




## Chapter 10: Concurrent and Distributed Programming

# 10.1 Concurrency and Thread-Safety

Ming Liu

May 8, 2019

# Outline

- 
- **What is Concurrent Programming?**
  - **Processes and threads**
  - **Interleaving and race condition**
  - **Thread safety**
    - Strategy 1: Confinement
    - Strategy 2: Immutability
    - Strategy 3: Using Threadsafe Data Types
    - Strategy 4: Locks and Synchronization
  - **How to Make a Safety Argument**
  - **Summary**



# 1 What is Concurrent Programming?

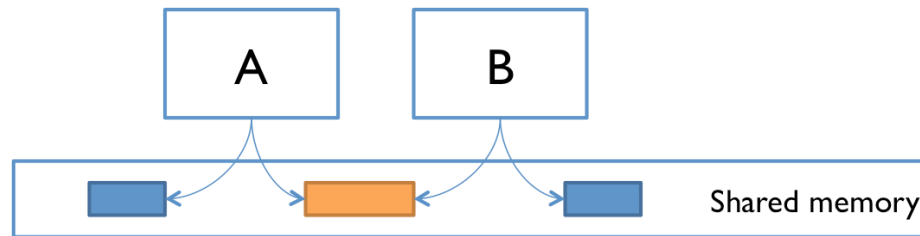


# Concurrency

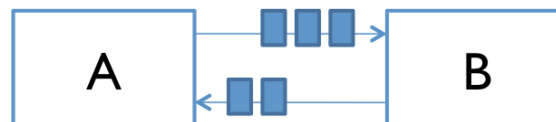
- **Concurrency means multiple computations are happening at the same time.**
- **Concurrency is everywhere in modern programming:**
  - Multiple computers in a network
  - Multiple applications running on one computer
  - Multiple processors in a computer (today, often multiple processor cores on a single chip)
- **Concurrency is essential in modern programming:**
  - Web sites must handle multiple simultaneous users.
  - Mobile apps need to do some of their processing on servers (“in the cloud”).
  - Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you’re still editing it.

# Two Models for Concurrent Programming

- There are two common models for concurrent programming: shared memory and message passing .
  - **Shared memory.** In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.



- **Message passing.** In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling.





# 2 Processes, Threads



# Process and Threads

- The message-passing and shared-memory models are about how concurrent modules communicate.
- The concurrent modules themselves come in two different kinds: **processes and threads**, two basic units of execution.
  - A process is an instance of a running program that is *isolated* from other processes on the same machine. In particular, it has its own private section of the machine's memory.
  - A thread is a locus of control inside a running program. Think of it as a place in the program that is being run, plus the stack of method calls that led to that place (so the thread can go back up the stack when it reaches return statements).



# (1) Process





# Process

- **The process abstraction is a virtual computer** (a self-contained execution environment with a complete, private set of basic run-time resources, in particular, memory space).
  - It makes the program feel like it has the entire machine to itself – like a fresh computer has been created, with fresh memory, just to run that program.
- **Just like computers connected across a network, processes normally share no memory between them.**
  - A process can't access another process's memory or objects at all.
  - Sharing memory between processes is possible on most operating systems, but it needs special effort.
  - By contrast, a new process is automatically ready for message passing, because it is created with standard input & output streams, which are the `System.out` and `System.in` streams you've used in Java.



## (2) Thread



# Thread and Multi-threaded programming

- **Just as a process represents a virtual computer, the thread abstraction represents a *virtual processor*, and threads are sometimes called *lightweight process***
  - Making a new thread simulates making a fresh processor inside the virtual computer represented by the process.
  - This new virtual processor runs the same program and shares the same resources (memory, open files, etc) as other threads in the process, i.e., “threads exist within a process”.
- **Threads are automatically ready for shared memory, because threads share all the memory in the process.**
  - It takes special effort to get “thread-local” memory that’s private to a single thread.
  - It’s also necessary to set up message-passing explicitly, by creating and using queue data structures.

# Threads vs. processes

- Threads are **lightweight**                      Processes **heavyweight**
- Threads **share** address space                      Processes have **own**
- Threads require **synchronization**                      Processes **don't**
  - Threads hold locks while mutating objects

# Thread

- **Multithreaded execution** is an essential feature of the Java platform.
- **Every application has at least one thread.**
- From the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to **create additional threads**.
  
- **Two ways to create a thread:**
  - (Seldom used) Subclassing **Thread**.
  - (More generally used) Implement the **Runnable** interface and use the **new Thread(..)** constructor.

# Ways to create a thread

## ■ Subclass **Thread**

- The **Thread** class itself implements **Runnable**, though its run method does nothing. An application can subclass

**Thread**, providing its own implementation of **run()**.

```
public class HelloThread extends Thread {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

## ■ Provide a **Runnable** object

- The **Runnable** interface defines a single method, **run()**, meant to contain the code executed in the thread. The **Runnable** object is passed to the **Thread** constructor.

## ■ To invoke **Thread.start()** in order to start the new thread.

```
public class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

# Ways to create a thread

- A very common idiom is starting a thread with an anonymous **Runnable**, which eliminates the named class:

```
new Thread(new Runnable() {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
}).start();
```


- The **Runnable** interface represents the work to be done by a thread.

```
public interface Runnable {  
    void run();  
}
```

# sleep() and interrupt()

- **Pausing Execution with `Thread.sleep(time)`:** causes the current thread to suspend execution for a specified period.
  - This is an efficient means of making processor time available to the other threads or other applications that might be running on the same computer.
- **To interrupt a running thread using `interrupt()` method and check if a thread is interrupted using `isInterrupted()` method.**
  - If the thread is frequently invoking methods that throw `InterruptedException`, it simply returns from the run method after it catches that exception.





## sleep() and interrupt()

```
public class GeneralInterrupt implements Runnable {
    public void run() {
        try {
            work();
        } catch (InterruptedException x) {
            return;
        }
    }

    public void work() throws InterruptedException {
        while (true) {
            if (Thread.currentThread().isInterrupted()) {
                Thread.sleep(2000);
            }
        }
    }

    public static void main(String[] args) {
        GeneralInterrupt si = new GeneralInterrupt();
        Thread t = new Thread(si);
        t.start();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException x) { }

        t.interrupt();
    }
}
```

# join()

- The `join()` method is used to hold the execution of currently running thread until the specified thread is dead (finished execution).
  - In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads.
  - by using `join()` method, we can make one thread to wait for another.

```
public class JoinExample2 {  
    public static void main(String[] args) {  
        Thread th1 = new Thread(new MyClass2(), "th1");  
        Thread th2 = new Thread(new MyClass2(), "th2");  
        Thread th3 = new Thread(new MyClass2(), "th3");  
  
        th1.start();  
        th2.start();  
        th3.start();  
    }  
}
```

# join()

```
public class JoinExample {
    public static void main(String[] args) {
        Thread th1 = new Thread(new MyClass(), "th1");
        Thread th2 = new Thread(new MyClass(), "th2");
        Thread th3 = new Thread(new MyClass(), "th3");

        th1.start();

        try {
            th1.join();
        } catch (InterruptedException ie) {}

        th2.start();

        try {
            th2.join();
        } catch (InterruptedException ie) {}

        th3.start();

        try {
            th3.join();
        } catch (InterruptedException ie) {}
    }
}
```

# Why use threads?

- **Performance in the face of blocking activities**
  - Consider a web server
- **Performance on multiprocessors**
- **Cleanly dealing with natural concurrency**
- **In Java threads are a fact of life**
  - Example: garbage collector runs in its own thread

Number of Threads	Seconds to run
1	22.0
2	13.5
3	11.7
4	10.8



# 3 Interleaving and Race Condition

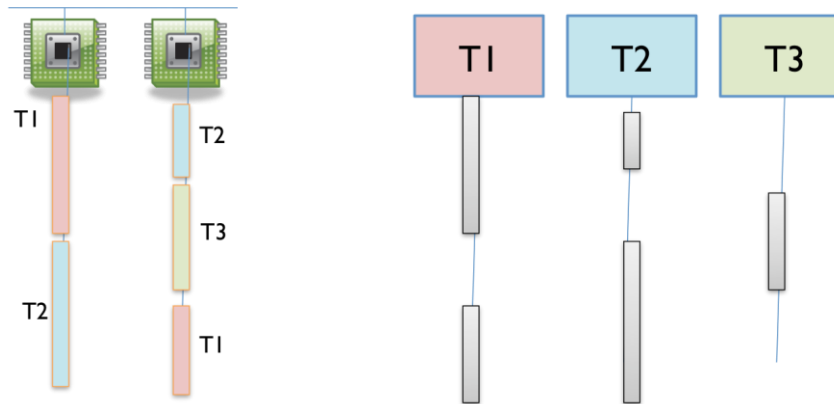


# Time slicing

- In computer systems that have a single execution core, only one thread is actually executing at any given moment.
  - Processing time for a single core is shared among processes and threads through an OS feature called **time slicing**.
  
