Object Oriented Programming and Design in Java

Session 18

Instructor: Bert Huang

Announcements

- Homework 4 due Mon. Apr. 19
  - No multithreading in programming part
- Final Exam
  Monday May 10, 9 AM - noon,
  173 MACY (this room)

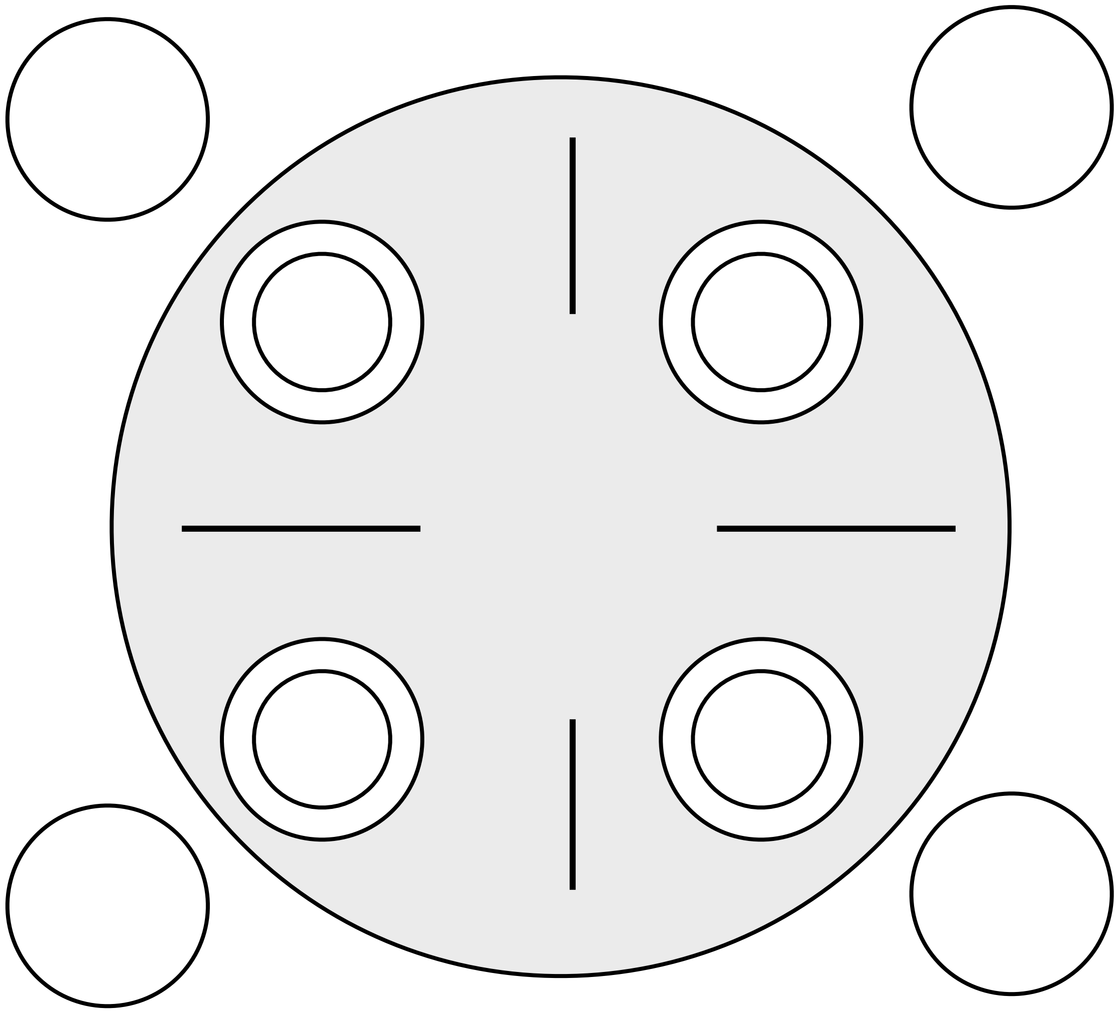
Review

- Multithreading
  - Thread, Runnable
- Handling Race conditions
  - Lock, Condition, synchronized
- Producer Consumer

