CS193J Handout #7
Summer, 2003 Manu Kumar

HW 1: Pencil me in![*]

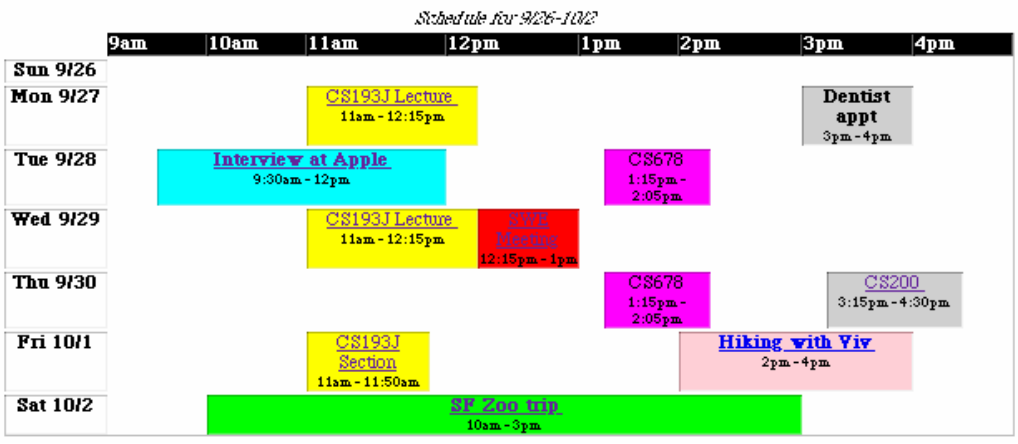
(This assignment was created by Julie Zelenski. It was updated by Nick Parlante. It was most recently updated by Manu Kumar.)

This assignment stresses the basic steps in OOP design – solving a large coding problem by dividing it up into several smaller classes. This is not a trivial little assignment. It's big enough to have some real substance to it. The assignment is due before midnight (11:59 PM) on Wed, July 9th, 2003.

With all the commitments you have vying for your attention, it's no wonder you're not getting enough sleep and are having trouble perking up for your (fascinating) Java class, not to mention accidentally scheduling your doctor's appointment during your PoliSci midterm, and calling your mom two days late to wish her a happy birthday. Did your new quarter resolutions contain such tried-and-true entries as "better time-management" and "be more organized" (not to mention "bring more chocolate to my favorite CS instructor")? Now with the miraculous power of Java in your life, you're ready to write the calendar keeper to make these sort of mishaps a thing of the past and reform yourself into an super-charged impressively organized wonder for all to behold. The goal of this assignment is to refresh your programming and data structure design skills and give you some experience with the syntax and features of Java as well as basic object-oriented programming. You will make use of several Java built-in classes (String, Date, collections, I/O classes) and design some of your own objects. We will give you a few pre-defined objects and you will write a few additional objects of your own to coordinate with them.

A brief overview

Pencil Me In is a program designed to read a file containing a calendar of events and generate a weekly schedule in HTML format suitable for framing, or even better, posting to your personal web page so all can keep track of your busy schedule. A sample HTML table is shown below:



[*] Okay, it's actually a plagiarized name. Pencil Me In is the name of one of the first serious productivity applications written in Java, (although originally was written for NeXTstep!). It came from Sarrus Software, a startup founded by friends of mine, which was acquired by Javasoft. Go Andy and Brian! And while we are thanking folks, thanks to Owen Astrachan at Duke for the idea of generating HTML schedules.

The program itself is operated from the command-line. The user is prompted to enter the name of the file to read events from, a few questions are asked to configure the desired output, and the program outputs an HTML file containing the schedule. The user can choose to generate schedules from this and other files until they are done. Here is a script from a sample run of the program:

```
Welcome to the Pencil Me In datebook program.
This program reads in event files and generates HTML weekly schedules. For
convenience, when you are asked a question, the answer given in [brackets]
is the default used if you don't choose to enter a response.

Enter name of file to read [events.txt]: sample.txt

Now ready to output HTML schedules.
Enter start date for schedule [9/27/99]: 9/26/99
Do you wish to create a table or list? [table]:
Enter name of file to output [table.html]: fall.html
Wrote weekly schedule to file.
Do you wish to create other schedules from this same file? [n]

Do you wish to read another event file? [n]
Goodbye!
```

Events and the file format

A personal datebook consists of a collection of events. An event represents an activity scheduled for a particular time interval on a given day and includes information about what the event is, a color, and an optional URL for more information. Events can be of two different varieties: regular or one-time. A regular event is one that is scheduled every week on the same day or days, such as a lecture that meets every Monday and Wednesday. A one-time event is scheduled on a particular date, such as 10/15/99, and doesn't repeat. In the previous sample table, one-time events were printed in boldface to distinguish them from regular events.

