Data Abstraction: Java Supplement

H. Conrad Cunningham

17 September 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of September 2018 is a recent version of Firefox from Mozilla.

Data Abstraction: Java Supplement

Introduction

This set of notes forms a Java-specific supplement to the lecture notes on Data Abstraction. However, most of the concepts apply to other object-oriented languages such as Scala. The discussion here is meant to be read after completion of the Data Abstraction notes.

This set of notes discusses use of Java to implement abstract data types (ADTs). It seeks to use good object-oriented programming practices, but it does not cover the principles and practices of object-oriented programming fully. For more information on object orientation, see the notes on Object-Oriented Software Development and Object-Based Paradigms.

Java Class Implementation for Day

The following implementation of the `Day` ADT is adapted from the like-named class in Chapter 4 of the book *Core Java 1.2: Volume I -- Fundamentals* (Fourth Edition) by Cay S. Horstmann and Gary Cornell.

In general, this set of notes assumes that the implementations of the ADTs use mutable (stateful) objects. The approach to design of mutator methods would be somewhat different if immutable objects are used. But otherwise the general approach to design applies to immutable objects.

Caveat: This document was originally written before the release of Java 5. However, other than needing to be updated to incorporate newer Java features such as generics, functional interfaces, and lambdas, the principles are still relevant to contemporary Java programming.

Java Classes

As a language construct, a Java `class` is similar to a user-defined `struct` type in C or user-defined record *type* in Pascal. A `class` is a template for constructing data items that have the same structure but differing values (states). We say that an item constructed by a class is a *class instance* (or, as we see later, an *object*).

Like the C structure type or Pascal record type, a Java class can consist of several components. In C and Pascal, all the components are data fields. However, in Java, functions and procedures may be included as components of a class. These procedures and functions are called *methods*.

Class and Instance Methods

A method declared in a class may be either a class method or instance method.

- A *class method* is associated with the class as a whole, not with any specific instance.
- An *instance method* is associated with an instance of the class.

We declare a method as a *class method* by giving the keyword `static` in the header of its definition. For example, a `main` method of a program is a class method of the class in which it is defined.

```
public static void main(String[] args)
{   //  beginning code for the program
}
```

If we *do not include* the keyword `static` in the header of a method definition, the method is an *instance method*. For example, consider methods to implement the `push` and `top` operations of a class that implements a `StackB`.

```
public void pop()
{   // code for pop operation
}

public Object top()
{   // code for top operation
}
```

Note that `pop()` is a *procedure* (i.e. it has return type `void`) method and `top` is a *function* method.

Scala note: The Scala language does not have static members of classes. However, the methods of a Scala singleton `object` have basically the same characteristics described above for "class methods". Often the "class methods" appear in the companion `object` for a class. i.e. the `object` with the same name as the class.

Class and Instance Variables

In a similar fashion, the variables (data fields) declared in a class may be either class variables or instance variables.

- A *class variable* is associated with the class as a whole; there is only one copy of the variable for the entire class. As with methods, the keyword `static` is used to declare a class variable.
- An *instance variable* is associated with an instance of the class; each instance has its own instance of the variable. As with methods, the absence of the keyword `static` denotes an instance variable.

An instance method has direct access to the instance variables of the class instance (object) to which it is applied. The instance's variables are implicit arguments of the method calls. (If needed to distinguish among names, the builtin variable `this` can be used to refer to the instance to which the method is applied.) The instance methods also have access to the class variables (if any).

Class methods only have access to the class variables. The methods do not have any implicit arguments. In fact, class methods can be called without any instances of the class being in existence.

Scala note: The Scala language does not have static members of classes. However, the variables of a Scala singleton `object` have basically the same characteristics described above for "class variables". Often the "class variables" appear in the companion `object` for a class. i.e. the `object` with the same name as the class.

Public and Private Accessibility

The components of a class can be designated as `public` or `private`.

- The `public` components of the class are accessible from anywhere in the program (i.e. from any package).
- The `private` components are only accessible from inside the class.

