Race Conditions and Thread Synchronization (Locking)

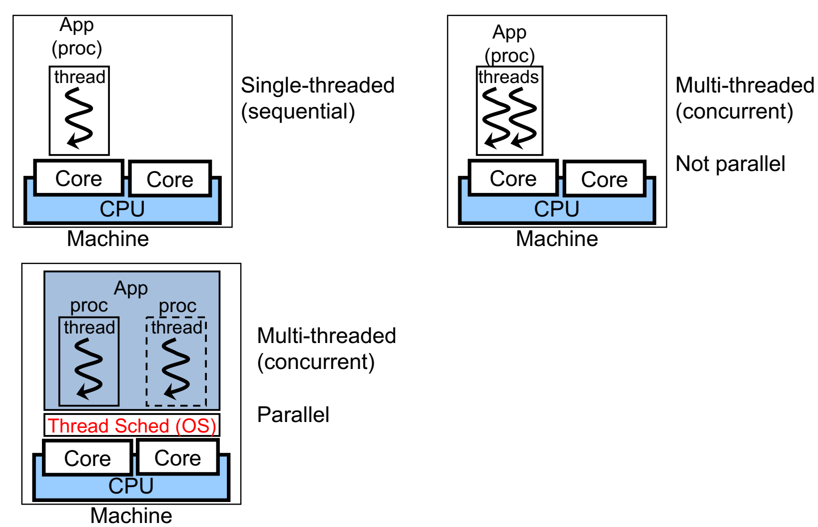
Thread Safety

- Threads are a powerful tool to make your code more responsive and efficient.
- However, multi-threaded code can raise ***"thread safety" issues*** if it is poorly written.
- 2 major thread safety issues
  - Race conditions (data races)
    - Mess up the consistency of data shared among threads
  - Deadlock
    - Make code execution stuck.
- Thread-safe code is free from those 2 issues.

Race Conditions (Data Races)

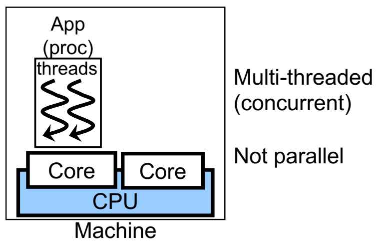
- Threads run *independently*.
  - No coordination among threads by default.
    - c.f. MCTest, PrimeNumberGenerator
    - join() allows threads to coordinate with each other.
- They can share objects/data.
  - Exception: Local variables are NOT shared among threads.
- They can mess up the consistency of the shared objects/data.
  - A thread can write some data to a variable when another thread is reading data from the variable.
  - A thread can write some data to a variable when another thread is writing different data to the variable.

Single- and Multi-threaded Programs



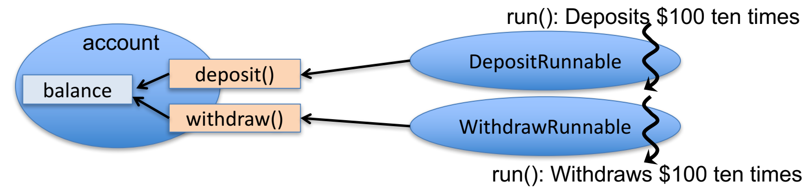
In CS681…

- We always assume a single CPU core that run multiple threads.
  – The most conservative scenario.
- If your code is thread-safe in the most conservative scenario, it is always thread-safe in less conservative scenarios as well.



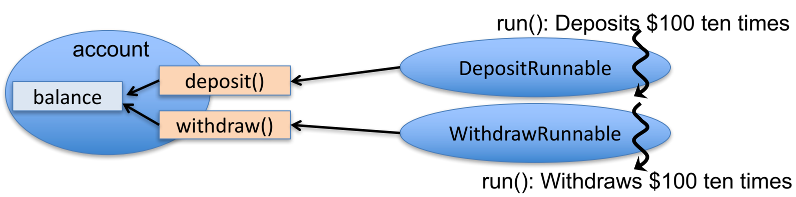
An Example Race Condition:

ThreadUnsafeBankAccount.java



- The variable “balance” is shared by 2 threads.
- They access the variable independently.