The input to Pencil Me In is a text file of events. Each event takes up two lines. On the first line is the name of the event, the second line gives the information about when and what in this format:

```
Name of Event
DaysOrDate StartTime EndTime Color URL
```

Name of Event The first line contains the name of the event, it may contain spaces.

DaysOrDate For a regular event, this will be a sequence of day abbreviations (SMTWHFA, upper or lower case). The days can be in any order. For a one-time event, this is a date in the format m/d/y, e.g. 6/12/99. There can be no spaces within a DaysOrDate string.

StartTime The time the event begins. Two time formats are accepted, either military 24-hour time or 12-hour time with am/pm. The minutes can be left off, in which case it is assumed they are zero. Thus, acceptable times would be 3:15pm or 4pm as well as 15:15 or just 16. There can be no spaces within a time string.

EndTime The time at which the event ends, using same format as start time.

Color	One of the 16 pre-defined HTML colors (black, white, red, green, blue, yellow, purple, silver, lime, olive, maroon, navy, teal, fuchsia, aqua, gray). The color is used for the font (list) or cell background (table) for this event.
URL	A URL to link to for more information on this event. If there is no link for this event, the field may be absent entirely. A URL cannot contain spaces.

Any line in the event file that is empty or that begins with a # character should be treated as a comment line and ignored. Here is the event file that was used to generate the table on the first page:

```
# Fall quarter events
CS193J Lecture
MW 11am 12:15  yellow   http://www.stanford.edu/class/cs193j/
CS193J Section
F  11:00 11:50 yellow  http://www.stanford.edu/class/cs193j/
SWE Meeting
W  12:15pm 1pm   red   http://www.stanford.edu/group/swe/
CS200
H  3:15pm 4:30pm  silver http://cse.stanford.edu/class/cs200/
CS678
TH 1:15pm 2:05pm fuchsia
Interview at Apple
9/28/99  9:30am 12 aqua http://www.apple.com/
Hiking with Viv
10/1/99  2pm 4pm pink mailto:Vivian_Houng@srch.org
SF Zoo trip
10/2/99 10am 3pm lime http://www.sfzoo.com/
Dentist appt
9/27/99 3pm 4pm silver
```

Each of the event fields on the second line cannot contain embedded spaces, but there may be arbitrary white space and tabs between the fields. The recommended way to separate the line into fields will be to use the java.util.StringTokenizer class to break the line apart into tokens and then convert each token to a time or date as necessary.

HTML output

After parsing the file, your program will allow the user to output one week's schedule based on the events read in from the file. Your program should prompt the user to choose the starting date for the schedule. The user is also allowed a choice between output formats. Your program should support both a list format and a table format. And finally, the user is allowed to choose the filename for the output, the default is table.html or list.html, depending on the format chosen. Both formats show all of the regularly scheduled events as well as any one-time events that fall in this week. A sample table was shown earlier; the list format for the same event file would look like:

For the list format, the days of the week are listed chronologically and the events within a day are given in time order. Each event is prefaced by its time interval, its name is printed in the event's chosen color, and link to the optional URL attached to the event name. The one-time events are printed in bold to distinguish them from the regular events.

For the table format, the hours of the day are the columns going from left to right across the top and days of the week are the rows. Rather than always spanning the same fixed time range, the table must be customized for the week being displayed. The table starts at the beginning of the earliest event occurring in the week and stops at the end of the latest (you can round outward to multiples of an hour for start and stop if you like). Along the top, you should provide markers for the time intervals, every hour or half hour or so should be enough. Along the left, each row is labelled with the date. Within a day, the events are drawn as boxes to indicate the span of the event's time interval. The size of the box should correspond to the event duration. It is acceptable to round the sizes to small multiples if you find that easier to manage (i.e. rounding the event start and stop time to the closest 5-minute boundary would be okay). The boxes should be colored in the event's color, the event name and time interval are printed, and the link to the optional URL is attached to the name. Again the one-time events are printed in bold to distinguish them from the regular events (or at least in some way distinguished).

