# CSC 307 Lecture Notes Week 7
# Design for Independent, Incremental Testing
# Refining Model Design Using the Java Library
# A Key Design Patterns for 307 Projects

## I. Designing for independently testable packages.

A. Team developers should design their individual packages and classes to be testable independently from other team members' packages.

B. Providing "canned" test data is one aspect of independently testable designs.

1. Such test data are used when data-producing packages are not yet implemented by other team members.
2. These data can also be very handy when an implemented package unexpectedly breaks, leaving users of the package stuck until the broken package is fixed.

C. In a Java-based implementation, having individualized `main` methods supports independent testing.

1. For convenience, these `mains` can be in model classes.
2. During the initial phases of development, it's fine to have some testing done directly in the model classes.
   a. These initial tests are designed to check that an implementation is going in the right direction.
   b. The initial tests will evolve into the formal tests that are in the `testing` project directory.

D. A testing `main` method in a model class does the following steps:

1. Construct model class(es) to be tested.
2. Construct and compose companion view(s).
3. Construct canned test data.
4. Show the top-level view(s).

E. Independently-testable designs allow *incremental* development, meaning the implementation can be tested in successively refined increments, each with more strenuous and less canned test data.

## II. Java Library packages for model and process data.

***Question:*** *How many packages and classes in the standard Java 7 library?*

***Answer:*** *209 and 4205, respectively*

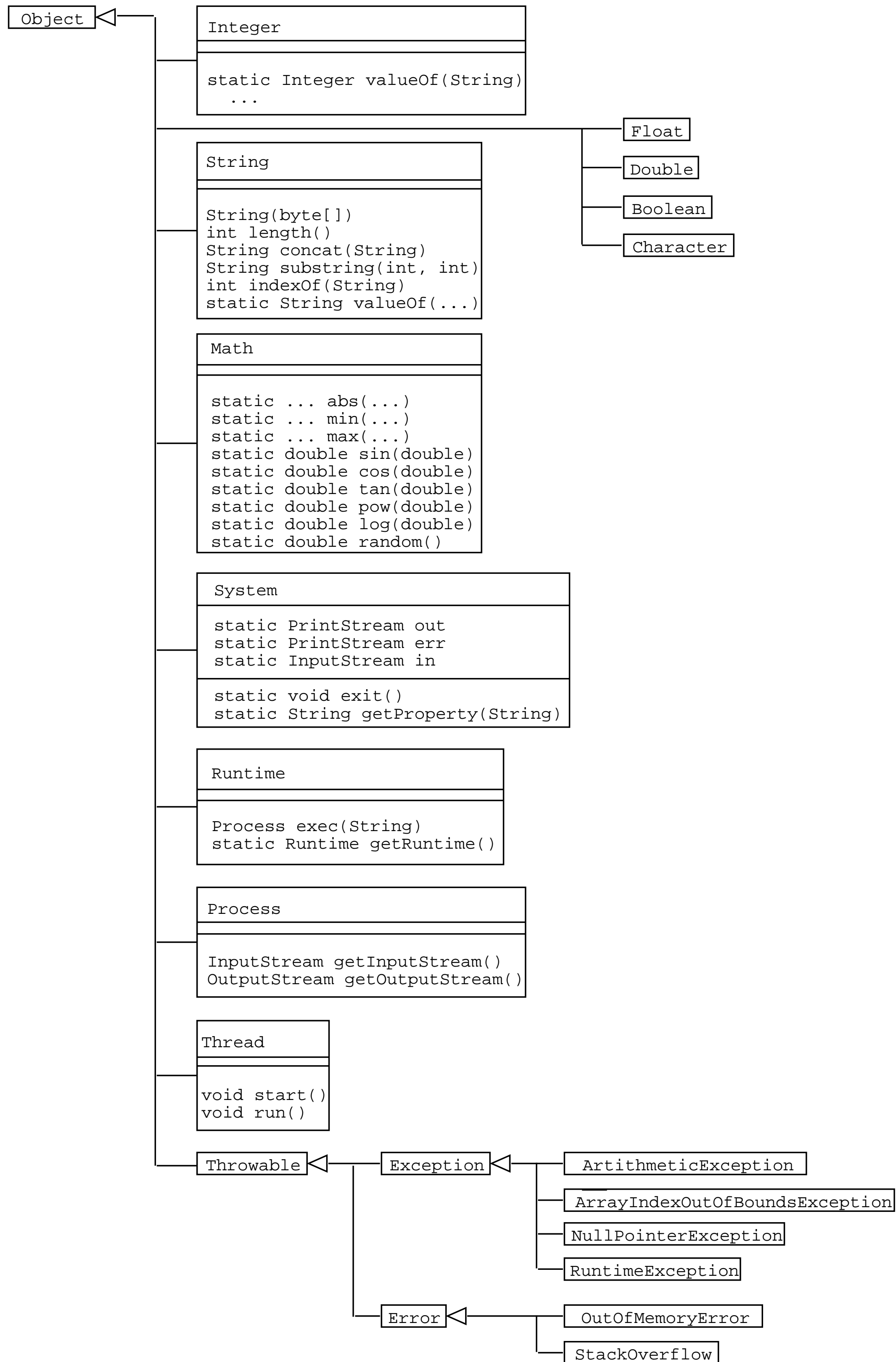
A. The key packages we'll cover in these notes are:

1. *java.lang* -- the fundamental Java language package, with classes such as Object, String, and System.
2. *java.util* -- the higher-level Java language package, including the collection classes, date/time classes, and others.
3. *java.io* -- file input/output and related processing.

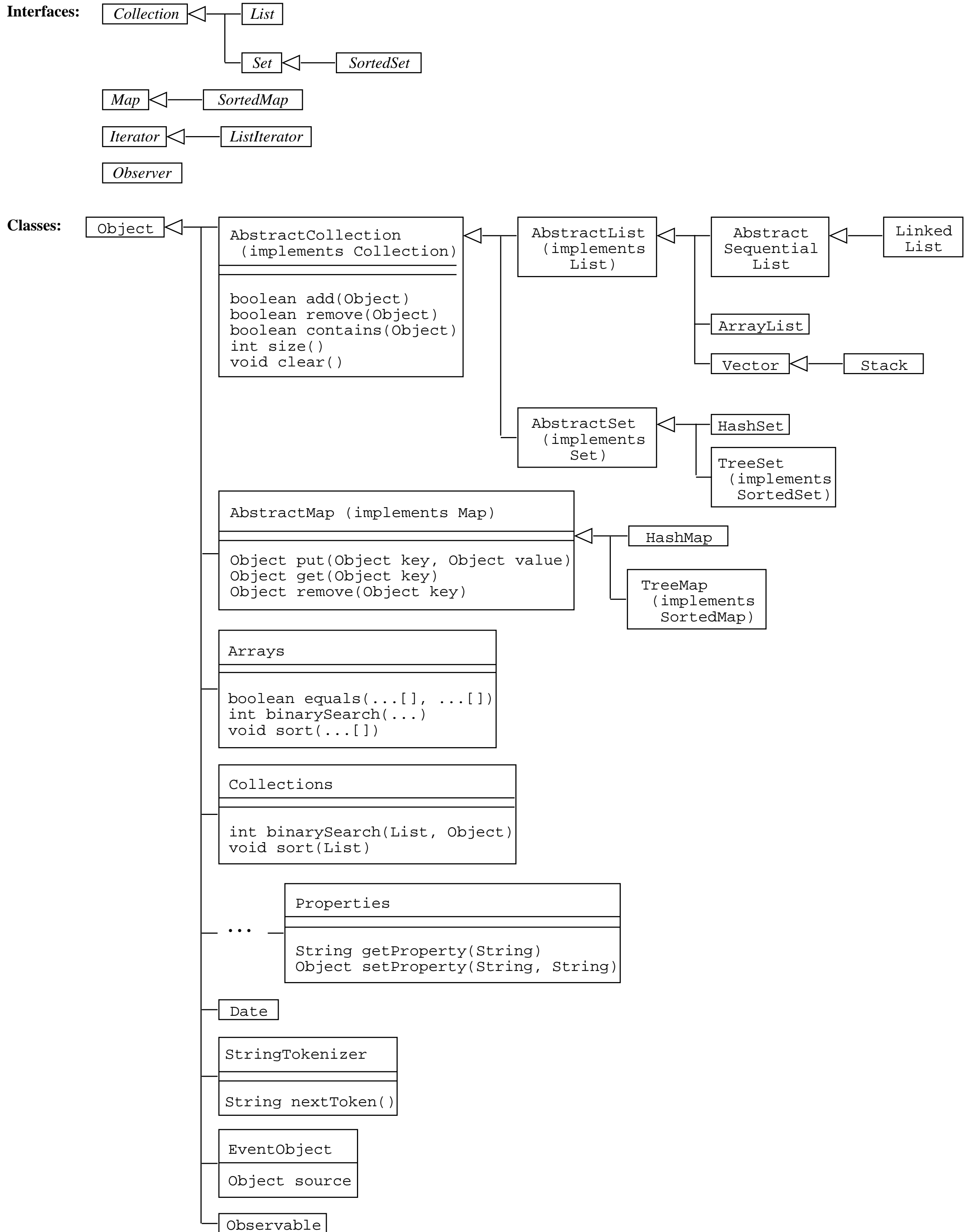
B. The functionality provided in these packages is central to the work you're doing in 309.

