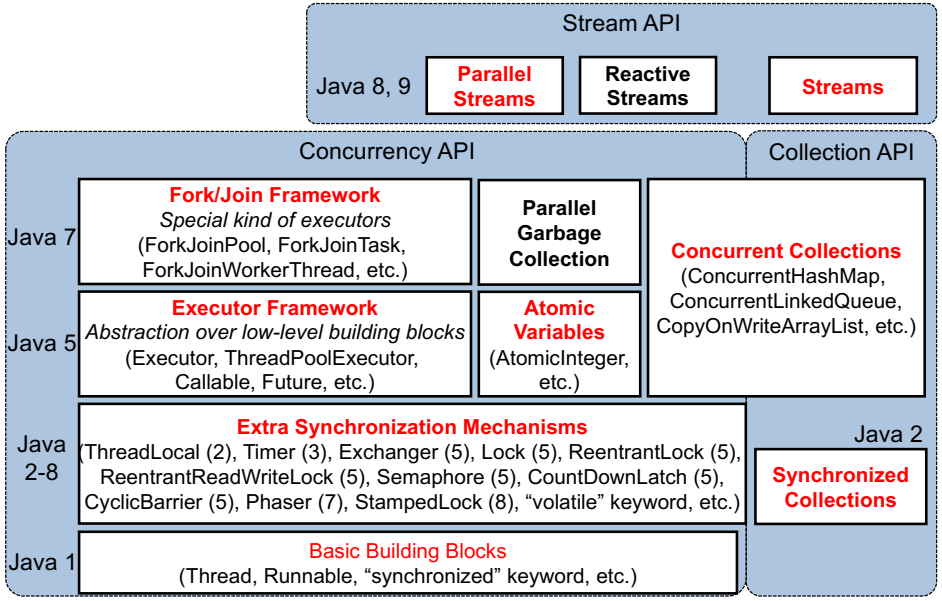
Optimistic Locking with Read-Write Locks

Concurrency API in Java

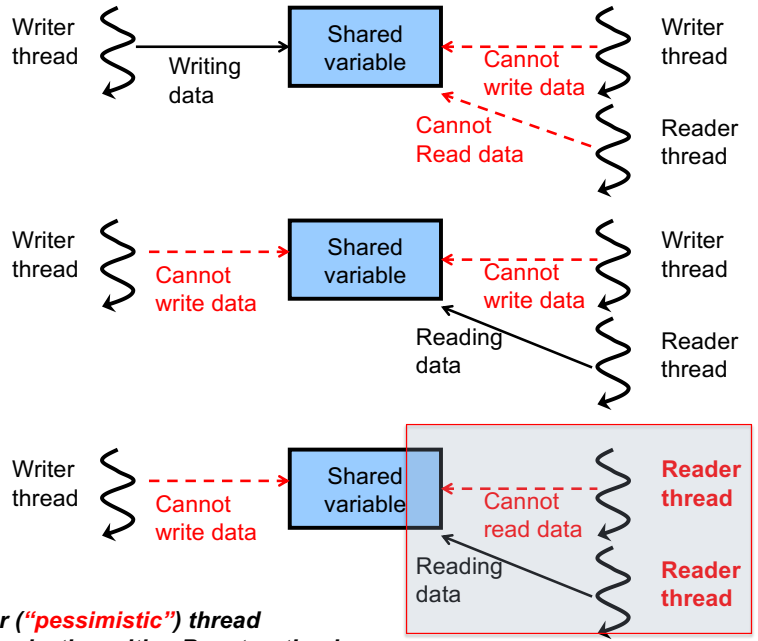


Read-Write Locks

- Regular lock (`ReentrantLock`)
  - To avoid race conditions by guarding a variable shared by multiple threads.
- Read-Write lock
  - A slight extension to `ReentrantLock`
  - A bit more *optimistic* than a regular lock to seek performance improvement.
    - `java.util.concurrent.locks.ReentrantReadWriteLock`

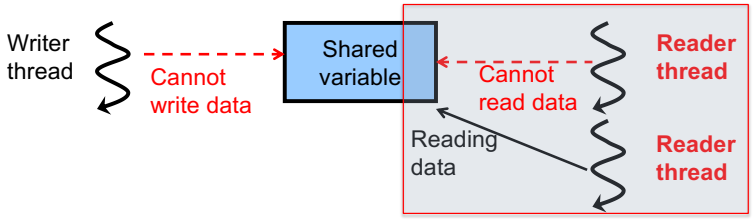
Room for Performance Improvement?

