Multithreading, part 2

Protecting your data

Review

Which is *not* a way to create threads in Java?

a) Use a class that implements Runnable

b) Use a class that extends Thread

c) Use a class that extends Threadable

d) Use an Executor

Review

What advantage does threading have?

a) It always makes your program faster

b) It never corrupts your data

c) It's easy to debug

d) It can improve performance, but it might corrupt your data and be hard to debug

Objectives

- Topics
  - Race conditions
  - Synchronization, locks, thread-safe data structures
- Goals: after this lecture, you will be able to
  - understand why data shared by two or more threads can become corrupted
  - describe the basic ways to protect shared data
  - write threaded programs using thread-safe techniques

Data Consistency

- If threads are independent – they work on separate data, like the subarrays above – then consistency is not a problem
  - More likely, they share data
- If data is shared by two or more threads, they must be careful updating the data. ***Race condition***: when shared data can become inconsistent (corrupted) due to updates from multiple threads.
- Even simple updates on primitive types (int, float, …) can have race conditions

Consistency, cont.

- ***Shared data*** means that two or more threads use a ***reference*** to the same thing
  - Remember that "reference" here means pointer to an object on the heap, not primitive data
- This happens when the main program creates an object and gives it to (passes it as a parameter to an overloaded constructor of) a class that implements Runnable
- Even if you increment an int inside a shared class, it could go wrong – data gets corrupted

Consistency, cont.

- Race conditions result from bad timing (on the part of the thread scheduler, not your fault), running threads in parallel, and lack of data protection (your fault!)
- It stems from the fact that when you operate on data, it has to be fetched from main memory into the cpu/core, operated on there, and written back to memory
  - This takes a small but non-zero amount of time
  - ***A thread can be interrupted in the middle of it*** – by the operating system for some good reason; you have no control over it – that's the scheduler part

Example

- Suppose two threads ***share*** this: `int count = 0;` (inside an object, because they cannot share a primitive)
- Suppose both threads execute this instruction: `count++;`
- If the threads execute ***serially*** – one after the other – then there's no problem, and `count` ends up as 2
- How could such a simple operation go wrong?

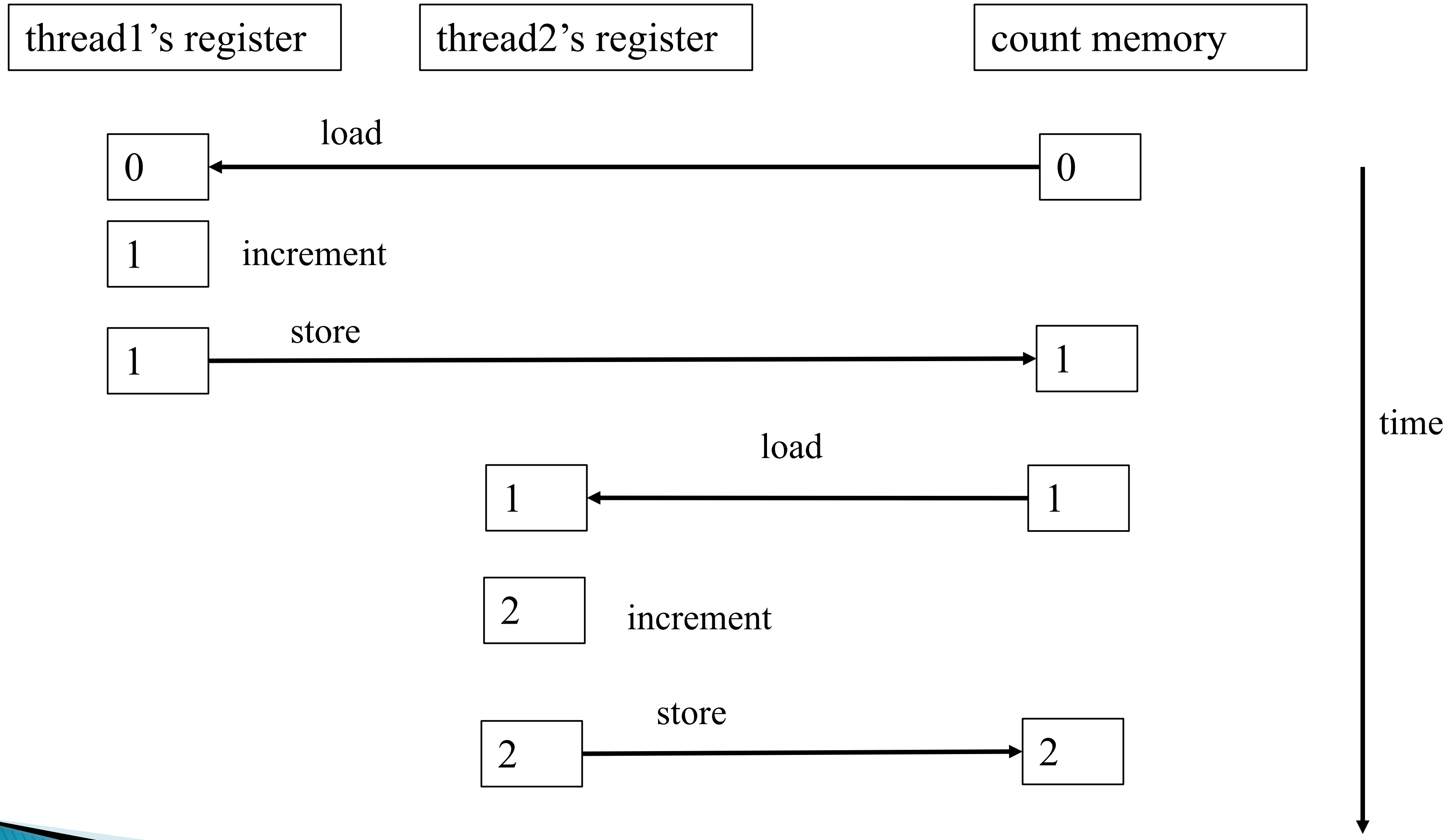
Example, cont.

