

Eclipse Concurrency in Action

John Arthorne and Michael Valenta
IBM Rational Software
Eclipse Platform Committers

What are we going to cover?

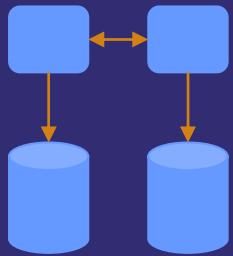
- 1) Principles of concurrency
- 2) Concurrency in eclipse
- 3) Concurrency and the user interface
- 4) Concurrency and the workspace
- 5) Concurrency in the large

Part 1: Principles of concurrency

- Concurrency ideal: each thread is completely independent
- Concurrency is reduced and programming challenges arise when threads become entangled in either space or time
 - Space: when threads share physical resources such as memory, devices
 - Time: when threads have temporal dependencies (x must run before y)
- Goal: minimize dependencies between threads, and carefully handle the places where threads intersect

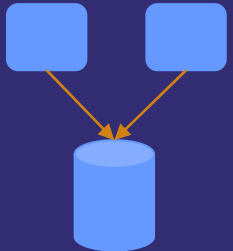
Dominant concurrency models

- Message passing



- No shared state between threads
- All interaction through asynchronous communication
- Typically employs observer pattern and event queues
- Eliminates tangling in space but increases tangling in time

- Shared state



- Multiple threads operating on and competing for shared resources
- Threads typically communicate via shared state
- Most common model, looks easy to implement, but error prone
- Manage thread interaction with locks

Pop quiz: Is this code thread safe?

```
List list = new ArrayList();
class One implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++)
            list.add(new Integer(i));
        for (int i = 0; i < 5; i++)
            list.remove(new Integer(i));
    }
}
class Two implements Runnable {
    public void run() {
        for (Iterator it = list.iterator(); it.hasNext();)
            System.out.println(it.next());
    }
}
public void doit() {
    new Thread(new One()).start();
    new Thread(new Two()).start();
}
```

Pop quiz (part 2): Is this good enough?

```
List list = Collections.synchronizedList(new ArrayList());
class One implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++)
            list.add(new Integer(i));
        for (int i = 0; i < 5; i++)
            list.remove(new Integer(i));
    }
}
class Two implements Runnable {
    public void run() {
        for (Iterator it = list.iterator(); it.hasNext();)
            System.out.println(it.next());
    }
}
public void doit() {
    new Thread(new One()).start();
    new Thread(new Two()).start();
}
```

Pop quiz (part 3): And now?

```
List list = Collections.synchronizedList(new ArrayList());
class One implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++)
            list.add(new Integer(i));
        for (int i = 0; i < 5; i++)
            list.remove(new Integer(i));
    }
}
class Two implements Runnable {
    public void run() {
        Object[] els = list.toArray(new Object[list.size()]);
        for (int i = 0; i < els.length; i++)
            System.out.println(elements[i]);
    }
}
...
```

Pop quiz: Synchronized solution

```
List list = Collections.synchronizedList(new ArrayList());
class One implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++)
            list.add(new Integer(i));
        for (int i = 0; i < 5; i++)
            list.remove(new Integer(i));
    }
}
class Two implements Runnable {
    public void run() {
        Object[] els = list.toArray(new Object[0]);
        for (int i = 0; i < els.length; i++)
            System.out.println(els[i]);
    }
}
...
```

Pop quiz: Copy on write solution

```
ArrayList list = new ArrayList();
class One implements Runnable {
    public void run() {
        ArrayList list2 = (ArrayList)list.clone();
        for (int i = 0; i < 10; i++)
            list2.add(new Integer(i));
        for (int i = 0; i < 5; i++)
            list2.remove(new Integer(i));
        list = list2;
    }
}
class Two implements Runnable {
    public void run() {
        for (Iterator it = list.iterator(); it.hasNext();)
            System.out.println(it.next());
    }
}
...
```

The lesson about shared state

- Coordinating access to shared state is a very tricky business
- Techniques for managing shared state include:
 - Make state immutable
 - Copy on write
 - Synchronize
- Synchronization introduces the possibility of deadlock
- Asynchronous message passing avoids these pitfalls but is not always possible

