CS193J: Programming in Java
Summer Quarter 2003

Lecture 8
Object Serialization, Threading

Manu Kumar

sneaker@stanford.edu

Handouts

- 2 Handouts for today!
  - #19: Threading
  - #20: Threading 2

Recap

- Last Time
  - Continued with Repaint
    - Repaint example code walkthrough
    - Erasing
  - Mouse Tracking
    - DotPanel example code walkthrough
  - Advanced Drawing
    - Region based drawing, Blinking, Smart Repaint
- Assigned Work Reminder
  - HW 2: Java Draw
    - Due before midnight on Wednesday, July 23rd, 2003

Lecture-Homework mapping revisited

- HW #2 will use
  – OOP concepts
    - Inheritance, overriding, polymorphism
    - Abstract classes
  – Drawing in Java
    - Layouts
    - paintComponent()
  – Event handling
    - Anonymous Inner classes
  – Repaint
  – Mouse Tracking
  – Advanced Drawing
  – Object Serialization (Today)

Today

- Object Serialization
  - Cloning
    - Not Dolly, but Java Objects ☺
  - Serializing
- Introduction to Threading
  - Motivation
  - Java threads
    - Simple Thread Example
- Threading 2
  - Race Conditions
  - Locking
  - Synchronized Method

Objects and Serialization (Handout #18)

- Equals revisited
  - a == b tests for pointer equality only
    - i.e. pointer a and b point to the same location/object
    - This is called “shallow semantics”
  - boolean Object.equals(Object other)
    - Defined in the Object class
      - Default implementation does a == b test (shallow semantics)
    - May override to do “deep comparison”
      - Example: String.equals()

Calling equals()

```
{
    String a = “hello”;
    String b = “hello”;

    (a == b) → false
    (a.equals(b)) → true
    (b.equals(a)) → true
}
```

Equals strategy

- boolean equals(Object other)
  - Take Object, return boolean
    - Must have exact prototype for overriding to work
  - Return true on (this == other)
  - Use (other instanceof Foo) too test class of other
    - False if not same class
  - Otherwise do a field-by-field comparison of this and other

Student equals() example

```
// in Student class...
boolean equals(Object obj) {
  if (obj == this) return(true);
  if (!(obj instanceof Student)) return(false);
  Student other = (Student)obj;
  return(other.units == units)
}
```

Cloning

- Used to create a copy of an object
  - Not just another pointer to the same object
  - Cloned object has it's own memory space
- Lets say Foo b = a.clone();
  - a == b will return false
  - a.equals(b) will return true!
- Copied object has same state
  - But its own memory
- We use this in HW#2 for cut-copy-paste!

Cloneable interface

- Used as a marker to indicate that the class implements the clone() method
  - Not compiler enforced
  - Object.clone() is pre-built
    - Create a new instance of the right class
    - Assign all fields over with '=' semantics
- Object.clone() will do above default behavior
  - If class implements the cloneable interface
  - Otherwise, it will through an exception

Implementing clone()

- Implement the Cloneable interface
  - Call the super classes clone method first to copy structure
    - copy = (Class) super.clone()
  - Copy fields where a simple ‘=‘ is not deep enough
    - Example, arrays, arraylists, objects

Alternative approaches

- Copy Constructor
  - MyClass(MyClass myObject)
    - Construct a new instance of MyClass based on the state of MyObject
- “Factory” method
  - Static method that makes new instances
    - static MyClass newInstance(MyClass myObject)
    - May use constructor internally
- Advantage
  - Simpler than Object.clone(), no new concepts
- Disadvantage
  - Client must know the class of the Object

Eq Code example

```
// Eq.java

/*
 Demonstrates a simple class that defines equals and clone.
*/
public class Eq implements Cloneable {
    private int a;
    private int[] values;

    public Eq(int init) {
        a = init;
        values = new int[10];
    }
```

