ADVANCED JAVA

Concurrency

Vivek Shah bonii@di.ku.dk

August 21, 2018

DIKU, University of Copenhagen

Writing correct concurrent code is about controlling concurrent access to shared data.

Writing correct concurrent code is about controlling concurrent access to shared data.

We need to make sure that every thead

1. only operates on consistent data, and
2. has an up-to-date view of the system state.

Writing correct concurrent code is about controlling concurrent access to shared data.

We need to make sure that every thead

1. only operates on consistent data, and
2. has an up-to-date view of the system state.

The synchronization mechanisms of Java provide ways to accomplish this goal.

Thread safety

Overview of synchronization mechanisms in Java

Intrinsic locks

Atomicity

Sharing and visibility

Manual locking

High-level concurrency abstractions

General concurrency hazards

WHAT IS THREAD SAFETY?

Thread safety may seem as a somewhat vague concept, with suspiciously circular descriptions such as

> *A piece of code is thread-safe if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time.*
> [https://en.wikipedia.org/wiki/Thread_safety]

- We often have an informal idea of what it means for a program to be *correct*.

THREAD SAFETY PRESUMES CORRECTNESS

- We often have an informal idea of what it means for a program to be *correct*.
- For example, the `ArrayList` class from the Java library is assumed to model list operations.

- We often have an informal idea of what it means for a program to be *correct*.
- For example, the `ArrayList` class from the Java library is assumed to model list operations.
- If `lst` is an `ArrayList` representation of $[v_1, v_2, ..., v_n]$, then after executing `lst.add(x)`, `lst` is a representation of $[v_1, v_2, ..., v_n, x]$.

THREAD SAFETY PRESUMES CORRECTNESS

- We often have an informal idea of what it means for a program to be *correct*.
- For example, the `ArrayList` class from the Java library is assumed to model list operations.
- If `lst` is an `ArrayList` representation of $[v_1, v_2, \ldots, v_n]$, then after executing `lst.add(x)`, `lst` is a representation of $[v_1, v_2, \ldots, v_n, x]$.
- Running `lst.add(x)` and `lst.add(y)` concurrently should result in a list containing `x` and `y` at the end, in some order.

- We often have an informal idea of what it means for a program to be *correct*.
- For example, the `ArrayList` class from the Java library is assumed to model list operations.
- If `lst` is an `ArrayList` representation of $[v_1, v_2, \ldots, v_n]$, then after executing `lst.add(x)`, `lst` is a representation of $[v_1, v_2, \ldots, v_n, x]$.
- Running `lst.add(x)` and `lst.add(y)` concurrently should result in a list containing `x` and `y` at the end, in some order.
- If e.g. one of the elements is lost, then `ArrayList` is not thread-safe.

A SOMEWHAT MORE PRECISE NOTION OF THREAD-SAFETY

A class is thread-safe *if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those thread by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.*

[Goetz, p.18]

A class is thread-safe *if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those thread by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.*

[Goetz, p.18]

To say that a class is thread-safe, you must have a precise idea of what correct behavior is.

Presented in order of preference.

Presented in order of preference.

1. **High level concurrency abstractions.**
    - Thread-safe classes from `java.util.concurrent`.

Presented in order of preference.

1. **High level concurrency abstractions.**
    - Thread-safe classes from `java.util.concurrent`.
2. **Low level locking.** Provides blocking synchronization.
    - `synchronized` methods and blocks.
    - Classes from `java.util.concurrent.locks`.

SYNCHRONIZATION MECHANISMS IN JAVA

Presented in order of preference.

1. **High level concurrency abstractions.**
   - Thread-safe classes from `java.util.concurrent`.
2. **Low level locking.** Provides blocking synchronization.
   - `synchronized` methods and blocks.
   - Classes from `java.util.concurrent.locks`.
3. **Low level primitives.** Provides non-blocking synchronization.
   - Volatile variables.
   - Classes from `java.util.concurrent.atomic`.

Presented in order of preference.

1. **High level concurrency abstractions.**
    - Thread-safe classes from `java.util.concurrent`.
2. **Low level locking.** Provides blocking synchronization.
    - `synchronized` methods and blocks.
    - Classes from `java.util.concurrent.locks`.
3. **Low level primitives.** Provides non-blocking synchronization.
    - Volatile variables.
    - Classes from `java.util.concurrent.atomic`.

Prefer `java.util.concurrent` over manual synchronization.

Presented in order of preference.

1. **High level concurrency abstractions.**
    - Thread-safe classes from `java.util.concurrent`.
2. **Low level locking.** Provides blocking synchronization.
    - `synchronized` methods and blocks.
    - Classes from `java.util.concurrent.locks`.
3. **Low level primitives.** Provides non-blocking synchronization.
    - Volatile variables.
    - Classes from `java.util.concurrent.atomic`.

Prefer `java.util.concurrent` over manual synchronization.

- A logical operation is often composed of many primitive actions.

