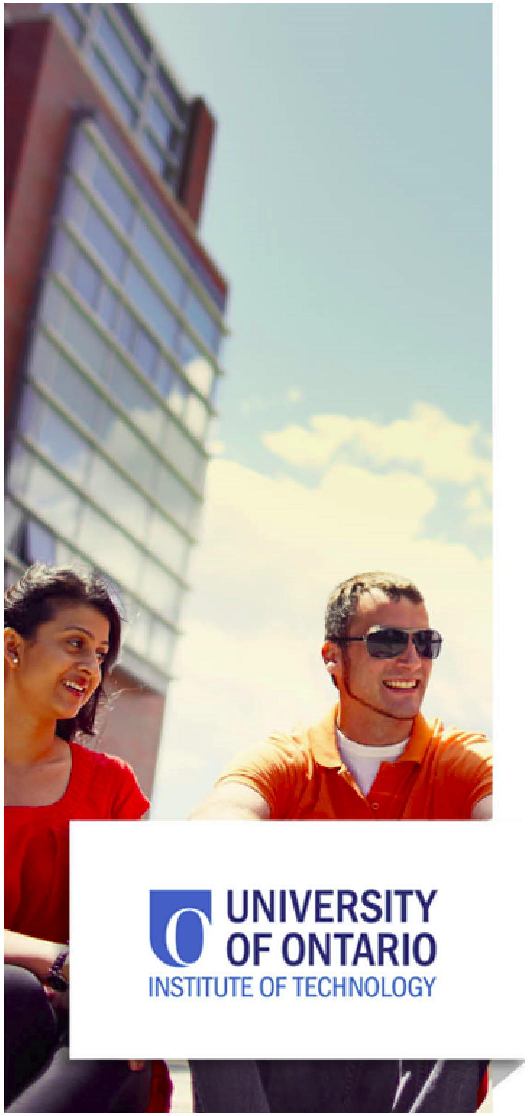


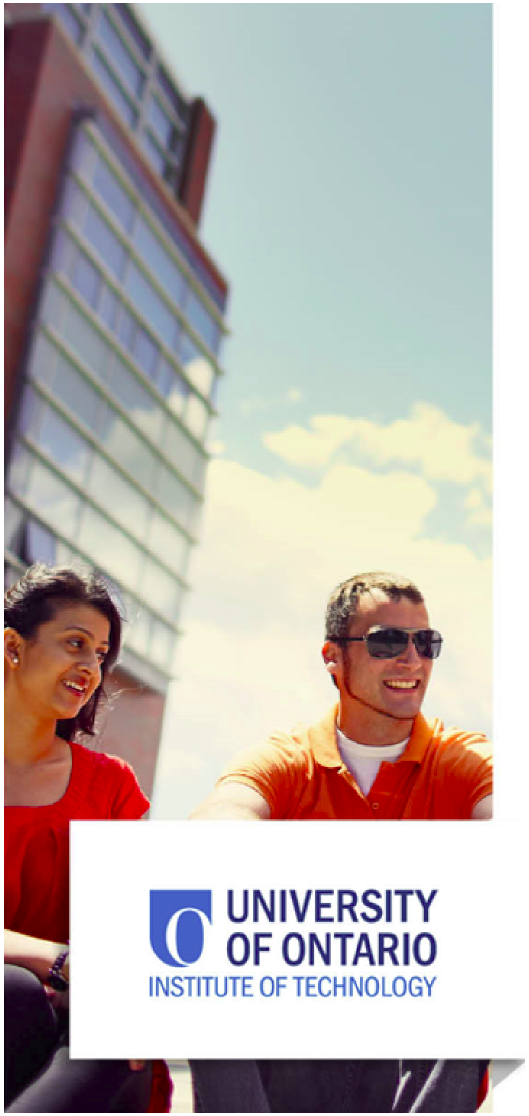
Multi-threading

Parallel Execution

Randy J. Fortier
randy.fortier@uoit.ca
@randy_fortier

Outline

- Threads vs. processes
- Pros and cons
- Implementation
  - Runnable
  - Thread
  - Interrupts
  - Joins
- Concurrency models
  - Parallel worker
  - Reactive
  - Fork/join
- Synchronization



