                      |
| Means of Combination | +, *,<br>==, &&,<br>  , ...       | and, or, not                | together,<br>concat,<br>transpose,<br>delay, ... |
| Means of Abstraction | variables,<br>methods,<br>classes | naming + methods<br>in Java | naming + functions in<br>Python                  |

➤ 6.01 calls this PCAP (the Primitive-Combination-Abstraction pattern)

© Stephen Chong 2011

## Summary

Review of many concepts we've seen in 6.005

➤ Abstract data types, recursive data types, interpreter/visitor, composite, immutability

### Code as data

➤ Recursive datatypes, visitors, and functional objects are all ways to express behavior as data that can be manipulated and changed programmatically

### Higher-order functions

➤ Operations that take or return functional objects

### Building languages to solve problems

➤ A language has greater flexibility than a mere program, because it can solve large classes of related problems instead of a single problem

➤ Composite, interpreter, visitor, and higher-order functions are useful for implementing powerful languages

➤ But in fact any well-designed abstract data type is like a new language

© Stephen Chong 2011

```

from music.MusicLanguage import *
from music.Instrument import *
from music import Pitch
from music.Pitch import OCTAVE
from java.lang.Double import POSITIVE_INFINITY # used for duration of forever()

#
//////////////////////////////////////
// Playing the music with MIDI
//////////////////////////////////////
#

from music.midi import MusicPlayer

def play(music):
 ''' play music with the MIDI synthesizer '''
 MusicPlayer().play(music)

#
//////////////////////////////////////
// General higher-order functions
//////////////////////////////////////
#

def identity(x):
 '''identity:X->X is the identity function.'''
 return x

def compose(f, g):
 '''f:A->B, g:B->C; returns h:A->C such that h=f o g.'''
 return lambda x: g(f(x))

def repeated(f, n):
 '''f:X->X, n:int >= 0. Returns f^n, i.e. f composed with itself n times.'''
 if n==0:
 return identity;
 else:
 return compose(repeated(f, n-1), f)

def series(e, binop, f, n):
 '''e:E, binop:ExE->E, f:E->E, n:int >= 1; also requires binop to be associative.
 Returns e binop f(e) binop f^2(e) binop ... binop f^{n-1}(e).'''
 return reduce(binop, reduce(lambda fs,i: fs + [f(fs[-1])], range(1, n), [e]))

#
//////////////////////////////////////
// Functional objects that transform Music in interesting ways
//////////////////////////////////////
#

def delayer(beats):

```



```

'''beats:int >= 0; returns f:Music->Music that delays music by that number of beats'''
return lambda music: delay(music, beats)

def tempoChanger(speedup):
 '''speedup:number > 0; returns f:Music->Music that speeds up music by a factor m'''
 return lambda music: changeTempo(music, speedup)

def instrumentChanger(instr):
 '''instr:Instrument; returns f:Music->Music that plays all the music with instr.'''
 return lambda music: changeInstrument(music, instr)

def instrumentReplacer(oldInstr, newInstr):
 '''oldInstr,newInstr:Instrument; returns f:Music->Music that replaces notes played by
oldInstr
 with the same note played by newInstr.'''
 return lambda music: replaceInstrument(music, oldInstr, newInstr)

def instrumentSequence(instrs):
 '''instrs:list(Instrument); returns f:Music->Music that replaces instrs[i] with
instrs[i+1].'''
 def swapArgs(f): return lambda a,b: f(b, a)
 return reduce(swapArgs(compose), map(instrumentReplacer, instrs[:-1], instrs[1:]),
identity)

def transposer(semitonesUp):
 '''semitonesUp:int; returns f:Music->Music that transposes music upward by
semitonesUp.'''
 return lambda music: transpose(music, semitonesUp)

octave_higher = transposer(OCTAVE)

octave_lower = transposer(-OCTAVE)

//////////////////////////////////////
// Operations for multiple voices: rounds, canons, counterpoint
//////////////////////////////////////

def counterpoint(music, voices, f):
 '''music:Music, f:Music->Music, voice:int >= 1
 Returns n-voice contrapuntal composition
 in which each voice i is given by f^i(m).'''
 return series(music, together, f, voices)

def canon(music, voices, beats, f=identity):
 '''music:Music, beats:int >= 0, f:Music->Music, voices:int >= 1
 Returns n-voice canon in which each voice i is given by f^i(m),
 entering after i*beats.'''
 return counterpoint(music, voices, compose(f, delayer(beats)))

#

```

```

// Operations for repeating
// Operations for repeating
// Operations for repeating
// Operations for repeating

def repeat(music, n, f=identity):
 '''music:Music, n:int >= 1, f:Music->Music.
 Returns n repetitions of music, where the ith repetition is f^{i-1}(music)'''
 return series(music, concat, f, n)

def accompany(music1, music2):
 '''music1,music2:Music.
 Requires music1 or music2 to run forever, or one's duration to be an even multiple
 of the other's duration.
 Returns a piece of music that plays music1 and music2 simultaneously,
 ending at the same time as well.'''
 if music1.duration() < music2.duration():
 return accompany(music2, music1)
 # so now music1.duration >= music2.duration
 elif music2.duration() == POSITIVE_INFINITY:
 return together(music1, music2)
 elif music1.duration() == POSITIVE_INFINITY:
 return together(music1, forever(music2))
 else:
 return together(music1, repeat(music2, round(m1.duration() / m2.duration())))

// Examples
// Examples
// Examples

Row Row Row Your Boat
rowYourBoat = notes("""
 C C C3/4 D/4 E |
 E3/4 D/4 E3/4 F/4 G2 |
 C'/3 C'/3 C'/3 G/3 G/3 G/3 E/3 E/3 E/3 C/3 C/3 C/3 |
 G3/4 F/4 E3/4 D/4 C2
""", PIANO)

play it and then play it again, an octave higher
rowTwice = concat(rowYourBoat, transpose(rowYourBoat, OCTAVE))

play it as a 4-voice round, each voice coming in after 4 beats
rowRound = canon(rowYourBoat, voices=4, beats=4)

same 4-voice canon, but each voice an octave higher
rowOctaves = canon(rowYourBoat, voices=4, beats=4, f=transposer(OCTAVE))

same thing, but repeated forever
rowForever = canon(forever(rowYourBoat), voices=4, beats=4, f=octave_higher);

```

```

Frere Jacques
frereJacques = notes("""
 F G A F | F G A F |
 A _B C'2 | A _B C'2 |
 C'/2 D'/2 C'/2 _B/2 A F |
 C'/2 D'/2 C'/2 _B/2 A F |
 F C F2 | F C F2
""", PIANO)

4-voice canon, come in after two measures, using four different instruments
frereRound = canon(frereJacques, voices=4, beats=frereJacques.duration()/4, f=
instrumentSequence([PIANO, TRUMPET, ACCORDION, CHOIR_AAHS]))

Pachelbel's canon
The melody line below isn't complete -- for the rest, see
http://www.musicaviva.com/ensemble/canon/music.tpl?filnavn=pachelbel-canon-3mndbc
pachelbelMelody = notes("""
 ^F'2 E'2 | D'2 ^C'2 | B2 A2 | B2 ^C'2 |
 D'2 ^C'2 | B2 A2 | G2 ^F2 | G2 E2 |
 D ^F A G | ^F D ^F E | D B, D A | G B A G |
 ^F D E ^C' | D' ^F' A' A | B G A ^F | D D' D3/2 .1/2 |
""", VIOLIN)

pachelbelCanon = canon(forever(pachelbelMelody), voices=3, beats=16)

add a bass line, which starts by itself and then accompanies the melody
pachelbelBass = notes("D,,2 A,,2 | B,,2 ^F,,2 | G,,2 D,,2 | G,,2 A,,2", CELLO);
pachelbel = concat(pachelbelBass, accompany(pachelbelCanon, pachelbelBass))

```



```
package music;
```

```
import static music.Pitch.OCTAVE;
```

```
import java.util.HashSet;
```

```
import java.util.Set;
```

```
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;
```

```
/**
```

```
 * MusicLanguage defines static methods for
 * constructing and manipulating Music expressions,
 * particularly to create
 * recursive music like rounds, canons, and fugues.
 */
```

```
public class MusicLanguage {
```

```
 // Prevent instantiation
```

```
 private MusicLanguage() {}
```

```
 //////////////////////////////////////
```

```
 // Factory methods
```

```
 //////////////////////////////////////
```

```
/**
```

```
 * Make Music from a string using a variant of abc notation
 * (see http://www.walshaw.plus.com/abc/examples/).
 * The notation consists of whitespace-delimited symbols representing either
 * notes or rests. The vertical bar | may be used as a delimiter
 * for measures; make() treats it as a space.
```

```
 * Grammar:
```

```
 * notes ::= symbol*
```

```
 * symbol ::= . duration for a rest
```

```
 * pitch duration for a note
```

```
 * pitch ::= accidental letter octave*
```

```
 * accidental ::= empty string for natural,
```

```
 * _ for flat,
```

```
 * ^ for sharp
```

```
 * letter ::= one of A-G
```

```
 * octave ::= ' to raise one octave
```

```
 * , to lower one octave
```

```
 * duration ::= empty string for one-beat duration,
```

```
 * /n for 1/n beat,
```

```
 * n for n-beat duration,
```

```
 * n/m for n/m-beat duration
```

```
 * Examples (assuming 4/4 common time, i.e. 4 beats per measure):
```

```
 * C quarter note, middle C
```

```
 * A'2 half note, high A
```

```
 * _D/2 eighth note, middle D flat
```

```
 *
```

```
 * @param notes string of notes and rests in simplified abc notation given above
```

```
 * @param instr instrument to play the notes with
```

```
 */
```

