

Concurrency

Michael C. Hackett
Computer Science Department

Community
College
of Philadelphia

Lecture Topics

- Concurrency
 - Concurrent/Parallel Programming
- Threads and Multithreading
 - The Thread Class
 - The Runnable Interface
- Synchronization
- Joining Threads
- Common Issues with Concurrency

Colors/Fonts

• Local Variable Names	—	Brown
• Primitive data types	—	Fuchsia
• Literals	—	Blue
• Keywords	—	Orange
• Object names	—	Green
• Operators/Punctuation	—	Black
• Field Names	—	Lt Blue
• Method Names	—	Purple
• Parameter Names	—	Gold
• Comments	—	Gray
• Package Names	—	Pink

Source Code — **Consolas**
Output — Courier New

Concurrency

- The programs you have been creating (though much of the code has been object-oriented and reusable) have all only been able to do one thing at a time. For example:
 - The program can't continue until some loop is finished.
 - We couldn't have the program doing something in the background while waiting for a user to type in a keyboard entry.
- In other words, we haven't made programs with the ability to multi-task.

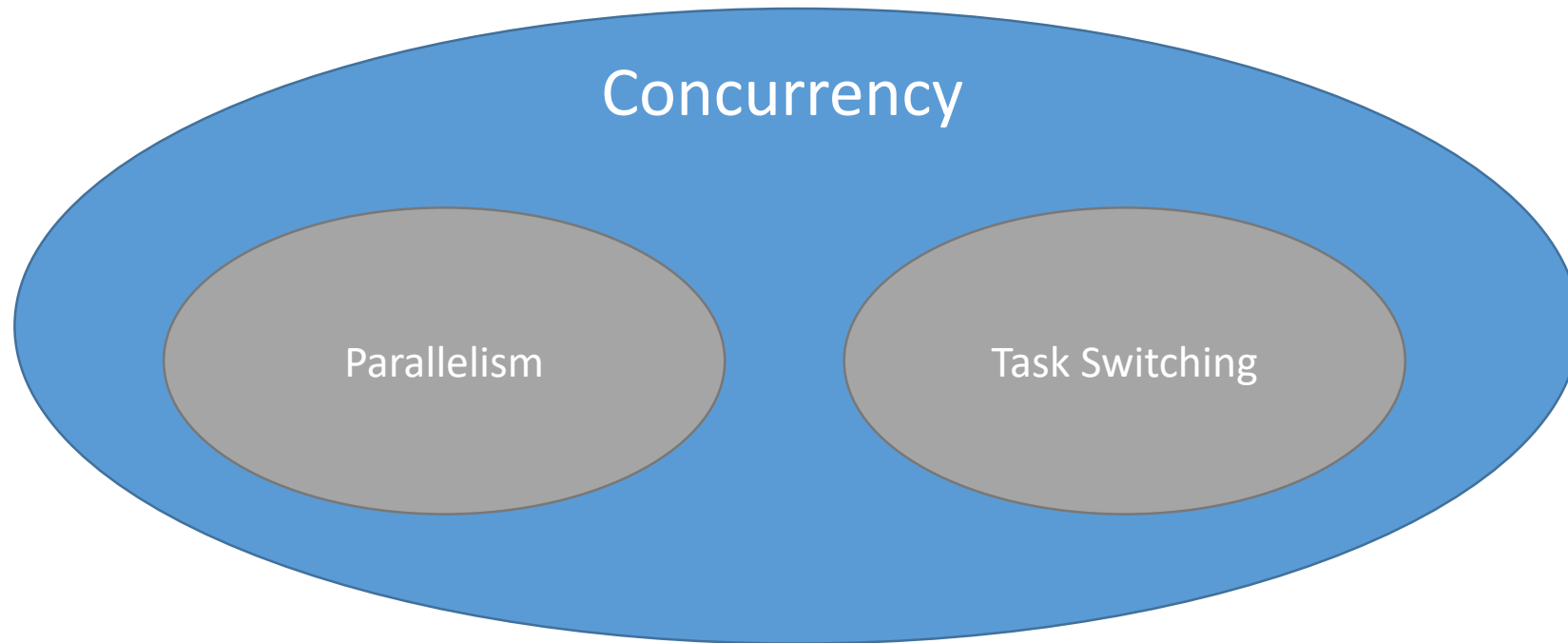
Concurrency

- **Multi-tasking** is the ability to do more than one thing at a time.
 - You may multi-task in your own everyday activities, though (depending on the difficulty of the task) humans are generally bad at it.
- When computers multi-task, they do it *concurrently*.

Concurrency

- **Concurrency** means that separate tasks in a computer program are simply *making progress* towards their completion at the same time.
- **Parallelism** means the tasks are executing *simultaneously*.
 - It's a type of concurrency.
- Another type of concurrency is **task-switching**, where the processor switches back and forth between making progress on multiple tasks.
 - *Appears* the tasks are running in parallel, but they are not.

Concurrency

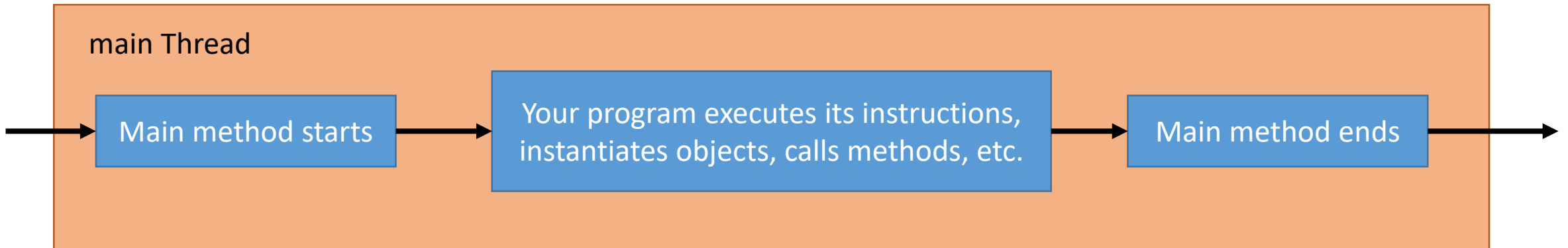


Concurrency

- **Concurrent programming** describes creating applications where different tasks in the program are completed at the same time.
 - Through either parallelism (simultaneous) or task-switching (back-and-forth) techniques.
- **Parallel programming** describes creating applications where tasks are programmed to be completed simultaneously.

Threads and Multithreading

- A **thread** (short for “*thread of execution*”) is a sequence of program instructions.
 - Our programs have all been single-threaded, in that they all execute their statements on Java’s “main” thread.



