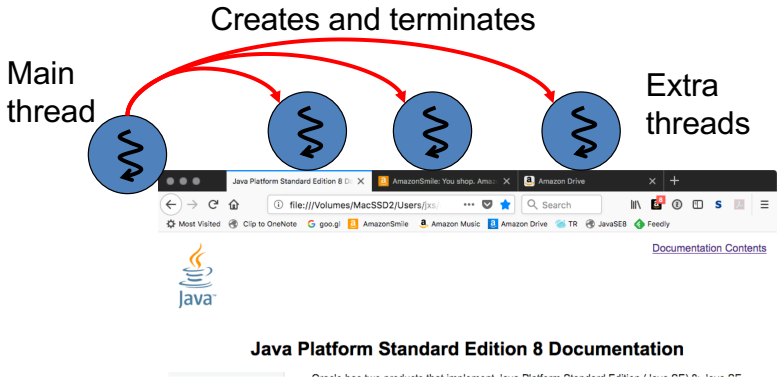
Race Conditions and Thread Synchronization (Locking)

Goals of Concurrency/Multi-threading

- **Responsiveness**



- **Efficiency**
  - c.f. MCTest, multi-threaded prime number generation

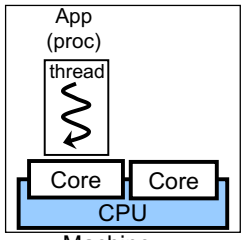
Thread Safety

- 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
    - Break the shared data
  - Deadlock
    - Make code execution stuck.
- Thread-safe code is free from those 2 issues.

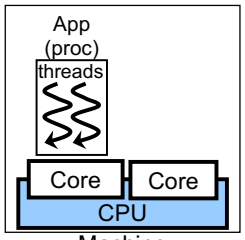
Race Conditions (a.k.a. Data Races)

- Threads run *independently*.
  - No coordination among threads by default.
    - c.f. MCTest, RunnablePrimeGenerator
    - join() allows threads to coordinate with each other.
- They can share variables (data fields).
  - Exception: Local variables are NEVER 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

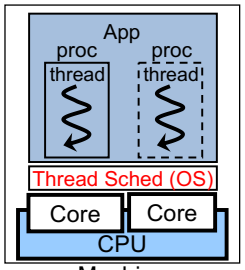


Single-threaded
(sequential)



Multi-threaded
(concurrent)

Not parallel

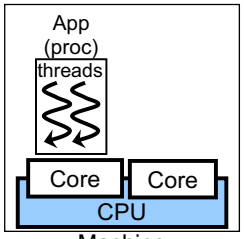


Multi-threaded
(concurrent)

Parallel

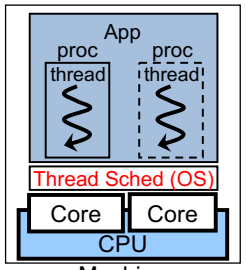
In CS681…

- We always assume a single CPU core that runs multiple threads.
  - The most conservative scenario in terms of parallel program execution.
- If your code is thread-safe in the most conservative scenario, it is always thread-safe in less conservative scenarios as well.



Multi-threaded
(concurrent)

Not parallel

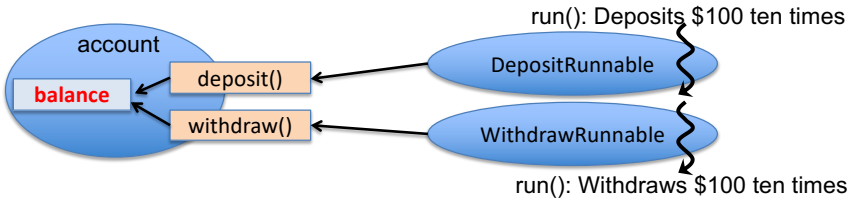


Multi-threaded
(concurrent)

Parallel

An Example Race Condition:

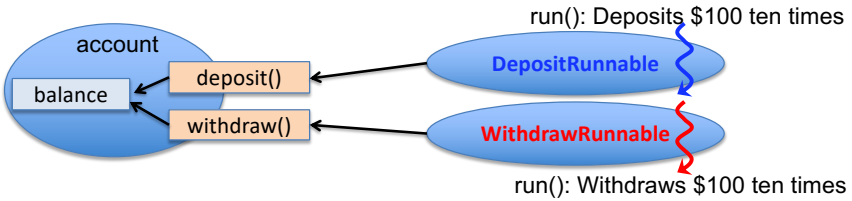
ThreadUnsafeBankAccount.java



- 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;
}

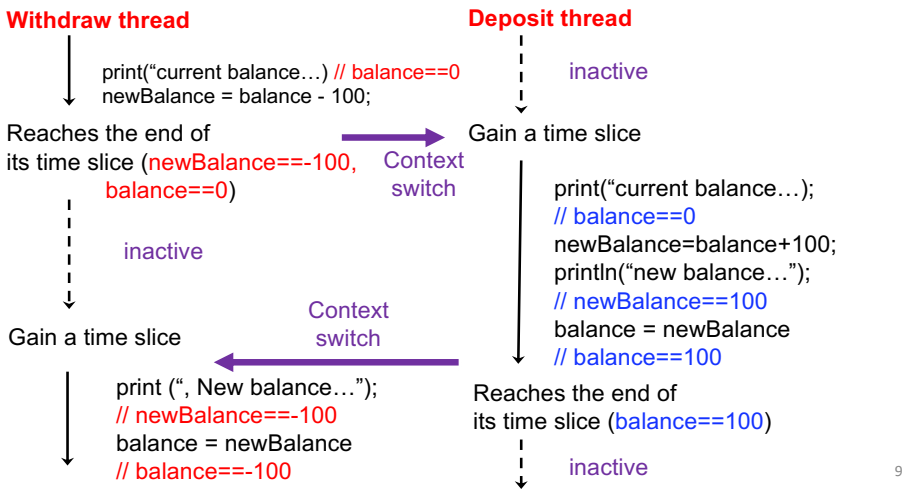
public void withdraw(double amount){
     System.out.print("Current balance (w): " + balance);
     double newBalance = balance - amount;
     System.out.println(", New balance (w): " + newBalance);
     balance = newBalance;
}
```

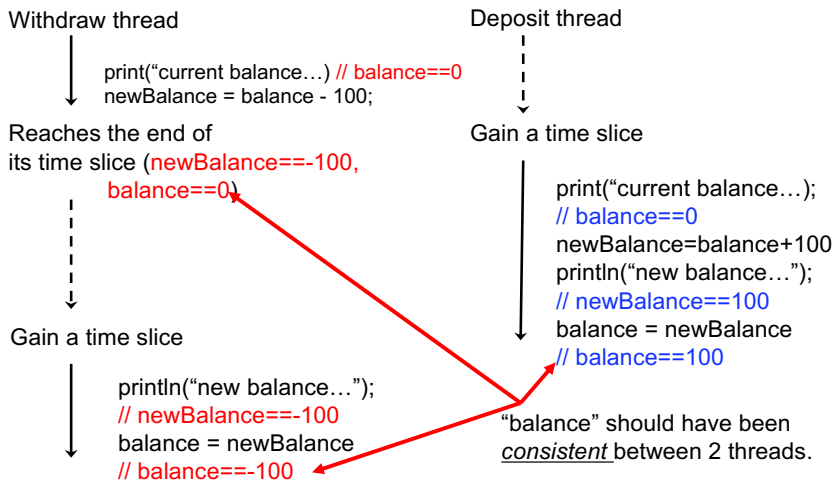
- Desirable output:
  - Current balance (w): 0.0, New balance (w): -100.0
  - Current balance (d): -100.0, New balance (d): 0.0
  - Current balance (w): 0.0, New balance (w): -100.0
  - Current balance (d): -100.0, New balance (d): 0.0
  - Current balance (d): 0.0, New balance (d): 100.0
  - Current 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`** is NOT thread safe.
  - Race conditions can occur.
- Race conditions occur due to visibility issue.