Part 2: Concurrency in Eclipse

- Job API: `org.eclipse.core.runtime.jobs`
- Job: a unit of work scheduled to run asynchronously
- Why not just `java.lang.Thread`?
 - Lighter weight: uses a shared thread pool
 - GUI integration: progress and cancelation
 - Priorities and mutual exclusion
 - Richer scheduling: run now, run later, run repeatedly
 - Job listeners can find out when jobs start, finish

Hello jobs world

```
class HelloJob extends Job {
    public HelloJob(String name) {
        super(name);
    }
    protected IStatus run(IProgressMonitor monitor) {
        System.out.println("Hello from " + getName());
        return Status.OK_STATUS;
    }
}
public void doit() {
    new HelloJob("Job1").schedule();
    new HelloJob("Job2").schedule();
}
```

Hello job output

- **Result:**

```
Hello from Job1
```

```
Hello from Job2
```

- **Or perhaps:**

```
Hello from Job2
```

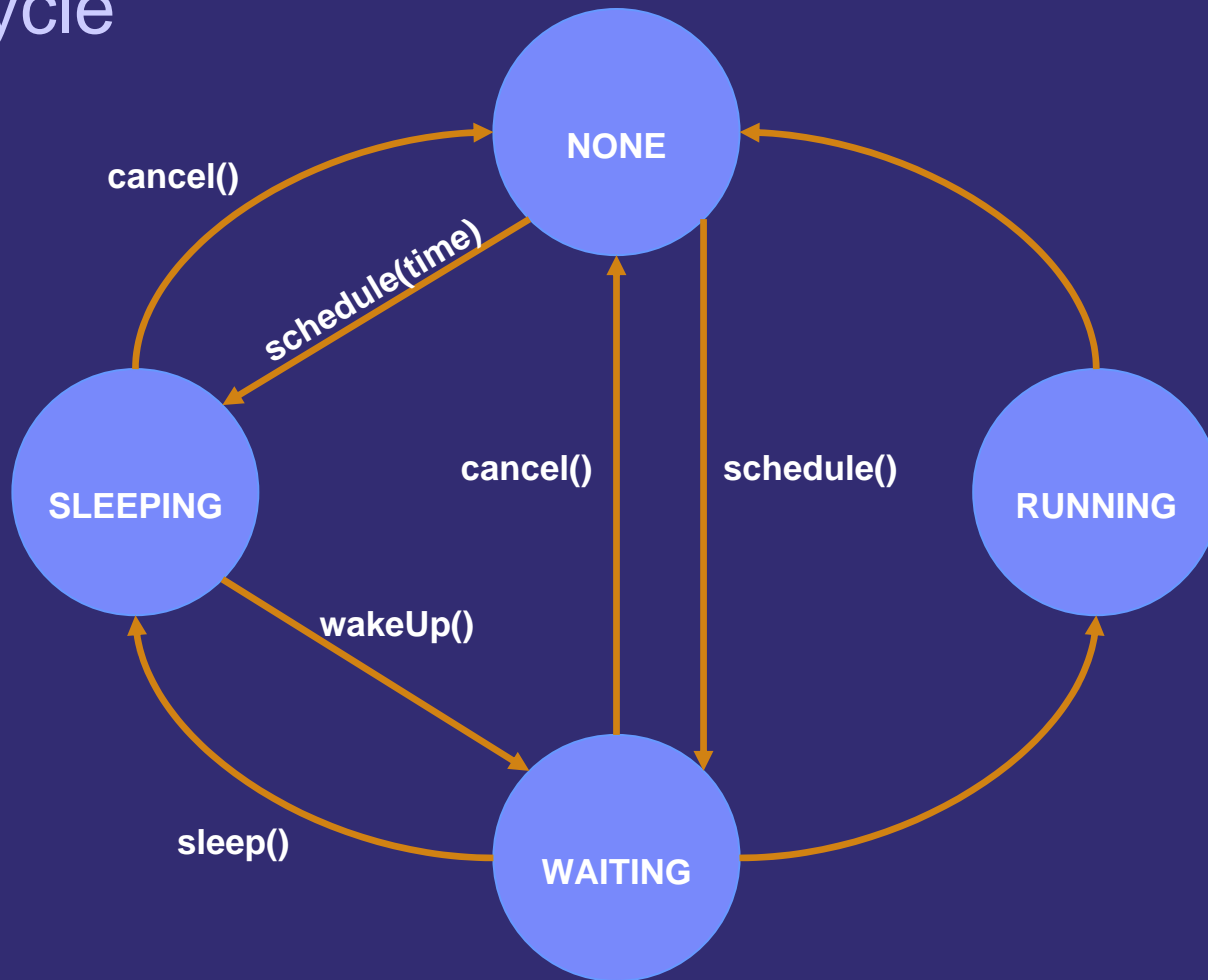
```
Hello from Job1
```

- In simple form, jobs offer no guarantees about execution time
- Multiple jobs can run at the same time