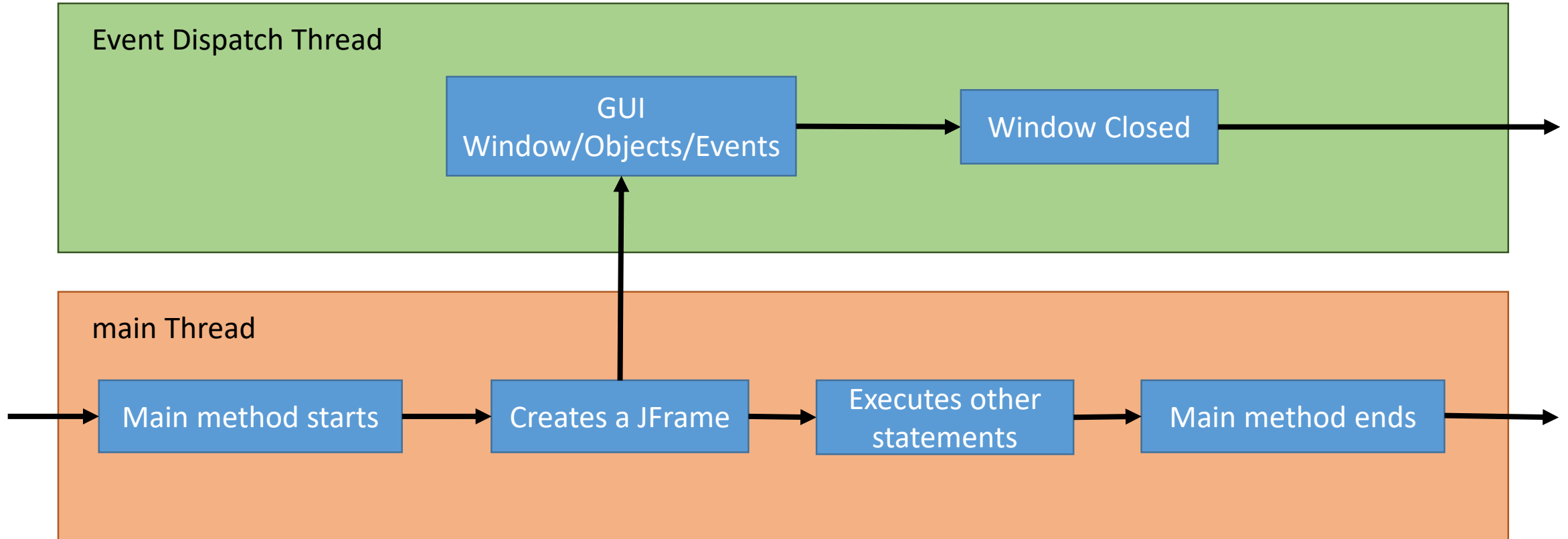
Threads and Multithreading

- This is not to say that the main method in a Java program is *the* main thread.
 - Methods and classes are not, by themselves, threads.
- The primary thread (from which other, additional threads are started from) is called “main”.

Threads and Multithreading

- GUI programs (specifically objects like JFrames) in Java execute on the *event dispatch thread*.
 - This is separate from the *main thread*.
- This is why GUI programs stay “alive” even after the main method has finished executing its statements.

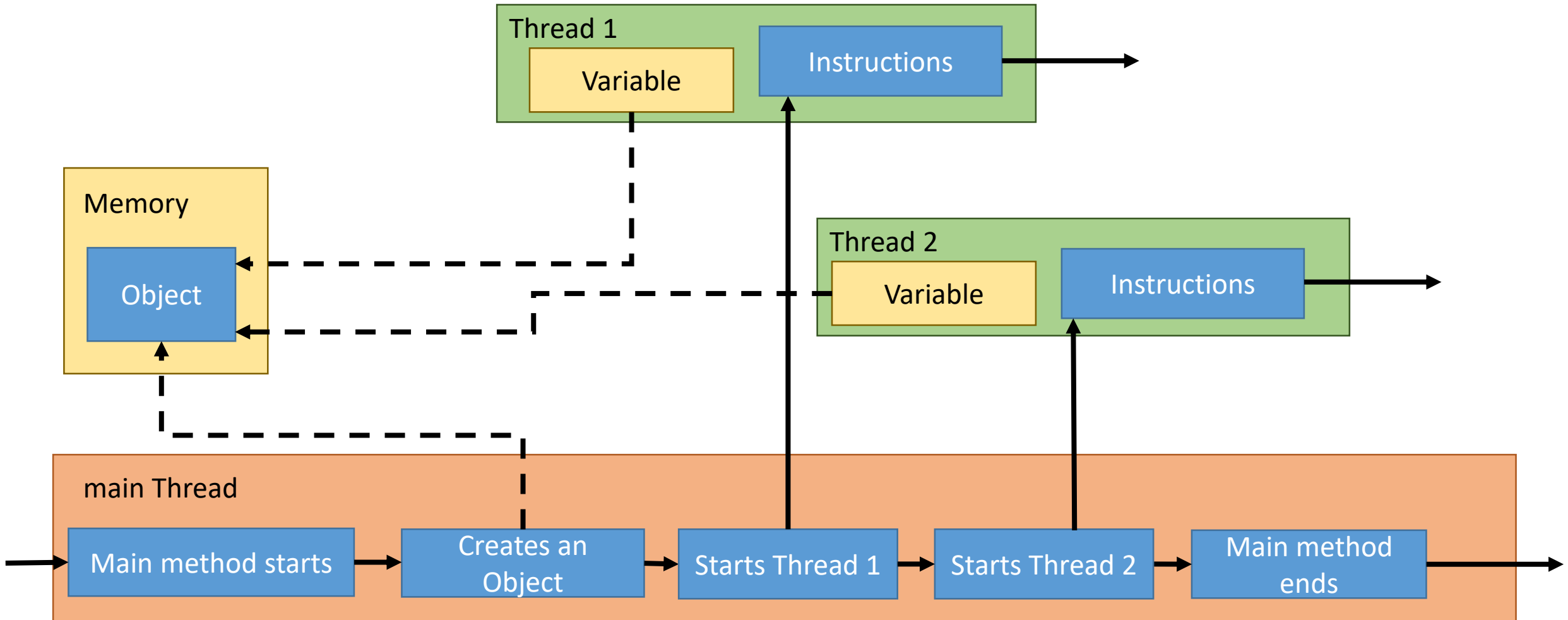
Threads and Multithreading



Threads and Multithreading

- A **multithreaded** program is one that completes its tasks using multiple threads of execution.
 - The threads are able to complete their tasks using some form of concurrency.
 - We'll be focused on parallelism.
- In multithreaded applications, the threads can share the same resources.
 - For example, two different threads can both have variables that reference the same object in memory.