- Thread synchronization is often computationally expensive.
  - It takes some time to acquire/release a lock.
  - A thread does nothing while it is in the Blocked state.
- Where to gain performance improvement?
  - When you have multiple "reader" threads that read data from a shared variable, do we have to mutually exclude them with a lock?
    - No, as far as the value of the shared variable never changes.
    - We can be *optimistic* NOT to mutually exclude "reader" threads.

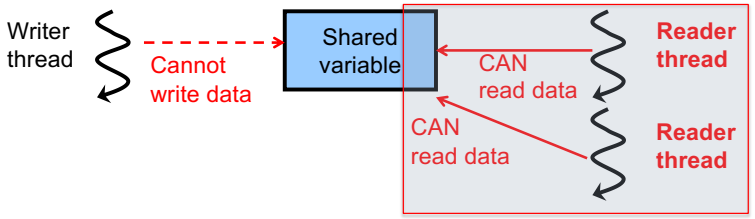


Regular ("pessimistic") thread synchronization with a ReentrantLock

Do we really need to mutually exclude these readers?



"Pessimistic" thread synchronization with a ReentrantLock



"Optimistic" thread synchronization with a ReentrantReadWriteLock

ReentrantReadWriteLock

```
public class ReentrantReadWriteLock implements ReadWriteLock{

    public class ReentrantReadWriteLock.ReadLock
        implements Lock{}

    public class ReentrantReadWriteLock.WriteLock
        implements Lock{}

    public ReentrantReadWriteLock.ReadLock readLock(){}
    public ReentrantReadWriteLock.WriteLock writeLock(){}
}
```

- Provides two locks
  - As inner *singleton* classes
    - Both implement the `Lock` interface.
  - `ReadLock` for reader threads to read data from a shared variable.
  - `WriteLock` for writer threads to write data to a shared variable.
- Provides factory methods for the two locks: `readLock()` and `writeLock()`.
  - C.f. `Singleton.getInstance()` in CS680

- A reader can acquire a read lock even if it is already held by another reader,
  - **AS FAR AS** no writers hold a write lock.
- Writers can acquire a write lock **ONLY IF** no other writers and readers hold read/write locks.

Can a thread acquire…? \ When another thread holds …?	ReadLock	WriteLock
ReadLock	**Yes**	No
WriteLock	No	No

This turns to be "No" if you use a regular lock.

An Example Optimistic Locking

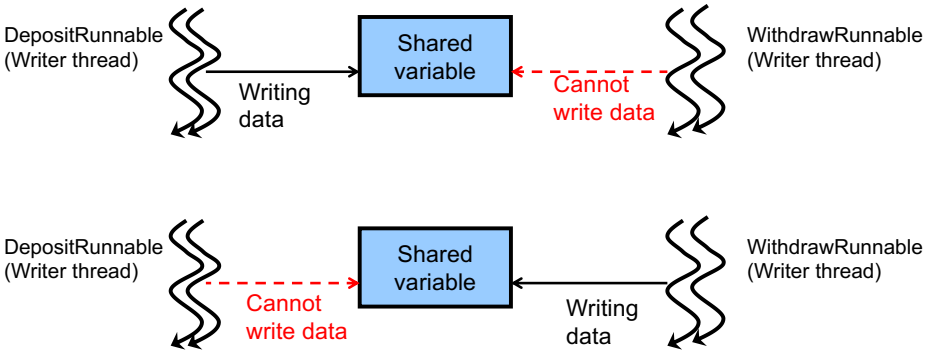
- ```
  int i;          // shared variable
  ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
  ```
- For reading data from the shared variable:
  - ```
    rwLock.readLock().lock();
    System.out.println(i);
    rwLock.readLock().unlock();
    ```
- For writing data to the shared variable
  - ```
    rwLock.writeLock().lock();
    i++;
    rwLock.writeLock().unlock();
    ```