- `count++` is not ***atomic*** (executes all at once or not at all) – because the bytecode looks something like this:

```
LOAD count;     // read count from main main memory into the core
INCR count;     // add one into the core
STORE count;    // write count back to main memory
```

- Each core has local memory, called registers, separate from main memory
  - The number of registers is fairly small compared to the amount of main memory
  - You cannot operate on data in memory directly (well …)

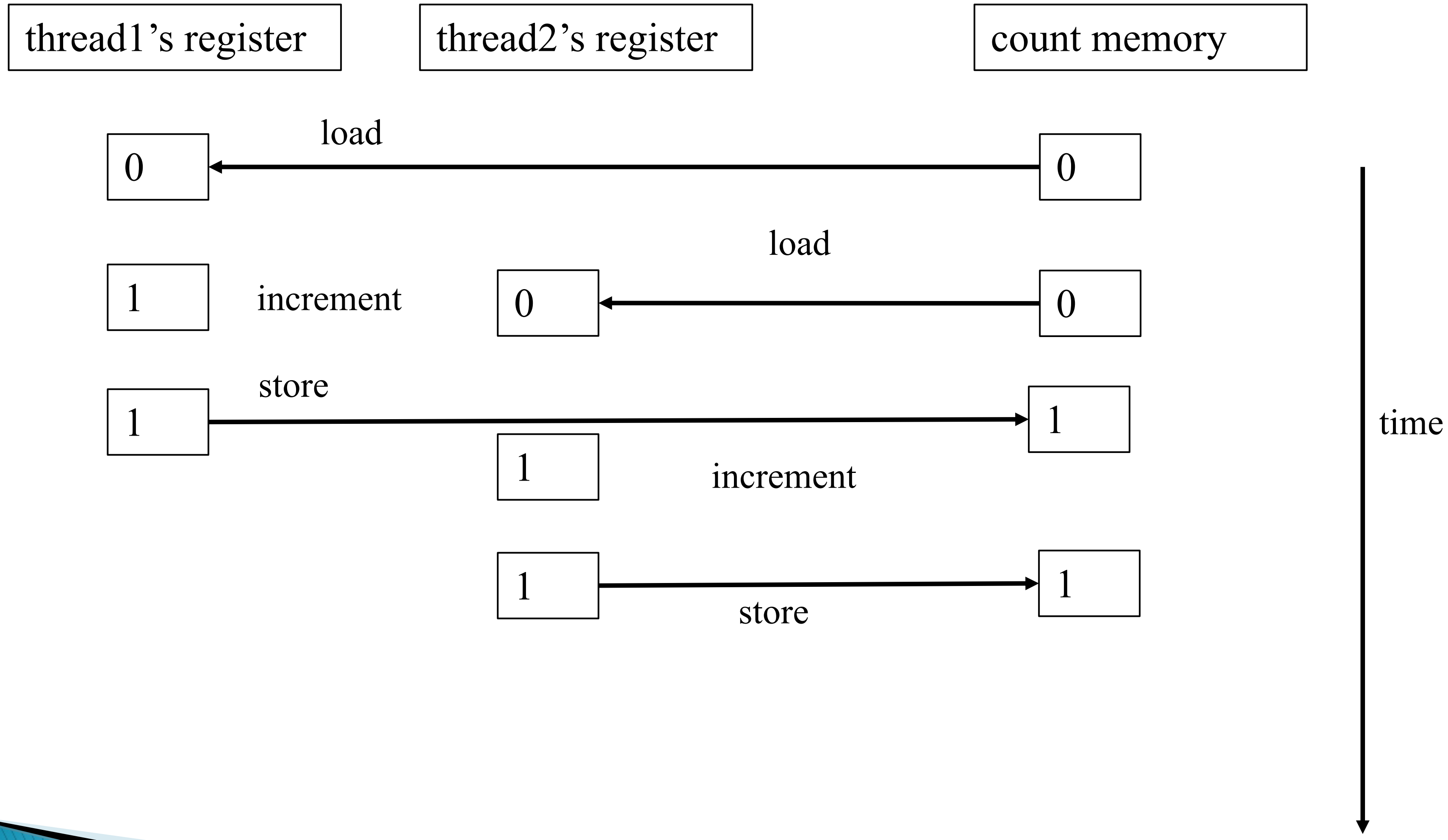
"Normal" operation



Example, cont.

- Suppose the first thread executes `LOAD count;` then `count` is 0 in main memory ***and*** in the register
- Now the second thread runs; it executes `LOAD count`; while the first thread is executing `increment`
- The first thread writes 1 back to memory
- Then the second thread writes 1 back to memory

Not so normal operation



Race Condition

- So we did `count++` twice to 0 and got 1, not 2.
  - Basically because both threads started with 0: they weren't serialized, and the second thread used "stale" data
- For larger data items, it only gets worse
- If the explanation doesn’t make sense to you, no problem, just remember this:

unprotected shared data can be corrupted by concurrent threads

Example

- `UnsafeSimpleData` contains one `int`
- That data is not thread-safe
  - We'll see how to make it thread-safe later

```
public class UnsafeSimpleData { // Wrapper for count
   private int count = 0;
   public void increment() {count++;}
   public int getCount() {return count;}
}
```

Example, cont.

- The `UnsafeCounter` class copies a reference to a `SimpleData` object containing the data
- If there's only one of these, there's no race condition

```
public class UnsafeCounter implements Runnable{
   private UnsafeSimpleData sd;
   public UnsafeCounter(UnsafeSimpleData sd) { this.sd = sd; }

   @Override
   public void run() {
      for (int i=0; i<1000; i++) { sd.increment(); }
      System.out.println("count = " + sd.getCount());
   }
}
```

Example, cont.