- Today's computer systems to have multiple processors or processors with multiple execution cores. **So, how can I have many concurrent threads with only one or two processors in my computer?**
  - When there are more threads than processors, concurrency is simulated by **time slicing**, which means that the processor switches between threads.

# An example of time slicing

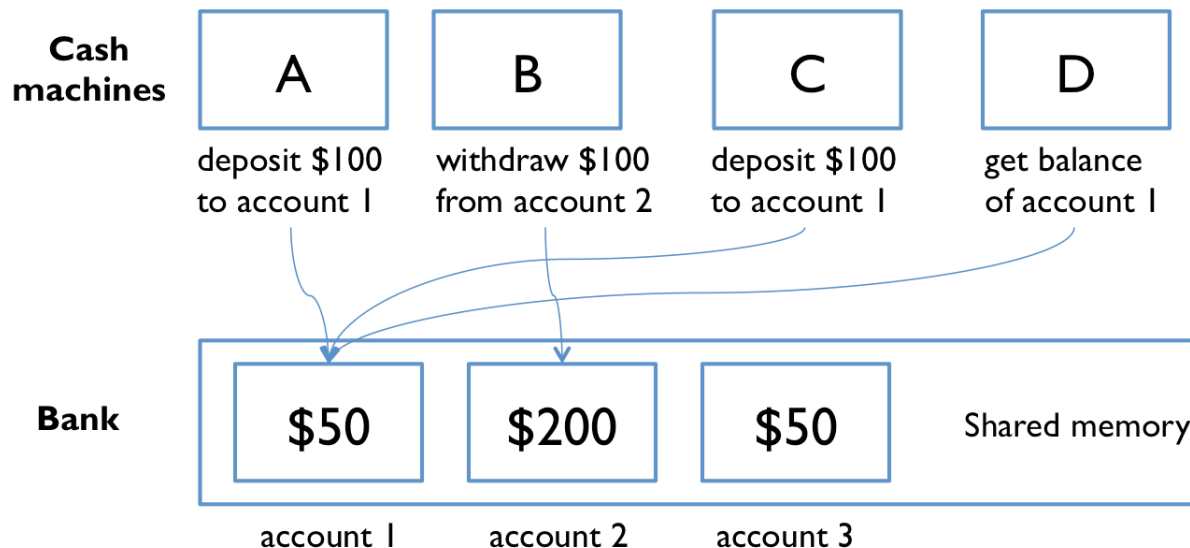
- **Three threads T1, T2, and T3 might be time-sliced on a machine that has two actual processors.**
  - At first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3.
  - Thread T2 simply pauses, until its next time slice on the same processor or another processor.



- **On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time.**

# Shared Memory Example

- **Shared memory could induce subtle bugs !!!**
- **Example:** a bank has cash machines that use a shared memory model, so all the cash machines can read and write the same account objects in memory.





# Shared Memory Example

- To illustrate what can go wrong, let's simplify the bank down to a single account, with a dollar balance stored in the balance variable, and two operations deposit and withdraw that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account
private static int balance = 0;

private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

- Customers use the cash machines to do transactions like this:

```
deposit(); // put a dollar in
withdraw(); // take it back out
```

# Shared Memory Example

- **Every transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged.**

- Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions.

```
// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
        deposit(); // put a dollar in
        withdraw(); // take it back out
    }
}
```

- **At the end of the day, regardless of how many cash machines were running, or how many transactions we processed, we should expect the account balance to still be 0.**
- But if we run this code, we discover frequently that the balance at the end of the day is not 0. If more than one `cashMachine()` call is running at the same time – say, on separate processors in the same computer – then balance may not be zero at the end of the day. Why not?

# Interleaving

- Suppose two cash machines, A and B, are both working on a deposit at the same time. Here's how the deposit() step typically breaks down into low-level processor instructions:
- When A and B are running concurrently, these low-level instructions interleave with each other...

```

get balance (balance=0)
add 1
write back the result (balance=1)

```

A	B
A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=2)

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result (balance=1)	
	B write back the result (balance=1)

# Race Condition

- **The balance is now 1 – A's dollar was lost!**

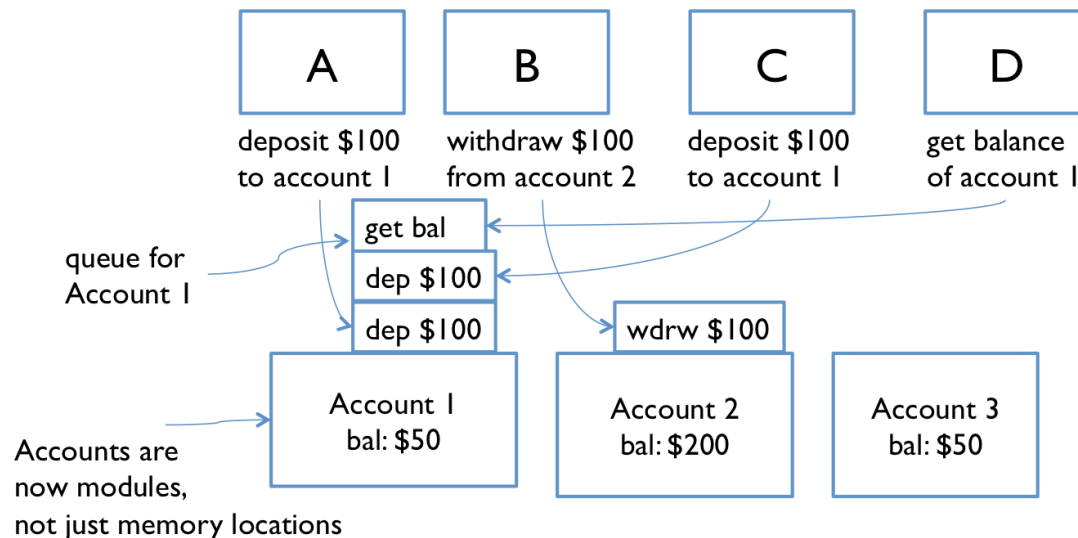
- A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other's deposit into account.

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result (balance=1)	
	B write back the result (balance=1)

- This is called **race condition**: the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations A and B.
- When this happens, we say “A is in a race with B.”
- Or called “Thread Interference”

# Message Passing Example

- Now not only are the cash machine modules, but the accounts are modules, too.
- Modules interact by sending messages to each other.
  - Incoming requests are placed in a queue to be handled one at a time.
  - The sender doesn't stop working while waiting for an answer to its request. It handles more requests from its own queue. The reply to its request eventually comes back as another message.



# Can message-passing solve race condition?

- **Unfortunately, message passing doesn't eliminate the possibility of race conditions.**
  - Suppose each account supports get-balance and withdraw operations, with corresponding messages.
  - Two users, at cash machines A and B, are both trying to withdraw a dollar from the same account.
  - They check the balance first to make sure they never withdraw more than the account holds, because overdrafts trigger big bank penalties.
- **The problem is again interleaving, but this time interleaving of the *messages* sent to the bank account, rather than the *instructions* executed by A and B.**
  - If the account starts with a dollar in it, then what interleaving of messages will fool A and B into thinking they can both withdraw a dollar, thereby overdrawing the account?

# Concurrency is hard to test and debug !

- **It's very hard to discover race conditions using testing.**
  - Even once a test has found a bug, it may be very hard to localize it to the part of the program causing it. ----WHY?
- **Concurrency bugs exhibit very poor reproducibility.**
  - It's hard to make them happen the same way twice.
  - Interleaving of instructions or messages depends on the relative timing of events that are strongly influenced by the environment.
  - Delays are caused by other running programs, other network traffic, OS scheduling decisions, variations in processor clock speed, etc.
  - Each time you run a program containing a race condition, you may get different behavior.

*Heisenbugs*

*Bohrbugs*

nondeterministic and hard to reproduce

# Concurrency is hard to test and debug !

- **A heisenbug may even disappear when you try to look at it with println or debugger!**
  - The reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving.
- **So inserting a simple print statement into the cashMachine():**

```
private static void cashMachine() {  
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {  
        deposit(); // put a dollar in  
        withdraw(); // take it back out  
        System.out.println(balance); // makes the bug disappear!  
    }  
}
```

...and suddenly the balance is always 0, as desired, and the bug appears to disappear. But it's only masked, not truly fixed.



