# Advanced Object-Oriented Programming

CPT204 – Lecture 13
Erick Purwanto

Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT204  Advanced Object-Oriented Programming

Lecture 13

# Priority Queue, Thread Safety, Locks,  Synchronization

## Welcome !

- Welcome to Lecture 13 !
- In this lecture we are going to
  - learn about priority queues
  - learn about making variables threadsafe
  - learn to use a lock to protect shared mutable data
  - learn to recognize deadlock and know strategies to prevent it

## Part 1 : Priority Queue

- Priority queue is an abstract data structure that supports the operations :
    - add an item: `add(T item)`
    - delete smallest item: `T delMin()`
- For example:

```
add("A")
add("B")
add("C")
delMin()      →     "A"
add("D")
delMin()      →     "B"
```

## Application Example

- Problem : Find the *largest* M items in a stream `InputStream` of N items
  - Fraud detection : isolate transactions
  - File maintenance : find biggest files or directories

N huge, M large

- Constraint : Not enough memory to store N items

```
MinPQ<Transaction> pq = new MinPQ<Transaction>();
while (InputStream.hasNextLine()) {
    String line = InputStream.readLine();
    Transaction item = new Transaction(line);
    pq.add(item);
    if (pq.size() > M)
        pq.delMin();
}
```

## Applications of Priority Queues

- Applications of Priority Queues:
  - Customers in a line, Colliding particles simulations
  - Reducing roundoff error in numerical computation
  - Huffman codes
  - Dijkstra's algorithm,  Prim's algorithm
  - Sum of powers
  - A* search
  - Maintain largest M values in a sequence
  - Load balancing, Interrupt handling
  - Bin packing,  Scheduling
  - Bayesian spam filter

## MinPQ Interface

- Minimum Priority Queue supports fast removal of the smallest item in the data structure and fast item addition
- To have a notion of smallest, either we implement the item to be comparable, or explicitly specify the priority of an item with a comparable value, for example an integer
- Interface `MinPQ` with comparable items:

```
public interface MinPQ<T extends Comparable<T>> {
    void add(T item);
    T getMin();
    T delMin();
    boolean isEmpty();
    int size();
}
```

Lab 13

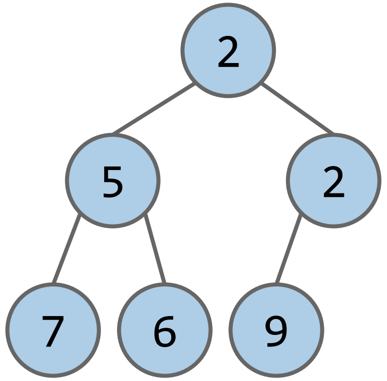
your course project Part A : Explicit Minimum Priority Queue `ExpMinPQ`

## Implementing MinPQ

- Implemented naively using an **array**, we can achieve constant-time item addition but linear-time smallest item removal
  - On the other hand, using an **ordered array**, constant-time removal but linear-time addition
- Our goal is then to have an implementation that can achieve fast running-time for *both* operations
  - Binary heap is suitable for this

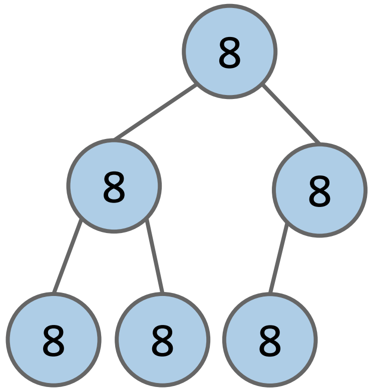
## Binary Heap

- **Binary heap** is a *binary tree* that is complete and satisfies the heap property :
    - **Complete**: balanced binary tree, could be left-aligned at the bottom level
    - **Heap property**: parent's item/priority is smaller or equal to the children's
- For example :

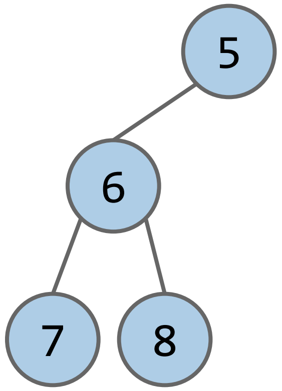


## In-Class Quiz 1

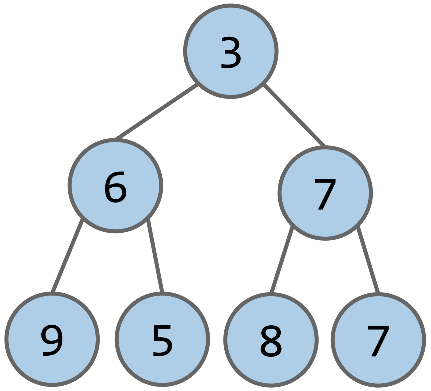
- Which of the following are binary heaps ?



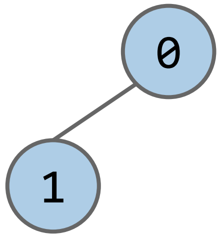
(i)



(ii)



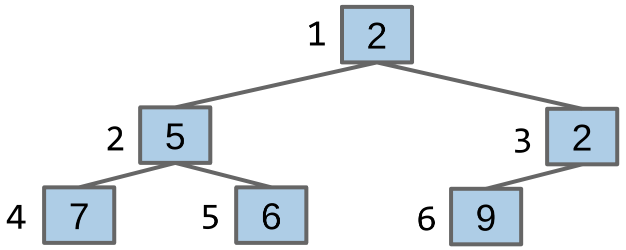
(iii)

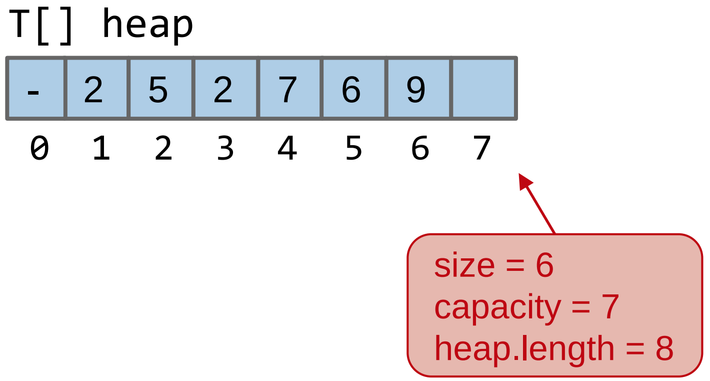


(iv)

## Binary Heap Representation

- If we represent the binary heap by an array or an `ArrayList`, set the indices to start at 1, and order the nodes containing the item and the priority by the heap-property :
  - there is no explicit links needed between parent and the children
  - we can use the indices to access an item's parent or children
  - parent of an item at index k is `k/2`
  - children of an item at index k is `2*k` and `2*k + 1`
- For example:





## Comparable Items and Swap

- Since the items are comparable, we can create a helper method to decide whether an item at index `i` is *greater than* the item at index `j`

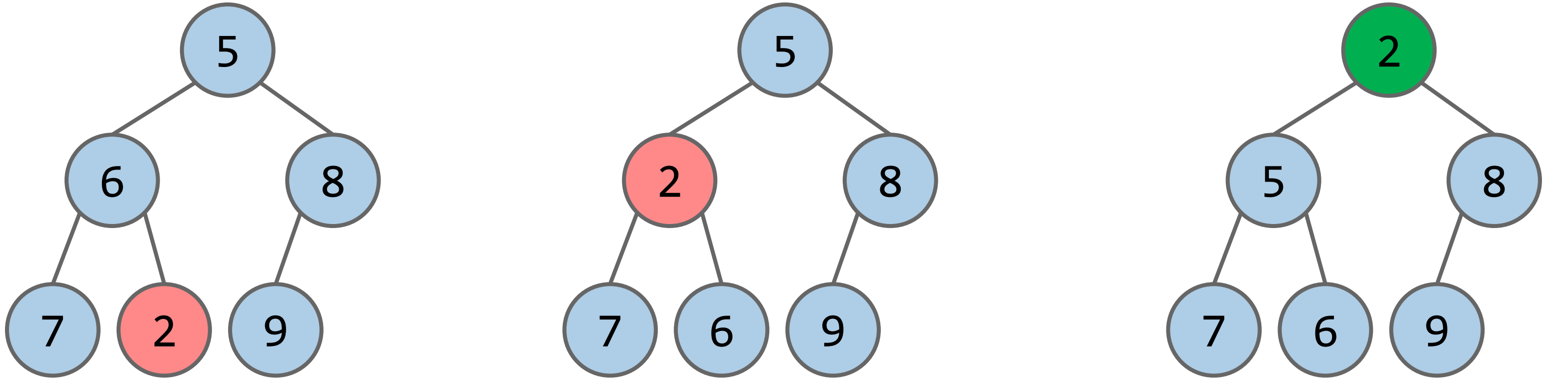
```
private boolean greater(int i, int j) {
    return ((Comparable<T>) heap[i]).compareTo(heap[j]) > 0;
}
```

- In the next slides, we will implement operations that swap (the reference to) the items

```
private void swap(int i, int j) {
    T temp = heap[i];
    heap[i] = heap[j];
    heap[j] = temp;
}
```

## Restoring Heap Property : Swim

- What should we do if a child becomes smaller than its parent ?

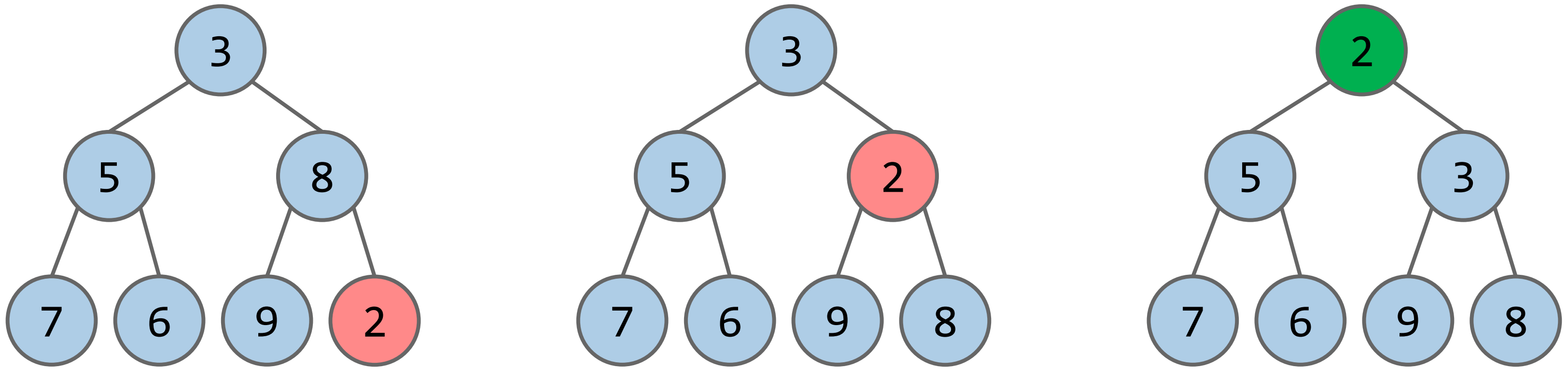


```
private void swim(int k) {
    while (k > 1 && greater(k/2, k)) {
        swap(k, k/2);
        k = k/2;
    }
}
```

swap child with the parent, until heap property is restored

## Item Addition

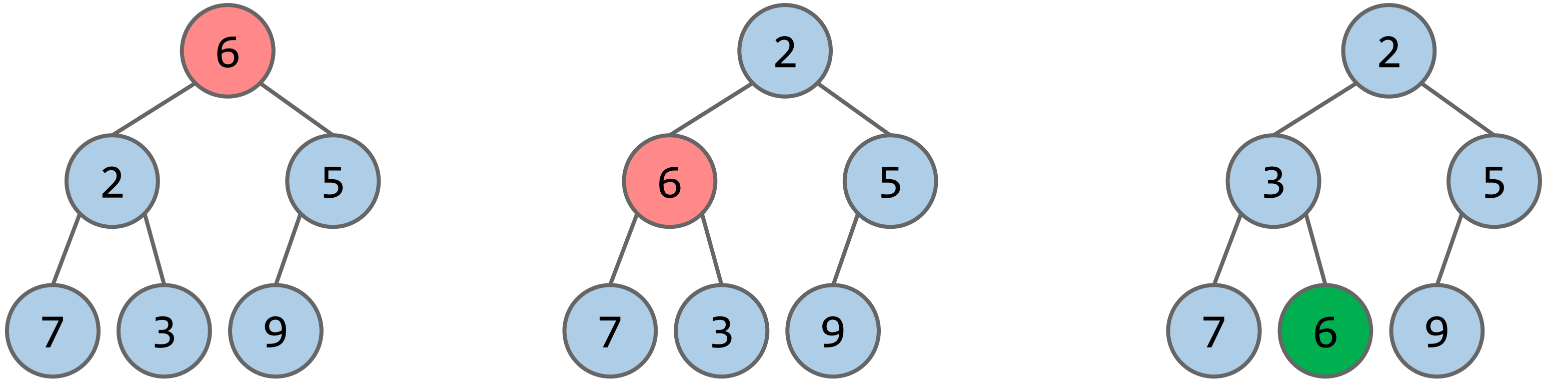
- Item addition:
  - add the new item at the end
  - then swim it : swap it with its parent repeatedly until the heap-property is restored



implement in Lab 13

## Restoring Heap Property : Sink

- What should we do if a parent becomes larger than one child / both of its children ?