- main passes the *same* `UnsafeSimpleData` reference to two copies of `UnsafeCounter` – i.e., two threads
- Now we have a race condition

```
UnsafeSimpleData sd = new UnsafeSimpleData();
Thread counter1 = new Thread( new UnsafeCounter(sd) );
Thread counter2 = new Thread( new UnsafeCounter(sd) );

counter1.start();   counter2.start();
counter1.join(); counter2.join();
System.out.println( sd.getCount() );
```

both UnsafeCounter objects use sd
- i.e., shared data

More on the example

- Both copies of `UnsafeCounter` point to the same data on the heap – that's shared data. Pay attention to how that's done
- If they executed concurrently (as a thread's code), then there's a race condition on that data

Race Conditions

- In practice, race conditions are easy to overlook
- They are also very hard to debug – you can test, test, test your app, everything looks good, then you deliver the app to the customer, and the problem shows up
  - So you have to be proactive about finding race conditions

Data Consistency Techniques

- The `count++` example is a very low-level view, but that's where the problem is
- The solution is at a much higher level: lock the data

1. Make getters/setters **synchronized** (and possibly other methods)
   - Implicitly locks on entry, unlocks on exit
2. Use thread-safe data structures (that use synchronized methods or locks)
3. Use explicit locks

Technique #1: `synchronized` Code Blocks

- The **synchronized** keyword can be applied to methods or to blocks of code (curly-bracketed)
- But not to an entire class

```
public synchronized void computeThisThing( ) { … }
```

Note: *must* be an object that is *shared* by the threads

or, alternatively:

```
synchronized (object) {
   … some code …
}
```

Example: Synchronized

- The `SafeSimpleData` class protects its data by adding **synchronized** to its methods (not the data)

```
public class SafeSimpleData { // Wrapper for count
   private int count = 0;
   public synchronized void increment() {count++;}
   public synchronized int getCount() {return count;}
}
```

Methods that work on the data are synchronized *EXCEPT* for constructors

Example, cont.