`ReadLock` and `WriteLock`

- Work similarly to `ReentrantLock`.
  - Support nested locking and thread reentrancy.
  - Support interruption via `Thread.interrupt()`.
- `WriteLock`
  - Returns a `Condition` object when `newCondition()` is called.
- `ReadLock`
  - Throws an `UnsupportedOperationException` when `newCondition()` is called.
    - Reader threads never need condition objects.
    - Readers threads never call `signal()` and `signalAll()`.

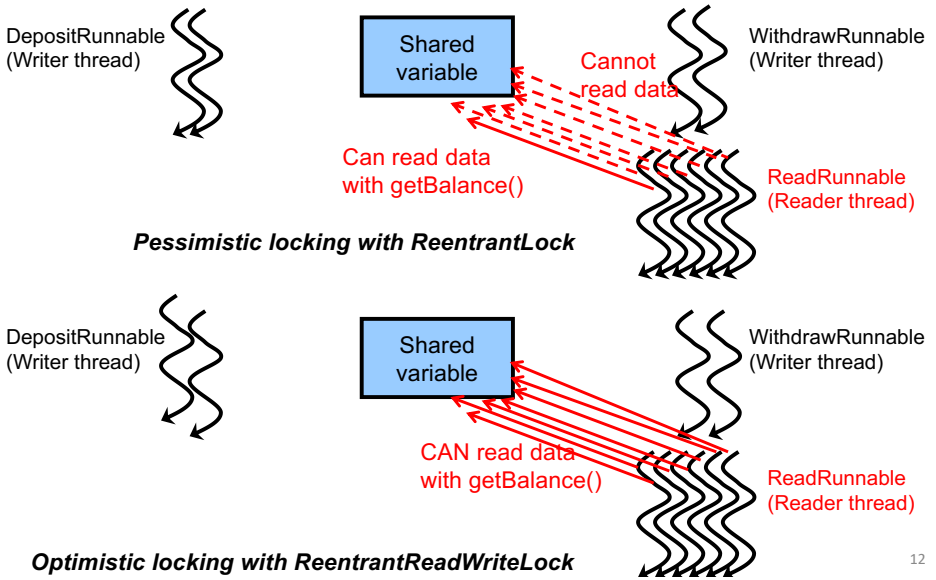
Sample Code

- ThradSafeBankAccount2



Always need regular ("pessimistic") locking with ReentrantLock for writer threads

- ThradSafeBankAccount3 and ThradSafeBankAccount4



Pessimistic locking with ReentrantLock

Optimistic locking with ReentrantReadWriteLock
    ```

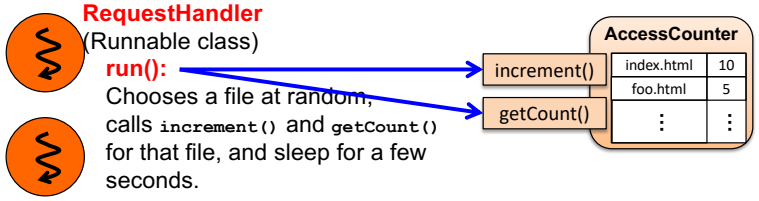
- ThradSafeBankAccount3
  - 43 msec
- ThradSafeBankAccount4
  - 33 msec
    - 23% (10/43) faster
      - thanks to optimistic locking

When to Use Optimistic Locking?

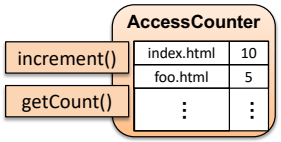
- When many reader threads run.
- When reader threads run more often than writer threads.
- When a read operation requires a long time to be completed.