Main file  
- Lots of supporting  
file not printing

```

public static Music notes(String notes, Instrument instr) {
 Music m = new Rest(0);
 for (String sym : notes.split("[\\s|]+")) {
 if (!sym.isEmpty()) {
 m = concat(m, parseSymbol(sym, instr));
 }
 }
 return m;
}

// Parse a symbol into a Note or a Rest.
private static Music parseSymbol(String symbol, Instrument instr) {
 Matcher m = Pattern.compile("([^/0-9]*)([0-9]+)?(/[0-9]+)?").matcher(symbol);
 if (!m.matches()) throw new IllegalArgumentException("couldn't understand " + symbol);

 String pitchSymbol = m.group(1);

 double duration = 1.0;
 if (m.group(2) != null) duration *= Integer.valueOf(m.group(2));
 if (m.group(3) != null) duration /= Integer.valueOf(m.group(3).substring(1));

 if (pitchSymbol.equals(".")) return new Rest(duration);
 else return new Note(duration, parsePitch(pitchSymbol), instr);
}

// Parse a symbol into a Pitch.
private static Pitch parsePitch(String symbol) {
 if (symbol.endsWith("'")) return parsePitch(symbol.substring(0, symbol.length()-1)).
transpose(OCTAVE);
 else if (symbol.endsWith(",")) return parsePitch(symbol.substring(0, symbol.length
()-1)).transpose(-OCTAVE);
 else if (symbol.startsWith("^")) return parsePitch(symbol.substring(1)).transpose(1);
 else if (symbol.startsWith("_")) return parsePitch(symbol.substring(1)).transpose(-1);
 else if (symbol.length() != 1) throw new IllegalArgumentException("can't understand
" + symbol);
 else return new Pitch(symbol.charAt(0));
}

/**
 * @param duration length of rest, must be >= 0
 * @return rest of duration beats
 */
public static Music rest(double duration) {
 return new Rest(duration);
}

```

```

////////////////////////////////////
// Producers

```

```

////////////////////////////////////
/**
 * @param m1 first piece of music
 * @param m2 second piece of music
 * @return m1 followed by m2
 */
public static Music concat(Music m1, Music m2) {
 return new Concat(m1, m2);
}

/**
 * @param m1 first piece of music
 * @param m2 second piece of music
 * @return m1 played at the same time as m2
 */
public static Music together(Music m1, Music m2) {
 return new Together(m1, m2);
}

/**
 * @param m music to loop forever
 * @return music that repeatedly plays m in an endless loop
 */
public static Music forever(Music m) {
 return new Forever(m);
}

////////////////////////////////////
// More operations on Music
////////////////////////////////////

/**
 * param m piece of music
 * @returns set of instruments used by m
 */
public static Set<Instrument> instrumentsUsed(Music m) {
 final Set<Instrument> instruments = new HashSet<Instrument>();
 m.accept(new VoidVisitor () {
 public Void on(Note m) {
 instruments.add(m.instrument());
 return null;
 }
 });
 return instruments;
}

/**
 * Pause before playing some music.
 * @param m piece of music

```



```

* @param beats length of delay in beats, must be >= 0
* @returns rest of duration beats followed by m
*/
public static Music delay(Music m, double beats) {
 return concat(new Rest(beats), m);
}

/**
* Speed up or slow down a piece of music by a factor of speedup.
* For example, changeTempo(m,2) returns music that plays twice as fast as m.
* @param m piece of music
* @param speedup factor to increase speed of music, must be > 0
* @returns m' such that m'.duration() = m.duration()/speedup
*/
public static Music changeTempo(Music m, final double speedup) {
 return m.accept(new IdentityVisitor() {
 @Override
 public Music on(Note m) {
 return new Note(m.duration()/speedup, m.pitch(), m.instrument());
 }
 @Override
 public Music on(Rest m) {
 return new Rest(m.duration()/speedup);
 }
 });
}

/**
* Change all the notes in a piece of music to use a single instrument.
* For example, changeInstrument(m,PIANO) returns music that is played
* entirely by the piano.
* @requires m, instr != null
* @returns m' such that for all notes n in m', n.instrument() == instr,
* but otherwise m' is identical to m
*/
public static Music changeInstrument(Music m, final Instrument instr) {
 return m.accept(new IdentityVisitor() {
 @Override
 public Music on(Note m) {
 return new Note(m.duration(), m.pitch(), instr);
 }
 });
}

/**
* Replace all notes that use oldInstr with notes that use newInstr instead.
* @requires m, oldInstr, newInstr != null
* @returns m' such that for all notes n in m such that n.instrument() == newInstr,
* the corresponding note n' in m' has n'.instrument() == newInstr.
* Otherwise m' is identical to m
*/
public static Music replaceInstrument(Music m, final Instrument oldInstr, final

```

```

Instrument newInstr) {
 return m.accept(new IdentityVisitor() {
 @Override
 public Music on(Note m) {
 if (m.instrument().equals(oldInstr)) return new Note(m.duration(), m.pitch
()), newInstr);
 else return m;
 }
 });
}

/**
 * Transpose all notes upward or downward in pitch.
 * @requires m != null
 * @returns m' such that for all notes n in m, the corresponding note n' in m'
 * has n'.pitch() == n.pitch().transpose(semitonesUp). Otherwise m' is identical
 * to m.
 */
public static Music transpose(Music m, final int semitonesUp) {
 return m.accept(new IdentityVisitor() {
 @Override
 public Music on(Note m) {
 return new Note(m.duration(), m.pitch().transpose(semitonesUp), m.instrument
());
 }
 });
}
}

```

6.005

11/16

P-set 7

Tune up

Need to lock model  
try change listener

↳ So model implements too?  
how can model tell?

↳ implement something on model?

So add change listener (change listener) {  
    changeListener.stateChanged();

}

But when do we call this to try?

F - this - I don't get it

---

Tune up some descriptions



Quiz 2 Review

- Last one!
  - Quiz review via Jeopardy-style game
  - Everything in class fair game
    - focus on last quiz stuff since
    - a lot on concurrency
    - MVC
    - GUI
    - Functional Programming
    - No Python syntax
- 

Power map question

$2^5 \quad 3^2 \quad 4^2$   
↳ 32   9   16

Concurrency 2 models

message passing  
memory sharing

②

Threads - share memory

Processes - don't share memory - virtual machine

---

Adding a precondition weakens the spec

↳ puts off on user

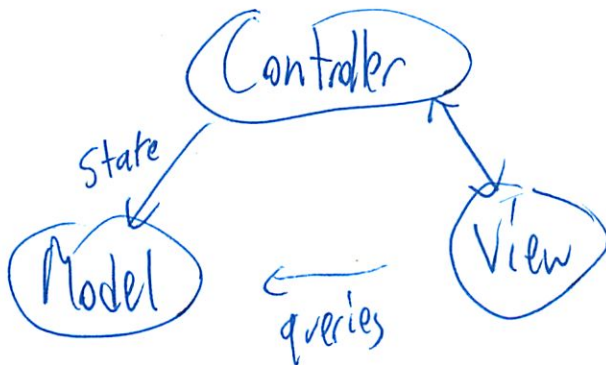
Specifying an exception is thrown strengthens spec

↳ More robust against change

---

MVC diagram

- arrows can depend on implementation



---

Method Signatures

Boolean contains element

etc - see section

3

For TreeMap - needs to implement comparable

Map/Reduce/Filter

$\text{reduce}(+, "", \text{reduce}(\text{append}, \text{empty}, \text{map}(\text{split}, \text{list})))$

↑ is a way to do this that is more  
"map reduce"

↳ to make parallelizable

Correct ans:

$\text{reduce}(\text{map}(\text{lambd}a\ x : \text{reduce}(+, x, \text{split}()), \text{list}))$

vs list try  
Concat

reduce (map(split, list))



4

## MVC

Controller - user input

View - output

Model - internal representation

---

## Deadlock or Slow

A, E

LC won't since consistent order

LA won't scale - blocking entire system

↳ can't change any contents

## Not maintaining RT sometimes

B, D  
only lock on 1

↑ problem b/w the from and to locks

Since b/w the 2 ~~read~~ "from" locks, rep is not wrong  
- but program can halt there

5

## Bias

- if take some ~~lot~~ votes lot
- they have higher %
- so will bias when take 1 %

C always uses ~~the~~ lowest first

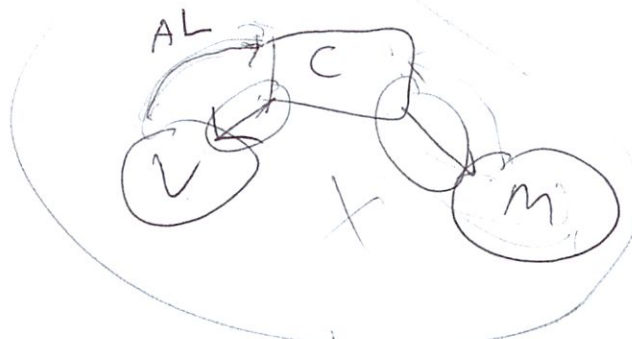
The American Idol TV program has changed its voting scheme. Instead of voting for one candidate, the callers are asked to indicate "candidate A is better than candidate B". Initially each candidate is given 10000 points and for each vote "A is better than B", 1% of points is deducted from candidate B's total and added to candidate A's total. You, as the wiz programmer for American Idol, have to implement the code that does the vote tallying. Since millions of people dial-in, to handle the peak load your code has to run on hundreds of processors simultaneously.

```
public class contestant {
 // RI: points >= 0
 int points;
 public final int myid; // unique id of each contestant
 // Assume there is a constructor that initiates all the fields correctly.
 void post(int i) { points += i; }
 int get() { return points; }
}

public class AmericanIdol extends thread {
 // RI: SUMfor all contestants points = 10000 * number_of_contestants
 static AmericanIdol AI; // the game object
 contestant con[];
 // Assume there are methods that initiates all the fields correctly and start the threads
 public void vote(contestant from, contestant to) {
 int mov;
 <BODY>
 }
}
```

For the next five questions, consider the following code for the <BODY> of the vote method.

|   |                                                                                                                                                                                                       |   |                                                                                                                                             |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------------------------------------------------------------|
| A | <pre>synchronized(AI) {     mov = from.get()/100;     from.post(-mov);     to.post(mov); }</pre>                                                                                                      | D | <pre>synchronized(from) {     mov = from.get()/100;     from.post(-mov);     to.post(mov); }</pre>                                          |
| B | <pre>synchronized(from) {     mov = from.get()/100; } synchronized(from) {     from.post(-mov); } synchronized(to) {     to.post(mov); }</pre>                                                        | E | <pre>synchronized(from) {     synchronized(to) {         mov = from.get()/100;         from.post(-mov);         to.post(mov);     } }</pre> |
| C | <pre>synchronized((from.myid &gt; to.myid)?from:to) {     synchronized((from.myid &gt; to.myid)?to:from) {         mov = from.get()/100;         from.post(-mov);         to.post(mov);     } }</pre> |   |                                                                                                                                             |