The list format is certainly the easier format, so we definitely recommend you start with that one. Once you have that working, move on to the more sophisticated table output. You can construct the table either as one large table where the events are configured as cells that span multiple columns or as an arrangement of nested tables, where each daily row is a table itself within the larger table. Either will work and both have their advantages and disadvantages.

HTML advice

Most of you have probably had experience with HTML, but for those of you who are unfamiliar with the details of lists and tables, we've added a link from the "Other materials" section of our web site with pointers to various HTML references on-line. Also in the other materials section, there are pointers to a sample output pages. Using the View Source option of your browser, you can examine the HTML we generated to help you plan your strategy.

Your output does not need to exactly match ours. You are welcome to make your own choices for fonts, sizes, layouts, borders, alignment, etc. to design an arrangement that you find visually pleasing. It needs to meet the basic requirement of properly arranging the events into the weekly schedule with colors and links (and with some visual distinction between the regular and one-events) but beyond that the aesthetic choices are up to you. Some examples: You are free to select the font styles and sizes you like for the different components. You can choose to make the rows and columns of a fixed height or allow the browser to apportion the space depending on need. You can choose how to represent a day that has no events scheduled (as an empty normal-height row, as a thin row, with some indication of "no events", whatever you find appealing). And so on…

Interacting with the user

The program has a simple command-line interface to allow the user to select a file to read and configure the output schedule. The interaction should be very fault-tolerant, given the sloppy nature of human typing. For example, if the user's choice of file cannot be opened, prompt again until they provide a valid entry. Similarly for choosing the starting date or other output parameters, you should gracefully handle invalid responses and ask again until you get an acceptable answer. For each question asked, there should be a default response offered and used when the user simply hits return. Our SimpleInput class (described below) offers help with these tasks.

Although we are accepting of variations in the HTML output, we expect your handling of user interaction to mimic the one given on page 2 of the handout to facilitate testing.

Input and output

You will need to use Java's I/O classes for this program. Unfortunately, these classes require the use of some language facilities (exceptions) that we won't get to until later in the quarter. To smooth this over, we have provided you with some wrapper classes to help handle these cases for you. Below is a very brief overview of the three classes. The provided source files are documented (in javadoc format) with more details on the methods and usage for each class.

The `SimpleInput` class provides a simple mechanism for reading input from the user. It has methods reminiscent of the CS106 simpio library such as `readLine()`, `readInteger()`, `readYesOrNo()`, etc. that prompt the user, check for correctly formatted input, and force the user to re-enter when needed.

`SimpleFileReader` is a small class that can open a file for reading and read its contents line-by-line. It works entirely in terms of strings. If you need to convert the lines into other formats, use string manipulations such as separating the lines into tokens using the StringTokenizer, or using the Integer class to convert strings to integer, etc.

`SimpleFileWriter` is another small class that opens a file for writing and can print to it using print and println methods like the standard System.out stream.

We may have a chance to discuss I/O in more detail at some later point, but mostly you will be responsible for reading up on this (in textbook or web or wherever) to pick up the myriad details as needed. Handling I/O is one of the more uninteresting language features— every language

does it, each is different in annoying ways, and there are always dozens of little details to absorb. We recommend the approach of looking up the details on a "need to know" basis.

Conversion utilities

Another class that we provide to you is the `Convert` class that offers a few routines to convert strings to dates and times. Like the I/O classes, we give this to you because it requires exception handling. All the operations are static methods (since they are effectively just functions) that you invoke on the Convert class itself, a few examples are shown below. The source file is documented (in javadoc format) with more details on the methods and usage for each method.

To convert strings to dates and vice versa:

```
Date today = Convert.stringToDate("10/1/99");
System.out.println("It is " + Convert.dateToString(today));
```

The dateToString operation is overloaded to accept custom format strings:

```
System.out.println("It is " + Convert.dateToString(today, "E M/d"));
```

To convert strings to times:

```
Time noon = Convert.stringToTime("12pm");
Time late = Convert.stringToTime("23:55");
```

Formatting output

In order to do formatted output (decimals to 2 places and the like), the Java strategy is to use a "formatter" object to convert the value into a formatted string before printing. The formatting classes are contained in the `java.text` package and are documented in the on-line class specifications and briefly at the end of Chapter 24 of your text. The Java formatters use patterns to describe the desired formatting that are similar but not the same as `printf`. Here are few simple examples that give you a taste:

To print integer numbers with a minimum of 4 digits and pad with zeros when necessary:

```
import java.text.DecimalFormat;

DecimalFormat formatter = new DecimalFormat("0000");
String formattedNum = formatter.format(someNumber);
```

To constrain a floating point number to at most 3 decimal places, but less if not needed:

```
DecimalFormat formatter =  new DecimalFormat("#.###");
String formattedNum = formatter.format(someNumber);
```

To print short dates in the form "Wed 1/13":

```
import java.text.SimpleDateFormat;

SimpleDateFormat formatter = new SimpleDateFormat("E M/d");
String formattedDate = formatter.format(someDate);
```

Requirements summary

Because my assignment handouts tend to run long and contain lots of details, I've been trying a new strategy of summarizing the assignment requirements here at the end as a check-off list to help ensure you haven't missed anything important. I hope you find this useful.

General

- Your source files should be easily readable on UNIX— i.e. end-of-line characters should be proper, lines shouldn't wrap in obnoxious places, etc. This is of particular importance to those of you moving your files to Solaris from elsewhere.
- The submitted project should include all necessary files. We should be able to issue the command `javac *.java` and all files should compile cleanly (i.e. with no warnings).
- Your main class should be named PencilMeIn and should require no command-line arguments. After compiling, we should be able to run your program with the command `java PencilMeIn`.
- The program should run as a Java application, not an applet.
- It should read in event files and output complete html files that can be loaded into Netscape, Internet Explorer, etc.

User interaction

- The command-line interface should mimic the interaction shown on p.2 of the assignment handout.
- It should offer default responses to all the questions. The default input file is "events.txt", the default starting date is today's date, the default output format is "table" and the default filename will be "table.html" or "list.html", depending on the format chosen by the user.
- If the user enters an improper response (bad date, non-existent file to read, etc.) you should re-prompt until they enter something valid.
- If the user chooses to output using a filename that already exists, your program should silently overwrite the previous file.
- After creating one schedule, your program should ask if the user wants to create another schedule using the same event file. Your program should not re-read the event file between generating sucessive schedules.
- Once the user has generated all the schedules they desire from an event file, your program should ask if they would like to read a new event file to generate schedules from.

File reading

- Your program must handle all files that are properly formatted in accordance with the specifications given on p. 2-3 of the handout. Please read the details there carefully to make sure you handle the few allowable variations such things as upper or lower case in the days strings, the optional URL, etc.
- Your program does not need to deal with incorrectly formatted files (those that are not proper event files, contain invalid dates or times, are missing essential fields, etc.). Although it would be nice to gracefully handle improper input and it would make your program more robust in the long run, we will not expect it to do so and we will not test on malformed files.
- Blank lines or lines that have a first character of # are comments and should be ignored.
- You may assume that each event will have a proper time interval (i.e. stop time is after start time) and no event can extend from one day into the next.
- You may assume that no events will overlap. Although it wouldn't matter for the list output, constructing the table becomes quite messy if events have to occupy the same time

slot on the same day. However, one event can end at the exact time another event starts and that is not considered overlap.

HTML output

- For both formats, the schedule should show one week's worth of events starting from the user's chosen date.
- Note that the week does not always start on Sunday, it can start on any day of the user's choosing. If the user enters a starting date that is a Wednesday, the schedule will start there and extend to the following Tuesday.
- All events that fall during the week, both regular and one-time, should be included. Any one-time events that occurred before or after that week are effectively ignored for this schedule.
- For the list format, it should pretty much mimic the output shown on p.3.
  - —The days are listed chronologically.
  - —The events within the day are listed in time order.
  - —The time interval and event are printed, if present, the optional URL is attached as a hyperlink anchored to the event name.
  - —The event name is printed in its color.
  - —One-time events are printed in bold to distinguish them from regular events (or in some way distinguished).
- For the table format, more variation is possible, but there are requirements that apply to all programs:
  - —The times go across the table chronologically.
  - —The table does not have a fixed start & stop time. The first column should be the beginning of the earliest event occurring in this week and the last column at the end of the latest (you can round outward to multiples of an hour for start and stop if you like).
  - — The days are marked along the left side of the table.
  - —Each event should be drawn in a box positioned in the right place for its scheduled time and sized to the event's duration. (you can round to 5-minute multiples if you like).
  - —The box is colored with the event's color.
  - —The boxed text consists of the time interval, the name, and the optional URL as a hyperlink anchored to the event name.
  - —One-time events are printed in bold to distinguish them from regular events (or in some way distinguished.

