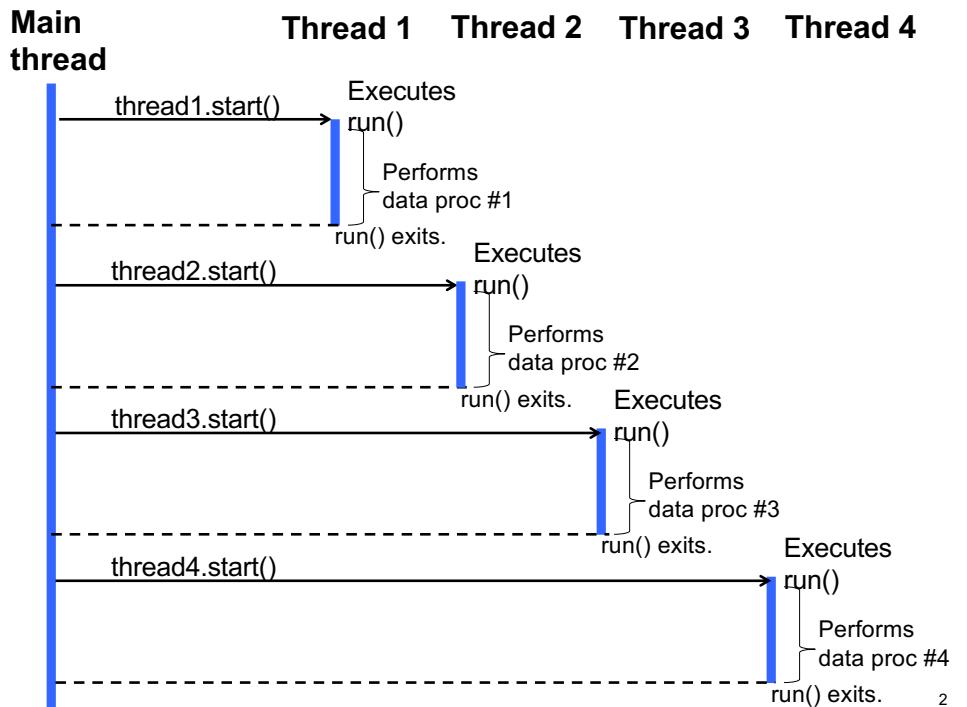


HW 5

- You worked on 4 types of data processing in HW 4.
- Run those data processing with 4 extra threads.
 - One data processing with one extra thread.
 - No need to do anything new, in terms of data processing
 - Reuse data processing code from HW 4.
- Have the main thread
 - wait for 4 data processing threads to be terminated with `join()`
 - collect data processing results from runnable objects.
- This is the last HW that will be due on March 19.



Thread Safety

- Threads are a powerful tool to make your code more **efficient** and **responsive/available**.
- However, multi-threaded code can cause "**thread safety**" issues when it is poorly written.
- 2 major thread safety issues
 - **Race condition (data race)**
 - Messes up the consistency of data shared among threads
 - Breaks the shared data
 - **Deadlock**
 - Makes code execution stuck forever.
- "**Thread-safe**" code is free from these 2 issues.

Race Conditions (Data Races)

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

- Threads run *independently*.

- They NEVER coordinate with each other (by default) regarding their task execution.
 - They don't know if other threads exist and run.
 - They don't know what other threads are doing.
 - They cannot control when their tasks are executed and completed.
 - They cannot control the order of their task execution.
 - C.f. `MCTest`, `RunnablePrimeGenerator`, `RunnablePrimeFactorizer`

- Exception: `Thread.join()` (a.k.a. “after you” operation)

- allows threads to coordinate with each other to some extent.
 - A thread (caller of `join()`) can wait to do something until another thread (callee of `join()`) is terminated.

5

- Threads can share variables (i.e., data fields).

- Exception: Local variables are NEVER shared among threads.

- They can mess up the consistency of the shared 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.