Split (" " "



```

/**
 * BugLocations represents the set of line numbers containing
 * bugs in a release of the Doors operating system.
 */
public class BugLocations {

 public Set<Number> elts;

 // RI: all elements of elts are > 0.

 public BugLocations() {
 elts = new Set<Number>();
 }

 public boolean contains(Number n) {
 return elts.contains(n);
 }

 public BugLocations copy() {
 BugLocations other = new BugLocations();
 other.addAll(this);
 return other;
 }

 public void add(Number n) {
 elts.add(n);
 }

 public void addAll(BugLocations bl) {
 for (Number n : bl.elts) {
 add(n);
 }
 }
}

```

## 6.005 Code Review

11/20

Oh I see call make Guess from a new thread in the GUI where can call update table

I had model represent all the data  
They use the GUI

No one else GUI tested

One did guess listener

Ohh can call do In Background ( ) in GUI

Everyone is tracking game state in GUI

All avoided my problem

Last quiz

Regular Expressions + Grammars

State Machines + Testing

Specifications

Abstract data types

Code review

Recursive data types

Visitor vs Interpreter

Reviewing last review

Specs

- think of audience - people who call
- precondition
- not about implementation
- modifies
- returns
- params



(2)

Checked exception - Things you think could happen  
throw() catch() or throws  
file locked,  
no permissions  
network down

Unchecked exception - errors don't think

Visitor

Interpreter - column

Visitor - row

|        | Operations |        |
|--------|------------|--------|
|        | var        | and or |
| eval   |            |        |
| hasNot |            |        |
| same   |            |        |

Recursive data type - calls itself

Ques Expr = Var(s: string) + And(l: Expr, r: Expr)

Grammar  $\rightarrow$  Expr ::= L Paren Expr R Paren | Expr Expr

Rep = <sup>internal</sup> representation

Check Rep - function to make sure rep has not changed

③

## Grammar

Non terminal  $::$  = expression of terminals +  
non-terminals and operators

### Sequence

$$A :: B C$$

iteration  $A :: B^*$

choice  $A :: B | C$

So  $*$  = 0 or more

$$^+ = 0 \text{ or } 1$$

$$+ = 1 \text{ or more } (B^+ = B B^*)$$

$$[a b c] = a | b | c$$

$$[^+ b] = \text{everything except } b$$

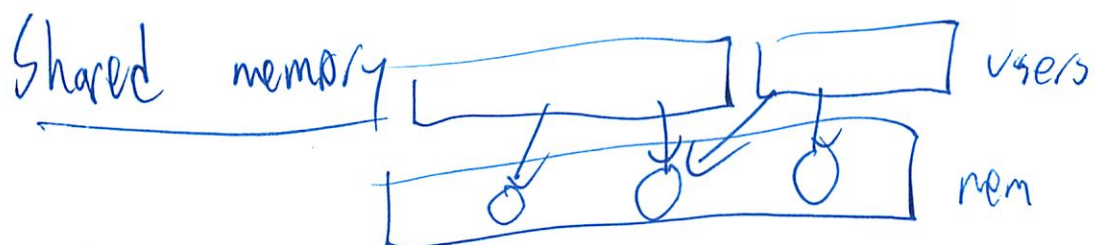
Can combine down into a regular expression

4

New stuff

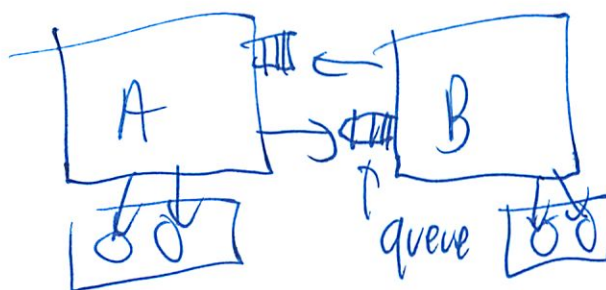
ABC project

L/O Concurrency



- if immutable - would not care

message passing



like web server / browser

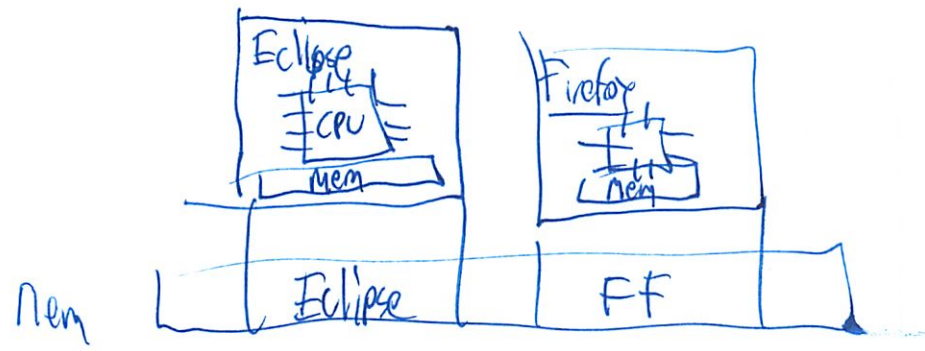


5

# Processes

Like a virtual computer

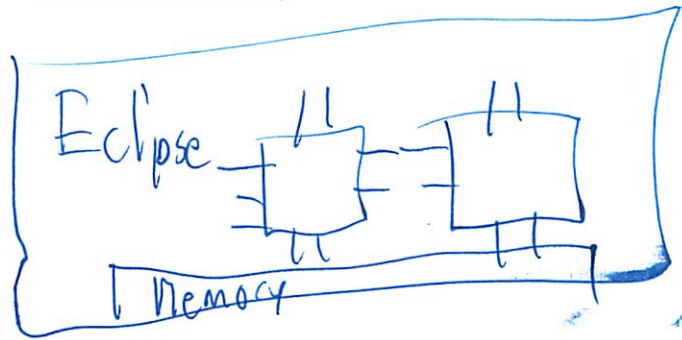
- simulation of being alone
- if it looked at time it would realize not the case



Thread Like a virtual processor inside program



Memory shared inside process



each has own stack/locus of control

⑥ Problem w/ shared memory : race condition

Leash machine example

Can't depend on order stuff is done  
is not consistently one way  
When breakpoints / printing things change

---

## L11 Processes + Sockets

Message passing over network sockets or processes

Server socket - listen for connection on fixed port

When it gets connection, it allocates a new port  
port are just virtual/file addresses in a computer

can read() or write()

It will block if no data available / queue is full

Finite sized buffer inside connection

Protocol - Grammar to define how client + server  
communicate to each other

⑦

Good protocols are ready for change + extensible

- platform independent
- client could be written in diff lang
- be simple and extensible
  - like web uses GET, PUT, DELETE

---

## LR Thread Safety

4 ways to do threads safely

### 1. Confinement

- don't share anything

### 2. Immutable

- variable final
- and what it points to does not change
- ref value never changes

### 3. Thread Safe Datatype

- is in the spec sometimes
- must be no preconditions on timing
- if it throws a warning it is NOT threadsafe



## ⑧ 4. Synchronization

- talk about 'in next lecture

### Threads

↳ Construct w/ a Runnable

- like a function you can pass around
- like a lambda in py

- Can only reference (what?) if final

- Java makes copy inside runnable

- Can make runnable ~~at~~ out to new class for even less access

- local variables ~~at~~ defined inside are always confined

- Static gets dangerous too

print at local variables confined

but can point to shared object

local variables that are initially shared - ~~the~~ must be final

global variables ("static") are not auto thread confined

- Should not use ~~or~~ volatile

better info  
↓

9

- or have only 1 thread access

Threadsate - if behaves correctly b/w multiple threads w/o requiring more coordination  
pre conditions

Immutability - no benevolent mutation allowed  
ie caching stuff on 1st read  
always look the same internally  
but could have still changed slightly

Goals

Safety - nothing bad happens

Liveness - including fairness  
does something good eventually happen?  
- performance

Threads cost resources

Sending data b/w threads = expensive

Make thread safety argument

- list threads
- how is each object thread safe?
  - one of our 4 things

(10)

## L13 Synchronization

"lock" stuff

if another thread has lock program will block (wait)  
till it gets the lock

locks automatically available on every object



only thread holding lock can touch object

`synchronized(obj) {`  
    code

`}`

Or as keyword `synchronized` (does on this)

Will can't # syncs added - so having multiple  
is not a problem

↳ passes through w/o problem

Can't easily sync constructor

↳ don't share references till built!



11

Monitor pattern - lock <sup>entire</sup> object w/ this  
- ~~the~~ keyword

Sync both mutators and view

if multiple shared values - must guard w/ same lock  
↳ invariant must be value before releasing lock

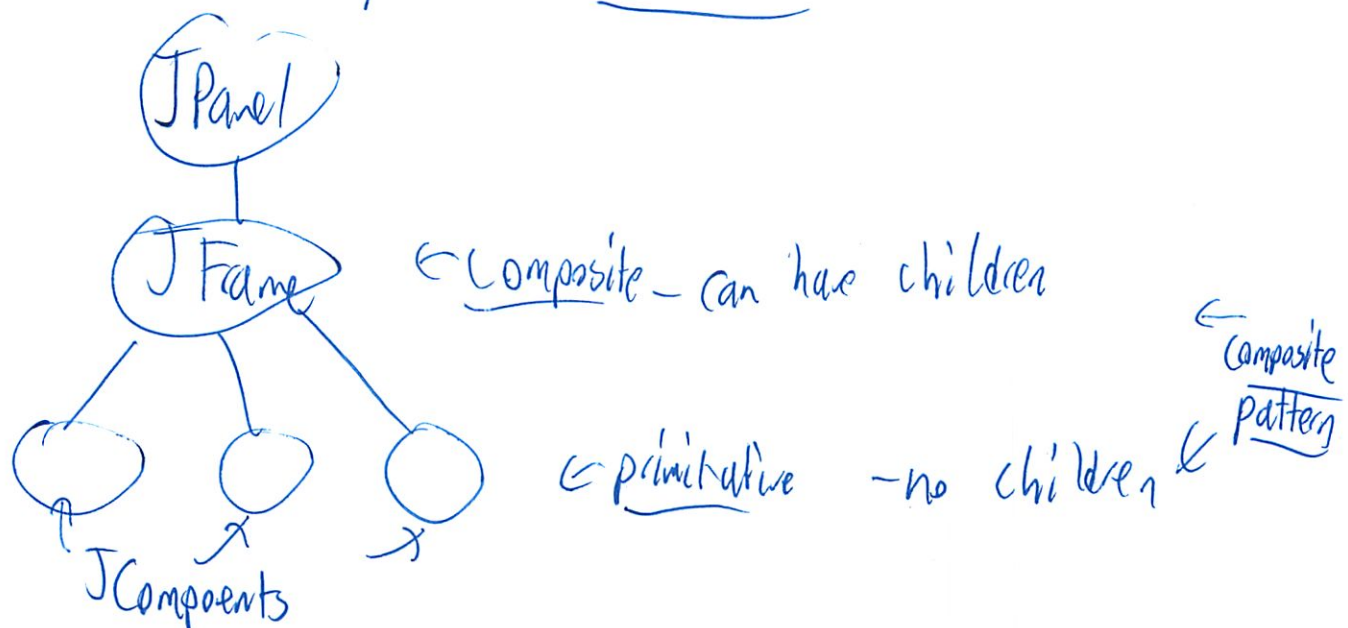
Don't sync everything

↳ program will slow to a crawl!

