Concurrency

Thread Safety

Overview

- Data race demo

- Solutions:
  - Atomic classes from the API
  - *synchronized* keyword
  - *Lock* interface

- DataRace.java

Atomic classes from the API

- An “atomic” operation is indivisible.

- We cannot guarantee that a thread will stay running throughout the atomic operation but we can guarantee that even if the thread moves in and out of the running state, no other thread can come in and act on the same data.

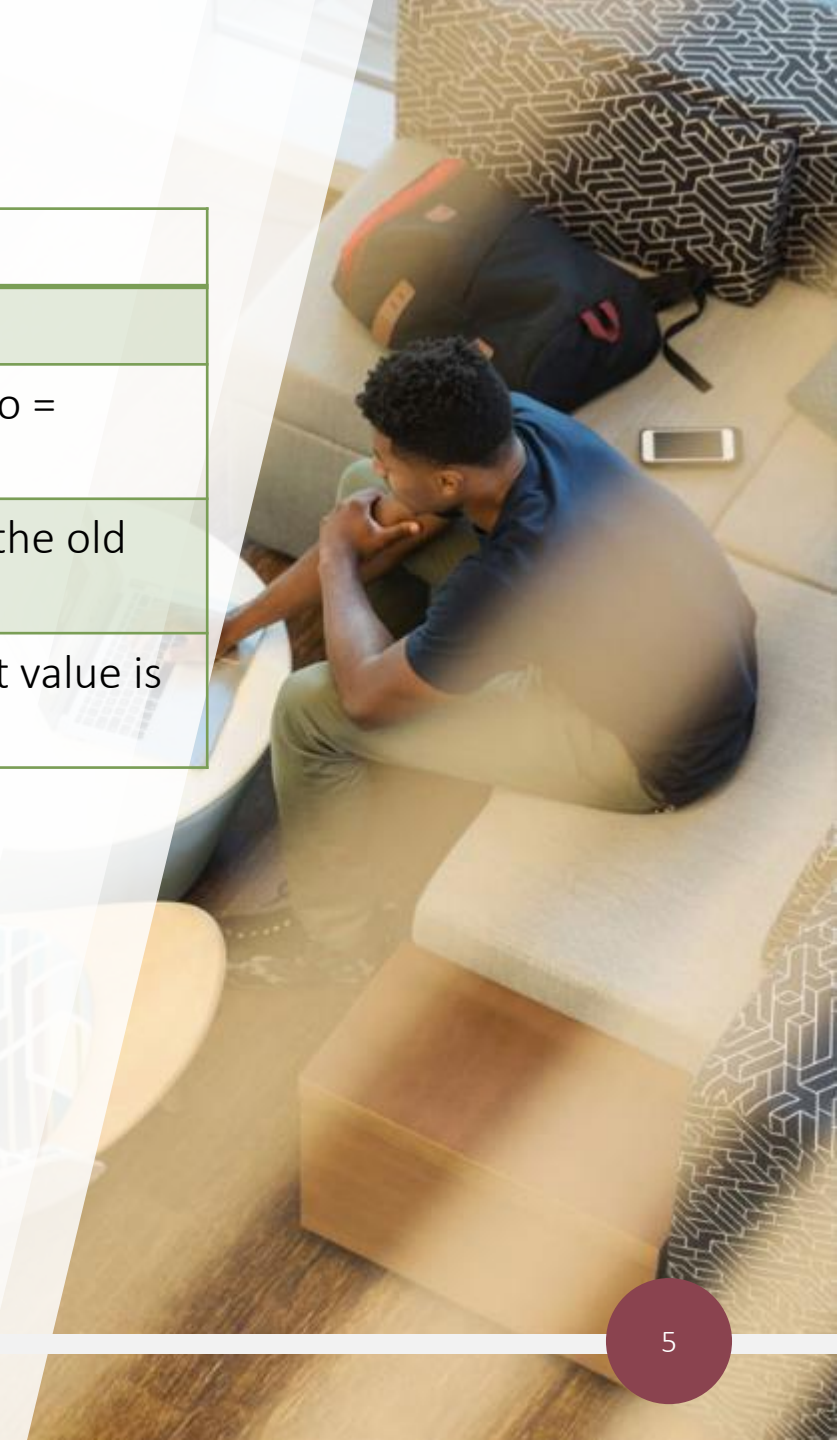
- *AtomicInteger*, *AtomicLong*, *AtomicBoolean*