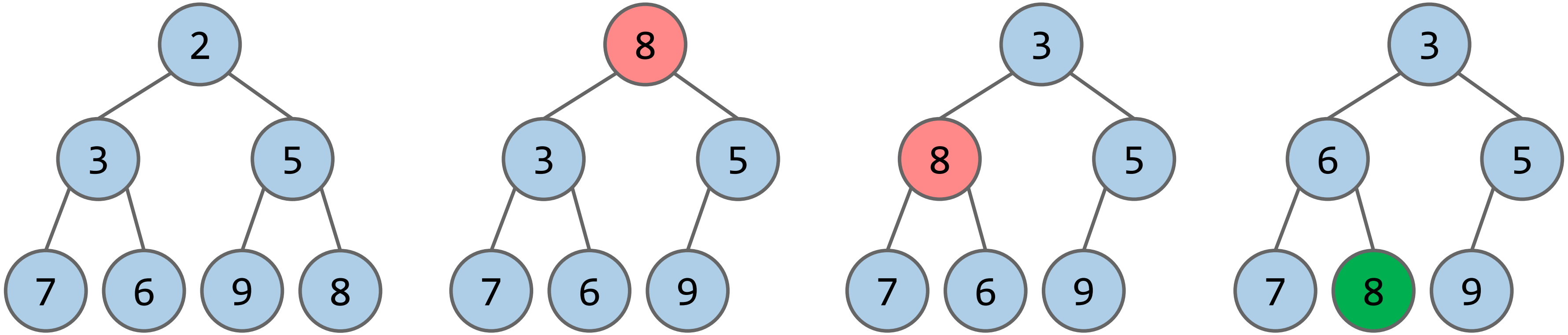


```
private void sink(int k) {
    while (2*k <= size) {
        int j = 2*k;
        if (j < size && greater(j, j+1)) j++;
        if (!greater(k, j)) break;
        swap(k, j);
        k = j;
    }
}
```

swap parent with the **smaller** child, until heap property is restored

## Minimum Item Removal

- Delete min:
  - replace the minimum item at the root with the last item
  - then sink it : swap it with the smaller children repeatedly until the heap-property is restored



implement in Lab 13

## Running Analysis of Binary Heap

How many comparisons?

- Item addition: O(height of a binary tree) = O(log N)
- Minimum item removal: O(2 x height of a binary tree) = O(log N)

| **Representation** | **add** | **delMin** |
|---|---|---|
| array | O(1) | O(N) |
| ordered array | O(N) | O(1) |
| binary heap | O(log N) | O(log N) |

## Part 2 : Thread Safety

- Last week we learn about race conditions
  - multiple threads share the same mutable variable without coordinating what they're doing
  - unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations

- This week we learn how to make variable access safe in shared-memory concurrency

## Thread Safety

- There are basically four ways to make variable access safe in shared-memory concurrency:
  - **Confinement** : don’t share variables or data between threads
  - **Immutability** : make the shared variables unreassignable or the shared data immutable
    - we’ve talked a lot about immutability already, but there are some additional requirements for concurrent programming that we’ll talk about in this lecture
  - **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 shared variables or data at the same time
    - synchronization is what you need to build your own threadsafe data type

## 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"

## Iterator Is Not Threadsafe

- For example : `Iterator`
  It is **not threadsafe** when used with a mutable collection
  - `Iterator`'s specification says that you can't modify a collection at the same time as you're iterating over it
    - except using the iterator’s own `remove` method
  - That's a timing-related precondition put on the caller, and `Iterator` makes no guarantee to behave correctly if you violate it

## Strategy 1: Confinement (1)

- Thread **confinement** is a simple idea: you avoid races on *reassignable references* and *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

## Strategy 1: Confinement (2)

- ***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 (in different threads or even at different levels of a single thread's stack, if the method is recursive),
  - but each of those invocations has its own *private copy* of the variable, so the variable itself is confined
- But be careful — the variable is thread confined, but if it's an object reference, you also *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

## Confinement Example

- We will see that confinement is what makes the accesses to `n`, `i`, and `result` safe in code like this:

```
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 = BigInteger.valueOf(1);
        for (int i = 1; i <= n; i++) {
            System.out.println("working on fact " + n);
            result = result.multiply(BigInteger.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);
    }
}
```

This code starts the thread for `computeFact(99)` with an anonymous `Runnable`

# In-Class Quiz 2

- How many threads are running when the program runs?

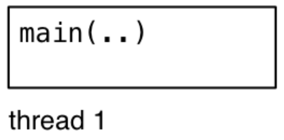
```java
public class Factorial {
    /* Computes n! and prints it on standard output.
     * @param n must be >= 0 */
    private static void computeFact(final int n) {
        BigInteger result = BigInteger.valueOf(1);
        for (int i = 1; i <= n; i++) {
            System.out.println("working on fact " + n);
            result = result.multiply(BigInteger.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);
    }
}
```

- ○ 1
- ○ 2
- ○ 100
- ○ 199

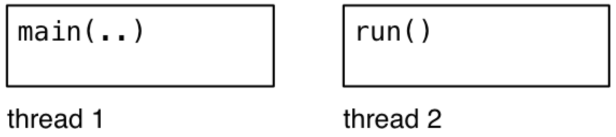
## Confinement Example (1)

- When we start the program, we start with one thread running `main`



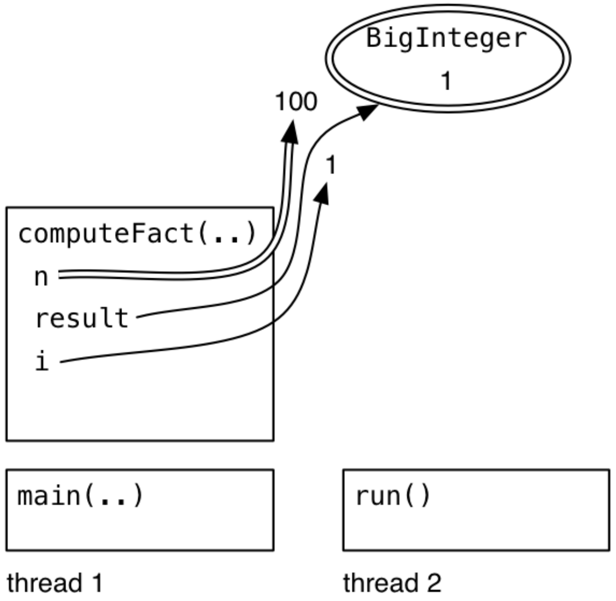
## Confinement Example (2)