---

## L14 GUIs

Structured w/ a View Tree



(12)

different layout ~~managers~~ managers

↳ like Graph Manager (most common)

input

↳ add Listeners

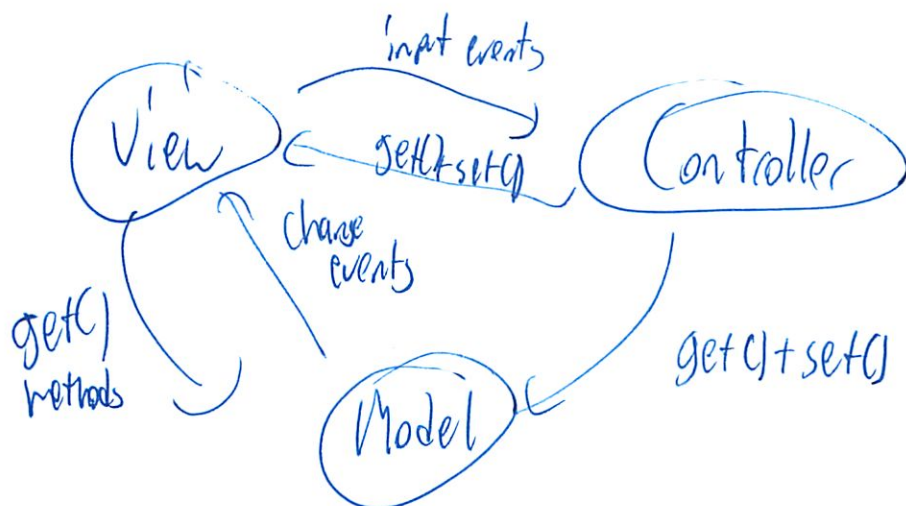
regions send commands to proper areas

Listeners are interfaces — you implement  
event Happened (Event event) methods

~~Model~~

Model View Controller (MVC)

— Diff chart than I saw before



(13)

look at Jwang's ps 7

↳ has a thread to make guess

— but another thread to ~~an~~ update guess

make guess click listener &  
button

new Thread() {

makeGuess()

} update table()

}

↳ in model

makeGuess() {

new Thread() {

web stuff --

returns --

}

}



(14)

Instances of MVC at many scales in GUI

└ text field itself is MVC

— or look at whole system

Problems Listener calls observer during mutation

Or Listener calls mutator during modification

don't have 2 things listening to each other's ~~status~~ changes!

---

### L15 Map/Filter/Reduce

- evil in Java
- nice in other langs
- called functional programming
- control (if, for) code disappears
- for working on sequences
- can run Py in Java

(15)

Seq <E> 'is seq of els in type E

Java {  
List <E>  
E[]  
Iterable <E>  
Iterable <E>. iterator()

Py {  
list - mutable []  
string  
tuple - immutable ()  
Streams  
iterable

map (sqrt, [1, 4, 9, 16])

↳ [1.0, 2.0, 3.0, 4.0]

functions being 1st class values

↳ can name  
pass as argument  
return it  
put it in lists, tuples, sets, ...

(16)

## Lambda expression

$[t.join \text{ for } t \text{ in threads}]$

Syntactic sugar for map

$\text{map}(2^k, [1, 2, 3, 4])$

```
def powerOfTwo(k)
 return 2**k
map(powerOfTwo,
 [1, 2, 3, 4])
```

$\text{map}(\text{lambda } k: 2^{**}k, [1, 2, 3, 4])$

Same thing

## Filter

$\text{filter}(p, [e_0, \dots, e_{n-1}]) =$   
 $[e_i \text{ s.t. } p(e_i)]$

aka

$[t.join() \text{ for } t \text{ in threads, if } \underbrace{t.isRunning()}_{\text{filter}}]$



(17)

ie

$\text{map}(\text{lambda } x: x \% 2 == 1, [1, 2, 3, 4, 5])$

$\hookrightarrow [1, 3, 5]$   $\uparrow$  so only returns if true

## Reduce

— most powerful

— can implement all the other

$\text{reduce}(\text{add}, [1, 2, 3], 0)$

$\uparrow$  base case

$$(((0 + 1) + 2) + 3) = 6$$

(It's a diff way of thinking I am not good at yet...)

Python folds from 'left'  $\rightarrow$

Order matters in some cases

Powerful since can split b/w machines

In Java, functional object or functor

$\hookrightarrow$  represents a lot class object

(18)

Also two more

Compose so have  $f_n \langle A, B \rangle$  and  $f_n \langle B, C \rangle$   
get  $f_n \langle A, C \rangle$

chain list of  $f_n$ s  $A \rightarrow A$  to compose

Very hard to  $\downarrow$  in Java

(do we have to know the Java implementation?)

seems like we usually expected to  
know both concepts + Java  
but sometimes Java very unclear)

---

## L16 Little Languages

We kinda used 1st class ~~func~~ functions before w/  
our ADT<sup>tree</sup> from the parser

also ~~with~~ the Visitor object

build datatype of item  
↳ song in our case

19

It actually copies behind the scenes

It uses interpreter + visitor pattern

Uses composite data type

like delay : Music  $\times$  double  $\rightarrow$  Music  
what is writing like this called

So can say `play( together( conYarBart, delay(transpose(  
conYarBart, octave), 4, )))`

Can you do  
forever Music  $\rightarrow$  Music ?



PLAP

1. Primitive
2. Combinations
3. Abstractions
4. Patterns



20

Online ppt

Constructors : create objects

Producers : return immutable object

Mutators : change state

Observers : report about state

Name: \_\_\_\_\_

Athena User Name: \_\_\_\_\_

Massachusetts Institute of Technology  
6.005: Elements of Software Construction  
Spring 2011

Quiz 2  
Wednesday, 13 April 2011

Practice 11/20

this will be 50 min



Instructions

This quiz is 80 minutes long. It contains 21 questions in 14 pages (including this page) for a total of 100 points.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name on the top of every page. Please write neatly. No credit will be given if we cannot read what you write. Good luck!

| Question Name    | Maximum Points | Question Numbers | Points Given |
|------------------|----------------|------------------|--------------|
| True/False       | 30             | 1-10             |              |
| Event-based Prog | 12             | 11-12            |              |
| MVC Pattern      | 10             | 13               |              |
| Generics         | 14             | 14-15            |              |
| Concurrency      | 14             | 16-17            |              |
| Equality         | 10             | 18-19            |              |
| Performance Eng. | 10             | 20-21            |              |
| <b>Total</b>     | <b>100</b>     | <b>1-21</b>      |              |

Name: \_\_\_\_\_

## True/False [30 points, 3 points each]

Circle the correct answer:

1. ☒ T / ☐ F  
The code below outputs the following: "Child Called"

```
public class InheritanceTest {
 public static void main(String[] args) {
 Parent c = new Child();
 }
 class Parent {
 public Parent() {
 System.out.println("Parent called");
 }
 }
 class Child extends Parent {
 public Child() {
 System.out.println("Child called");
 }
 }
}
```

*Edna*  
*Try in*  
*So does*  
*Parent Called*  
*Child Called*  
*↳ runs superclass constructor*

*I would fail*  
*- type not match?*

2. ☐ T / ☒ F  
The value of the variable b is true.

```
Integer i = 100;
Integer j = 100;
boolean b = (i == j);
```

*editor since Java does*  
*caching*

*no can be less specific (object)*  
*T is most unspecific*

*diff objects needs equals()*

3. ☒ T / ☐ F  
The value of the variable b is true.

```
int i = 1000;
int j = 1000;
boolean b = (i == j);
```

*here base object*

4. ☒ T / ☐ F  
Event-based programming can be implemented without multi-threading.

*threads behind the scenes*

5. ☒ T / ☐ F  
A protected variable can only be accessed by any class in the same package.

*forget what this means*

*why don't say why?*



Name: \_\_\_\_\_

6. ☒ T / ☐ F  
The following code is thread safe:

```
class A {
 private int count;
 A() {
 this.count = 0;
 }

 synchronized public void incrementCount() {
 count = count + 1;
 }
}
```

Constructor isn't  
- don't pass refs too early  
↳ but only 1 action  
so ok

7. ☒ T / ☐ F  
Always use inheritance in order to reuse code that is implemented in another class.

always bad word

8. ☒ T / ☐ F  
MVC pattern, as described in lecture, decouples the Controller from the Model.

ideally - but not really here

9. ☒ T / ☐ F  
Always make a copy of listeners before you iterate over them.