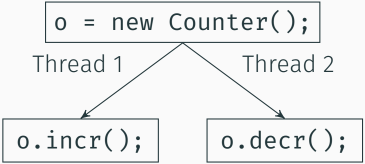
- A logical operation is often composed of many primitive actions.

```
class Counter {
  private int c = 0;
  public void incr()
  { c++; }
  public void decr()
  { c--; }
  public int value()
  { return this.c; }
}
```

A NEED FOR MUTUAL EXCLUSION

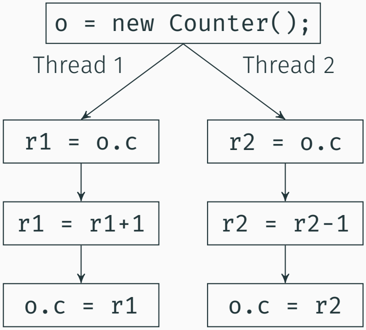
- A logical operation is often composed of many primitive actions.

```
class Counter {
  private int c = 0;
  public void incr()
  { c++; }
  public void decr()
  { c--; }
  public int value()
  { return this.c; }
}
```



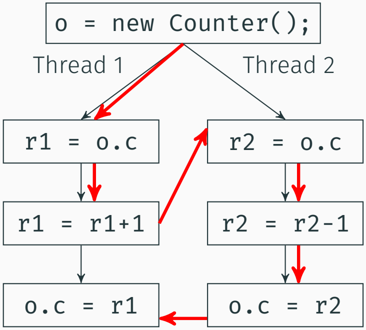
- A logical operation is often composed of many primitive actions.

```
class Counter {
  private int c = 0;
  public void incr()
  { c++; }
  public void decr()
  { c--; }
  public int value()
  { return this.c; }
}
```



- A logical operation is often composed of many primitive actions.

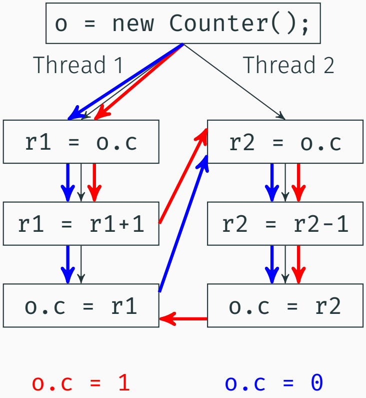
```
class Counter {
  private int c = 0;
  public void incr()
  { c++; }
  public void decr()
  { c--; }
  public int value()
  { return this.c; }
}
```



o.c = 1

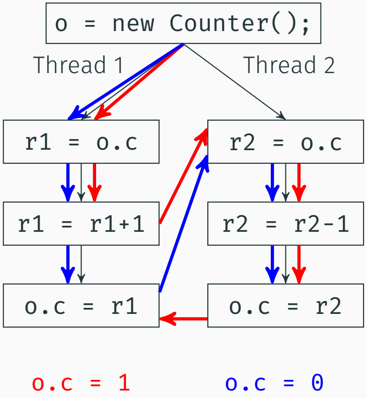
- A logical operation is often composed of many primitive actions.

```
class Counter {
  private int c = 0;
  public void incr()
  { c++; }
  public void decr()
  { c--; }
  public int value()
  { return this.c; }
}
```



- A logical operation is often composed of many primitive actions.

```
class Counter {
  private int c = 0;
  public void incr()
  { c++; }
  public void decr()
  { c--; }
  public int value()
  { return this.c; }
}
```



Lost update possible due to interleaving of compound actions.

- Interleaving happens on the level of instructions generated by the VM.

- Interleaving happens on the level of instructions generated by the VM.
- Compound actions are broken up by VM, leading to interleaving.

- Interleaving happens on the level of instructions generated by the VM.
- Compound actions are broken up by VM, leading to interleaving.
- Program behavior hard to predict.

- Interleaving happens on the level of instructions generated by the VM.
- Compound actions are broken up by VM, leading to interleaving.
- Program behavior hard to predict.
- Visibility issues not even taken into account here (details later).

- We fix the `Counter` program by using `synchronized`:

- We fix the `Counter` program by using `synchronized`:

```
class SynchronizedCounter {
  private int c = 0;
  public synchronized void incr() { c++; }
  public synchronized void decr() { c--; }
  public synchronized int value() { return this.c; }
}
```

- We fix the `Counter` program by using `synchronized`:

```
class SynchronizedCounter {
  private int c = 0;
  public synchronized void incr() { c++; }
  public synchronized void decr() { c--; }
  public synchronized int value() { return this.c; }
}
```

- Thread must acquire exclusive lock associated with object to enter method body.

- We fix the `Counter` program by using `synchronized`:

```
class SynchronizedCounter {
  private int c = 0;
  public synchronized void incr() { c++; }
  public synchronized void decr() { c--; }
  public synchronized int value() { return this.c; }
}
```

- Thread must acquire exclusive lock associated with object to enter method body.
- This is known as an intrinsic lock or a monitor lock.

- We fix the `Counter` program by using `synchronized`:

```
class SynchronizedCounter {
  private int c = 0;
  public synchronized void incr() { c++; }
  public synchronized void decr() { c--; }
  public synchronized int value() { return this.c; }
}
```

- Thread must acquire exclusive lock associated with object to enter method body.
- This is known as an intrinsic lock or a monitor lock.
- At most one thread inside any synchronized method at any time.