- Set up the thread class the same way: it uses the data class

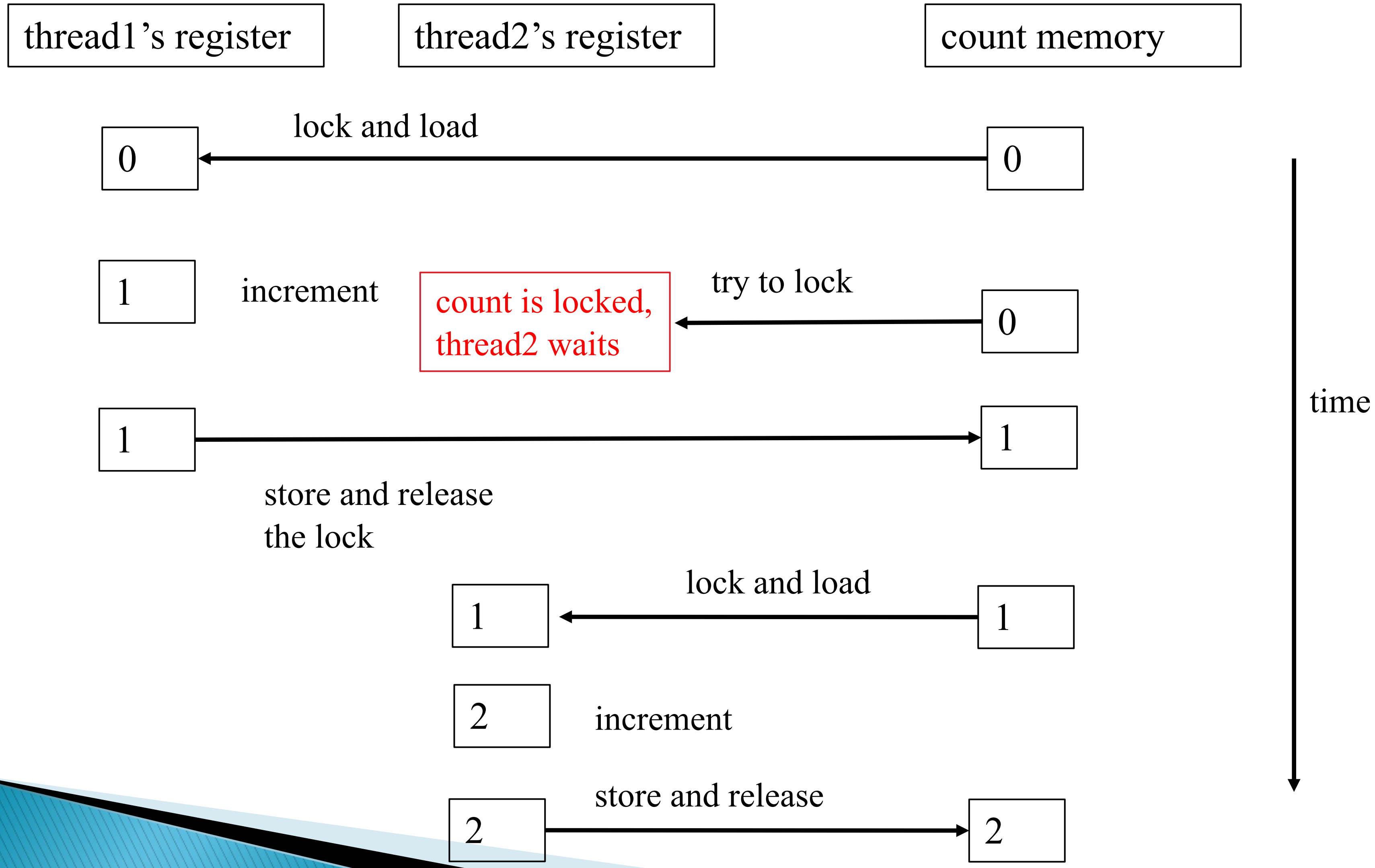
```
public class SafeCounter implements Runnable{
   private SafeSimpleData sd;
   public SafeCounter(SafeSimpleData sd) { this.sd = sd; }

   @Override
   public void run() {
      for (int i=0; i<1000; i++) { sd.increment(); }
      System.out.println("count = " + sd.getCount());
   }
}
```

Example, cont.