Grading

The bulk of the score for this assignment comes from evaluating its functionality. Functionality covers your program's behavior from an external perspective. Without looking at the code, does it work as expected? This is usually tested via scripts to run your program through its paces and determine if it correctly handles the requirements, including any specified error conditions. Although figuring much less prominently in the grade, we also expect that your programs will be cleanly written and easy to understand. Your program should exhibit decent sensible coding practice, such as appropriate algorithm choices, effective decomposition, no duplicated code, as well as things such as appropriate

commenting, well-chosen identifiers, consistent indentation and capitalization, and most importantly, good object encapsulation. If your program fails to work on any of the functionality tests and we go hunting in your source code to understand the problem, how easy your code is to understand will have a large effect on our ability to give partial credit.

Getting started

We have a class directory on leland **/usr/class/cs193j/** where we will place materials for the assignments. For this assignment, there is a project directory `hw1` that contains the code for the classes we give you. You want to make a copy of the directory (i.e. `cp -r /usr/class/cs193j/assignments/hw1 ~`) to get started. There is also a link to this directory from our web page. You are welcome to build the project on the platform of your choice, just be sure to leave a little time to test it back on leland to make sure it works there too. No matter where you do your development work, when done, you must be sure your project will compile and run on the leland workstations. All assignments will be electronically submitted there and that is where we will compile and test your project. All students (SITN, local, remote, or otherwise) will use the same process to submit assignments.

Electronic submission

Our goal is an entirely paperless class, so you will not submit printouts, but instead use a submit script to electronically deliver your entire project directory. Some advice— leave enough time before the due time to run the submit script and deal with any submission problems. This goes triple for those doing your work on another platform. You will need time to move the files over, recompile them on Solaris, re-test the code, remove the .class files, and submit. You may want to do a trial run of compiling and submitting on Solaris in advance to become familiar with the process and avoid last minute panic.

The submit script is located at `/usr/class/cs193j/bin/submit`

Instructions for submitting are provided in README-submit in the same directory. Please be sure to read the instructions before attempting to submit. (Instructions are not copied here since they are common for all homework submissions)

Late days

Refer to handout #1 or the web site policy page for the course policy on late work. If you are choosing to use one of your self-granted extension days, you do not need to confirm with us, just submit your work using the same process and it will be time-stamped accordingly.

P/NC students

Those of you taking the course with the P/NC grading option only need to implement the list format and do not need to implement the HTML format.

HW1 Design Ideas

The rest of this handout is a road map with suggestions about how to work through the first homework project. It sketches out some strategies for how to break down the task at hand, the classes to consider building, and some milestones to work toward in your design. You can take our ideas and implement them as we have suggested or choose a different path of your own.

Designing classes

An object-oriented program is decomposed into a number of cooperating classes. The handy thing about OO programs is that the data structure can often cleanly model the real-world ideas being represented. As the designer, your job is to determine what objects you are trying to model and what sort of operations these objects perform. To define a class, you need to answer two fundamental questions:

- What data is required to represent this object?
- What behavior will I need from this object?

The answers to these questions tell you what instance variables you need and what methods the class must implement. Behavior that is associated with an object should be provided as a message you can send to the object to ask it to perform some action, rather than having the client reach into the object and muck around with its data. For example, a client asks a Time object to find out whether it is before another Time object. If you want an Event printed, you send a message to that Event object asking it to print itself.

To introduce you gently to this process, only a few simple classes are needed for this program. These objects include the Time, Event, DailySchedule, and Datebook classes, along with a few helper-utility classes and routines. Each of the main classes has a pretty clear real-world analog to help guide your design and the relationships between the classes are not too complex. Let's give you an overview of each of the classes so you'll know how to proceed.

The Time class

First consider the Time class we used in lecture as our first simple object. It has the straightforward job of representing a particular time in the day. In the starting project, we give you a primitive version of this class, but you'll need to further develop the class to be fully useful.

The Time class is your first opportunity to work through designing a useful and robust object with a sensible and complete interface. Carefully think through your decisions and make choices of which you can be proud. It's not a very complex class, so it is a good one to tackle early, so you have a simple introduction to designing and manipulating data in the object-oriented paradigm.