- Synchronization must be applied to all methods where the `c` field is accessed.

SYNCHRONIZED METHODS, CONTINUED

- Synchronization must be applied to all methods where the `c` field is accessed.

Wrong!

```
class WrongSynchronizedCounter {
  private int c = 0;
  public synchronized void incr() { c++; }
  public synchronized void decr() { c--; }
  public int value() { return this.c; }
}
```

SYNCHRONIZED METHODS, CONTINUED

- Synchronization must be applied to all methods where the `c` field is accessed.

```
class WrongSynchronizedCounter {
  private int c = 0;
  public synchronized void incr() { c++; }
  public synchronized void decr() { c--; }
  public int value() { return this.c; }
}
```

Wrong!

- One thread can read `c` while another thread is modifying it.

- General form of `synchronized()` aquires intrinsic lock on arbitrary object.

```
synchronized(obj) {
  /* runs while holding lock associated with obj */
}
```

- General form of `synchronized()` aquires intrinsic lock on arbitrary object.

```
synchronized(obj) {
  /* runs while holding lock associated with obj */
}
```

- Synchronized methods syntactic sugar for general form:

```
class Dummy{
  public synchronized void method { /* statements */ }
}
```

is equivalent to

```
class Dummy{
  public void method {
    synchronized(this) { /* statements */ }
  }
}
```

- Use `synchronized` to protect critical regions of code by forcing *serialized* access to data.

INTRINSIC LOCKS

- Use `synchronized` to protect critical regions of code by forcing *serialized* access to data.
- Provides structured locking mechanism - lock and unlock actions well-scoped.

- Use `synchronized` to protect critical regions of code by forcing *serialized* access to data.
- Provides structured locking mechanism - lock and unlock actions well-scoped.
- `synchronized` is reentrant.
    - Same thread can re-aquire the same lock multiple times.

- Use `synchronized` to protect critical regions of code by forcing *serialized* access to data.
- Provides structured locking mechanism - lock and unlock actions well-scoped.
- `synchronized` is reentrant.
    - Same thread can re-aquire the same lock multiple times.
    - Avoids deadlocks for synchronized methods using recursion, callbacks or inheritance.

INTRINSIC LOCKS

- Use `synchronized` to protect critical regions of code by forcing *serialized* access to data.
- Provides structured locking mechanism - lock and unlock actions well-scoped.
- `synchronized` is reentrant.
  - Same thread can re-aquire the same lock multiple times.
  - Avoids deadlocks for synchronized methods using recursion, callbacks or inheritance.
- `synchronized` blocks provide more fine-grained concurrency than methods.
  - Also allows for more interleaving. Use with care.

Let’s have a break.

Thread safety

Overview of synchronization mechanisms in Java

Intrinsic locks

Atomicity

Sharing and visibility

Manual locking

High-level concurrency abstractions

General concurrency hazards

Presented in order of preference.

1. **High level concurrency abstractions.**
    - Thread-safe classes from `java.util.concurrent`.
2. **Low level locking.** Provides blocking synchronization.
    - `synchronized` methods and blocks.
    - Classes from `java.util.concurrent.locks`.
3. **Low level primitives.** Provides non-blocking synchronization.
    - Volatile variables.
    - Classes from `java.util.concurrent.atomic`.

Prefer `java.util.concurrent` over manual synchronization.

- The `java.util.concurrent.atomic` package provides lock-free abstractions for primitive types.

- The `java.util.concurrent.atomic` package provides lock-free abstractions for primitive types.
- All abstractions provide type-specific operations along with a *compare-and-set* operation

```
boolean compareAndSet(expectedValue, updateValue);
```

- The `java.util.concurrent.atomic` package provides lock-free abstractions for primitive types.
- All abstractions provide type-specific operations along with a *compare-and-set* operation

```
boolean compareAndSet(expectedValue, updateValue);
```

- Use caution. While thread-safe, lock-free data structures are notoriously difficult to reason about.
    - See e.g. lock-free queue by Michael & Scott.

```java
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
  private AtomicInteger c;
  public void incr() {
    c.incrementAndGet();
  }
  public void decr() {
    c.decrementAndGet();
  }
  public int value() {
    return c.get();
  }
  public AtomicCounter() {
    this.c = new AtomicInteger();
  }
}
```

"If you can write a high-performance JVM for a modern microprocessor, then you are qualified to think about whether you can avoid synchronizing."

