Control and Data Models

Learning Goals

You must be able to:

- ✓ **produce intra-method control-flow models (i.e., flowchart) from given Java source code**
- ✓ **produce inter-method control-flow models (i.e., call graph) from given Java source code**
- ✓ **produce a data model (i.e., UML class diagram) from given Java source code**
- ✓ **use a debugger to help produce control-flow models and data structure definitions from given Java source code**
- ✓ **produce a UML sequence diagram from given Java source code**

Understanding Source Code

Control and data models help a software developer comprehend existing source code and can be used to help design new code. Software developers seldom write systems from scratch. Most often, software systems are long-lived and a software developer joining a team will be contributing to an existing system. Even when the construction of a new system begins, the system will typically be built using libraries of existing software components. As a result, software developers spend as much as half their time or more reading and understanding existing source code.

Software developers use many different techniques to help understand a part of a large source code base. In this reading, we will consider how developers use control-flow and data models to begin to understand source code. The models presented in this section of the course are intended to get you started reading existing (simple) Java code. We will build and expand on these basic concepts in the remainder of the course.

As you read this chapter, be prepared to **not** understand all of what is happening in the code presented. The reading is meant to help point out those concepts you should be starting to try to understand. Through examples in readings, lectures and labs, we are incrementally going to introduce you to more and more of the Java programming language. You are expected to learn the details of each part we introduce on your own and we will be pointing out resources to help guide you in this self-directed learning. When you get stuck in your self-directed learning, you should not hesitate to ask questions of your instructors, TAs and on piazza. We are here to help guide you in how to learn a language so you can carry these skills forward to the next programming language you encounter. Watch piazza for postings about what you should be familiar with in Java at different checkpoints over the term.

We will start with a simple model that helps determine how to read code in one of the smallest units in a Java program, a *method*.

Control-flow Models

In a Java program, a *method* is the construct that describes a part of a computation to be performed. By calling methods in different ways and in different combinations, a program can support a range of computations and behave in different ways. The Java method shown below returns the value `true` if a specified year is a leap year according to the Gregorian calendar and the value `false` otherwise. The year of interest is passed to the method as a parameter named `year`.

```
/*
 * Determine whether a given year is a leap year according to the
 * Gregorian calendar.
 */
boolean isALeapYear( int year ) {
   // declare a variable for the value to be returned
   boolean isLeap = false;

   // if the year is divisible by 4 and not by 100
   // it is a leap year…
   if ( ( year mod 4 == 0 ) && ( year mod 100 != 0 ) )
       isLeap = true;
   else if ( year mod 400 == 0 )
       isLeap = true;

   return isLeap;
}
```

The code in the method is preceded by a comment (e.g., starts with /* and ends with */) that specifies the purpose of the method. Within the method's definition, lines starting with two slashes (e.g., //) are also comments.[1]

When this method is invoked to determine if a given year is a leap year, the flow of control in the computation starts at the first line of code in the method, where the `isLeap` variable is assigned `false` and continues through the method. The `isLeap` variable is used to store state as the method executes. To understand how the method works, it is helpful to consider the order in which statements of the method may execute. We refer to the order in which code executes as *control-flow*. When we consider how control flows within in a single method, we are considering *intra-method* control-flow.

[1] This code is simplified and will not run directly if typed into a Java compiler or environment.

When one method must call another method to perform a desired computation, we must consider *inter-method* control-flow.

Intra-method control-flow

The `isALeapYear` method includes two conditional statements that may alter which statements in the method execute depending upon the particular year being considered.

There are many different ways we might represent the intra-method control-flow for the `isALeapYear` method. We will use notation from a flowchart, an approach that has been used since the early days of computer science to plan programs. Figure 1 describes the four kinds of symbols that we will use in a flowchart for an intra-method control-flow model. Arrows connect these symbols to illustrate the flow of control.

