Multithreading
BackgroundWorker

BackgroundWorker is a helper class in the `System.ComponentModel` namespace for managing a worker thread.

- A "cancel" flag for signaling a worker to end without using Abort
- A standard protocol for reporting progress, completion and cancellation
- An implementation of `IComponent` allowing it be sited in the Visual Studio Designer
- Exception handling on the worker thread
- The ability to update Windows Forms controls in response to worker progress or completion.
**Control.Invoke**

- In a multi-threaded Windows Forms application, it's illegal to call a method or property on a control from any thread other than the one that created it.

- All cross-thread calls must be explicitly marshalled to the thread that created the control (usually the main thread), using the Control.Invoke or Control.BeginInvoke method.

- The BackgroundWorker class wraps worker threads that need to report progress and completion, and automatically calls Control.Invoke as required.
The BackgroundWorker class uses the thread-pool, which recycles threads to avoid recreating them for each new task.

This means one should never call Abort on a BackgroundWorker thread.
Using the BackgroundWorker

- Instantiate BackgroundWorker, and handle the **DoWork** event.
- Call **RunWorkerAsync**, optionally with an object argument.
- Handle the **reporting events** for progress, completion and cancellation.
Any argument passed to RunWorkerAsync will be forwarded to DoWork's event handler, via the event argument's **Argument property**.

```csharp
private void bw1_DoWork(object sender, DoWorkEventArgs e)
```

*e. Argument is of type System Object.*

Therefore one can pass any number of arguments packed in a class, structure or array.
RunWorkerCompleted

- The **RunWorkerCompleted** event fires after the DoWork event handler has done its job.

- Handling RunWorkerCompleted is not mandatory, but one usually does so in order to query any exception that was thrown in DoWork.

- Code within a RunWorkerCompleted event handler is able to update Windows Forms controls without explicit marshalling; code within the DoWork event handler cannot.
Progress Reporting

1. Set the WorkerReportsProgress **property** to true.
2. Periodically call **ReportProgress** from within the DoWork event handler with a "percentage complete" value, and optionally, a user-state object.
3. Handle the **ProgressChanged event**, querying its event argument's ProgressPercentage property.

Code in the ProgressChanged event handler is free to **interact with UI** controls just as with RunWorkerCompleted. This is typically where you will update a progress bar.
Cancellation Support

- Set the `WorkerSupportsCancellation property` to true.
- Periodically check the `CancellationPending` property from within the DoWork event handler – if true, set the event argument's `Cancel property` true, and return. (The worker can set Cancel true and exit without prompting via CancellationPending – if it decides the job's too difficult and it can't go on).
- Check `e.Cancelled` in the RunWorkerCompleted event handler.
- Call `CancelAsync` to request cancellation.
Subclassing BackgroundWorker

- BackgroundWorker is not sealed!
- It provides a virtual OnDoWork method.
- When writing a potentially long-running method, one can write a version returning a subclassed BackgroundWorker.
- Pre-configured to perform the job asynchronously.
- The consumer then only needs to handle the RunWorkerCompleted and ProgressChanged events.
PRACTICAL EXAMPLES ...
QUESTIONS?
Resources

- Course Homepage: se.inf.ethz.ch/teaching/ss2007/251-0290-00

- Exercise Material: www.inf.ethz.ch/personal/thomas.fuchs
public class Client {
    Dictionary<string, int> GetFinancialTotals(int foo, int bar) {
        ...
    }
    ...
}

public class Client {
    public FinancialWorker GetFinancialTotalsBackground(int foo, int bar) {
        return new FinancialWorker(foo, bar);
    }
}

public class FinancialWorker : BackgroundWorker {
    public Dictionary<string, int> Result; // We can add typed fields.
    public volatile int Foo, Bar; // We could even expose them
    // via properties with locks!
    public FinancialWorker() {
        WorkerReportsProgress = true;
        WorkerSupportsCancellation = true;
    }

    public FinancialWorker (int foo, int bar) : this() {
        this.Foo = foo; this.Bar = bar;
    }
}
protected override void OnDoWork (DoWorkEventArgs e) {
    ReportProgress (0, "Working hard on this report..." );
    // Initialize financial report data

    while (!finished report ) {
        if (CancellationPending) {
            e.Cancel = true;
            return;
        }
        Perform another calculation step
        ReportProgress (percentCompleteCalc, "Getting there..." );
    }
    ReportProgress (100, "Done!");
    e.Result = Result = completed report data;
}
Whoever calls `GetFinancialTotalsBackground` then gets a `FinancialWorker` – a wrapper to manage the background operation with real-world usability. It can report progress, be cancelled, and is compatible with Windows Forms without `Control.Invoke`. It's also exception-handled, and uses a standard protocol (in common with that of anyone else using `BackgroundWorker`!)