Job states

- A job instance can only be in one state at a given time
 - NONE: Job has not been scheduled or has finished running
 - WAITING: Job is queued to run as soon as possible
 - SLEEPING: Job should be run at some time in the future
 - RUNNING: Job is currently running
- Find out job state using `Job.getState()`; only predictable when called from the job's own thread
- Various job methods can be used to alter the state of a job
- Jobs are reusable; often a singleton instance is used

Job lifecycle

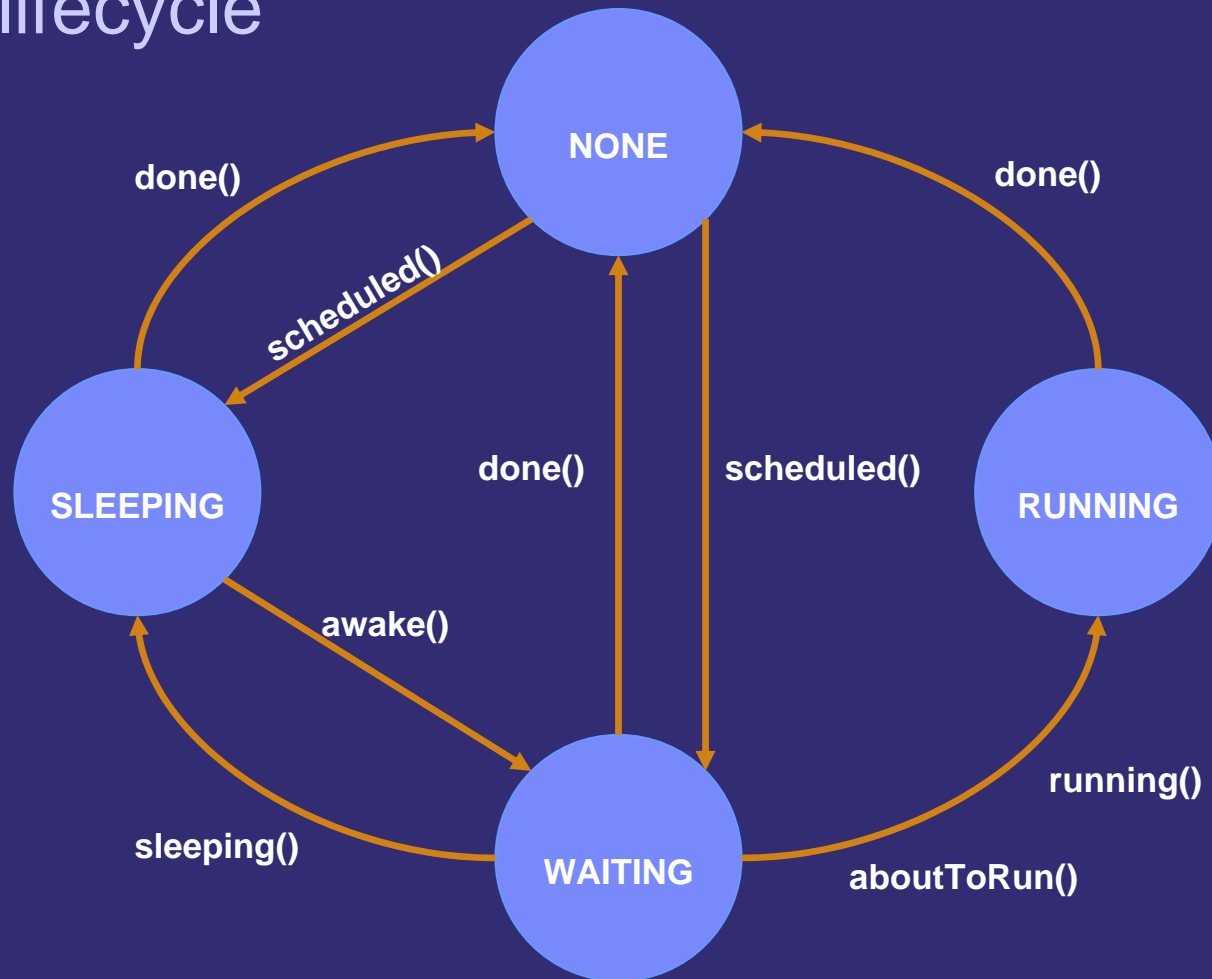


Job Listeners

- Anyone can add a listener to a particular job to keep track of its life cycle (`Job.addJobChangeListener(...)`)
- In rare cases listen directly to job manager to track state of all jobs (`IJobManager.addJobChangeListener(...)`)

```
class MyJobListener extends JobChangeAdapter {  
    public void done(IJobChangeEvent event) {  
        System.out.println("Done: " + event.getJob());  
    }  
}
```

Listener lifecycle



Job scheduling

- Job priorities are used as a hint about how soon the job should be run (interactive, short, long, build, decorate)
- Job priority is unrelated to Thread priority
- Can schedule a job to run now or run later
- Scheduling a job that is already running will cause it to be rescheduled as soon as it completes
- Can override `shouldRun()` or `shouldSchedule()` to make last minute decisions about whether to run or rerun a job

Exercise: A job creation view

- Create a subclass of `AbstractJobView`
- Implement the `runJob()` method to create and schedule jobs
- Call `runJobBody()` convenience method from the job's `run` method
- Print job status by calling `setStatus(String)`

Scheduling Rules