# A short summary

- **Concurrency:** multiple computations running simultaneously
- **Shared-memory & message-passing** paradigms
- **Processes & threads**
  - Process is like a virtual computer; thread is like a virtual processor
- **Race conditions**
  - When correctness of result (postconditions and invariants) depends on the relative timing of events
  - Multiple threads **sharing the same mutable variable** without coordinating what they're doing.
  - This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.



# 4 Thread Safety



# What **threadsafe** means

- A data type or static method is **threadsafe** if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code.
  - “behaves correctly” means satisfying its specification and preserving its rep invariant;
  - “regardless of how threads are executed” means threads might be on multiple processors or timesliced on the same processor;
  - “without additional coordination” means that the data type can’t put preconditions on its caller related to timing, like “you can’t call `get()` while `set()` is in progress.”
- Remember Iterator ? It’s not threadsafe. Iterator’s specification says that you can’t modify a collection at the same time as you’re iterating over it. That’s a timing-related precondition put on the caller, and Iterator makes no guarantee to behave correctly if you violate it.

# Four ways of threadsafe

Don't share: isolate mutable state in individual threads

Don't mutate: share only immutable state

If must share mutable state, **synchronize**

- **Confinement.** Don't share the variable between threads.
- **Immutability.** Make the shared data immutable. There are some additional constraints for concurrent programming.
- **Threadsafe data type.** Encapsulate the shared data in an existing threadsafe data type that does the coordination for you.
- **Synchronization.** Use synchronization to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own threadsafe data type.



# Strategy 1: Confinement



# Strategy 1: Confinement

- **Thread confinement is a simple idea:**
  - You avoid races on mutable data by keeping that data confined to a single thread.
  - Don't give any other threads the ability to read or write the data directly.
- **Since shared mutable data is the root cause of a race condition, confinement solves it by *not sharing* the mutable data.**
  - Local variables are always thread confined. A local variable is stored in the stack, and each thread has its own stack. There may be multiple invocations of a method running at a time, but each of those invocations has its own private copy of the variable, so the variable itself is confined.
  - If a local variable is an object reference, you need to check the object it points to. If the object is mutable, then we want to check that the object is confined as well – there can't be references to it that are reachable from any other thread.

# Strategy 1: Confinement

```
public class Factorial {  
  
    /**  
     * Computes n! and prints it on standard output.  
     * @param n must be >= 0  
     */  
    private static void computeFact(final int n) {  
        BigInteger result = new BigInteger("1");  
        for (int i = 1; i <= n; ++i) {  
            System.out.println("working on fact " + n);  
            result = result.multiply(new BigInteger(String.valueOf(i)));  
        }  
        System.out.println("fact(" + n + ") = " + result);  
    }  
  
    public static void main(String[] args) {  
        new Thread(new Runnable() { // create a thread using an  
            public void run() {      // anonymous Runnable  
                computeFact(99);  
            }  
        }).start();  
        computeFact(100);  
    }  
}
```

# Avoid Global Variables

- **Global static variables are not automatically thread confined.**
- If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly.
- **Better, you should eliminate the static variables entirely.**

```
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if x is prime with high probability
 */
public static boolean isPrime(int x) {
    if (cache.containsKey(x)) return cache.get(x);
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
    cache.put(x, answer);
    return answer;
}

private static Map<Integer, Boolean> cache = new HashMap<>();
```



# Avoid Global Variables

- This class has a race in the `getInstance()` method – two threads could call it at the same time and end up creating two copies of the `PinballSimulator` object, which violates the rep invariant.

```
// This class has a race condition in it.
public class PinballSimulator {

    private static PinballSimulator simulator = null;
    // invariant: there should never be more than one PinballSimulator
    //              object created

    private PinballSimulator() {
        System.out.println("created a PinballSimulator object");
    }

    // factory method that returns the sole PinballSimulator object,
    // creating it if it doesn't exist
    public static PinballSimulator getInstance() {
        if (simulator == null) {
            simulator = new PinballSimulator();
        }
        return simulator;
    }
}
```

To fix this race using the thread confinement approach, you would specify that only a certain thread is allowed to call `getInstance()`.



# Strategy 2: Immutability



## Strategy 2: Immutability

- The second way of achieving thread safety is **by using immutable references and data types**.
  - Immutability tackles the shared-mutable-data cause of a race condition and solves it simply by making the shared data *not mutable*.
- **final** variables are immutable references, so a variable declared **final** is safe to access from multiple threads.
  - You can only read the variable, not write it.
  - Because this safety applies only to the variable itself, and we still have to argue that the object the variable points to is immutable.

## Strategy 2: Immutability

- **Immutable objects are usually also threadsafe.**
- **We say “usually” here because our current definition of immutability is too loose for concurrent programming.**
  - A type is immutable if an object of the type always represents the same abstract value for its entire lifetime.
  - But that actually allows the type the freedom to mutate its rep, as long as those mutations are invisible to clients, such as **beneficent mutation**.
- **For concurrency, this kind of hidden mutation is not safe.**
  - An immutable data type that uses beneficent mutation will have to make itself threadsafe using locks.

# Stronger definition of immutability

- **In order to be confident that an immutable data type is threadsafe without locks, we need a stronger definition of immutability:**
  - No mutator methods
  - All fields are private and final
  - No representation exposure
  - **No mutation whatsoever of mutable objects in the rep – not even beneficent mutation**
- **If you follow these rules, then you can be confident that your immutable type will also be threadsafe.**



# Strategy 3: Using Threadsafe Data Types



## Strategy 3: Using Threadsafe Data Types

- The third major strategy for achieving thread safety is **to store shared mutable data in existing threadsafe data types**.
- When a data type in the Java library is threadsafe, its documentation will explicitly state that fact.
- It's become common in the Java API to find two mutable data types that do the same thing, one threadsafe and the other not.
- The reason is what this quote indicates: **threadsafe data types usually incur a performance penalty compared to an unsafe type**.

# Threadsafe Collections

- The collection interfaces in Java – **List** , **Set** , **Map** – have basic implementations that are not threadsafe.
  - The implementations namely **ArrayList** , **HashMap** , and **HashSet** , cannot be used safely from more than one thread.
- Just like the Collections API provides wrapper methods that make collections immutable, it provides another set of wrapper methods to make collections threadsafe, while still mutable.
  - These wrappers effectively make each method of the collection atomic with respect to the other methods.
  - An atomic action effectively happens all at once – it doesn't interleave its internal operations with those of other actions, and none of the effects of the action are visible to other threads until the entire action is complete, so it never looks partially done.

```
private static Map<Integer, Boolean> cache =  
    Collections.synchronizedMap(new HashMap<>());
```



# A few points



## ■ Don't circumvent the wrapper.

- Make sure to throw away references to the underlying non-threadsafe collection, and access it only through the synchronized wrapper.
- The new `HashMap` is passed only to `synchronizedMap()` and never stored anywhere else.
- The underlying collection is still mutable, and code with a reference to it can circumvent immutability.

# A few points

- **Iterators are still not threadsafe.**

- Even though method calls on the collection itself ( `get()` , `put()` , `add()` , etc.) are now threadsafe, iterators created from the collection are still not threadsafe.
- So you can't use `iterator()` , or the for loop syntax:

```
for (String s: lst) { ... }
```

**// not threadsafe, even if lst is a synchronized list wrapper**

- The solution to this iteration problem will be to acquire the collection's lock when you need to iterate over it.

# A few points

- **Atomic operations aren't enough to prevent races: the way that you use the synchronized collection can still have a race condition.**
- **Consider this code, which checks whether a list has at least one element and then gets that element:**  

```
if ( ! lst.isEmpty()) { String s = lst.get(0); ... }
```
- **Even if you make `lst` into a synchronized list, this code still may have a race condition, because another thread may remove the element between the `isEmpty()` call and the `get()` call.**

# Threadsafe wrappers

- Wrapper implementations delegate all their real work to a specified collection but add extra functionality on top of what this collection offers.
- This is an example of the decorator pattern.
- These implementations are anonymous; rather than providing a public class, the library provides a static factory method.
- All these implementations are found in the `Collections` class, which consists solely of static methods.
- The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection.