Today's Plan

- Deadlocks and the Dining Philosophers Problem
- More on Threads in Java
  - Thread, Runnable, Object javadoc
  - Keywords synchronized and volatile
  - ReentrantLock
- Programming by contract and threads

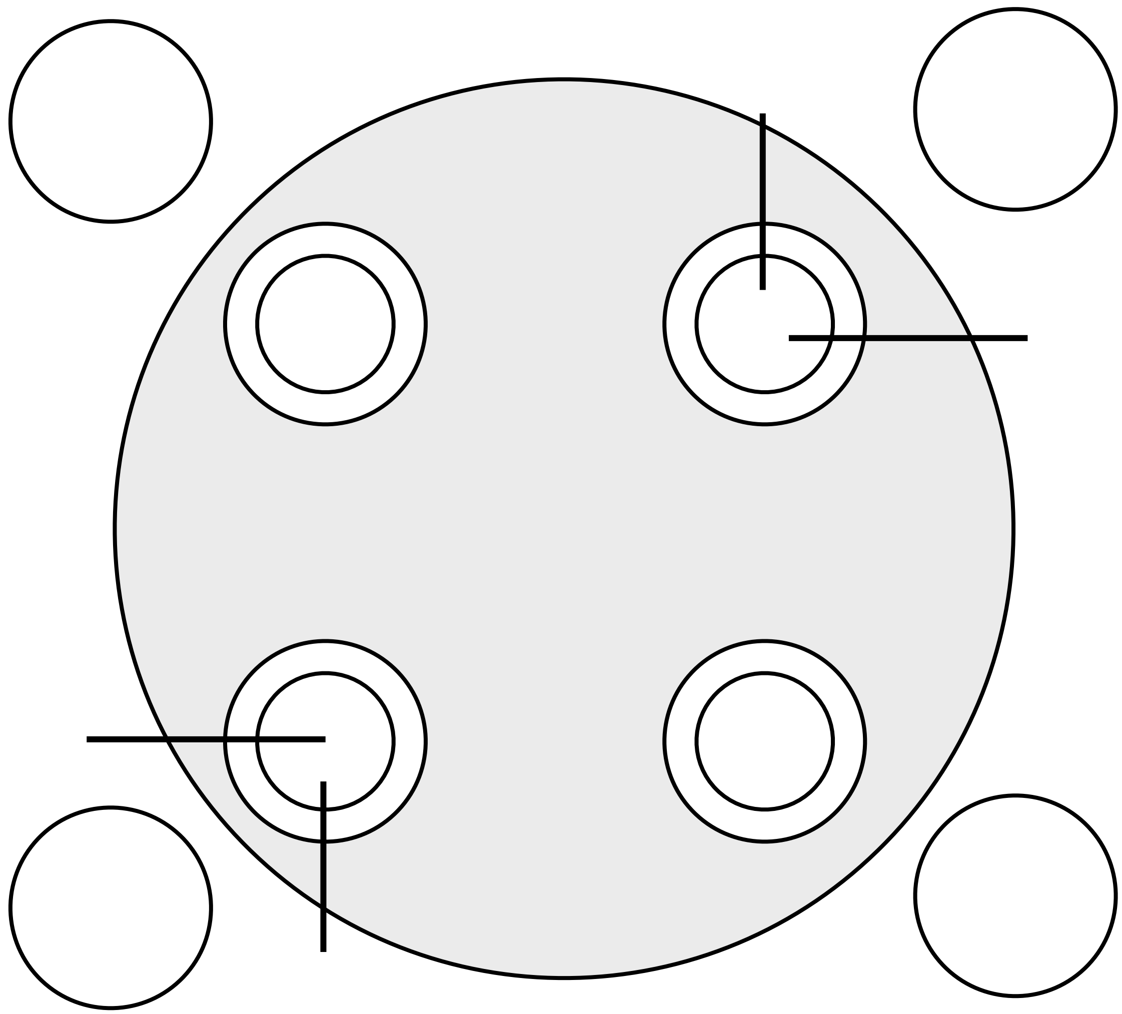
Dining Philosophers