Multi-threading

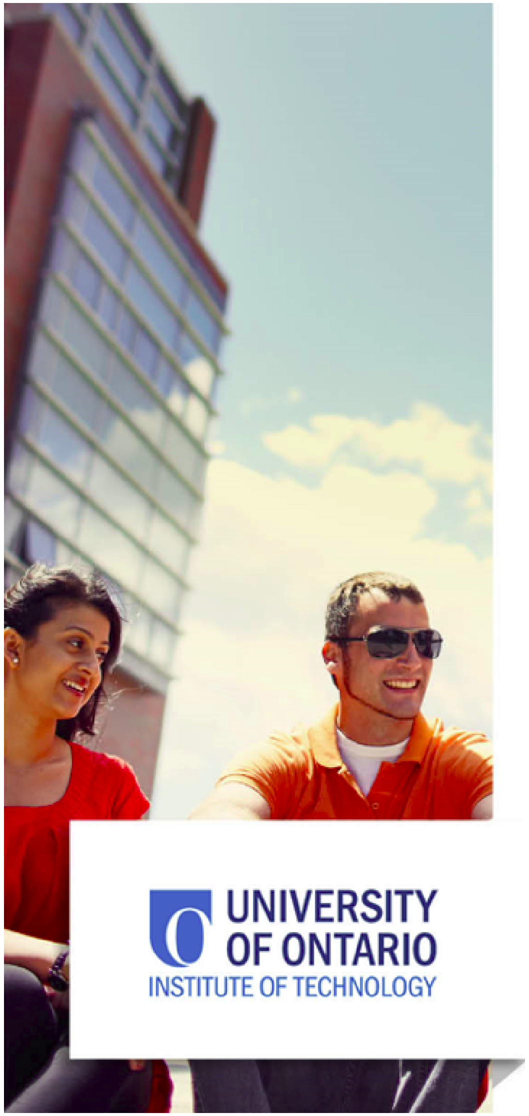
What are threads?

Threads vs. Processes

- Process
  - Can run in parallel
  - Created at the OS level
  - Has own memory space (segmentation)
- Thread
  - Can run in parallel (within a process)
  - Created at the language or OS level
  - Share a single memory space
  - Lightweight context switching
  - Simpler communication

Threads

- Pros
  - Take advantage of multi-core processors
  - Responsive UI
- Cons
  - Complexity
  - Unpredictability
  - Shared data synchronization



Multi-threading

Implementing Threads

Runnable

- An interface that you can implement
- You can pass this to a thread to execute

```
public class ClientConnectionHandler implements Runnable {
    public void run() {
        // do something concurrently
    }
}
...
ClientConnectionHandler handler = new ClientConnectionHandler();
Thread handlerThread = new Thread(handler, "Handler Thread");
handlerThread.start();
```

Thread

- You can also sub-class Thread directly

```
public class ClientThread extends Thread {
    public ClientThread(String name) {
        super(name);
    }

    public void run() {
        // do something concurrently
    }
}
...
ClientThread handlerThread = new ClientThread("Handler Thread");
handlerThread.start();
```

Interrupts

- For long calculations, we may want to interrupt a thread
  - e.g. if we figure out that this entire calculation is non-optimal

```
public class ClientConnectionHandler implements Runnable {
    public void run() {
        boolean moreToDo = true;
        while (moreToDo) {
            ... do some work ...
            if (Thread.interrupted())
                return;
            ... update moreToDo ...
        }
    }
}
...
Thread handlerThread = new Thread(new ClientConnectionHandler());
handlerThread.start();
...
handlerThread.interrupt();
```

Joins

- When sub-threads are no longer doing anything, you can join them:
  - t.join(): Wait indefinitely for thread 't' to die
  - t.join(millis): Wait up to 'millis' milliseconds for thread 't' to die
  - t.join(millis, nanos): Wait up to 'millis' milliseconds and 'nanos' nanoseconds for thread 't' to die

```
ClientThread handlerThread = new ClientThread("Handler Thread");
handlerThread.start();
... do stuff ...
handlerThread.join(200);
```