  - The current (most up-to-date) value of a shared variable (e.g. "balance") is *not visible* for all threads.

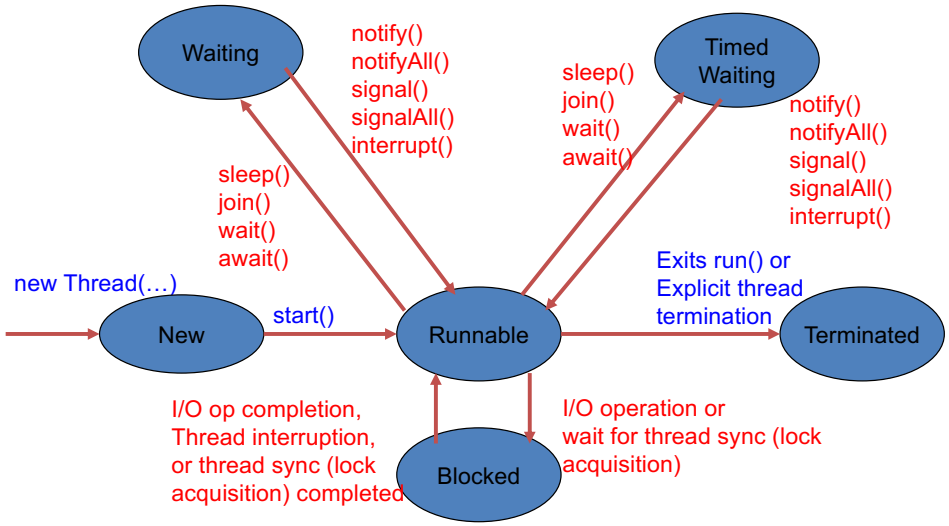
Race Conditions (a.k.a. Data Races)

- 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 much task a thread can perform in a single CPU time slice/quota.
  - No guarantees on the end result on shared data.

Note: Thread States

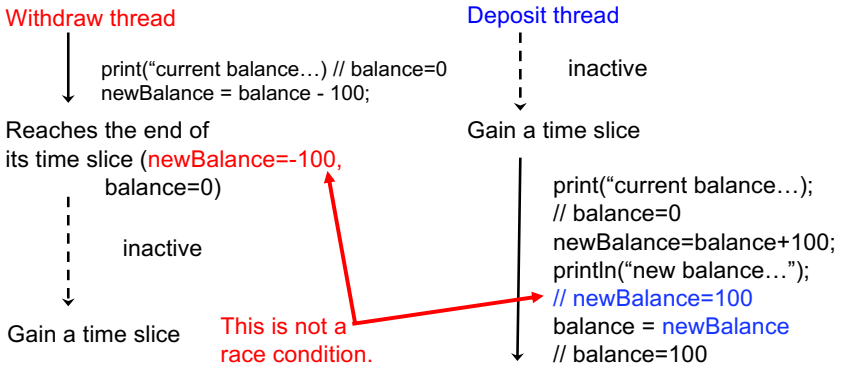
- Both "active" and "inactive" threads are in the *Runnable* state.
  - The *Runnable* state does NOT distinguish if a thread is "actively running" on a CPU core or it is "inactively waiting" for its next turn.
  - The Waiting state does NOT mean that a thread is runnable but inactive.

States of a Thread



Note: Local Variables

- Local variables are NEVER 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 on local variables.
- Focus on non-local (i.e. shared) variables in debugging threaded code.
- Another local variable: **`amount`**