HW 15

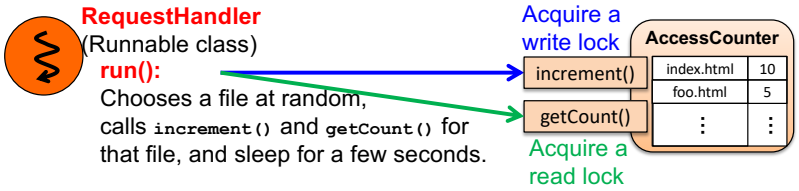
- Recall a previous HW to implement a concurrent access counter, assuming the development of a web server
- **`AccessCounter`**
  - Maintains a map that pairs a relative file path and its access count.
    - Assume `java.util.HashMap<Path, Integer>`
  - **`void increment(Path path)`**
    - accepts a file path and increments its access count.
  - **`int getCount(Path path)`**
    - accepts a file path and returns its access count.



- **`AccessCounter`**'s **`increment()`** and **`getCount()`** need to perform thread synchronization.
  - **`increment()`**
    - lock.lock();
      if(A requested path is in AC){
        increment the path's access count. }
      else{
        add the path and the access count of 1 to AC. }
      lock.unlock();
  - **`getCount()`**
    - lock.lock();
      if(A requested path is in AC){
        get the path's access count and return it. }
      else{
        return 0. }
      lock.unlock();



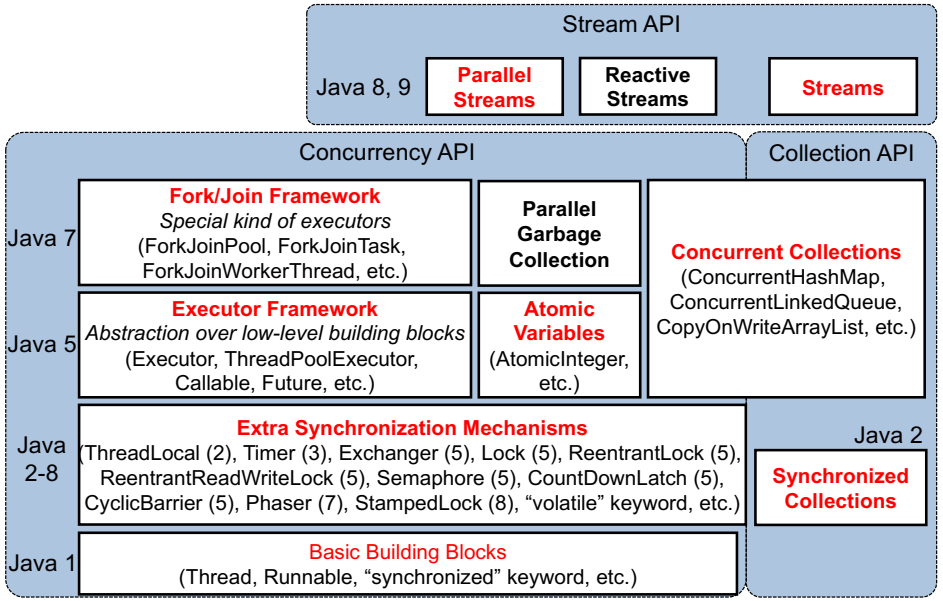
- Replace `ReentrantLock` with `ReentrantReadWriteLock` in `AccessCounter`
  - `increment()`
    - `rwLock.writeLock().lock();`
      if(A requested path is in AC){
        increment the path's access count. } // Write
      else{
        add the path and the access count of 1 to AC. } // Write
      `rwLock.writeLock().unlock();`
  - `getCount()`
    - `rwLock.readLock().lock();`
      if(A requested path is in AC){
        get the path's access count and return it. } // Read
      else{
        return 0. }
      `rwLock.readLock().lock();`



- As always, call unlock() in the finally clause, and call lock() before the try clause.
- Deadline: November 20 (Tue) midnight

Thread-Specific Storage (TSS)

