Safe Threads

```
public class PrimeCounter implements Runnable {

  private int from_;
  private int to_;
  private int count_;
  ...
  @Override
  public void run () {
    for ( int n = from_ ; n < to_ ;
          n++ ) {
      if ( isPrime(n) ) {
        count_++;
      }
    }
  }

  public int getCount () {
    return count_;
  }
}
```

```
for ( int i = 0 ; i < n ; i++ ) {
  tasks[i] = new PrimeCounter(i * chunk,
                              (i + 1) * chunk);
  threads[i] = new Thread(tasks[i],"primes-" + i);
  threads[i].start();
}

int total = 0;
for ( int i = 0 ; i < n ; i++ ) {
  for ( ; threads[i].isAlive() ; ) {
    try {
      threads[i].join();
    } catch ( InterruptedException e ) {}
  }
  total += tasks[i].getCount();
}
```

- the threads split the work – each thread counts primes in part of the range
- each thread has its own `PrimeCounter` object and thus its own `count_` – no shared variables between counter threads
- `main` only reads `count_` (via `getCount()`) after that thread is done
  - no overlap between reading and writing

Key Points – Thread Safety

Problems can arise when there are *shared variables* whose values are changed.

- the compiler and hardware may cache values, delay writes, and reorder memory operations, affecting the *visibility* of writes
  - a thread may not notice the new value after another thread changes it, or may see changes late or out of order
- *race conditions* can lead to an invalid result because of the timing of when different threads run
  - the scheduler can pause a thread between any two instructions, letting another thread run in the interim
  - instructions are effectively interleaved in an unpredictable order when threads run simultaneously on separate cores

Visibility

- for a speed, the compiler and hardware may cache values, delay writes, and reorder memory operations
  - a thread may not notice the new value after another thread changes it, or may see changes late or out of order
- spinner demo

`volatile`

```
public class Spinner implements Runnable {
  private volatile boolean shutdown_;
  private int spins_;
  …
}
```

- ensures that –
  - a read always sees the latest write from any thread – no caching
  - there's a happens-before relationship for memory operations
    - everything a thread did before writing a `volatile` variable is visible to a thread that then reads it
  - the compiler won't optimize away reads
  - single reads and writes (`x = 10`) are atomic
    - (Java specs don't guarantee for `long` and `double` without `volatile`)
- but –
  - it applies only to the variable, not the object the variable refers to
  - updates (`x = x+10`, `x++`) are *not* atomic
- use it for a flag or a value that one thread writes and others read
  - typically primitive types or immutable types like `String`

`volatile` or Not?

```
SlowCountdown2 countdown1 = new SlowCountdown2(10,"");
SlowCountdown2 countdown2 = new SlowCountdown2(20,"   ");

Thread thread1 = new Thread(countdown1);
Thread thread2 = new Thread(countdown2);
```

```
System.out.println("shutting down " + thread2.getName());
countdown2.shutdown();
```

`start()` ensures visibility of what happened before – so the new threads see what happened in the constructor

after the constructor, `indent_` is only ever read – safe

`value_` is changed after the constructor, but it is only used in `run()` and only one thread executes `run()` on a given `SlowCountdown2` object – safe

`shutdown_` is set by `shutdown()` and read by `run()` – different threads, needs `volatile`

`thread_` is set by `run()` and read by `shutdown()` – different threads, needs `volatile`

`volatile` applies only to the variable `thread_`, not the contents of the `Thread` object – that's what we need here

`thread_.interrupt()` will throw a `NullPointerException` if `shutdown()` is called before `run()` begins – should have `if ( thread_ != null ) … }` in `shutdown()`

```
public class SlowCountdown2 implements Runnable {

  private int value_;
  private String indent_;

  private volatile boolean shutdown_;
  private volatile Thread thread_;

  public SlowCountdown2 ( int value, String indent ) {
    value_ = value;
    indent_ = indent;
    shutdown_ = false;
    thread_ = null;
  }

   * (non-Javadoc)
  @Override
  public void run () {
    thread_ = Thread.currentThread();
    for ( ; value_ >= 0 && !shutdown_ ; value_-- ) {
      System.out.println(indent_ + value_);
      try {
        Thread.sleep(1000);
      } catch ( InterruptedException e ) {
        System.out.println(indent_ + "interrupted!");
        if ( shutdown_ ) { break; }
      }
    }
    System.out.println(indent_ + "done!");
  }

  public void shutdown () {
    shutdown_ = true;
    thread_.interrupt();
  }

}
```


Race Conditions

- garden demo
- ATM demo

Race Conditions

the exact timing of the steps during simultaneous withdrawals can lead to invalid results

person A initiates a withdrawal from an ATM –

withdraw $100 from account 123
- balance = getBalance(123) → 1000

person B initiates a withdrawal from a different ATM –

withdraw $100 from account 123
- balance = getBalance(123)
- setBalance(123,balance-100)
- dispense $100 cash