Eq Code example: equals

```
/*
 Does a "deep" compare of this vs. the other object.
*/
public boolean equals(Object other) {
        if (other == this) return(true);
        if (!(other instanceof Eq)) return(false);

        Eq e = (Eq) other;

        // now test if this vs. e
        if (a != e.a) return(false);

        if (values.length != e.values.length) return(false);
        for (int i=0; i<values.length; i++) {
                if (values[i] != e.values[i]) return(false);
        }
        return(true);
}
```

Eq Code example: clone()

```
/*
 Returns a deep copy of the object.
*/
public Object clone() {
    try {
        // first, this creats the new memory and does '=' on all fields
        Eq copy = (Eq)super.clone();

        // copy the array over -- arrays respond to clone() themselves
        copy.values = (int[]) values.clone();
        return(copy);
    }
    catch (CloneNotSupportedException e) {
        return(null);
    }
}
```

Eq Code example

```
    public static void main(String[] args) {
            Eq x = new Eq(1);
            Eq y = new Eq(2);
            Eq z = (Eq) x.clone();

            System.out.println("x == z" + (x==z));        // false
            System.out.println("x.equals(z)" + (x.equals(z))); // true

    }
}
```

Serialization

- Motivation
  - A lot of code involves boring conversion from a file to memory
    - Write code in 106A to translate by hand
    - HW#1 read ASCII file and required parsing
  - This is a common problem!
- Java's answer:
  - Serialization
    - Object know how to write themselves out to disk and to read themselves back from disk into memory!
- We use this in HW#2 to load and save!

Serialization / Archiving

- Objects have state in memory
- Serialization is the process of conversting objects into a streamed state (Network, Disk)
  - No notion of an address space
  - No pointers
- Serialization is also called
  - Flattening, Streaming, Dehydrate (rehydrate = read), Archiving

How it works?

- To write out an object
  - ObjectOutputStream out;
  - out.writeObject(obj)
- To read that object back in
  - ObjectInputStream in;
  - obj = in.readObject();
- Must be of the same type
  - class and version

Java: Automatic Serialization

- Serializable Interface
  - By implementing this interface a class declares that is it willing to be read/written by automatic serialization machinery
- Automatic Writing
  - System knows how to recursively write out the state of an object
  - Recursively follows pointers and writes out those objects too!
  - Can handle most built in types
    - int, array, Point etc.
- "transient" keyword to mark a field that should not be serialized
  - Transient fields are returned as null on reading
- Override readObject() and writeObject() for customizations
- Versioning
  - Can detect version changes

Circularity: not an issue

- Serialization machinery will take circular references into account and do the right thing!

Dot example

- Build on DotPanel example!
- saveSerial(File f)
  - Given a file, write the data model to it with Java serialization.
  - Makes an Point[] array of points and writes it which avoids the bother of iteration.
    - We use an array instead of the ArrayList to avoid requiring a 1.2 VM to read the file, although maybe the ArrayList would have been fine
- loadSerial(File f)
  - Inverse of saveSerial.
  - Reads an Point[] array of Points, and adds them to our data model.

Dot example code

```
public void saveSerial(File file)  {
        try {
                ObjectOutputStream out = new ObjectOutputStream(
                        new FileOutputStream(file));

                // Use the standard collection -> array util
                // (the Point[0] tells it what type of array to return)
                Point[] points = (Point[]) dots.toArray(new Point[0]);

                out.writeObject(points);      //  serialization!

                out.close();          // polite to close on the way out
                setDirty(false);
        }
        catch (Exception e) {
                e.printStackTrace();
        }
    }
```