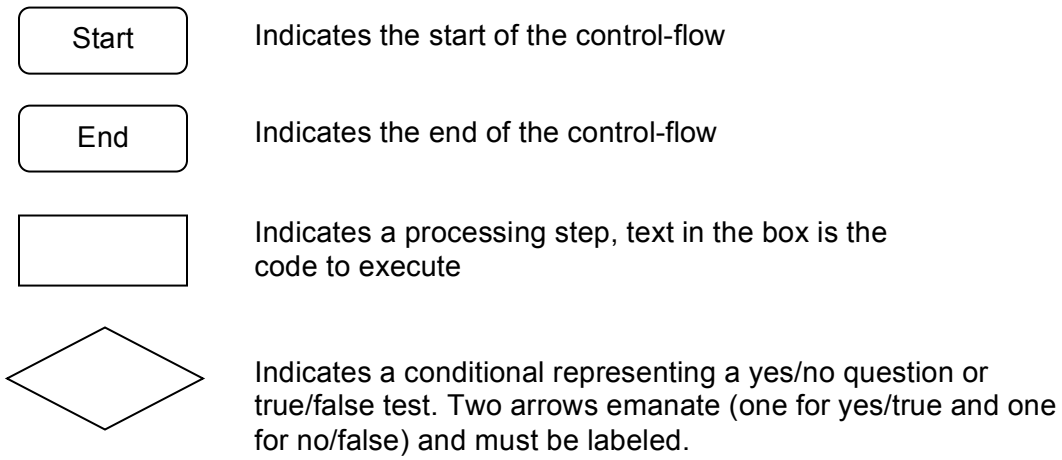


Figure 1 Flowchart symbols

Figure 2 shows a flowchart for the `isALeapYear` method. In this flowchart, the first statement that can and must be executed is the assignment of the value `false` to the variable named `isLeap`. As our purpose in creating the flowchart is to understand the order in which statements may execute, we could simply number the statements in the method and use the numbers in the nodes of the flowchart. Instead, we use abbreviated forms of the statements to make it easier to map between the code in the method and the flowchart. The next statement to always execute is the conditional that tests whether the given year is divisible by 4 and not by 100. The arrow connecting the node that assigns `isLeap` the value `false` to the node that represents this conditional statement shows that control flows from one node to the next. The conditional statement represents a decision point or *branch* in the code and is represented by a diamond node.

If the value of the conditional test is `true`, control next flows to the statement that sets the value of `isLeap` to `true`. Otherwise, control flows to the next conditional test to determine if the year is divisible by 400. No matter which route the flow of control takes through the various conditional tests, the final statement to be executed is the explicit return of the value of the `isLeap` variable as the value produced by the method.

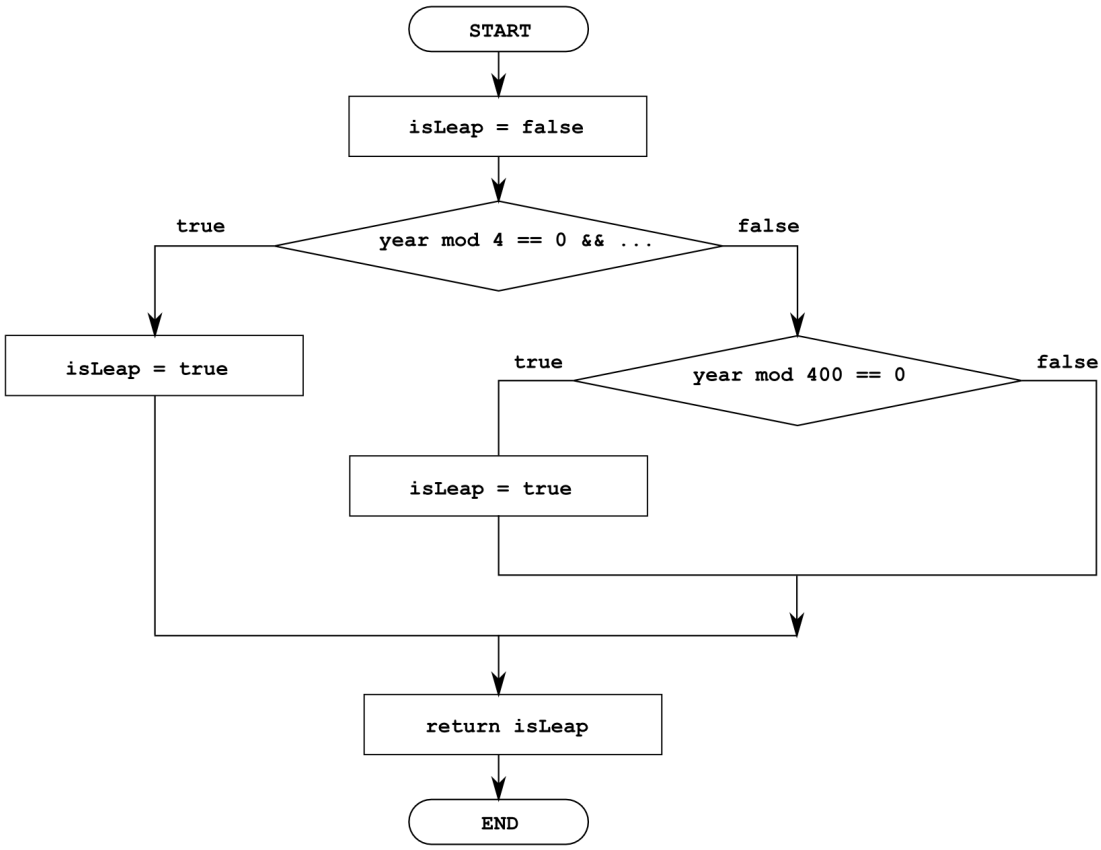
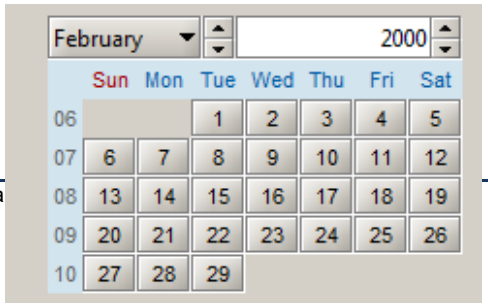