Popular atomic methods

Method name	Description
get()	returns the current value
set(newValue)	sets the value to ‘newValue’; equivalent to = operator
getAndSet(newValue)	sets the value to ‘newValue’ and returns the old value
compareAndSet(expectedValue, newValue)	sets the value to ‘newValue’ if the current value is == to ‘expectedValue’

Numeric classes only

Method name	Description
incrementAndGet()	pre-increment i.e. ++x
getAndIncrement()	post-increment i.e. x++
decrementAndGet()	pre-decrement i.e. --x
getAndDecrement()	post-decrement i.e. x--



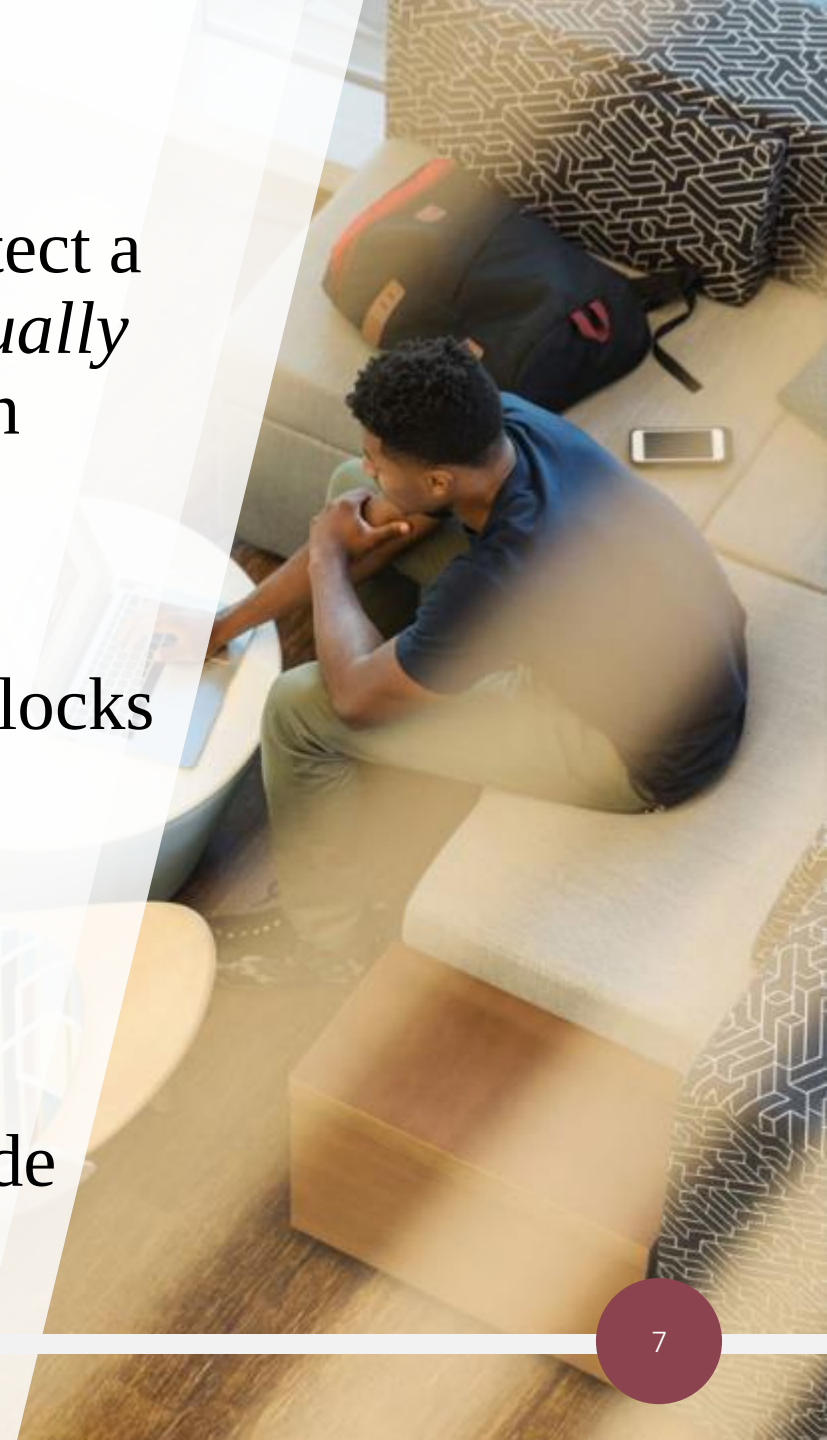
- AtomicIntegerExample.java

synchronized keyword

- Atomic classes do not give us the ability to guard/protect a block of code i.e. {}. In effect, we want to create a *mutually exclusive* piece of code i.e. only one thread at a time can execute the code block.

- In operating systems, these mutually exclusive code blocks are known as *critical sections* and structures known as *monitor*s enables their implementation.

- Every object in Java, has a built-in lock/monitor that automatically kicks in when used with *synchronized* code blocks.