As a general rule, the data fields of a class should be *private instance variables*, meaning that they are associated with a specific instance and are only accessible by the instance methods. This hides, or encapsulates, the data fields within the class instance.

Note: Actually, the instance methods of a Java class can access the instance variables of any instance of that class, not just the current instance.

In general, avoid public instance variables. They break the principle of information hiding, leading to potential entanglements among modules.

A public method of a class is a service provided by that instance to other parts of a program. The private methods of a class can be used in implementing the public methods.

Class methods and variables should be used sparingly. These are more or less the types of subprograms and global variables found in languages like C and Pascal. Their excessive use can greatly reduce the potential benefits that can be realized from object-oriented techniques.

Java note: There are two other types of accessibility, "friendly" and `protected`, but `public` and `private` are sufficient for our discussion of ADT implementations.

Scala note: Although similar in concept to that of Java, the accessibility features of Scala differ somewhat. By default, all features are public in Scala, but

accessibility can restricted in a more fine-grained manner than in Java. The unmodified keyword `private` has the same meaning as in Java.

Primitive and Reference Variables

A Java variable is a strongly typed "container" in memory that is declared to hold either:

- a *value* of the associated primitive data type such as integers (`int`), floating point numbers (`double`), booleans (`boolean`), and single characters (`char`).
- a *reference* to (i.e. memory address of) an instance of the associated class (or other reference) type.

Java note: Although arrays are not class instances, array variables hold a reference to an instance of the array.

The class instances themselves are stored in the dynamically managed heap memory area. Java allocates memory from the heap to hold newly constructed instances of a class. Java's garbage collector reclaims the memory for instances that are no longer needed by the program.

Note: Recent versions of Java can sometimes hide the differences between primitive values and references by automatically "boxing" primitive values as instances of the corresponding wrapper classes (e.g. `int` values as `Integer` instances). Scala goes further in that primitives and references are in the same type hierarchy. However, both languages run on the Java Virtual Machine, which makes a distinction between primitive values and references (i.e. pointers), so it is not possible to avoid the distinction entirely.

Implementing ADTs as Java Classes

If only one implementation of an ADT is needed, the following techniques can be used to implement an ADT using Java.

The implementation techniques discussed in this section implement the ADT in an imperative way. That is, instead of returning a new instance of the ADT with a modified state, a mutator operation usually modifies the state of the existing instance.

Caveat: The discussion of Java in these notes does not use generic type parameters. For the `StackB` ADT (defined in the Data Abstraction notes), the type of the `Item` values stored in the stack can be a parameter of the `StackB` class.

1. **Use the Java `class` construct to represent the entire ADT.** If we want to allow access to the class from anywhere in the program, we will make the class `public`.

   For the `StackB` ADT, we can use the following structure for the class:

```
public class StackB
{   // implementation of instance methods and data here
}
```

2. **Use an instance of the Java class to represent an instance of the ADT and, hence, variables of the class type to hold references to instances.**

   For example, to declare a variable that can hold a reference to a `StackB` instance, we can use the following declaration:

   ```
   StackB stk;
   ```

3. **As each component of the class is defined, ensure that the semantics of the ADT operations are implemented appropriately.** That is, make sure:

   - an appropriate implementation (representation) invariant is defined to capture what it means for the internal state of an instance to be valid,
   - the interface and implementation invariants are established (i.e. made true) by the constructors and preserved (i.e. kept true) by the mutator and accessor methods,
   - each method's postcondition is established by the method in any circumstance when it is called with the precondition true.

   The class and its methods should be documented with the invariants, preconditions, and postconditions.

4. **Represent the ADT's constructors by Java constructor methods. In most circumstances, also include a parameterless default constructor.**

   A Java constructor is a method with the same name as the class. It does not have a return type specified. Upon creation of an instance of the class, the constructor initializes the instance's state so that the class invariants are established.

   A constructor is normally invoked by the Java operator `new`. The operator `new` allocates memory on the heap for the instance, calls the constructor to initialize the new instance, and then returns a reference to the new instance.

   For example, we can represent the ADT operation `create` by the constructor method `StackB`.