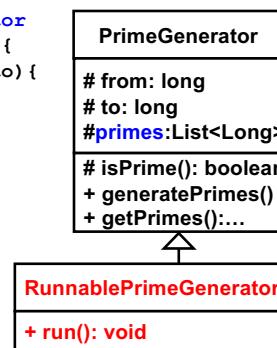
6

Example 1: RunnablePrimeGenerator

```
• class RunnablePrimeGenerator extends PrimeGenerator
    implements Runnable {
    public RunnablePrimeGenerator(long from, long to) {
        super(from, to); }

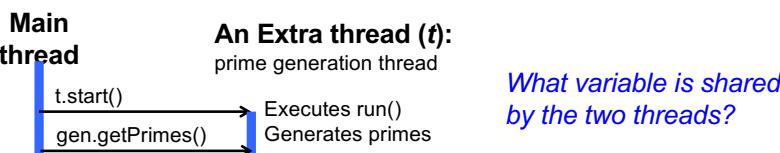
    public void run() {
        generatePrimes(); }

    public List<Long> getPrimes() {
        return primes;
    }
}
```



- Multi-threaded client code

```
• RunnablePrimeGenerator gen =
    new RunnablePrimeGenerator(...);
Thread t = new Thread(gen);
t.start();
gen.getPrimes().forEach(...); //join() is not called!
```



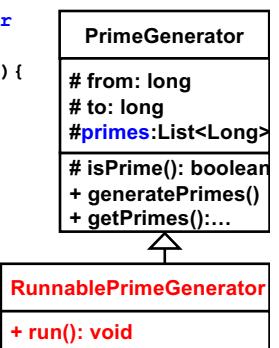
```
• Class RunnablePrimeGenerator extends PrimeGenerator
    implements Runnable {
    public RunnablePrimeGenerator(long from, long to) {
        super(from, to); }

    public void run() {
        generatePrimes(); }

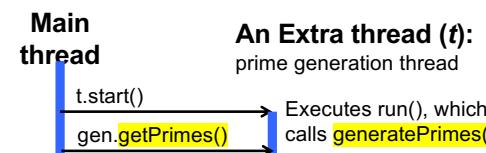
    public List<Long> getPrimes() {
        return primes;
    }
}
```

- Multi-threaded client code

```
• RunnablePrimeGenerator gen =
    new RunnablePrimeGenerator(...);
Thread t = new Thread(gen);
t.start();
gen.getPrimes().forEach(...);
```



The two threads share primes.



7

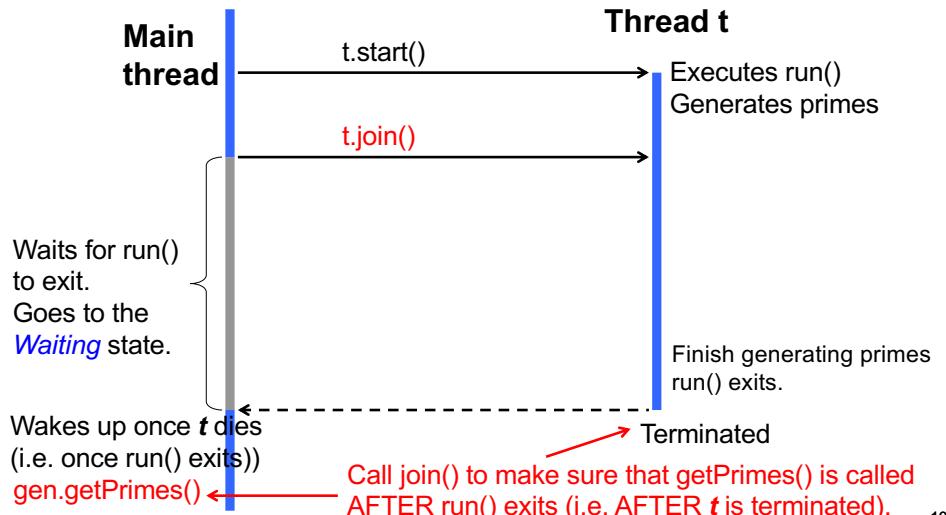
8

Thread.join()

- The best-case (lucky) scenario
 - The main thread obtains the complete set of primes
 - if it happens to call `getPrimes()` after the prime generation thread is terminated.
- Race condition scenarios
 - The main thread obtains an incomplete set of primes
 - if it happens to call `getPrimes()` before the prime generation thread is terminated.
 - The worst-case scenario
 - The main thread and prime gen thread collide on the shared variable `primes` and break prime numbers in it.

