Welcome!

Administrivia

- Resources
  - Web page
    - http://faculty.washington.edu/rmoul/java/advanced
  - Email
    - rmoul@uw.edu
  - Phone
    - (253) 657-9568
- Schedule
  - Lecture schedule
  - Office hours
  - I always respond to e-mail

Highlights From Syllabus

- 8 assignments
  - Assignment 4 has double weight
- Must complete every assignment
- Must attend 8 out of 10 lectures

Grading - Unforgivable Sins

- NullPointerException
  - Easily corrected
    - Stack trace identifies the exact line of code
    - This line must access a reference that is null
  - Common causes
    - Un-initialized variables, especially member variables
    - Chained methods calls, intermediate methods returns null
- Caught and ignored exceptions
  - Every caught exception must either
    - Be handled, the originating problem resolved
    - Generate a stack trace
  - Absent this - very difficult to diagnose/debug

Topics Covered

- Preferences & Logging
- Database use & DAO
- Structured I/O
- Persistence & Text Processing
- Generics
- Multi-Threading
- Networking
- Remote Method Invocation
- Deployment and Basic Security
- Review

Logging Framework

Concepts

- Namespaces
  - All loggers are given a name
    - Typically the dot separated class name of the component
- Levels
  - What level of message should be logged

`Level`

- The `Level` class provides a standard, high-level means of controlling which messages are output
  - `SEVERE` (highest value)
  - `WARNING`
  - `INFO`
  - `CONFIG`
  - `FINE`
  - `FINER`
  - `FINEST`
- Two special levels
  - `OFF`
  - `ALL`

Logger

- **Used to log messages for a specific class/ components**
- **By convention given components dot separated name**
  - **May have an arbitrary name**
- **Knows its parent**
  - `Logger getParent()`
  - `void setUseParentHandlers( boolean )`

Logging

- **Methods for logging**
  - `void log( Level, String )`
  - `void log( Level, String, Object )`
  - `void log( Level, String, Object[] )`
  - `void log( Level, String, Throwable )`
- **Convenience methods**
  - `void info( String )`
  - `void warning( String )`
  - `void severe( String )`
  - **Etc…**

Logger

- Has an associated level, the minimum level it is concerned with
  - If null level is inherited from parent
- Has a set of Handlers
- Support localization

More Logging

- **More convenience methods**
  (logged at the FINER **level)**

  -
    ```
    void entering( String srcClass,
                   String srcMthd )
    ```

  -
    ```
    void exiting( String srcClass,
                  String srcMthd )
    ```

  -
    ```
    void throwing( String srcClass,
                   String srcMthd,
                   Throwable t )
    ```
- **Avoiding un-needed work**
  - `boolean isLoggable( Level )`

LogRecord

- Created by loggers, represents the message
  - Level
  - Sequence
  - Time
  - Message
- Used to pass messages between threads or JVMs
- Includes localization information

Handler

- **Accepts messages from a logger and exports them**
- **Has a level, formatter & filter**
- **Provided handlers**
  - `MemoryHandler`
  - `StreamHandler`
    - `ConsoleHandler`
    - `FileHandler`
    - `SocketHandler`

Custom Handler

- **Extend the** `Handler` **class**
- **Override abstract methods of** `Handler`
  - `void close()`
  - `void flush()`
  - `void publish( LogRecord )`

Filter

- **Allows fine grained control over decision to log a message**
- **Interface has a single operation**
  - `boolean isLoggable( LogRecord )`

Formatter

- **The** `Formatter` **class**
  - **Supports formatting of** `LogRecords` **as strings**
  - **Abstract**
- **Standard formatters**
  - `SimpleFormatter`
  - `XMLFormatter`

Custom Formatter

- Extend the `Formatter` class
  - Override abstract method of `Formatter`
    - `String format( LogRecord )`
  - May not be require, if the result isn't a formatted string

Formatter Example