   ```
   public class StackB
   {   public StackB(int size)
       {   // initialization code
       }
   ```

```
// rest of StackB methods and data ...
}
```

A user of the `StackB` class can then declare a variable and initialize it to hold a reference to a new stack with a capacity of 100 items as follows:

```
StackB stk = new StackB(100);
```

The expression `new StackB(100)` allocates a `StackB` instance in the heap storage and calls the constructor above to initialize the data fields encapsulated within the instance.

5. **Represent the ADT operations by instance methods of the class.** Thus the state of the ADT instance, which is given explicitly in the ADT signatures, becomes an *implicit argument* of all method calls. Mutators also have the state as an implicit return.

   We can apply a method to a class instance by using the selector (i.e. "dot") notation. This notation is similar to the notation for accessing `record` components in Pascal.

   For example, in the case of the `StackB` ADT we can represent the operations as instance methods of class `StackB`. The explicit `StackB` parameters and return values of the operations thus become implicit.

   Suppose we want to push an item `x` onto the `stk` created above. We can do that with the following code:

   ```
   if (!stk.full())
       stk.push(x);
   ```

   We can then examine the top item and remove it:

   ```
   if (!stk.empty())
   {   it = stk.top();
       stk.pop();
   }
   ```

6. **Make the constructors, mutators, accessors, and destructors `public` methods of the class.** That is, precede the method's definition by the keyword `public`.

7. **Represent the ADT mutator operations by Java procedure (i.e. `void`) methods, except those mutator operations that explicitly require new instances to be generated (e.g. a copy or `clone` operation).**

   For example, the `pop` method of `StackB` would have the following structure:

   ```
   public void pop()
   {   //  code to implement operation
   }
   ```

A mutator method modifies the encapsulated state of the class instance (which is the implicit argument of the method). In any circumstance in which its precondition and the class invariants hold on entry, the method must establish its postcondition and reestablish the invariants upon exit. (The invariant might not hold in the middle of the method's execution.)

Comment: Implementing mutator operations as procedure calls that modify the stored state is really an optimization. All mutators can be implemented in the applicative style, returning a modified copy of the instance. This implementation might, however, be inefficient in use of processor time and memory.

8. **For certain mutator operations (e.g. copy or `clone`), implement the corresponding Java methods to return new instances of the class rather than to modify the current instance (i.e. their implicit arguments).**

   Any mutator method must, of course, establish its postcondition and reestablish the invariants for the current instance. In addition, these applicative mutators must also establish the invariants for the new instance returned.

9. **Represent the ADT accessor operations by Java function methods of the proper return type.**

   For example, the `empty` method of `StackB` would have the following structure:

   ```
   public boolean empty()
   {   //  code to implement operation
   }
   ```

   An accessor method accesses the encapsulated state of the class instance (which is the implicit argument) and computes a value to be returned. In any circumstance in which its precondition and the class invariants hold on entry, the method must establish its postcondition and reestablish the invariants upon exit. (The invariant might not hold in the middle of the method's execution.)

10. **If necessary for deallocation of internal resources, represent the ADT destructor methods by explicit Java procedures; in most cases, however. just allow the automatic garbage collection to reclaim instances that are no longer being used.**

    For example, in the `StackB` class, we might include an explicit destroy operation that releases the storage resources and disables further use of the instance.

    ```
    public void destroy()
    {   //  code to free resources
    }
    ```

Note: The Java framework allows a `finalize()` method to be included in each class. This method is called implicitly whenever the garbage collector detects that the instance is no longer in use. However, since it is difficult to predict when (if ever) this method will be executed, it is safer to include explicit destructors when resources are in short supply and must be explicitly managed.

11. **Use `private` data fields of the Java class to represent the encapsulated state of the instance needed for a particular implementation.** By making the data fields `private` they are still available to the instance's methods, but are not visible outside the class.