SEQUENTIAL CONSISTENCY

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A:	$M[x] := 1$	$r_1 := M[z]$	$M[y] := r_1$
Program B:	$M[z] := 1$	$r_2 := M[x]$	$M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A: $M[x] := 1$ | $r_1 := M[z]$ | $M[y] := r_1$

Program B: $M[z] := 1$ | $r_2 := M[x]$ | $M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$ $W(x)\ 1$

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A: $M[x] := 1$ | $r_1 := M[z]$ | $M[y] := r_1$

Program B: $M[z] := 1$ | $r_2 := M[x]$ | $M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$ $W(x)\ 1$ $R(z)\ 0$

SEQUENTIAL CONSISTENCY

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A: $M[x] := 1$ $r_1 := M[z]$ $M[y] := r_1$

Program B: $M[z] := 1$ $r_2 := M[x]$ $M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$ $W(x)\ 1$ $R(z)\ 0$ $W(z)\ 1$

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A: $M[x] := 1$ | $r_1 := M[z]$ | $M[y] := r_1$

Program B: $M[z] := 1$ | $r_2 := M[x]$ | $M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$ | $W(x)\ 1$ | $R(z)\ 0$ | $W(z)\ 1$ | $R(x)\ 1$

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A: $M[x] := 1$ $r_1 := M[z]$ $M[y] := r_1$

Program B: $M[z] := 1$ $r_2 := M[x]$ $M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$ $W(x)\ 1$ $R(z)\ 0$ $W(z)\ 1$ $R(x)\ 1$ $W(y)\ 0$

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A: $M[x] := 1$ | $r_1 := M[z]$ | $M[y] := r_1$

Program B: $M[z] := 1$ | $r_2 := M[x]$ | $M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$ | $W(x)\ 1$ | $R(z)\ 0$ | $W(z)\ 1$ | $R(x)\ 1$ | $W(y)\ 0$ | $W(y)\ 1$

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A: $M[x] := 1$ $r_1 := M[z]$ $M[y] := r_1$

Program B: $M[z] := 1$ $r_2 := M[x]$ $M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$ $W(x)\ 1$ $R(z)\ 0$ $W(z)\ 1$ $R(x)\ 1$ $W(y)\ 0$ $W(y)\ 1$

- Reads see the most recent writes.

- *Sequential consistency* means execution of a multi-threaded program appears as if executed one statement at a time following program order.
- How people intuitively think about concurrency.

Program A: $M[x] := 1$ | $r_1 := M[z]$ | $M[y] := r_1$

Program B: $M[z] := 1$ | $r_2 := M[x]$ | $M[y] := r_2$

Execution:

$\boxed{x = y = z = 0}$ $W(x)\ 1$ $R(z)\ 0$ $W(z)\ 1$ $R(x)\ 1$ $W(y)\ 0$ $W(y)\ 1$

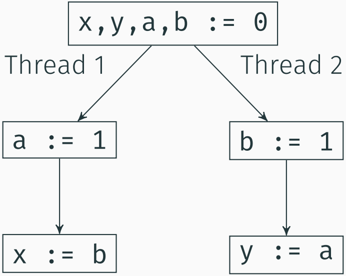
invisible! $R(x)\ 0$ $W(y)\ 0$

- Reads see the most recent writes.
- Not how modern architectures work. Visibility not guaranteed!

- The JVM has a *relaxed memory model*.

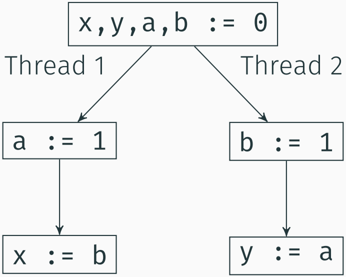
- The JVM has a *relaxed memory model*.
- It is allowed to reorder memory actions while preserving within-thread consistency.

- The JVM has a *relaxed memory model*.
- It is allowed to reorder memory actions while preserving within-thread consistency.
- What are possible values of (x, y) after this program has finished?



REORDERING

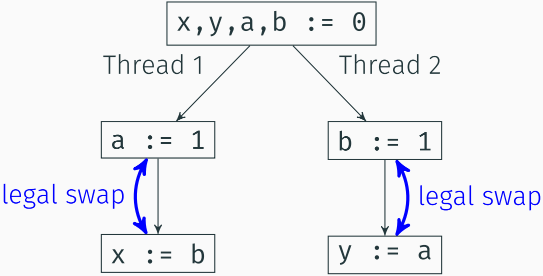
- The JVM has a *relaxed memory model*.
- It is allowed to reorder memory actions while preserving within-thread consistency.
- What are possible values of (x, y) after this program has finished?



Answer: $(1, 0)$, $(0, 1)$, $(1, 1)$ and $(0, 0)$?!

REORDERING

- The JVM has a *relaxed memory model*.
- It is allowed to reorder memory actions while preserving within-thread consistency.
- What are possible values of (x, y) after this program has finished?



Answer: $(1, 0)$, $(0, 1)$, $(1, 1)$ and $(0, 0)$?!

- No within-thread data dependencies, so reordering is legal.

(lack of) sequential consistency

- Sequential consistency is not provided by Java.

(lack of) sequential consistency

- Sequential consistency is not provided by Java.
    - Program variables stored in registers or cache invisible to other CPUs.

(lack of) sequential consistency

- Sequential consistency is not provided by Java.
    - Program variables stored in registers or cache invisible to other CPUs.
    - The JVM might optimize code by reordering actions.

(lack of) sequential consistency

- Sequential consistency is not provided by Java.
    - Program variables stored in registers or cache invisible to other CPUs.
    - The JVM might optimize code by reordering actions.
    - Optimizations are important; sequential consistency would prohibit them.