Threads and Multithreading



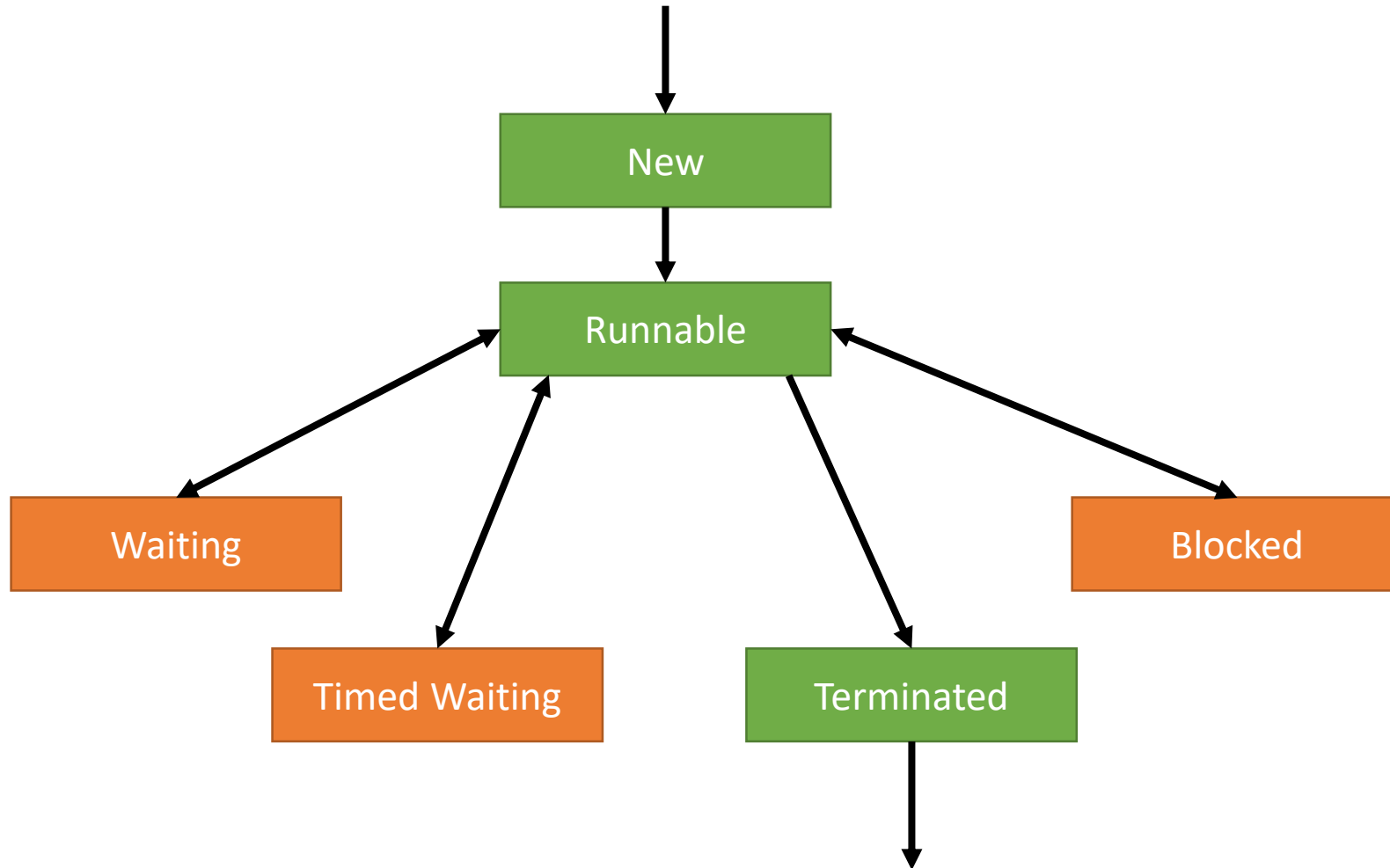
Threads and Multithreading

- Naturally, issues with concurrency can arise from this.
 - One thread could alter the state of an object while a different thread is in the middle of using it.
- Threads can coordinate their actions through **synchronization**, so that one thread doesn't step on the other one.
 - Allowing one thread to wait to use a resource until another is finished with it.

The Thread Life Cycle

- A thread can exist in various states between the time it is created and the time it finishes.
- We'll be working with examples that indirectly illustrate all of these states.

The Thread Life Cycle



The Thread Life Cycle

- A thread is in the **new** state when the thread has been created and is waiting to be started.
- When the thread is started, it enters the **runnable** state.
 - The thread is executing its instructions when in this state.
- When the thread has finished executing its instructions it enters the **terminated** or *“dead”* state.

The Thread Life Cycle

- A thread is in the **waiting** state when it is waiting for a different thread to finish (enter its terminated state) before continuing.
- A thread enters the **timed waiting** state when it is told to pause or “*sleep*” for some interval of time.
 - It’s not necessarily waiting for another thread to finish, it’s just waiting for some pre-determined amount of time to pass.
 - When the time is up, the thread resumes executing its instructions.

The Thread Life Cycle

- A thread enters the **blocked** state when it is unable to finish its tasks because a resource it needs is in use.
 - For example, another thread is using an object that it needs and can't proceed until that hold is released.
 - Isn't necessarily waiting for another thread to terminate/die (like the waiting state), only waiting for the other thread to give up the resource it needs.

The Thread Class

- One way to create a thread in Java is to have a class extend the Thread class.
 - Creating a subclass of Thread.
 - Does not need to be imported.

```
public class myThread extends Thread {  
  
}
```

The Thread Class

- The Thread class contains an abstract method called run that must be defined.
 - The statements in the run method are executed when the thread starts.

```
public class MyThread extends Thread {  
  
    public void run() {  
        //Code that executes when the thread is started  
    }  
  
}
```

The Thread Class

- Review the `MyThread` and `MyThreadDemo` classes in the Module 9 download.
 - In the `SampleCode_Threads` folder.
- The `MyThread` class (a `Thread` subclass) prints a countdown (pausing for a specified interval between outputs) when started.
- The `MyThreadDemo` class creates two `MyThread` objects and starts them both.
 - The threads execute their countdowns simultaneously.

