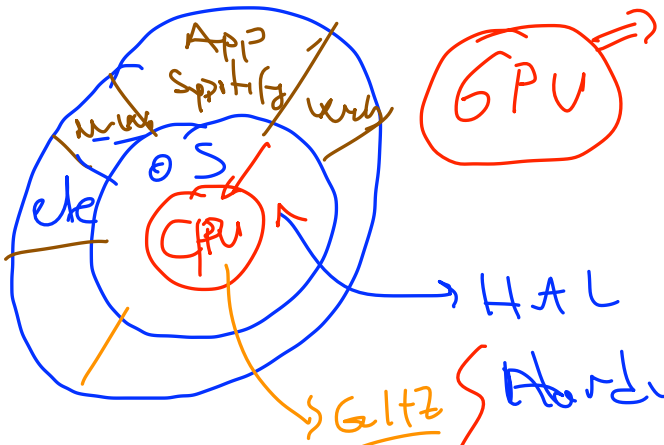
12 Threading

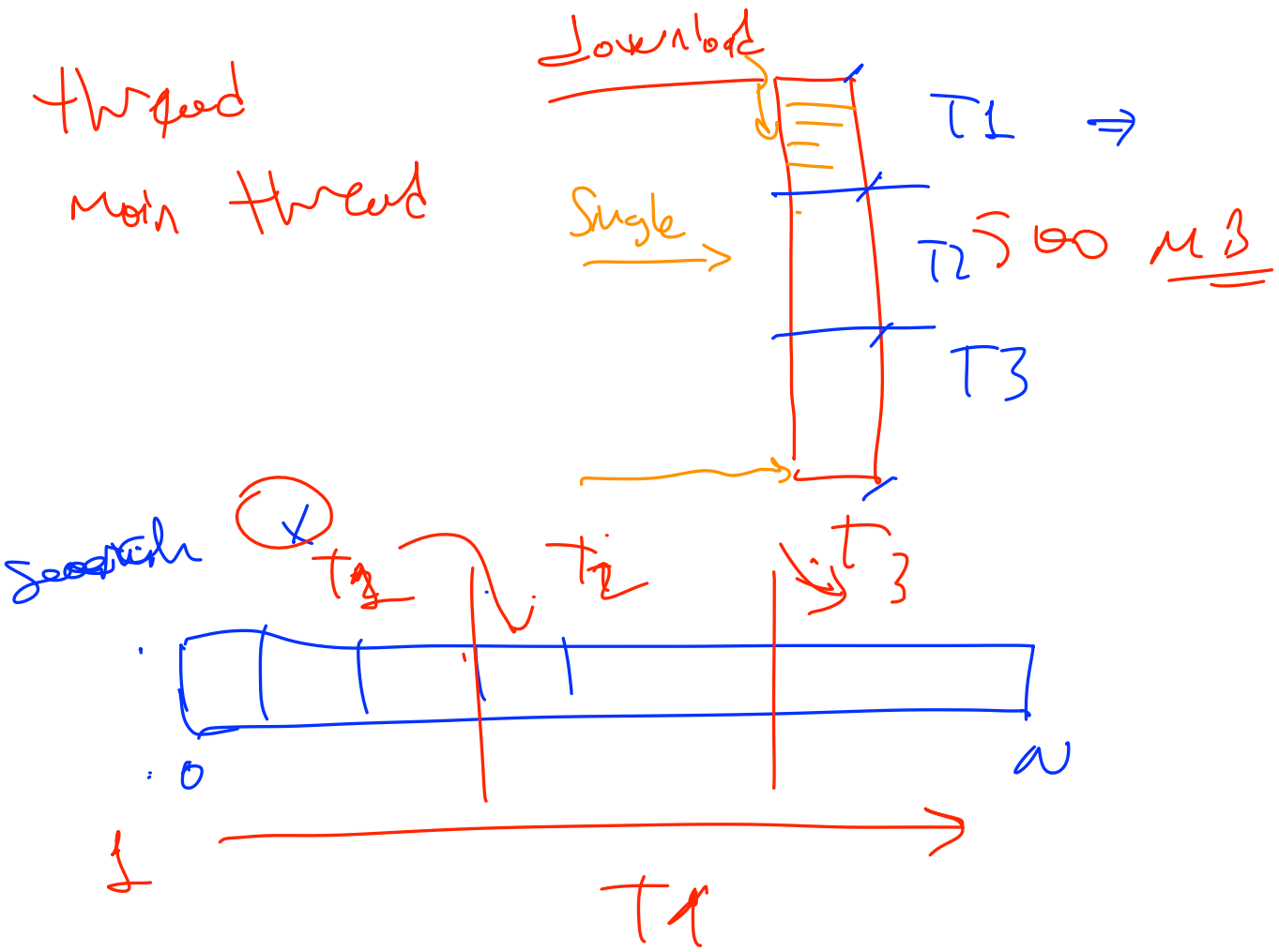


OS = Windows
Linux
Mac
Solaris
Unix

P1 P2 P3 P4

P1 P2 P3 P1 P2 P3 P4

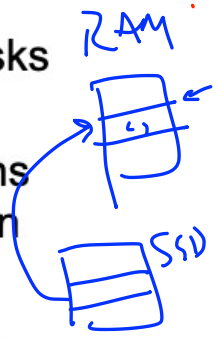
T

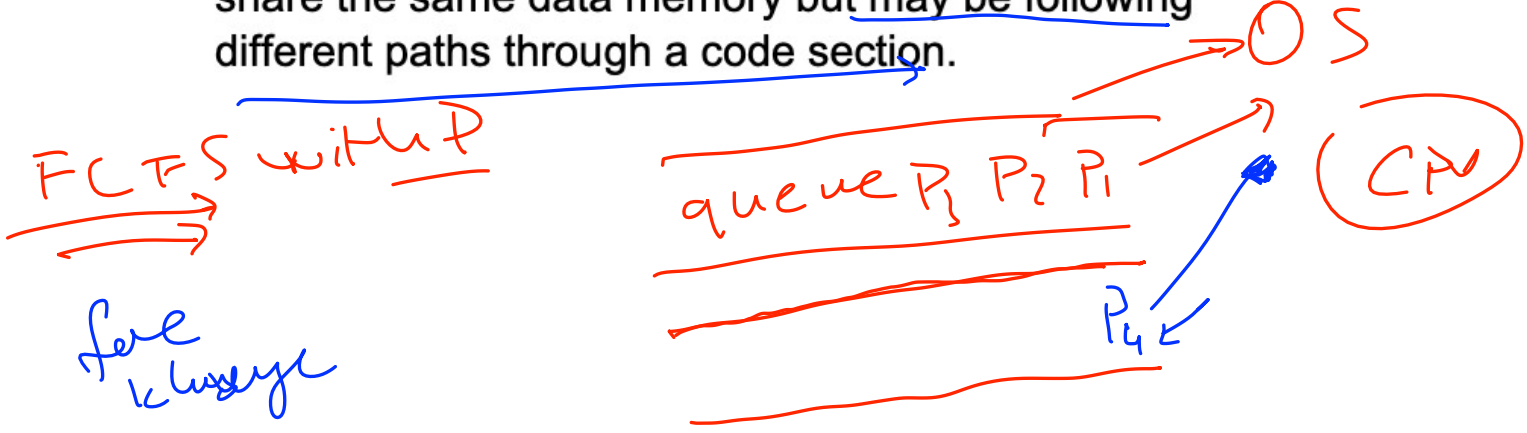


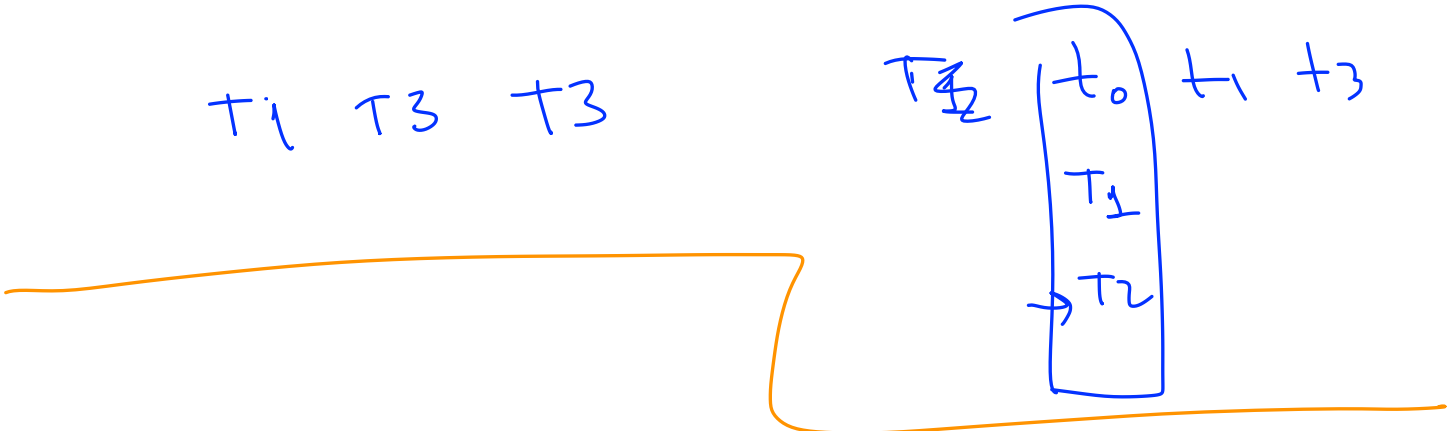
Task Scheduling

Modern operating systems use preemptive multitasking to allocate CPU time to applications. There are two types of tasks that can be scheduled for execution:

- Processes: A process is an area of memory that contains both code and data. A process has a thread of execution that is scheduled to receive CPU time slices.
- Thread: A thread is a scheduled execution of a process. Concurrent threads are possible. All threads for a process share the same data memory but may be following different paths through a code section.