(lack of) sequential consistency

- Sequential consistency is not provided by Java.
  - Program variables stored in registers or cache invisible to other CPUs.
  - The JVM might optimize code by reordering actions.
  - Optimizations are important; sequential consistency would prohibit them.
- If you try to reason about program behavior without sequential consistency you will lose your mind.

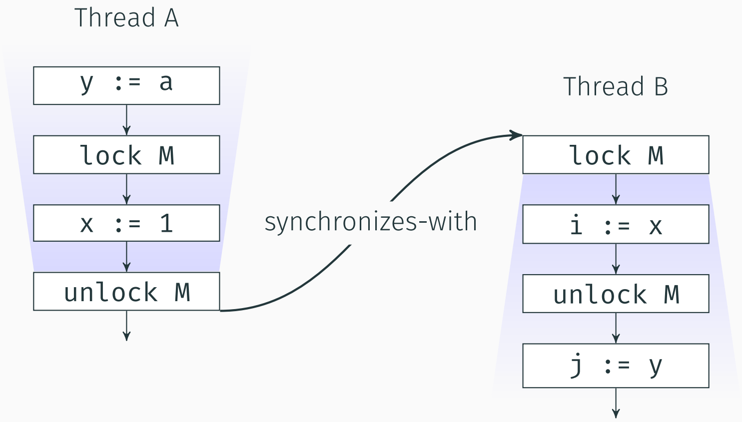
(lack of) sequential consistency

- Sequential consistency is not provided by Java.
    - Program variables stored in registers or cache invisible to other CPUs.
    - The JVM might optimize code by reordering actions.
    - Optimizations are important; sequential consistency would prohibit them.
- If you try to reason about program behavior without sequential consistency you will lose your mind.
- Luckily, Java guarantees sequential consistency for programs that are correctly synchronized.

(lack of) sequential consistency

- Sequential consistency is not provided by Java.
    - Program variables stored in registers or cache invisible to other CPUs.
    - The JVM might optimize code by reordering actions.
    - Optimizations are important; sequential consistency would prohibit them.
- If you try to reason about program behavior without sequential consistency you will lose your mind.
- Luckily, Java guarantees sequential consistency for programs that are correctly synchronized.
- This notion has a clear definition.

Synchronization establishes causality between actions.



All state modifications before the unlock of `M` are visible to all actions coming after the lock of `M`.

HAPPENS-BEFORE

- Synchronization mechanisms establishes happens-before.

- Synchronization mechanisms establishes happens-before.
  - A statement happens-before subsequent statements in program order.
  - Leaving a `synchronized(obj)` block happens-before entering a `synchronized(obj)` block (in another thread).
  - `t.start()` happens-before the thread `t` starts executing.
  - Thread termination for a thread `t` happens-before `t.join()`.
  - A volatile write happens-before a volatile read of the same variable.
  - … (see supplementary material, JSR-133)

- Synchronization mechanisms establishes happens-before.
  - A statement happens-before subsequent statements in program order.
  - Leaving a `synchronized(obj)` block happens-before entering a `synchronized(obj)` block (in another thread).
  - `t.start()` happens-before the thread `t` starts executing.
  - Thread termination for a thread `t` happens-before `t.join()`.
  - A volatile write happens-before a volatile read of the same variable.
  - ... (see supplementary material, JSR-133)
- A program is correctly synchronized if for all sequential executions and for every action `r` reading variable *v*, the most recent write to *v* happens-before `r`.

Presented in order of preference.

1. **High level concurrency abstractions.**
    - Thread-safe classes from `java.util.concurrent`.
2. **Low level locking.** Provides blocking synchronization.
    - `synchronized` methods and blocks.
    - Classes from `java.util.concurrent.locks`.
3. **Low level primitives.** Provides non-blocking synchronization.
    - Volatile variables.
    - Classes from `java.util.concurrent.atomic`.

Prefer `java.util.concurrent` over manual synchronization.

```java
class NoVisibility extends Thread {
    boolean stop = false;

    public static void main(String[] args) {
        NoVisibility t = new NoVisibility();
        t.start();
        Thread.sleep(1000);
        t.stop = true;
        System.out.println("Waiting for thread to stop");
        t.join();
    }
    public void run() {
        while (!stop) {}
        System.out.println("Stop received!");
    }
}
```

VOLATILE VARIABLES

```
class NoVisibility extends Thread {
    boolean stop = false;

    public static void main(String[] args) {
        NoVisibility t = new NoVisibility();
        t.start();
        Thread.sleep(1000);
        t.stop = true;
        System.out.println(”Waiting for thread to stop”);
        t.join();
    }
    public void run() {
        while (!stop) {}
        System.out.println(”Stop received!”);
    }
}
```

Wrong!

Correct

```
class NoVisibility extends Thread {
    volatile boolean stop = false;

    public static void main(String[] args) {
        NoVisibility t = new NoVisibility();
        t.start();
        Thread.sleep(1000);
        t.stop = true;
        System.out.println("Waiting for thread to stop");
        t.join();
    }
    public void run() {
        while (!stop) {}
        System.out.println("Stop received!");
    }
}
```

Let's have a break.

Presented in order of preference.