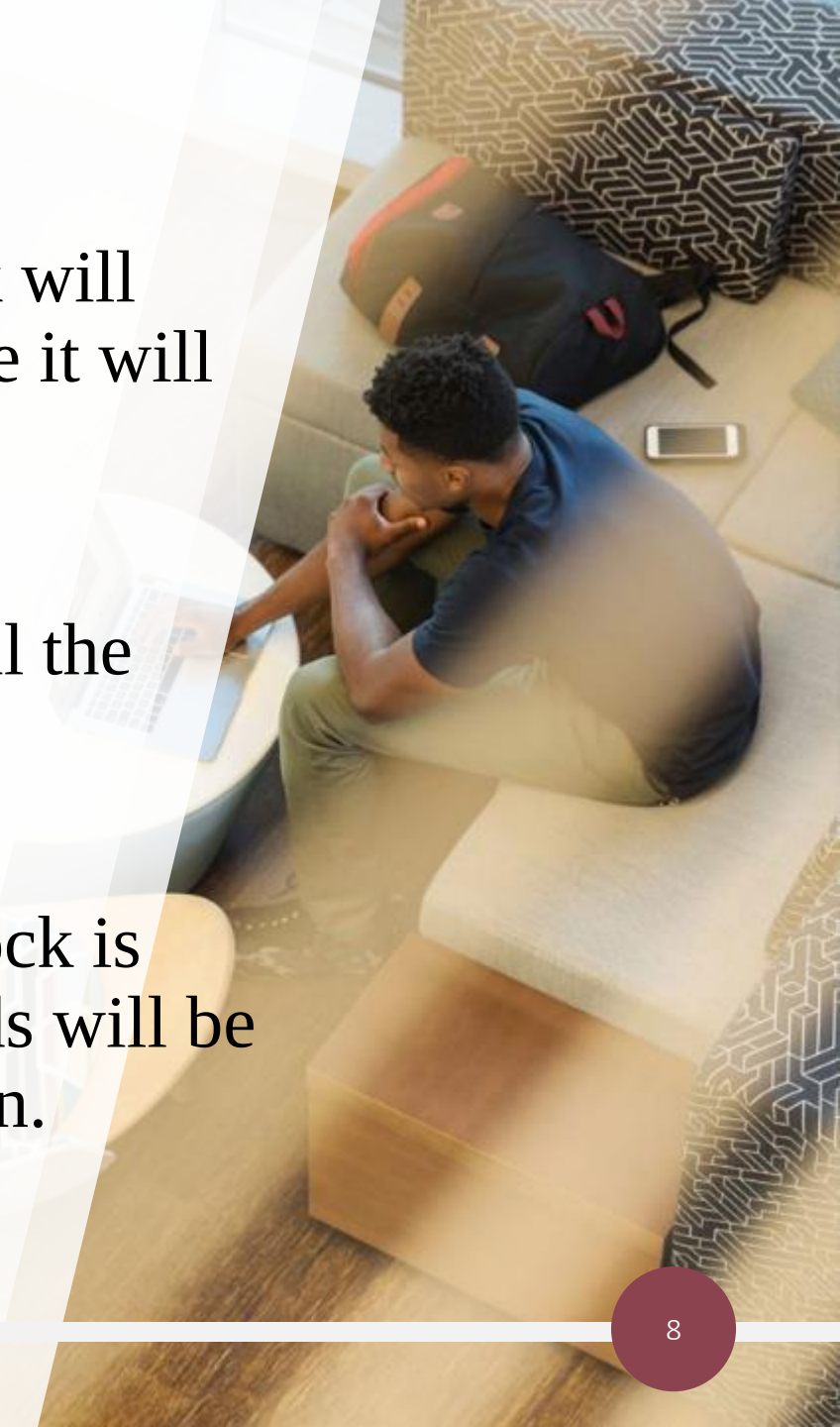


synchronized keyword

- A thread wishing to enter a *synchronized* code block will automatically try to acquire the lock. If the lock is free it will get the lock.

- Any other thread now arriving will have to wait until the first thread is finished in the critical section.

- When the first thread exits the critical section, the lock is released automatically. Now one of the waiting threads will be allowed to obtain the lock and enter the critical section.



synchronized keyword

- Note that threads must be using the same object i.e. if the threads are using different objects then they are using different locks and we will encounter data races.

- We can use the *synchronized* keyword on methods as well as code blocks.

```
class UseCounter {
    private int x;
    public void incrementA() { // <=> incrementB()
        // non-static blocks lock on an object e.g. 'this'
        synchronized(this){
            x++;
        }
    }
    // non-static methods lock on 'this'
    public synchronized void incrementB() { // <=> incrementA()
        x++;
    }

    private static int y;
    public static void decrementA() {
        // static blocks lock on the class object
        // Every class is associated with an object of Class type
        // accessible using Classname.class
        synchronized(UseCounter.class){
            y--;
        }
    }
    // static methods lock on the class object
    public static synchronized void decrementB() {
        y--;
    }

}
```