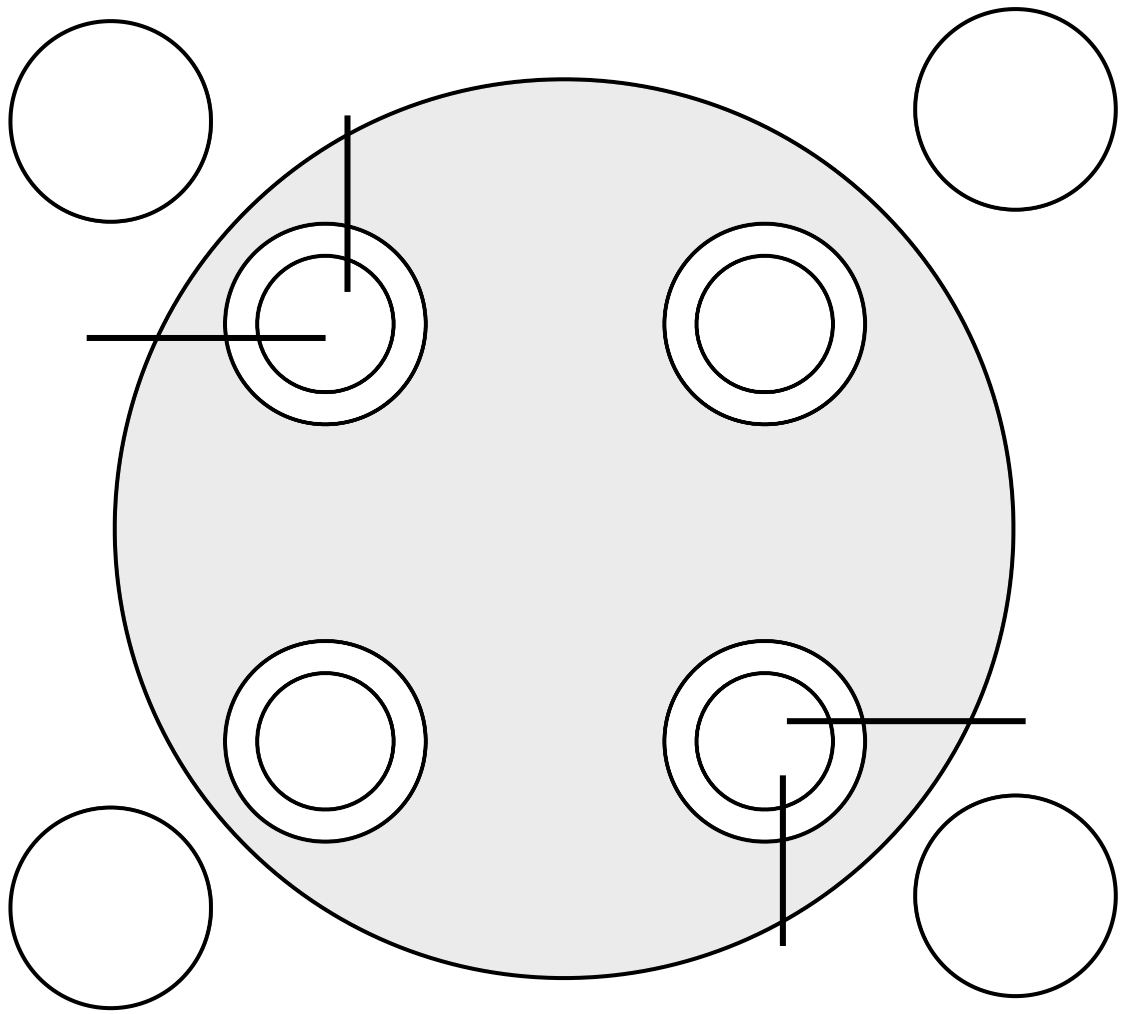
Dining Philosophers

- Example of deadlock when threads need two or more locks (e.g., moving objects from list to list)
- Each diner locks chopsticks then eats
  - leftChopstick.lock()
    rightChopstick.lock()
    eat()
    rightChopstick.unlock()
    leftChopstick.unlock()