```
– 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

- Eliminates the local variable **`newBalance`**.

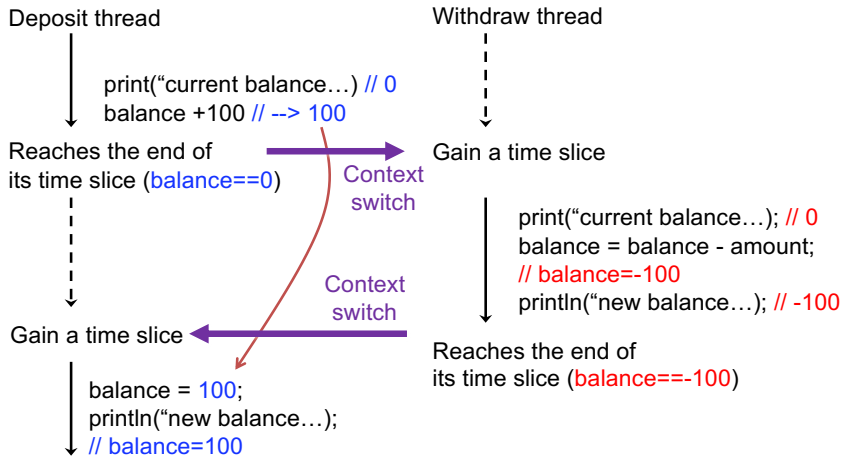
```
– public void deposit(double amount){
        System.out.print("Current balance (d): " + balance);
        balance = balance + amount;
        System.out.println(", New balance (d): " + balance); }

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

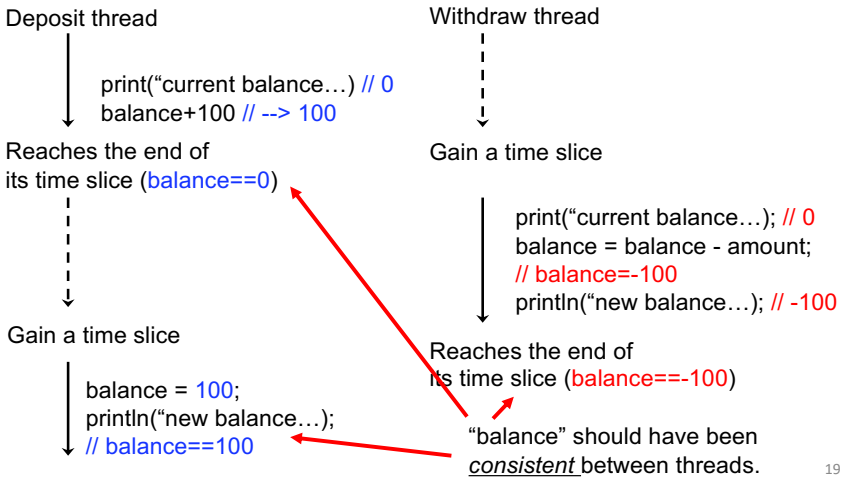
- Output
  - Current balance (d): 0.0, New balance (d): 100.0
  - Current balance (w): 100.0, New balance (w): 0.0
  - Current balance (d): 0.0, New balance (d): 100.0
  - Current balance (w): 100.0, New balance (w): 0.0
  - Current balance (d): 0.0Current balance (w): 0.0, New balance (w): -100.0
 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



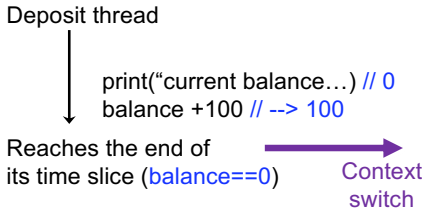
- Current balance (d): 0.0Current balance (w): 0.0, New balance (w): -100.0
- , New balance (d): 100.0



The Source of the Problem: Visibility

- **`ThreadUnsafeBankAccount2`** is still NOT thread-safe.
- Race conditions can still occur due to visibility issue.
  - The current (most up-to-date) value of the shared **`balance`** is *not visible* for all threads.
- Eliminating the local variable **`newBalance`** is not a solution to avoid race conditions.
  - In fact, race conditions never occur on local variables.

- Race conditions (i.e. visibility issue) come out if a context switch occurs where/when we do not want it to occur.
  - e.g., `balance = balance + amount;`
    - `balance + amount`
    - Context switch
    - `balance` = the result of (`balance + amount`)



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.
  - No context switches occur during the execution of a single byte code instruction.

Where can a Context Switch Occur?

- It is important to know where/when a context switch can occur in your code.
- A context switch can occur across atomic operations.
  - `int i = 1; i = 2;`
    - Sequence of 2 atomic operations (2 steps)
  - `balance = balance + amount;`
    - Sequence of 5 atomic operations (5 steps)
  - `balance += amount;`
    - Syntactic sugar for `balance = balance + amount;`

- The read and write operations for primitive data types, except double and long (64-bit) types, are atomic.
  - Given `int x;`, Thread A does: `x=1;` Thread B does: `x=2;`
    - An assignment of an int value (write operation) is atomic.
    - `x` contains 1 or 2 depending on which thread performs assignment earlier.
    - `x` never contain other values (e.g., 0 and 3) or corrupted data.
      - An example of corrupted data
        - » The first 16-bit of `x` is assigned by Thread A and the remaining part is assigned by 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
  - `return i;` // 2 steps
  - A race condition can occur due to a context switch *in between* atomic operations/steps.

What about 64-bit Types?

- The read and write operations for double and long variables are NOT atomic.
  - Given `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., 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.
  - Given `String s;`, Thread A does: `s="a";` Thread B does: `s="b";`
    - `s` contains "a" or "b" depending on which thread performs assignment earlier.
    - `s` never contain other values (e.g., "x") or corrupted data.
  - A compound of atomic operations
    - e.g., `Foo foo = temp;` // 2 byte code instructions
    - A race condition can occur due to a context switch *in between* atomic steps.

Atomicity of Constructors

- Only one thread can run a constructor on a class instance that is being created and initialized.