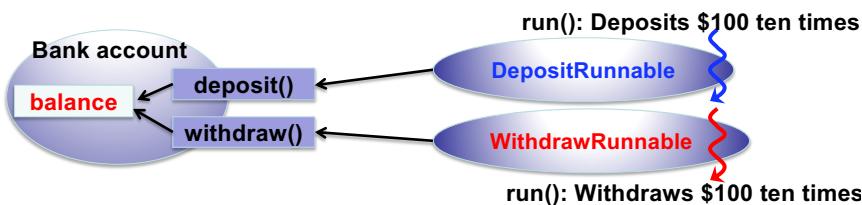
9

- The main thread has to call `join()` to obtain the complete set of primes for sure.



10

Example 2: ThreadUnsafeBankAccount



- 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);
 balance = balance + amount;
 System.out.println(", New balance (d): " + balance);
}

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

```

11

- Desirable 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
- ...

- In reality:

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

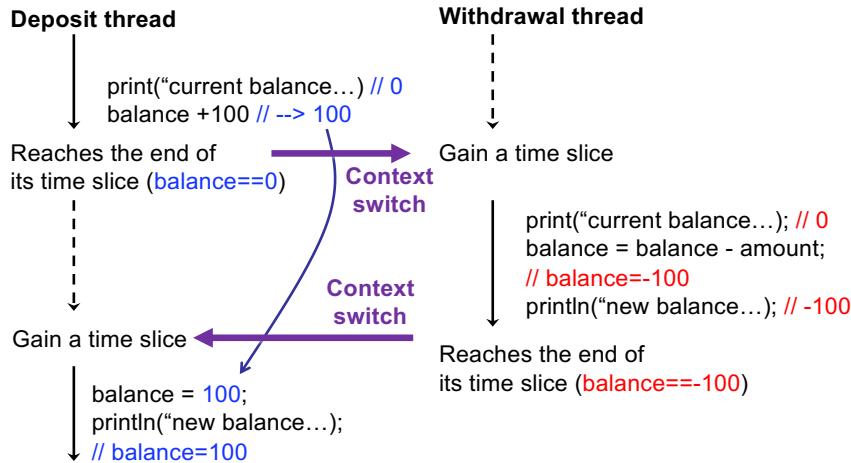
12

12

12

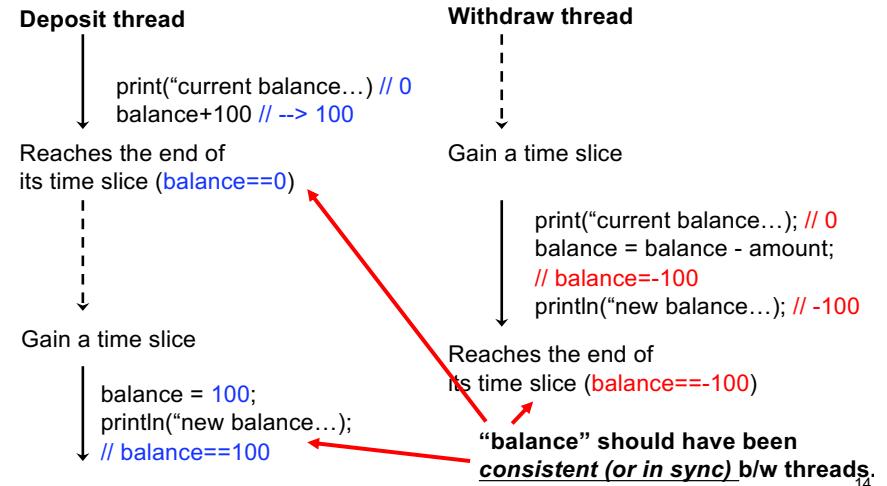
# How Can This Happen?

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



13

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

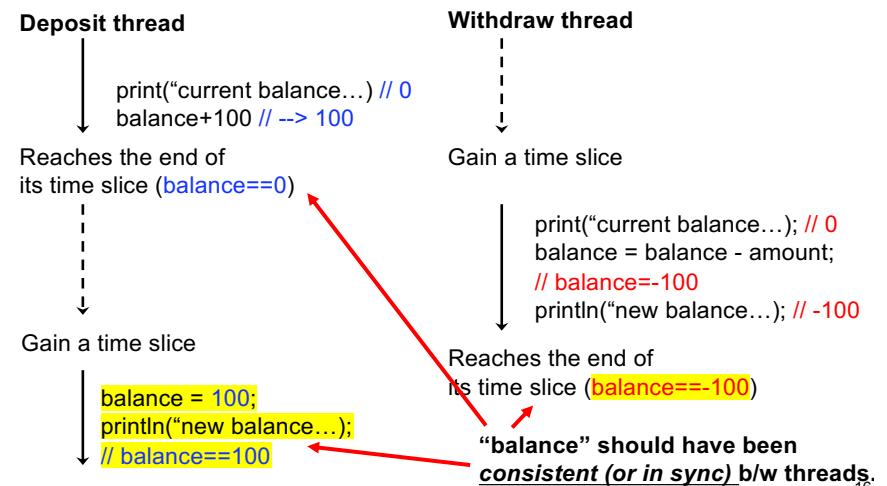


14

## Source of the Problem: Visibility

- **ThreadUnsafeBankAccount** is NOT thread-safe.
  - Race conditions (data races) can occur.
- Race conditions occur due to **visibility** issue.
  - The most up-to-date value of the shared variable **balance** is not visible for all threads.

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

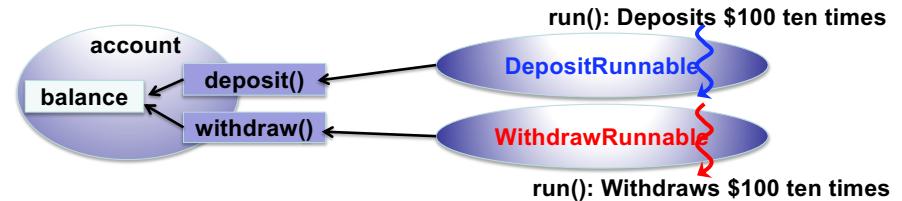


16

# Where can a Context Switch Occur?

- To analyze thread safety, it is important to know where/when a context switch can occur in your code.
- A context switch can occur across atomic operations.

- `intValue = 1;`
  - Sequence of 2 atomic operations
- `balance = balance + amount;`
  - Sequence of 5 atomic operations
- `balance += amount;`
  - Syntactic sugar for `balance = balance + amount;`