- When we start the program, we start with one thread running `main`
- `main` creates a second thread using the anonymous `Runnable` idiom, and starts that thread



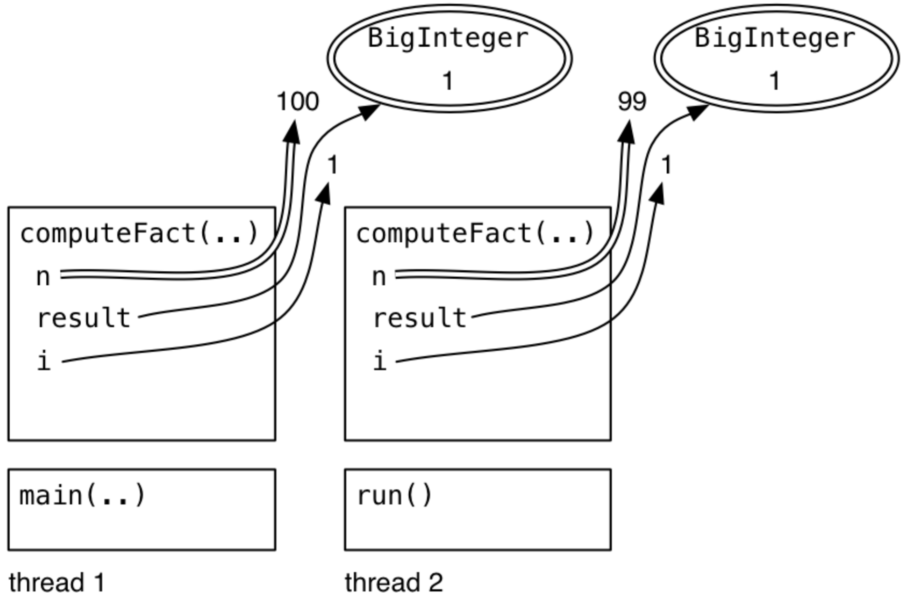
## Confinement Example (3)

- When we start the program, we start with one thread running `main`
- `main` creates a second thread using the anonymous `Runnable` idiom, and starts that thread
- At this point, we have two concurrent threads of execution
  - their interleaving is unknown!
  - but *one possibility* for the next thing that happens is that thread 1 enters `computeFact`



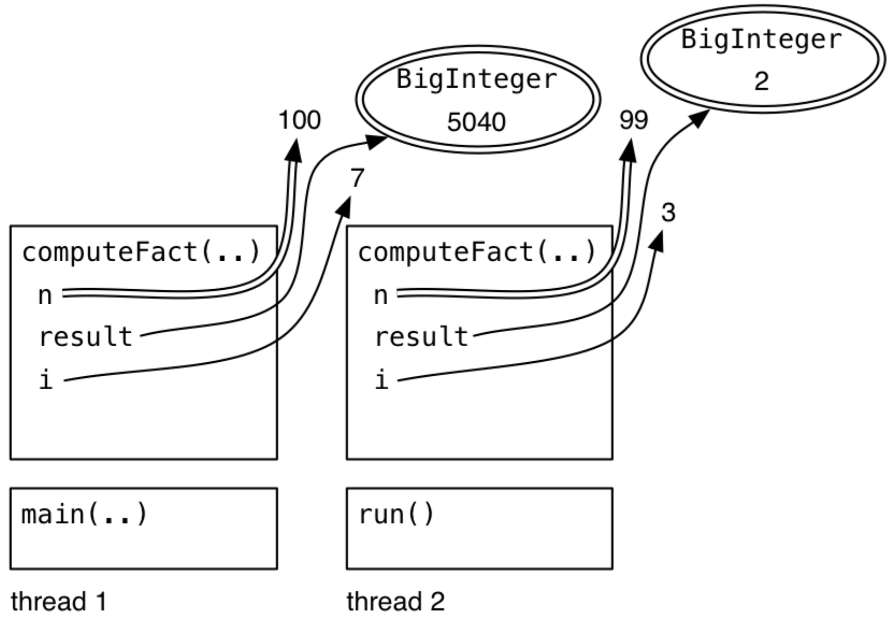
## Confinement Example (4)

- Then, the next thing that *might* happen is that thread 2 also enters `computeFact`
- At this point, we see how confinement helps with thread safety
  - Each execution of computeFact has ***its own*** `n`, `i`, and `result` variables, confined to that thread
  - The data they point to are also confined, and immutable
  - If the `BigInteger` objects were not confined – if they were aliased from multiple threads – then we would need to rely on their immutability to guarantee thread safety (next strategy, strategy #2)



## Confinement Example (5)

- The `computeFact` computations proceed ***independently***, updating their respective variables



## Avoid Global Variables (1)

- Unlike local variables, ***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; even better, you should eliminate the static variables entirely, for example:

```java
// 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;
     }
}
```

## Avoid Global Variables (2)

- 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 we don't want

- To fix this race using the thread confinement approach
  - you would specify that only a certain thread (maybe the “pinball simulation thread”) is allowed to call `PinballSimulator.getInstance()`