The Thread Class

Threads started

Thread B finished
(Terminated state)

Thread A finished
(Terminated state)

main thread (and main method)
running and finishing independent
of Threads A and B

```
Run: MyThreadDemo (2) x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Main Method --- Starting threads.
Main Method --- Both threads have been started.
Pausing main Thread for 10 seconds --- main
THREAD A --- START
THREAD B --- START
Pausing THREAD B for 5 seconds.
Pausing THREAD A for 5 seconds.
THREAD B --- Countdown --- 10
THREAD A --- Countdown --- 10
THREAD B --- Countdown --- 9
THREAD A --- Countdown --- 9
THREAD B --- Countdown --- 8
THREAD B --- Countdown --- 7
THREAD A --- Countdown --- 8
THREAD B --- Countdown --- 6
Resuming main Thread --- main
MAIN METHOD FINISHED
THREAD B --- Countdown --- 5
THREAD A --- Countdown --- 7
THREAD B --- Countdown --- 4
THREAD B --- Countdown --- 3
THREAD A --- Countdown --- 6
THREAD B --- Countdown --- 2
THREAD B --- Countdown --- 1
THREAD A --- Countdown --- 5
THREAD B --- FINISHED
THREAD A --- Countdown --- 4
THREAD A --- Countdown --- 3
THREAD A --- Countdown --- 2
THREAD A --- Countdown --- 1
THREAD A --- FINISHED
Process finished with exit code 0
```


The Thread Class

- Classes that extend the Thread class are instantiated normally.
 - Putting the Thread in the **new** state.

```
Thread threadA = new MyThread("THREAD A", 2000);
```

- Since MyThread is a Thread subclass, the object can be referenced by a variable of the Thread type.
 - Recall polymorphism from earlier in the semester.
- The constructor can be designed to the programmer's specifications.
 - MyThread was designed to accept two arguments: A String (for the name of the thread) and an int (used by MyThread to control how long the thread waits before printing each number in the countdown)

The Thread Class

- Classes that extend the Thread class are first moved into the **runnable** state by calling its start method.
 - The start method (doesn't need to be overridden by the subclass) calls the run method.

`threadA.start();`

- The run method will execute its statement on its own thread of execution, separate from other threads like the main thread.

The Thread Class

```
public class MyThread extends Thread {
```

```
    private int interval;
```

← This is used in the run method to control how long to wait between printouts.

Sets the name of the thread
(Not required, but helpful)

```
    public MyThread(String nameIn, int intervalIn) {  
        super(nameIn);
```

```
        interval = intervalIn;
```

```
    }
```

```
}
```

The Thread Class

- To put a thread into the **timed waiting** state, use the following:

```
Thread.sleep(10000);
```

- When called, it causes the thread to wait the specified number of milliseconds.
 - The above example would cause its thread to wait 10000 milliseconds (10 seconds).
- After the time passes, the thread moved back to the runnable state and resumes executing its statements.

The Thread Class

- The sleep method has a checked InterruptedException.
- Be sure it is called within a try/catch.

The Thread Class

- To get a thread's name (the one provided when created), use the following:

```
Thread.currentThread().getName();
```

The Thread Class

- Review the classes in the `SampleCode_ThreadPracticalExample`
- Dialog boxes typically cause your program to wait until the dialog is closed.
- By putting the dialog in its own thread, it allows the main thread (or any other threads) to continue executing.
 - Multithreading is common in GUI applications to move the more intensive and time consuming processing to other threads in order to keep the GUI from “locking up” while it waits.

The Runnable Interface

- Java classes can only extend one class at a time.
- If a class already extends a class, it can't extend Thread.
- An alternative is to use the Runnable interface.
 - Classes can implement more than one interface.

The Runnable Interface

- Does not need to be imported.

```
public class myRunnable implements Runnable {  
  
}
```

The Runnable Interface

- You'll create a run method just as you did for a Thread subclass.

```
public class myRunnable implements Runnable {  
  
    public void run() {  
        //Code that executes when the thread is started  
    }  
  
}
```

The Runnable Interface

- Review the MyRunnable and MyRunnableDemo classes in the Module 9 download.
 - In the SampleCode_Runnables folder.
- The functionality is the same as MyThread/MyThreadDemo

The Runnable Interface

```
Run: MyRunnableDemo x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Main Method --- Starting threads.
Main Method --- Both threads have been started.
Pausing main Thread for 10 seconds --- main
RUNNABLE B --- START
Pausing RUNNABLE B for 5 seconds.
RUNNABLE A --- START
Pausing RUNNABLE A for 5 seconds.
RUNNABLE B --- Countdown --- 10
RUNNABLE A --- Countdown --- 10
RUNNABLE B --- Countdown --- 9
RUNNABLE A --- Countdown --- 9
RUNNABLE B --- Countdown --- 8
RUNNABLE B --- Countdown --- 7
RUNNABLE A --- Countdown --- 8
RUNNABLE B --- Countdown --- 6
Resuming main Thread --- main
MAIN METHOD FINISHED
RUNNABLE B --- Countdown --- 5
RUNNABLE A --- Countdown --- 7
RUNNABLE B --- Countdown --- 4
RUNNABLE B --- Countdown --- 3
RUNNABLE A --- Countdown --- 6
RUNNABLE B --- Countdown --- 2
RUNNABLE B --- Countdown --- 1
RUNNABLE A --- Countdown --- 5
RUNNABLE B --- FINISHED
RUNNABLE A --- Countdown --- 4
RUNNABLE A --- Countdown --- 3
RUNNABLE A --- Countdown --- 2
RUNNABLE A --- Countdown --- 1
RUNNABLE A --- FINISHED
Process finished with exit code 0
```

Threads started

Thread B finished

Thread A finished

main thread (and main method) running and finishing independent of Threads A and B

The Runnable Interface

- Classes that implement the Runnable Interface are instantiated as follows:

```
Thread threadA = new Thread(new MyRunnable(2000), "RUNNABLE A");
```

- Instantiated using the Thread class.
 - First argument is an instance of the class that implements Runnable
 - Second argument is the Thread's name.
- The thread is still started by calling its start method.

Synchronization

- Frequently, multiple different threads will need to use the same objects or resources.
- To make sure the threads don't step on each others work, their use of these resources can be synchronized.
- Synchronization, in this case, doesn't mean they use the object or resource in-tandem.
 - It means one thread will be blocked until the other is finished using the object.