```
class Account{
  float balance;                        // Shared variable

  public Account(int initialAmount){// Manipulates the shared
    balance = initialAmount;        // variable thru multiple
  }                                 // atomic steps...but...
                                    // THREAD-SAFE!

  public deposit(float amount){ ... }   // Not thread-safe
  public withdraw(float amount){ ... }} // Not thread-safe
```

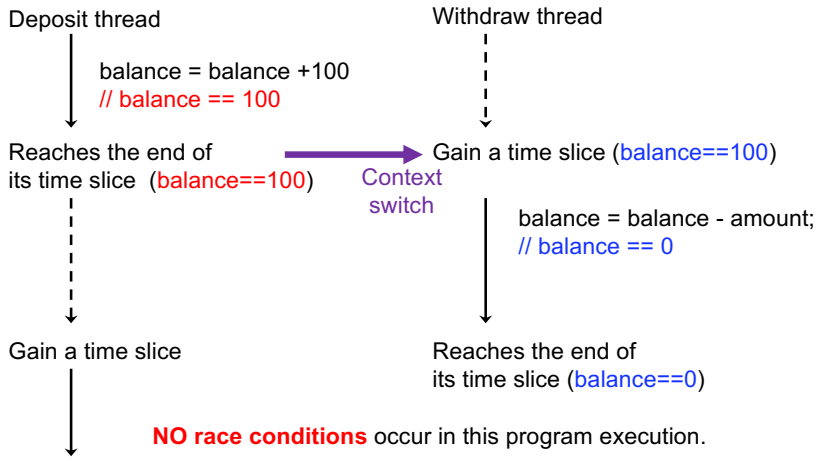
  - Until a thread returns/completes a constructor on a class instance, no other threads can call public methods on that instance.

What's Tricky in Thread Programming

- Your test code may or may not be able to detect race conditions
  - even if you run it a lot of times.

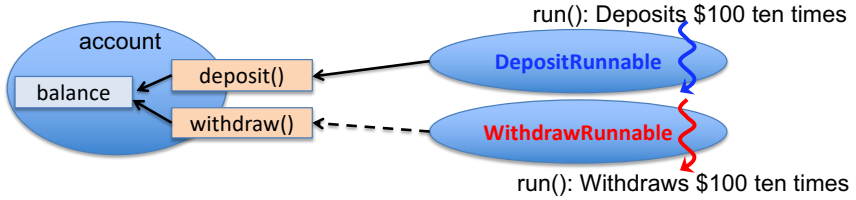
Consider this Lucky Case

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

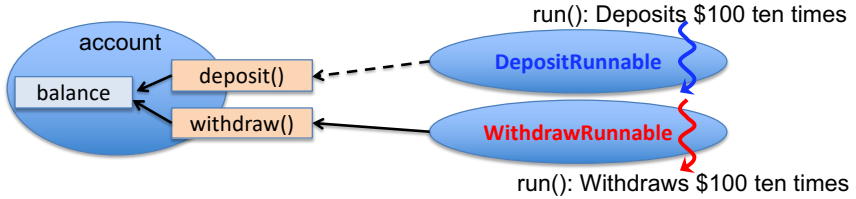


Solution to Avoid Race Conditions

- Thread synchronization
  - Synchronizes threads
  - Enables serialized (or mutually-exclusive) access to a shared variable
  - Allows only one thread to access a shared variable at a time
    - Forces all other threads to wait and take turn to access it.



- Thread synchronization
  - Prevents the "withdraw" thread from withdrawing money from "balance" when the "deposit" thread is depositing money to "balance."
  - Prevents the "deposit" thread from depositing money to "balance" when the "withdraw" thread is withdrawing money from "balance."



Thread Synchronization with Java

- Java implements thread synchronization by
  - Providing locks
  - Allowing you to write *atomic code* (a.k.a critical section) with locks.
    - Atomic code: A piece of code that is executed by multiple threads in a serialized (or mutually-excluded) manner.
      - When a thread is running atomic code, no other threads can run it.
    - No intermediate results/states produced in atomic code can be revealed/exposed to other threads.

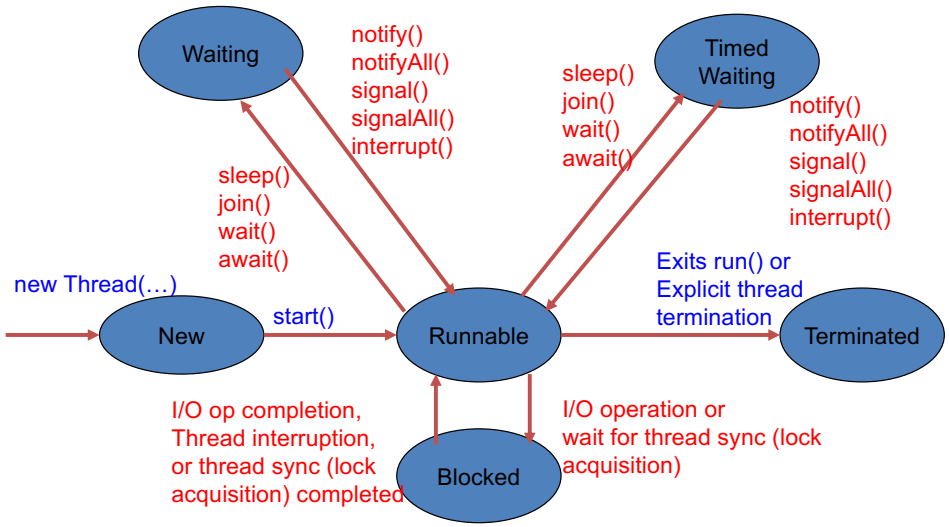
Locks in Java

- Used to synchronize (or serialize, or mutually-exclude) multiple threads that access shared data.
- `java.util.concurrent.locks.Lock` interface
  - `ReentrantLock` class: the most commonly-used class for thread synchronization
    - Defines methods that
      - allow threads to access shared data in a synchronized (or serialized, or mutually-excluded) manner.
      - allow you to write atomic code.
- Atomic code is surrounded by `lock()` and `unlock()` method calls.

```
ReentrantLock aLock = new ReentrantLock();
aLock.lock();
atomic code (access to shared data)
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 run atomic code until the lock is released.
- 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

- Call unlock() in a finally clause.

```
ReentrantLock aLock = new ReentrantLock();
aLock.lock();
try {
    atomic code  (access to a shared variable)
}
finally {
    aLock.unlock();
}
```

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

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

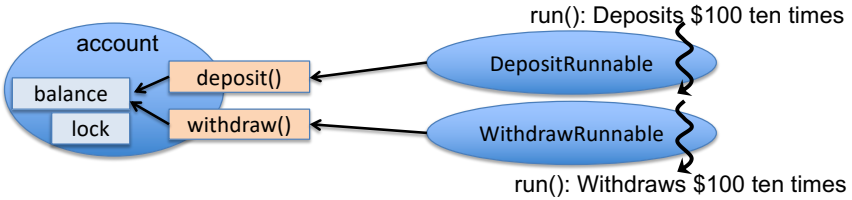
DO THIS!