Yield

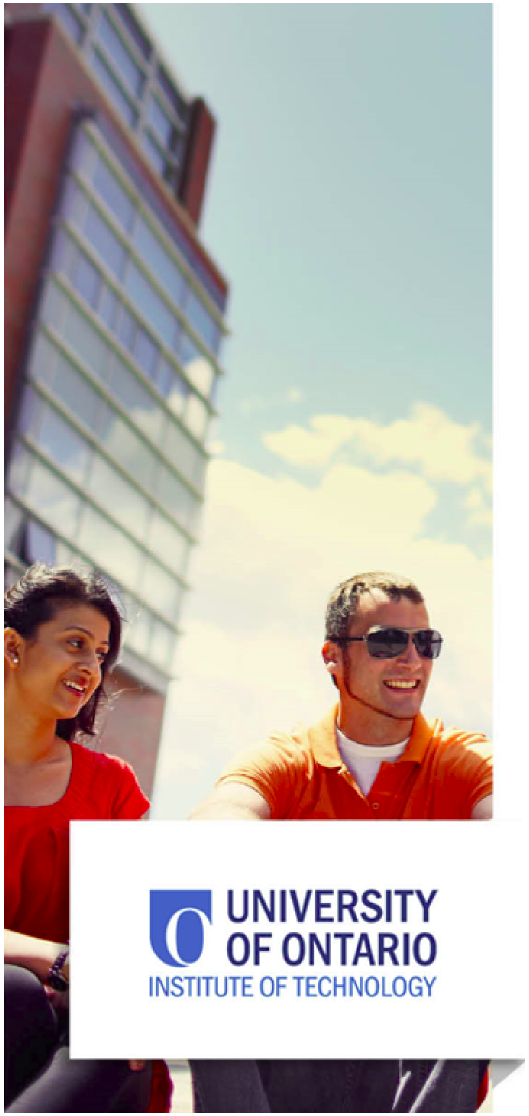
- To hint that the thread scheduler that another thread should be given CPU time instead
- Unlike join(), this thread may still have more work to do

```
ClientThread handlerThread = new ClientThread("Handler Thread");
handlerThread.start();
... do stuff ...
Thread.yield();
```

Thread Methods

- currentThread(): Get the current thread
- sleep(1500): Put the current thread into sleep state
- getName(): Get a thread's name

```
ClientThread handlerThread = new ClientThread("Handler Thread");
handlerThread.start();
...
Thread.sleep(1000);
System.out.println("Thread name: " + Thread.currentThread().getName(
...
```



Multi-threading

Concurrency Models

Concurrency Models

- Parallel Worker
- Reactive
- Fork/Join

Concurrency Models

- Parallel Worker

```
WorkerThread[] workerThreads = new WorkerThread[NUM_WORKERS];
SharedState sharedState = ...
for (int i = 0; i < NUM_WORKERS; i++) {
   workerThreads[i] = new WorkerThread(sharedState);
   workerThreads[i].start();
}
```

Concurrency Models

- Reactive
  - When events occur, create threads as needed
  - e.g. new chat client -> create thread to handle it

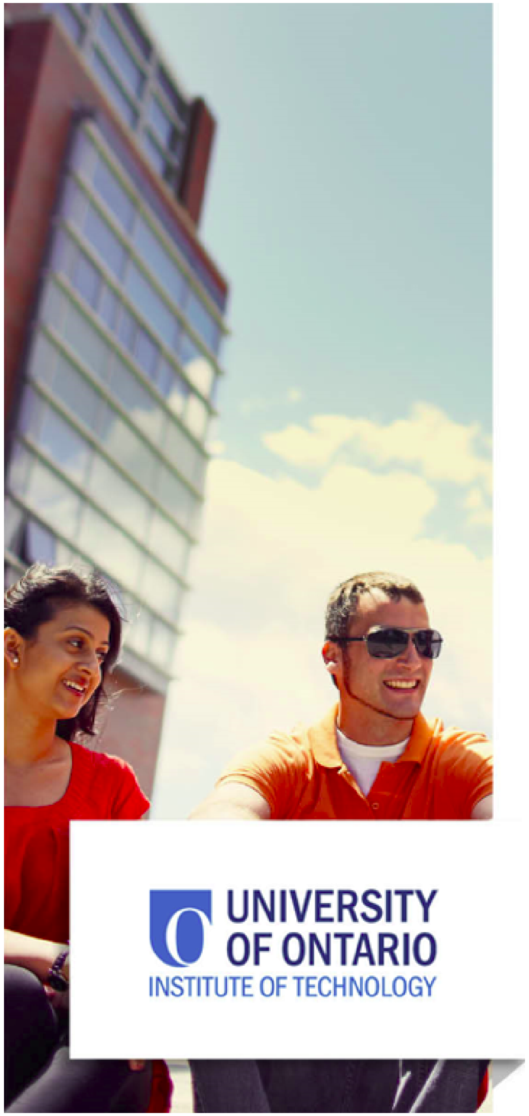
```
app.get('/clients', function(req, res){
  res.send('clients:');
  for (var i = 0; i < clients.length; i++) {
    res.send(clients[i]);
  }
});

app.get('/products', function(req, res){
  res.send('products:');
  for (var i = 0; i < products.length; i++) {
    res.send(products[i]);
  }
});
```