Synchronization

- A synchronized statement will allow a thread to have an exclusive “lock” on an object.
- The syntax for a synchronized statement is shown below:

```
synchronized(object) {  
    //Code that executes with a lock on this object  
}
```

Synchronization

- Review the CountdownPrinter, MyThread, MyThreadDemo classes in the Module 9 download.
 - In the SampleCode_Synchronization folder.
- The functionality is similar to MyThread/MyThreadDemo

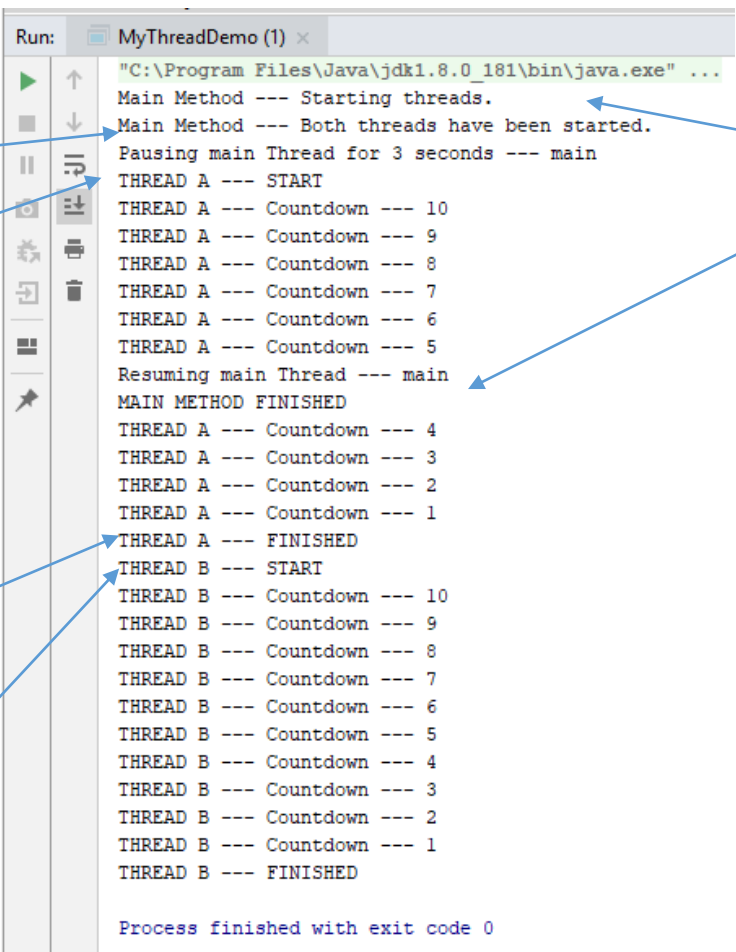
Synchronization

- CountdownPrinter
 - This class has a “print” method that prints a countdown.
 - It’s not threaded, but will cause the thread that uses this object to pause/sleep between printouts.
- MyThread
 - Has a field/reference to a CountdownPrinter object
 - It’s run method holds a lock on the object with a synchronized statement until it is finished printing its output.

Synchronization

- MyThreadDemo
 - Creates an instance of a CountdownPrinter object
 - Instantiates two MyThread objects, providing each with a reference to this object.
 - The second thread will be blocked until the lock on the CountdownPrinter object is released by the first thread.

Synchronization



Threads started

Thread A running.
Thread B is blocked until
it is finished with the
CountdownPrinter.

Thread A finished (no longer
using the CountdownPrinter)

Thread B no longer blocked

main thread (and main method)
running and finishing independent
of Threads A and B

```
Run: MyThreadDemo (1) x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Main Method --- Starting threads.
Main Method --- Both threads have been started.
Pausing main Thread for 3 seconds --- main
THREAD A --- START
THREAD A --- Countdown --- 10
THREAD A --- Countdown --- 9
THREAD A --- Countdown --- 8
THREAD A --- Countdown --- 7
THREAD A --- Countdown --- 6
THREAD A --- Countdown --- 5
Resuming main Thread --- main
MAIN METHOD FINISHED
THREAD A --- Countdown --- 4
THREAD A --- Countdown --- 3
THREAD A --- Countdown --- 2
THREAD A --- Countdown --- 1
THREAD A --- FINISHED
THREAD B --- START
THREAD B --- Countdown --- 10
THREAD B --- Countdown --- 9
THREAD B --- Countdown --- 8
THREAD B --- Countdown --- 7
THREAD B --- Countdown --- 6
THREAD B --- Countdown --- 5
THREAD B --- Countdown --- 4
THREAD B --- Countdown --- 3
THREAD B --- Countdown --- 2
THREAD B --- Countdown --- 1
THREAD B --- FINISHED

Process finished with exit code 0
```

Joining Threads

- Synchronization is a way to block a thread before continuing.
 - Blocked state in the life cycle
- A way to make a thread wait until another thread is finished is to join the thread with the other.
 - Waiting state in the life cycle.

Joining Threads

- To join a first thread with second, the join method is called on the second thread from within the first thread.

`threadA.join();`

- The first thread will wait to continue executing its instructions until the second is finished/in the terminated state.
 - In this case, whichever thread called the above statement would then wait for threadA to finish before continuing.

Joining Threads

- The join method has a checked InterruptedException.
- Be sure it is called within a try/catch.