Figure 2 Flowchart for isALeapYear method

We will talk more in lecture about how to map the code to the flowchart. After lecture, re-examine the flowchart and make sure you understand how to translate the `isALeapYear` method to the flowchart representation.

Inter-method control-flow

A single method by itself does not provide much interesting functionality to a user. A user might want to invoke the `isALeapYear`



method to test whether a particular year is a leap year but it is unlikely. Instead, the user is likely to want the right dates to appear in a date chooser displayed by an application, such as the one shown to the right of this paragraph (from Kai Toedter, www.toedter.com/en/jcalendar/index.html).

The code below is a (greatly) simplified segment of Java code (from the site specified above) to support the drawing of the days of the specified year and month.[2]

```
/**
 * JDayChooser is a class for choosing a day
 *
 * @author Kai Toedter
 */
class JDayChooser {

   Calendar calendar;

   JDayChooser() {
      calendar = new Calendar();
      init();
   }

   void init() {
      drawDayNames();
      drawDays();
   }

   void drawDayNames() {
      int firstDayOfWeek = calendar.getFirstDayOfWeek();
      // generate and draw names
   }

   void drawDays() {
      // do a bunch of drawing
   }
}
```

This code outlines a partial definition for a `JDayChooser` Java class,[3] which is a graphical user interface (GUI) widget that supports the selection of a day in a displayed month of a specified year. This partial class definition includes one field, `calendar`. A

[2] As with the `isALeapYear` method, this code has been simplified to the point that it will not run. Comments have also been elided to keep the code on one page.

[3] You may want to skip ahead to the first part of the Intermezzo section below in this reading for a quick overview of classes and objects.

field in a Java class stores state for an object constructed from the class. The statement "`Calendar calendar`" states that the field, `calendar` (note the lower case `c`) is of type `Calendar`. We will talk about the specific meaning of types in a week or two in lecture. For now, you can think of a type as describing what computations can be performed on a calendar. Following the definition of the field is a definition of a *constructor* (a special kind of Java method) that is used to create an object (or an "*instance*" in other terminology) of the `JDayChooser` class; in this case a `JDayChooser` object is one visible representation of a day chooser widget. Multiple objects can be created from a class so a software application could show multiple `JDayChooser` objects at one time. Each `JDayChooser` object would have separate fields (e.g., in this case, separate copies of `calendar`s). Three methods, `init`, `drawDayNames` and `drawDays`, are defined following the `JDayChooser` constructor. (As you look at the code for this class, think about how you might distinguish fields from constructors from methods.)

To start understanding how the `JDayChooser` class works, it is often helpful to first understand how the methods of the class work together, before diving into the details of how each method exactly works. This strategy is not that different than ones you might use when reading a chapter of a textbook, where you might first skim the chapter to see the topics and how they are organized before diving into the details. Just like you might read the chapter a few times to understand it, you may find it useful to read code in various iterations.

One way to depict the flow of control between methods is to use a method-based call graph, which is simply a diagram in which the nodes are methods and arrows depict possible flows of control between the methods. Figure 3 shows a method-based call graph for the `JDayChooser` code. If the methods are named well so that the name provides a correct clue to their function, a call graph can help a software developer understand—at an abstract level—how the software functions before diving into the details. Note that the call graph does not explicitly show where control starts or ends as a flowchart can. The diagram also does not indicate if a node always calls methods it points to or the order in which they are called.