Dot example code

```
private void loadSerial(File file)  {
    try {
        ObjectInputStream in = new ObjectInputStream(new
    FileInputStream(file));

        // Read in the object -- the CT type should be exactly as it was written
        // -- Point[] in this case.
        // Transient fields would be null.
        Point[] points =  (Point[])in.readObject();
        for (int i=0; i<points.length; i++) {
                dots.add(points[i]);
        }

        in.close();             // polite to close on the way out
        setDirty(false);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

HW#2 note

- CS193J classes for serialization
  - shield you from the exceptions, but otherwise behave like ObjectOutputStream and ObjectInputStream

SimpleObjectWriter w;

SimpleObjectWriter w =
    SimpleObjectWriter.openFileForWriting(filename);

w.writeObject(<object>) -- write an array or object (Point[] in above example)

w.close()

SimpleObjectReader r;

SimpleObjectReader r =
    SimpleObjectReader.openFileForReading(filename);

obj = r.readObject() -- returns the object written -- cast to what it is (Point [] in above example)

r.close()

Threading (Handout #19)

- Introduction to Threading
  - Motivation
  - Java threads
    - Simple Thread Example

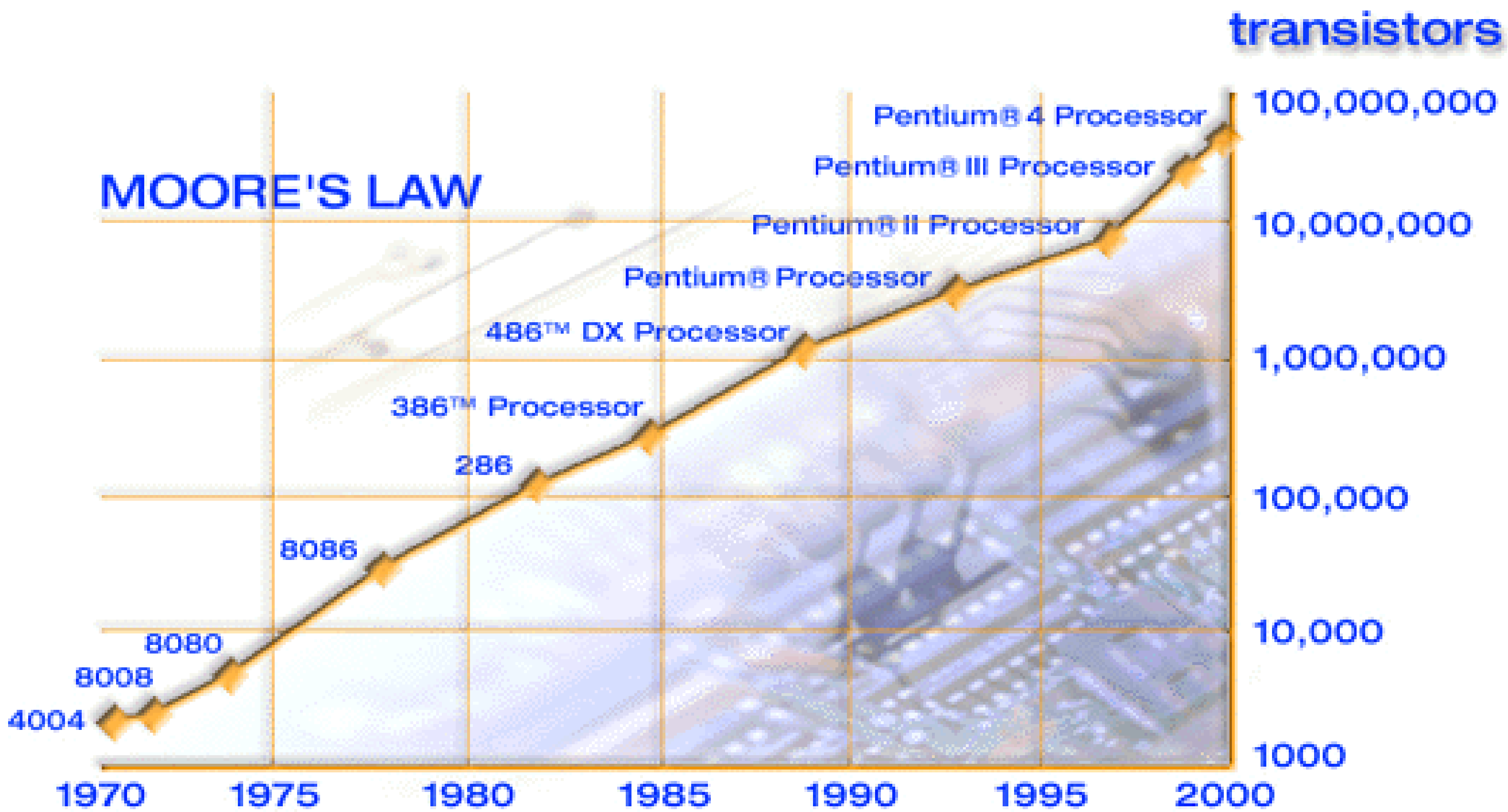
Faster Computers

- Why are computers today faster than 10 year ago?
  - Process improvements
    - Chips are smaller and run faster
  - Superscalar pipelining parallelism techniques
    - Doing more than one thing at a time from the one instruction stream
- Instruction Level Parallelism (ILP)
  - There is a limit to the amount of parallelism that can be extracted from a single instruction stream
    - About 3x to 4x
    - We are well within the diminishing returns region here

Hardware Trends

- Moore's Law
  - Moore's Law states that the number of of transistors on a microchip will double every 18 months (Promulgated by Gordon Moore in 1965)
  - The density of transistors we can fit per square mm seems to double every 18 months
    - Transistors become smaller and smaller
- What should we do with all these transistors??

Moore's Law at work…



Source: Intel Corporation, http://www.intel.com/research/silicon/mooreslaw.htm

In Numbers…

	Year of introduction	Transistors
4004	1971	2,250
8008	1972	2,500
8080	1974	5,000
8086	1978	29,000
286	1982	120,000
386™ processor	1985	275,000
486™ DX processor	1989	1,180,000
Pentium® processor	1993	3,100,000
Pentium II processor	1997	7,500,000
Pentium III processor	1999	24,000,000
Pentium 4 processor	2000	42,000,000

Source: Intel Corporation, http://www.intel.com/research/silicon/mooreslaw.htm

In Dollars…

- The cost of a chip is related to its size in square mm
  - Cost is a super linear function – doubling the size more than doubles the cost
- Recent processors

1989	**486**	**1.0um**	**1.2M transistors**	**79 mm^2**
1995	**Pentium MMX**	**035um**	**5.5 M transistors**	**128 mm^2**
1997	**AMD Athlon**	**0.25 um**	**22 M transistors**	**184 mm^2**