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

DON'T DO THIS!

- Make sure to call lock() BEFORE a "try" clause.
- 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().

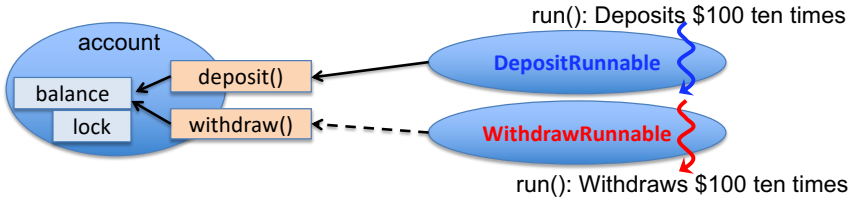
ThreadSafeBankAccount.java



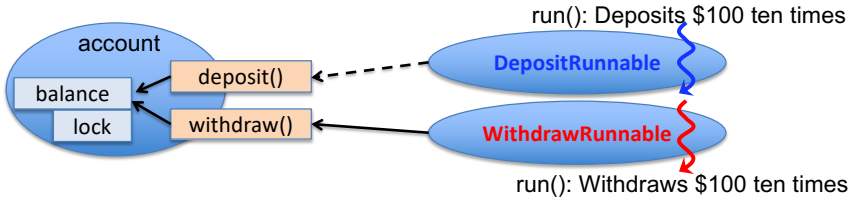
-
```
private ReentrantLock lock = new ReentrantLock();
```

-
```
public void deposit(double amount){
    lock.lock();
     try{
        balance += amount;  // atomic code
    }finally{
        lock.unlock(); }
}
```

-
```
public void withdraw(double amount){
    lock.lock();
    try{
        balance -= amount;  // atomic code
    }finally{
        lock.unlock(); }
}
```



- Thread synchronization
  - Prevents the "withdraw" thread from withdrawing money from "balance" when the "deposit" thread is depositing money to "balance."
  - Prevents the "deposit" thread from depositing money to "balance" when the "withdraw" thread is withdrawing money from "balance."




- ```
  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 exclusive) access to a compound operation.
  - Allows only one thread to perform a compound op at a time.

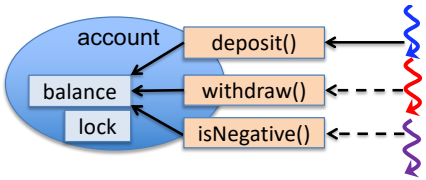
```
- ReentrantLock aLock = new ReentrantLock();
  aLock.lock();
  try{
     balance = balance + amount; // atomic code
  }
  finally{
      aLock.unlock();
  }
```

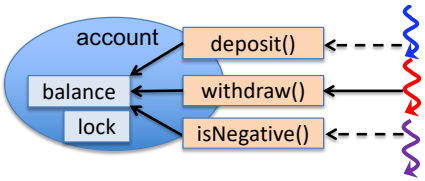
Summary: How to Avoid Race Conditions?

- When multiple threads share and access a variable concurrently,
  - Make sure to guard the shared variable with a lock.
    - Identify ALL read and write logic to be performed on the variable
    - Surround each of them with lock() and unlock().

```
- e.g., public void deposit(double amount){
        lock.lock();
          try{
              balance = balance+amount; // atomic code (read & write)
        }finally{
              lock.unlock(); }
  }

- public void withdraw(double amount){
        lock.lock();
         try{
              balance = balance-amount; // atomic code (read & write)
        }finally{
              lock.unlock(); }
  }
```



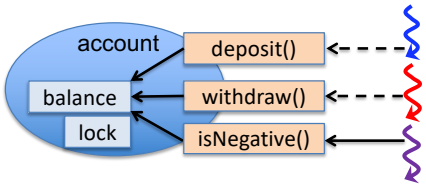



- ```
  private ReentrantLock lock = new ReentrantLock();
  ```

- ```
  public void deposit(double amount){
      lock.lock();
      balance += amount;
      lock.unlock(); }
  ```

- ```
  public void withdraw(double amount){
      lock.lock();
      balance -= amount;
      lock.unlock(); }
  ```

- ```
  Public boolean isNegative(){
      lock.lock();
      if(balance<0){ return true; }
      else if{ return false; }
      lock.unlock(); }
  ```



- Thread synchronization
  - Mutually excludes those 3 threads, so only one of them can run atomic code associated with "lock" at a time.
  - It is important to use the same lock in ALL read and write logic to be performed on "balance." Otherwise, threads are NOT mutually excluded.
  ```

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()
    // because it is called from atomic code.
    }
}
```


Thread Reentrancy

```
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() {
    lock.lock();
    balance -= PENALTY;         // 5 atomic steps
    lock.unlock(); } }
```

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


```
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.