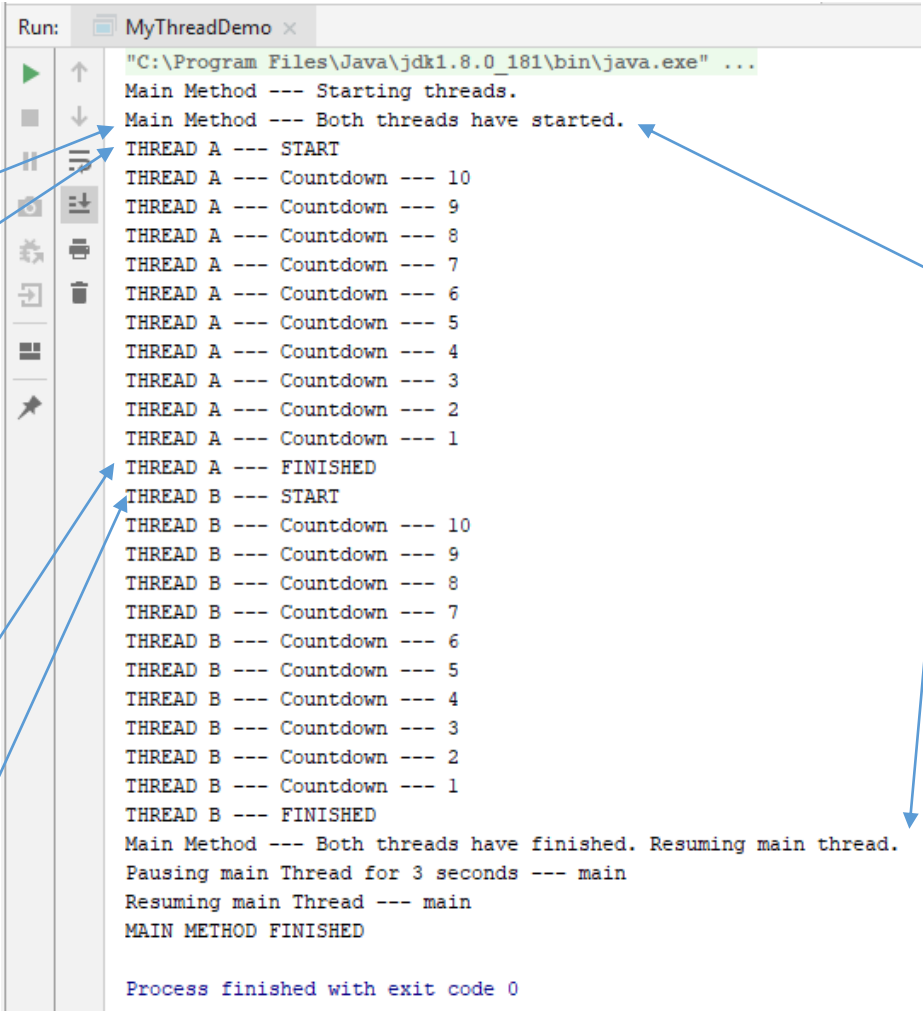
Joining Threads

- Review the CountdownPrinter, MyThread, MyThreadDemo classes in the Module 9 download.
 - In the SampleCode_JoiningThreads folder.
- The functionality is nearly identical to the examples in the SampleCode_Synchronization folder.

Joining Threads

- MyThreadDemo
 - Creates an instance of a CountdownPrinter object
 - Instantiates two MyThread objects, providing each with a reference to this object.
 - The second thread will be blocked until the lock on the CountdownPrinter object is released by the first thread.
- The join method is called on both.
 - The main thread will wait for threadA to finish. It then joins threadB and waits for it to finish.
 - When threadB finished, the main thread will finish executing its remaining statements.

Joining Threads



The screenshot shows a Java IDE window titled "MyThreadDemo" with a console output. The output logs the execution of a main method and two sub-threads, A and B, each performing a 10-second countdown. Annotations with blue arrows point to specific lines in the output:

- Threads started**: Points to the line "Main Method --- Both threads have started."
- Thread A running. Thread B is blocked until it is finished with the CountdownPrinter.**: Points to the first "THREAD A --- Countdown" line.
- Thread A finished (no longer using the CountdownPrinter)**: Points to the "THREAD A --- FINISHED" line.
- Thread B no longer blocked**: Points to the first "THREAD B --- Countdown" line.
- main thread (and main method) WAITS until Threads A and B are finished before executing the remaining code in the main method.**: Points to the line "Main Method --- Both threads have finished. Resuming main thread."

```
Run: MyThreadDemo x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Main Method --- Starting threads.
Main Method --- Both threads have started.
THREAD A --- START
THREAD A --- Countdown --- 10
THREAD A --- Countdown --- 9
THREAD A --- Countdown --- 8
THREAD A --- Countdown --- 7
THREAD A --- Countdown --- 6
THREAD A --- Countdown --- 5
THREAD A --- Countdown --- 4
THREAD A --- Countdown --- 3
THREAD A --- Countdown --- 2
THREAD A --- Countdown --- 1
THREAD A --- FINISHED
THREAD B --- START
THREAD B --- Countdown --- 10
THREAD B --- Countdown --- 9
THREAD B --- Countdown --- 8
THREAD B --- Countdown --- 7
THREAD B --- Countdown --- 6
THREAD B --- Countdown --- 5
THREAD B --- Countdown --- 4
THREAD B --- Countdown --- 3
THREAD B --- Countdown --- 2
THREAD B --- Countdown --- 1
THREAD B --- FINISHED
Main Method --- Both threads have finished. Resuming main thread.
Pausing main Thread for 3 seconds --- main
Resuming main Thread --- main
MAIN METHOD FINISHED

Process finished with exit code 0
```

Common Issues with Concurrency

- Aside from threads possibly changing the data of objects in the middle of use by other threads, another issue can arise from concurrent programming.
- **Indefinite postponement** is when threads never get started or get stuck.

Common Issues with Concurrency

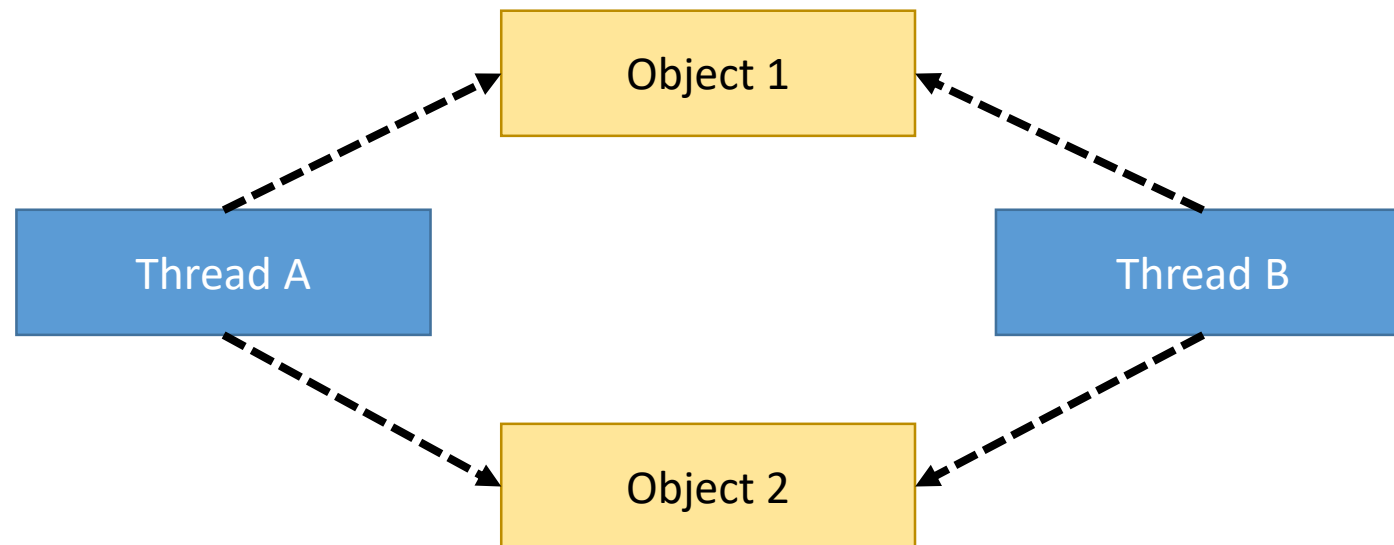
- One example of indefinite postponement is called **starvation**.
- The Operating System handles how threads are scheduled for execution by the processor.
- It prioritizes threads so that high-priority threads are executed first.
 - If a thread has too low of a priority, it possible it will never make it to the processor.