1. **High level concurrency abstractions.**
    - Thread-safe classes from `java.util.concurrent`.
2. **Low level locking.** Provides blocking synchronization.
    - `synchronized` methods and blocks.
    - Classes from `java.util.concurrent.locks`.
3. **Low level primitives.** Provides non-blocking synchronization.
    - Volatile variables.
    - Classes from `java.util.concurrent.atomic`.

Prefer `java.util.concurrent` over manual synchronization.

- `synchronized` is based on implicit reentrant locks.

- `synchronized` is based on implicit reentrant locks.
- Locks have the same memory visibility guarantees as `synchronized` and `volatile`.

- `synchronized` is based on implicit reentrant locks.
- Locks have the same memory visibility guarantees as `synchronized` and `volatile`.
- Why manual locking when `synchronized` seems to work fine?

- `synchronized` is based on implicit reentrant locks.
- Locks have the same memory visibility guarantees as `synchronized` and `volatile`.
- Why manual locking when `synchronized` seems to work fine?
    - There is no way to interrupt a `synchronized` waiting for a lock.
    - Cannot specify a wait timeout for `synchronized` blocks.
    - Acquire/release must occur in the same block. Also defined statically.

The `java.util.concurrent.locks` package provide explicit locking mechanisms.

```
public interface Lock {
    void lock();
    void lockInterruptibly()
      throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit)
      throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

- Always wrap critical region in `try-finally`.

```
class Dummy {
  public void f() {
    Lock lock = new ReentrantLock();
    /* ... */
    lock.lock();
    try {
     /* update object state */
    }
    finally { lock.unlock(); }
  }
}
```

- Failing to do so is a ticking bomb – deadlocks will happen on exceptions.
- Not necessary when code is guaranteed not to throw exceptions.

```java
class LockedCounter {
    private ReentrantLock lock = new ReentrantLock();
    private int c = 0;
    public void incr() {
        lock.lock();
        c++;
        lock.unlock();
    }
    public void decr() {
        lock.lock();
        c--;
        lock.unlock();
    }
    public int value() {
        lock.lock();
        return this.c;
        lock.unlock();
    }
}
```

EXAMPLE

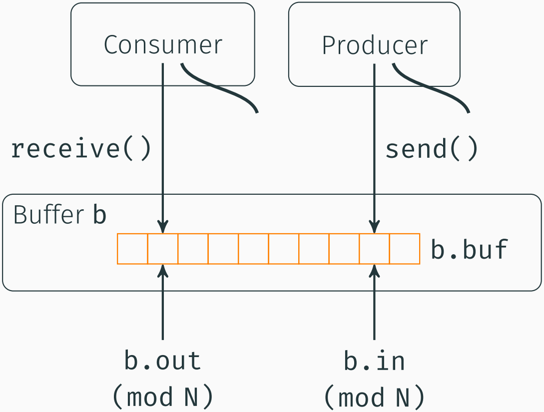
```java
class LockedCounter {
    private ReentrantLock lock = new ReentrantLock();
    private int c = 0;
    public void incr() {
        lock.lock();
        c++;
        lock.unlock();
    }
    public void decr() {
        lock.lock();
        c--;
        lock.unlock();
    }
    public int value() {
        lock.lock();
        return this.c;
        lock.unlock();
    }
}
```

Wrong!

EXAMPLE

Correct

```java
class LockedCounter {
    private ReentrantLock lock = new ReentrantLock();
    private int c = 0;
    public void incr() {
        lock.lock();
        c++;
        lock.unlock();
    }
    public void decr() {
        lock.lock();
        c--;
        lock.unlock();
    }
    public int value() {
        int temp;
        lock.lock();
        temp = this.c;
        lock.unlock();
        return temp;
    }
}
```

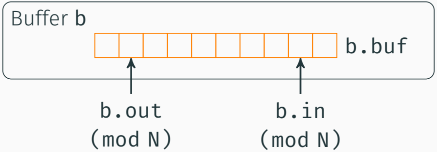

Consumer
Producer
receive()
send()
Buffer b
b.buf
b.out
(mod N)
b.in
(mod N)

BOUNDED BUFFER PSEUDOCODE

(Assume visibility of all writes)

```
def send(b, msg):
  while b.in - b.out == N: do nothing
  b.buf[b.in mod N] := msg
  b.in := b.in + 1

def receive(b):
  while b.in = b.out: do nothing
  msg := b.buf[b.out mod N]
  b.out := b.out + 1
  return msg
```



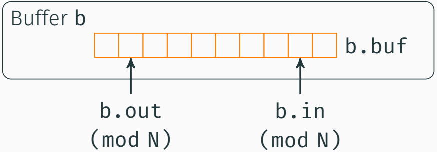
BOUNDED BUFFER PSEUDOCODE

(Assume visibility of all writes)