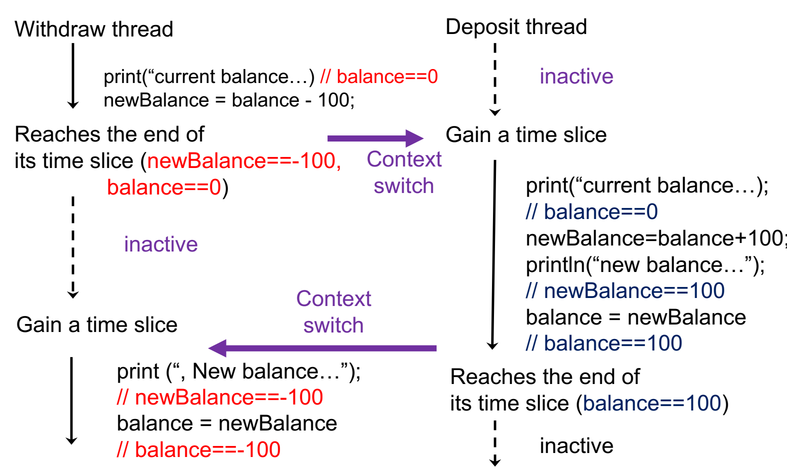
```
• public void deposit(double amount){
      System.out.print("Current balance (d): " + balance);
      double newBalance = balance + amount;
      System.out.println(", New balance (d): " + newBalance);
      balance = newBalance;
  }
• public void withdraw(double amount){
      System.out.print("Current balance (w): " + balance);
      double newBalance = balance - amount;
      System.out.println(", New balance (w): " + newBalance);
      balance = newBalance;
  }
```

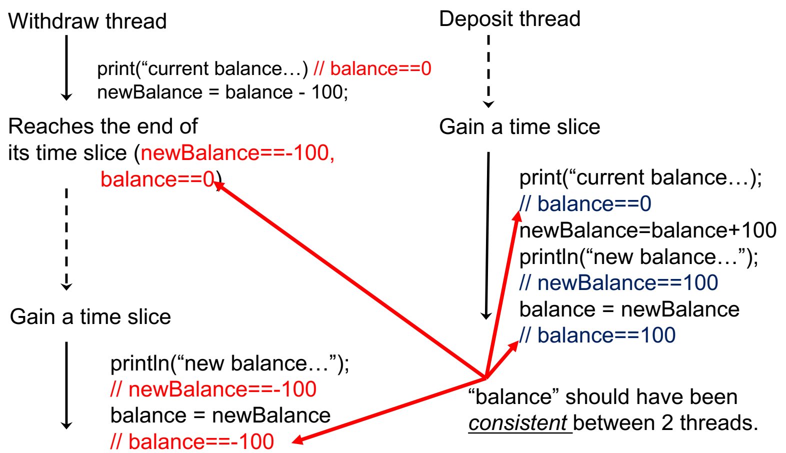
- Desirable output:
  - `Current balance (w): 0.0, New balance (w): -100.0`
  - `Current balance (d): -100.0, New balance (d): 0.0`
  - `Current balance (w): 0.0, New balance (w): -100.0`
  - `Current balance (d): -100.0, New balance (d): 0.0`
  - `Current balance (d): 0.0, New balance (d): 100.0`
  - `Current balance (w): 100.0, New balance (w): 0.0`
  - …
- In reality:
  - `Current balance (w): 0.0Current balance (d): 0.0, New balance (d): 100.0`
  - `, New balance (w): -100.0`

How Can This Happen?

- `Current balance (w): 0.0Current balance (d): 0.0, New balance (d): 100.0`
- `, New balance (w): -100.0`



- `Current balance (w): 0.0Current balance (d): 0.0, New balance (d): 100.0`
- `, New balance (w): -100.0`



The Source of the Problem: Visibility

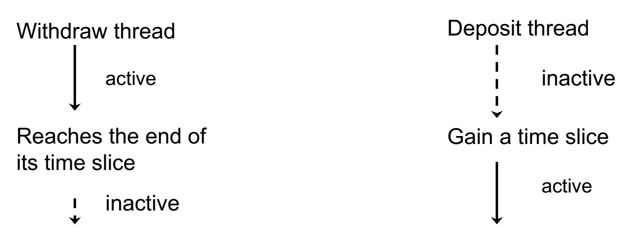
- ThreadUnsafeBankAccount.java is NOT thread safe.
  - Race conditions can occur.
- Race conditions occur due to visibility issues.
  - The current (most up-to-date) value of the shared variable (e.g. "balance") is ***not visible*** for all threads.