- Allow you to avoid certain jobs running at the same time
 - Example: Can't copy a file while deleting it

```
public interface ISchedulingRule {  
    public boolean contains(ISchedulingRule rule);  
    public boolean isConflicting(ISchedulingRule rule);  
}
```

- `isConflicting()` method specifies if your rule can run at same time as others
- `contains()` method specifies if a rule is a subset of another rule

Scheduling Rules (continued)

- Scheduling rules can also be used outside of jobs
- Prevents a block of code from running concurrently with any job or other block using that rule

```
IJobManager jobMan = Platform.getJobManager();  
final ISchedulingRule rule = new MySchedulingRule();  
try {  
    jobMan.beginRule(rule, null);  
    // do some work  
} finally {  
    jobMan.endRule(rule);  
}
```

MultiRule

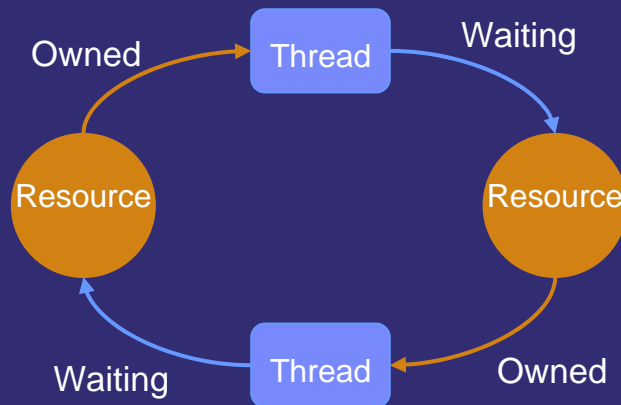
- At any given time, a single thread can only own one rule
- Topmost scheduling rule must be all-inclusive
 - Could use highest node in hierarchy
 - This inhibits concurrency
 - What if you need several unrelated rules?
- Need a means to combine two or more rules
 - MultiRule aggregates rules
 - `MultiRule.combine(rule1, rule2)`
 - `MultiRule.combine(rules);`

Locks

- ILock: re-entrant lock, similar to Java object monitors
- Acquires are granted in FIFO order
- Aware of syncExec: carries over to UI thread
- Deadlock reporting and recovery
 - Not a magic bullet!!
 - Intended as a programming aid

The necessary conditions for deadlock

- Mutual exclusion: resource held by a single thread
- Non-preemptible: resource can't be taken away
- Hold and wait: Each thread holds a resource and waits for another
- Circular wait: Waiting threads form a circle