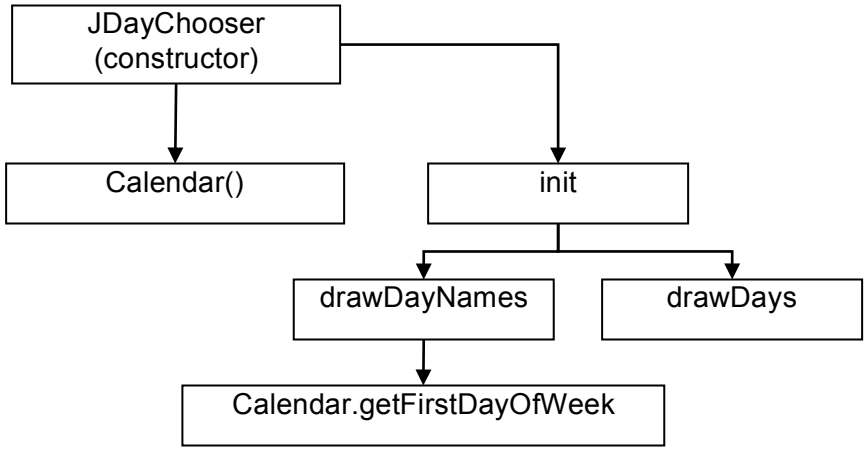


Figure 3 Method-based call graph

Inter-method control flow diagrams, such as the method-based call graph, are often used in the manner depicted in Figure 3; that is a call graph of a particular part of a software system may be determined to understand a part of the system at a time.

We will go over the details of creating a method-based call graph in class. In particular, we will be more careful about showing both the name of the class containing a method and the method name when drawing call graphs from this point on. Figure 3 is meant to capture and provide a simple case. After we go over call graphs in lecture, make sure you know how to create the above call graph from the given code.

Data Model

Most software systems need to interact with, and have a representation, of the real world. For example, a software system that helps to control a chemical processing plant is likely to have sensors that provide data about the temperature in vessels and may need to monitor the changes in those temperatures over time. As another example, iTunes tracks whether or not a device being used for playback of digital content is authorized to play that content.

In addition to tracking information about the real world, most software systems need to maintain information from one execution of the system to the next. An email reader is not much use if it does not track the messages that were read in previous interactions the user had with the reader. A digital address book is not much use if it does not remember the contacts that have been added in the past.

In general, the data required by a software system undergoes less change over time than computations performed on that data. For instance, a contact stored in an address book is very likely to have a first name, last name, email address, work phone number, home phone number, mobile phone number, and so on. These primary fields of information about a contact have not changed for many years. On the other hand, social network programs, like twitter, Facebook and LinkedIn, are expanding how this data is used (i.e., the computations that are performed on the data).

The importance of the data tracked by a software system to the computations that can be supported and the stability of that data over various versions of the software system mean that gaining an understanding of the data model used in a system is critical to understanding how the system works. By a data model, we mean the shape of data structures used to represent the information of interest in a system.

We will start by considering the shape of individual data structures. As we look at more and more involved programs, we will look at expanding our notion of a data model to capture dependences between data structures explicitly.

Consider the following partial definition of the `Calendar` class from the Java standard libraries.[4] This class provides support for determining the year, month, etc. from a specified instant in time and helps to compute such values as the date of the next week.

```
class Calendar {

   // Current time for this calendar in milliseconds after
   // January 1, 1970, 0:00:00 GMT
   long time;
```

[4] Full source available at http://www.docjar.com/html/api/java/util/Calendar.java.html. Many details elided for example purposes.

```
    // True if the value of time is valid
    boolean isTimeSet;

    // The timezone used by the calendar
    TimeZone timezone;

    // True if out-of-range values of time are allowed
    boolean lenient;

    // First day of week (e.g., Sunday or Monday)
    int firstDayOfWeek;

}
```

When we extract the shape of the `Calendar` data structure, we are looking for fields of the class that describe essential properties of a calendar. For `Calendar`, we can describe the data structure as

```
Calendar
    long time
    TimeZone timezone
```

because the `Calendar` is defined as providing operations around an instant in time (provided by the time field) within a particular time zone. The other fields in the class definition (i.e., `lenient` and `firstDayOfWeek`) have more to do with the implementation of `Calendar` and the precise operations provided (not shown) than being essential `Calendar` properties. When describing the data structure, we include the kind (or type) of data, (such as `time` being a `long` value), so we can understand the composition of data being used to define the structure of interest. Determining which data of a class is essential data takes practice and there are no hard and fast rules. As we discuss how to make software modular, the heuristics for determining the essential parts of a data structure will become clearer. For now, if you are asked to create a data model from a class, include all of the fields of the class.