10. ☒ T / ☐ F  
The code segment below is legal java that will compile and run.

```
...
List<?> list1 = new ArrayList<Integer>();
list1.add(new Integer(6));
...
```

why would ya iterate over lister

no

Name: \_\_\_\_\_

## Event-based Programming [10 points]

Ben Bitdiddle decides to implement his own file system as shown below.

```
class Filesystem {
 private Map<File, List<File>> cache;

 public List<File> getContents(File folder) {
 // check for folder in cache, otherwise
 // read it from disk and update cache
 }

 public void deleteContents(File folder) {
 for (File f : getContents(folder)) {
 f.delete();
 // notify listeners that f was deleted
 fireChangeEvent(f, REMOVED);
 }
 cache.remove(folder); // update cache
 }
}
```

*Handwritten notes:*  
- what is this? - the contents?  
- no list of files  
- does this read from disc  
- this is not key of cache

11. His friend Alyssa P. Hacker finds a potential problem with the above code. Can you identify it? [4 points]

*Handwritten answer:*  
Um folder is not a key on the cache?  
Not synced: - event listener fires last  
✓ See topic to use the test to take the test

12. Ben wants you to modify the method `deleteContents()` so the problem goes away. Your code should compile (with appropriate packages). [8 points]

```
public void deleteContents(File folder) {
```

*Handwritten:* Synchronized (cache)

*Handwritten:* etc Not the point of this lecture

*Handwritten:* } fire event at end of everything

Name: \_\_\_\_\_

}



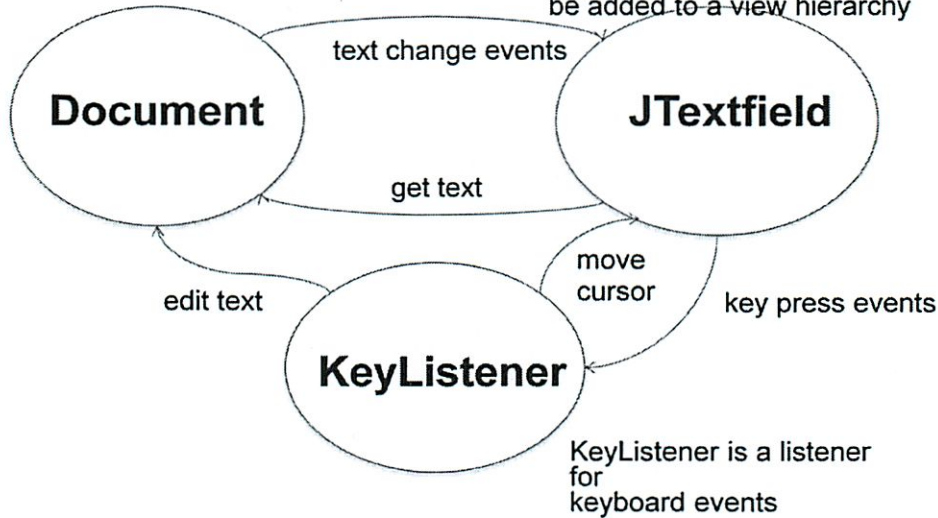
Name: \_\_\_\_\_

## Model View Controller (MVC) Pattern [10 points]

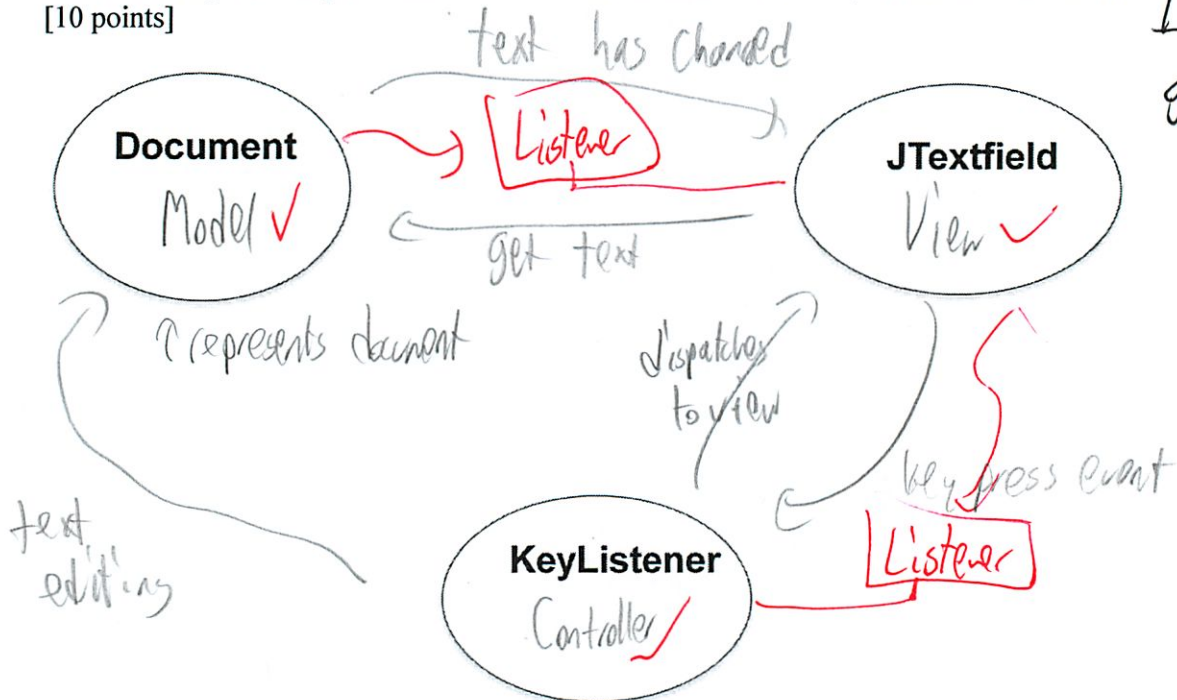
Ben is excited about the application of an MVC pattern to a textbox.

Document represents a mutable string of characters

TextField is a JComponent that can be added to a view hierarchy



13. Help Ben by specifying which circles correspond to the Model, View and Controller below by writing them into the ellipses. Show how a listener interface can be used to decouple the dependencies shown above between the Model, View and Controller. [10 points]



Name: \_\_\_\_\_

## Generics [10 points]

Ben Bitdiddle has been charged with the task of creating a data structure for a hotel in a game where the room's door type is chosen by the guests.

```
public class Room<E>
{
```

```
 private E door;
```

```
 public void setDoor(E door) {
 this.door = door;
 }
```

```
 public E getDoor(E door) {
 return E;
 }
}
```

Also he was asked to write a hotel worker class that has a static function that checks if an EnchantedDoor is protected by a firewall to avoid getting burnt.

```
class Door {...}
```

```
class EnchantedDoor extends Door{...}
```

```
class HotelWorker {
```

```
 public static boolean isFirewallProtected(EnchantedDoor door) {
 ... // implementation is not relevant here
 }
```

```
}
```

14. He was then asked to write a function that can only insert an EnchantedDoor or any subclass of EnchantedDoor that is protected by a firewall. Fill in the blanks for this function: [9 Points]

```
public static void setRoomFirewallDoor
 (Door door, Room<E> room) {
```

```
 if (isFirewallProtected(door)) {
 room.setDoor(door);
 }
```

```
}
```

Name: \_\_\_\_\_

15. In addition Ben was asked to write a method in class `HotelWorker` that checks if a room has a firewall door. He decided to write pretty code and use overloading. Will the following code work? If not, what is wrong with it? [5 points]

```
public static boolean isRoomFirewallProtected(Room<EnchantedDoor> room)
{
 EnchantedDoor door = room.getDoor();
 return isFirewallProtected(door);
}
```

```
public static boolean isRoomFirewallProtected(Room<HarryPotterDoor>
room) {
 HarryPotterDoor door = room.getDoor();
 // assume there exists isFireWallProtected for HarryPotterdoor
 return isFirewallProtected(door);
}
```

not  
No - if another type of door  
Not ready for change!

Java will not compile since compiler  
will remove the generics and both methods  
have same sig

I don't think we learned this...



Name: \_\_\_\_\_

### Concurrency [10 points]

16. Alyssa P. Hacker writes a concurrent program that contains the following code.

```
private static int a = 5;

private int square(int x) {
 return x * x;
}
```

```
public void doStuff() {
 int tmp = a;
 tmp = square(tmp);
 a = tmp;
}
```

She then challenges you to answer the following question. Suppose two threads both call the method `doStuff`. Which of the following values might the variable `a` contain after both have completed execution? (Circle all possible values) [5 points]

a. 0

b. 5

☒ c. 25

d. 125

☒ e. 625

↳ both save ✓  
↳ one after other, normal ✓

Name: \_\_\_\_\_

17. Alyssa decides to teach her friend, Ben, how to design a simple database system in Java. She hacks up the following example.

```
class AlyssaDB {
 private AlyssaRow[] alyssaRows;

 public AlyssaDB(int numRows) {
 if (numRows < 0 || numRows > MAXROWS) {
 throw new BadInputException();
 }
 alyssaRows = new AlyssaRow[numRows];
 for (int j = 0; j < numRows; ++j) {
 alyssaRows[j] = AlyssaRow.createAlyssaRow(j);
 }
 }

 /**
 * swap data from rows with indices index1, index2
 *
 * @param index1 index of first row
 * @param index2 index of second row
 */
 public void swapAlyssaRowData(int index1, int index2) {
 if (index1 >= alyssaRows.length ||
 index2 >= alyssaRows.length ||
 index1 < 0 || index2 < 0)
 throw new BadInputException();
 synchronized (alyssaRows[index1]) {
 synchronized (alyssaRows[index2]) {
 int x = alyssaRows[index1].getAlyssaData();
 int y = alyssaRows[index2].getAlyssaData();
 alyssaRows[index1].setAlyssaData(y);
 alyssaRows[index2].setAlyssaData(x);
 }
 }
 }
}
```

I smell deadlock

Suppose an instance of `AlyssaDB` is accessed by multiple threads. Are there any major problems that could arise? (Is the code thread-safe?) Describe a change to the code that would be necessary to ensure thread-safety, such that it would behave as originally intended [9 points] (write your answer in the next page)

well deadlock ✓ - always pick smallest row  
to lock let 10  
alyssaRows[index1] > index2 ? index1 : index2  
- may not be fair  
I need to remember this  
to not

Name: \_\_\_\_\_



Name: \_\_\_\_\_

## Equality [10 points]

After four years at MIT, Alyssa P. Hacker wants to believe that the MIT students are just a subset of people who happen to have a random number associated.