Eliminating deadlock

- Avoid mutual exclusion
 - Use non-locking read operations as much as possible
 - Copy on write to avoid locking entirely
- Avoid hold and wait:
 - ISchedulingRule: allows jobs to specify requirements before they even start (two phase locking)
 - Impossible to hold a rule and be waiting for a rule
 - Avoid holding locks while client code is called
 - Avoid syncExec and use asyncExec where possible
- Avoid circular wait
 - Always acquire locks in a consistent order
- Preemption:
 - If all else fails, preempt the thread that introduced deadlock and report error in log

Pop quiz: deadlock?

```
class Deadlock extends Job {
    Deadlock() {super("Deadlock");}
    public synchronized IStatus run(IProgressMonitor pm) {
        Display display = window.getShell().getDisplay();
        display.syncExec(new Runnable() {
            public void run() {
                block();
            }
        });
        return Status.OK_STATUS;
    }
    public synchronized void block() {
    }
}
```

Solving deadlock with ILock

```
class NoDeadlock extends Job {
    ILock lock = Platform.getJobManager().newLock();
    public IStatus run(IProgressMonitor monitor) {
        lock.acquire();
        Display display = window.getShell().getDisplay();
        display.syncExec(new Runnable() {
            public void run() {
                block();
            }
        });
        lock.release();
        return Status.OK_STATUS;
    }
    public void block() {
        lock.acquire();
        lock.release();
    }
}
```

Party 3: Concurrency and the UI

- Topics to cover:
 - Progress and cancelation
 - User Jobs
 - Scheduling Rules
 - Job Configuration
 - Interaction with the UI thread
- Exercises involving a simple plug-in that:
 - Backups up one or more projects to a Backup project
 - Backup path is /Backups/<project name>/<backup key>/...
 - Restores a project from the backups

Exercise: Backup and restore plug-in

- Load the provided plug-in into your workspace
- Launch a run-time Eclipse
- Import the provided test plug-ins
- Perform some backups and restores
- How was the progress feedback?

Progress and cancelation

- Long running operations need to have:
 - Visual feedback of progress
 - Busy cursor
 - Graphical progress bar
 - Textual task/subtask descriptions
 - Ability to cancel
- Abstraction required between reporting and showing of progress
 - Using IProgressMonitor API for reporting progress
 - UI mechanisms for displaying progress

IProgressMonitor

- API features
 - reports progress and supports cancelation
 - Can be nested within other monitors
- How do we get a progress monitor?

```
public void doIt(IProgressMonitor monitor) {
    monitor.beginTask("Starting something big", 100);
    monitor.subTask("Doing the first part");

    doPartOne(new SubProgressMonitor(monitor, 40));

    if (monitor.isCanceled()) return;

    monitor.subTask("Doing the second part");
    // Do stuff
    monitor.worked(60);

    monitor.done();
}
```

Obtaining an IProgressMonitor

- IRunnableContext/IRunnableWithProgress
 - APIs for providing a progress monitor
 - Clients implements IRunnableWithProgress
- Context provided by
 - ProgressMonitorDialog
 - Show progress to the user using a dialog
 - Supports cancelation
 - IProgressService
 - run
 - Similar behavior to ProgressMonitorDialog
 - busyCursorWhile
 - Shows a busy cursor for a short time before opening a dialog

Progress services

```
IRunnableWithProgress runnable = new IRunnableWithProgress() {
    public void run(IProgressMonitor monitor) {
        doIt(monitor);
    }
};

IWorkbench workbench = PlatformUI.getWorkbench()
IRunnableContext context = workbench.getProgressService();

IRunnableContext context = new ProgressMonitorDialog(getShell());

context.run(true /* fork */, true /* cancelable */, runnable);

IProgressService service = workbench.getProgressService();
service.busyCursorWhile(runnable);
```

Exercise: Add progress and cancelation

- Convert the BackupAction and RestoreAction to use one of the previously mentioned progress services.
 - Wrap the appropriate method calls in the run methods
 - Wrap CoreException in InvocationTargetException
 - Why not just prompt at that point?
 - Push the monitor down to the IResource API calls
- How is progress and cancelation now?

Reporting progress from jobs

- Jobs also use a progress monitor
- Execution characteristics differ from other progress services
 - Executed asynchronously
 - Job specific progress feedback
 - Handling of errors returned from `Job.run()`
- There are several “types” of job progress
 - System jobs do not show progress anywhere
 - `Job.setSystem(true)`
 - Other jobs are shown in the progress view and status line
 - User jobs present a progress dialog
 - Option to run in the background (once or always)
 - `Job.setUser(true);`

Exercise: Convert to jobs

- Convert the BackupAction and RestoreAction to use Jobs
 - Don't forget to schedule the job
 - Try it out.
 - How was the user experience?
 - Try setting the job to be a user job: `Job.setUser(true)`
 - How does it look now?
 - Error handling?
 - Return the status of the CoreException for now (see next slide)
- Was converting to a Job easy?
 - Why (or why not)?

