# CS1007: Object Oriented Design and Programming in Java

Lecture #9

Feb 14

Shlomo Hershkop
*shlomo@cs.columbia.edu*

1

# Outline

- Queues
- Unit Testing
- Recursion and problem solving
- Sorting Algorithms

- Reading: 4.5 - 4.8, 4.9

2

# Announcements

- Homework 2 due Feb 27
  - Hint: Start early
  - Will help you learn the theory
  - Will help in planning hw3
- Midterm reminder:
  - Feb 28

3

# Queues

- A queue object holds things in fifo order
- We can add to the queue and get the first item
- Encapsulation: no idea on how things are actually kept in the queue

4

## First implementation

- Simple array
- Get will remove element 0, and advance everyone …..why??

5

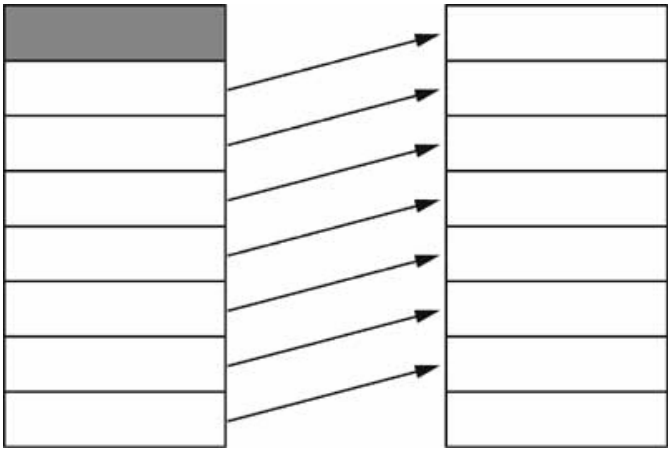
## •CWDB

6

- Rethink entire idea
- Goal: Efficient implementation of bounded queue
- Avoids inefficient shifting of elements

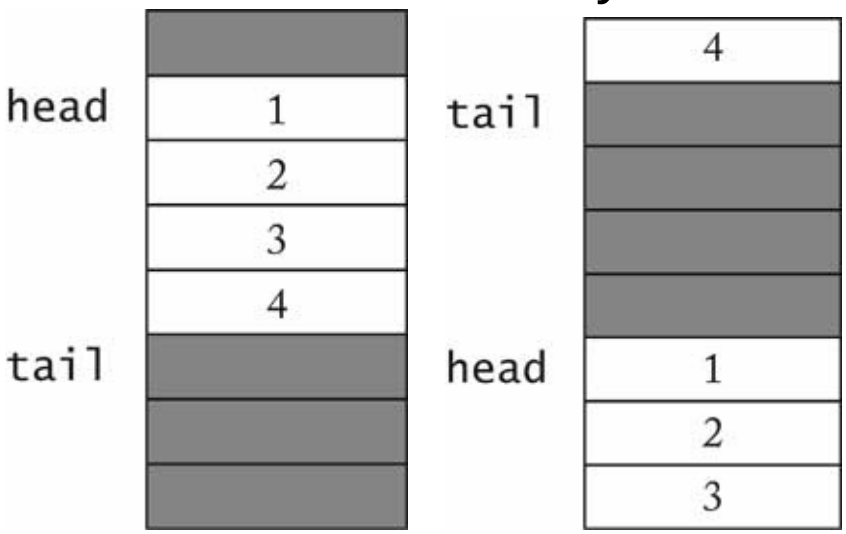
7

# Problem with Array: get the first



8

# Circular Array




9

# Preconditions

- In circular array implementation, failure of remove precondition corrupts queue!
- Bounded queue needs precondition for add
- Naive approach:

```
@precondition size()< elements.length
```

- Precondition should be checkable by caller
- Better:

```
@precondition size() < getCapacity()
```

10

## Lesson

- Encapsulation allows the same operation, but now we can do it much faster
- Prove this!

11

## Testing conditions

- Lets say you want to have your code
- And you want to test for certain conditions
- What do you usually do?

12

# Java Assertion Command

- Mechanism for warning programmers
- Can be turned off after testing
- Useful for warning programmers about precondition failure
- Syntax:

```
assert condition;
```

- Assumes condition is **true**
- Throws AssertionError if condition false and checking enabled

13

- `assert condition : expression;`
- This version pass the expression to the error statement, allowing you to print out why the assert failed.

14

# Example

```
public Message remove()
{
   assert count > 0 : "violated precondition size()
  > 0";
   Message r = elements[head];
   . . .
}
```

- During testing, run with

```
java -enableassertions MyProg
```

- Or shorter, java -ea

15

# Remember Document your Exceptions

```
/**
   . . .
   @throws NoSuchElementException if queue is empty
*/
public Message remove()
{
   if (count == 0)
      throw new NoSuchElementException();
   Message r = elements[head];
   . . .
}
```

- Exception throw part of the contract
- Caller can rely on behavior
- Exception throw not result of precondition violation
- This method has no precondition

16

# Postconditions

- Conditions that the service provider guarantees
- Every method promises description, @return
- Sometimes, can assert additional useful condition
- Example: add method

```
@postcondition size() > 0
```

- Postcondition of one call can imply precondition of another:

```
q.add(m1);
m2 = q.remove();
```

17

# Class Invariant

- Logical condition which holds for objects of a class before and after any method call, but can not guaranteed during method call.

- Example: Othello game
  How can the code guarantee that the game state is in a legal state??