```
public class BriefLoggingFormatter extends Formatter {
   private static final String LINE_TERM =
                        System.getProperty("line.separator");

   public String format(LogRecord record) {
      return record.getLevel().toString() + ": "
           + formatMessage(record) + LINE_TERM;
   }

   public String formatMessage(LogRecord record) {
      return record.getMessage();
   }
}
```

LogManager

- Single shared instance
- Methods for accessing loggers
- Methods for accessing logger properties
    - `String getProperty(String propName)`
- Allows configuration of loggers
    - Default
        - `<JAVA_HOME>/jre/lib/logging.properties`
    - Optional configurations
        - `java.util.logging.config.class`
        - `java.util.logging.config.file`

Logging Configuration

- Properties ending in “`.level`” specify the logging level for a class or package
- Properties accessible through `getProperty` method

The Process

1. Obtain logger from the `LogManager`
   - Existing logger is returned, or the appropriate logger is created and returned
2. A message is logged with the logger
3. The message is discarded if it does not satisfy the loggers level
4. `Logger` creates a `LogRecord`
5. The log record is tested against the filter if one exists

The Process (continued)

6. The log record is published to the logger’ s handlers and the parent logger
7. The message is discarded if it does not satisfy the handlers level
8. The handler check the log record against their filter, if one exists
9. The handler uses it’ s `Formatter` to format the message
10. The handler exports the message

Logging Example

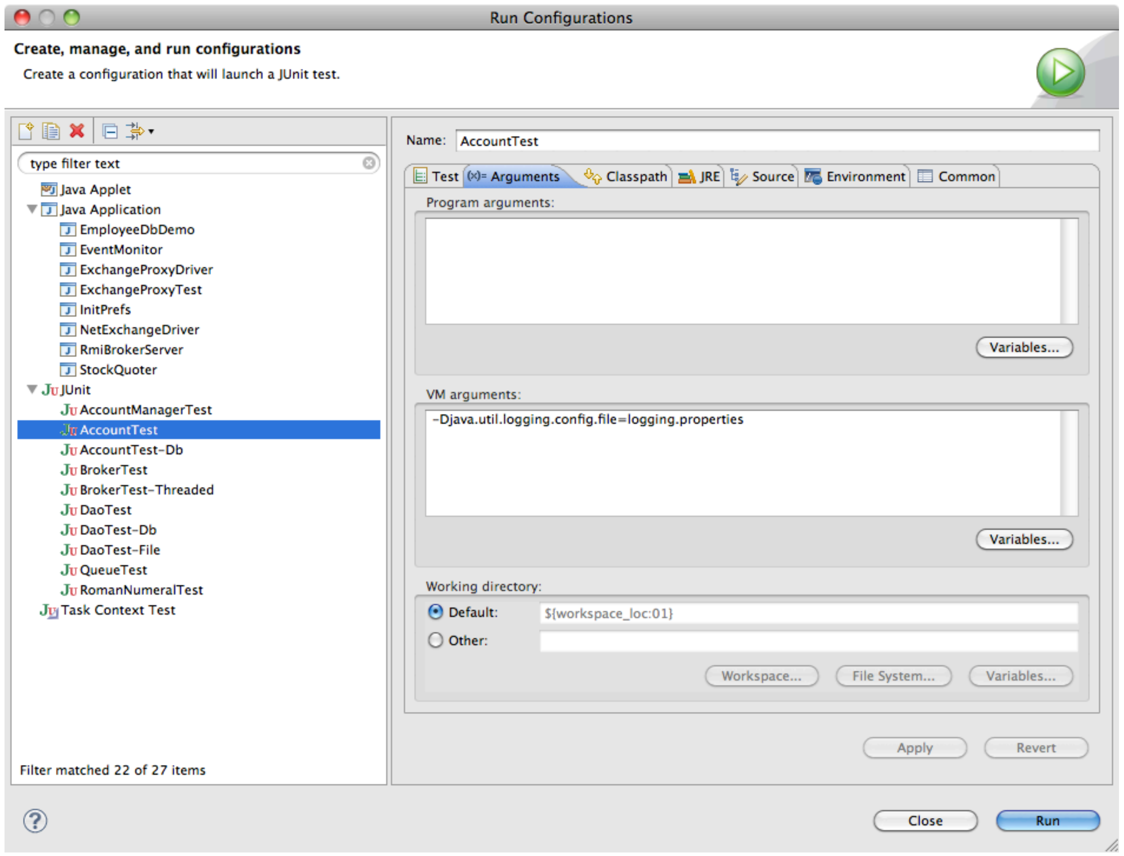
```
public class SomeClass {
   private static Logger logger =
                  Logger.getLogger(SomeClass.class.getName());

