Cameron Spry
A.J. Piergiovanni

Threads allow an application to do multiple things at the same time by taking advantage of modern multi-threaded processors. Threads allow your program to process data and update the screen at the same time so the user never has to wait for something to complete in order to interact with your program. Every application has at least one thread, the main thread that you program runs in. Java has a Thread class and there are two ways to use Thread objects in your program. You can simply instantiate a Thread object each time you need one or you can pass the tasks to an executor.

Creating a thread requires that you implement the Runnable interface which has one method, run() which must be implemented. This Runnable Object is passed to the Thread's constructor which will call the run function in a new Thread. You can also subclass Thread, because Thread implements Runnable, though Thread's run method does nothing.

Once a Thread is created, you may want to pause the thread. This is done by using the Thread.sleep() method. Sleep takes an integer parameter for how many milliseconds the Thread should pause for. Threads also have a join method. This method tells the thread that calls it to wait until the thread that is being "joined" is finished.

Synchronization is important when using threads. When using threads to modify fields of an object, unexpected results (race conditions) can occur. If two threads modify a field at the same time, then one of the values will be overwritten. Synchronization prevents this. When you add synchronized to a method signature, it makes the method so that when two threads are trying to use that method, one waits until the first one is finished. This keeps the object from being overwritten and the value of the object synchronized.

One problem with threading is Deadlock. Deadlock occurs when at least two threads are blocked forever waiting for the other one. For example, say you have a Fighter class who has a punch method and this punch method moves the hand to the other Fighter and then waits until the other Fighter punches to remove his fist. If both fighters are told to punch at the same time in different threads, both will move their fists to the other and wait to be punched to remove their fists. However, they will never be punched by the other, because the other already punched. This is deadlock, both threads are waiting for the other thread to do something which will never happen, and so nothing will ever happen.

Another problem is Starvation. When two threads rely on the same synchronized method, and that method takes time to complete, and one thread calls it frequently, no other threads will get enough access to it and won't be able to complete their jobs. Livelock occurs when a thread is responding to another thread, which is responding to another thread. Each thread is busy responding to another thread and making no progress.

One important usage of threads is to keep a user interface responsive while a program performs complex or lengthy tasks. This is normally simple, but there is one important thing to note: Swing is not thread-safe, as it relies on a single-threaded event handler. To work around this, there is a class called SwingUtilities that contains a method called invokeLater(Runnable t), that should be called by a thread when it wants to update the GUI. This runs the Runnable on the event handler thread, synchronizing it with the rest of the events. If this is not done, conflicts can occur.

Creating and starting a thread:

```
Thread t = new Thread(new RunnableClass());
t.start();
```