The C++ Timer and TimerTask Classes

The `Timer` class allows you to schedule some code for once-only or repeated execution after some time interval elapses. The code to be executed resides in a class you derive from `TimerTask`:

C++

```
class Timer;
typedef IceUtil::Handle<Timer> TimerPtr;

class TimerTask : virtual public IceUtil::Shared {
public:
    virtual ~TimerTask() { }
    virtual void runTimerTask() = 0;
};

typedef IceUtil::Handle<TimerTask> TimerTaskPtr;
```

Your derived class must override the `runTimerTask` member function; the code in this method is executed by the timer. If the code you want to run requires access to some program state, you can pass that state into the constructor of your class or, alternatively, set that state via member functions of your class before scheduling it with a timer.

The `Timer` class invokes the `runTimerTask` member function to run your code. The class has the following definition:

C++

```
class Timer : /* ... */ {
public:
    Timer();
    Timer(int priority);

    void schedule(const TimerTaskPtr& task, const IceUtil::Time& interval);

    void scheduleRepeated(const TimerTaskPtr& task, const IceUtil::Time& interval);

    bool cancel(const TimerTaskPtr& task);

    void destroy();
};

typedef IceUtil::Handle<Timer> TimerPtr;
```

Intervals are specified using `Time` objects.

The constructor is overloaded to allow you specify a thread priority. The priority controls the priority of the thread that executes your task.

The `schedule` member function schedules an instance of your timer task for once-only execution after the specified time interval has elapsed. Your code is executed by a separate thread that is created by the `Timer` class. The function throws an `IllegalArgumentException` if you invoke it on a destroyed timer.

The `scheduleRepeated` member function runs your task repeatedly, at the specified time interval. Your code is executed by a separate thread that is created by the `Timer` class; the same thread is used every time your code runs. The function throws an `IllegalArgumentException` if you invoke it on a destroyed timer.

If your code throws an exception, the `Timer` class ignores the exception, that is, for a task that is scheduled to run repeatedly, an exception in the current execution does not cancel the next execution.

If your code takes longer to execute than the time interval you have specified for repeated execution, the second execution is delayed accordingly. For example, if you ask for repeated execution once every five seconds, and your code takes ten seconds to complete, then the second execution of your task starts five seconds after the previous execution finishes, that is, the interval specifies the wait time between successive executions.

A `TimerTask` instance that has already been scheduled with a `Timer` instance cannot be scheduled again with the same `Timer` instance until the task has completed or been canceled.

For a single `Timer` instance, the execution of all registered tasks is serialized. The wait interval applies on a per-task basis so, if you schedule task A at an interval of five seconds, and task B at an interval of ten seconds, successive runs of task A start no sooner than five seconds after the previous task A has finished, and successive runs of task B start no sooner than ten seconds after the previous task B has finished. If, at the time a task is scheduled to run, another task is still running, the new task's execution is delayed until the previous task has finished.

If you want scheduled tasks to run concurrently, you can create several `Timer` instances; tasks then execute in as many threads concurrently as there are `Timer` instances.

The `cancel` member function removes a task from a timer's schedule. In other words, it stops a task that is scheduled from being executed. If you cancel a task while it is executing, `cancel` returns immediately and the currently running task is allowed to complete normally; that is, `cancel` does not wait for any currently running task to complete.

The return value is true if `cancel` removed the task from the schedule. This is the case if you invoke `cancel` on a task that is scheduled for repeated execution and this was the first time you cancelled that task; subsequent calls to `cancel` return false. Calling `cancel` on a task scheduled for once-only execution always returns false, as does calling `cancel` on a destroyed timer.

The `destroy` member function removes all tasks from the timer's schedule. If you call `destroy` from any thread other than the timer's own execution thread, it joins with the currently executing task (if any), so the function does not return until the current task has completed. If you call `destroy` from the timer's own execution thread, it instead detaches the timer's execution thread. Calling `destroy` a second time on the same `Timer` instance has no effect. Similarly, calling `cancel` on a destroyed timer has no effect.

Note that you must call `destroy` on a `Timer` instance before allowing it to go out of scope; failing to do so causes undefined behavior.

Calls to `schedule` or `scheduleRepeated` on a destroyed timer raises an `IceUtil::IllegalArgumentException`.

See Also

- The C++ Time Class
- The C++ Thread Classes