C. The classes and interfaces in these packages are summarized in the UML diagrams that follow.

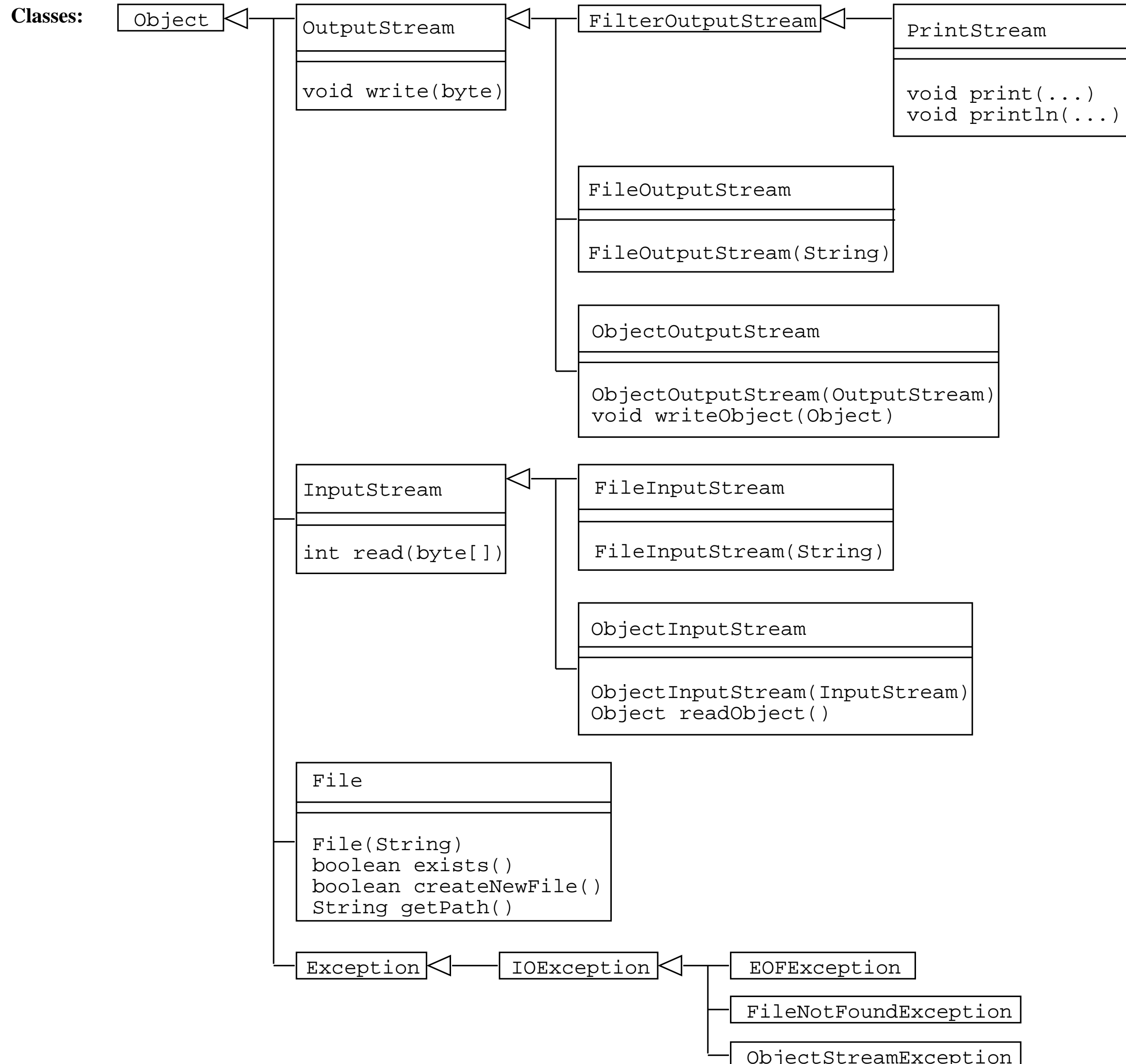
D. Package **java.lang**



E. Package **java.util**



F. Package **java.io**




III. **Using `java.io.File` and `javax.swing.JFileChooser` for platform-independent file access.**

A. Useful general operations of class `File` include:

1. canRead
2. canWrite
3. createNewFile
4. delete
5. isDirectory
6. isFile
7. mkdir

B. Package `java.io` provides very classes to read and write file data.

1. To enable a class to be written to and read from a file, the class implements the *Serializable* interface.

a. *Serializable* has no methods, so all you need to do is implement it.
b. Doing so is allows the class data to be "serialized" for sending to a file or any input/output stream.

2. For output, the classes are `FileOutputStream` and `ObjectOutputStream`
3. For input, the classes are `FileInputStream` and `ObjectInputStream`
4. These classes are used in conjunction with serializable objects.
5. E.g., suppose we have

```
public class SomeModelClass implements Serializable {

     ...
}
```

6. To write out such a model object do the following:

```
model = new SomeModelClass();

/* Put some data in model ... */

FileOutputStream outFile =
    new FileOutputStream(
        "model.dat");

ObjectOutputStream outStream =
    new ObjectOutputStream(outFile);

outStream.writeObject(model);
```

7. To read the object back in:

```
FileInputStream inFile =
    new FileInputStream("model.dat");

ObjectInputStream inStream =
    new ObjectInputStream(inFile);

model = (SomeModelClass)
    inStream.readObject();
```

C. javax.swing.JFileChooser provides a platform-independent UI for file selection; particularly useful methods:

1. showOpenDialog