Race Conditions (Data Races)

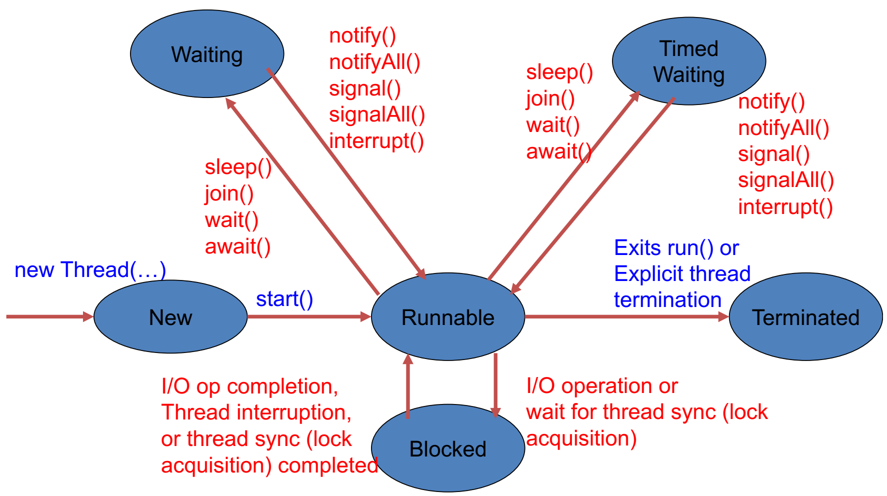
- All threads
  - Run in their *race* to complete their tasks.
  - Manipulate a shared object/data independently.
- The end result depends on which of them happens to win the race.
  - No guarantees on the order of thread execution.
  - No guarantees on how many tasks a thread can perform in a single CPU time slice.
  - No guarantees on the end result on shared data.

Note: Thread States

- Both "active" and "inactive" threads are in the *Runnable* state.
  - The *Runnable* state does NOT distinguish if a thread is actively running on a CPU core or it is inactively waiting for its next turn.
  - The Waiting state does NOT mean that a thread is runnable but inactive.

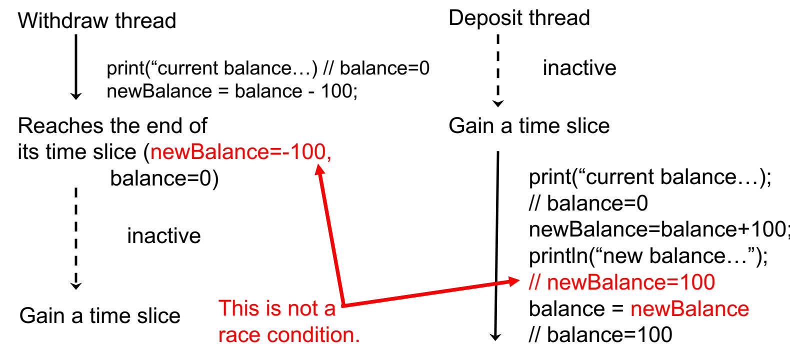


States of a Thread



Note: Local Variables

- Local variables are NOT shared by threads.
  - It is created and maintained in a thread-by-thread manner.
    - The "withdraw" thread has no access to a value of newBalance that the "deposit" thread has created.
    - The "deposit" thread has no access to a value of newBalance that the "withdraw" thread has created.



- Race conditions never occur due to local variables.
- Focus on non-local (i.e. shared) variables in debugging threaded code.

- Two other example local variables

```
– public void deposit(double amount){
        System.out.print("Current balance (d): " + balance);
        double newBalance = balance + amount;
        System.out.println(", New balance (d): " + newBalance);
        balance = newBalance;
  }
– public void withdraw(double amount){
        System.out.print("Current balance (w): " + balance);
        double newBalance = balance - amount;
        System.out.println(", New balance (w): " + newBalance);
        balance = newBalance;
  }
```

Another Example:

ThreadUnsafeBankAccount2

- Local variables (newBalance) are removed from ThreadUnsafeBankAccount

```
– public void deposit(double amount){
        balance = balance + amount;
  }
– public void withdraw(double amount){
        balance = balance - amount;
  }
```

- Output

```
– Current balance (d): 0.0, New balance (d): 100.0
– Current balance (w): 100.0, New balance (w): 0.0
– Current balance (d): 0.0, New balance (d): 100.0
– Current balance (w): 100.0, New balance (w): 0.0
– Current balance (d): 0.0Current balance (w): 0.0, New balance (w):
  -100.0
– , New balance (d): 100.0
```

How Can This Happen?