    For example, the `StackB` class might have the following data fields:

    ```
    public class StackB
    {   //  public operations of class instance

        //  encapsulated data fields of class instance
        private int topItem;   // Pointer to next index for insertion
        private int capacity;  // Maximum number of items in stack
        private Object[] stk;  // the stack
    }
    ```

12. **Do not use `public` data fields in the class. These violate the principle of information hiding. Instead introduce appropriate accessor and mutator methods to allow manipulation of the hidden state.**

13. **Include, as appropriate, `private` methods to aid in implementation.**

    Functionality common to several methods can be placed in separate functions and procedures as needed. However, since these are `private`, they can only be accessed from within the class and thus can be changed without affecting the public interface of the class.

14. **Add any other methods needed to make the ADT fit into the Java environment.**

    For example, it is frequently useful to add public `toString` and `clone` methods. The `toString` method returns a Java `String` reflecting the "value" of the instance in a format suitable for printing. The `clone` method creates a new instance that has the same value as the current instance.

15. **In general, avoid use of class (i.e. `static`) variables.** Since a class variable is shared among all instances of the class, it may be difficult to preserve the invariants for individual instances as the value of the class variable changes.

    However, it is a good programming practice to **use class *constants* where appropriate.** These are data fields declared with both the `static`

and `final` modifiers. Their values may be initialized but cannot be changed thereafter.

These constants may be declared `private` if usage is to be restricted to the class or `public` if the users of the class also need access.

By convention, the names of constants are normally written with all uppercase letters. For example, the following defines a symbolic name for the integer code used for Sunday as a day of the week in the `Day` class defined later.

```
public static final int SUNDAY = 1;
```

Caveat: When this set of notes was originally written, Java did not yet have generics. So the examples below handle the type parameters of the ADT in other ways. A Java generic provides a class facility that can be parameterized with types (like the C++ `template` or Ada `generic` mechanisms).

For example, in the implementation below we represent the set `Item` of the `StackB` ADT by the class `Object`. As we will see when we discuss inheritance, the `Object` type will allow us to store an instance of any class on the `StackB`. With this definition, any data of a reference type can appear in the stack, but values of the primitive types cannot. A better implementation would have `Item` as type parameter of the class.

The next section gives a Java implementation of the `StackB` ADT. A similar constructive definition and two implementations of a Queue ADT are available in a separate document.

Java Implementation of Bounded Stack

In this section, we give an implementation of the `StackB` ADT that uses an array of objects and an integer "pointer" to represent the stack. (This implementation does not use Java generic classes.)

This implementation is not robust; each operation assumes that its precondition holds. A more robust implementation might check whether the precondition holds and throw an exception if it does not.

Remember that the invariants are implicitly pre- and postconditions of all mutator and accessor methods, postconditions of the constructor, and preconditions of the destructor.

```
// A Bounded Stack ADT
public class StackB
{   // Interface Invariant:  Once created and until destroyed, this
    //     stack instance has a valid and consistent internal state

    public StackB(int size)
```

```
// Pre:   size >= 0
// Post:  initialized new instance with capacity size && empty()
{   stk = new Object[size];
    capacity = size;
topItem = 0;
}

public void push(Object item)
// Pre:   not full()
// Post:  item added as the new top of this instance's stack
{   stk[topItem] = item;
    topItem++;
}

public void pop()
// Pre:   not empty()
// Post:  item at top of stack removed from this instance
{   topItem--;
    stk[topItem] = null;
}

public Object top()
// Pre:   not empty()
// Post:  return item at top of this instance's stack
{   return stk[topItem-1];
}

public boolean empty()
// Pre:   true
// Post:  return true iff this instance's stack has no elements
{   return (topItem <= 0);
}

public boolean full()
// Pre:   true
// Post:  return true iff this instance's stack is at full capacity
{  return (topItem >= capacity);
}

public void destroy()
// Pre:   true
// Post:  internal resources released;  stack effectively deleted
{   stk = null;
capacity = 0;
    topItem = 0;
}
```

```
    // Implementation Invariant for informal model:
    //     0 <= topItem <= capacity &&
    //     stack is in array section stk[0..topItem-1]
    //         with the top at stk[topItem-1], etc.

    // Implementation Invariant for more formal model representing stack
    // as tuple (integer max, sequence stkseq)
    //     m == capacity && 0 <= topItem <= capacity &&
    //     stackInArray(stk,topItem,stkseq)
    //     where stackInArray(arr,t,ss) = if t == 0 then ss == []
    //                                    else arr[t-1] == head(ss)
    //                                       && stackInArray(arr,t-1,tail(ss))

    private int topItem;   // Pointer to next index for insertion
    private int capacity;  // Maximum number of items in stack
    private Object[] stk;  // the stack
}
```

Better Approach to Implementing ADTs in Java

If several different implementations of an ADT are needed, then the Java specification of an ADT's interface should be separated from the class implementation. The interface specification can be reused among several classes and various implementations of the interface can be used interchangeably.