2. showSaveDialog
3. getSelectedFile
4. setSelectedFile

## IV. Review of "canned" model data for initial testing.

A. For initial testing of a model/view design, it is very useful to have sample model data, that are displayed in corresponding views.

1. In the beginning, these data can be entirely "canned", i.e., defined as constants in the model classes.
2. A good place to get concrete example data values is from the requirements.

B. The model data can be delivered to the view using the same methods that will ultimately produce the real data.

1. For example, if an iterator method is used to access model data, the body of the method can produce the canned data.
2. If collection-valued data are returned by the model, canned data can be generated by temporary testing methods

C. Examples of the preceding two forms of canned data generation are illustrated in the Calendar Tool code

from the Week 3 notes.

1. The iterator methods in the `MonthlyAgenda` model class deliver a fixed month of data to the `MonthlyAgendaDisplay` view class.
2. In the `Lists` model class, there is a `generateSampleList()` method that generates a sample data value that appears in the requirements; this piece of data is returned by `Lists.viewAppointmentList()` to the `AppointmentsListDisplay` view class.

V. **View data collection and validation.**

A. When the user enters data in a GUI, a View class collects it in raw form.

B. For example, the `getText` method extracts the string data from a `JTextField`.

C. Once raw data are collected they are:

1. Converted by the model, from their raw view form into whatever form the model needs, e.g., a string-to-numeric conversion.
2. Validated by the model, based on preconditions to a model processing method.
3. Processed by the model as appropriate.