- `Current balance (d): 0.0Current balance (w): 0.0, New balance (w): -100.0`
- `, New balance (d): 100.0`

Deposit thread

print("current balance…) // 0
balance +100 // --> 100

Reaches the end of
its time slice (balance==0)

Context
switch

Withdraw thread

Gain a time slice

print("current balance…); // 0
balance = balance - amount;
// balance=-100
println("new balance…); // -100

Context
switch

Gain a time slice

Reaches the end of
its time slice (balance==-100)

balance = 100;
println("new balance…);
// balance=100

- `Current balance (d): 0.0Current balance (w): 0.0, New balance (w): -100.0`
- `, New balance (d): 100.0`

Deposit thread

print("current balance…) // 0
balance+100 // --> 100

Reaches the end of
its time slice (balance==0)

Withdraw thread

Gain a time slice

print("current balance…); // 0
balance = balance - amount;
// balance=-100
println("new balance…); // -100

Gain a time slice

Reaches the end of
its time slice (balance==-100)

balance = 100;
println("new balance…);
// balance==100

"balance" should have been
consistent between threads.

- This is not a solution: `balance -= amount;`
  - Just a syntactic sugar for `balance = balance - amount;`
- All threads
  - Run in their *race* to complete their tasks.
  - Manipulate a shared object/data independently.
- The end result depends on which of them happens to win the race.
  - No guarantees on the order of thread execution.
  - No guarantees on how many tasks a thread can perform in a single CPU time slice.
  - No guarantees on the end result on shared data.

Thread Synchronization

- Need to *synchronize* threads
  - i.e., Need to *serialize* their concurrent access to a shared variable
- Thread synchronization enables serialized (or atomic or exclusive) access to a shared variable
  - Allows only one thread to access a shared variable at a time
    - Forces the other threads to wait and take turn to access it.
    - Prevents the "deposit" thread from depositing money to "balance" when the "withdraw" thread is withdrawing money from "balance."
    - Prevents the "withdraw" thread from withdrawing money from "balance" when the "deposit" thread from depositing money to "balance."

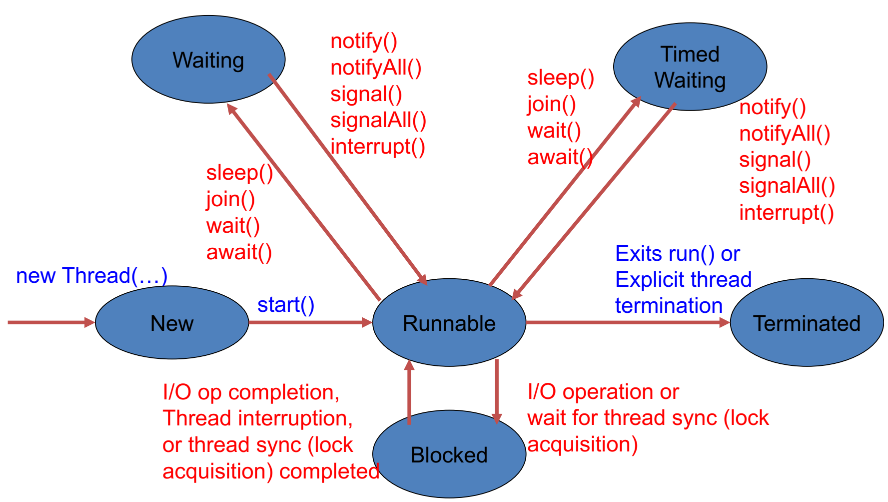
- Thread synchronization allows programmers to write *atomic code* (a.k.a critical section).
  - When a thread is running it, no other threads can run it.
  - No intermediate result/state can be revealed/exposed to other threads.