To describe the data structures of an object-oriented program, we use a UML class diagram. UML stands for the Unified Modeling Language. It is a standardized modeling language for describing various perspectives of an object-oriented software system. Through the course, we will be looking at various parts (but not all) of the modeling language. UML can be used to either describe an existing software system or to design a new software system. For particular domains, it is possible to generate parts of a software system from designs expressed in UML.

UML class diagrams can represent complex relationships about the data structures in an object-oriented program. We are going to use the simplest form of a UML class diagram for now. We will expand our use of the UML class diagram notation later in the course.

A class in a UML class diagram is represented by a three-part box. Each box in the diagram represents a separate class in the system. In the top part of the box, we state the name of the class. In the middle part of the box, we state the essential data of an object of that class. In the last part of the box, we can specify the operations (i.e., methods) available on an object of the class; we'll skip using this part of the box for now. The UML class diagram for the `JDayChooser` system outlined above is shown in Figure 4.[5]

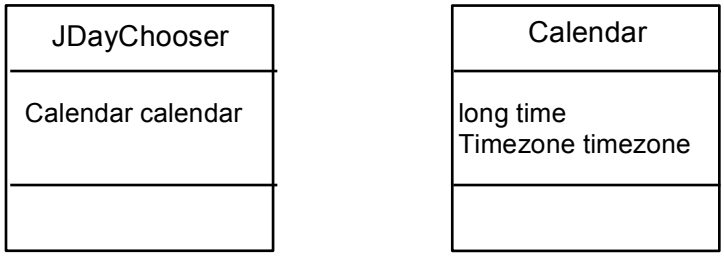


Figure 4 UML Class Diagram (data model) for `JDayChooser`

[5] Note that if `Timezone` is also a class defined in the `JDayChooser` system than it would have a box representing it as well in the UML class diagram. For now, we assume `Timezone` is defined in another library and do not depict it as part of the data model of the `JDayChooser` system.

Intermezzo: Java Classes and Objects

Object-oriented programming is a programming style in which software is expressed as objects—collections of state and related behaviour—that often model entities in the real world. The state of an object is data that must be remembered so that operations on that object can be performed. For example, in an iTunes-like software system, an object can represent a song or piece of digital content. The state of such an object would include information such as the title of the track and the number of times it has been played. The behaviour of an object is the collection of computations that can be performed on or by the objects. For example, in an iTunes-like software system, the object representing a song would include behaviour to support playing the song.

Java supports programming in an object-oriented style. A Java class provides a blueprint for creating objects of a particular kind. For instance, the `Calendar` class sketched in the last section provides a blueprint for creating calendars representing different times in different time zones. A particular `Calendar` object can be created from this class by using the `new` syntax.

```
// assume currentTime, pacificTimeZone and easternTimeZone are
// variables with appropriate values
Calendar calendar1 = new Calendar(currentTime, pacificTimeZone);
Calendar calendar2 = new Calendar(currentTime, easternTimeZone);
```

`calendar1` is a different object than `calendar2`. Each has state as described by the fields defined in the `Calendar` class. Each may have different values for those fields (i.e., separate state). For example, `calendar1` captures the current time according to the pacific time zone whereas `calendar2` is set to use the eastern time zone. Each provides the behaviour defined in the `Calendar` class. Behaviour is provided by the methods defined on a class. When a new `Calendar` object is created, a constructor defined in the `Calendar` class is executed.

For more information about Java classes and objects, consult the Java tutorial reference material. Here are relevant web pages, with a description of what to ignore for now.

- http://java.sun.com/docs/books/tutorial/java/concepts/index.html, look at the "What is an Object?" and "What is a Class?" sections.
- http://java.sun.com/docs/books/tutorial/java/concepts/object.html, focus on the bicycle example. We will be talking about data encapsulation, information hiding, modularity, code reuse and pluggability in upcoming classes.
- http://java.sun.com/docs/books/tutorial/java/concepts/class.html, don't worry about the details of a `main` method for now, just read the code inside the `main` method.