   void someMethod() {
      if (logger.isLoggable(Level.INFO)) {
         logger.info( "For your information..." + getInfo() );
      }
      try {
         ...
      } catch( Exception ex ) {
         logger.log( Level.SEVERE, "Uh oh...", ex );
      }
   }
}
```

Eclipse Logging Configuration

- Need to set the `java.util.logging.config.file` property:
  1. Open the "Run Configurations" dialog
  2. Select the desired run configuration
  3. Select the "Arguments" tab
  4. Add the property to the "VM arguments"
  5. Click the "Apply" button
  6. Close the dialog

Eclipse Configuration

- **Open the "Run Configurations" dialog**
  - **Menu: Run->Run Configurations ...**



Run Configurations

Create, manage, and run configurations

Create a configuration that will launch a JUnit test.

type filter text

- Java Applet
- Java Application
  - EmployeeDbDemo
  - EventMonitor
  - ExchangeProxyDriver
  - ExchangeProxyTest
  - InitPrefs
  - NetExchangeDriver
  - RmiBrokerServer
  - StockQuoter
- JUnit
  - AccountManagerTest
  - AccountTest
  - AccountTest-Db
  - BrokerTest
  - BrokerTest-Threaded
  - DaoTest
  - DaoTest-Db
  - DaoTest-File
  - QueueTest
  - RomanNumeralTest
- Task Context Test

Filter matched 22 of 27 items

Name: AccountTest

Test | Arguments | Classpath | JRE | Source | Environment | Common

Program arguments:

VM arguments:

```
-Djava.util.logging.config.file=logging.properties
```

Working directory:

(●) Default: ${workspace_loc:01}

( ) Other:

Log4j

- A precursor to the current logging framework
  - Developed by Apache
  - Conceptually very similar
  - Somewhat more complex
- Equivalencies
  - `Loggers`
  - `Levels`
  - `Handler/Appender`
  - `Filter`

Preferences Framework

Preferences

- `java.util.prefs`
- Hierarchical collection of preference nodes
- Dual hierarchy, system & user
- Preferences class
  - Represents a node
  - Abstract
- Individual preferences accessible by path like string
  - '/' delimiter
- Typed values, not just strings

Setting and Getting

- **Getters and setters for string preferences**
  - `void put( String key, String value )`
  - `String get( String key, String default )`
- **Getters and setters for primitive types**
  - void put*Type*(String key, *type* value)
  - *type* get*Type* (String key, *type* default)

Backing Store

- Implementation dependent
  - Flat file
    - `/etc/.java/.systemPrefs`
    - `$HOME/.java/.userPrefs`
  - Registry
    - `HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Prefs`
    - `HKEY_CURRENT_USER\Software\JavaSoft\Prefs`
  - LDAP
  - RDBMS
  - Import/Export from/to XML

Preference Listeners

- `NodeChangeListener`
  - **Monitor addition or removal of child nodes**
- `PreferenceChangeListener`
  - **Monitor changes to preferences within a node**

Package Association

- **Classes generally translate package name to preference path**
  - `Preferences userNodeForPackage( Class )`
  - `Preferences systemNodeForPackage( Class )`
- **The unnamed package is “**`<unnamed>`**”**
- **Root user and system nodes**
  - `Preferences userRoot()`
  - `Preferences systemRoot()`

Threading

- Asynchronous
  - Write operations are asynchronous
  - When used by a single JVM will be equivalent to a some serial execution
- Thread-safe
  - When used by multiple JVMs backing store will not be corrupted – no other guarantees

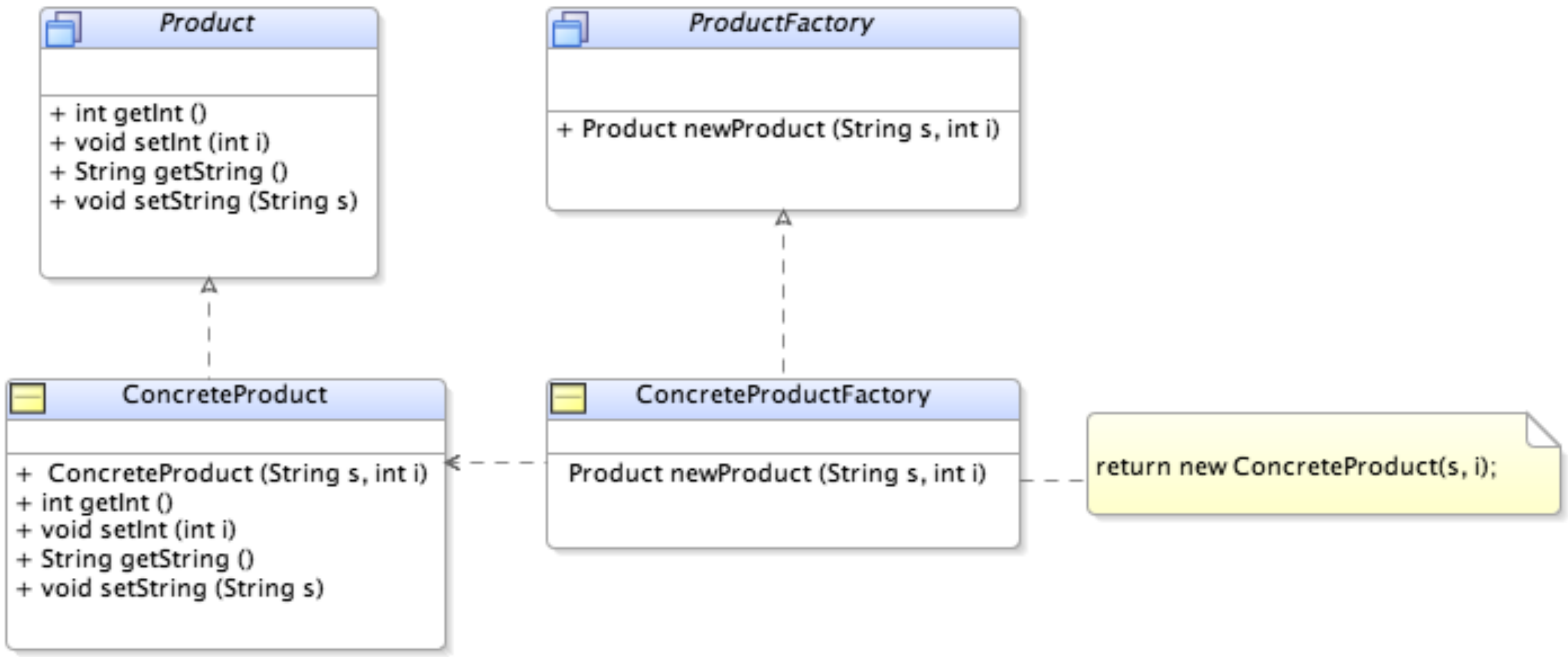
Preferences Example

```
package edu.washington.example;
import java.util.prefs.Preferences;

public class PrefExample {
   Preferences p = Preferences.userNodeForPackage( this.getClass() );
   private String workingDir = ".";

   private void getState() {
      workingDir = p.get( "lastWorkingDir", "." );
   }

   private void setState() {
      p.put( "lastWorkingDir", workingDir );
   }
   ...
}
```

XML Export

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM 'http://java.sun.com/dtd/preferences.dtd'>
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map />
    <node name="edu">
      <map />
      <node name="washington">
        <map />
        <node name="example">
          <map>
            <entry key="lastWorkingDir" value="/usr/home/russ" />
          </map>
        </node>
      </node>
    </node>
  </root>
</preferences>
```

Factory Method


Product
+ int getInt ()
+ void setInt (int i)
+ String getString ()
+ void setString (String s)
ProductFactory
+ Product newProduct (String s, int i)
ConcreteProduct
+ ConcreteProduct (String s, int i)
+ int getInt ()
+ void setInt (int i)
+ String getString ()
+ void setString (String s)
ConcreteProductFactory
Product newProduct (String s, int i)
return new ConcreteProduct(s, i);

Why Factory Method

- Factory determines implementation class
- Allows imposition of construction arguments
  - Interfaces can't specify constructors
  - Or static methods