2001	**Pentium 4**	**0.18 um**	**42 M transistors**	**217 mm^2**

What can we use transistors for?

- More cache
- More functional units
  - Instruction Level Parallelism
- Multiple threads

- In 2002, Intel speculated that they could build a 1 billion transistor Itanium chip made of 4 Itanium cores and a huge shared cache

Hardware vs. Software

- Writing single threaded software is easier
  - – Therefore we have used hardware to drive the performance of software
- Hardware is however hitting a limit
  - – Not on the number of transistors yet
    - But on how much parallelism it can use based on single-threaded model code
  - – *Programmers must start writing explicitly parallel code in order to take benefit of the improvements in hardware!*

Hardware concurrency trends

- Multiple CPU's
  - Cache coherency must make expensive off-chip trip
- Multiple cores on a single chip
  - Can share on-chip cache
  - Good way to use up more transistors without doing more design
- Simultaneous Multi-Threading (SMT)
  - One core with multiple sets of registers
    - Shifts between one thread and another quickly
    - Hide latency by overlapping a few active threads
  - HyperThreading (Intel Pentium 4 processor)
- By 2005 – 2-4 cores with each being 2-4 way multi-threaded
  - Appears to have 4-16 CPUs

Software concurrency

- Processes
  - Unix-style concurrency
  - The ability to run multiple applications at once
    - Example: Unix processes launched from a shell, piped to another process
  - Separate address space
  - Cooperate using read/write streams (pipes)
  - Synchronization is easy
    - Since there is no shared address space

Threads

- The ability to do multiple things at once within the same application
  - Finer granularity of concurrency
- Lightweight
  - Easy to create and destroy
- Shared address space
  - Can share memory variables directly
  - May require more complex synchronization logic because of shared address space

Advantages of threads…

- Use multiple processors
  - Code is partitioned in order to be able to use n processors at once
    - This is not easy to do! But Moore's Law may force us in this direction
- Hide network/disk latency
  - While one thread is waiting for something, run the others
  - Dramatic improvements even with a single CPU
    - Need to efficiently block the connections that are waiting, while doing useful work with the data that has arrived
  - Writing good network codes relies on concurrency!
    - Homework #3b will be a good example of this
- Keeping the GUI responsive
  - Separate worker threads from GUI thread

Why Concurrency is a Hard Problem

- No language construct to alleviate the problem
  - Memory management can be solved by a garbage collector, no analog for concurrency
- Counter-intuitive
  - Concurrency bugs are hard to spot in the code
  - Difficult to get into the concurrency mindset
- No fixed programmer recipe either
- Client may need to know the internal model to use it correctly
  - Hard to pass the Clueless-Client test
- Concurrency bugs are random