Constructing a software system expressed in an object-oriented programming language is about orchestrating the invocation of the appropriate methods on the appropriate objects at the right times.

Bringing Control and Data together: UML Sequence Diagrams

When we extracted inter-method control-flow diagrams (i.e., call graphs) earlier in this section, we ignored the sequence in which methods may execute. For many tasks a software developer performs, an understanding of the code in terms of call graphs is sufficient. In other cases, more knowledge about the sequence of how the methods execute is needed. In an object-oriented program, it is also sometimes necessary to understand on which objects methods are executing as the software system will produce different results depending on which objects are involved. One way to capture this information is to use a UML sequence diagram.

A UML sequence diagram depicts the order of interactions between objects to accomplish an operation of interest. We will just consider the basics of UML sequence diagrams here, leaving the variations in notation available to depict different kinds of interactions for the future. In a UML sequence diagram, the objects involved in the operation of interest are listed horizontally across the top of the diagram. Each object is given a (vertical dashed) lifeline to represent when it participates in the operation. The vertical part of the diagram shows the sequence of method calls on the various objects.

Figure 5 shows a UML sequence diagram for part of the `JDayChooser` example presented earlier in this section. Specifically, the diagram shows the execution of the part of the code shown for the `JDayChooser.drawDayNames` method. Horizontally, the diagram captures the lifelines of two objects: a `JDayChooser` object (the graphical widget displayed on a user's screen) called `aDayChooser` and a calendar object that is part of `JDayChooser's` data structure. Vertical boxes on the lifelines indicate the execution of a method. When the `drawDayNames` method executes in the context of the `aDayChooser` object, a call is made to the method `getFirstDayOfWeek` on the `calendar` object that is part of the data stored and used by the `aDayChooser` object. The arrow across to the `calendar` object labeled with `getFirstDayOfWeek` indicates the call to that method, the vertical box indicates how long the method executes and the return arrow to `aDayChooser` shows when the execution of the `getFirstDayOfWeek` method completes. Compare the UML sequence diagram for this code to the call graph provided in Figure 3.

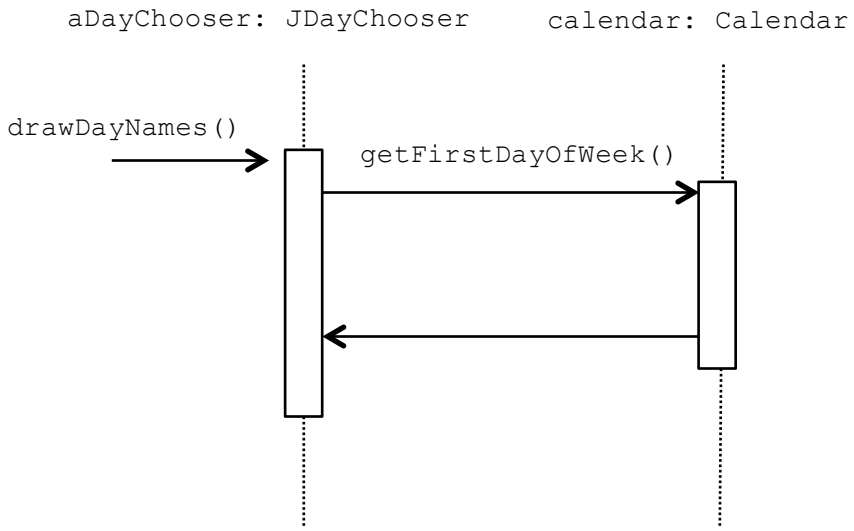


Figure 5 UML Sequence Diagram Example

UML sequence diagrams can be drawn to explain how a single method works, as in Figure 5, or can be drawn to show how many methods work together. We will focus in this course on drawing UML sequence diagrams to explain how one method works, which may involve more than one other object as shown in Figure 5. You will see more sophisticated examples in lecture.

References and Further Reading

The following sections of the Java tutorial describe some of the syntax that you've seen in the `isLeapYear` method. Read through these parts of the tutorial and then read through this document again.

- Equality, Relational and Conditional Operators: read the sections entitled *The Equality and Relational Operators* and *The Conditional Operators* http://download.oracle.com/javase/tutorial/java/nutsandbolts/op2.html

- The if-then and if-then-else statements http://download.oracle.com/javase/tutorial/java/nutsandbolts/if.html