1000 ← setBalance(123,900)

$900

- setBalance(123,balance-100)
- dispense $100 cash

setBalance(123,900)

$900

if person A's withdrawal happens first – or if person B's withdrawal happens first – the final balance is (correctly) $800

Race Conditions

- a *race condition* is a situation where the exact timing of the execution of different threads can lead to invalid results
  - a *valid* result is one consistent with the operations happening one at a time, with each finishing operation before the next starts
    - which order is not stipulated, just that the result can arise from *some* order – different orders may lead to different results
  - an *invalid* result isn't consistent with any such order
- race conditions can occur when there are multi-step operations involving shared data
- testing can't reliably find race conditions
  - they cause problems only when the timing is exactly wrong – a test can pass many times and fail only rarely
  - you must reason about the code to ensure there's no problem

Atomic Operations

- an *atomic* operation happens as one indivisible step
  - to other threads, either it completed fully or it never started – they don't ever see it partway through
- one line of Java code is not necessarily one step
  - single reads and writes (`x = 10`) *are* (mostly) atomic
    - (Java specs don't guarantee for non-`volatile` `long` and `double`)
  - updates (`x = x+10`, `x++`) are *not* atomic
  - `account_.setBalance(account_.getBalance()-10)` is *not* atomic
- any non-atomic operation can be subject to race conditions if the shared data can be written by another thread

Common Races

- two common kinds of race conditions
  - non-atomic updates (read, change, write back) – another thread's update can be lost
    - e.g. garden counter, ATM withdrawal
  - check-then-act – another thread can make a change that invalidates the check between the check and the act
    - e.g. withdrawing only if the balance is enough

```
if ( balance_ > amount ) {
  balance_ -= amount;
}
```


Mutual Exclusion

- race conditions are fixed with *mutual exclusion*
  - only one thread a time may be in a *critical section*
- achieves correct semantics – one thread must exit the critical section before another can enter
  - result is consistent with some ordering
- does not guarantee any particular ordering, just that the threads don't overlap
- `synchronized`
  - works like a lock – only one thread can hold a given lock at a time
    - before using a shared resource, claim the lock
    - other threads that want the lock wait (`BLOCKED`) until the lock is released
  - in Java, any object can be a lock

Mutual Exclusion With `synchronized`

- two forms
  - a `synchronized` block
    - to achieve mutual exclusion, all accesses of a particular variable/object must be protected with the same lock object
    - use when relatively small chunks of code need mutual exclusion
  - a `synchronized` method
    - equivalent to enclosing the method body with `synchronized ( this ) { … }`
    - locks the whole object – all methods in the same class use the same lock
    - use when any access of the object requires mutual exclusion
- "all accesses" includes reads
  - a getter that reads without the lock may see a stale value, so if other threads can change the value, use `synchronized` (preferred) or `volatile`

Mutual Exclusion With `synchronized`

- `synchronized` block
  - ensures no other threads can execute this code on the same `Account` object at the same time
    - two threads working with different accounts don't block each other
  - only need mutual exclusion when the balance is being updated, not the entire loop
  - `account_` isn't shared between threads but the `Account` object `account_` refers to is

```
public void run () {
  for ( int i = 0 ; i < 50 ; i++ ) {
    synchronized ( account_ ) {
      int balance = account_.getBalance();

      account_.setBalance(balance - 10);
    }
  }
  System.out.println("total withdrawn: $500");
}
```


Mutual Exclusion With `synchronized`

- `synchronized` methods
  - ensures no other threads can execute methods on the same object at the same time
  - constructor doesn't need `synchronized` – impossible for another thread to access object before constructor finishes
  - `getValue()` needs `synchronized` to ensure visibility of changed values

```
public class SynchCounter implements Counter {

  private int count_;

  public SynchCounter () {
    count_ = 0;
  }

  /* (non-Javadoc)
  @Override
  public synchronized int getValue () {
    return count_;
  }

  /* (non-Javadoc)
  @Override
  public synchronized void increment () {
    int value = count_;

    count_ = value + 1;
  }

  public synchronized void decrement () {
    int value = count_;

    count_ = value - 1;
  }
}
```


Demo

- synchronized garden, ATM

Mutual Exclusion Must Be Mutual

- demo
  - ATM half synchronized

Enforcing Mutual Exclusion

- a `synchronized` block is like posting a “do not enter” sign on a closed door
  - it keeps others out only if they read the sign
- to ensure mutual exclusion, lock the door
  - make data only accessible through `private` instance variables
  - the only methods to access the data are `synchronized` / contain `synchronized` blocks
- example – `BankAccount`



  - provide `setBalance`, expect callers that implement withdrawal or deposit operations to protect `getBalance`/`setBalance` sequence



  - provide `withdraw`, `deposit` with `synchronized` `getBalance`/`setBalance` sequence