Below is her implementation of Person and its subclass MIT Student:

```
public class Person {
 private String name;
 ...
 public boolean equals(Object obj) {
 if (!(obj instanceof Person))
 return false;
 Person a = (Person) obj;
 return this.name.equals(a.name);
 }
}

public class MITStudent extends Person {
 private int course;
 ...
 public boolean equals(Object obj) {
 if (!(obj instanceof MITStudent))
 return false;
 MITStudent a = (MITStudent) obj;
 return this.name.equals(a.name) &&
 this.course == a.course;
 }
}
```

Confident that a correct implementation of equals will reassure her that MIT students and people are the same at heart, Alyssa hits run on the following main method:

```
public static void main(String[] args) {
 Person person = new Person("Alyssa");
 Student student = new Student("Alyssa", 6);
 System.out.println(person.equals(student));
 System.out.println(student.equals(person));
}
```

Name: \_\_\_\_\_

18. What boolean values are printed and why? [5 points]

- a. true, true: correct implementation
- ☒ b. true, false: violates symmetry ✓
- c. false, true: violates symmetry
- d. false, false: person is not course 6
- e. the boolean values vary due to the non-deterministic nature of this equals function

In order to more fully explore the relationship between MITStudent and Person, Alyssa makes the following change to the equals method of MITStudent:

```
public boolean equals(Object obj) {
 if (!(obj instanceof Student))
 return super.equals(obj);
 Student a = (Student) obj;
 return this.name.equals(a.name) && this.course == a.course;
}
```

*MIT student?*  
*that*  
*what changed?*

19. What boolean values are printed by the main method above and why? [5 points]

- ☒ a. true, true: correct implementation ✓
  - b. true, false: violates symmetry
  - c. false, true: violates symmetry
  - d. false, false: person is not course 6
  - e. the boolean values vary due to the non-deterministic nature of the equals function
- unless some trick*

*but violates transitivity so all ans taken*

*Transitivity → if  $A \rightarrow B$  and  $B \rightarrow C$   
then  $A \rightarrow C$*

*In Java:*

*It looks like ~~everything~~ <sup>equals</sup> should be transitive*

Name: \_\_\_\_\_

## Performance Engineering [8 points]

*↓ may end up being slower*

20. Ben wants to create a superfast version of his class project. He is experimenting with three statements that allocate the variables T1, T2 and T3. Each provides a different way to initialize a large integer array to hold a two dimensional data set.

```
static final int LARGEARRAY = 100000;
static final int SMALLARRAY = 400;

int T1[][] = new int[LARGEARRAY][SMALLARRAY];
int T2[][] = new int[SMALLARRAY][LARGEARRAY];
int T3[] = new int[LARGEARRAY*SMALLARRAY];
```

Then Ben measured the time for each of the above three statements. Which one of these choices should best describe the time taken by each of the statements? [4 points]

- a. T1=124 ms, T2=123 ms, T3=126 ms
- b. T1=310 ms, T2=308 ms, T3=95 ms
- ☒ c. T1=830 ms, T2=231 ms, T3=82 ms
- d. T1=241 ms, T2=820 ms, T3=79 ms
- e. T1=0 ms, T2=0 ms, T3=0 ms

*2D arrays are arrays of arrays*

*T3 = 1 allocation*

*T2 = small array + 1 allocation*

*T1 = large array + 1 allocation*



Name: \_\_\_\_\_

21. When the variable T3 is used, which is allocating a 2D array as a flattened 1D data structure, Ben tried out two alternative methods for initializing the array.

```
initA() {
 for(int i =0; i < SMALLARRAY; i++)
 for(int j =0; j < LARGEARRAY; j++)
 T3[i*LARGEARRAY+j] = 0;
}
```

```
initB() {
 for(int i =0; i < SMALLARRAY; i++)
 for(int j =0; j < LARGEARRAY; j++)
 T3[i+j*SMALLARRAY] = 0;
}
```

Again Ben measures the time it took for each of the initialization routines. Which one of the following choices should best describe the time taken by each of the methods? [4 points]

- a. initA=420 ms, initB=423 ms, nearly identical as both code segments do the same initialization
- ☒ b. initA=530 ms, initB=1988 ms as initA has better cache behavior due to unit stride access.
- c. initA=1767 ms, initB=423 ms as initB has better cache behavior due to unit stride access.
- d. initA=1891 ms, initB=562 ms as initA initializes more data than initB.

Name: \_\_\_\_\_

Athena User Name: \_\_\_\_\_

Massachusetts Institute of Technology  
6.005: Elements of Software Construction  
Spring 2011  
Quiz 2  
Wednesday, 13 April 2011

Instructions

This quiz is 80 minutes long. It contains 21 questions in 14 pages (including this page) for a total of 100 points.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name on the top of every page. Please write neatly. No credit will be given if we cannot read what you write. Good luck!

| Question Name    | Maximum Points | Question Numbers | Points Given |
|------------------|----------------|------------------|--------------|
| True/False       | 30             | 1-10             |              |
| Event-based Prog | 12             | 11-12            |              |
| MVC Pattern      | 10             | 13               |              |
| Generics         | 14             | 14-15            |              |
| Concurrency      | 14             | 16-17            |              |
| Equality         | 10             | 18-19            |              |
| Performance Eng. | 10             | 20-21            |              |
| <b>Total</b>     | <b>100</b>     | <b>1-21</b>      |              |

Name: \_\_\_\_\_

**True/False [30 points, 3 points each]**

**Circle the correct answer:**

1. T / F

The code below outputs the following: "Child Called"

```
public class InheritanceTest {
 public static void main(String[] args) {
 Parent c = new Child();
 }
}
class Parent {
 public Parent() {
 System.out.println("Parent called");
 }
}
class Child extends Parent {
 public Child() {
 System.out.println("Child called");
 }
}
```

2. T / F (java does caching, we need to ask srini whether to accept true and false)  
The value of the variable b is true.

```
Integer i = 100;
Integer j = 100;
boolean b = (i == j);
```

3. T / F

The value of the variable b is true.

```
int i = 1000;
int j = 1000;
boolean b = (i == j);
```

4. T / F

Event-based programming can be implemented without multi-threading.

5. T / F

A protected variable can only be accessed by any class in the same package.

Name: \_\_\_\_\_

6. T / F  
The following code is thread safe:

```
class A {
 private int count;
 A() {
 this.count = 0;
 }

 synchronized public void incrementCount() {
 count = count + 1;
 }
}
```

7. T / F  
Always use inheritance in order to reuse code that is implemented in another class.
8. T / F  
MVC pattern, as described in lecture, decouples the Controller from the Model.
9. T / F  
Always make a copy of listeners before you iterate over them.
10. T / F  
The code segment below is legal java that will compile and run.

```
...
List<?> list1 = new ArrayList<Integer>();
list1.add(new Integer(6));
...
```

Name: \_\_\_\_\_

### Event-based Programming [10 points]

Ben Bitdiddle decides to implement his own file system as shown below.

```
class Filesystem {
 private Map<File, List<File>> cache;

 public List<File> getContents(File folder) {
 // check for folder in cache, otherwise
 // read it from disk and update cache
 }

 public void deleteContents(File folder) {
 for (File f : getContents(folder)) {
 f.delete();
 // notify listeners that f was deleted
 fireChangeEvent(f, REMOVED);
 }
 cache.remove(folder); // update cache
 }
}
```

11. His friend Alyssa P. Hacker finds a potential problem with the above code. Can you identify it? [4 points]

Listeners are notified that the file was deleted before the end of the method, and they might try to display the files that are currently being deleted (and may be corrupted).

12. Ben wants you to modify the method `deleteContents()` so the problem goes away. Your code should compile (with appropriate packages). [8 points]

```
public void deleteContents(File folder) {

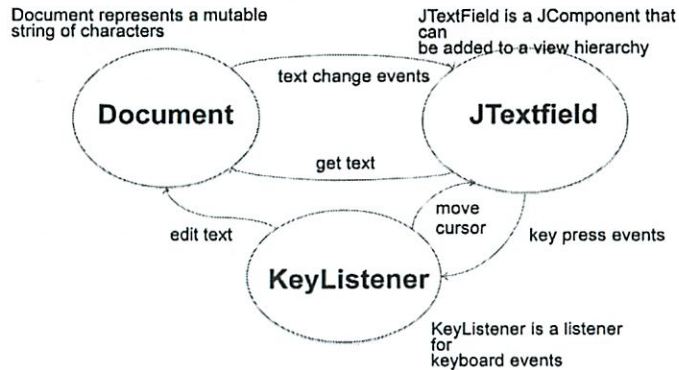
 List<File> removed = getContents(folder);
 for (File f : removed) {
 f.delete();
 }
 cache.remove(folder);
 for (File f: removed) {
 fireChangeEvent(f, REMOVED);
 }
}
```



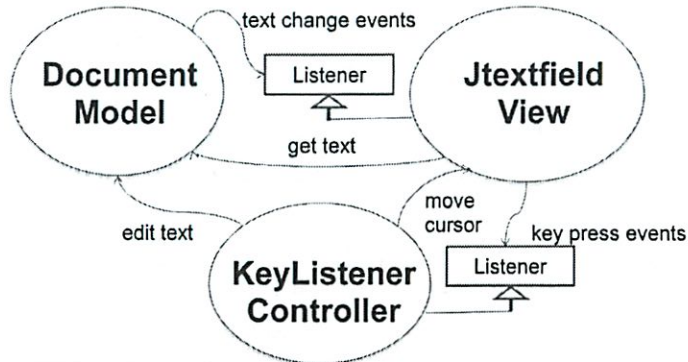
Name: \_\_\_\_\_

### Model View Controller (MVC) Pattern [10 points]

Ben is excited about the application of an MVC pattern to a textbox.