Dining Philosophers



Dining Philosophers



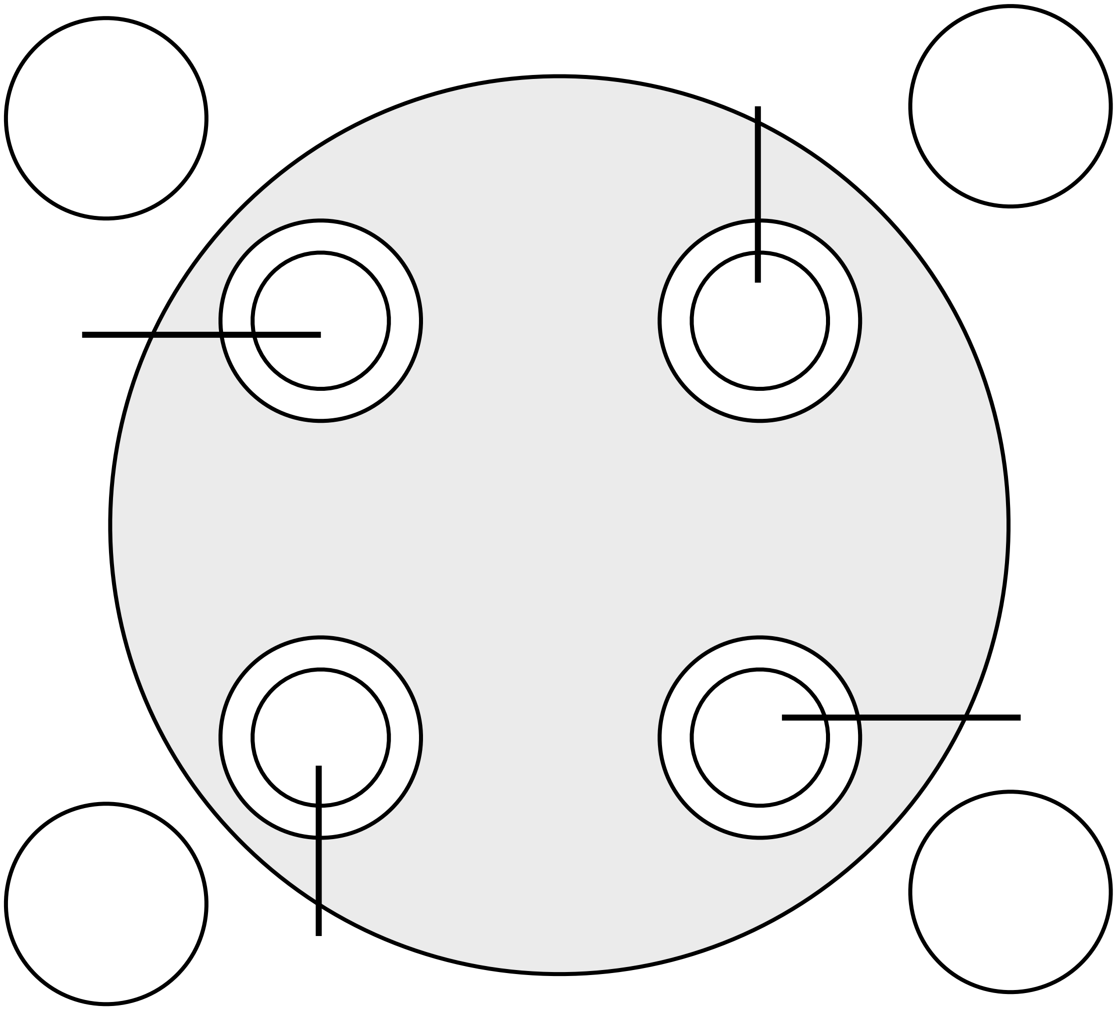
First Problem: Starvation

- Since we don’t know how OS will schedule threads, two diners may never get to eat
- ReentrantLock has a **fairness** flag that makes sure locks are granted first-come-first-served
  - new ReentrantLock(true);

Second Problem: Deadlock

- If all diner threads start simultaneously, we can get stuck in a *deadlock*
- Each philosopher locks his left chopstick, waits for right chopstick
- Even if we use conditions and release the chopsticks, we could have *livelock*
    - Infinite loop of simultaneously locking and releasing the left chopsticks