- ```
public void deposit(double amount) {  
    balance = balance + amount; }
```
- ```
public void withdraw(double amount) {
 balance = balance - amount; }
```
- Race conditions can occur in the following cases:
  - Has read `balance`, but hasn't read `amount` yet.  
**Context switch** occurs, followed by a balance change by another thread
  - Has read `balance` and `amount`, but hasn't done addition/subtraction yet.  
**Context switch** occurs, followed by a balance change by another thread
  - Has read `balance` and `amount` and finished addition/subtraction, but hasn't updated `balance` yet  
**Context switch** occurs, followed by a balance change by another thread

## 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 byte code instruction.

- 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` has 1 or 2 in the end, 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==true) // 3 steps
- if(done) // 3 steps
- i = j; // 2 steps
- j = i + 1; // 5 steps
 • Reading the value of i, reading/loading the value of 1, summing up i and 1,
 storing the sum to a certain memory space, and assigning it to j.
- i = i + 1; // 5 steps
- i++; // 5 steps
- return i; // 2 steps

- A context switch can occur in between atomic operations/steps.
- A race condition may occur due to the context switch.
```

## 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** has 1L or 2L in the end.
  - **x** can contain another value (e.g., 3L) or corrupted data.
    - An example of corrupted data
      - » The first 32-bit of **x** is assigned by Thread A and the remaining part is assigned by Thread B.
- **aLongVar = 100L;** // More than 2 (~4) steps
- **if(aLongVar==100)** // More than 3 (~6) steps
- **aLongVar++** // More than 5 (~10) steps

22

## Atomicity of Operations for Reference Types

- The **read** and **write** operations for reference types are **atomic**.

- Given **String s**, Thread A does **s="a"** Thread B does **s="b"**
  - **s** has “a” or “b” in the end, depending on which thread performs assignment earlier.
  - **s** never contain other values (e.g., “x”) or corrupted data.

