CS301 Software Development

Instructor: Peter Kemper

Introduction to Multithreading - Part 2

Multithreading

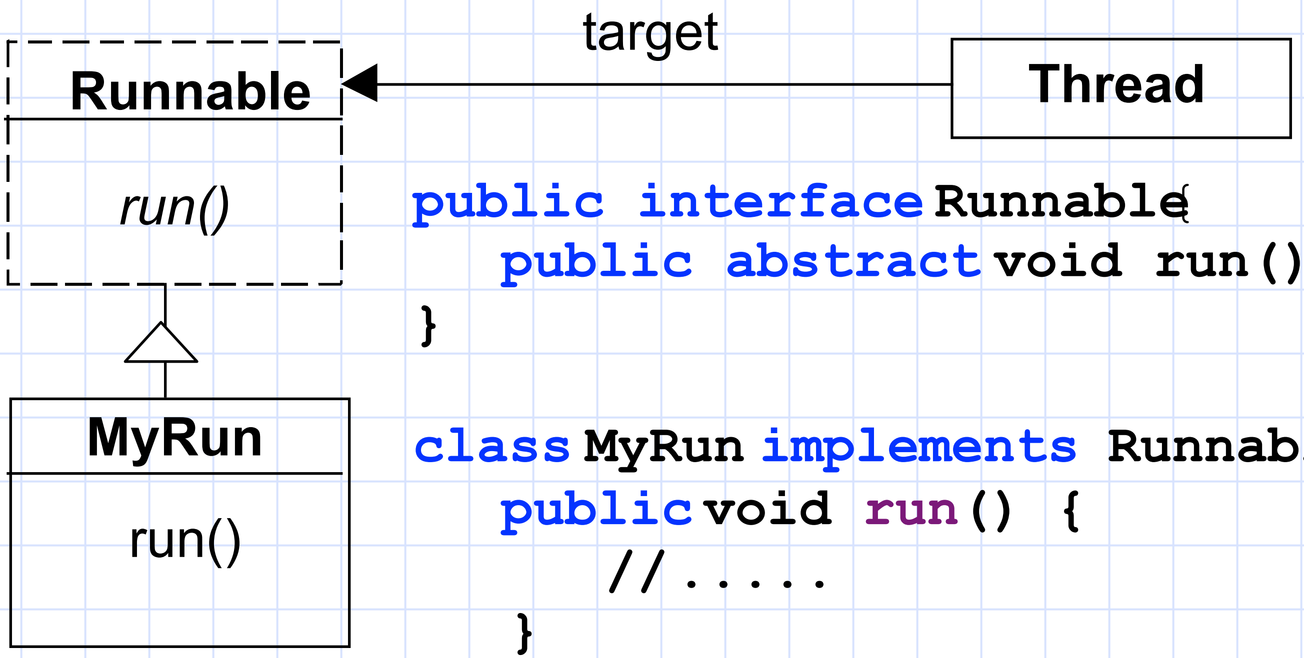
- So far:
  - Create threads
  - Assign work to a thread
  - Terminate a thread
- Communicate data between threads
  - Race condition
  - Locks and synchronized method calls
- Based on examples from
  - Magee & Kramer, Concurrency: State Models & Java Programs, 2nd ed, Wiley.

Fun with Threads: Examples from Magee and Kramer

- Introduction, 1 thread: CountDown Timer
- Concurrent Execution: Thread Demonstration
- Shared Objects & Interference: Ornamental garden
- Monitors & Condition Synchronization
  CarPark, Semaphore Demonstration, Bounded Buffer,
  Nested Monitor example and Fixed Nested Monitor example
- Deadlock:
  Dining Philosophers and Fixed Dining Philosophers
- Safety & Liveness
  Single Lane Bridge
  Readers and Writers and variants (writers priority, fair)

Example: Countdown Timer class

Implement interface “Runnable”.