13. Help Ben by specifying which circles correspond to the Model, View and Controller below by writing them into the ellipses. Show how a listener interface can be used to decouple the dependences shown above between the Model, View and Controller. [10 points]



Name: \_\_\_\_\_

### Generics [10 points]

Ben Bitdiddle has been charged with the task of creating a data structure for a hotel in a game where the room's door type is chosen by the guests.

```

public class Room<E>
{
 private E door;

 public void setDoor(E door) {
 this.door = door;
 }

 public E getDoor() {
 return door;
 }
}

```

Also he was asked to write a hotel worker class that has a static function that checks if an EnchantedDoor is protected by a firewall to avoid getting burnt.

```

class Door {...}
class EnchantedDoor extends Door {...}

class HotelWorker {
 public static boolean isFirewallProtected(EnchantedDoor door) {
 ... // implementation is not relevant here
 }
}

```

14. He was then asked to write a function that can only insert an EnchantedDoor or any subclass of EnchantedDoor that is protected by a firewall. Fill in the blanks for this function: [9 Points]

```

public static <A extends EnchantedDoor> void setRoomFirewallDoor
 (A door, Room<? super A> room) {
 }

 if (isFirewallProtected(door)) {
 room.setDoor(door);
 }
} // correct implementation

public static ____ void setRoomFirewallDoor

```

Name: \_\_\_\_\_

```
(EnchantedDoor door, Room<EnchantedDoor> room) {
 if (isFirewallProtected(door)) {
 room.setDoor(door);
 }
} //the call setRoomFireWallDoo(new EnchantedDoor(), new
Room<Door>()); would not be allowed even though we are allowed to
insert new EnchantedDoor to Room<Door>
```

```
public static ____ void setRoomFirewallDoor
 (EnchantedDoor door, Room<? extends EnchantedDoor>
room) {
 if (isFirewallProtected(door)) {
 room.setDoor(door); //2
 }
} // compile error in line 2
```

```
public static ____ void setRoomFirewallDoor
 (EnchantedDoor door, Room<? super EnchantedDoor>
room) {
 if (isFirewallProtected(door)) {
 room.setDoor(door); //2
 }
} //the call setRoomFireWallDoo(new EnchantedDoorChild(), new
Room<EnchantedDoorChild>()); would not be allowed even though we
are allowed to insert new EnchantedDoorChild to Room<
EnchantedDoorChild>
```

15. In addition Ben was asked to write a method in class HotelWorker that checks if a room has a firewall door. He decided to write pretty code and use overloading. Will the following code work? If not, what is wrong with it? [5 points]

```
public static boolean isRoomFirewallProtected(Room<EnchantedDoor> room)
{
 EnchantedDoor door = room.getDoor();
```

Name: \_\_\_\_\_

```
 return isFirewallProtected(door);
 }
 public static boolean isRoomFirewallProtected(Room<HarryPotterDoor>
room) {
 HarryPotterDoor door = room.getDoor();
 // assume there exists isFireWallProtected for HarryPotterdoor
 return isFirewallProtected(door);
 }
```

java will not compile this code since the compiler will remove the generics and both methods will have the same signature

Name: \_\_\_\_\_

### Concurrency [10 points]

16. Alyssa P. Hacker writes a concurrent program that contains the following code.

```
private static int a = 5;

private int square(int x) {
 return x * x;
}

public void doStuff() {
 int tmp = a;
 tmp = square(tmp);
 a = tmp;
}
```

She then challenges you to answer the following question. Suppose two threads both call the method `doStuff`. Which of the following values might the variable `a` contain after both have completed execution? (Circle all possible values) [5 points]

- a. 0
- b. 5
- c. 25
- d. 125
- e. 625

Name: \_\_\_\_\_

17. Alyssa decides to teach her friend, Ben, how to design a simple database system in Java. She hacks up the following example.

```
class AlyssaDB {
 private AlyssaRow[] alyssaRows;

 public AlyssaDB(int numRows) {
 if (numRows < 0 || numRows > MAXROWS) {
 throw new BadInputException();
 }
 alyssaRows = new AlyssaRow[numRows];
 for (int j = 0; j < numRows; ++j) {
 alyssaRows[j] = AlyssaRow.createAlyssaRow(j);
 }
 }

 /**
 * swap data from rows with indices index1, index2
 *
 * @param index1 index of first row
 * @param index2 index of second row
 */
 public void swapAlyssaRowData(int index1, int index2) {
 if (index1 >= alyssaRows.length ||
 index2 >= alyssaRows.length ||
 index1 < 0 || index2 < 0)
 throw new BadInputException();
 synchronized (alyssaRows[index1]) {
 synchronized (alyssaRows[index2]) {
 int x = alyssaRows[index1].getAlyssaData();
 int y = alyssaRows[index2].getAlyssaData();
 alyssaRows[index1].setAlyssaData(y);
 alyssaRows[index2].setAlyssaData(x);
 }
 }
 }
}
```

Suppose an instance of `AlyssaDB` is accessed by multiple threads. Are there any major problems that could arise? (Is the code thread-safe?) Describe a change to the code that would be necessary to ensure thread-safety, such that it would behave as originally intended [9 points] (write your answer in the next page)



Name: \_\_\_\_\_

The code is not thread-safe. A deadlock can occur if setAlyssaData is called with inputs (x,y) and (y,x) simultaneously. Placing an order on the locks will ensure that no deadlock occurs. This can be done by changing

```
synchronized (alyssaRows[index1]) {
 synchronized (alyssaRows[index2])
```

to

```
synchronized (alyssaRows[index1>index2?index1:index2]) {
 synchronized (alyssaRows[index1>index2?index2:index1])
```

Name: \_\_\_\_\_

### Equality [10 points]

After four years at MIT, Alyssa P. Hacker wants to believe that the MIT students are just a subset of people who happen to have a random number associated.

Below is her implementation of Person and its subclass MIT Student:

```
public class Person {
 private String name;
 ...
 public boolean equals(Object obj) {
 if (!(obj instanceof Person))
 return false;
 Person a = (Person) obj;
 return this.name.equals(a.name);
 }
}

public class MITStudent extends Person {
 private int course;
 ...
 public boolean equals(Object obj) {
 if (!(obj instanceof MITStudent))
 return false;
 MITStudent a = (MITStudent) obj;
 return this.name.equals(a.name) &&
 this.course == a.course;
 }
}
```

Confident that a correct implementation of equals will reassure her that MIT students and people are the same at heart, Alyssa hits run on the following main method:

```
public static void main(String[] args) {
 Person person = new Person("Alyssa");
 Student student = new Student("Alyssa", 6);
 System.out.println(person.equals(student));
 System.out.println(student.equals(person));
}
```

Name: \_\_\_\_\_

18. What boolean values are printed and why? [5 points]

- a. true, true: correct implementation
- b. true, false: violates symmetry
- c. false, true: violates symmetry
- d. false, false: person is not course 6
- e. the boolean values vary due to the non-deterministic nature of this equals function

In order to more fully explore the relationship between MITStudent and Person, Alyssa makes the following change to the equals method of MITStudent:

```
public boolean equals(Object obj) {
 if (!(obj instanceof Student))
 return super.equals(obj);
 Student a = (Student) obj;
 return this.name.equals(a.name) && this.course == a.course;
}
```

*The intended answer was a, but the fix violates transitivity, so we accepted all answers.*

19. What boolean values are printed by the main method above and why? [5 points]

- a. true, true: correct implementation
- b. true, false: violates symmetry
- c. false, true: violates symmetry
- d. false, false: person is not course 6
- e. the boolean values vary due to the non-deterministic nature of the equals function

Name: \_\_\_\_\_

### Performance Engineering [8 points]

20. Ben wants to create a superfast version of his class project. He is experimenting with three statements that allocate the variables T1, T2 and T3. Each provides a different way to initialize a large integer array to hold a two dimensional data set.

```
static final int LARGEARRAY = 100000;
static final int SMALLARRAY = 400;

int T1[][] = new int[LARGEARRAY][SMALLARRAY];
int T2[][] = new int[SMALLARRAY][LARGEARRAY];
int T3[] = new int[LARGEARRAY*SMALLARRAY];
```

Then Ben measured the time for each of the above three statements. Which one of these choices should best describe the time taken by each of the statements? [4 points]

- a. T1=124 ms, T2=123 ms, T3=126 ms
- b. T1=310 ms, T2=308 ms, T3=95 ms
- c. T1=830 ms, T2=231 ms, T3=82 ms
- d. T1=241 ms, T2=820 ms, T3=79 ms
- e. T1=0 ms, T2=0 ms, T3=0 ms

*Two dimensional arrays are allocated as arrays-of-arrays. T3 is only one allocation, T2 is SMALLARRAY+1 and T1 is LARGEARRAY+1 allocations.*

Name: \_\_\_\_\_

21. When the variable T3 is used, which is allocating a 2D array as a flattened 1D data structure, Ben tried out two alternative methods for initializing the array.

```
initA() {
 for(int i =0; i < SMALLARRAY; i++)
 for(int j =0; j < LARGEARRAY; j++)
 T3[i*LARGEARRAY+j] = 0;
}

initB() {
 for(int i =0; i < SMALLARRAY; i++)
 for(int j =0; j < LARGEARRAY; j++)
 T3[i+j*SMALLARRAY] = 0;
}
```

Again Ben measures the time it took for each of the initialization routines. Which one of the following choices should best describe the time taken by each of the methods? [4 points]

- a. initA=420 ms, initB=423 ms, nearly identical as both code segments do the same initialization
- b. initA=530 ms, initB=1988 ms as initA has better cache behavior due to unit stride access.
- c. initA=1767 ms, initB=423 ms as initB has better cache behavior due to unit stride access.
- d. initA=1891 ms, initB=562 ms as initA initializes more data than initB.



Massachusetts Institute of Technology  
6.005: Elements of Software Construction

Fall 2011

Quiz 2

November 21, 2011

Name: Michael Plamier

Athena User Name: theplaz

Instructions

This quiz is 50 minutes long. It contains 8 pages (including this page) for a total of 100 points. The quiz is closed-book, closed-notes.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name on the top of every page. Please write neatly. No credit will be given if we cannot read what you write. Good luck!

| Question Name       | Page | Maximum Points | Points Given |
|---------------------|------|----------------|--------------|
| Design Patterns     | 2    | 20             | 20           |
| Interpreter/Visitor | 3    | 16             | 8            |
| Map/Filter/Reduce   | 4    | 12             | 12           |
| Concurrency         | 5    | 16             | 16           |
| Deadlock            | 6    | 16             | 16           |
| Thread Safety       | 7-8  | 20             | 19           |

91

Name: Plasma

### Design Patterns [20 pts]

For each of the following statements, name the design pattern that it **best** describes, from the list below.

Interpreter  
Visitor  
Event Listener  
Map/Filter/Reduce  
Client/Server  
Model/View/Controller  
Composite

You may use a design pattern more than once in your answers. If you're torn between two best answers, you can give both, but in that case you should justify both answers.

- (a) This design pattern produces tree-like data structures.