Locks

- Used to synchronize/serialize the threads that manipulate shared data.
- `java.util.concurrent.locks.Lock` interface
  - `ReentrantLock` class: the most commonly-used class for locking
    - Defines methods that allows threads to access shared data in a synchronized/serialized way.
- Atomic code is surrounded by `lock()` and `unlock()` method calls.
  - ```
    ReentrantLock aLock = new ReentrantLock();
    aLock.lock();
    atomic code
    aLock.unlock();
    ```


- Once a thread calls `lock()`,
  - it acquires and owns a ***lock*** until it calls `unlock()`.
  - No other threads can acquire the lock until it is released by `unlock()`.
    - No other threads can execute atomic code until the lock is released with `unlock()`.
- If a thread calls `lock()` when another thread already owns the lock,
  - it goes to the *blocked* state and *gets blocked* (cannot do anything further) until the lock is released.

States of a Thread


    ```

How Can a Blocked Thread Run Again?

- JVM's thread scheduler
  - Periodically reactivates all blocked threads so that they can try to acquire the target lock.
    - If the lock is still unavailable, they get blocked again.
  - Detects a release of the target lock (i.e. completion of atomic code).
    - May notify all blocked threads so that one of them can acquire the target lock.
    - May choose one of the blocked threads to acquire the lock.
- Each blocked thread can eventually acquire the target lock when it is available.

Coding Idiom for Locking

- Call unlock() in a finally clause.
  - ReentrantLock aLock = new ReentrantLock();

```
aLock.lock();
try {
    atomic code
}
finally {
    aLock.unlock();
}
```

- unlock() is never invoked
  - if run() returns from atomic code
  - if atomic code throws an exception
  - A deadlock occurs.
    - Atomic code is locked forever, and no other threads can acquire the lock to execute the atomic code.

-
```
aLock.lock();
try{
    atomic code
}
finally{
    aLock.unlock();
}
```

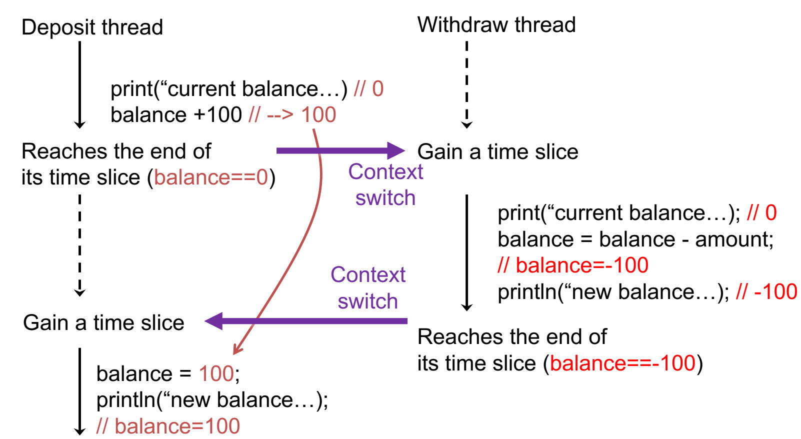
DO THIS!

-
```
try{
    aLock.lock();
    atomic code
}
finally{
    aLock.unlock();
}
```

DON'T DO THIS!

- Make sure to call lock() before a "try" block.
- If a thread throws an exception in lock(), it will not acquire the lock. However, it will call unlock().
  - lock() can throw an InterruptedExcepetion when another thread call interrupt().

When Could a Context Switch Occur?



When Could a Context Switch Occur?

- Across atomic operations.
- In a compound operation.

Atomicity of Operations for Primitive Types

- The read and write operations for primitive data types, except double and long (64-bit) types, are atomic.
  - An "atomic" operation is transformed to a single bytecode instruction for a JVM.
  - While a thread works on an atomic operation, no other threads can work on it.
  - **`int x;`**
    - Thread A does: **`x=1;`** Thread B does: **`x=2;`**
    - An assignment of an int value is atomic.
    - x contains 1 or 2 depending on which thread performs assignment earlier.
      - x never contains other values (e.g., 0 and 3) or corrupted data.
      - An example of corrupted data
        - » Some part of x (e.g., the first 16-bit of x) comes from Thread A and the remaining part (e.g., the other 16bit of x) comes from Thread B.

Compound Operations

- A *compound* of atomic operations is NOT atomic.
  - int i; boolean done;
  - done = true;// 2 steps
  - i = 1; // 2 steps
  - if(done) // 2 steps
  - i = j; // 2 steps
  - j = i +1; // 5 steps
    - Reading the value of i, reading/loading the value of 1, doing i+1, storing the result of i+1 to a certain memory space, and assigning the result to j.
  - i = i + 1; // 5 steps
  - i++ // 5 steps
  - A race condition can occur due to a context switch *in between* atomic operations/steps.

An Example Race Condition

- **`i = i + 1`**
  - A compound of 5 atomic operations.
  - There are 4 places where race conditions can occur.
- Thread synchronization enables serialized (or atomic or exclusive) access to a compound operation.
  - Allows only one thread to perform a compound operation at a time.