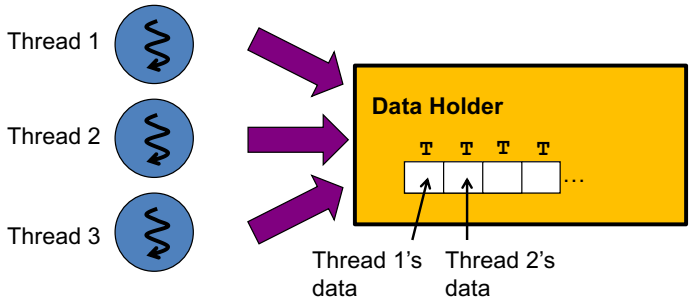
Concurrency API in Java



Thread-Specific Storage (TSS)

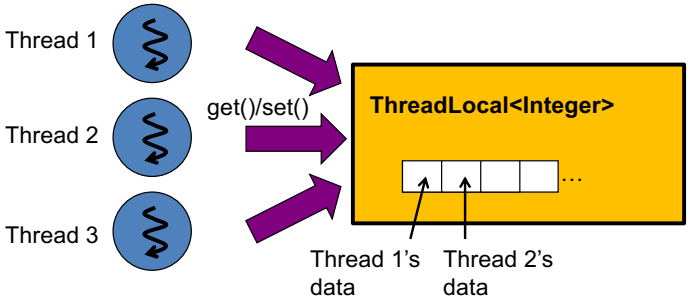
- Storage that is allocated/reserved per thread.
  - The storage is NOT accessible by other threads.
  - Implemented in `java.lang.ThreadLocal<T>`

An Example Scenario

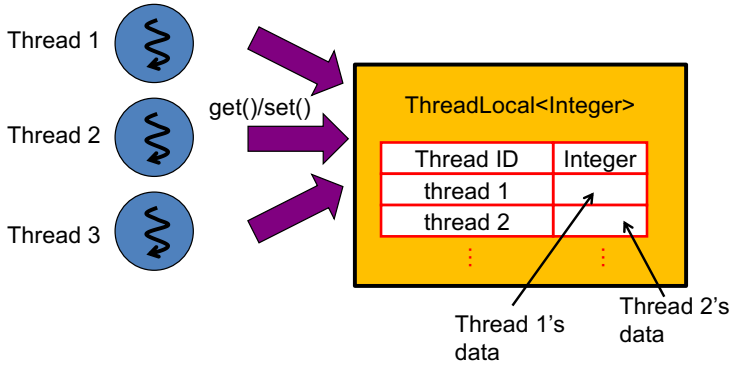


- Different threads
  - generate different data of the same type (**T**)
  - store them in a data holder
  - read them from the data holder.
- Need to guard the data holder.
  - Locking required.

With `ThreadLocal<T>`

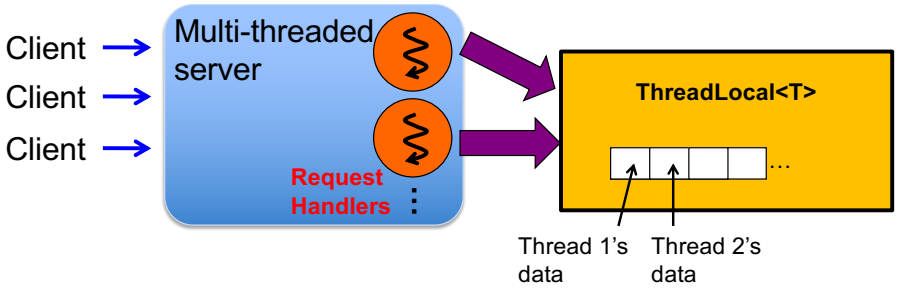


- Use `ThreadLocal` if each data is generated and accessed only by a particular thread.
  - Locking is encapsulated in `ThreadLocal`
  - No locking necessary in client code!



- `ThreadLocal` allows threads to access data through their (thread) IDs.
  - Each thread cannot access any data generated and maintained by the other threads.

TSS in a Concurrent Web Server



- Each request hander (thread):
  - Parses an incoming HTTP request, retrieves a requested file, increments its access count, logs the file access, etc. etc. and returns the requested file
  - May need customer info (e.g. customer ID) from a browser cookie to display some personalized content (e.g. shopping cart items)
  - May need client-specific information (e.g., client OS name and browser name) to display some client-specific content.