# Threadsafe wrappers

## ■ Wrappers:

- `public static <T> Collection<T> synchronizedCollection(Collection<T> c);`
- `public static <T> Set<T> synchronizedSet(Set<T> s);`
- `public static <T> List<T> synchronizedList(List<T> list);`
- `public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);`
- `public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);`
- `public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);`

## ■ Usage:

```
List<Type> c = Collections.synchronizedList(new ArrayList<Type>());  
synchronized(c) {    // to be introduced later (the 4-th threadsafe way)  
    for (Type e : c)  
        foo(e);  
}
```



# Strategy 4: Locks and Synchronization



# Recall

- **Thread safety for a data type or a function:** behaving correctly when used from multiple threads, regardless of how those threads are executed, without additional coordination.

**Principle: the correctness of a concurrent program should not depend on accidents of timing .**

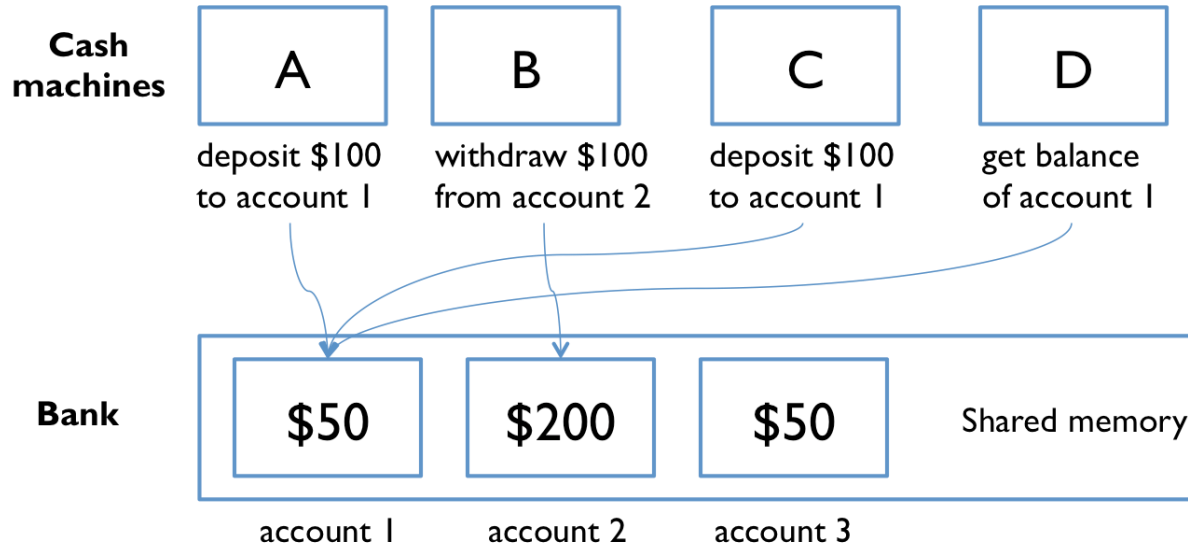
- **There are four strategies for making code safe for concurrency:**
  - Confinement : don't share data between threads.
  - Immutability : make the shared data immutable.
  - Use existing threadsafe data types : use a data type that does the coordination for you.
  - **Synchronization:** prevent threads from accessing the shared data at the same time. This is what we use to implement a threadsafe type, but we didn't discuss it at the time.

# Synchronization and Locks

- Since race conditions caused by concurrent manipulation of shared mutable data are disastrous bugs — hard to discover, reproduce, and debug — we need a way for concurrent modules that share memory to **synchronize** with each other.
- **Locks** are one synchronization technique.
  - A lock is an abstraction that allows at most one thread to *own* it at a time. *Holding a lock* is how one thread tells other threads: “I’m changing this thing, don’t touch it right now.”
- **Locks have two operations:**
  - **acquire** allows a thread to take ownership of a lock. If a thread tries to acquire a lock currently owned by another thread, it blocks until the other thread releases the lock. At that point, it will contend with any other threads that are trying to acquire the lock. At most one thread can own the lock at a time.
  - **release** relinquishes ownership of the lock, allowing another thread to take ownership of it.

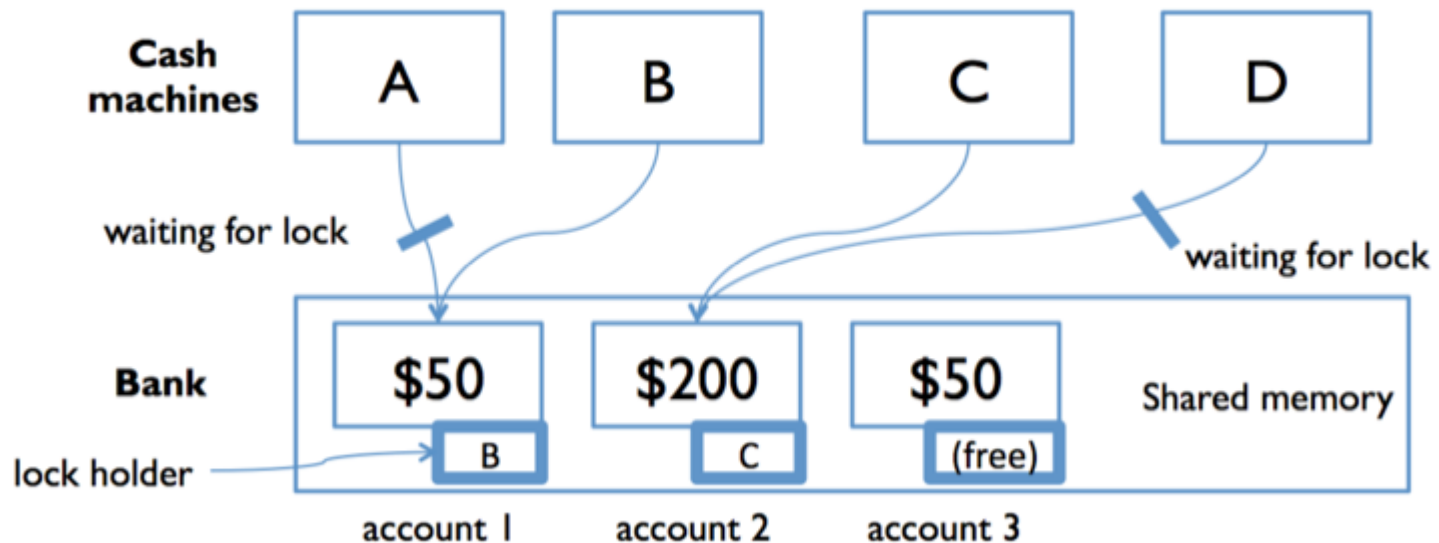


# Bank account example



- To solve this problem with locks, we can add a lock that protects each bank account.
- Now, before they can access or update an account balance, cash machines must first acquire the lock on that account.

# Bank account example



- Both A and B are trying to access account 1.
- Suppose B acquires the lock first. Then A must wait to read and write the balance until B finishes and releases the lock.
- This ensures that A and B are synchronized, but another cash machine C is able to run independently on a different account (because that account is protected by a different lock).

# Recall: Steps to develop an ADT

- **Specify:** define the operations (method signatures and specs).
  - **Test:** develop test cases for the operations.
  - **Rep: choose a rep.**
    - Implement a simple, brute-force rep first.
    - Write down the rep invariant and abstraction function, and implement `checkRep()` which asserts the rep invariant at the end of every constructor, producer, and mutator method.
- 
- **+++ Synchronize**
    - Make an argument that your rep is threadsafe.
    - Write it down explicitly as a comment in your class, right by the rep invariant, so that a maintainer knows how you designed thread safety into the class.

# Use synchronization to develop a threadsafe ADT

- Suppose we're building a multi-user editor that allows multiple people to connect to it and edit it at the same time.
- We'll need a mutable datatype to represent the text in the document.
- Here's the interface → basically it represents a string with insert and delete operations.

```

/** An EditBuffer represents a threadsafe mutable
 * string of characters in a text editor. */
public interface EditBuffer {
    /**
     * Modifies this by inserting a string.
     * @param pos position to insert at
     *             (requires 0 <= pos <= current buffer length)
     * @param ins string to insert
     */
    public void insert(int pos, String ins);

    /**
     * Modifies this by deleting a substring
     * @param pos starting position of substring to delete
     *             (requires 0 <= pos <= current buffer length)
     * @param len length of substring to delete
     *             (requires 0 <= len <= current buffer length - pos)
     */
    public void delete(int pos, int len);

    /**
     * @return length of text sequence in this edit buffer
     */
    public int length();

    /**
     * @return content of this edit buffer
     */
    public String toString();
}

```

# Three Reps for EditBuffer

- **A String**

- `private String text;`

Every time we do an insert or delete, we have to copy the entire string into a new string. That gets expensive.

- **A character array, with space at the end.**

- `private char[] text;`

If the user is typing at the beginning of the document, then we're copying the entire document with every keystroke.