This can be done as follows:

1. **Define a Java `interface` that specifies the type signatures for the ADT's mutator and accessor (and, if needed, destructor) operations.** These method signatures should have the same characteristics as described above in the discussion of class-based specification.
2. **Specify and document the `interface` by the interface invariants, preconditions, and postconditions that must be supported by any implementation of the ADT.** There are no implementation invariants for an interface, but individual classes that implement the `interface` will have them.

   For example, a bounded stack `interface` might be specified as follows:

```
public interface StackADT
{   // Interface Invariant:  Once created and until destroyed, this
    //     stack instance has a valid and consistent internal state

    public void push(Object item);
    // Pre:   not full()
```

```
    // Post:  item added as the new top of this instance's stack

    ...

    public Object top();
    // Pre:   not empty()
    // Post:  return item at top of this instance's stack

    ...
}
```

3. **Provide one or more concrete classes that implement the `interface`.**

   For example, an array-based `StackADT` could be implemented similarly to the `StackB` definition given in the previous section.

```
public class StackInArray implements StackADT
{   // Interface Invariant:  Once created and until destroyed, this
    //     stack instance has a valid and consistent internal state

public StackInArray(int size)
// Pre:   size >= 0
// Post:  initialized new instance with capacity size && empty()
{   stk = new Object[size];
capacity = size;
         topItem = 0;
    }

    public void push(Object item)
    // Pre:   not full()
    // Post:  item added as the new top of this instance's stack
    {   stk[topItem] = item;
        topItem++;
    }

    ...

    public Object top()
    // Pre:   not empty()
    // Post:  return item at top of this instance's stack
    {   return stk[topItem-1];
    }

    ...

    // Implementation Invariant for informal model:
```

```
    //     0 <= topItem <= capacity &&
    //     stack is in array section stk[0..topItem-1]
    //         with the top at stk[topItem-1], etc.

    // Implementation Invariant for more formal model representing stack
    // as tuple (integer max, sequence stkseq)
    //     m == capacity && 0 <= topItem <= capacity &&
    //     stackInArray(stk,topItem,stkseq)
    //     where stackInArray(arr,t,ss) =
    //               if t == 0 then ss == []
    //               else arr[t-1] == head(ss)
    //                    && stackInArray(arr,t-1,tail(ss))

    private int topItem;   // Pointer to next index for insertion
    private int capacity;  // Maximum number of items in stack
    private Object[] stk;  // the stack
}
```

4. **Declare variables of the ADT's `interface` type to hold instances of any concrete class that implements the `interface`.** Any of the operations defined in the `interface` can be applied to the instance to which this variable refers.

   For example, a variable of type `StackADT` can hold instances of any concrete class that implements the `interface StackADT`.

   ```
   StackADT theStack = new StackInArray(100);
   theStack.push("Hello World");
   ```

For an ADT specification and implementations that follow this approach, see the description of the Ranked Sequence ADT case study given in a separate document. In addition to Java interfaces, the Ranked Sequence case study uses other Java features such as exceptions, enumerations, packages, and Javadoc annotations.

Java Class Implementation for Day

The following implementation of the `Day` ADT is adapted from the like-named class in Chapter 4 of the book *Core Java 1.2: Volume I — Fundamentals* (Fourth Edition) by Cay S. Horstmann and Gary Cornell (Sun Microsystems Press/Prentice Hall, 1999).