VI. **Using exception handling in a model/design, to perform data validation.**

A. There are a number of ways to perform input data validation in a model/view design.

B. In general, most if not all of the validation should be done by the model.

1. In jargon terms, we have a "smart model and stupid view".
2. What this means is that the view does not know anything "smart" about the data, in terms of its structure, or whether it's semantically valid.
3. The view is in charge of displaying data in the UI, and managing the event-based interactions with the user.
4. The model is in charge of storing the data, and managing all of the content-based access and manipulation of the data, including validation.

C. In a design that includes formally-specified of methods, a useful way to handle data validation is with exception handling

D. This style of design is discussed in the next several points of the notes.

VII. **Quick review of exception handling concepts.**

A. Normally, when a method is entered via a call, it exits by returning to the caller.

B. In a language with an exception handling mechanism, there is an "abnormal" way for a method to exit -- by throwing (a.k.a., raising) an exception.

1. Raising an exception allows a method to exit in a manner separate from its normal return-to-caller mode.
2. When an exception is thrown in a method, control does not return directly to the caller, but rather some other active method that catches (a.k.a., handles) the exception.
3. The catch can be in the immediate caller, but it is not limited to be.
4. The catch does have to be performed by a currently active method; i.e., a method that directly or indirectly called the method in which the exception occurred.

C. Different programming languages provide different styles of exception handling, and details differ widely between languages.

1. At the design level, we use an abstract graphical notation to depict exception handling.
2. At the implementation, we will use standard Java notation for exception handling.

VIII. **Design diagram notation for exception handling.**

A. In our high-level function diagram notation, exception handling is shown with labeled arrows leading into and out from a method.

B. Figure 1 is an excerpt from the 309 handout on the graphical modeling notation which illustrates the exception diagramming format.

1. In the Figure, `MethodX` calls three submethods `X1`, `X2`, and `X3`.
2. `MethodX2` and `MethodX2` return in the normal way.
3. `MethodX1` can return in the normal way, but also throw an exception that is caught by `MethodX`.

C. This notation is used in the example diagrams that follow.

IX. **An example of data collection, exception handling, and related Model-View communication.**

A. Figure 2 illustrates the use of exception handling to validate user input data, in a Calendar Tool `ScheduleEvent` dialog.

B. The figure shows the model method `Schedule.scheduleEvent` throwing an `ScheduleEventPreCondViolation` exception to the companion view method `OKScheduleEventButtonListener.actionPerformed`.

C. The exception is thrown when one or more input errors is detected, based on the processing done the the `scheduleEvent` method.

D. What follows is the code for `OKScheduleEventButtonListener.actionPerformed()` and friends, corresponding to the design shown in Figure 2.

```
public class OKScheduleEventButtonListener implements ActionListener {

    /**
     * Construct this with the given Schedule model and parent dialog view.
     * Access to the model is for calling its scheduleEvent method.  Access to
     * the parent view is for gathering data to be sent to scheduleEvent.
     */
    public OKScheduleEventButtonListener(Schedule schedule,
            ScheduleEventDialog dialog) {
        this.schedule = schedule;
        this.dialog = dialog;
    }

    /**
     * Respond to a press of the OK button by calling ScheduleEvent with a new
     * Event.  The Event data are gathered from the JTextFields and JComboBox
```