  - Show up rarely, often not deterministic/reproducible easily
  - Rule of thumb: if something bizarre happens try and note the current state as well as possible

Java Threads

- Java includes built-in support for threading!
  - Other languages have threads bolted-on to an existing structure
- VM transparently maps threads in Java to OS threads
  - Allows threads in Java to take advantage of hardware and operating system level advancements
  - Keeps track of threads and schedules them to get CPU time
  - Scheduling may be pre-emptive or cooperative

Current Running Thread

- "Thread of control" or "Running thread"
  - The thread which is currently executing some statements
- A thread of execution
  - Executing statements, sending messages
  - Has its own stack, separate from other threads
- A message send sends the current running thread over to execute the code in the receiver

An idiom explained even more!

- Remember:
  - public static void main(String[] args)
- Well…
  - When you run a Java program, the VM creates a new thread and then sends the main(String[] args) message to the class to be run!
  - Therefore, there is ALWAYS at least one running thread in existence!
    - We can create more threads which can run concurrently

Java Thread class

- A Thread is just another object in Java
  - It has an address, responds to messages etc.
  - Class Thread
    - in the default java.lang package
- A Thread object in Java is a token which represents a thread of control in the VM
  - We send messages to the Thread object; the VM interprets these messages and does the appropriate operations on the underlying threads in the OS

Creating Threads in Java

- Two approaches
  - Subclassing Thread
    - Subclass java.lang.Thread
    - Override the run() method
  - Implementing Runnable
    - Implement the runnable interface
    - Provide an implementation for the run() method
    - Pass the runnable object into the constructor of a newThread Object

Why two approaches?

- Remember: Java supports only single-inheritance
  - If you need to extend another class, then cannot extend thread at the same time
    - Must use the Runnable pattern
- Two are equivalent
  - Whether you subclass Thread or implement Runnable, the resulting thread is the same
  - Runnable pattern just gives more flexibility

Thread Lifecycle

- Steps in the lifecycle of a thread
  - Instantiate new Thread Object (thread)
    - Subclass of Thread
    - Thread with a runnable object passed in to constructor
  - Call thread.start()
    - This begins execution of the run() method
  - Thread finishes or exits when it exits the run() method
    - Idiom – run() method will have some form of loop in it!
  - Optional - thread.sleep or thread.yield()
  - Thread.stop(), thread.suspend() and thread.resume() are deprecated!
    - See http://java.sun.com/j2se/1.4.1/docs/guide/misc/threadPrimitiveDeprecation.html

Thread.currentThread()

- Static utility method in the Thread class
  - Returns a pointer to the Thread object that represents the current thread of control
- Example