This implementation represents the calendar as three integers. It converts the dates to and from Julian dates to do some of the operations.

```
//  This class implementation is adapted from the Day class in
//  Horstmann and Cornell, Core Java 1.2: Volume I - Fundamentals
//  (Fourth Edition), Prentice Hall, 1999.
```

```
import java.util.*;
import java.io.*;

public class Day
{
    // Interface Invariant:  Once created and until destroyed, this
    //    instance contains a valid date.  gateau() != 0 &&
    //    1 <= getMonth() <= 12 && 1 <= getDay() <= #days in getMonth().
    //    Also calendar date getMonth()/getDay()/getYear() does not
    //    fall in the gap formed by the change to the modern
    //    (Gregorian) calendar.

  //  Constants for days of the week

    public static final int SUNDAY    = 1;
    public static final int MONDAY    = 2;
    public static final int TUESDAY   = 3;
    public static final int WEDNESDAY = 4;
    public static final int THURSDAY  = 5;
    public static final int FRIDAY    = 6;
    public static final int SATURDAY  = 7;

  // Constructors

    public Day()
    //  Pre:   true
    //  Post:  the new instance's day, month, and year set to today's
    //             date (i.e. the date of creation of the instance)
    //
    //  Implementation uses GregorianCalendar class from the Java API
    //  to get today's date.
    //
    {   GregorianCalendar todaysDate = new GregorianCalendar();
        year  = todaysDate.get(Calendar.YEAR);
        month = todaysDate.get(Calendar.MONTH) + 1;
        day   = todaysDate.get(Calendar.DAY_OF_MONTH);
    }

    public Day(int y, int m, int d)
           throws IllegalArgumentException
    //  Pre:   y != 0 && 1 <= m <= 12 && 1 <= d <= #days in month m
    //         (y,m,d) does not fall in the gap formed by the
    //             change to the modern (Gregorian) calendar.
    //  Post:  the new instance's day, month, and year set to y, m,
    //         and d, respectively
```

```
  //  Exception:  IllegalArgumentException if y m d not a valid date
  {   year  = y;
      month = m;
      day   = d;
      if (!isValid())
          throw new IllegalArgumentException();
  }

// Mutators

  public void setDay(int y, int m, int d)
              throws IllegalArgumentException
  //  Pre:   y != 0 && 1 <= m <= 12 && 1 <= d <= #days in month m
  //         (y,m,d) does not fall in the gap formed by the
  //             change to the modern (Gregorian) calendar.
  //  Post:  this instance's day, month, and year set to y, m,
  //         and d, respectively
  //  Exception:  IllegalArgumentException if y m d not a valid date
  {   year  = y;
      month = m;
      day   = d;
      if (!isValid())
          throw new IllegalArgumentException();
  }

  public void advance(int n)
  //  Pre:   true
  //  Post:  this instance's date moved n days later.  (Negative n
  //             moves to an earlier date.)
  {   fromJulian(toJulian() + n);
  }

// Accessors

  public int getDay()
  //  Pre:   true
  //  Post:  returns the day from this instance, where
  //             1 <= getDay() <= #days in this instance's month
  {   return day;
  }

  public int getMonth()
  //  Pre:   true
  //  Post:  returns the month from this instance's date, where
  //             1 <= getMonth() <= 12
  {   return month;
```

```
  }

  public int getYear()
  //  Pre:   true
  //  Post:  returns the year from this instance's date, where
  //             getYear() != 0
  {   return year;
  }

  public int getWeekday()
  // Pre:   true
  // Post:  returns the day of the week upon which this instance
  //            falls, where 1 <= getWeekday() <= 7;
  //            1 == Sunday, 2 == Monday, ..., 7 == Saturday
  {   //  calculate day of week
      return (toJulian() + 1) % 7 + 1;
  }

  public boolean equals(Day dd)
  //  Pre:   dd is a valid instance of Day
  //  Post:  returns true if and only if this instance and instance
  //             dd denote the same calendar date
  {   return (year == dd.getYear() && month == dd.getMonth()