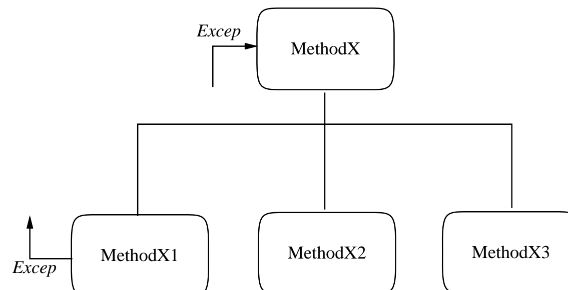



**Figure 1:** Event diagramming notation.

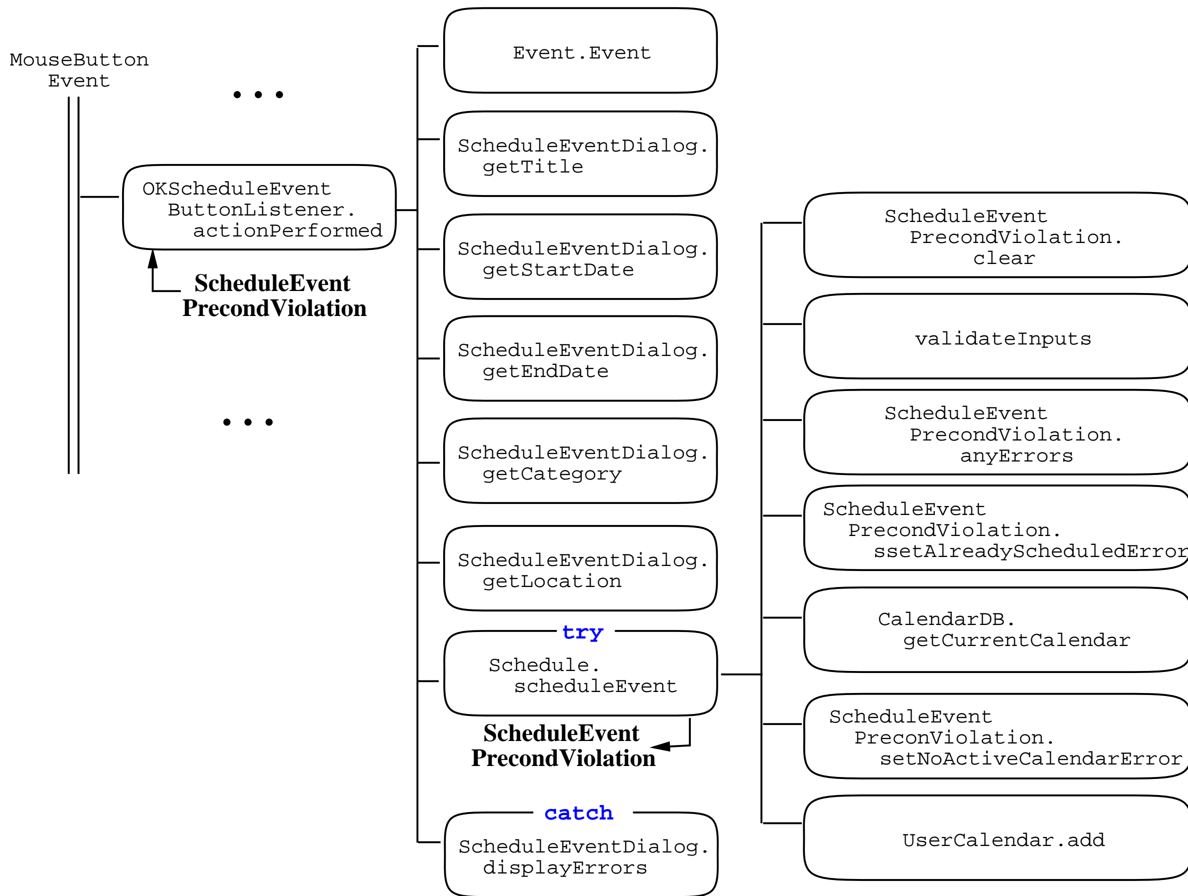



**Figure 2:** User input data collection and validation for scheduling an event.

```
     * in the parent dialog.
     */
    public void actionPerformed(ActionEvent e) {

        try {

            schedule.scheduleEvent(
                new caltool.schedule.Event(
                    dialog.getTitle(),           // Title as a string
                    dialog.getStartDate(),       // Start date as a Date
                    dialog.getEndDate(),         // Start date as a Date
                    dialog.getCategory(),        // Category as a Category
                    dialog.getLocation()         // Location as a string
                )
            );
        }
        catch (ScheduleEventPrecondViolation errors) {
            dialog.displayErrors(errors);
        }

    }

    /** The companion model */
    protected Schedule schedule;
```

```
    /** The parent view */
    protected ScheduleEventDialog dialog;

}
```