RunnableCancellablePrimeGenerator

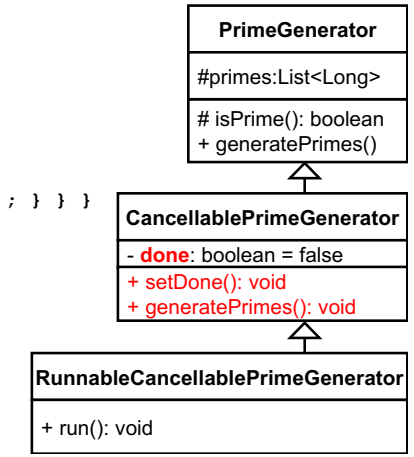
```
class CancellablePrimeGenerator extends PrimeGenerator {
  private boolean done = false;

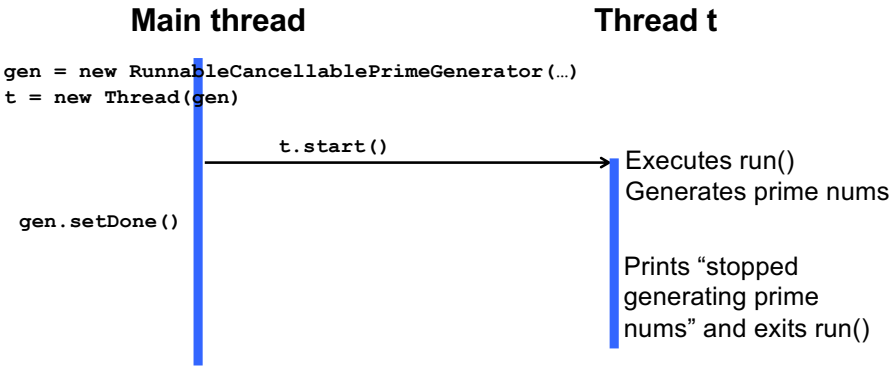
  public void setDone(){
    done = true; }

  public void generatePrimes(){
    for( long n = from; n <= to; n++ ){
      if(done){
        System.out.println("Stopped...");
        this.primes.clear();
        break;
      }
      if( isPrime(n) ){ this.primes.add(n); } } }

class RunnableCancellablePrimeGenerator
  extends CancellablePrimeGenerator
  implements Runnable {

  public void run(){
    generatePrimes(); } }
```

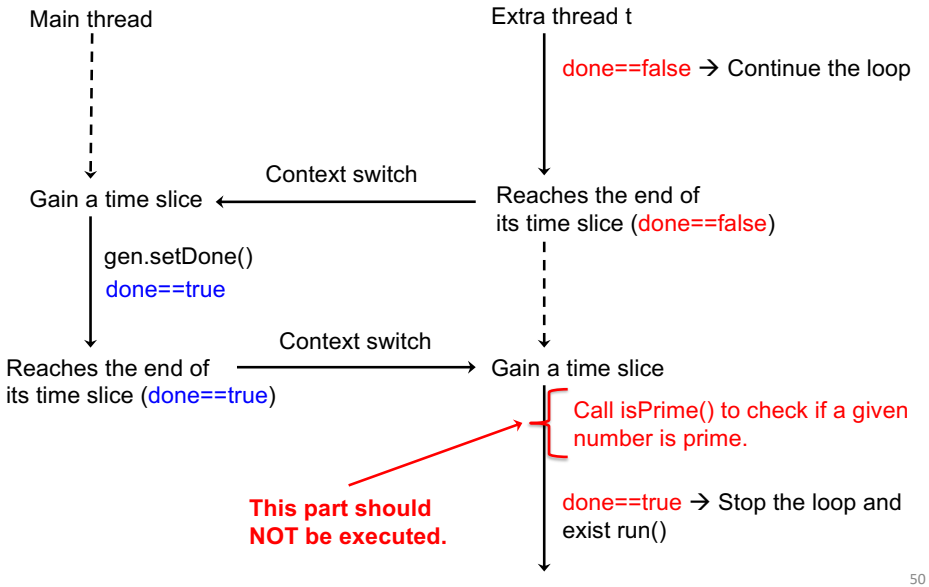





```
for(long n = from; n <= to; n++){
    if(done){
        System.out.println("Stopped generating prime nums.");
        this.primes.clear();
        break;
    }
    if( isPrime(n) ){ this.primes.add(n); }}
```

- This code is NOT thread-safe. Race conditions can occur.