- **A gap buffer**

- A character array with extra space in it, but instead of having all space at the end, the extra space is a *gap* that can appear anywhere in the buffer.
- Whenever an insert or delete operation is needed, the datatype first moves the gap to the location of the operation, and then does the insert or delete.
- If the gap is already there, then nothing needs to be copied — an insert just consumes part of the gap, and a delete just enlarges the gap.
- Gap buffers are particularly well-suited to representing a string that is being edited by a user with a cursor, since inserts and deletes tend to be focused around the cursor, so the gap rarely moves.

# Gap buffer

Initial state:

This is the way [            ]out.

User inserts some new text:

This is the way the world started [   ]out.

User moves the cursor before "started"; system moves "started " from the first buffer to the second buffer.

This is the way the world [   ]started out.

User adds text filling the gap; system creates new gap:

This is the way the world as we know it [            ]started out.

# Three Reps for EditBuffer

- **Gap buffer**

[illegible]



# (1) Synchronized Blocks and Methods





# Locking

- **Locks are so commonly-used that Java provides them as a built-in language feature.**
  - Every object has a lock implicitly associated with it — a `String`, an array, an `ArrayList`, and every class and all of their object instances have a lock.
  - Even a humble `Object` has a lock, so bare `Object` are often used for explicit locking:

```
Object lock = new Object();
```

- **You can't call `acquire` and `release` on Java's intrinsic locks, however. Instead you use the `synchronized` statement to acquire the lock for the duration of a statement block:**

```
synchronized (lock) { // thread blocks here until lock is free
    // now this thread has the lock
    balance = balance + 1;
    // exiting the block releases the lock
}
```

# Locking

- Synchronized regions like this provide **mutual exclusion**: only one thread at a time can be in a synchronized region guarded by a given object's lock.
- In other words, you are back in sequential programming world, with only one thread running at a time, at least with respect to other synchronized regions that refer to the same object.

```
synchronized (lock) { // thread blocks here until lock is free
    // now this thread has the lock
    balance = balance + 1;
    // exiting the block releases the lock
}
```

# Monitor pattern

- When you are writing methods of a class, the most convenient lock is the object instance itself, i.e. **this**.
- As a simple approach, we can guard the entire rep of a class by wrapping all accesses to the rep inside **synchronized(this)**.
- **Monitor pattern:** a monitor is a class whose methods are mutually exclusive, so that only one thread can be inside an instance of the class at a time.

```

/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        synchronized (this) {
            text = "";
            checkRep();
        }
    }
    public void insert(int pos, String ins) {
        synchronized (this) {
            text = text.substring(0, pos) + ins + text.substring(pos);
            checkRep();
        }
    }
    public void delete(int pos, int len) {
        synchronized (this) {
            text = text.substring(0, pos) + text.substring(pos+len);
            checkRep();
        }
    }
    public int length() {
        synchronized (this) {
            return text.length();
        }
    }
    public String toString() {
        synchronized (this) {
            return text;
        }
    }
}

```

# Monitor pattern

- If you add the keyword **synchronized** to a method signature, then Java will act as if you wrote **synchronized(this)** around the method body.
- What's the difference between a **synchronized** method and a **synchronized(this)** block?

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        text = "";
        checkRep();
    }
    public synchronized void insert(int pos, String ins) {
        text = text.substring(0, pos) + ins + text.substring(pos);
        checkRep();
    }
    public synchronized void delete(int pos, int len) {
        text = text.substring(0, pos) + text.substring(pos+len);
        checkRep();
    }
    public synchronized int length() {
        return text.length();
    }
    public synchronized String toString() {
        return text;
    }
}
```

# Synchronized Methods

- It is not possible for two invocations of synchronized methods on the same object to interleave.
- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- When a synchronized method exits, it automatically establishes a **happens-before** relationship with *any subsequent invocation* of a synchronized method for the same object.
  - This guarantees that changes to the state of the object are visible to all threads.

# Synchronized Statements/Block

- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock.
- Synchronized statements are useful for improving concurrency with **fine-grained synchronization**.

```
public class MsLunch {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
  
    public void inc2() {  
        synchronized(lock2) {  
            c2++;  
        }  
    }  
}
```

# Thread safety argument with synchronization

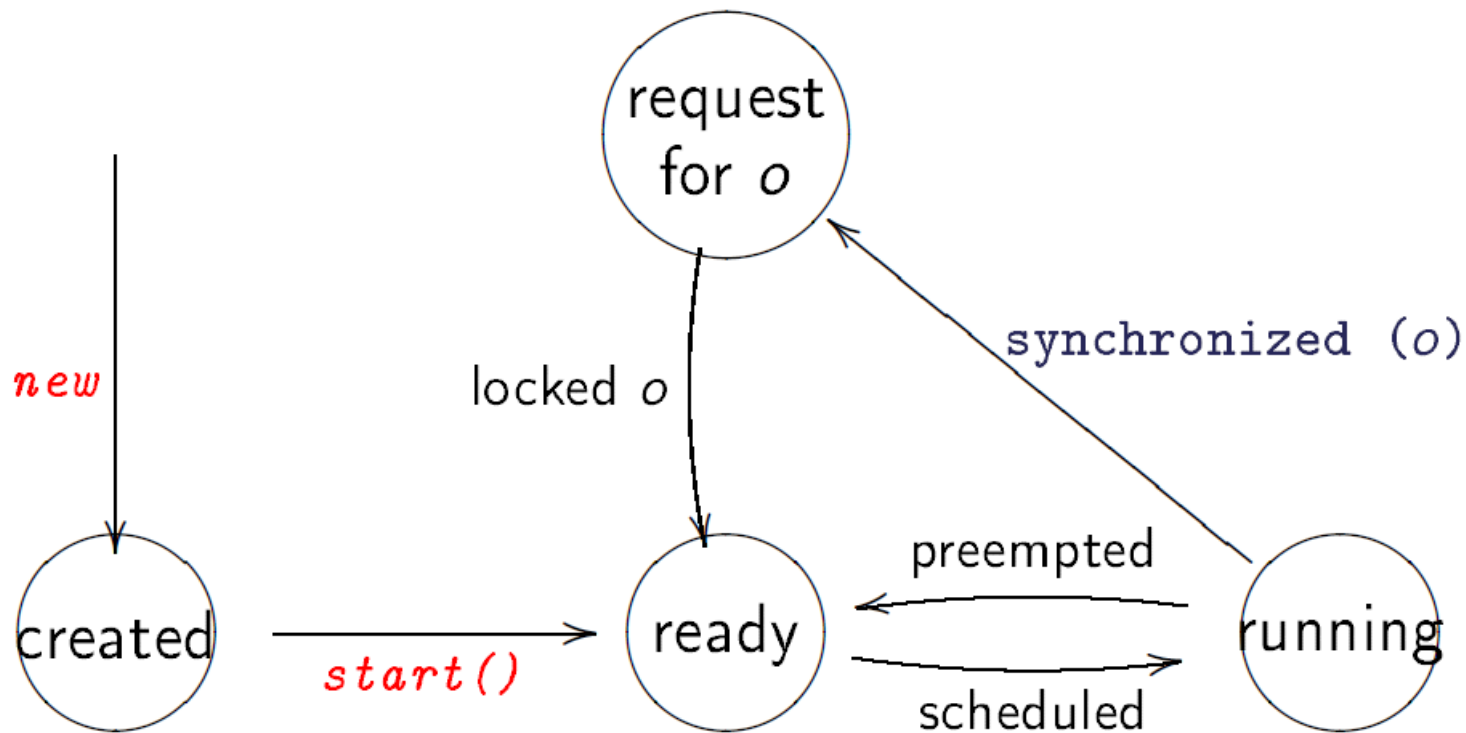
```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */  
public class SimpleBuffer implements EditBuffer {  
    private String text;  
    // Rep invariant:  
    //   text != null  
    // Abstraction function:  
    //   represents the sequence text[0],...,text[text.length()-1]  
    // Thread safety argument:  
    //   all accesses to text happen within SimpleBuffer methods,  
    //   which are all guarded by SimpleBuffer's lock
```

# Locking discipline

- A locking discipline is a strategy for ensuring that synchronized code is threadsafe.
- We must satisfy two conditions:
  - Every shared mutable variable must be guarded by some lock. The data may not be read or written except inside a synchronized block that acquires that lock.
  - If an invariant involves multiple shared mutable variables (which might even be in different objects), then all the variables involved must be guarded by the *same* lock. Once a thread acquires the lock, the invariant must be reestablished before releasing the lock.
- The monitor pattern as used here satisfies both rules. All the shared mutable data in the rep — which the rep invariant depends on — are guarded by the same lock.



# State model for Java threads: locking





## (2) Atomic operations



# Keyword **volatile** for Atomic data Access