## Avoid Global Variables (3)

- In general, static variables are very risky for concurrency
  - They might be hiding behind a harmless function that seems to have no side-effects or mutations
  - Consider this example:

```
// is this method threadsafe?
/**
 * @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 (4)

- This function stores the answers from previous calls in case they're requested again
    - this technique is called **memoization**, and it's a sensible optimization for slow functions like exact primality testing
- But now the `isPrime` method is not safe to call from multiple threads, and its clients may not even realize it
- The reason is that the `HashMap` referenced by the static variable `cache` is shared by all calls to `isPrime()`, and `HashMap` is ***not*** threadsafe
- If multiple threads mutate the map at the same time, by calling `cache.put()`, then the map can become corrupted in the same way that the bank account became corrupted in the last lecture
- If you're lucky, the corruption may cause an exception deep in the hash map, like a `NullPointerException` or `IndexOutOfBoundsException`
    - but it also may just quietly give wrong answers, as we saw in the bank account example

## Strategy 2: Immutability (1)

- Our second way of achieving thread safety is by using **unreassignable references** and **immutable data types**
  - immutability tackles the shared-mutable-data cause of a race condition and solves it simply by making the shared state **not mutable**
- Final variables are unreassignable/immutable references, so a variable declared final is safe to access from multiple threads
  - you can only read the variable, not write it
  - be careful, 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 (2)

- Immutable objects are usually also threadsafe
- We say "usually" here because our current definition of immutability is too loose for concurrent programming
    - We've said that 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
    - We saw an example of this notion when we looked at an immutable list that cached its length in a mutable field the first time the length was requested by a client
- For concurrency, though, this kind of hidden mutation is **not safe**
    - An immutable data type that uses some mutation will have to make itself threadsafe using locks (the same technique required of mutable data types we will discuss later)

## Threadsafe Immutability

- So in order to be confident that an immutable data type is threadsafe without locks, we need a ***stronger*** definition of immutability
- A type is **threadsafe immutable** if it has:
  - no mutator methods
  - all fields declared `private` and `final`
  - no representation exposure
  - no mutation whatsoever of mutable objects in the rep
- If you follow these rules, then you can be confident that your immutable type will also be threadsafe
- In Java Tutorials, read A Strategy for Defining Immutable Objects

## Strategy 3: Using Threadsafe Data Types (1)

- Our 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
- For example, here's what `StringBuffer` says:
    - *[StringBuffer is] A threadsafe, mutable sequence of characters. A string buffer is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.*
- `StringBuffers` are **safe** for use by multiple threads
    - the methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved

## Strategy 3: Using Threadsafe Data Types (2)

- This is in contrast to `StringBuilder`:
  - *[StringBuilder is] A mutable sequence of characters. This class provides an API compatible with `StringBuffer`, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for `StringBuffer` in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.*

## Strategy 3: Using Threadsafe Data Types (3)

- 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

- It's deeply unfortunate that `StringBuffer` and `StringBuilder` are named so similarly, without any indication in the name that thread safety is the crucial difference between them
    - It's also unfortunate that they *don't share a common interface*, so you can't simply *swap* in one implementation for the other for the times when you need thread safety
    - The Java collection interfaces do much better in this respect, as we'll see next

## Threadsafe Collections (1)

- The collection interfaces in Java — `List`, `Set`, `Map` — have basic implementations that are **not** threadsafe
  - The implementations of these that you've been used to using, namely `ArrayList`, `HashMap`, and `HashSet`, cannot be used safely from more than one thread
- Fortunately, 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*

## Threadsafe Collections (2)

- Now we see a way to fix the `isPrime()` method we had earlier in the slides:

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

- **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
  - that happens automatically in the line of code above, since the `new HashMap` is passed only to `synchronizedMap()` and never stored anywhere else
  - we saw this same warning with the unmodifiable wrappers: the underlying collection is still mutable, and code with a reference to it can circumvent immutability

## Threadsafe Collections (3)

- **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 enhanced for loop syntax:

```
for (String s: list) { ... }  // not threadsafe
```

    even if `list` 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, which we'll talk about in the next strategy

## Threadsafe Collections (4)

- Finally, **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 ( ! list.isEmpty() ) {
    String s = list.get(0);
    ...
}
```

- Even if you make `list` 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 Collections (5)

- Even the `isPrime()` method still has potential races:

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

- The synchronized map ensures that `containsKey()`, `get()`, and `put()` are now atomic, so using them from multiple threads won't damage the rep invariant of the map
    - But those three operations can now *interleave* in arbitrary ways with each other, which might break the invariant that `isPrime` needs from the cache: if the cache maps an integer `x` to a value `b`, then `x` is prime if and only if `b` is `true`
    - If the cache ever fails this invariant, then we might return the wrong result

## Threadsafe Collections (6)

- So we have to argue that the races between `containsKey()`, `get()`, and `put()` don't threaten this invariant :
  1. Suppose `containsKey(x)` returns `true`, but then another thread mutates the cache before the `get(x)` call; this is not harmful because we never remove items from the cache – once it contains a result for `x`, it will continue to do so
  2. Alternatively, suppose `containsKey(x)` returns `false`, but another thread mutates the cache before `put(x, …)`; It may end up that the two threads 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(x, answer)` with the same value for 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 threadsafe data types — is the main reason that concurrency is hard

## Strategy 4: Synchronization

- **Synchronization**: prevent threads from accessing the shared data at the same time
  - this is what we use to implement a threadsafe type
  - we want to learn to use synchronization to implement our own data type that is safe for shared-memory concurrency
    - the correctness of a concurrent program should not depend on accidents of timing

## Locks (1)

- Since race conditions caused by concurrent manipulation of shared mutable data are disastrous bugs — hard to discover, hard to reproduce, hard to 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 working with this thing, ***don't*** touch it right now."

## Locks (2)

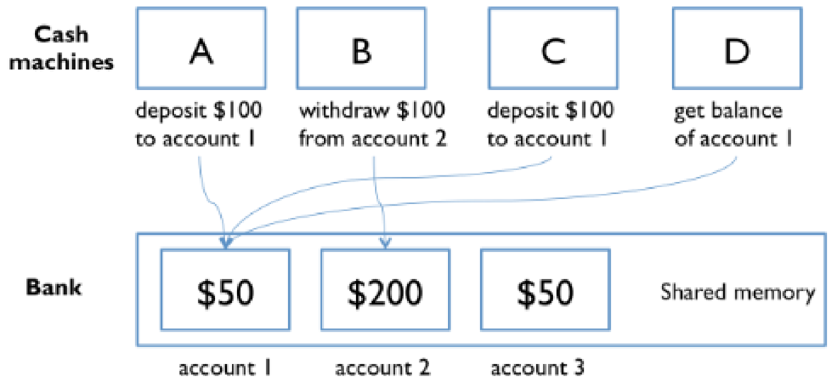
- Locks have two operations:
  - **`acquire`** allows a thread to take ownership of a lock
    - if a thread tries to acquire a lock currently owned by another thread, it **blocks** until the other thread releases the lock
    - at that point, it will *compete* with any other threads that are trying 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
- Using a lock also tells the compiler and processor that you're using shared memory concurrently, so that *registers* and *caches* will be flushed out to shared storage
  - this avoids the problem of ***reordering***, ensuring that the owner of a lock is always looking at up-to-date data

## Blocking

- **Blocking** means that a thread waits (without doing further work) until an event occurs
- An `acquire(ℓ)` on thread 1 will block if another thread, say thread 2, is holding lock $\ell$
    - the event it waits for is thread 2 performing `release(ℓ)`
    - at that point, if thread 1 can acquire $\ell$, it continues running its code, with ownership of the lock
    - it is possible that another thread, say thread 3, was also blocked on `acquire(ℓ)`
    - if so, either thread 1 or 3 (the winner is nondeterministic) will take the lock $\ell$ and continue
    - the other will continue to block, waiting for `release(ℓ)` again