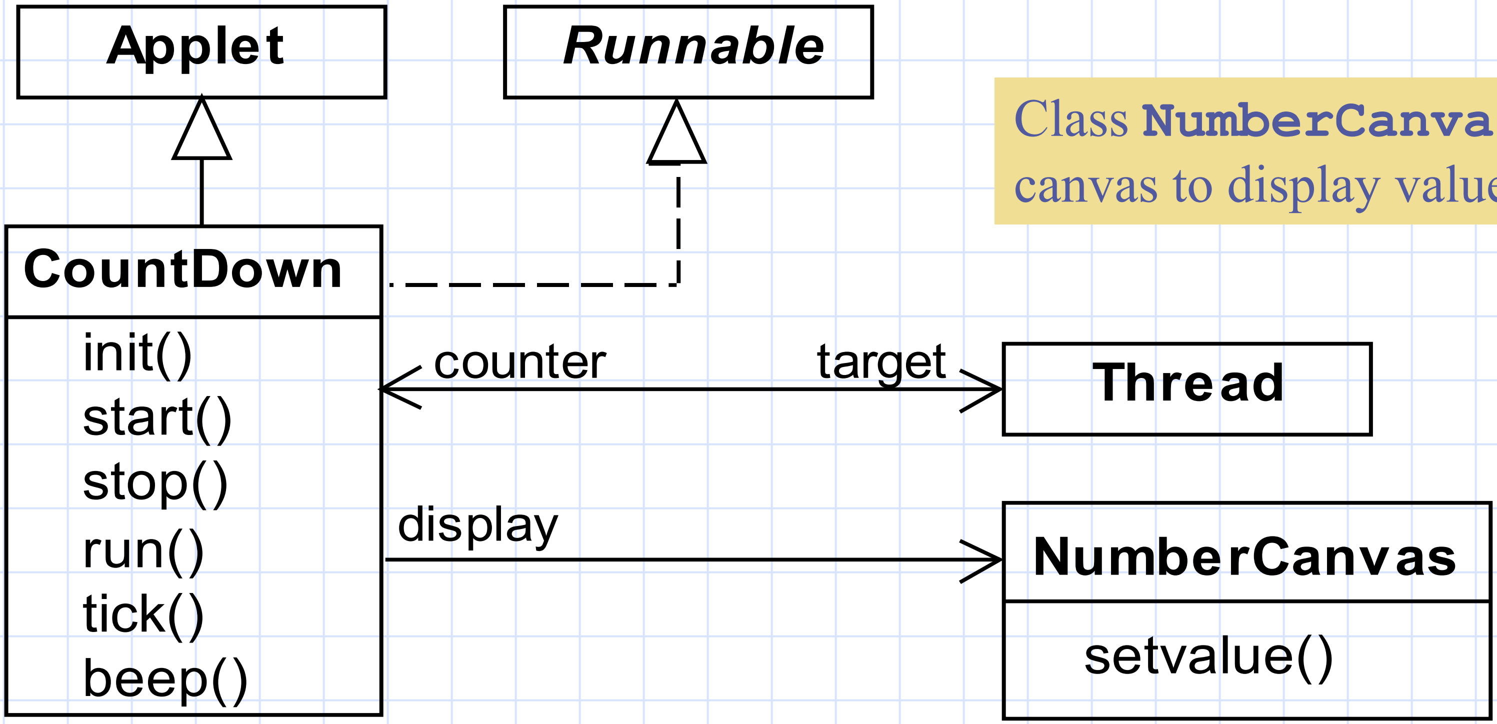

```
public interface Runnable{
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}
```

How to create thread with own run() method in MyRun?

Thread lokalMyRun = new Thread(this) ;

Example: CountDown timer - class diagram



Class **NumberCanvas** gives canvas to display value.

Class **CountDown** inherits from **Applet**, implements **run()** method for a thread.

Example: CountDown Class

```
public class CountDown extends Applet
                       implements Runnable
{
  Thread counter; int i;
  final static int N = 10;
  AudioClip beepSound, tickSound;
  NumberCanvas display;
  public void init()  {...}
  public void start() {...}
  public void stop()  {...}
  public void run()   {...}
  private void tick() {...}
  private void beep() {...}
}
```

Class CountDown - start(), stop() and run()

```
public void start() {
    counter = new Thread(this);
    i = N; counter.start();
 }
 public void stop() {
    counter = null;
 }
 public void run() {
    while(true) {
      if (counter == null) return;
      if (i>0)  { tick(); --i; }
      if (i==0) { beep(); return;}
    }
 }
```

WARNING:

Name clash:

Thread - Applet

wrt Start(), stop()

Here: Applet creates new thread in start() and starts it.