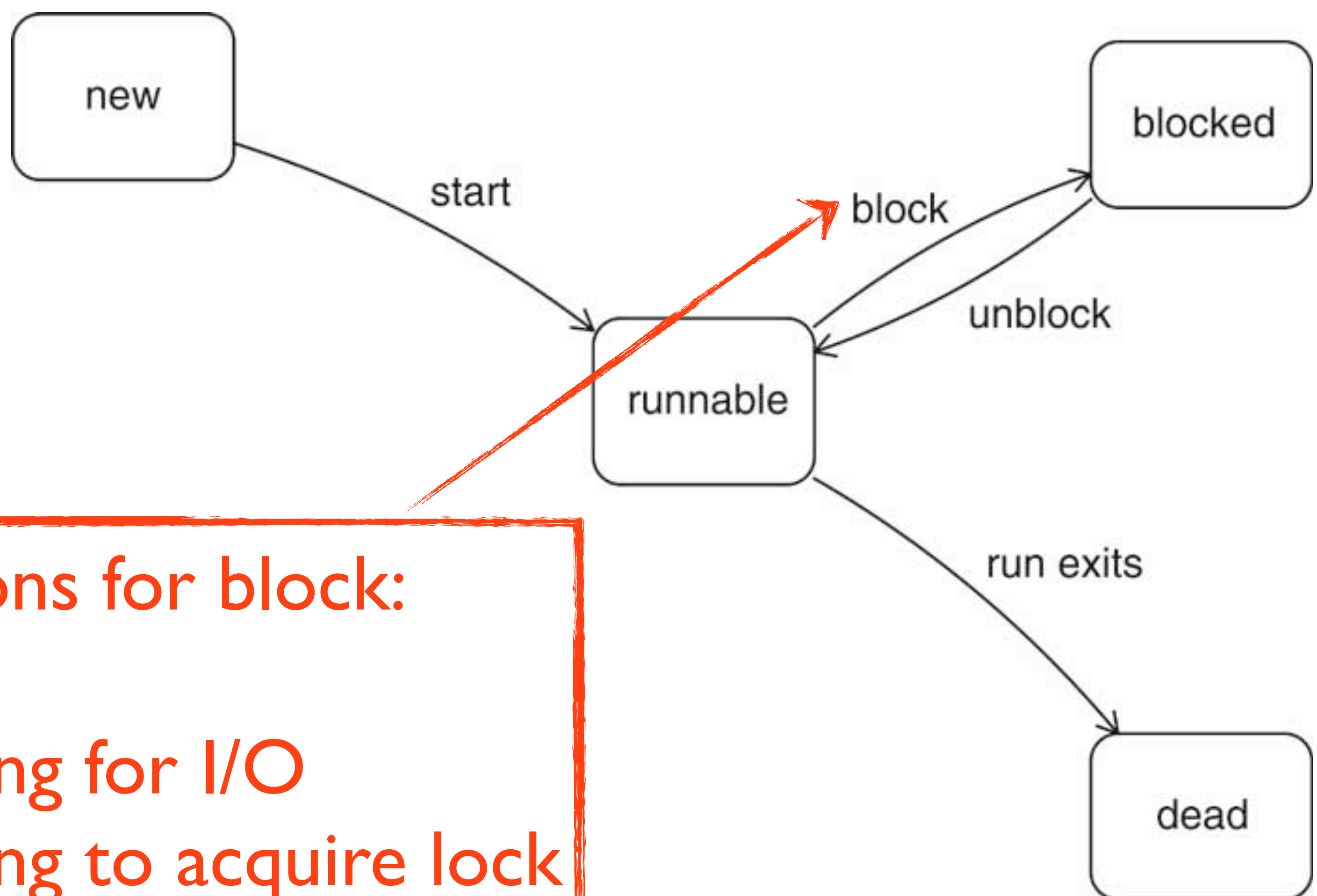
Dining Philosophers



Two Deadlock Solutions

- Order the chopsticks; locks must be acquired in the same order
  - No circular deadlock, but now some threads have higher priority
- Require master lock to lock any chopsticks
  - master.lock()
    leftChopstick.lock(); rightChopstick.lock();
    master.unlock();
    eat()
    leftChopstick.unlock(); rightChopstick.unlock()

Thread States



Reasons for block:
Sleep
Waiting for I/O
Waiting to acquire lock
Waiting for condition

Thread (abridged)

- `void join()` - Waits for this thread to die
- `static void sleep(long millis)` - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- `void start()` - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- `static void yield()` - Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Runnable

Method Summary

`void`	`run()` When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

Method Detail

run

`void run()`

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

Object

Method Summary

protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this `Object`.
int	hashCode() Returns a hash code value for the object.
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
String	**toString**() Returns a string representation of the object.
void	**wait**() Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
void	**wait**(long timeout) Causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos) Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

synchronized

- Methods with keyword `synchronized` automatically lock the containing object when called
- We can explicitly acquire the object lock
  ```
  synchronized(objectToLock) { ... }
  ```
- This allows us to use unsafe objects safely
  ```
  synchronized(myArrayList) {
      myArrayList.add(i);
  }
  ```

Volatile Fields

- A misunderstood method to make synchronize threads is to declare fields with keyword *volatile*