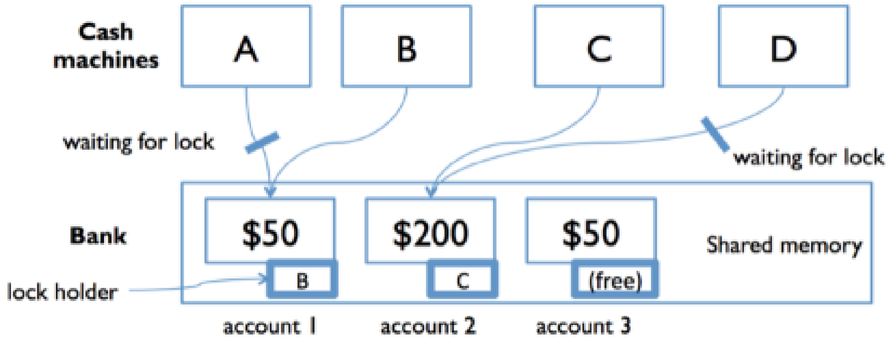
## Bank Account Example (1)

- Recall our first example of shared memory concurrency was *a bank with cash machines*
- The bank has several cash machines, all of which can read and write the same account objects in memory
- Of course, without any coordination between concurrent reads and writes to the account balances, things went horribly wrong



## Bank Account Example (2)

- 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



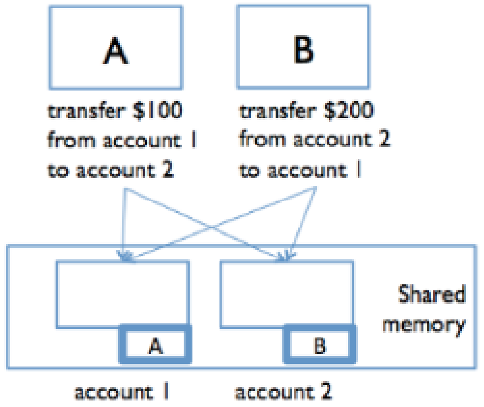
- In the figure, both A and B are trying to access account 1
  - suppose B acquires the lock first
  - then A must wait to read and write the balance until B finishes and releases the lock
  - this ensures that A and B are synchronized, but another cash machine C is able to run *independently* on a different account (because that account is protected by a different lock)

## Deadlock (1)

- 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 (2)

- In the figure, suppose A and B are making simultaneous transfers between two accounts in our bank
- A transfer between accounts ***needs to lock both accounts***, so that money can't disappear from the system
- A and B each acquire the lock on their respective "from" account: A acquires the lock on account 1, and B acquires the lock on account 2
- Now, each must acquire the lock on their "to" account: so A is waiting for B to release the account 2 lock, and B is waiting for A to release the account 1 lock
- Stalemate! A and B are frozen in a "deadly embrace", and accounts are locked up!


A
B
transfer $100 from account 1 to account 2
transfer $200 from account 2 to account 1
A
B
Shared memory
account 1
account 2

## Deadlock (3)

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

## Deadlock (4)

- You can also have deadlock without using any locks
- For example, a message-passing system, say a client and a server, can experience deadlock when message buffers fill up
    - if the client fills up the server's buffer with requests, and then blocks waiting to add another request,
    - the server may then fill up the client's buffer with results and then block itself
    - so the client is waiting for the server, and the server waiting for the client, and neither can make progress until the other one does
    - again, deadlock ensues

- In the Java Tutorials, read Deadlock

## Developing a Threadsafe Abstract Data Type

- Let's see how to use synchronization to implement a threadsafe ADT
- Suppose we're building a multi-user editor, like Google Docs, that allows multiple people to connect to it and edit it at the same time
  - We'll need a mutable data type 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);
```

## EditBuffer Interface

```
    /**
     * 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();
}
```

## String Rep

- A very simple rep for this data type would just be ***a string***:

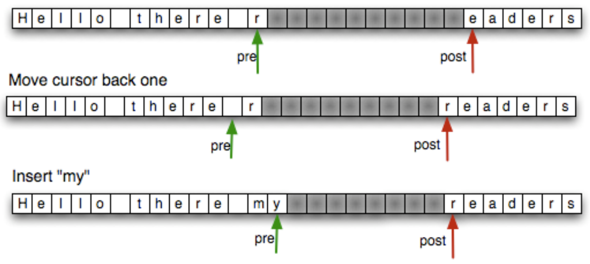
```
public class SimpleBuffer implements EditBuffer {
    private String text;
    // Rep invariant:
    //   text != null
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
```

## Character Array Rep

- The downside of this rep is that every time we do an insert or delete,  we have to copy the entire string into a new string
    - that gets expensive

- Another rep we could use would be ***a character array***,  with space at the end
    - that's fine if the user is just typing new text at the *end* of the document (we don't have to copy anything),
    - but if the user is typing at the *beginning* of the document,  then we're copying the entire document with every keystroke

## Gap Buffer Rep (1)


H e l l o t h e r e r e a d e r s
pre
post
Move cursor back one
H e l l o t h e r e r r e a d e r s
pre
post
Insert "my"
H e l l o t h e r e m y r e a d e r s
pre
post

- A more interesting rep, which is used by many text editors in practice, is called **a gap buffer**
  - It is a character array with extra space in it, but instead of having all the extra space at the end, the extra space is a gap that can appear anywhere in the buffer
  - Whenever an insert or delete operation needs to be done, 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 Rep  (2)

```java
/** GapBuffer is a non-threadsafe EditBuffer that is optimized
 *  for editing with a cursor, which tends to make a sequence of
 *  inserts and deletes at the same place in the buffer. */
public class GapBuffer implements EditBuffer {
    private char[] a;
    private int gapStart;
    private int gapLength;
    // Rep invariant:
    //   0 <= gapStart <= a.length
    //   0 <= gapLength <= a.length - gapStart
    // Abstraction function:
    //   represents the sequence a[0],...,a[gapStart-1],
    //                           a[gapStart+gapLength],...,a[a.length-1]
```

- In a multi user scenario,  we wouldd want multiple gaps
    - one for each user's cursor,  but we'll use a single gap for now

## Steps to Develop The Data Type (1)

Recall our recipe for designing and implementing an ADT:

1. **Specify**: define the operations (method signatures and specs)
   - We did that in the `EditBuffer` interface
2. **Test**: develop test cases for the operations
   - Build a test suite with a testing strategy based on partitioning the parameter space of the operations
3. **Rep**: choose a rep
   - We chose two of them for `EditBuffer`, and this is often a good idea :

## Steps to Develop The Data Type (2)

a) **Implement a simple, brute-force rep first**

- It's easier to write, you're more likely to get it right, and it will *validate your test cases* and *your specification* so you can fix problems in them before you move on to the harder implementation
- This is why we implemented `SimpleBuffer` before moving on to `GapBuffer`
- Don't throw away your simple version, either — keep it around so that you have something to test and compare against in case things go wrong with the more complex one

## Steps to Develop The Data Type (3)

**b) Write down the rep invariant and abstraction function, and implement `checkRep()`**

- `checkRep()` asserts the rep invariant at the end of every constructor, producer, and mutator method
- It's typically *not necessary* to call it at the end of *an observer*, since the rep hasn't changed
- In fact, assertions can be very useful for testing complex implementations, so it's not a bad idea to also assert the postcondition at the end of a complex method

## Steps to Develop The Data Type (4)

- In all these steps, we're working entirely **single-threaded** at first
  - *Multithreaded* clients should be in the back of our minds at all times while we're writing specs and choosing reps
    - we'll see later that careful choice of operations may be necessary to avoid race conditions in the clients of your datatype
  - But get it working, and thoroughly tested, in a *sequential*, *single-threaded* environment first

## Steps to Develop The Data Type (5)

Now we're ready for the next step:

4. **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
    - this part of the lecture is about how to do step 4

## Steps to Develop The Data Type (6)

And then the extra step we hinted at above:

5. **Iterate**
   - You may find that your choice of operations makes it hard to write a threadsafe type with the guarantees clients require
   - You might discover this in step 1, or in step 2 when you write tests, or in steps 3 or 4 when you implement
   - If that's the case, go back and refine the set of operations your ADT provides

## Locking (1)

- **Locks** are so commonly-used that Java provides them as *a built-in language feature*

- In Java, *every* object has a lock implicitly associated with it — a `String`, an array, an `ArrayList`, and every class you create, all of their object instances have a lock
    - Even a humble `Object` has a lock, so bare `Objects` are often used for explicit locking:

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

## Locking (2)

- 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
}
```