```
private volatile int counter;
```

- Using **volatile** variables reduces the risk of memory consistency errors, because any write to a **volatile** variable establishes a **happens-before** relationship with subsequent reads of that same variable.
- This means that changes to a **volatile** variable are always visible to other threads.
- What's more, it also means that when a thread reads a **volatile** variable, it sees not just the latest change to the **volatile**, but also the side effects of the code that led up the change.
- This is a lightweight synchronization mechanism.

# But **volatile** cannot deal with all situations

```
private static volatile int counter = 0;
```

```
private void concurrentMethodWrong() {  
    counter = counter + 5;  
    //do something  
    counter = counter - 5;  
}
```

```
private static final Object counterLock = new Object();
```

```
private static volatile int counter = 0;
```

```
private void concurrentMethodRight() {  
    synchronized (counterLock) {  
        counter = counter + 5;  
    }  
    //do something  
    synchronized (counterLock) {  
        counter = counter - 5;  
    }  
}
```

# Atomic operations

- Consider a find-and-replace operation on the EditBuffer datatype:

```
/** Modifies buf by replacing the first occurrence of s with t.
 * If s not found in buf, then has no effect.
 * @returns true if and only if a replacement was made
 */
public static boolean findReplace(EditBuffer buf, String s, String t) {
    int i = buf.toString().indexOf(s);
    if (i == -1) {
        return false;
    }
    buf.delete(i, s.length());
    buf.insert(i, t);
    return true;
}
```

- This method makes three different calls to buf. Even though each of these calls individually is atomic, the findReplace method as a whole is not threadsafe, because other threads might mutate the buffer while findReplace is working, causing it to delete the wrong region or put the replacement back in the wrong place.
- To prevent this, findReplace needs to synchronize with all other clients of buf .

# Giving clients access to a lock

- It's sometimes useful to make your datatype's lock available to clients, so that they can use it to implement higher-level atomic operations using your datatype.
- So one approach to the problem with `findReplace` is to document that clients can use the `EditBuffer`'s lock to synchronize with each other:

```
/** An EditBuffer represents a threadsafe mutable string of characters
 * in a text editor. Clients may synchronize with each other using the
 * EditBuffer object itself. */
public interface EditBuffer {
    ...
}
```

```
public static boolean findReplace(EditBuffer buf, String s, String t) {
    synchronized (buf) {
        int i = buf.toString().indexOf(s);
        if (i == -1) {
            return false;
        }
        buf.delete(i, s.length());
        buf.insert(i, t);
        return true;
    }
}
```

To ensure that all three methods are executed without interference from other threads.

# To implement higher-level atomic operations

- The effect of this is to enlarge the synchronization region that the monitor pattern already put around the individual `toString`, `delete`, and `insert` methods, into a single atomic region that ensures that all three methods are executed without interference from other threads.

```
public static boolean findReplace(EditBuffer buf, String s, String t) {  
    synchronized (buf) {  
        int i = buf.toString().indexOf(s);  
        if (i == -1) {  
            return false;  
        }  
        buf.delete(i, s.length());  
        buf.insert(i, t);  
        return true;  
    }  
}
```



(3) Sprinkling **synchronized**  
everywhere?





# Sprinkling **synchronized** everywhere?

- So is thread safety simply a matter of putting the **synchronized** keyword on every method in your program? Unfortunately not.
- First, you actually don't want to synchronize methods willy-nilly.
  - **Synchronization imposes a large cost on your program.**
  - Making a synchronized method call may take significantly longer, because of the need to acquire a lock (and flush caches and communicate with other processors).
  - Java leaves many of its mutable datatypes unsynchronized by default exactly for these performance reasons. **When you don't need synchronization, don't use it.**

# Sprinkling **synchronized** everywhere?

- Another argument for using **synchronized** in a more deliberate way is that it minimizes the scope of access to your lock.
- Adding **synchronized** to every method means that your lock is the object itself, and every client with a reference to your object automatically has a reference to your lock, that it can acquire and release at will.
- Your thread safety mechanism is therefore public and can be interfered with by clients.
- Contrast that with using a lock that is an object internal to your rep, and acquired appropriately and sparingly using **synchronized()** blocks.

# Sprinkling **synchronized** everywhere?

- **Finally, it's not actually sufficient to sprinkle synchronized everywhere.**
  - Dropping synchronized onto a method without thinking means that you're acquiring a lock without thinking about which lock it is, or about whether it's the right lock for guarding the shared data access you're about to do.
- **Suppose we had tried to solve findReplace's synchronization problem simply by dropping synchronized onto its declaration:**

```
public static synchronized boolean findReplace(EditBuffer buf, ...)
```

  - It would indeed acquire a lock — because findReplace is a static method, it would acquire a static lock for the whole class that findReplace happens to be in, rather than an instance object lock.
  - As a result, only one thread could call findReplace at a time — even if other threads want to operate on different buffers, which should be safe, they'd still be blocked until the single lock was free. So we'd **suffer a significant loss in performance**.



## (4) Liveness: deadlock, starvation and livelock



# Liveness

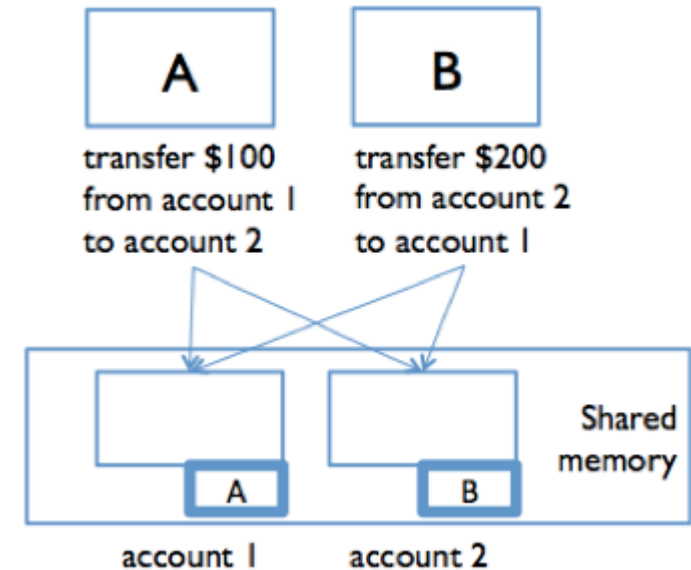
- 
- **A concurrent application's ability to execute in a timely manner is known as its liveness.**

# (1) Deadlock

- When used properly and carefully, locks can prevent race conditions. But then another problem rears its ugly head.
- Because the use of locks requires threads to wait ( acquire blocks when another thread is holding the lock), it's possible to get into a situation where two threads are waiting for each other – and hence neither can make progress.
- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.

# Deadlock

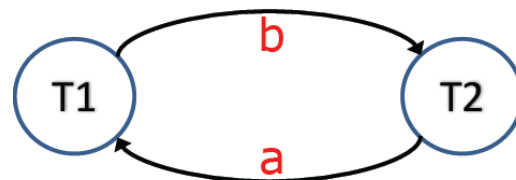
- **Deadlock** occurs when concurrent modules are stuck waiting for each other to do something.
- A deadlock may involve more than two modules: the signal feature of deadlock is a **cycle of dependencies**, e.g. A is waiting for B which is waiting for C which is waiting for A. None of them can make progress.




```

T1: synchronized(a){ synchronized(b){ ... } }
T2: synchronized(b){ synchronized(a){ ... } }

```



# Deadlock rears its ugly head

- 
- The locking approach to thread safety is powerful, but (unlike confinement and immutability) it introduces blocking into the program.
  - Threads must sometimes wait for other threads to get out of synchronized regions before they can proceed.
  - With locking, deadlock happens when threads acquire multiple locks at the same time, and two threads end up blocked while holding locks that they are each waiting for the other to release.
  - The monitor pattern unfortunately makes this fairly easy to do.