Composite

- (b) This design pattern is used for operating over sequences of elements.

Map/Filter/Reduce

- (c) This design pattern uses higher-order functions.

Map/Filter/Reduce

- (d) This design pattern is used to separate concerns in user interfaces.

Model/View/Controller

- (e) This design pattern is used for message passing over a network.

Client/Server

Name: Plasma

### Interpreter/Visitor [16 pts]

You want to write a program to perform operations on all your Foos.

A Foo can perform lots of different tricks, like bazzle and glibble.

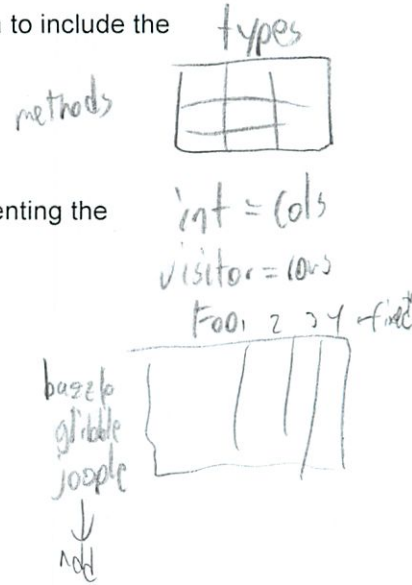
There are (and will always be) exactly 4 types of Foo, each of which does something different when they bazzle or glibble.

But every so often your Foos learn a new trick, and you must update your program to include the new operation.

For example, last week your Foos learned how to joople.

a) Would it be better to use the interpreter pattern or the visitor pattern for implementing the datatype representing a Foo?

A ~~visitor~~ pattern - because that lets us easily add more methods like kloppe() or hizzle() or snibble() or foople()



b) Assuming you designed your program according to your choice in part (a), now you want to add the joople operation. Explain what classes and methods you will change, or what classes and methods you would add, in order to support the joople operation.

in class Foo() {  
    bazzle() { ... }  
    glibble() { ... }  
    joople() { ... }  
}

add →

what we actually do

this is interpreter, not visitor

class Foo1 extends Foo() {  
    bazzle() {  
        super();  
    }  
    glibble() {  
        super();  
    }  
    joople() {  
        super();  
    }  
}

8

100% did not study!  
totally wrong



Name: Plasmeier

### Map/Filter/Reduce [12 pts]

Suppose you want to rewrite the following Python code using map, filter, and reduce:

```
def ssp(list): # sum of squares of positive numbers in list
 result = 0
 for x in list:
 if x > 0:
 result (+= sum*sum) (correction) $x \cdot x$
 return result
```

Fill in the blanks in the map/filter/reduce version below.

```
def ssp(list): # sum of squares of positive numbers in list
 return reduce(r, map(m, filter(f, list)), 0)
```

```
def f(x):
 return (x > 0)
```

filter presents one item at the time  
returns true if keep

```
def m(x):
 return $x \cdot x$
```

← square

```
def r(x, y):
 return $x + y$
```

← sum  
the 0 auto provided  
as 1st element

Name: \_\_\_\_\_

Plasma

## Concurrency [16 pts]

Read the following code:

```
public static void main() {
 Thread t1 = new Thread(new Runnable() {
 public void run() {
 System.out.print("O");
 System.out.print("Y");
 }
 });
 Thread t2 = new Thread(new Blue());
 System.out.print("R");
 t1.start();
 System.out.print("G");
 t2.start();
 System.out.print("I");
 t1.join();
 System.out.print("V");
 t2.join();
 System.out.print("K");
}
```

wait for  
thread to  
finish

R OY G B I V K  
always G O Y I B  
last O G Y

```
public static class Blue implements Runnable {
 public void run() {
 System.out.print("B");
 }
}
```

Assume that print() is threadsafe and atomic. Which of the following sequences can be printed by this code? Circle possible or impossible.

ROYGBIVK

possible

impossible

ROYBGIVK

possible

impossible

RGOYIBVK

possible

impossible

OYBRGIVK

possible

impossible

Name: \_\_\_\_\_

Plasma

### Deadlock [16 pts]

You have two threads (T0 and T1) and two locks (X and Y). Which of the following situations can lead to deadlock? If deadlock can occur, circle the method call in each thread where the thread would stop in the event of deadlock. If deadlock is impossible, circle "no deadlock."

a)

T0:

X.acquire();  
Y.acquire();  
Y.release();  
X.release();

T1:

X.acquire();  
Y.acquire();  
X.release();  
Y.release();

← always  
acquires in  
same order - ✓ good  
release doesn't matter  
(didn't take about)

no deadlock

b)

T0: (same as T0 above)

X.acquire();  
Y.acquire();  
Y.release();  
X.release();

T1:

Y.acquire();  
X.acquire();  
X.release();  
Y.release();

no deadlock

c)

T0: (same as T0 above)

X.acquire();  
Y.acquire();  
Y.release();  
X.release();

T1:

Y.acquire();  
Y.release();  
X.acquire();  
X.release();

won't deadlock?  
- yeah y acq + releases  
no problem

no deadlock



Name: \_\_\_\_\_

Plasmeier

### Thread Safety [20 pts]

Consider the following code, and answer the questions on the next page.

```
public class Widget extends Thread {
 public static List<String> strings = new ArrayList<String>();
 public int count;
 public List<Integer> numbers;
```

```
 public Widget() {
 count = 0;
 numbers = new ArrayList<Integer>();
 }
```

) constructor

```
 public void run() {
 for (int i = 0; i < 1000; ++i) {
 synchronized (this) {
 count++;
 synchronized (numbers) {
 numbers.add(i);
 }
 synchronized (Widget.strings) {
 Widget.strings.add("x");
 }
 }
 }
 }
}
```

← instance of widget  
← don't need

← global

```
public static void main(String[] args) {
 List<Widget> widgets = new ArrayList<Widget>();
 for (int i = 0; i < 1000; ++i) {
 Widget w = new Widget();
 widgets.add(w);
 w.start();
 }
```

← no add method  
← or start  
) are threads!

```
 for (Widget w : widgets) {
 synchronized (w) {
 w.count++;
 synchronized (w.numbers) {
 w.numbers.add(1000);
 }
 }
 }
```

← accessing w/o getter + setter  
but synched - so ok

```
 synchronized (Widget.strings) {
 Widget.strings.clear();
 }
```

← static reference

```
 for (Widget w : widgets) {
 w.join();
 }
```

```
}
```

Name: \_\_\_\_\_

You are reviewing a concurrency argument about this code. Circle whether you agree or disagree with each of the following statements in the concurrency argument, and add a brief (1 sentence) justification of your answer.

(a) Accesses to the `widgets` list are safe because the list is confined to the main thread.

AGREE

DISAGREE

✓ Safe for now when only 1 thread using

~~If others used in future - might be a problem~~

Is properly confined - but if change to pass ref to

(b) Accesses to the `numbers` list are safe because they acquire the list's lock.

AGREE

DISAGREE

Well the whole widget is locked as well

-1 <sup>L looks good</sup> Not just that it's locked, but it is guarded by a

(c) Assuming that the program terminates without throwing an exception, `count` for every widget is 1001 at the end of main.

AGREE

DISAGREE

✓ Called 1000 x in `main` and 1 x in `main`

~~Synced~~ Synced so no problems

common lock...

(d) Assuming that the program terminates without throwing an exception, `strings` has size 0 at the end of main.

AGREE

DISAGREE

✓ We don't know which ends first  
adding x's or clearing

END OF QUIZ

6.005  
Quiz 2  
Debrief

1/2

Sucked at Visitors

Think did Ok → Good on everything else

Did not think visitor would be on there

↳ Did not study

- Still don't really get

Should have looked at earlier/before

But would have forgot exact syntax in the month

Should have studied



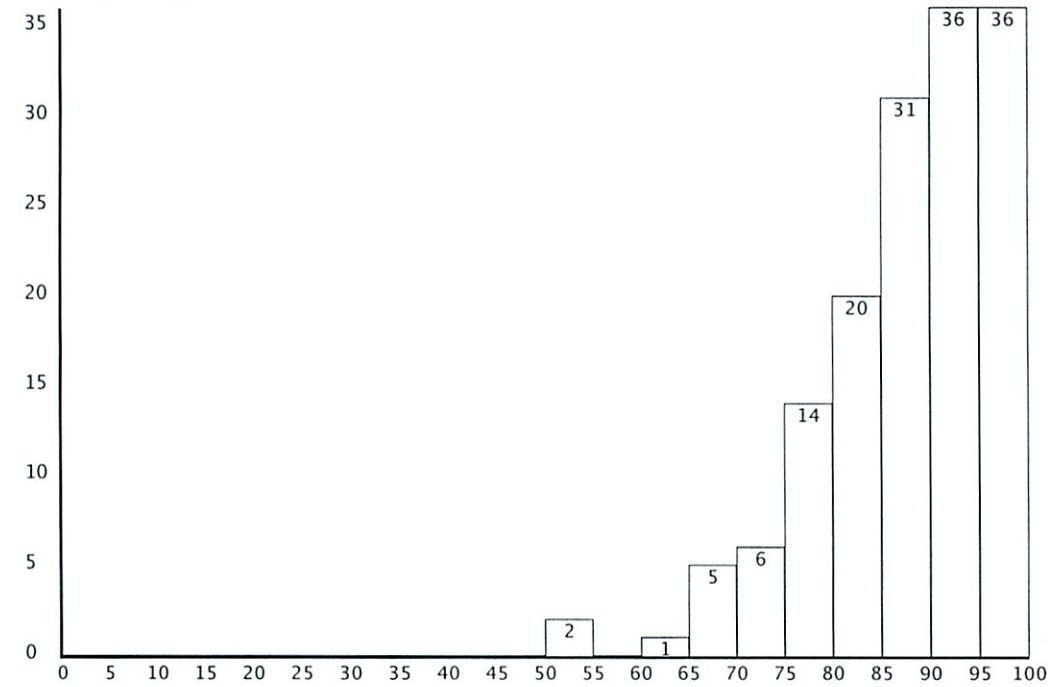
6.005 Software Construction

Dashboard

Students

Assignments

Grading Summary for Quiz 2



Number of Scores: 151  
Average: 87.63  
Standard Deviation: 9.31