First, evaluate the current interface of the Time class. Is it missing important features? Does it have unnecessary functionality you want to remove? Think through the rest of the program and what operations you will need to manipulate times and plan to include them as methods for the Time class. Feel free to remove any current methods that are redundant or unnecessary in your

design. Be sure to only make those things public which make sense as part of the Time's external interface and which you are committed to supporting forever.

This Time object has the hour/minute/am representation that is intuitive and familiar, but it can be awkward to manipulate in that form. What other representations might work? What are the tradeoffs in terms of space and convenience of these other choices? Consider what operations are easier for a given representation and which ones are more difficult. Think through these before committing on your representation and then go forth and complete the implementation.

Instead of moving on, write some simple test code that exercises the Time class and allows you to find and correct any problems now. Can you reliably create Times, message them, print them, compare them, shift them, etc. using your operations and get the correct results? It's much easier to isolate and fix bugs when you're just dealing with one class at a time than trying to sort out everything at once when you have lumped everything together into a bunch of untested classes.

One convenient place to put your test code is in a static `main` method on the class itself. You can then compile and execute that class itself. You don't even need to remove the testing code before submitting unless it really interferes with the overall readability.

The Event class

Now consider the Event class. An Event object encapsulates all of the details for a particular datebook entry. An Event needs to track its name, color, and url. All three of these fields can easily be represented using Strings. An Event also needs to track its time interval and perhaps its date or days and whether it is regular or one-time event. In truth, the Event class is not that much more sophisticated than a C struct, but it serves to encapsulate the data into a clean abstract unit. A good milestone to aim for before moving on from Event is that you are able to read the events from the file, printing each out as read, to verify that the Event class is doing its job.

The TimeInterval class

Managing the time interval for an Event is little more complex than the other simple String fields. You may want to create a helper class, TimeInterval, just to encapsulate this concept. I choose to do so and I found it a helpful abstraction, but your mileage may vary. It is certainly possible to just directly track the start time and duration in the Event object as an alternative.

Internally, a TimeInterval could store its data as an array of two Times, one start, one stop, or perhaps a starting Time object and duration. Or perhaps something else entirely. Wrestling with these sorts of decisions is the most interesting part of designing a class. As a client of the TimeInterval, you don't care how it is internally represented, all you care about is that you can get the right results when you ask it to print and whether it starts before another TimeInterval and so on. But as the implementor, how you choose to represent it can make quite a difference in terms of ease of writing the code, the resulting efficiency, the size of the object, how difficult it is to modify later, etc. In general, we encourage clean design and clearly written code that may be less efficient over the terse, complicated alternatives that are produced in the name of efficiency.

The DailySchedule class

To manage all the events scheduled on a particular day, you will create the DailySchedule class. Because the number of events is not known in advance or constrained to any fixed size, an array

is not appropriate, you will need use some sort of resizable collection to store the events. The java.util.Vector class has been around since 1.0 days and handles a simple, linearly indexed collection of objects. Java2 adds an entire family of new collections with more expansive and full-featured alternatives. The new collections are covered extensively in your text and we will briefly tour them in class, but since they rely on inheritance and interfaces, two concepts we won't cover until later, we think you'll find it simplest if you stick with the simple Vector class for this assignment. However, if you want to read ahead and do some independent investigation, by all means, you're welcome to use the newer facilities.

In order to facilitate orderly printing of the daily schedule, it is convenient to keep the vector of events sorted by start time. The DailySchedule class will include functionality to add events and print out the schedule in various formats. Writing this class is a good way to become familiar with the basic workings of the using collection classes, Vector and its ilk are heavily used in Java programming.

The Datebook class

And finally, you will create a Datebook class that tracks the schedules for all the various days. You will need to manage DailySchedules both for the regular events, as well as those odd one-time events scattered throughout the datebook. For the regular events, you might want an array of DailySchedules, one per day of the week. Those dates with one-time events will require their own separate DailySchedules. It is expected that most dates won't have any one-time events scheduled and thus, you should avoid wasteful storage. Rather than keep a large array with many null references or empty DailySchedules, you should create schedules only for those dates that have one-time events. Using the Date objects as keys and the associated DailySchedule object as the value will facilitate quick lookup of daily schedule by Date, and when no events have been scheduled, the lookup will return null to show there are no one-time events scheduled for that day. To associate a Date object with its DailySchedule, you will want to use one of the built-in key-value "map" objects. The long-standing Hashtable class is the easiest to make use of, so that is what we recommend. The new collections offers some fancier alternatives that would require a little more effort on your part to pick up, but are also fine choices to consider.

