**TOP-DOWN PROCEDURAL PROGRAMMING**

*Top-down programming* is an incremental strategy where you implement the most general modules first and work towards implementing those that provide specific functionality. With procedural programs, this means that you start coding methods at the top of the structure chart and work your way towards its bottom.

To unit test a method when you haven’t yet implemented its children requires the use of stubs. A *stub* is a method that lacks full functionality and is present merely so that its parent can call it. This allows you to compile, execute and test the program even though it is not entirely finished.

**Example**
Here’s an illustration of top-down procedural programming. The structure chart represents a program’s architectural design.

<table>
<thead>
<tr>
<th>Implementation begins with the top-most method; its children are coded as stubs. Bottom-most methods are not coded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The stubs allow the top method to be compiled, executed and fully tested.</td>
</tr>
</tbody>
</table>

![Structure Chart Example](image-url)
Implementation continues with the methods at the next level; their children are coded as stubs.

The newly implemented methods are tested.

Implementation continues iteratively until the entire program is implemented and tested.
Top-Down Procedural Programming Example – Date Validation

The date validation program is designed in the topic *Procedural Design Methodology*. The structure chart and method specifications are reprinted below.

### Structure Chart for the Date Validation Program

```
                    String
                        ↓
                Convert to Valid Date

Validate Date

Extract mo, day, yr

Validate Date
```

### Method Specifications for the Date Validation Program

```java
public int [] convertToValidDate( String date )
/* Convert the given string to a date on the Gregorian calendar.
   * Precondition: none.
   * Postcondition: If the given date is a true Gregorian date in the form mm-dd-yyyy or mm/dd/yyyy, it returns an array with a[0]=mm, a[1]=dd and a[2]=yyyy.
   * It returns null if:
   *   date is null
   *   date doesn't have the indicated form.
   *   mm-dd-yyyy is not a true Gregorian date.
   */
```
public boolean isValidDateFormat( String date )
/* Determine if the given date has valid format;
   * specifically, ##-##-#### or ##/##/####.
   * Precondition: date is not null.
   * Postcondition: Returns true iff date has valid format.
   */

public int [] extractMoDayYr( String date )
/* Extract integers from the given date representing
   * its month, day and year.
   * Precondition: date has form mm-dd-yyyy or mm/dd/yyyy.
   */

public boolean isValidDate( int month, int day, int year )
/* Determine if month/day/year is a true Gregorian date.
   * Precondition: none.
   * Postcondition: returns true iff month/day/year is a true
   *   Gregorian date.
   */

Starting with convertToValidDate, its algorithm is straightforward since the real work is
done by its children:

Algorithm for convertToValidDate
1. if the date string is a null pointer, return null
2. check the date string for validity
3. if not valid, return null
4. extract the month, day and year
5. check that month, day and year make a real date
6. if not, return null
7. return month, day, year in an array
Next is `isValidDateFormat`, which has a simple brute-force algorithm since the date format is so restrictive:

![Algorithm for isValidDateFormat](image)

For our first coding increment, we:

1. Code a test wrapper.
2. Fully code `convertToValidDate` and `isValidDateFormat` following the algorithms given above.
3. Code `extractMoDayYr` and `isValidDate` as stubs.

```java
import static java.lang.Character.*;

public class ConvertToValidDateTestWrapper
{
    public static void main( String [] args )
    {
        new ConvertToValidDateTestWrapper( );
    }

    public ConvertToValidDateTestWrapper( )
    {
        int [] date;
        date = convertToValidDate( "12/07/1941" );
        System.out.println( );
    }
}`
public int [] convertToValidDate( String date )
/* Convert the given string to a date on the Gregorian
 * calendar.
 * Precondition: none.
 * Postcondition: If the given date is a true Gregorian
 * date in the form mm-dd-yyyy or mm/dd/yyyy, it returns
 * It returns null if:
 * date is null
 * date doesn't have the indicated form.
 * mm-dd-yyyy is not a true Gregorian date.
 */
{
    // check for null pointer
    if ( date == null ) return null;
    // check for expected form
    if ( !isValidDateFormat( date ) ) return null;
    // extract integers from the string
    int [] mdy = extractMoDayYr( date );
    // check for a real date
    if ( !isValidDate( mdy[0], mdy[1], mdy[2] ) )
        return null;
    // everything is A-OK
    return mdy;
}

public boolean isValidDateFormat( String date )
/* Determine if the given date has valid format;
 * specifically, ##-##-#### or ##/##/####.
 * Precondition: date is not null.
 * Postcondition: Returns true iff date has valid format.
 */
{
    // check length
    if ( date.length( ) != 10 ) return false;
    // check first two digits
    if ( !isDigit( date.charAt( 0 ) ) ) return false;
    if ( !isDigit( date.charAt( 1 ) ) ) return false;
// check for / or -
if ( date.charAt( 2 ) != '/' && date.charAt( 2 ) != '-' )
    return false;
// more digits
if ( !isDigit( date.charAt( 3 ) ) ) return false;
if ( !isDigit( date.charAt( 4 ) ) ) return false;
// another / or -
if ( date.charAt( 5 ) != '/' && date.charAt( 5 ) != '-' )
    return false;
// final four digits
if ( !isDigit( date.charAt( 6 ) ) ) return false;
if ( !isDigit( date.charAt( 7 ) ) ) return false;
if ( !isDigit( date.charAt( 8 ) ) ) return false;
if ( !isDigit( date.charAt( 9 ) ) ) return false;
return true;
}

public int[] extractMoDayYr( String date )
/* Extract integers from the given date representing
 * its month, day and year.
 * Precondition: date has form mm-dd-yyyy or mm/dd/yyyy.
 */
{
    return new int[3];
}

public boolean isValidDate( int month, int day, int year )
/* Determine if month/day/year is a true Gregorian date.
 * Precondition: none.
 * Postcondition: returns true iff month/day/year is true.
 */
{
    return true;
}
Exercises
These exercises have you employ the top-down procedural programming strategy to completely implement the date validation program.

1. Enter application ConvertToValidDateTestWrapper into a file. Save, compile and execute it. Fully test the convertToValidDate and isValidDateFormat methods. Because of the lower-level stubs, this will require some ingenuity and caution. Step through the program using the symbolic debugger and alter the stubs if needed. Complete a test diary that fully documents your testing activities.

2. Code extractMoDayYr and incorporate it into the test application. Fully test it and complete a test diary.

3. Complete the implementation and testing of the date validation program by implementing isValidDate.

This involves determining whether or not the month, day and year constitute a valid date on the Gregorian calendar. The common understanding of such a date is that the month must be 1 to 12, the day an integer from 1 to whatever is appropriate for the month (thirty days hath September, April, June and November; all the rest have 31 etc.) Let’s say the year must at least 1900. Here’s the algorithm:

```
Algorithm for isValidDate
1. if year < 1900 then
   return false
   end if
2. if month is not 1 through 12 then
   return false
   end if
3. if day is not 1 through whatever is appropriate for the month (and year if month is February) then
   return false
   end if
4. return true
```
For step 3, use a method to determine *whatever is appropriate for the month*, which itself requires a method to determine whether or not the year is a leap year. Here’s the structure chart.