```

public class Wizard {
    private final String name;
    private final Set<Wizard> friends;
    // Rep invariant:
    //   name, friends != null
    //   friend links are bidirectional:
    //       for all f in friends, f.friends contains this
    // Concurrency argument:
    //   threadsafe by monitor pattern: all accesses to rep
    //   are guarded by this object's lock

    public Wizard(String name) {
        this.name = name;
        this.friends = new HashSet<Wizard>();
    }

    public synchronized boolean isFriendsWith(Wizard that) {
        return this.friends.contains(that);
    }

    public synchronized void friend(Wizard that) {
        if (friends.add(that)) {
            that.friend(this);
        }
    }

    public synchronized void defriend(Wizard that) {
        if (friends.remove(that)) {
            that.defriend(this);
        }
    }
}

```

```

Wizard harry = new Wizard("Harry Potter");
Wizard snape = new Wizard("Severus Snape");

```

// thread A		// thread B
harry.friend(snape);	?	snape.friend(harry);
harry.defriend(snape);		snape.defriend(harry);

It modifies the reps of both objects, because they use the monitor pattern means acquiring the locks to both objects.

# Deadlock solution 1: lock ordering

- One way to prevent deadlock is to **put an ordering on the locks that need to be acquired simultaneously, and ensuring that all code acquires the locks in that order.**
  - In the example, we might always acquire the locks on the Wizard objects in alphabetical order by the wizard's name.

```
public void friend(Wizard that) {
    Wizard first, second;
    if (this.name.compareTo(that.name) < 0) {
        first = this; second = that;
    } else {
        first = that; second = this;
    }
    synchronized (first) {
        synchronized (second) {
            if (friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

## Deadlock solution 2: coarse-grained locking

- **A more common approach than lock ordering is to use coarser locking – use a single lock to guard many object instances, or even a whole subsystem of a program.**
  - For example, we might have a single lock for an entire social network, and have all the operations on any of its constituent parts synchronize on that lock. In the code below, all Wizards belong to a Castle, and we just use that Castle object's lock to synchronize.
- **However, it has a significant performance penalty.**
  - If you guard a large pile of mutable data with a single lock, then you're giving up the ability to access any of that data concurrently.
  - In the worst case, having a single lock protecting everything, your program might be essentially sequential.

```
public class Wizard {  
    private final Castle castle;  
    private final String name;  
    private final Set<Wizard> friends;  
    ...  
    public void friend(Wizard that) {  
        synchronized (castle) {  
            if (this.friends.add(that)) {  
                that.friend(this);  
            }  
        }  
    }  
}
```

## (2) Starvation

- **Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.**
  - This happens when shared resources are made unavailable for long periods by "greedy" threads.
  - For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

## (3) Livelock

- A thread often acts in response to the action of another thread.
- If the other thread's action is also a response to the action of another thread, then *livelock* may result.
- As with deadlock, livelocked threads are unable to make further progress.
- However, the threads are not blocked — they are simply too busy responding to each other to resume work.
- This is comparable to two people attempting to pass each other in a corridor:
  - Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass.
  - Seeing that they are still blocking each other, Alphonse moves to his right, while Gaston moves to his left. They're still blocking each other, so...



(5) wait(), notify(), and  
notifyAll()



# Guarded Blocks

- **Guarded block:** such a block begins by polling a condition that must be true before the block can proceed.
- **Suppose, for example** `guardedJoy` is a method that must not proceed until a shared variable `joy` has been set by another thread.
  - Such a method could simply loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting.

```
public void guardedJoy() {  
    // Simple loop guard. Wastes  
    // processor time. Don't do this!  
    while(!joy) {}  
    System.out.println("Joy has been achieved!");  
}
```

# wait(), notify(), and notifyAll()

- **The following is defined for an arbitrary Java object o:**
  - `o.wait()`: release lock on o, enter o's wait queue and wait
  - `o.notify()`: wake up one thread in o's wait queue
  - `o.notifyAll()`: wake up all threads in o's wait queue



# Using `wait()` in Guarded Blocks

- The invocation of `wait()` does not return until another thread has issued a notification that some special event may have occurred – though not necessarily the event this thread is waiting for.
- The `Object.wait()` causes current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.

```
public synchronized void guardedJoy() {  
    // This guard only loops once for each special event,  
    // which may not be the event we're waiting for.  
    while(!joy) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    System.out.println("Joy and efficiency have been achieved!");  
}
```

# Using `wait()` in Guarded Blocks

- When `wait()` is invoked, the thread releases the lock and suspends execution.
- At some future time, another thread will acquire the same lock and invoke `Object.notifyAll()`, informing all threads waiting on that lock that something important has happened:

```
public synchronized notifyJoy() {  
    joy = true;  
    notifyAll();  
}
```

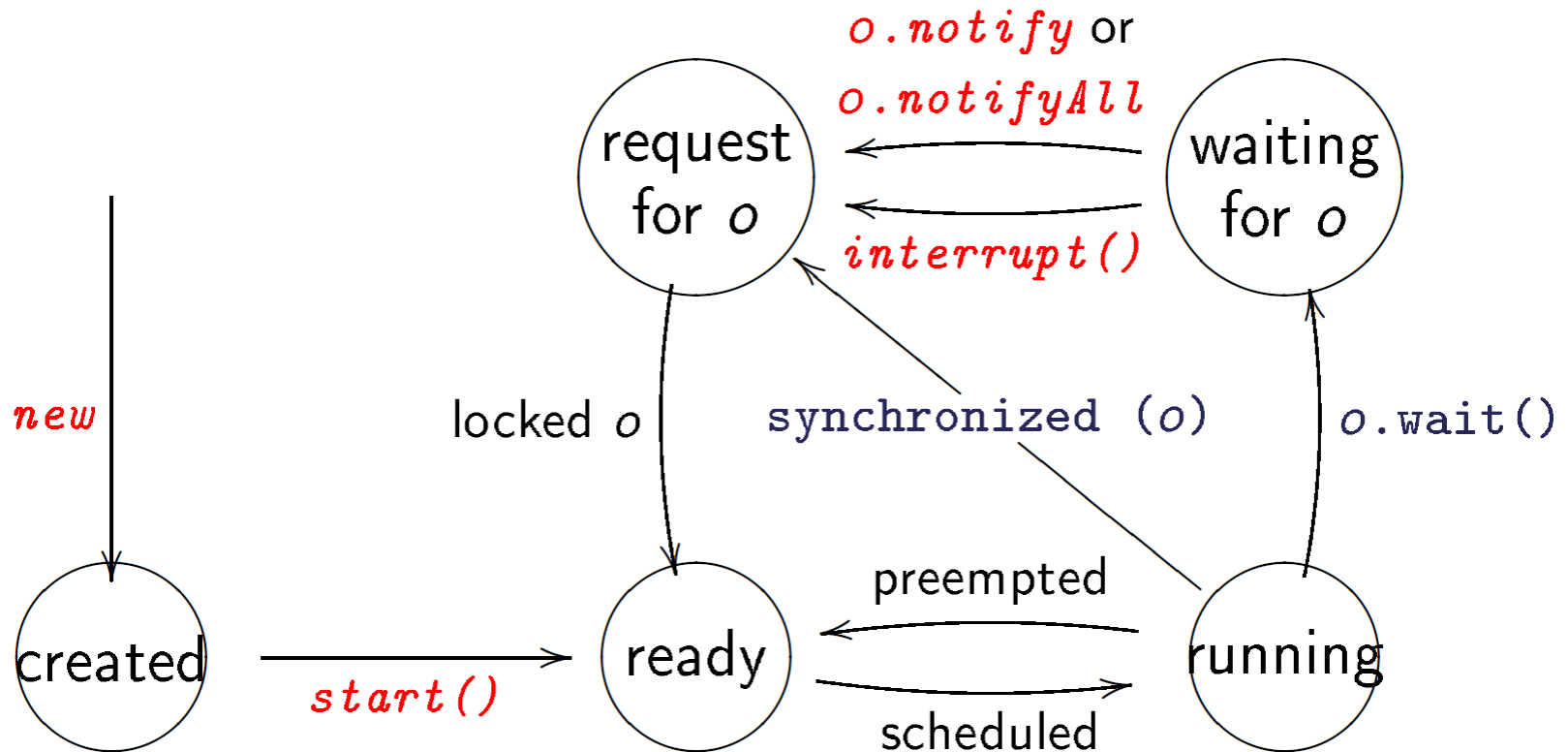
- Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of `wait`.
- A complete example of `wait()` and `notifyAll()` can be found in [http://www.tutorialspoint.com/java/lang/object\\_wait.htm](http://www.tutorialspoint.com/java/lang/object_wait.htm)

# wait(), notify(), and notifyAll()

- A thread that calls methods on object o must have locked o beforehand, typically:

```
synchronized (o) {  
    ...  
    o. wait () ;  
    ...  
}
```

# State model for threads: locking and waiting

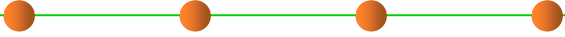




# 5 How to Make a Safety Argument



# Make a safety argument



```
if (cache.containsKey(x))  
    return cache.get(x);  
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);  
cache.put(x, answer);
```

# Make a safety argument

- **We have to argue that the races `containsKey()` , `get()` , and `put()` don't threaten this invariant.**
  - The race between `containsKey()` and `get()` is not harmful because we never remove items from the cache – once it contains a result for `x`, it will continue to do so.
  - There's a race between `containsKey()` and `put()`. As a result, it may end up that two threads will both test the primeness of the same `x` at the same time, and both will race to call `put()` with the answer. But both of them should call `put()` with the same answer, so it doesn't matter which one wins the race – the result will be the same.
- **The need to make these kinds of careful arguments about safety – even when you're using thread-safe data types – is the main reason that concurrency is hard.**

# Make a safety argument

- We've seen that concurrency is hard to test and debug.
- So if you want to convince yourself and others that your concurrent program is correct, the best approach is to make an explicit argument that it's free from races, and write it down.
- A safety argument needs to catalog all the threads that exist in your module or program, and the data that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: **confinement, immutability, threadsafe data types, or synchronization**.
- When you use the last two, you also need to argue that all accesses to the data are appropriately atomic – that is, that the invariants you depend on are not threatened by interleaving.



