CSE 1325 - Object-Oriented Programming

Multithreading in Java

Alex Dillhoff

University of Texas at Arlington

Multithreaded Applications

An OS implements multi-tasking by spawning multiple **processes**.

At a lower level, each **process** can implement multi-tasking by spawning multiple **threads**.

Multithreaded Applications

Multithreading is especially critical in GUI applications.

The rendering thread must not waste any time performing data operations that require an excessive amount of computation.

Polling

In a single-threaded application, a loop runs continuously and checks an event queue for the next action to take.

This pattern is referred to as **polling**.

Polling

The downside to polling is that the thread will waste CPU time waiting for the next action.

This has the effect of making the application look unresponsive.

Polling

Java provides built-in support for multithreading through several convenient API calls.

Multithreading allows developers to perform multiple tasks without interrupting the main application.

Multithreaded Applications

In a single-core system, multiple threads are not able to execute truly simultaneously.

Instead, the CPU switches between threads to simulate it.

Multithreaded Applications

Even on a single-core system, multithreading is still more desirable than a single thread.

Instead of waiting for a task to complete, the CPU can simply switch to another thread that is ready to work.

Multithreaded Applications

In multi-core systems, two threads may truly execute simultaneously.

At the software level, we are not concerned with how the underlying architecture handles multiple threads.

We only need to utilize the appropriate API calls.

Synchronization

An important consideration when it comes to multithreading is that of **synchronization**.

When multiple threads work with a shared data object, it is important that one thread does not interfere with the work of another.

Synchronization

For example, imagine two threads working with a single file.

If both threads attempt to write data to the same file at the same time, the data could become corrupted.

Synchronization

This is typically handled through the use of **synchronization**.

Java implements this through the use of what is called a **monitor**.

Synchronization

When a thread accesses a **synchronized** object, it enters a monitor that can only contain a single thread.

Any other thread that needs access to the object must wait until the monitor is available.

This occurs when the original thread releases a **lock** and exits the monitor.

Threads in Java

Multithreading is implemented in Java mainly through the `Thread` class.

Java applications that need multithreading will either subclass `Thread` or `implement Runnable`.

Threads in Java

Every Java program begins by executing on the main thread.

From here, other threads can be created and managed as necessary.

Threads in Java

As a general rule of thumb, the main thread should always be the last thread to finish executing code.

There are several methods available to make sure this is always the case.

Threads in Java

In the next example, we will look at how to gain access to the main thread and call `Thread` methods on it.

Example: `MainThreadExample`

Creating a New Thread

There are two primary approaches to creating a new thread.

1. Implement the `Runnable` interface.
2. Extend the `Thread` class.

Implementing `Runnable`

When implementing `Runnable`, you only need to override the `run` method.

This object can then be used as an argument to the `Thread` constructor.

Implementing Runnable

Example: `NewThreadExample`

Extending `Thread`

Extending the `Thread` class allows full customization over what happens in the thread.

Other threads can also utilize a form of communication, as we will see in the next example.

Extending Thread

Example: `CustomThreadExample`

Extending `Thread`

Which approach is best?

A good rule of thumb is to only subclass another class if you need fine-grained control over other methods.

If you only need to implement the `run` method, stick to implementing `Runnable`.

Using `Thread` Methods

In the previous examples, we used timing to make sure the main thread will finish last.

This can be achieved regardless of fixed timing through the use of some useful methods, such as `join` and `isAlive`.

Using `Thread` Methods

Example: `MultiThreadsExample`

Setting Thread Priorities

All threads are managed by a thread scheduler. An OS will typically decide which threads to execute before others.

Thread priority can be set in Java through the use of `setPriority(int level)`, where the `level` can be any value in the range `[MIN_PRIORITY, MAX_PRIORITY]`.

Setting Thread Priorities

The default values for the range are $[1, 10]$.

In practice, it is best to keep the threads at their default priority (5).

Synchronization

Threading obviates polling. This means less wasted CPU time.

However, even multithreaded applications may introduce wasted CPU time.

Synchronization

For example, a producer may have to wait for a consumer before fetching more data.

This can be streamlined through the use of the methods `wait`, `notify`, and `notifyAll`.

Synchronization

These methods may only be used within a **synchronized** context.

The documentation for `wait` discusses the possibility of a *spurious wakeup*.

The solution is to wrap the calls to `wait` inside of a loop, as we will see in the next example.

Synchronization

Example: `DataLoaderExample`

Synchronization

In the previous example, the client fetches the same data repeatedly while the data loader attempts to update.

The intended use is for the client to wait for new data. We will see how to fix this in the next example.

Synchronization

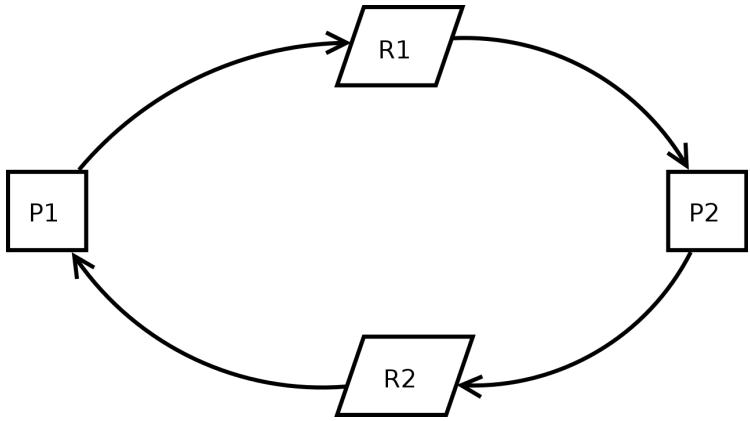
Example: `DataLoaderFixedExample`

Deadlocks

Sometimes, multiple threads may have a circular dependency on an object or set of objects.

This can lead to an error called a **deadlock**.

Deadlocks



From wikipedia: P1 has R2 and needs R1. P2 has R1 and needs R2.

Deadlocks

Example: `DeadlockExample`

Deadlocks

Deadlocks are resolved in various ways including re-ordering the threads or using calls to `join`.

A deeper analysis into deadlock solutions is beyond the scope of this course.

SwingWorker

`Thread` and `Runnable` were introduced very early on in the Java API.

Since the introduction of `Swing`, a worker class for multithreading was also introduced to accommodate background tasks in GUI applications.

SwingWorker

SwingWorker involves 3 different threads: the *current* thread, a *worker* thread, and the *Event Dispatch* thread.

Executing a SwingWorker object will create the thread immediately and return the control to the event dispatch thread.

SwingWorker

`SwingWorker` has the benefit of being able to work in the background and update the GUI once it is finished.

In the next example, we will implement background file saving using a `Runnable` object.

SwingWorker

Example: `FileOpThreadExample`

SwingWorker

To compare using a `Runnable` object to implement the background process, we will look at one last example that uses `SwingWorker` instead.

Example: `FileOpSwingWorkerExample`