E. Here is the code for companion model method that potentially throws the exception.

```
public class Schedule extends Model {

    ...

    /**
     * ScheduleEvent adds the given Event to the given CalendarDB, if an event
     * of the same start date and title is not already scheduled.
     *
     * pre: Details in next week's notes
     *
     * post: Details in next week's notes
     */
    public void scheduleEvent(Event event)
            throws ScheduleEventPrecondViolation {

        /*
         * Clear out the error fields in precond violation exception object.
         */
        scheduleEventPrecondViolation.clear();

        /*
         * Throw a precond violation if the validly check fails on the start or
         * end date.
         */
        if (validateInputs(event).anyErrors()) {
            throw scheduleEventPrecondViolation;
        }

        /*
         * Throw a precond violation if an event of the same start date and
         * title is already scheduled.
         */
        if (alreadyScheduled(event)) {
            scheduleEventPrecondViolation.setAlreadyScheduledError();
            throw scheduleEventPrecondViolation;
        }

        /*
         * Throw a precond violation if there is no currently active calendar.
         * Note that this condition will not be violated when interacting
         * through the view, since the 'Schedule Event' menu item is disabled
         * whenever the there is no active calendar.
         */
        if (calDB.getCurrentCalendar() == null) {
            scheduleEventPrecondViolation.setNoActiveCalendarError();
            throw scheduleEventPrecondViolation;
        }

        /*
         * If preconditions are met, add the given event to the currently
         * active calendar.
         */
        calDB.getCurrentCalendar().add(event);
```

```
    }
}
```

F. Here is the code for the ScheduleEventPrecondViolation

```
package caltool.schedule;

import caltool.PrecondViolation;
import java.util.*;

/****
 *
 * Class ScheduleEventPrecondViolation defines and exception containing error
 * conditions for the Schedule.scheduleEvent method.  It contains a list of
 * the specific error messages that may be output in response to a precondition
 * having been violated by a call th scheduleEvent.
 *
 */
public class ScheduleEventPrecondViolation extends Exception
        implements PrecondViolation {

    /**
     * Construct this by initializing the error message list to an empty list,
     * initializing the numErrors count to 0, and initializing local copies of
     * the error message text for each of the possible errors from
     * Schedule.scheduleEvent.
     */
    public ScheduleEventPrecondViolation() {

        errors = new ArrayList();

        alreadyScheduledMessage = new String(
           "An event of the given start date and title is already scheduled.");
        invalidStartDateMessage = new String(
            "Invalid start date.");
        invalidEndDateMessage = new String(
            "Invalid end date.");
        noActiveCalendarMessage = new String(
            "There is no active calendar in the Calendar Tool workspace.");

        numErrors = 0;
    }

    /*-*
     * Implemented interface methods.
     */

    /**
     * Return the error list.
     */
    public String[] getErrors() {
        return (String[]) errors.toArray(new String[1]);
    }

    /**
     * Clear all error messages.
     */
    public void clear() {
        errors = new ArrayList();
        numErrors = 0;
    }
```

```
/**
 * Return true if any errors have been set.
 */
public boolean anyErrors() {
    return (numErrors > 0);
}

/**
 * Return the number of errors.
 */
public int numberOfErrors() {
    return numErrors;
}


/*-*
 * Error-setting methods
 */

/**
 * Set the already scheduled error message.
 */
public void setAlreadyScheduledError() {
    errors.add(alreadyScheduledMessage);
    numErrors++;
}

/**
 * Set the invalid start date error message.
 */
public void setInvalidStartDateError() {
    errors.add(invalidStartDateMessage);
    numErrors++;
}

/**
 * Set the invalid end date error message.
 */
public void setInvalidEndDateError() {
    errors.add(invalidEndDateMessage);
    numErrors++;
}

/**
 * Set the no active calendar error message.
 */
public void setNoActiveCalendarError() {
    errors.add(noActiveCalendarMessage);
    numErrors++;
}


/*-*
 * Data fields
 */

/** List of current error messages */
protected ArrayList errors;

/** Error message count */
protected int numErrors;
```

```
    /** Error message for event of same date,title already scheduled */
    protected String alreadyScheduledMessage;

    /** Error message for invalid start date */
    protected String invalidStartDateMessage;

    /** Error message for invalid end date */
    protected String invalidEndDateMessage;

    /** Error message for no currently active calendar in the workspace */
    protected String noActiveCalendarMessage;

}
```