```
def send(b, msg):
  while b.in - b.out == N: do nothing
  b.buf[b.in mod N] := msg
  b.in := b.in + 1

def receive(b):
  while b.in = b.out: do nothing
  msg := b.buf[b.out mod N]
  b.out := b.out + 1
  return msg
```

Race condition if more than one producer



BUSY WAITING

- Guarded blocks are the most common co-ordination pattern.
- Keep polling a condition across threads.
- Wastes machine cycle with useless work due to polling if the condition is not satisfied.

- Guarded blocks are the most common co-ordination pattern.
- Keep polling a condition across threads.
- Wastes machine cycle with useless work due to polling if the condition is not satisfied.
- Implementing bounded buffer by busy waiting:
    - Lock free implementation possible $\rightarrow$ low synchronization overhead.
    - We will have to use locks to support multiple producers/consumers.
    - Will waste CPU cycles if producer and consumer does not work at same pace, or if throughput low.

See `BusyWaitBoundedBuffer.java` in handout material.

See `BusyWaitBoundedBuffer.java` in handout material.

1. If you suspend a thread, respond to interrupts.
2. Thread safety is relative to specification/usage. Document the supported threading scenarios.

- All `Object`s in Java provide the following methods

```
class Object {
   /* ... */
   void wait();      void wait(long timeout);
   void notify();    void notifyAll(); }
```

WAIT/NOTIFY

- All `Objects` in Java provide the following methods

```
class Object {
   /* ... */
   void wait();      void wait(long timeout);
   void notify();   void notifyAll(); }
```

- When thread *t* calls `obj.wait()`:
    - Intrinsic lock on `obj` is released (thread must own it!)

wait/notify

- All `Object`s in Java provide the following methods

```
class Object {
   /* ... */
   void wait();      void wait(long timeout);
   void notify();   void notifyAll(); }
```

- When thread *t* calls `obj.wait()`:
  - Intrinsic lock on `obj` is released (thread must own it!)
  - Thread is suspended.

wait/notify

- All `Object`s in Java provide the following methods

```
class Object {
   /* ... */
   void wait();      void wait(long timeout);
   void notify();    void notifyAll(); }
```

- When thread *t* calls `obj.wait()`:
  - Intrinsic lock on `obj` is released (thread must own it!)
  - Thread is suspended.
- When *t* resumes it re-aquires lock; or blocks if unavailable.

WAIT/NOTIFY

- All `Object`s in Java provide the following methods

```
class Object {
   /* ... */
   void wait();      void wait(long timeout);
   void notify();    void notifyAll(); }
```

- When thread *t* calls `obj.wait()`:
    - Intrinsic lock on `obj` is released (thread must own it!)
    - Thread is suspended.
- When *t* resumes it re-aquires lock; or blocks if unavailable.
- Thread *t* resumes when one of the following happens:
    - Another thread calls `obj.notifyAll()`; or `obj.notify()` and *t* was chosen arbitrarily by runtime.

WAIT/NOTIFY

- All `Object`s in Java provide the following methods

```
class Object {
   /* ... */
   void wait();     void wait(long timeout);
   void notify();   void notifyAll(); }
```

- When thread *t* calls `obj.wait()`:
  - Intrinsic lock on `obj` is released (thread must own it!)
  - Thread is suspended.
- When *t* resumes it re-aquires lock; or blocks if unavailable.
- Thread *t* resumes when one of the following happens:
  - Another thread calls `obj.notifyAll()`; or `obj.notify()` and *t* was chosen arbitrarily by runtime.
  - Timer runs out (if timed version was used).

- All `Object`s in Java provide the following methods

```
class Object {
   /* ... */
   void wait();      void wait(long timeout);
   void notify();    void notifyAll(); }
```

- When thread *t* calls `obj.wait()`:
  - Intrinsic lock on `obj` is released (thread must own it!)
  - Thread is suspended.
- When *t* resumes it re-aquires lock; or blocks if unavailable.
- Thread *t* resumes when one of the following happens:
  - Another thread calls `obj.notifyAll()`; or `obj.notify()` and *t* was chosen arbitrarily by runtime.
  - Timer runs out (if timed version was used).
  - *t* was interrupted.

WAIT/NOTIFY

- All `Objects` in Java provide the following methods

```
class Object {
   /* ... */
   void wait();      void wait(long timeout);
   void notify();    void notifyAll(); }
```

- When thread *t* calls `obj.wait()`:
  - Intrinsic lock on `obj` is released (thread must own it!)
  - Thread is suspended.
- When *t* resumes it re-aquires lock; or blocks if unavailable.
- Thread *t* resumes when one of the following happens:
  - Another thread calls `obj.notifyAll()`; or `obj.notify()` and *t* was chosen arbitrarily by runtime.
  - Timer runs out (if timed version was used).
  - *t* was interrupted.
  - Spurious wake (rare, but possible).

wait/notify: correct usage

- It is an error to call `wait()` or `notify()` outside a `synchronized` block.

- It is an error to call `wait()` or `notify()` outside a `synchronized` block.
- Due to possibility of spurious wakes, always *spin* on a wait condition:

```
synchronized(obj) {
  while (!condition) { obj.wait(); }
}
```

See `WaitNotifyBoundedBuffer.java` in handout material.

- `Condition`s enable threads to wait and notify on *conditions*.

- `Condition`s enable threads to wait and notify on *conditions*.
- Condition objects created via `Lock` interface.

- `Condition`s enable threads to wait and notify on *conditions*.
- Condition objects created via `Lock` interface.
- Can be seen as a fine-grained wait/notify mechanism:
    - `wait()` $\simeq$ `await()`
    - `notify()` $\simeq$ `signal()`
    - `notifyAll()` $\simeq$ `signalAll()`
- Optimized syntactic sugared version of wait/notify.

BOUNDEDBUFFER WITH CONDITION OBJECTS

See `ConditionBoundedBuffer.java` in handout material.

Presented in order of preference.

1. **High level concurrency abstractions.**
    - Thread-safe classes from `java.util.concurrent`.
2. **Low level locking.** Provides blocking synchronization.
    - `synchronized` methods and blocks.
    - Classes from `java.util.concurrent.locks`.
3. **Low level primitives.** Provides non-blocking synchronization.
    - Volatile variables.
    - Classes from `java.util.concurrent.atomic`.

Prefer `java.util.concurrent` over manual synchronization.

- Low-level, manual synchronization is the *assembly-language of concurrent programming*. Powerful, but hard to get right.

- Low-level, manual synchronization is the *assembly-language of concurrent programming*. Powerful, but hard to get right.
- The `java.util.concurrent` package contains many useful concurrency abstractions which are thread-safe without any extra synchronization.

- Low-level, manual synchronization is the *assembly-language of concurrent programming*. Powerful, but hard to get right.
- The `java.util.concurrent` package contains many useful concurrency abstractions which are thread-safe without any extra synchronization.
- They are correct and fast – **use them before rolling your own concurrency abstraction!**

- Concurrent collections from `java.util.concurrent`.
    - `ConcurrentHashMap`, `CopyOnWriteArrayList`, ...

- Concurrent collections from `java.util.concurrent`.
    - `ConcurrentHashMap`, `CopyOnWriteArrayList`, …
    - Offers greater concurrency than external synchronization with little risk.

- Concurrent collections from `java.util.concurrent`.
    - `ConcurrentHashMap`, `CopyOnWriteArrayList`, ...
    - Offers greater concurrency than external synchronization with little risk.
    - Concurrent versions of common compound operations, e.g. `ConcurrentHashMap.putIfAbsent(key, value)`.

CONCURRENT COLLECTIONS

- Concurrent collections from `java.util.concurrent`.
    - `ConcurrentHashMap`, `CopyOnWriteArrayList`, ...
    - Offers greater concurrency than external synchronization with little risk.
    - Concurrent versions of common compound operations, e.g. `ConcurrentHashMap.putIfAbsent(key, value)`.
    - Compound actions still require synchronization!

- Bounded buffers as seen earlier in this lecture. E.g. `BlockingQueue`, `BlockingDeque`, ...

- Bounded buffers as seen earlier in this lecture. E.g. `BlockingQueue`, `BlockingDeque`, ...
- Great for safe, throttled communication between threads.

SYNCHRONIZERS

- A *synchronizer* is an object that coordinates control flow of threads based on its state.

- A *synchronizer* is an object that coordinates control flow of threads based on its state.
- Useful synchronizers from `java.util.concurrent`:

  Latches. Blocks until condition satisfied, then all threads are allowed to proceed. For example `CountDownLatch`.

- A *synchronizer* is an object that coordinates control flow of threads based on its state.
- Useful synchronizers from `java.util.concurrent`:

Latches. Blocks until condition satisfied, then all threads are allowed to proceed. For example `CountDownLatch`.

Asynchronous tasks. Spawn computation and use its result later. If it is not finished, block. See e.g. `FutureTask` and the Executor framework.

- A *synchronizer* is an object that coordinates control flow of threads based on its state.
- Useful synchronizers from `java.util.concurrent`:

Latches. Blocks until condition satisfied, then all threads are allowed to proceed. For example `CountDownLatch`.

Asynchronous tasks. Spawn computation and use its result later. If it is not finished, block. See e.g. `FutureTask` and the Executor framework.

Semaphores. Provides blocking “resource permits”. Can be used to implement e.g. a bounded queue on top of a regular queue. See `Semaphore`.

- A *synchronizer* is an object that coordinates control flow of threads based on its state.
- Useful synchronizers from `java.util.concurrent`:

Latches. Blocks until condition satisfied, then all threads are allowed to proceed. For example `CountDownLatch`.

Asynchronous tasks. Spawn computation and use its result later. If it is not finished, block. See e.g. `FutureTask` and the Executor framework.

Semaphores. Provides blocking “resource permits”. Can be used to implement e.g. a bounded queue on top of a regular queue. See `Semaphore`.

Barriers. Similar to latches, but may be reset. All threads must meet at a particular point in time. Often used in simulations. See `CyclicBarrier`, `Exchanger`.

LIVENESS

- Safety: *"Nothing bad ever happens"*.
- Liveness: *"Something good eventually happens"*.

- Safety: *"Nothing bad ever happens"*.
- Liveness: *"Something good eventually happens"*.
- Hazards to liveness:

Deadlock Threads wait for each other indefinitely, making no progress.

Starvation A thread needs a resource but is persistently denied access to it.

Livelock A thread does not make progress because it persistently fails and retries an operation. May happen with e.g. *optimistic concurrency*.

Synchronization is hard - try not to be too clever. Use existing abstractions when possible.