```
ReentrantLock aLock = new ReentrantLock();
aLock.lock();
try{
    i = i + 1; // treated as an atomic operation
}
finally{
    aLock.unlock();
}
```

Another Example

- ```
  public void deposit(double amount){
      balance = balance + amount;
  }
  ```
  - A compound of 5 atomic operations.
  - There are 4 places where race conditions can occur.
- Thread synchronization enables serialized (or atomic or exclusive) access to a compound operation.
  - Allows only one thread to perform a compound op at a time.

```
– ReentrantLock aLock = new ReentrantLock();
  aLock.lock();
  try{
    balance = balance + amount; // treated as an atomic op
  }
  finally{
      aLock.unlock();
  }
```

What about 64-bit Types?

- The read and write operations for double and long variables are <u>NOT</u> atomic.
  - long x;
    - Thread A does: x = 1L;
    - Thread B does: x = 2L;
    - No guarantee that x contains 1L or 2L.
    - x can contain another value (e.g., 0L or 3L) or corrupted data.
  - aLongVar = 100L; // 2+ bytecode instructions
  - If(aLongVar) // 2+ bytecode instructions
  - aLongVar ++ // 5+ bytecode instructions

Atomicity of Operations for Reference Types

- The read and write operations reference types are atomic.
  - A compound of atomic operations
    - e.g., Foo foo = temp; // requires multiple steps
  - A race condition can occur due to a context switch <u>*in between*</u> atomic steps.

ThreadSafeBankAccount

- Output
  - Lock obtained
  - Current balance (d): 0.0, New balance (d): 100.0
  - Lock released
  - Lock obtained
  - Current balance (w): 100.0, New balance (w): 0.0
  - Lock released
  - Lock obtained
  - Current balance (d): 0.0, New balance (d): 100.0
  - Lock released
  - Lock obtained
  - Current balance (w): 100.0, New balance (w): 0.0
  - Lock released
```

Be VERY Careful

- When multiple threads share and access a variable concurrently.
  - Make sure to guard the shared variable with a lock.
    - Surround reading and writing parts with lock() and unlock().
      - To treat the reading/writing parts as atomic code
- When a loop performs a conditional check with a shared variable (e.g., flag).
  - Surround reading part (i.e., conditional block) and writing part (i.e., flag-flipping part) with lock() and unlock()
    - To treat the reading/writing parts as atomic code

What's Tricky in Thread Programming

- Your test code may or may not be able to detect race conditions.
  - It may not be able to detect race conditions even if you run it a lot of (e.g. few hundred) times.

Consider this Lucky Case

```
public void deposit(double amount){
    balance = balance + amount; }
```

Deposit thread	Withdraw thread
balance = balance +100 // balance == 0	
Reaches the end of its time slice (balance==0)	Gain a time slice
	balance = balance - amount; // balance == -100
Gain a time slice	Reaches the end of its time slice (balance==-100)

Nested Locking

```
class BankAccount {
 private double balance;
 private ReentrantLock lock;

 public void deposit(double amount) {
     lock.lock();
     balance += amount;          // 5 atomic steps
     if(balance < MIN_BALANCE)   // 4 atomic steps
         subractPenaltyFee();
     lock.unlock(); }

 private void subractPenaltyFee() {
     balance -= PENALTY;         // 5 atomic steps
     // NO NEED TO SURROUND THIS LINE BY LOCK() and UNLOCK()
     // It is a part of atomic code.
     }
}
```

Thread Reentrancy

- ```
  class A{
    private B b;
    private ReentrantLock lock;
    public void a1(){
      b = new B();
      lock.lock();
      b.b1(this); //nested locking
      lock.unlock();
    }
    public void a2(){
      lock.lock();
      do something.
      lock.unlock();
    }
  }
  ```

- ```
  Class B{
    public void b1(A a){
      a.a2();
    }
  }
  ```

If a thread performs:

```
A a = new A();
a.a1();
```

it ***re-enters (or re-acquires)*** the same lock that it already owns.

- This code does not have a deadlock problem.
- A thread can ***re-enter*** the same lock as far as it already owns the lock.