G. Finally, here is the code for the general interface that all precondition violation exceptions must implement:

```
package caltool;

/****
 *
 * Interface PrecondViolation defines the methods that all precondition
 * violation exceptions must implement.
 *
 */
public interface PrecondViolation {

    /**
     * Return the concrete error list for precondition violation.  Each
     * position in the list corresponds to violation of a particular
     * precondition clause.
     */
    public String[] getErrors();

    /**
     * Clear out all of the error messages in this.
     */
    public void clear();

    /**
     * Return true if one or more error messages has been set.
     */
    public boolean anyErrors();

    /**
     * Return the number of error messages.
     */
    public int numberOfErrors();

}
```

X. **A note on model-view communication via direct data reference versus dynamic method computation.**

A. Consider the Calendar Tool view for an individual scheduled item versus a monthly view of items.

1. In the case of a single item view, the display shows data that are persistently stored within the model data.
2. In the case of a monthly view, some of the data are persistently stored, but the other parts of the data, in particular the month/day layout, are dynamically computed.

B. These are examples of a data persistence design pattern we'll discuss further in upcoming lectures.

## XI. A key design pattern for use in 307 projects -- Observer/Observable

A. Useful when multiple views change, based on changing model, e.g.,

1. ***CalTool:*** daily, weekly, monthly views
2. ***Testtool:*** question dialogs, question DB, tests

B. In Java, this pattern is defined with the *Observable* class and the *Observer* interface, which are summarized as follows:

```
interface Observer {
  public void update(
    Observable o,
    Object arg)
}

class Observable {
  void addObserver(Observer o)
  void setChanged()
  boolean hasChanged()
  void notifyObservers()
  void notifyObservers(Object arg)
}
```

C. At the top level of the design, the MVP `View` class implements `Observer` and the MVP `Model` class extends `Observable`:

```
public class View implements Observer {

        . . .
}

public class Model extends Observable implements Serializable {

        . . .
}
```

D. Here's an example of typical use in the Calendar Tool, where a `MonthlyAgenda` view observes a `User-Calendar`, so that the agenda display will be automatically updates whenever the calendar changes.

```
public class UserCalendar extends Model {

     . . .

    public void add(ScheduledItem item) {

        . . .

        items.add(item);
        setChanged();
    }
}

  . . .

public class OKScheduleEVentButtonListener implements ActionListener {

    public void actionPerformed() {

        try {
            schedule.scheduleEvent(
                new caltool.schedule.Event(
                    ...
                )
            );
```

```
        }

        . . .

        schedule.notifyObservers();

        . . .
    }
}

 . . .

public class MonthlyAgenda extends View {

    public MonthlyAgenda(caltool.view.View view, UserCalendar userCalendar) {

        . . .

        userCalendar.addObserver(this)
    }

    public void update(Observable o, Object arg) {

        /* Get items for this month from MonthylyAgenda model and update display */

    }
}
```

E. The key aspects of this example are the following:

1. Observable classes extend `java.util.Observable`. This extension can be direct, or in the example above, indirect via `mvp.Model`.
2. Observer classes implement `java.util.Observer`. This implementation can be direct, or in the example above, indirect via `mvp.View`.
3. The constructor of an observer class calls `addObserver` on all the model classes it wants to observe.
4. In observable model classes, all mutating methods call `Observable.setChanged` when they perform a mutation.
   a. A *mutating* method is any that changes the state of the data in its class.
   b. In the above example, it's the `UserCalendar.add` method, which adds an item to a calendar.
   c. Examples of other mutating methods are those that delete or modify items from the calendar, i.e., `UserCalender.delete` and `UserCalender.change`.
5. In companion view classes, the `actionPerformed` method that calls a mutating model method subsequently calls `Observable.notifyObservers`, when the mutating methods succeeds.
   a. This separation of calls to `setChanged` and `notifyObservers` enforces a clean separation of model/view duties.
   b. The model calls `setChanged` in all of its mutating methods, but not `notifyObservers`.
   c. The view calls `notifyObservers` after a successful call to a mutating model method.
   d. By not calling `notifyObservers`, the model maintains its independence from its views, and other classes that are observing it.
   e. By not calling `setChanged`, the view maintains its independence from the model classes that know when and when not to signify changes in the state of model data.
6. In observing view classes, the `Observable.update` method is invoked via `notifyObservers`; when invoked, `update` calls appropriate model methods to query the model state and access model data.