- volatile guarantees that the field is never cached by a thread
- whereas nonvolatile fields may be copied in other threads by compiler optimizations
- volatile will not help synchronization when the problems come from multiple operations

ReentrantLock

- Allows multiple lock acquisitions by a single thread
- Thread that owns it may call lock() again many times

```
myLock.lock();  // acquires ownership of myLock
myLock.lock();  // acquires a 2nd lock on myLock
```

- ReentrantLock will not unlock until unlock() is called the same number of times

```
myLock.unlock(); // releases the 2nd lock
myLock.unlock(); // releases the original lock
```

Recursive Locks

- Recursive locks are controversial
  - They encourage code that allows threads to hold onto locks longer
  - Locks stop concurrency
- But they help preserve encapsulation and abstraction:
  - you can make recursive calls without having each call know about the state of the lock

Threads and Invariants

- We prove class invariants by showing that the invariant is true when all methods finish
- Multithreading allows interaction before methods finish
- Preserve invariants by locking around blocks of code where the invariant may not be true
- e.g., `A[size]` is the next empty slot of the array

Threads and Preconditions

- A precondition that is true when a method is called may not be true when the relevant logic is executed
- Preserve the precondition by locking the objects involved at method call
  - maybe too restrictive

Multithreading

- Multithreading is small-scale parallel computing, *i.e.*, a practice ground for the future of computing
- Relatively new challenge in software design; multicore only popularized recently in consumer machines
- Encapsulation, good OOP are still major challenges,
  - e.g., a synchronized, threadsafe ArrayList may lock too much for some applications

Reading

- Horstmann Ch. 9