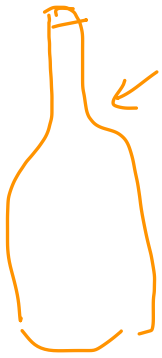


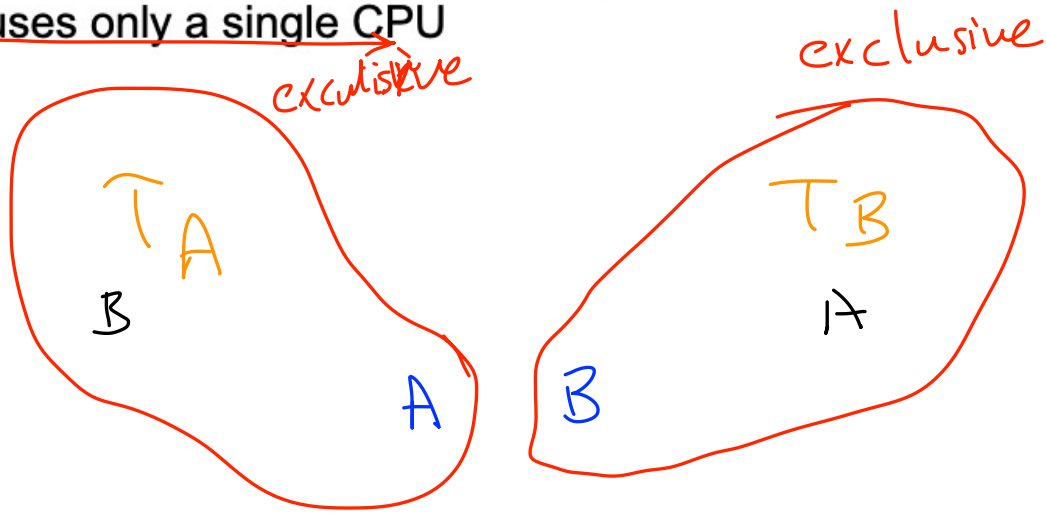


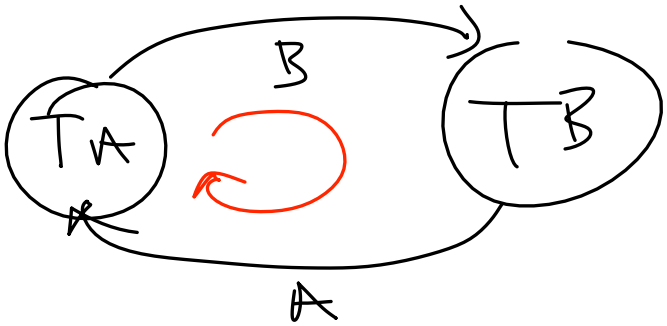
Why Threading Matters

To execute a program as quickly as possible, you must avoid performance bottlenecks. Some of these bottlenecks are:

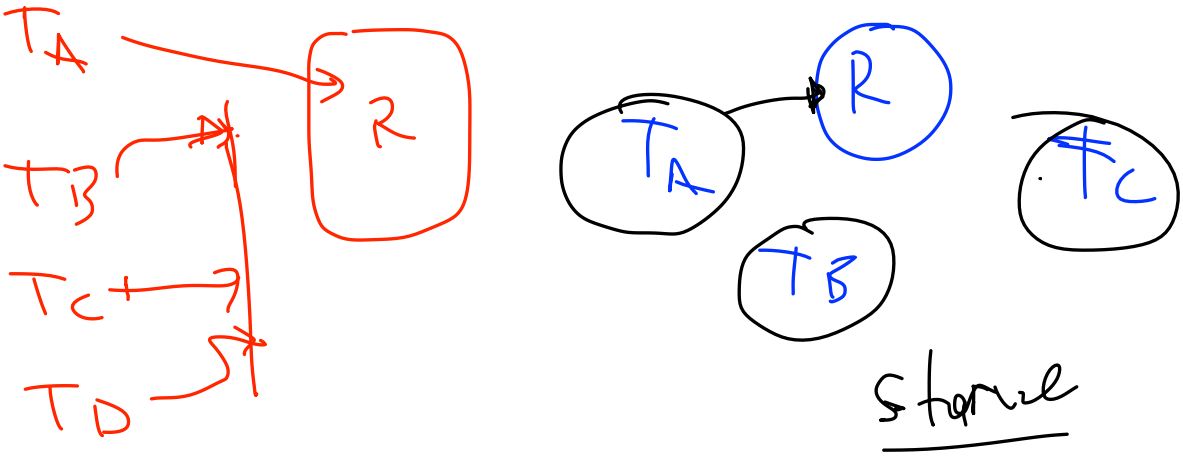
- Resource Contention: Two or more tasks waiting for exclusive use of a resource
- Blocking I/O operations: Doing nothing while waiting for disk or network data transfers
- Underutilization of CPUs: A single-threaded application uses only a single CPU

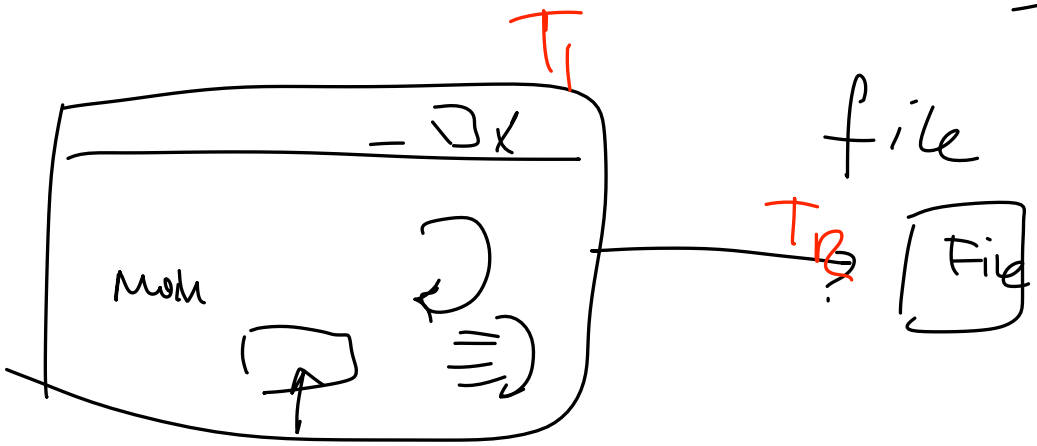






deadlock





The Thread Class

The `Thread` class is used to create and start threads. Code to be executed by a thread must be placed in a class, which does either of the following:

- Extends the `Thread` class
  - Simpler code
- Implements the `Runnable` interface
  - More flexible
  - `extends` is still free.