A Potential Race Condition




```
class CancellablePrimeNumberGenerator extends PrimeNumberGenerator{
  private boolean done = false;

  public void run(){
    for( long n = from; n <= to; n++ ){
      if(done){
        System.out.println("Stopped generating prime nums.");
        this.primes.clear();
        break;
      }
      if( isPrime(n) ){ this.primes.add(n); }
    }

  public void setDone(){
    done = true;
  }
}
```

Thread *t*

Context switch


```
class CancellablePrimeNumberGenerator extends PrimeNumberGenerator{
  private boolean done = false;

  public void run(){
    for( long n = from; n <= to; n++ ){
      if(done){
        System.out.println("Stopped generating prime nums.");
        this.primes.clear();
        break;
      }
      if( isPrime(n) ){ this.primes.add(n); }
    }

  public void setDone(){
    done = true;
  }
}
```

Thread *t*

Context switch

Main thread

Context switch

Visibility Issue

```
class CancellablePrimeNumberGenerator extends PrimeNumberGenerator{
  private boolean done = false;

  public void run(){
    for( long n = from; n <= to; n++ ){
      if(done){
        System.out.println("Stopped generating prime nums.");
        this.primes.clear();
        break;
      }
      if( isPrime(n) ){ this.primes.add(n); }
    }

  public void setDone(){
    done = true;
  }
}
```

Thread *t*

Context switch

Main thread

Context switch

- The current (most up-to-date) value of the shared variable "done" is not *visible* for all threads.
- Solution:
  - Identify all read and write logic on the shared variable "done"
  - Surround each read/write logic with lock() and unlock() invocations on the same `ReentrantLock`

Solution: Locking and Balking

- A General form of the "balking" idiom

```
booblean done = false;
ReentrantLock lock = new ReentrantLock();
...
while(true){
  lock.lock();
  try{
    if(done) break; // Balk
    ...             // Do some task
  }finally{
    lock.unlock();
  }
}

void setDone(){
  lock.lock();
  try{
    done = true;
  }finally{
    lock.unlock();
  }
}
```

- Threads must use the same instance of `ReentrantLock`.

Be Careful for Potential Race Conditions

- When multiple threads share and access a variable concurrently.
  - Make sure to guard the shared variable
    - By surrounding each read/write logic with lock() and unlock().
- When a loop performs a conditional check with a shared variable (i.e., flag).
  - Surround read logic (i.e., conditional) and write logic (i.e., flag-flipping statement) with lock() and unlock()
  - Try NOT to surround the entire loop with with lock() and unlock()! Why?
    - Does not enjoy concurrency.
    - May result in a deadlock.

Treating the Entire Loop as Atomic Code Does NOT Enjoy Concurrency

- DO **NOT** do this.

```
long n;
lock.lock();
try{
  for(n = from; n <= to; n++){
    if(done==true) break;
    if(isPrime(n)){
      this.primes.add(n); }
  }
}finally{
  lock.unlock();
}
```

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

- Do this.

```
long n;
for(n = from; n <= to; n++){
  lock.lock();
  try{
    if(done==true) break;
    if(isPrime(n)){
      this.primes.add(n); }
  }finally{
    lock.unlock(); }
}
```

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

- If a thread acquires the lock and starts generating prime numbers, it will release the lock when *n* > *to*.
  - No other threads cannot flip the flag until the loop ends.

```
try{
  lock.lock();
  for(n = from; n <= to; n++){
    if(done==true) break;
    if(isPrime(n)){
      this.primes.add(n); }
  }
}finally{
  lock.unlock();
}
```

The purple thread can acquire the lock after all prime numbers have been generated.

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

- This code is thread-safe, but it does not enjoy concurrency.
  - While the green thread generates prime numbers for a given range in between "from" and "to," the purple thread cannot stop the green thread.

```
try{
  lock.lock();
  for(n = from; n <= to; n++){
    if(done==true) break;
    if(isPrime(n)){
      this.primes.add(n); }
  }
}finally{
  lock.unlock();
}
```

lock() returns when the green thread releases the lock.

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock(); }
```

Treating the Entire Loop as Atomic Code May Result in a Deadlock

- DO **NOT** do this.

```
try{
  lock.lock();
  while(!done){
    // Do some task
  }
}finally{
  lock.unlock();
}
```

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

- Do this.

```
while(true){
  lock.lock();
  try{
    if(done) break; // Balk
    // Do some task
  }finally{
    lock.unlock();
  }
}
```

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

HW 6

- If a thread acquires the lock and starts printing #s, it will print #s forever.
  - No other threads cannot flip the flag forever (deadlock!)

```
lock.lock();
while(!done){                    // read logic
System.out.println("#");          // performs some task
}
lock.unlock();
```

The purple thread gets stuck here forever because the green thread never release the lock.

```
lock.lock();
done = true;                     // write logic
lock.unlock();
```

  - In `setDone()`,
    - Use try-finally blocks.
      - Call `unlock()` in a finally block. Always do this in all subsequent HWs.
  - In `generatePrimes()`
    - Use try-finally blocks; Call `unlock()` in a finally block
    - Use *balking* to implement explicit thread termination in a thread-safe manner
    - Do not surround the entire "for" loop with `lock()` and `unlock()`.
- Deadline: March 19 (Tue) midnight

- Revise `RunnableCancellablePrimeGenerator.java` to be thread-safe.
  - Add a `ReentrantLock` to guard the shared variable `done`
  - Revise `generatePrimes()` and `setDone()` to access `done` with the lock