- FixRaceWithSynchronized.java

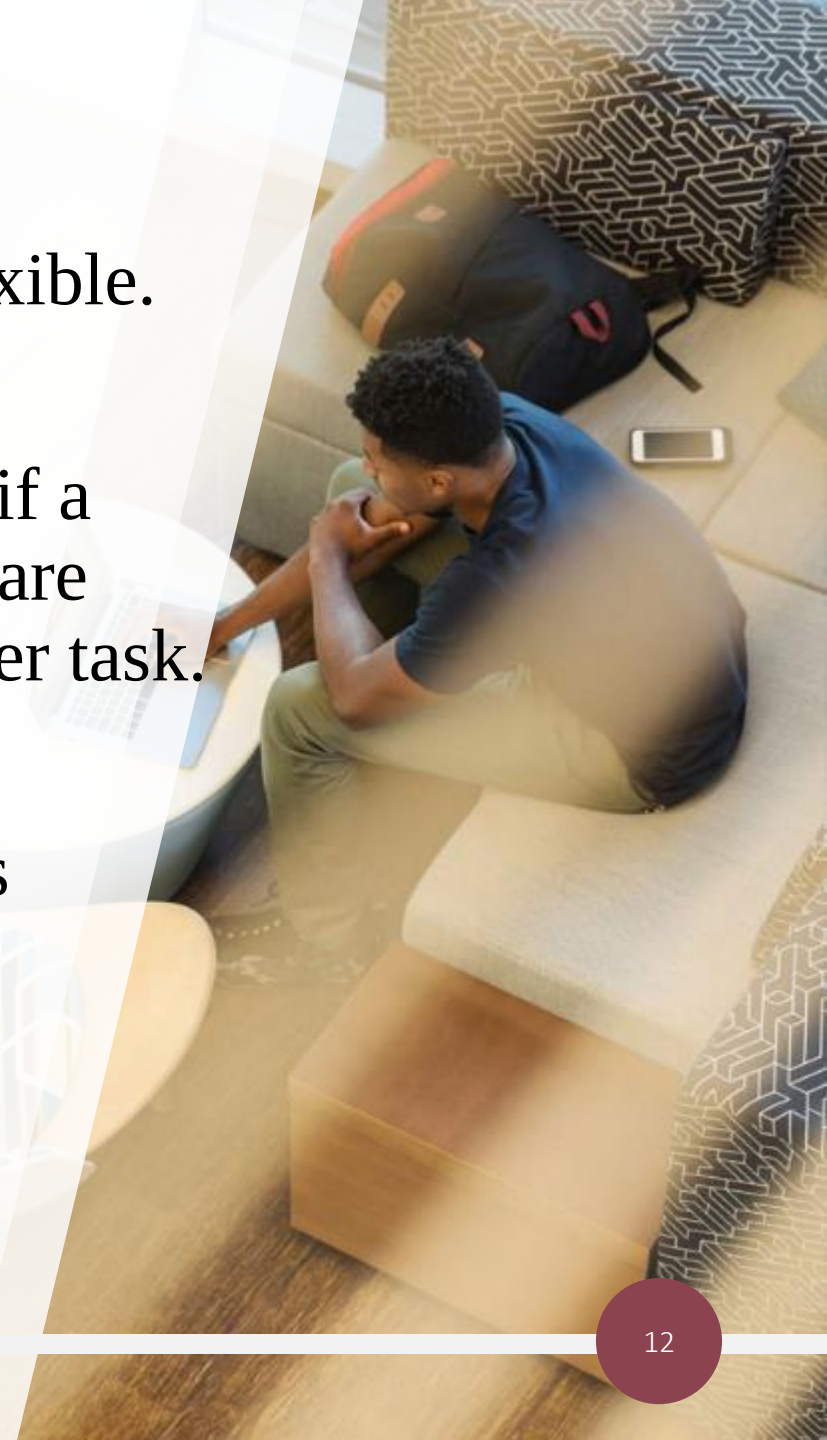
Lock interface

- Although similar to *synchronized*, a *Lock* is more flexible.

- For example, with *synchronized*, a thread is blocked if a previous thread has the lock whereas with *Lock*, if we are unable to get the lock we are free to perform some other task.

- We must explicitly lock on an object that implements *Lock* (as opposed to synchronising on any object).

- Also, we must explicitly free the lock when finished (the *finally* block is useful for this).



```
public static void blockingVersion(){
    Lock lock = new ReentrantLock();
    try{
        lock.lock(); // blocking call
        // critical section
    } finally{
        lock.unlock();// return the lock
    }
}
```

```
public static void nonBlockingVersion(){
    Lock lock = new ReentrantLock();
    // non-blocking call i.e. returns immediately:
    //    true : have the lock
    //    false: could not get the lock
    if(lock.tryLock()){
        try{
            // do not get the lock a second time as you must then
            // unlock it twice
              lock.lock(); // blocking call
            // critical section
        } finally{
            lock.unlock();// return the lock
        }
    }else{
        // did not get the lock, do something else
    }
}
```