Extending Thread

Extend `java.lang.Thread` and override the `run` method:

```
public class ExampleThread extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

Starting a Thread

After creating a new `Thread`, it must be started by calling the `Thread`'s `start` method:

```
public static void main(String[] args) {
    ExampleThread t1 = new ExampleThread();
    t1.start();
}
```

Schedules the run method to be called

Implementing `Runnable`

Implement `java.lang.Runnable` and implement the `run` method:

```
public class ExampleRunnable implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

public interface Runnable {

public void run();

}

Executing `Runnable` Instances

After creating a new `Runnable`, it must be passed to a `Thread` constructor. The `Thread`'s `start` method begins execution:

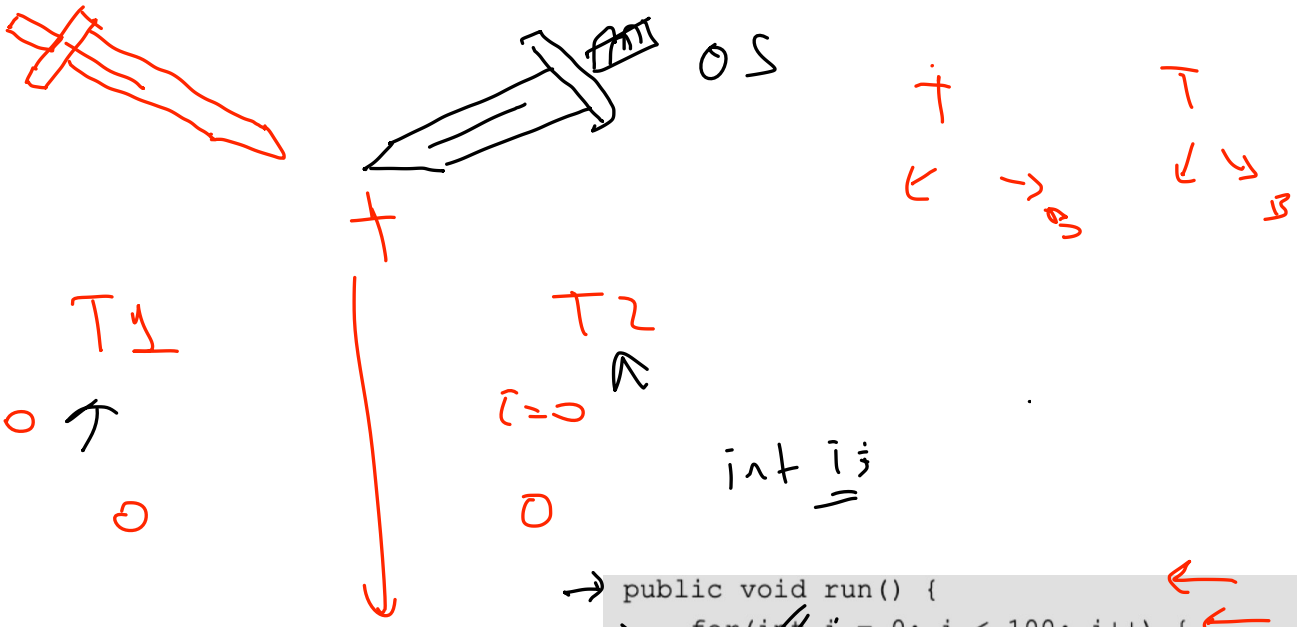
```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
}
```

One Runnable: Multiple Threads

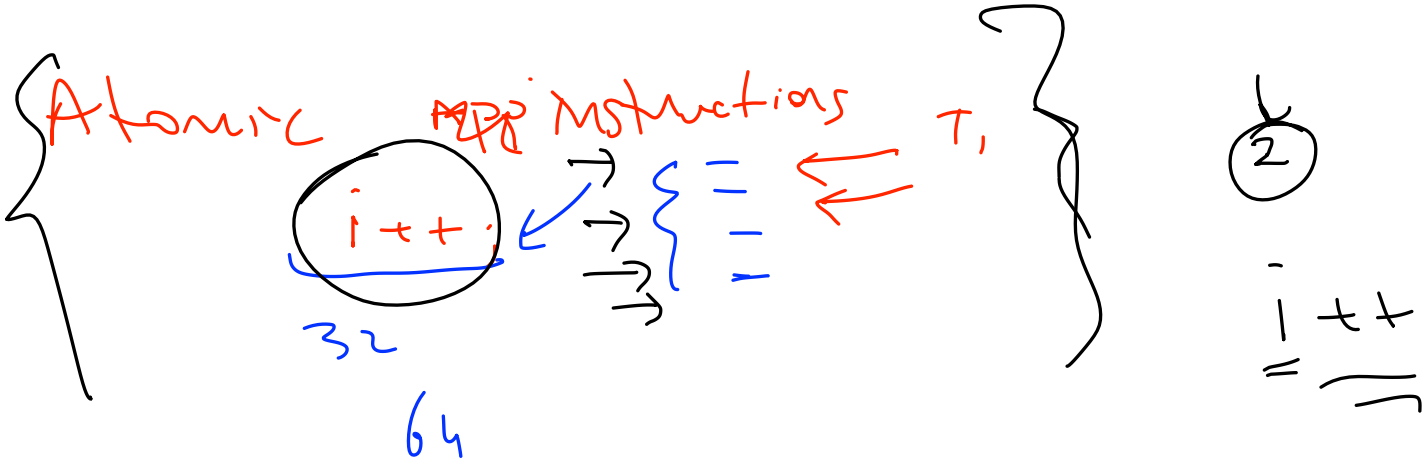
An object that is referenced by multiple threads can lead to instance fields being concurrently accessed.