Common Issues with Concurrency

- Operating Systems use a technique called *aging* to avoid this from happening.
- It gradually increases the priority of threads so that, eventually, every thread is executed.
- This is a bigger issue in operating systems programming than it is a concern of an application developer.
 - If the OS is bad at scheduling jobs, there's not much an application developer can do about it.

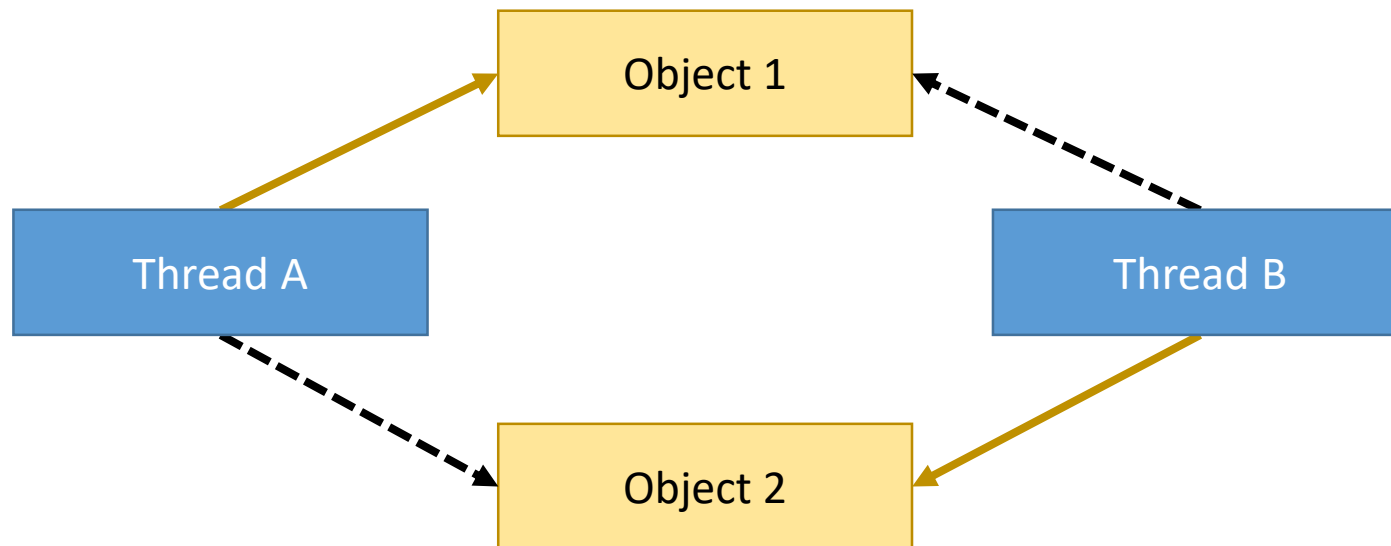
Common Issues with Concurrency

- The more common issue application developers face is situations like this:
 - Imagine two threads that both need exclusive use of two objects.
 - (You can probably already guess where I'm going with this)