```
int i = 6;
int sum = 7 + 12;          // regular computation

Thread me = Thread.currentThread();
// "me" is the Thread object that represents our thread of
// control (the thread that computed the sum above)
```


Joining

- Used when a thread wants to wait for another thread to complete its run()
  - Sent the thread2.join() message
    - Causes the current running thread to block efficiently until thread2 finishes its run() method
    - Must catch InterruptedException
      - We will talk about exceptions more later, for now just treat it as an idiom

Join Example

```
// create a thread
Runnable runner = new Runnable() {
    public void run() {
        // do something in a loop
    };
Thread t = new Thread(runnner);

// start a thread
t.start();

// at this point, two threads may be running -- me and t
// wait for t to complete its run
try {
    t.join();
}
catch (InterruptedException ignored) {}
// now t is done (or we were interrupted)
```

Simple Thread Example

```
/*
 Demonstrates creating a couple worker threads, running them,
 and waiting for them to finish.

 Threads respond to a getName() method, which returns a string
 like "Thread-1" which is handy for debugging.
*/
public class Worker1 extends Thread {
	public void run() {
		long sum = 0;
		for (int i=0; i<100000; i++) {
			sum = sum + i;	// do some work

			// every n iterations, print an update
			// (a bitwise & would be faster -- mod is slow)
			if (i%10000 == 0) {
				System.out.println(getName() + " " + i);
			}
		}
	}
```

Simple Thread Example

```
    public static void main(String[] args) {
        Worker1 a = new Worker1();
        Worker1 b = new Worker1();

        System.out.println("Starting...");
        a.start();
        b.start();

        // The current running thread (executing main()) blocks
        // until both workers have finished
        try {
            a.join();
            b.join();
        }
        catch (Exception ignored) {}

        System.out.println("All done");
    }
}
```

Simple Thread Example Output

Starting...
Thread-0 0
Thread-1 0
Thread-0 10000
Thread-0 20000
Thread-1 10000
Thread-0 30000
Thread-1 20000
Thread-0 40000
Thread-1 30000
Thread-0 50000
Thread-1 40000
Thread-0 60000
Thread-1 50000
Thread-0 70000
Thread-1 60000
Thread-0 80000
Thread-0 90000
Thread-1 70000
Thread-1 80000
Thread-1 90000
All done

Threading 2 (Handout #20)

- Two Threading Challenges
  - Mutual Exclusion
    - Keeping the threads from interfering with each other
    - Worry about memory shared by multiple threads
  - Cooperation
    - Get threads to cooperate
      - Typically centers on handing information from one thread to the other, or signaling one thread that the other thread has finished doing something
    - Done using join/wait/notify

Critical Section

- A section of code that causes problems if two or more threads are executing it at the same time
  – Typically as a result of shared memory that both thread may be reading or writing
- Race Condition
  – When two or more threads enter a critical section, they are supposed to be in a race condition
    - Both threads want to execute the code at the same time, but if they do then bad things will happen

Race Condition Example

```
class Pair {
    private int a, b;