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    Thread t2 = new Thread(r1);
    t2.start();
}
```

A single Runnable instance




```
public void run() {
    for(int i = 0; i < 100; i++) {
        System.out.println("i:" + i);
    }
}
```



Thread Safe →

Problems with Shared Data

Shared data must be accessed cautiously. Instance and static fields:

- Are created in an area of memory known as heap space
- Can potentially be shared by any thread
- Might be changed concurrently by multiple threads
  - There are no compiler or IDE warnings.
  - "Safely" accessing shared fields is your responsibility.

The preceding slides might produce the following:

```
i:0,i:0,i:1,i:2,i:3,i:4,i:5,i:6,i:7,i:8,i:9,i:10,i:12,i:11 ...
```

Zero produced twice

Out of sequence

Nonshared Data

Some variable types are never shared. The following types are always thread-safe:

- Local variables → run() ✓
- Method parameters →
- Exception handler parameters →

Atomic Operations

Atomic operations function as a single operation. A single statement in the Java language is not always atomic.

- `i++;`
  - Creates a temporary copy of the value in `i`
  - Increments the temporary copy
  - Writes the new value back to `i`
- `l = 0xffff_ffff_ffff_ffff;`
  - 64-bit variables might be accessed using two separate 32-bit operations.

What inconsistencies might two threads incrementing the same field encounter?

What if that field is long?

Inconsistent Behavior

One possible problem with two threads incrementing the same field is that a lost update might occur. Imagine if both threads read a value of 41 from a field, increment the value by one, and then write their results back to the field. Both threads will have done an increment but the resulting value is only 42. Depending on how the Java Virtual Machine is implemented and the type of physical CPU being used, you may never or rarely see this behavior. However, you must always assume that it could happen.

If you have a long value of 0x0000_0000_ffff_ffff and increment it by 1, the result should be 0x0000_0001_0000_0000. However, because it is legal for a 64-bit field to be accessed using two separate 32-bit writes, there could temporarily be a value of 0x0000_0001_ffff_ffff or even 0x0000_0000_0000_0000 depending on which bits are modified first. If a second thread was allowed to read a 64-bit field while it was being modified by another thread, an incorrect value could be retrieved.

Out-of-Order Execution

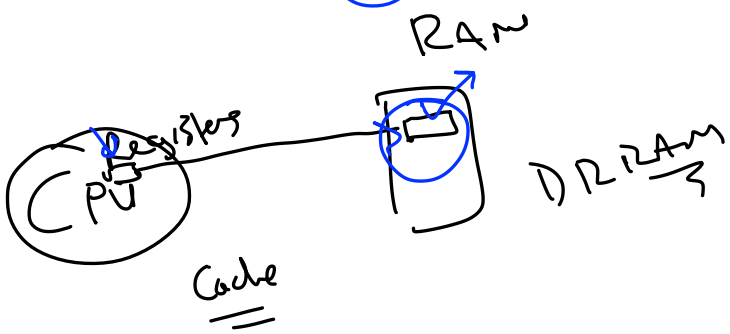
- Operations performed in one thread may not appear to execute in order if you observe the results from another thread.
  - Code optimization may result in out-of-order operation.
  - Threads operate on cached copies of shared variables.
- To ensure consistent behavior in your threads, you must synchronize their actions.
  - You need a way to state that an action happens before another.
  - You need a way to flush changes to shared variables back to main memory.

The `volatile` Keyword

A field may have the `volatile` modifier applied to it:

```
public volatile int i;
```

- Reading or writing a `volatile` field will cause a thread to synchronize its working memory with main memory.
- `volatile` does not mean atomic.
  - If `i` is `volatile`, `i++` is still not a thread-safe operation.



Stopping a Thread

A thread stops by completing its `run` method.