Error handling

- The run method of a Job returns a status
 - ERROR – user is prompted with the error
 - WARNING, INFO – logged
- Jobs should create a new status object when errors occur
 - Tempting to return the status of a CoreException
 - Stack trace would then be lost
- Should exceptions terminate the job
 - Decision made on a case by case basis
 - If not, exceptions should be accumulated and a MultiStatus returned

Job feedback

- For long-running operations, the user wants to know
 - What is the operation doing?
 - Is the operation done?
 - Did the operation succeed?
 - What were the results?
- There are two mechanisms provided for background job feedback
 - A site associated with an originating or associated view
 - `IWorkbenchSiteProgressService`
 - The Progress view
- Can also use custom approaches
 - the synchronize view italicizes resources that are currently being operated on

Feedback in an associated view

- IWorkbenchSiteProgressService

- Associated with a part (e.g. view)

```
part.getSite().getAdapter(IWorkbenchSiteProgressService.class);
```

- schedule(job, delay, busyCursor)

- schedule jobs through the service to get busy feedback (*italic title*) and job-busy cursor

- showBusyForFamily(family)

- Show busy whenever a job of the given family is running

- warnOfContentChange()

- Bold view title to indicate the contents of the view have changed

The Progress view

- The Progress view is where job progress is shown
- Job properties are used to configure the view
- Properties defined in IProgressConstants
 - KEEP_PROPERTY – keep all finished jobs in the progress view
 - KEEPONE_PROPERTY – keep only the last job in the view
 - ICON_PROPERTY – the icon that appears with the job in the view
 - URL or ImageDescriptor
 - ACTION_PROPERTY – action for job’s hyperlink
 - PROPERTY_IN_DIALOG – set by progress service to indicate whether the progress dialog is visible or not
- For KEEP and KEEPONE, the text of the returned status is used as the hyperlink text

Exercise: Add properties to Backup/Restore

- Schedule the jobs through the site progress service
- Add the KEEP and ICON properties to the Backup/Restore
 - `ProjectAction.getImageDescriptor()` - example image
- Add an ACTION property that shows the ConsoleView
 - Use a subclass of `org.eclipse.jface.Action`
 - Override the run method
 - Invoke `ConsoleView.showView()`
- Output to Console
 - Add calls to `ConsoleView.writeLine(String)` to appropriate places
 - Will need to be in UI thread
 - `Display.getDefault().asyncExec(new Runnable(...`

Updating UI from a job

- All UI updates must be performed from the UI thread
- Low-level means of updating UI (SWT)
 - `Display.syncExec(Runnable)` - synchronous
 - `Display.asyncExec(Runnable)` – asynchronous
- Higher-level
 - `UIJob` – asynchronous with return status
 - `WorkbenchJob` (extends `UIJob`) – won't start jobs after workbench shutdown

Considerations when updating UI

- Do minimal amount of work in the UI thread
- Batch updates whenever possible
 - Reduces Job/syncExec overhead
 - Reduces UI “disco-ball” effect
 - How often does the user need to see feedback?

Exercise: Batching updates

- Load and try out the Delta2Console example
- This code runs an `asyncExec` for each change
- Convert this example to use a `WorkbenchJob` to update the console
- How do we add batching?
- How do we know when to dispatch?
 - After a certain number of outputs
 - After a certain amount of time?
- How do we ensure that no output is lost?
- How do we handle errors?

Part 4: Concurrency and the workspace

- Changes made in 3.0 to make the workspace more concurrency-friendly
- How to run jobs that modify resources
- How the workspace uses jobs

Resource scheduling

- Eclipse resources are thread-safe
 - Can “lock” at individual resource level
 - Operations on resources obtain appropriate scheduling rules
- We can perform resource operations without worrying about it
- Why do we need to provide the job with a scheduling rule?
 - Job only runs when all rules are available
 - Reduced overhead
 - Operation is performed as an atomic unit on the resources
- What rules do we need for Backup and Restore?