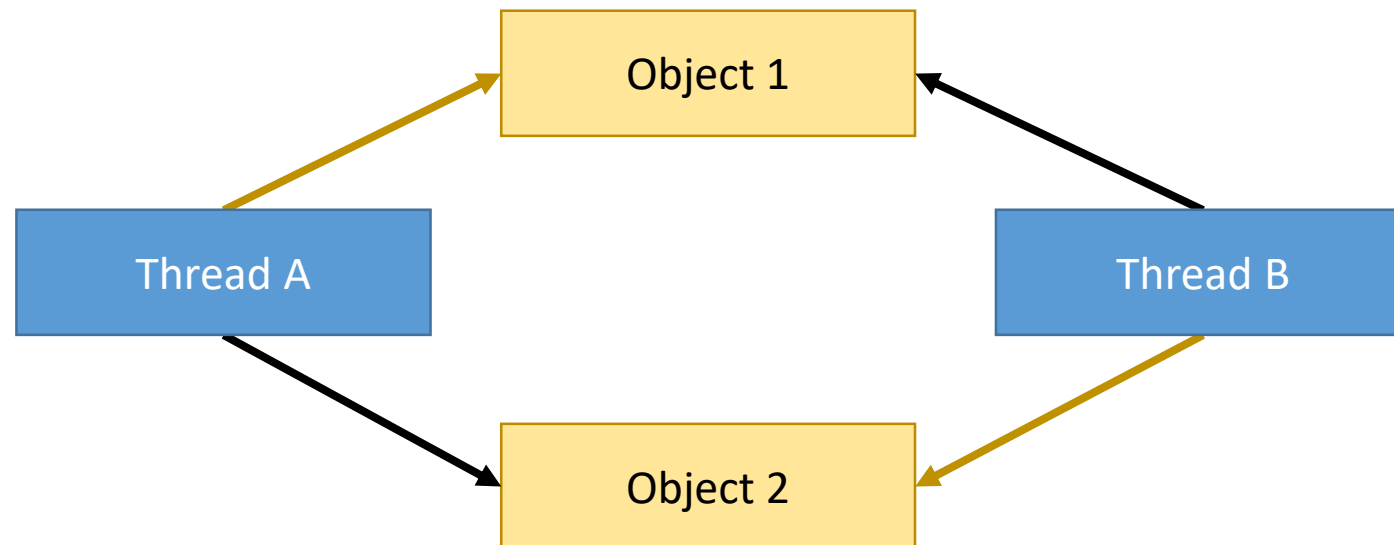
Common Issues with Concurrency

- Thread A synchronizes with Object 1
- Thread B synchronizes with Object 2



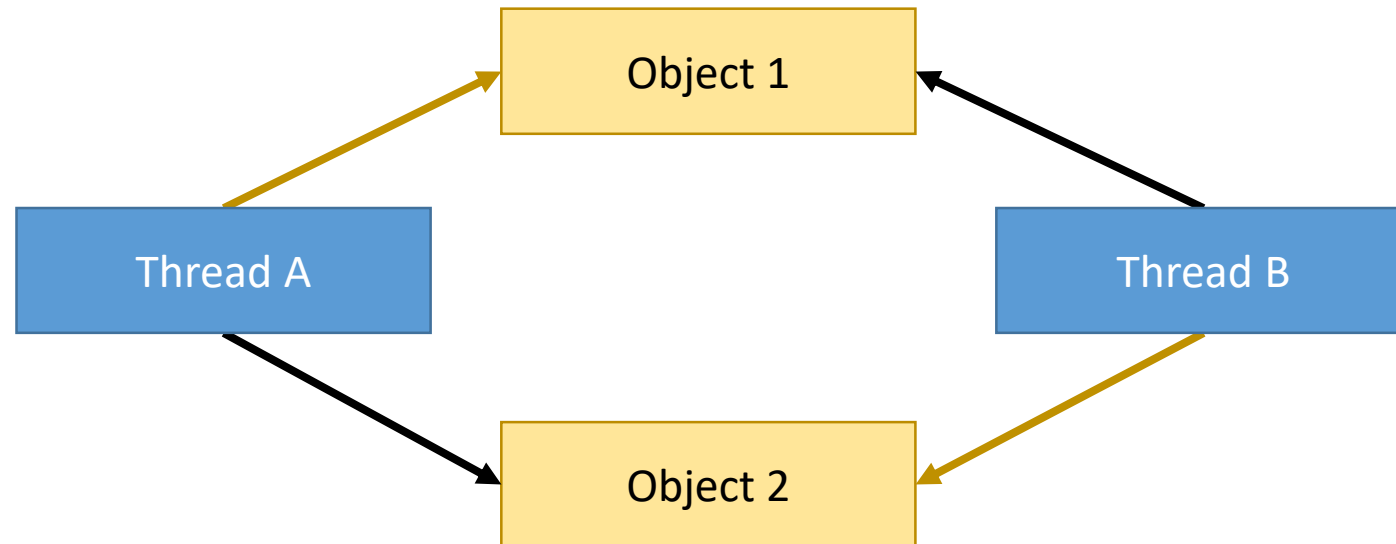
Common Issues with Concurrency

- Thread A needs Object 2 in order to finish.
 - It can't have it because Thread B still has a lock on it.
- Thread B needs Object 1 in order to finish.
 - It can't have it because Thread A still has a lock on it.



Common Issues with Concurrency

- The threads are stuck in a standoff.
 - There's no way either can finish without the other resource being released.



Common Issues with Concurrency

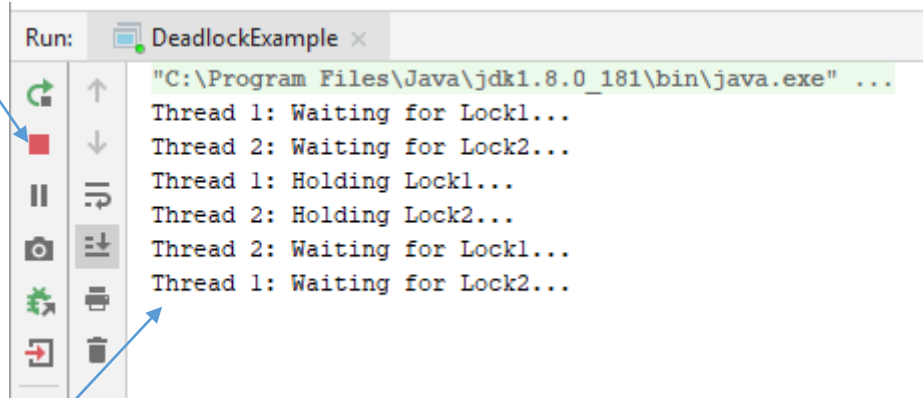
- This scenario is called **deadlock**.
- Only way to prevent it is to make sure you program your threads so that they never arrive at a situation like this.
 - It's like how programmers need to be careful that they don't accidentally create an infinite loop.

Common Issues with Concurrency

- Review the DeadlockExample and DeadlockSolution classes in the Module 9 download.
 - In the SampleCode_Deadlock folder.
- DeadlockExample
 - Purposely creates a deadlock situation like the one described.
 - End the program by clicking the red square icon.
- DeadlockSolution
 - Same program, but avoids deadlock by having both threads use the objects in the same order.

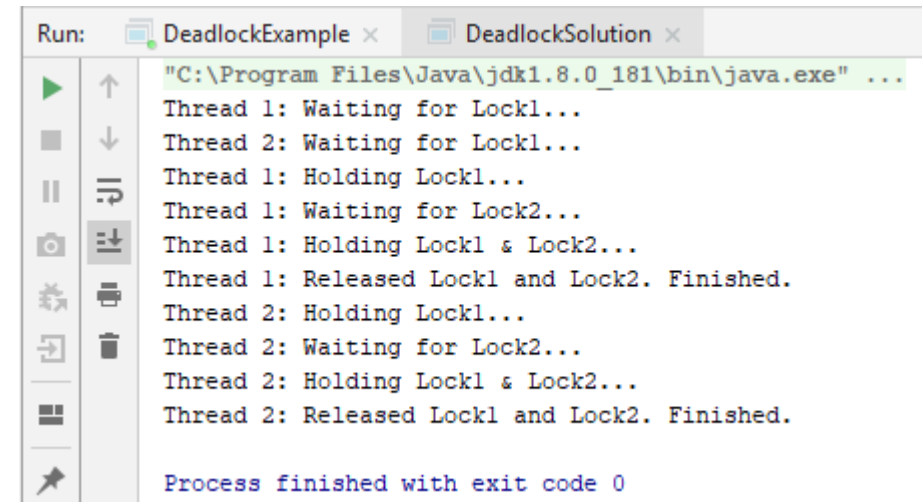
Common Issues with Concurrency

Click to end



```
Run: DeadlockExample x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Thread 1: Waiting for Lock1...
Thread 2: Waiting for Lock2...
Thread 1: Holding Lock1...
Thread 2: Holding Lock2...
Thread 2: Waiting for Lock1...
Thread 1: Waiting for Lock2...
```

Will never get past here



```
Run: DeadlockExample x DeadlockSolution x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Thread 1: Waiting for Lock1...
Thread 2: Waiting for Lock1...
Thread 1: Holding Lock1...
Thread 1: Waiting for Lock2...
Thread 1: Holding Lock1 & Lock2...
Thread 1: Released Lock1 and Lock2. Finished.
Thread 2: Holding Lock1...
Thread 2: Waiting for Lock2...
Thread 2: Holding Lock1 & Lock2...
Thread 2: Released Lock1 and Lock2. Finished.
Process finished with exit code 0
```