```
public class ExampleRunnable implements Runnable {
    public volatile boolean timeToQuit = false;

    @Override
    public void run() {
        System.out.println("Thread started");
        while(!timeToQuit) {
            // ...
        }
        System.out.println("Thread finishing");
    }
}
```

Shared volatile variable

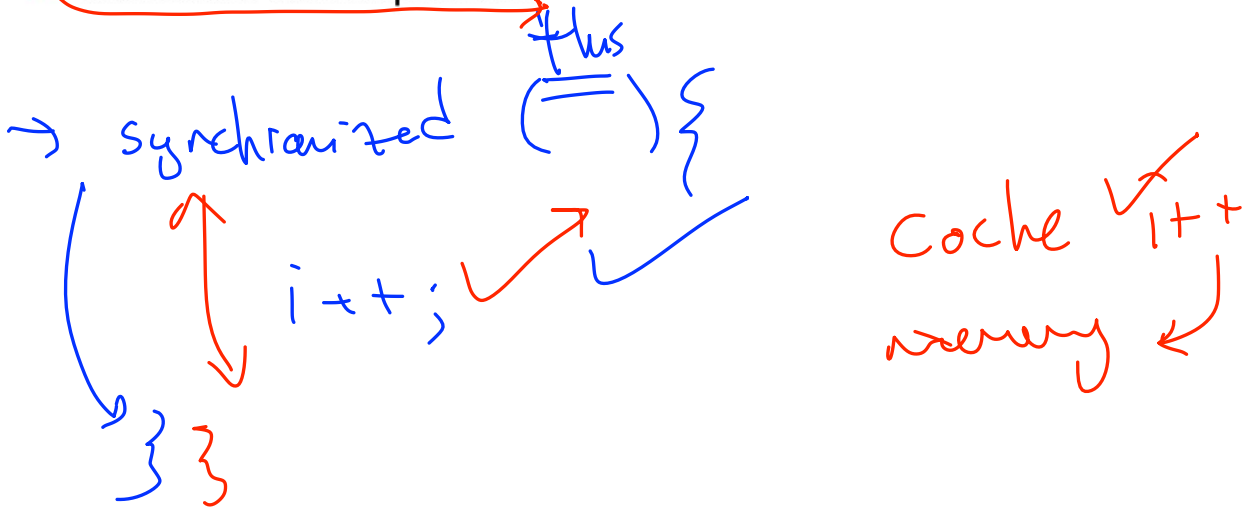
Stopping a Thread

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    // ...
    r1.timeToQuit = true;
}
```

The `synchronized` Keyword

The `synchronized` keyword is used to create thread-safe code blocks. A `synchronized` code block:

- Causes a thread to write all of its changes to main memory when the end of the block is reached
  - Similar to `volatile`
- Is used to group blocks of code for exclusive execution
  - Threads block until they can get exclusive access
  - Solves the atomic problem



synchronized Methods

```
public class ShoppingCart {
    private List<Item> cart = new ArrayList<>();
    public synchronized void addItem(Item item) {
        cart.add(item);
    }
    public synchronized void removeItem(int index) {
        cart.remove(index);
    }
    public synchronized void printCart() {
        Iterator<Item> ii = cart.iterator();
        while(ii.hasNext()) {
            Item i = ii.next();
            System.out.println("Item:" + i.getDescription());
        }
    }
}
```

```
public void printCart() {
    StringBuilder sb = new StringBuilder();
    synchronized (this) {
        Iterator<Item> ii = cart.iterator();
        while (ii.hasNext()) {
            Item i = ii.next();
            sb.append("Item:");
            sb.append(i.getDescription());
            sb.append("\n");
        }
    }
    System.out.println(sb.toString());
}
```

Object Monitor Locking

Each object in Java is associated with a monitor, which a thread can lock or unlock.

- `synchronized` methods use the monitor for the `this` object.
- `static synchronized` methods use the classes' monitor.
- `synchronized` blocks must specify which object's monitor to lock or unlock.

```
synchronized ( this ) { }
```

- `synchronized` blocks can be nested.

Detecting Interruption

Interrupting a thread is another possible way to request that a thread stop executing.

```
public class ExampleRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread started");
        while(!Thread.interrupted()) {
            // ...
        }
        System.out.println("Thread finishing");
    }
}
```

static `Thread` method

Interrupting a Thread

Every thread has an `interrupt()` and `isInterrupted()` method.

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    // ...
    t1.interrupt();
}
```

Interrupt a thread

Thread.sleep()

A `Thread` may pause execution for a duration of time.