- Now the threads take turns
  - each `SafeSimpleData` object has a ***lock***, initially open
  - the first thread to access one of the methods ***acquires*** the lock – i.e. locks the code
  - if a second thread tries to access the object, it waits until first thread ***releases*** the lock

```
SafeSimpleData sd = new SafeSimpleData();
Thread counter1 = new Thread( new SafeCounter(sd));
Thread counter2 = new Thread( new SafeCounter(sd));
counter1.start();   counter2.start();
counter1.join(); counter2.join();
System.out.println( sd.getCount() );
```

Synchronized Data



More on the example

- Both copies of `SafeCounter` point to the same data on the heap – that's shared data, just like before
- If they executed concurrently (as a thread's code), there's no race condition, because we synchronized access to the data
  - this ***serializes*** access to the data

More, cont.

- Note that this does ***not*** imply taking turns - the first thread might run a loop many times before the second one gets a chance, or vice versa – we have no control over the scheduler, we only have control over the lock on the data
- Alternate version: synchronize on something else

```
public void increment() {
    synchronized(this) {count++;}
}
```

Technique #2: Thread Safe Data Structures

- Some Java data structures have built-in synchronization
- These are called ***thread safe***, because usage by multiple threads will not corrupt their data
- For example, `ArrayList<T>` is not thread safe
  - But if your program doesn't use threads, or the `List` is not shared, use `ArrayList` to avoid the overhead of locking – synchronization is not free
  - Check the docs to be sure

This is not thread-save

- Example: Don't do this, it's not safe

```
public static void main( ) {
    ArrayList<String> mylist = new ArrayList<>();
    ... fill mylist with data ...
    Thread t1 = new Thread( new MyTask(mylist) );
    Thread t2 = new Thread( new MyTask(mylist) );
    t1.start( );
    t2.start( );
}
```

both get the same unsafe object

Thread Safe Data Structures

- `java.util.concurrent` contains thread safe data structures like `ArrayBlockingQueue` and `ConcurrentHashMap`
- Basic `Collection` classes are ***not*** thread safe: `ArrayList`, `HashMap`, …
- But they can be made thread safe by ***wrapping*** them:

```
List<E> safeArrayList =
Collections.synchronizedList(new ArrayList<E>());
```

wrapping it to make it thread-safe

Thread Safe Data Structures, cont.

- `Vector` is a thread safe data structure similar to `ArrayList`
- So it can be used when shared between threads

"As of the Java 2 platform v1.2, this class was retrofitted to implement the List interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector." – https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

Example

- `ArrayBlockingQueue<>` is thread-safe and ***bounded*** – here, a maximum of 5 elements can be queue.
- `LinkedBlockingQueue<>` is thread-safe and ***unbounded***
- Example uses `BlockingQueue` as shared data

```
private BlockingQueue<Integer> blockingQueue = new
      ArrayBlockingQueue<Integer>(5);

Thread producer = new Thread( new NumberMaker( blockingQueue ) );
Thread consumer = new Thread(  new QueueProcessor( blockingQueue ) );

producer.start(); consumer.start();
producer.join(); consumer.join();
```

set to 5 items

Example, cont.

```
public class NumberMaker implements Runnable {
   private BlockingQueue blockingQueue;

   public NumberMaker(BlockingQueue blockingQueue) {
       this.blockingQueue = blockingQueue;
   }

   @Override
   public void run() {
      for (int i = 0; i < 20; i++ ) {          // put 20 Integer's
         try {
            blockingQueue.put(new Integer(i));
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

   }
}
```

Example, cont.

```
public class QueueProcessor implements Runnable {
   private BlockingQueue blockingQueue;

   public QueueProcessor(BlockingQueue blockingQueue) {
      this.blockingQueue = blockingQueue;
   }


   @Override
   public void run() {
      Integer i = null;
      try {
         do {                                    // take 20 Integer's
            i = (Integer)blockingQueue.take();
            System.out.println("Got " + i);
         } while (i != 20);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }

}
```