## Compound Operations

- A *compound* of atomic operations is NOT atomic.

- **Account account1 = ...;**
**Account account2 = ...;**
**Account sourceOfWireTransfer = account1;** // 2 steps
**Account destinationOfWireTransfer = account2;** // 2 steps
- **ArrayList<Car> cars = ...;**
**Car car = cars.get(0);** // Many steps
 // ArrayList is NOT thread-safe.
 // All of its public methods are not.
 // Assignment is atomic/1-step though.

# Atomicity of Constructors

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

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

 public Account(double initAmount){ // Assigns a value to the shared
 balance = initAmount; // variable thru 2+ steps...
 } // but...THREAD-SAFE!

 public deposit(float amount){ ... }
 public withdraw(float amount){ ... }
}
```

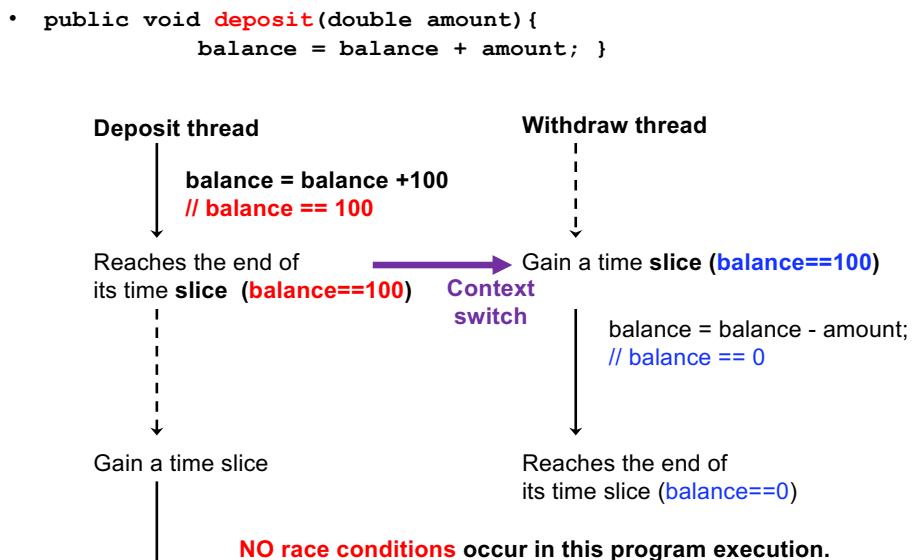
- Until a thread returns/completes a constructor on a class instance, no other threads can call any public methods on that instance.

- Account account = new Account(0);  
 // Account's constructor is thread-safe (kinda atomic).  
 // Assignment is atomic (1-step).  
 // However, a compound of the two 2 ops is NOT atomic.
- Student student = new Student ("John", "Smith");
- Car car = new Car ("Honda", "CR-V", 2010, ...);
- String s = "umb"; // This is a syntactic sugar of:  
• String s = new String("umb");  
 // String's constructor is thread-safe.  
 // Assignment is atomic (1-step).  
 // However, a compound of the two ops is NOT atomic.
- Integer i = 10; // This is a syntactic sugar of:  
• Integer i = Integer.valueOf(10);  
 // valueOf() calls Integer's constructor internally.  
 // Integer's constructor is thread-safe.  
 // Assignment is atomic (1-step).  
 // However, a compound of the two ops is NOT atomic.

# What's Tricky in Multi-threading

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

# Consider this “Lucky” Case



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

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

- No race conditions occur if the “deposit” thread:
  - Hasn’t read `balance`.  
**Context switch**, followed by a balance update by another thread.
  - Has completed to update `balance`  
**Context switch**, followed by a balance update by another thread.

- 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

- The state of a thread has nothing to do with...
  - whether the thread is “actively running” on a CPU core or it is “inactively waiting” for its next turn.
- Both “active” and “inactive” threads can be in the *Runnable* state.
- The *Waiting* or *Blocked* state does NOT mean that a thread is “inactive.”
- Threads
  - change their states by themselves or via thread-to-thread interactions.
  - Has no control over active-inactive cycles (i.e., context switches).



## States of a Thread