Concurrency Models

- Fork/Join
- Tutorial using ForkJoin library in Java

```
if (problem.size < SMALL_PROBLEM) {
    ... solve the problem directly ...
} else {
    ... split problem into problem1 and problem2 ...
    ProblemThread t1 = new ProblemThread(problem1);
    ProblemThread t2 = new ProblemThread(problem2);
    t1.start();
    t2.start();
}
```



Multi-threading

Data Synchronization

Data Synchronization

- Many concurrency models involve shared data
- Simultaneous shared access to data can cause problems
- e.g.
  - Thread 1: Reads account balance ($200)
  - Thread 2: Reads account balance ($200)
  - Thread 1: Reduces balance ($200-$150=$50)
  - Thread 2: Reduces balance ($50-$100=-$50)
- This is an example of a *race condition*

Critical Section

- The previous example had two threads with *critical sections*
  - A critical section is a block of code that requires *mutual exclusion* for one or more data values
  - Some programming languages support a *mutex* (the mutex acts as a resource lock when the resource is in use)
  - For more detailed access control a *semaphore* can be used (an object/registry that manages access to the resource)

Mutex

- A mutex can be as simple as a boolean variable:

```
class SharedState {
    private boolean inUse = false;
    private List>Customer> customers = null;
    public void addCustomer(Customer cust) throws ResourceInUseExcep
        if (inUse) {
            throw new ResourceInUseException("Customers list in use"
        } else {
            inUse = true;
            ... manipulate customers ...
            inUse = false;
        }
    }
}
```

Data Synchronization

- An application is called *thread safe* if it takes measures to avoid race conditions:
  - Immutable/copied data (similar to pass by value)
    - Shared state is copied
    - Copy cannot be modified
  - Resource locks
    - Only one thread can hold the lock at once
    - The thread that has the lock can write

Shared Resources

- Shared resources:
  - Objects to which multiple threads have access
  - e.g. Public/static variable
  - e.g. Database, files
- Non-shared resources:
  - Local variables in the run() method
  - Private instance variables in the Thread/Runnable
  - Except: These these variables store object references/pointers to objects defined elsewhere (on the heap)

Volatile

- Due to each core of a CPU having its own cache, it is possible that changes to a shared object cannot be observed by other threads
- The solution: volatile
- Volatile guarantees the strict ordering of reads/writes

```
class SharedState {
    public volatile Map<String,Customer> customers = ...
}
```

Synchronized

- A synchronized block prevents multiple threads executing code at once
- Varieties:
  - Methods
  - Static methods
  - Code blocks

Synchronized

- A synchronized block prevents multiple threads executing code at once
- Varieties:
  - Methods:

```
class SharedState {
    public synchronized void placeOrder(Order order) {
        ...
    }
}
```

Synchronized

- A synchronized block prevents multiple threads executing code at once
- Varieties:
  - Methods
  - Static methods:

```
class SharedState {
    static Map<String,Customer> customers;
    public static synchronized Map<String,Customer> getCustomers() {
        return customers;
    }
}
```

Synchronized

- A synchronized block prevents multiple threads executing code at once
- Varieties:
  - Methods
  - Static methods
  - Code blocks:

```
class SharedState {
    private float balance = 0f;
    private List<Transaction> transactions = null;
    public boolean preauthorize(float amount) {
        if (amount > balance) {
            synchronized {
                transactions.add(new Transaction("Suspicious preauth
            }
            return false;
        }
        return true;
    }
}
```

Wrap-Up

- In this section we learned about:
  - Threads
  - Concurrency models
  - Shared resources