- 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

## Locks Guard Access to Data (1)

- Locks are used to **guard** a shared data variable, like the `balance` shown in the previous slide
  - if all accesses to a data variable are guarded (surrounded by a synchronized block) by the same lock object, then those accesses will be guaranteed to be *atomic* — uninterrupted by other threads
- Locks only provide mutual exclusion with other threads that acquire the ***same*** lock
  - all accesses to a data variable must be guarded by the same lock
  - you might guard an entire collection of variables behind a single lock, but all modules must agree on which lock they will all acquire and release

- Because every object in Java has a lock implicitly associated with it, you might think that simply owning an object's lock would prevent other threads from accessing that object
  - that is **not** the case

## Locks Guard Access to Data (2)

- When a thread `t` acquires an object’s lock using `synchronized (obj) { ... }`

  it does one thing and one thing only:
  prevents other threads from entering **their own `synchronized(exp)` block**,
  where `exp` refers to the same object as `obj`

  - until thread `t` finishes its synchronized block
  - that's it

- Even while `t` is in its synchronized block, another thread can dangerously mutate the object,
  - simply by neglecting to use `synchronized` itself

- In order to use an object lock for synchronization,
  - you have to ***explicitly*** and carefully guard every such access with an appropriate `synchronized` block

## Monitor Pattern (1)

- 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)`

```
/** 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();
        }
    }
```

## Monitor Pattern (2)

```
    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 (3)

- Note the very careful discipline here
  - *every* method is guarded with the lock — even apparently small and trivial ones like `length()` and `toString()`
  - this is because reads must be guarded as well as writes — if reads are left unguarded, then they may be able to see the rep in a partially-modified state
- This approach is called **the 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
- Java provides some syntactic sugar for the 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

## Monitor Pattern (4)

- So the code below is an equivalent way to implement the synchronized `SimpleBuffer`:

```java
/** 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();
    }
```

## Monitor Pattern (5)

```
    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;
    }
}
```

- Notice that the `SimpleBuffer` constructor doesn't have a `synchronized` keyword
- Java actually forbids it, syntactically, because an object under construction is expected to be confined to a single thread until it has returned from its constructor
  - so synchronizing constructors should be *unnecessary*

## Monitor Pattern  (6)

In the Java Tutorials,  read:

- Synchronized Methods
- Intrinsic Locks and Synchronization

## Thread Safety Argument with Synchronization (1)

- Now that we're protecting SimpleBuffer's rep with a lock, we can write a better thread safety argument:

```
/** 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
```

## Thread Safety Argument with Synchronization (2)

- The same argument works for `GapBuffer`, if we use the monitor pattern to synchronize all its methods
- Note that the encapsulation of the class, the absence of rep exposure, is very important for making this argument
  - If text were public:

    ```
    public String text;
    ```

    then clients outside `SimpleBuffer` would be able to read and write it without knowing that they should first acquire the lock,

    and `SimpleBuffer` would *no longer be threadsafe*

## Locking Discipline

- A locking discipline is a strategy for ensuring that synchronized code is threadsafe
- We must satisfy two conditions:
    1. 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
    2. 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

## Atomic Operations (1)

- 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;
}
```

notice that `findReplace` is a static method

## Atomic Operations (2)

- This method makes three different calls to `buf` — to convert it to a string in order to search for `s`, to delete the old text, and then to insert `t` in its place
  - 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 (1)

- 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:

```java
/** 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 {
   ...
}
```

## Giving clients access to a lock (2)

pass the corresponding `EditBuffer` object/instance

- And then `findReplace` can synchronize on `buf`:

```
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;
    }
}
```

- 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

## Sprinkling synchronized everywhere? (1)

- 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 haphazardly
  - 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 data types unsynchronized by default exactly for these performance reasons
  - *when you don't need synchronization, don't use it*

## Sprinkling synchronized everywhere?  (2)

- Second, it's not actually sufficient to add `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, … ) {
```

## Sprinkling synchronized everywhere? (3)

- This **wouldn't** do what we want
  - 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, because only one user of our massive multiuser editor would be allowed to do a find-and-replace at a time,
    even if they're all editing different documents

## Sprinkling synchronized everywhere?  (4)

- Worse, however, it wouldn't provide useful protection, because other code that touches the document probably **wouldn't be acquiring the same lock**
  - it **wouldn't** actually eliminate our race conditions

- The `synchronized` keyword is *not a panacea*
  - thread safety requires a discipline — using confinement, immutability, or locks to protect shared data
  - and that discipline needs to be written down,  or maintainers won't know what it is

## Designing a Data Type for Concurrency (1)

- `findReplace`'s problem can be interpreted another way: that the `EditBuffer` interface really isn't that friendly to multiple simultaneous clients
    - it relies on integer indexes to specify insert and delete locations, which are extremely brittle to other mutations
    - if somebody else inserts or deletes before the index position, then the index becomes invalid

## Designing a Data Type for Concurrency (2)

- So if we're designing a data type specifically for use in a concurrent system, we need to think about providing operations that have better-defined semantics when they are interleaved
- For example, it might be better to pair `EditBuffer` with a `Position` data type representing a cursor position in the buffer, or even a `Selection` data type representing a selected range
- Once obtained, a `Position` could hold its location in the text against the wash of insertions and deletions around it, until the client was ready to use that `Position`
- If some other thread deleted all the text around the `Position`, then the `Position` would be able to inform a subsequent client about what had happened (perhaps with an exception), and allow the client to decide what to do
- These kinds of considerations come into play when designing a data type for concurrency

## Designing a Data Type for Concurrency (3)

- As another example, consider the `ConcurrentMap` interface in Java
  - this interface extends the existing `Map` interface, adding a few key methods that are commonly needed as atomic operations on a shared mutable map, e.g.:
  - **`map.putIfAbsent(key,value)`** is an atomic version of

    ```
    if ( ! map.containsKey(key)) map.put(key, value);
    ```
  - **`map.replace(key, value)`** is an atomic version of

    ```
    if (map.containsKey(key)) map.put(key, value);
    ```

## Deadlock rears its ugly head (1)

- The synchronization 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
    - and blocking raises the possibility of deadlock

- As we saw in the previous few slides before, 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*

# Deadlock rears its ugly head (2)

- Here's an example: suppose we're modeling the social network in Hogwarts:

```
public class Wizard {
    private final String name;
    private final Set<Wizard> friends;
    // Rep invariant:
    //    friend links are bidirectional:
    //    for all f in friends, f.friends contains this
    // Thread safety 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>();
    }
```

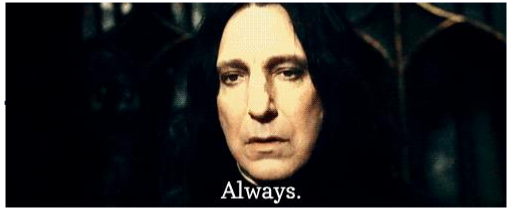
## Deadlock rears its ugly head (3)

```
    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);
        }
    }
}
```

## Deadlock rears its ugly head  (4)



- Like Facebook,  this social network is *bidirectional*:
  if x is friends with y,  then y is friends with x
  - the `friend()` and `defriend()` methods enforce that invariant by modifying the reps of both objects,
  - which because they use the monitor pattern means acquiring the locks to both objects as well

- Let's create a couple of wizards:

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

## In-Class Quiz 3

- And then think about what happens when two independent threads are repeatedly running:

```
 // thread A                     // thread B
while(true) {                   while(true) {
    harry.friend(snape);             snape.friend(harry);
    harry.defriend(snape);           snape.defriend(harry);
}                               }
```

- Deadlock will happen when:
    - Thread A acquires lock on `harry`, Thread B acquires lock on `snape`
    - Thread A acquires lock on `snape`, Thread B acquires lock on `harry`
    - Thread A acquires lock on `harry`, Thread B acquires lock on `harry`
    - Thread A acquires lock on `snape`, Thread B acquires lock on `snape`

## Deadlock rears its ugly head (5)

- And then think about what happens when two independent threads are repeatedly running:

```
 // thread A                          // thread B
while(true) {                        while(true) {
    harry.friend(snape);                 snape.friend(harry);
    harry.defriend(snape);               snape.defriend(harry);
}                                    }
```

- We will deadlock very rapidly :
  - Suppose thread A is about to execute `harry.friend(snape)`, and thread B is about to execute `snape.friend(harry)`
  - Thread A acquires the lock on `harry` (because the `friend` method is synchronized)
  - Then thread B acquires the lock on `snape` (for the same reason)
  - They both update their individual reps independently, and then try to call `friend()` on the other object — which requires them to acquire the lock on the other object

# Deadlock rears its ugly head (6)

- So A is holding Harry and waiting for Snape, and B is holding Snape and waiting for Harry
- Both threads are stuck in `friend()`, so neither one will ever manage to exit the synchronized region and release the lock to the other
- This is a classic deadly embrace: the program was *stupefied*

- The essence of the problem is acquiring multiple locks, and holding some of the locks while waiting for another lock to become free
- Notice that it is *possible* for thread A and thread B to interleave such that deadlock does not occur: perhaps thread A acquires and releases both locks before thread B has enough time to acquire the first one
  - If the locks involved in a deadlock are also involved in a race condition — and very often they are — then the deadlock will be just *as difficult to reproduce or debug*

## Deadlock Solution 1:  Lock Ordering  (1)

- 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 our Hogwarts social network example, we might always acquire the locks on the `Wizard` objects in *alphabetical* order by the wizard's name
    - since thread A and thread B are both going to need the locks for Harry and Snape, they would both acquire them in that order:  Harry's lock first,  then Snape's
    - if thread A gets Harry's lock before B does,  it will also get Snape's lock before B does,  because B can't proceed until A releases Harry's lock again
    - the ordering on the locks forces an ordering on the threads acquiring them,  so there's no way to produce a cycle in the waiting-for graph

## Deadlock Solution 1: Lock Ordering (2)

- Here's what the code might look like:

```
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 1:  Lock Ordering  (3)

- Note that the decision to order the locks alphabetically by the person's name would work fine for this wizarding world, but it wouldn't work in a real life social network
    - Why not?  What would be better to use for lock ordering than the name?
- Although lock ordering is useful (particularly in code like operating system kernels), it has a number of drawbacks in practice
- First, it's **not modular** — the code has to know about all the locks in the system, or at least in its subsystem
- Second, it may be difficult or impossible for the code **to know exactly which of those locks it will need** before it even acquires the first one
    - it may need to do some computation to figure it out
    - think about doing a *depth-first search* on the social network graph,  for example — how would you know which nodes need to be locked,  before you've even started looking for them?

## Deadlock Solution 2: Coarse-grained Locking (1)

- A more common approach than lock ordering, particularly for application programming (as opposed to operating system or device driver programming), 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*

## Deadlock Solution 2: Coarse-grained Locking (2)

- In the code below, all `Wizards` belong to a `Castle`, and we just use that `Castle` object's lock to synchronize:

```
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);
            }
        }
    }
}
```

## Deadlock Solution 2: Coarse-grained Locking (3)

- Coarse-grained locks can have 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* — only one thread is allowed to make progress at a time

## Goals of Concurrent Program Design (1)

- Building concurrent software is challenging
- We can break the issues into two general classes: when we ask whether a concurrent program is safe from bugs, we care about two 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?***

## Goals of Concurrent Program Design (2)

- Deadlocks threaten liveness
- Liveness may also require **fairness**, which means that concurrent modules are given processing capacity to make progress on their computations
- Fairness is mostly a matter for the operating system's thread scheduler,
but you can influence it (for good or for ill) by setting thread priorities

## Thank you for your attention !

- In this lecture,  you have learned:
    - about the priority queue and binary heap data structure
    - to achieve safety from race conditions on shared mutable data by confinement, immutability, threadsafe data types, and synchronization
    - how to use lock and synchronized methods
    - about deadlock and how to prevent it

- Please continue to work on **Lecture Quiz 13**,  **Lab 13** :
    - to do Lab Exercise 13.1 - 13.4

  and  **Lab 14 Task A, B**