```
long start = System.currentTimeMillis();
try {
    Thread.sleep(4000);
} catch (InterruptedException ex) {
    // What to do?
}
long time = System.currentTimeMillis() - start;
System.out.println("Slept for " + time + " ms");
```

`interrupt()` called while sleeping

Additional `Thread` Methods

- There are many more `Thread` and threading-related methods:
  - `setName(String)`, `getName()`, and `getId()`
  - `isAlive()`: Has a thread finished?
  - `isDaemon()` and `setDaemon(boolean)`: The JVM can quit while daemon threads are running.
  - `join()`: A current thread waits for another thread to finish.
  - `Thread.currentThread()`: `Runnable` instances can retrieve the `Thread` instance currently executing.
- The `Object` class also has methods related to threading:
  - `wait()`, `notify()`, and `notifyAll()`: Threads may go to sleep for an undetermined amount of time, waking only when the `Object` they waited on receives a wakeup notification.

Methods to Avoid

Some `Thread` methods should be avoided:

- `setPriority(int)` and `getPriority()`
  - Might not have any impact or may cause problems
- The following methods are deprecated and should never be used:
  - `destroy()`
  - `resume()`
  - `suspend()`
  - `stop()`

Deadlock

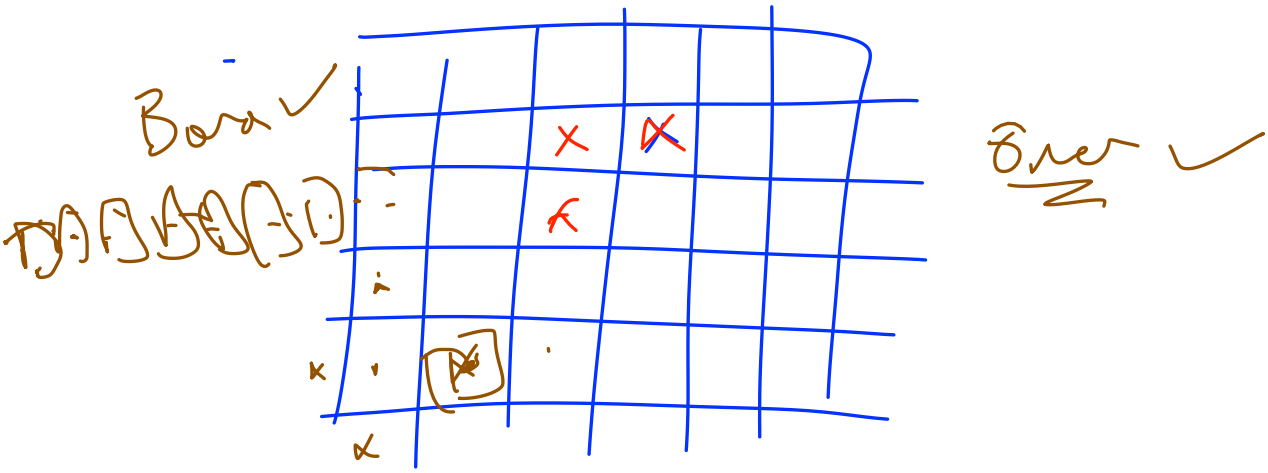
Deadlock results when two or more threads are blocked forever, waiting for each other.

```
synchronized(obj1) {
  synchronized(obj2) {
  }
}
```

Thread 1 pauses after locking obj1's monitor.

```
synchronized(obj2) {
  synchronized(obj1) {
  }
}
```

Thread 2 pauses after locking obj2's monitor.



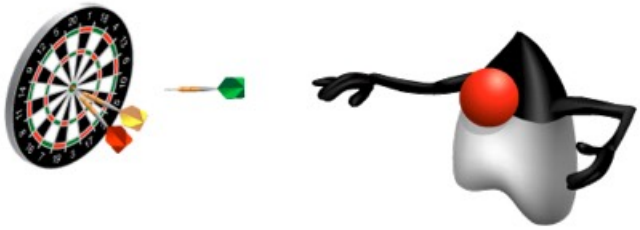
13

Concurrency

Objectives

After completing this lesson, you should be able to:

- Use atomic variables
- Use a `ReentrantReadWriteLock`
- Use the `java.util.concurrent` collections
- Describe the synchronizer classes
- Use an `ExecutorService` to concurrently execute tasks
- Apply the Fork-Join framework



The `java.util.concurrent` Package

Java 5 introduced the `java.util.concurrent` package, which contains classes that are useful in concurrent programming. Features include:

- Concurrent collections
- Synchronization and locking alternatives
- Thread pools
  - Fixed and dynamic thread count pools available
  - Parallel divide and conquer (Fork-Join) new in Java 7

The java.util.concurrent.atomic Package

The java.util.concurrent.atomic package contains classes that support lock-free thread-safe programming on single variables

```
AtomicInteger ai = new AtomicInteger(5);
if(ai.compareAndSet(5, 42)) {
    System.out.println("Replaced 5 with 42");
}
```

An atomic operation ensures that the current value is 5 and then sets it to 42.

The java.util.concurrent.locks Package

The java.util.concurrent.locks package is a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.

```
public class ShoppingCart {
    private final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();

    public void addItem(Object o) {
        rwl.writeLock().lock();
        // modify shopping cart
        rwl.writeLock().unlock();
    }
```

A single writer, multi-reader lock

Write Lock

java.util.concurrent.locks

```
public String getSummary() {
    String s = "";
    rwl.readLock().lock();
    // read cart, modify s
    rwl.readLock().unlock();
    return s;
}
public double getTotal() {
    // another read-only method
}
}
```

Read Lock

All read-only methods can concurrently execute.

Thread-Safe Collections

The `java.util` collections are not thread-safe. To use collections in a thread-safe fashion:

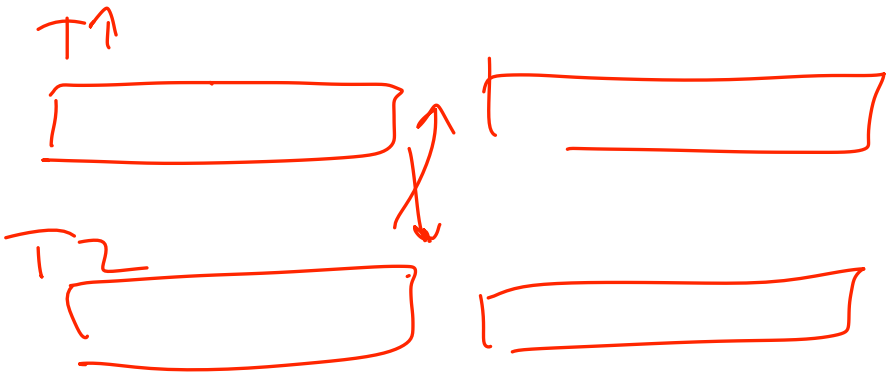
- Use synchronized code blocks for all access to a collection if writes are performed
- Create a synchronized wrapper using library methods, such as `java.util.Collections.synchronizedList(List<T>)`
- Use the `java.util.concurrent` collections

Note: Just because a `Collection` is made thread-safe, this does not make its elements thread-safe.

Synchronizers

The `java.util.concurrent` package provides five classes that aid common special-purpose synchronization idioms.

Class	Description
Semaphore	Semaphore is a classic concurrency tool.
CountDownLatch	A very simple yet very common utility for blocking until a given number of signals, events, or conditions hold
CyclicBarrier	A resettable multiway synchronization point useful in some styles of parallel programming
Phaser	Provides a more flexible form of barrier that may be used to control phased computation among multiple threads
Exchanger	Allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs



`java.util.concurrent.CyclicBarrier`

The `CyclicBarrier` is an example of the synchronizer category of classes provided by `java.util.concurrent`.