Resource scheduling rule factory

- If we want to modify a resource, we need to get the proper rule
- The rules required to perform resource operations may vary
 - Some resource changes are linked to other resources
 - (I.e. changes to a link affect the .project file)
 - Repository provider may have additional requirements
 - operations may modify meta-files
 - Future requirements
- To Modify a project, we need to get the modify rule
 - `ResourcesPlugin.getWorkspace().getRuleFactory().modifyRule(project)`
- There are similar factories for other operation types
 - Create, copy, move, delete, validateEdit, etc

Workspace changes and builds

- What happens after a resource is modified?
 - Change notification sent to listeners
 - Auto-build is started
- What if several resources are modified by a single operation?
 - Don't want to send notification and start a build for each
- Several mechanisms provided to batch changes
 - `IWorkspace.run` methods
 - `ResourcesPlugin.getWorkspace().run(IWorkspaceRunnable, ISchedulingRule, int, IProgressMonitor)`
 - `WorkspaceJob`
- Auto-build is postponed and change notifications are batched
 - `POST_CHANGE` notifications occur periodically within an operation

Background builds and change notifications

- Workspace builds occur in the background
 - All workspace modifications are blocked during a build
 - To improve responsiveness, builds will yield to blocked jobs
 - Using `Job.isBlocking()`
 - Yield point is determined by the active builder
 - Will occur at a point where the builder can easily resume
- Change notification also occurs in the background
 - Again, workspace modifications are blocked
 - Within a `POST_CHANGE` resource change listener:
 - workspace modifications are not allowed
 - Scheduling rules cannot be obtained

Exercise: Workspace considerations

- Do we need change batching?
 - Make the job a WorkspaceJob
- Do we need scheduling rules?
 - What is happening without any additional rules?
 - Each resource operation obtains the most specific rule possible
 - This leaves lots of opportunity for conflicting modifications
- Would like to add rules to ensure atomicity
 - What is needed for Restore?
 - What is needed for Backup?

Part 5: Concurrency in the large

- Best practices and principles for creating large-scale concurrent applications built on Eclipse
- Techniques for making a large existing application concurrent
- Wrap up

Designing concurrent/responsive applications

- Clearly separate the bulk of application logic from presentation (GUI) code
- GUI code always runs in UI thread – no worries about concurrency and thread safety of data structures
- Build API walls around pieces of application logic
- APIs must be prepared to be called concurrently in any thread
- APIs must specify “concurrency requirements” – locks needed, assumptions about calling thread, etc
- Don’t hold locks while calling into API, or when core code calls out

Making an existing application responsive

- Step 0 - If you make no changes, you will be ok (even slightly better)
- Step 1 - Revisit locks to reduce contention with background jobs
- Step 2 – Move long read only operations to background
- Step 3 – Move long writing operations to background

Step 0 – What if I do nothing?

- Backwards compatibility: no worse than Eclipse 2.1
- Blockage will always be reported to the user
- The UI is kept alive (painting) at all costs
- Cancellation is now always possible (even during deadlock)
- Lack of responsiveness in one component can kill responsiveness gains in other components

Step 1 - Revisit locks to reduce contention

- Identify where threads can become tangled
- Remove locks where possible
- Fine-grained locks to increase concurrency
- Lock for shorter periods of time
- Use scheduling rules to lock specific resources

Step 2 – Move read-only operations to background

- Less risk in moving read-only operations into background
- Can often be done with little or no contention
- Watch for assumptions about UI thread and thread safety
- Examples: searching, indexing, decoration, repository view
- Make sure things running silently in the background don't interrupt user tasks

Step 3 – Move writing operations to background

- Identify the big win operations – what common tasks will the user want to be able to perform in the background?
- Trade-off is added complexity of code versus important responsiveness gains
- Need to be aware of concurrency requirements of code you're calling: locks acquired, etc
- Be aware of deadlock risks and know avoidance strategies

References and resources

- Examples plug-in ([dev.eclipse.org](http://dev.eclipse.org/org.eclipse.ui.examples.jobs))
org.eclipse.ui.examples.jobs
- *GUI Bloopers*, Jeff Johnson – Chapter 7: Responsiveness Bloopers
- *Concurrent Programming in Java*, Doug Lea
- *Modern Operating Systems*, Andrew S. Tanenbaum

Questions?