# Thread Safety Arguments for Confinement

- Confinement is not usually an option when we're making an argument just about a data type, because you have to know what threads exist in the system and what objects they've been given access to.
  - If the data type creates its own set of threads, then you can talk about confinement with respect to those threads.
  - Otherwise, the threads are coming in from the outside, carrying client calls, and the data type may have no guarantees about which threads have references to what.
- So confinement isn't a useful argument in that case.
- Usually we use confinement at a higher level, talking about the system as a whole and arguing why we don't need thread safety for some of our modules or data types, because they won't be shared across threads by design.

# Thread Safety Arguments for Immutability

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //   - a is final
    //   - a points to a mutable char array, but that array is encapsulated
    //     in this object, not shared with any other object or exposed to a
    //     client
}
```

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    private final int start;
    private final int len;
    // Rep invariant:
    //   0 <= start <= a.length
    //   0 <= len <= a.length-start
    // Abstraction function:
    //   represents the string of characters a[start],...,a[start+length-1]
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //   - a, start, and len are final
    //   - a points to a mutable char array, which may be shared with other
    //     MyString objects, but they never mutate it
    //   - the array is never exposed to a client
}
```

# Bad Safety Arguments

```
/** MyStringBuffer is a threadsafe mutable string of characters. */  
public class MyStringBuffer {  
    private String text;  
    // Rep invariant:  
    //   none  
    // Abstraction function:  
    //   represents the sequence text[0],...,text[text.length()-1]  
    // Thread safety argument:  
    //   text is an immutable (and hence threadsafe) String,  
    //   so this object is also threadsafe
```

- **Why doesn't this argument work?**
  - String is indeed immutable and threadsafe; but the rep pointing to that string, specifically the text variable, is not immutable.
  - text is not a final variable, and in fact it can't be final in this data type, because we need the data type to support insertion and deletion operations.
  - So reads and writes of the text variable itself are not threadsafe.
- **This argument is false.**

# Bad Safety Arguments

```
public class Graph {
    private final Set<Node> nodes =
        Collections.synchronizedSet(new HashSet<>());
    private final Map<Node, Set<Node>> edges =
        Collections.synchronizedMap(new HashMap<>());

    // Rep invariant:
    //   for all x, y such that y is a member of edges.get(x),
    //       x, y are both members of nodes
    // Abstraction function:
    //   represents a directed graph whose nodes are the set of nodes
    //       and whose edges are the set (x,y) such that
    //           y is a member of edges.get(x)
    // Thread safety argument:
    //   - nodes and edges are final, so those variables are immutable
    //     and threadsafe
    //   - nodes and edges point to threadsafe set and map data types
}
```

## ■ Is it a good safety argument?

- Graph relies on other threadsafe data types to help it implement its rep
- That prevents some race conditions, but not all, because the graph's rep invariant includes a relationship *between* the node set and the edge map. All nodes that appear in the edge map also have to appear in the node set.

# Bad Safety Arguments

```
public void addEdge(Node from, Node to) {  
    if ( ! edges.containsKey(from)) {  
        edges.put(from, Collections.synchronizedSet(new HashSet<>()));  
    }  
    edges.get(from).add(to);  
    nodes.add(from);  
    nodes.add(to);  
}
```

## ■ What if this code is executed?

- This code has a race condition in it. There is a crucial moment when the rep invariant is violated, right after the edges map is mutated, but just before the nodes set is mutated.
- Another operation on the graph might interleave at that moment, discover the rep invariant broken, and return wrong results.

# Bad Safety Arguments

```
public void addEdge(Node from, Node to) {  
    if ( ! edges.containsKey(from)) {  
        edges.put(from, Collections.synchronizedSet(new HashSet<>()));  
    }  
    edges.get(from).add(to);  
    nodes.add(from);  
    nodes.add(to);  
}
```

- Even though the threadsafe set and map data types guarantee that their own `add()` and `put()` methods are atomic and noninterfering, they can't extend that guarantee to interactions between the two data structures. So the rep invariant of Graph is not safe from race conditions.
- Just using immutable and threadsafe-mutable data types is not sufficient when the rep invariant depends on relationships between objects in the rep.

# A short summary

- **Three major ways to achieve safety from race conditions on shared mutable data:**
  - Confinement: not sharing the data.
  - Immutability: sharing, but keeping the data immutable.
  - Threadsafe data types: storing the shared mutable data in a single threadsafe datatype.
  
- **Safe from bugs.**
  - We're trying to eliminate a major class of concurrency bugs, race conditions, and eliminate them by design, not just by accident of timing.



# 6 Summary





# Goals of concurrent program design

- Is a concurrent program *safe from bugs*?
- We care about three properties:
  - **Safety.** Does the concurrent program satisfy its invariants and its specifications? Races in accessing mutable data threaten safety. Safety asks the question: can you prove that some bad thing never happens ?
  - **Liveness.** Does the program keep running and eventually do what you want, or does it get stuck somewhere waiting forever for events that will never happen? Can you prove that some good thing eventually happens ? Deadlocks threaten liveness.
  - **Fairness.** Concurrent modules are given processing capacity to make progress on their computations. Fairness is mostly a matter for an OS' thread scheduler, but you can influence it by setting thread priorities.

**Safety failure:**  
Incorrect  
computation

**Liveness failure:**  
No computation  
at all

# Concurrency in practice

- **What strategies are typically followed in real programs?**
  - **Library data structures either use no synchronization** (to offer high performance to single-threaded clients, while leaving it to multithreaded clients to add locking on top) or the **monitor pattern**.
  - **Mutable data structures with many parts typically use either coarse-grained locking or thread confinement.** Most graphical user interface toolkits follow one of these approaches, because a graphical user interface is basically a big mutable tree of mutable objects. Java Swing, the graphical user interface toolkit, uses thread confinement. Only a single dedicated thread is allowed to access Swing's tree. Other threads have to pass messages to that dedicated thread in order to access the tree.

Safety failures offer a false sense of security.  
Liveness failures force you to confront the bug.  
Temptation to favor liveness over safety.

# Concurrency in practice

- **What strategies are typically followed in real programs?**
  - **Search often uses immutable datatypes.** It would be easy to make multithreaded, because all the datatypes involved were immutable. There would be no risk of either races or deadlocks.
  - **Operating systems often use fine-grained locks** in order to get high performance, and use lock ordering to deal with deadlock problems.
  - **Databases avoid race conditions using *transactions***, which are similar to synchronized regions in that their effects are atomic, but they don't have to acquire locks, though a transaction may fail and be rolled back if it turns out that a race occurred. Databases can also manage locks, and handle locking order automatically.

to be introduced in Database Systems course.

# Summary

- Producing a concurrent program that is safe from bugs, easy to understand, and ready for change requires careful thinking.
  - Heisenbugs will skitter away as soon as you try to pin them down, so debugging simply isn't an effective way to achieve correct threadsafe code.
  - Threads can interleave their operations in so many different ways that you will never be able to test even a small fraction of all possible executions.
- Make thread safety arguments about your datatypes, and document them in the code.

# Summary

- Acquiring a lock allows a thread to have exclusive access to the data guarded by that lock, forcing other threads to block — as long as those threads are also trying to acquire that same lock.
- The *monitor pattern* guards the rep of a datatype with a single lock that is acquired by every method.
- Blocking caused by acquiring multiple locks creates the possibility of deadlock.



The end

May 8, 2019