    public Pair() {
      a = 0;
      b = 0;
    }
    // Returns the sum of a and b. (reader)
    public int sum() {
      return(a+b);
    }
    // Increments both a and b. (writer)
    public void inc() {
      a++;
      b++;
    }
}
```

Reader/Writer Conflict

- Case
  - thread1 runs inc(), while thread2 runs sum()
    - thread2 could get an incorrect value if inc() is half way done
    - This happens because the lines of sum() and inc() interleave
- Note
  - Even a++ and b++ are *not* atomic statements
    - Therefore, interleaving can happen at a scale finer than a single statement!
    - a++ is really three steps: read a, increment a, write a
  - Java guarantees 4-byte reads and writes will be atomic
  - This is only a problem if the two threads are touching the same object and therefore the same piece of memory!

Writer/Writer Conflict

- Case
  - thread1 runs inc() while thread2 runs inc() on the same object
    - The two inc()'s can interleave in order to leave the object in an inconsistent state
- Again
  - a++ is not atomic and can interleave with another a++ to produce the wrong result
  - This is true in most languages

Heisenbugs

- Random Interleave – hard to observe
  - Race conditions depend on having two or more threads “interleaving” their execution in just the right way to exhibit the bug
    - Happens rarely and randomly, but it happens
  - Interleaves are random
    - Depending on system load and number of processors
    - More likely to observe issue on multi-processor systems
- Tracking down concurrency bugs can be hard
  - Reproducing a concurrency bug reliable is itself often hard
  - Need to study the patterns and use theory in order to pre-emptively address the issue

Java Locks

- Java includes built-in support for dealing with concurrency issues
  - Includes keywords in order to mark critical sections
  - Includes object locks in order to limit access to a single thread when necessary
- Java designed to encourage use of threading and concurrency
  - Provides the tools needed in order to minimize concurrency pitfalls

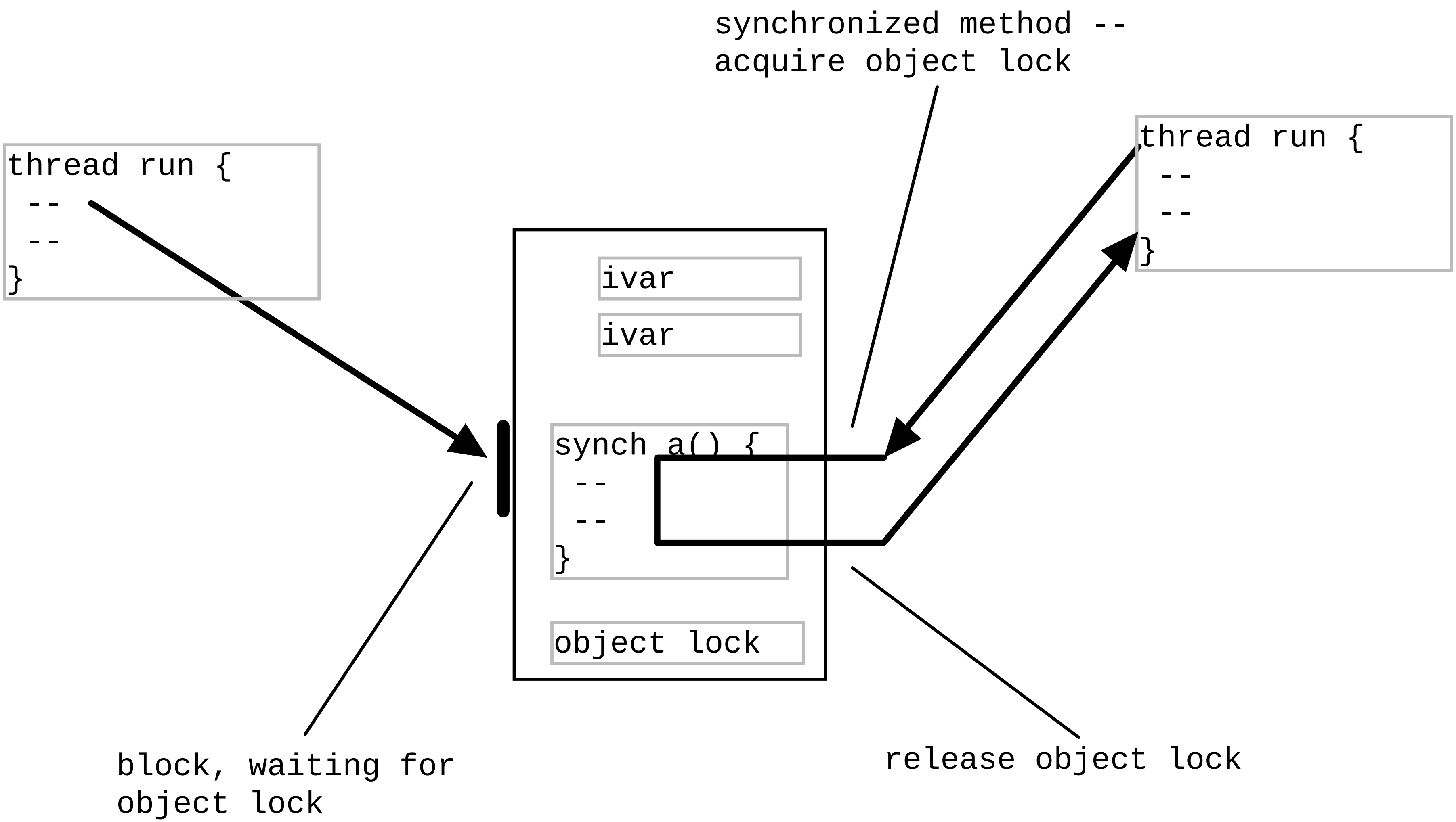
Object Lock and Synchronized keyword

- Every Java Object has as lock associated with it
- A "synchronized" keyword respects the lock of the receiver object
  - For a thread to execute a synchronized method against a receiver, it must first obtain the lock of the receiver
  - The lock is released when the method exits
  - If the lock is held by another thread, the calling thread blocks (efficiently) till the other thread exits and the lock is available
  - Multiple threads therefore take turns on who can execute against the receiver

Receiver Lock

- The lock is in the receiver object
  - Provides mutual exclusion mechanism for multiple threads sending messages to **that object**
  - Other objects have their own lock
- If a method is not sychronized
  - The thread will not acquire the lock before executing the method

Sychronized Method Picture



Synchronized Method Example