When you are printing out a week’s schedule, you will find that you need to display both the regular events with any one-time events that occur during that week. There are several ways to accomplish this task, some easier than others. You may want to think about it for a bit before making your decision about how to handle it.

A good milestone at this point would be producing the list output, which is quite a bit simpler than the table. Constructing the table will probably be the last task you complete for the program.

The built-in utility classes

In lecture and next week’s section, we may briefly touch on the built-in utility classes, such as String, StringTokenizer, Vector, Date, Hashtable and foreshadow a bit of the new Collections, but we will not cover these classes in detail. Between your textbook and the on-line class specifications (see link from "Other materials" on our web site), you have access to pretty solid documentation. In general, in this course, we will expect you to be fairly resourceful in researching features and usage of the built-in classes (i.e. we are not going to use lecture time to painstakingly go through the details of the standard Java classes). This assignment is a great first

step toward becoming familiar with how to explore the Java packages on your own. Feel free to ask questions if you find the documentation unclear.

A note about **deprecated**: Since the Java libraries are still evolving, you will notice at times the compiler or documentation will indicate a method or class is "deprecated". In moving from Java 1.0 to 1.1 to 2, some methods and classes were replaced with different and improved means of handling that functionality. The old classes/methods are still there but have been deprecated to show that their use is discouraged and they will eventually be removed from the libraries. As an example, the Date class changed quite a bit and much of the original Date class has been deprecated and replaced with facilities in the Calendar class. (There is more info about this in Chapter 16 of your text). We expect you to avoid using deprecated API and instead use its replacement. Feel free to ask if you need help sorting it out.

Some class design suggestions

A working program is definitely a good thing, but a truly worthwhile accomplishment is one that also excels in design and readability: is the program sensibly divided into classes? Are the classes themselves complete and clean? Do classes take care to maintain consistency and encapsulation of the object state? Are the identifier names well chosen? Are complicated operations broken up in helper methods to manage complexity? Would you want to take over a project that had this as its starting code base? Would you find it easy to extend the program to new functionality? …

Most of you are experts on the usual CS106/CS107 style, decomposition, and commenting standards. If you haven't taken those courses here, check the "other materials" part of the web site where I posted some style handouts from those courses that you might want to peruse.

Here's some specific issues to consider and basic rules of thumb that we believe in:

- All instance variables should be `private`. An object should tightly encapsulate its data and not allow outside access.
- If a client will need access to another object's instance variable, the class can provide a public accessor method (a "getter" such as `length()` or `getLength()`). However, be wary about handing out references into your internal data—e.g.. do you see why a stack object shouldn't hand out a reference to its Vector /array and have the client add elements to it? Instead the Stack can have a `push()` method which takes the client's element and adds it to the internal list itself.
- When needed, you can also provide a "setter" function for an instance variable. Be careful about this, just because you have a private instance variable `length` doesn't mean you have to have a method `setLength()`. For that example, most likely the length is changed as needed when elements are added or removed and shouldn't be externally settable. Only include the setter if the client's use will require it and there is no better way to provide that functionality. Take precautions in the setter function so that it cannot be used maliciously to corrupt the internal consistency of your object—i.e. don't allow a client to change a count to a negative number or something that would cause your object to get confused or misbehave.

- Most methods will be `public`, since they are usually intended for public use. However, any helper methods only for use of the class implementor should be `private`.
- Give responsibility for behavior to the object itself rather than manipulating it using setters/getters from the outside. For example, you want to include methods in the class which can construct a new object given its starting state or print the data out nicely rather than having some other object stuff the data in field by field or extract the data to print it.
- You'll note that object decomposition leads to a different sort of code structuring than you're used to. For example, in C, you would likely group all the printing functions into one unit. In Java, each object takes responsibility for printing its own data, which will have the effect of distributing the code for printing around various classes. This can be a little disconcerting. Although object decomposition is an effective tool for managing complex projects, this sort of consequence is one of the downsides to it.
- At times, you will encounter some code that doesn't fit into the object-oriented paradigm, for example, consider the "main" code that handles the interaction with the user. Don't let this get to you, sometimes the paradigm just doesn't quite fit the task. It's okay to add static methods to a utility class to handle this. Just write the necessary methods in a good readable style and organize them sensibly.