```
final CyclicBarrier barrier = new CyclicBarrier(2);

new Thread() {
    public void run() {
        try {
            System.out.println("before await - thread 1");
            barrier.await();
            System.out.println("after await - thread 1");
        } catch (BrokenBarrierException|InterruptedException ex) {

        }
    }
}.start();
```

Two threads must await before they can unblock.

May not be reached

CyclicBarrier Behavior

In this example, if only one thread calls await() on the barrier, that thread may block forever. After a second thread calls await(), any additional call to await() will again block until the required number of threads is reached. A CyclicBarrier contains a method, await(long timeout, TimeUnit unit), which will block for a specified duration and throw a TimeoutException if that duration is reached.

High-Level Threading Alternatives

Traditional `Thread` related APIs can be difficult to use properly. Alternatives include:

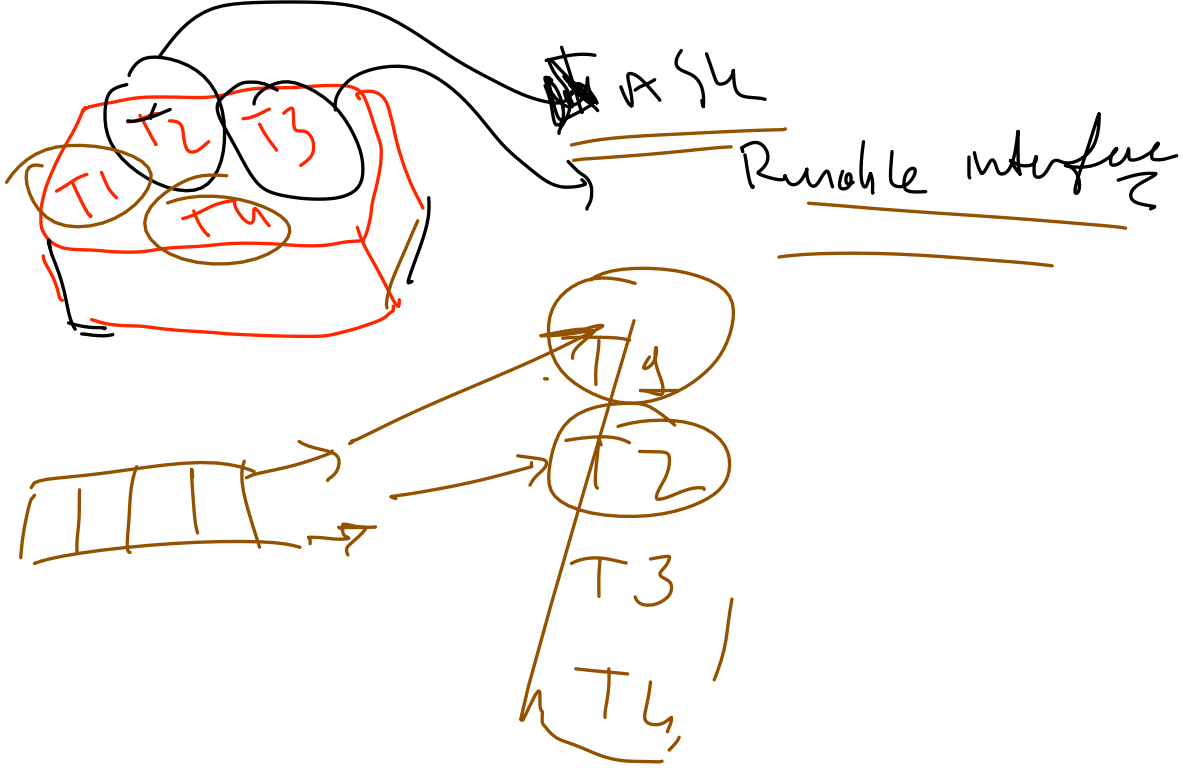
- `java.util.concurrent.ExecutorService`, a higher level mechanism used to execute tasks
  - It may create and reuse `Thread` objects for you.
  - It allows you to submit work and check on the results in the future.
- The Fork-Join framework, a specialized work-stealing `ExecutorService` new in Java 7

java.util.concurrent.ExecutorService

An `ExecutorService` is used to execute tasks.

- It eliminates the need to manually create and manage threads.
- Tasks **might** be executed in parallel depending on the `ExecutorService` implementation.
- Tasks can be:
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
- Implementing instances can be obtained with `Executors`.

```
ExecutorService es = Executors.newCachedThreadPool();
```



https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

https://sourceforge.net/projects/javaconcurrenta/