```
/*
 A simple class that demonstrates using the 'synchronized'
 keyword so that multiple threads may send it messages.
 The class stores two ints, a and b; sum() returns
 their sum, and inc() increments both numbers.

 <p>
 The sum() and incr() methods are "critical sections" --
 they compute the wrong thing if run by multiple threads
 at the same time. The sum() and inc() methods are declared
 "synchronized" -- they respect the lock in the receiver object.
*/
class Pair {
     private int a, b;

     public Pair() {
             a = 0;
             b = 0;
     }
```

Synchronized Method Example

```
    // Returns the sum of a and b. (reader)
    // Should always return an even number.
    public synchronized int sum() {
            return(a+b);
    }
    // Increments both a and b. (writer)
    public synchronized void inc() {
            a++;
            b++;
    }
}
```

Synchronized Method Example

```
/*
 A simple worker subclass of Thread.
 In its run(), sends 1000 inc() messages
 to its Pair object.
*/
class PairWorker extends Thread {
      public final int COUNT = 1000;
      private Pair pair;
      // Ctor takes a pointer to the pair we use
      public PairWorker(Pair pair) {
                this.pair = pair;
      }
      // Send many inc() messages to our pair
      public void run() {
                for (int i=0; i<COUNT; i++) {
                                pair.inc();
                }
      }
```


Synchronized Method Example

```
/*
 Test main -- Create a Pair and 3 workers.
 Start the 3 workers -- they do their run() --
 and wait for the workers to finish.
*/
public static void main(String args[]) {
        Pair pair = new Pair();
        PairWorker w1 = new PairWorker(pair);
        PairWorker w2 = new PairWorker(pair);
        PairWorker w3 = new PairWorker(pair);
        w1.start();
        w2.start();
        w3.start();
        // the 3 workers are running
        // all sending messages to the same object
```

Synchronized Method Example

```
// we block until the workers complete
try {
        w1.join();
        w2.join();
        w3.join();
}
catch (InterruptedException ignored) {}

System.out.println("Final sum:" + pair.sum());   // should be 6000
/*
 If sum()/inc() were not synchronized, the result would
 be 6000 in some cases, and other times random values
 like 5979 due to the writer/writer conflicts of multiple
 threads trying to execute inc() on an object at the same time.
*/
    }
}
```

Summary

- Today
  - Object Serialization
    - Cloning and Serializing
  - Introduction to Threading
    - Motivation
    - Java threads
      - Simple Thread Example
  - Threading 2
    - Race Conditions
    - Locking
    - Synchronized Methods
- Assigned Work Reminder
  - HW 2: Java Draw
    - Due before midnight on Wednesday, July 23rd, 2003
    - Start no later than TODAY!