Thread constructor with parameter (this)

calls counter.start()

which in turn calls this.run().

Sensing request for termination with condition

counter == NULL

and not with interrupt.

So far

Typical situation in compositional approaches

- 1st step: get the components
  - Define threads, get them running, make them terminate

- 2nd step: combine components to overall system
  Here: communication among threads?
  - In principle: simple,
    threads communicate via shared memory, i.e. objects
    e.g. explicitly via an object “messageQueue”
    e.g. implicitly via objects that hold data and are accessed by several threads (not necessarily aware of one another)

  - In practice: shared data must be handled with care!

Example by Magee and Kramer: Ornamental Garden

- Scenario: Garden visitors counted at 2 turnstiles
  - Global variable: int counter = 0 ;
  - Threads A and B
    - Do 20 times
    - Read counter, increment value, update counter

```
for (i = 0 ; i < 20 ; i++)
{
    x = counter ; x++ ; counter = x ;
}
```

  - Both threads communicate via shared variable counter
  - Expected result: 40
  - Does it work?
    - Often, but there is no guarantee ....
    - Occasionally we see results like 32, 37, 39, ...
  - Note:

    the real code for the example is modified to increase likelihood that error occurs, but does not artificially inject the error!

Race Condition

A race condition occurs if the effect of multiple threads on shared data depends on the order in which threads are scheduled.

- Obviously a bad thing.
  - Common requirement

    We want deterministic programs that reliably produce same correct output for same input.

  - Additional difficulty

    User: Sometimes,
    code does not work correctly

    Detection: Race condition errors do not show up all the times.

    Investigation:

      Effect of race condition may be difficult to reproduce,

      difficult to debug, …

  Essentially a nightmare for debugging and testing

  Developer: code works
  fine for me …

- So we need:
  - A design strategy to avoid race conditions
  - A technical, language mean that helps us programming a solution

Solution to Race Conditions

- Access to shared objects requires protection
  => Mutual exclusion:
  - Thread/process can access object or enter a critical code section only one at a time
  - Generalization, at most n threads/processes, common: n reader, 1 writer
- Key issue:
  - Thread can do that computation without being disturbed, interference with others
- Design strategy:
  - Check for all classes, objects with read/write access from several threads
  - Use encapsulation & restrict access
- In Java:
  - Synchronize access to shared objects to have at most n threads access data
  - Means to describe that: Locks
  - Important:
    - Concept should avoid busy waiting! (Which is the case here)

flickr, ©cc from Brian Luster
Rusted Lock

Object Locks

- Each object in Java has a lock
- Lock activated with keyword "synchronized"
- Calling a synchronized method acquires lock of implicit parameter
- Leaving the synchronized method releases lock
- Easier than explicit Lock objects

But granularity fixed to method calls!

```
public class BoundedQueue<E>
{
   public synchronized void add(E newValue) { . . . }
   public synchronized E remove() { . . . }
   . . .
}
```

flickr, ©cc from Brian Luster
Rusted Lock

Summary

- Create threads
- Assign work to a thread
- Terminate a thread
- Communicate data between threads
  - Shared memory, shared objects
  - Race condition
  - Locks and synchronized method calls

Challenges in testing a multithreaded class

- **Scenario:**
  - Class under test C performs time intensive internal operation g() with separate thread if some method C.f() is called.
  - C.f() returns immediately, and g() communicates results back later by calling a method h() of the calling class.
  - Testing the class is difficult as method call C.f() returns immediately and well before calculation of g() is done.
- **Necessary:**
  - Postpone testing results of f() till g() has finished.
  - Create a stub object (an artificial environment) to call C.f() that also provides a call back method h().
- **Options:**
  - Add test code to call back method h().
  - Make test code wait for extra thread to terminate
    - wait long enough (make own thread sleep), or
    - wait for termination of extra thread, i.e. thread.join()