18

# Class Invariants

- Condition that is
  - true after every constructor
  - preserved by every method
    (if it's true before the call, it's again true afterwards)
- Useful for checking validity of operations

19

# Example 2

- Example: Circular array queue

```
0 <= head && head < elements.length
```

- First check it's true for constructor
  - Sets head = 0
  - Need precondition size > 0!
- Check mutators. Start with add
  - Sets headnew = (headold + 1) % elements.length
  - We know headold > 0 (Why?)
  - % operator property:

```
0 <= headnew && headnew < elements.length
```

- What's the use? Array accesses are correct!

```
return elements[head];
```

20

# 3 proposals in Testing

- Don't:
  - Hope to be bought out before anyone realizes
    - Requirements: ticket to get out of town once they realize
- Assemble everything and then test
  - See above
- Test individual components before integrating
  - Ability to only test each piece separately, but allows you to work out many bugs early on.

21

# Automated testing

- Humans hate testing…
- Fast verification that new feature has not broken code
- Verify all code on a regular basis
- No grumble if test to rerun test ☺

22

# Unit Testing

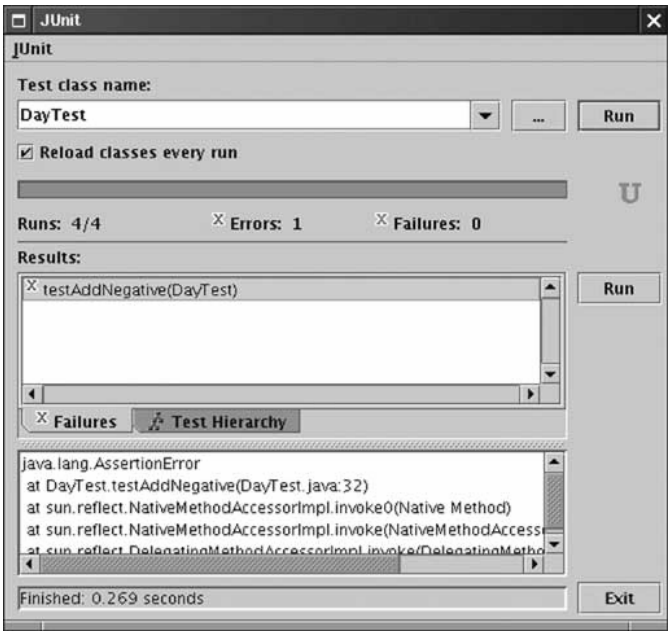
- Unit test = test of a single class
- Design test cases during implementation
- Run tests after every implementation change
- When you find a bug, add a test case that catches it

23

# JUnit

- http://www.junit.org/
- Framework for running tests on your code
- Need to plan out tests not random
  - Code a special class to run tests on your other classes
  - Will explore this in next programming homework, so please play with this now
- Need to realize advantages and disadvantages of this framework
  - It is only a TOOL!

24




25

# JUnit

- Test class name = tested class name + Test
- Test methods start with test

```
import junit.framework.*;
public class DayTest extends TestCase
{
   public void testAdd() { ... }
   public void testDaysBetween() { ... }
   . . .
}
```

26

# JUnit

- Each test case ends with assertion
- Test framework catches assertion failures

```
public void testAdd()
{
   Day d1 = new Day(1970, 1, 1);
   int n = 1000;
   Day d2 = d1.addDays(n);
   assertTrue(d2.daysFrom(d1) == n);
}
```

27

# GPS coordinates

- We've wrapped up chapter 1-3, will cover 4 before the midterm.
- Am going now to sidetrack a little

28

# Shift gears

- General problem solving:
  - Recursion
  - Memoization
- Sorting Algorithms
  - Measurements
  - Coding

29

# Recursion

- A solution that is partially defined in terms of itself

30

# Example

Factorial of N

N!

N * N-1 * N-2 * ..

31

```
public int factorial(int n){

 if(n == 1)
    return 1;
 else
    return factorial(n-1) * n;
 }
```

32

## Rules of recursion

1. Base Case
   Need to define exit strategy

2. Make Progress
   Need to move towards solution

3. Always assume recursive call works
   no need to trace out long program

33

## Fibonacci Series

$$F_i = F_{i-1} + F_{i-2}$$

$$F_0 = 0 \qquad F_1 = 1$$

34

- How would you define the solution recursively?

35

```
public static long fib(int n) {

   if (n < 1) {
        return n;
   }
   else {
        return fib(n-1) + fib(n-2);
   }
}
```

36

- Problem:
  Fib(100)
  should be 99 additions, actually takes about 10 minutes ☺

37

## More recursive rules

- Never duplicate the same work by solving the same instance of the problem in a separate recursive call

38

## So how would we figure out fib?

39

## Sort

- A sorting algorithm takes an unordered array of objects and returns an array of objects in ascending or descending order.
- Ascending is defined by the programmer or object type.
- We can use integers as an example
  - Number line defines order

40

# Bubble Sort

- The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required.
- The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order).
- This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

41

# Algorithm

- Input: List of integers $X_0, X_1 \ldots X_n$
- Output: ascending order

1. i = 0, swap =0
2. If $X_i > X_{i+1}$ swap $X_i$ and $X_{i+1}$ swap =1
3. i = i + 1
4. if i < n goto step 2
5. If swap > 0 goto step 1
6. Return sorted list

42

# Pseudo code

```
void bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```



# Example

- 6 9 3 1 8

## How to analyze this algorithm

- Will be taught in data structures
- Enough to know:
  - Slowest sort in general
  - Run time:
    - Anyone know how many comparisons required?
  - Advantages:
    - For small number of items, ok to use
    - Simple

45

## Polymorphism

- Definition:
  - Programming language's ability to process objects independent of their data type or class with the same set of code
  - i.e. example draw a shape on the screen
    - Triangle
    - Square
    - Circle
  - Perfect for Object Oriented design

46

# Next time

- Read chapter 4-4.6
- Start hw2

47