              && day == dd.getDay());
  }

  public int daysBetween(Day dd)
  //  Pre:   dd is a valid instance of Day
  //  Post:  returns the number of calendar days from the dd
  //             instance's date to this instance's date, where
  //             equals(dd.advance(n)) would hold
  {   //  implementation code
      return toJulian() - dd.toJulian();
  }

  public String toString()
  //  Pre:   true
  //  Post:  returns this instance's date expressed in the format
  //             "Day[year,month,day]"
  {
      return "Day[" + year + "," + month + "," + day + "]";
  }

//  Destructors -- None needed

//  Private Methods
```

```
private boolean isValid()
//  Pre:   true
//  Post:  returns true iff this is a valid date
{   Day t = new Day();
    t.fromJulian(this.toJulian());
    return t.day == day && t.month == month
           && t.year == year;
}

private int toJulian()
//  Pre:   true
//  Post:  returns Julian day number that begins at noon of this day
//
//  A positive year signifies A.D., negative year B.C.
//  Remember that the year after 1 B.C. was 1 A.D. (i.e. no year 0).
//
//  A convenient reference point is that May 23, 1968, at noon
//  is Julian day 2440000.
//
//  Julian day 0 is a Monday.
//
//  This algorithm is from Press et al., Numerical Recipes
//  in C, 2nd ed., Cambridge University Press 1992.
//
{   int jy = year;
    if (year < 0)
        jy++;
    int jm = month;
    if (month > 2)
        jm++;
    else
    {   jy--;
        jm += 13;
    }
    int jul = (int) (java.lang.Math.floor(365.25 * jy)
              + java.lang.Math.floor(30.6001*jm) + day + 1720995.0);

    int IGREG = 15 + 31*(10+12*1582);
        // Gregorian Calendar adopted Oct. 15, 1582

    if (day + 31 * (month + 12 * year) >= IGREG)
       // change over to Gregorian calendar
    {   int ja = (int)(0.01 * jy);
        jul += 2 - ja + (int)(0.25 * ja);
    }
```

```
    return jul;
}

private void fromJulian(int j)
//  Pre:   true
//  Post:  this calendar Day is set to Julian date j
//
//  This algorithm is from Press et al., Numerical Recipes
//  in C, 2nd ed., Cambridge University Press 1992
//
{   int ja = j;

    int JGREG = 2299161;
        /* the Julian date of the adoption of the Gregorian
           calendar
        */

    if (j >= JGREG)
    /* correct for crossover to Gregorian Calendar */
    {   int jalpha = (int)(((float)(j - 1867216) - 0.25)
            / 36524.25);
        ja += 1 + jalpha - (int)(0.25 * jalpha);
    }
    int jb = ja + 1524;
    int jc = (int)(6680.0 + ((float)(jb-2439870) - 122.1)
          /365.25);
    int jd = (int)(365 * jc + (0.25 * jc));
    int je = (int)((jb - jd)/30.6001);
    day = jb - jd - (int)(30.6001 * je);
    month = je - 1;
    if (month > 12)
        month -= 12;
    year = jc - 4715;
    if (month > 2)
        --year;
    if (year <= 0)
        --year;
}

//  Implementation Invariants:
//      year != 0 && 1 <= month <= 12 && 1 <= day <= #days in month
//      (year,month,day) not in gap formed by the change to the
//      modern (Gregorian) calendar

private int year;
private int month;
```

```
    private int day;
}
```

Acknowledgments

These notes were originally part of the Data Abstraction notes. I separated them into a separate document for the Lua-based offerings of CSci 658 in Fall 2013. See the Acknowledgments section of those notes for more information.

For the Elixir-based offering of CSci 556 Multiparadigm Programming in Spring 2015 and the Scala-based offering of CSci 555 Functional Programming in Spring 2016, I modified the notes to be more language independent.

I modified this document slightly in 2017 to be launched from the revised (Pandoc Markdown) version of the Data Abstraction document. I reformatted this document to use Pandoc Markdown in Spring 2018.

The Java `Day` implementation is adapted from:

- Cay S. Horstmann and Gary Cornell. *Core Java 1.2: Volume I — Fundamentals*, Fourth Edition, Sun Microsystems Press, Prentice-Hall, 1999.

Concepts

TODO