More on the example

- The `Runnable` classes use the shared `blockingQueue` to safely move data
- Calls to `put()` and `take()` are synchronized, as are other methods
- But watch out: if you make consecutive method calls, each *individual* call is synchronized, but the sequence of calls is not – i.e., a race condition
  - If you need to do that, synchronize it some other way

```
// This is bad:
if ( !blockingQueue.contains(thing)  ) {
  blockingQueue.add(thing);  // Race condition
}
```

Example

- The **AtomicInteger** class is also thread safe
- It's already a class, so it doesn't need any more packaging
- Its methods are not straightforward …
  - for example, `incrementAndGet()` adds one and returns the new value, instead of just incrementing

```
AtomicInteger sd = new AtomicInteger();  // Don't need an enclosing class

Thread counter1 = new Thread( new SafeCounter(sd)); // Need to change SafeCounter
Thread counter2 = new Thread( new SafeCounter(sd)); //   to handle AtomicInteger
counter1.start();   counter2.start();
counter1.join(); counter2.join();
System.out.println(sd.get() );
```

volatile

- The **volatile** keyword is related to thread safety, but …
- … it does *NOT* make a data item thread-safe
- It tells the JVM to automatically write the variable back to memory when it changes value – i.e., it's about visibility, not races – so there's a race condition here
- Only use it for ”small” data – like a boolean flag

```
static volatile boolean done = false;

... then in some thread code:

if (done) { … finish up …}
```

Technique #3: Explicit Locks

- Java has several kinds of locks; the basic one is **ReentrantLock**.
  - Among its methods: `lock()` and `unlock()`
- So instead of declaring `SafeSimpleData.increment()` as `synchonized`, add a `ReentrantLock` as member data and lock it yourself:

```
 private ReentrantLock mylock = new ReentrantLock();
mylock.lock();
   // code here
mylock.unlock();
```

Note: lock must be shared data, too

```java
import java.util.concurrent.locks.*;
public class SafeSimpleData {  // Alternate implementation
   private int count = 0;
   private ReentrantLock mylock = new ReentrantLock();
   public void increment() {
      mylock.lock();
      try { count++; } // try block is required
      finally {
         mylock.unlock();  // Always unlock in finally
      }
   }
   // similarly for getCount() ...
}
```

More on the example

- This version is more awkward – lower level programming – than using `synchronized`
- It is used when you only need to lock part of your code, not an entire method
  - Lock the part that uses shared data – called the ***critical region*** – and leave any safe parts unlocked. This is a (small) optimization: the less stuff you lock, the more chance for parallelism, because you're not blocking threads from running

GUI Threading

- A JavaFX application runs on a thread started by `launch()`
- If you have any tasks that take a long time, the GUI will be unresponsive until that task completes
  - Makes users unhappy
- Instead, you can create another thread for the task to run in
- This keeps the GUI responsive – the main UI thread will respond to button clicks, etc.
- Writing your own task class is possible, but there’s a built-in threading mechanism called ***Task<T>***
  - It implements `Runnable`

GUI Threading, cont.

- `Task<>` still needs to be put in a Thread, then call `start()`, which invokes `call( )`
- When the Task is finished, it sends a `WorkerStateEvent` that can be handled by `Task.setOnSucceeded()` by overriding its `handle()` method
  - The pattern here is common, but the details are JavaFX-specific: other GUIs will have other ways to do all this
- This part is done on the UI thread after the Task has finished

```
drawHamburg.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        // Task<> is built in to JavaFX, implements Runnable,
        //   requires override of call() instead of run()
        Task<Void> task = new Task<Void>() {
                @Override protected Void call() throws Exception {
                // Load the image file
                File file = new File("hamburg.png");
                ... more code to load the image ...
        }

        task.setOnSucceeded(new EventHandler<WorkerStateEvent>() {
                @Override
                public void handle(WorkerStateEvent event) {
                        System.out.println("in setOnSucceeded()");
                        imageView.setImage(image);
                }
        });
       Thread t = new Thread(task);
       t.start();
    }
});
```

More on the example

- This is a simple example – the file is local and loading it doesn’t take long, so a long operation is simulated with `sleep()`
- The other GUI buttons remain responsive during the operation
- The non-threaded version stalls the GUI until it finishes
- Other GUI’s have different versions of the same thing - for example, Android has `AsyncTask<>`