Lecture 22:
Looping, Part II

Sierra College
CSCI-12
Spring 2015
Mon 04/20/15

Announcements

- **General**
  - Don't sit on the assignments: only one out at a time, but they will come one after the another, and build on each other
  - Get started on next assignment: the next one after this one will be out next Weds (one day overlap in lab)
- **Schedule**
  - Check schedule: some assignment start/due date tweaks
- **Past due assignments**
  - PRGM19: Age Utils, accepted thru Sun 4/26 @ 11pm (-3 pts/day)
- **New assignments**
  - PRGM22: Menu For Demo (due Weds 4/29 @ 11pm) lab this wk
    - An exercise in nested logic (*while*, *switch*, *for*, and *if-elseif*)
    - Update readInt() for type-safe data handling, and also add a readChar()
    - Create a user-controlled, menu-based "for-loop demo"
    - You will reuse this structure on the LAST assignment, understand how it works!

Lecture Topics

- **Last time**:
  - Looping in general
  - Event driven looping: *while* and *do-while* loops
- **Today**:
  - Endless looping
  - Count controlled looping: *for* loops
  - Examples needed for your next program
    - *while* loop wrapping a *switch* statement (keyboard input handling)
    - *for* looping

For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Program**
  - Get started on the next assignment
  - Suggestions:
    - First, review the new assignment before lab Weds
    - Start by updating your readInt() and creating readChar()
    - Then, implement a simple endless *while* loop
    - Then, control it using an update read of a *char*
    - Then, handle the keyboard input *char* in a *switch* statement
    - Then, add the *switch* cases (*int* reads and a *for* loop)
    - Then, add the needed logic around the *for* loop (*if-else* logic)

Type-Safe Input: The Problem

- *Scanner* can read in any types of data using its *next…()* methods
- But, what if there is a datatype mismatch against what's expected, or the user enters something unpredictable?
  - An *InputMismatchException* is generated
    - **Exceptions** are "bad news": something has gone wrong in the code
    - The program "barfs" and stops in its tracks
    - These are ugly, discomforting to users, and unprofessional
- With potentially unpredictable user I/O, this becomes a "defensive programming" issue

Type-Safe Input: Problem Example

- Input reading FAILS if the data is anything other than the expected *int*
- An exception **stack trace** details the calling sequence which has failed

```
----jGRASP exec: java LoopingWhileTypeUnsafe

Enter your age > 21
Your age is 21

----jGRASP: operation complete.

----jGRASP exec: java LoopingWhileTypeUnsafe

Enter your age > 21.0
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:909)
        at java.util.Scanner.next(Scanner.java:1530)
        at java.util.Scanner.nextInt(Scanner.java:2160)
        at java.util.Scanner.nextInt(Scanner.java:2119)
        at LoopingWhileTypeUnsafe.main(LoopingWhileTypeUnsafe.java:26)

----jGRASP wedge2: exit code for process is 1.
----jGRASP: operation complete.

----jGRASP exec: java LoopingWhileTypeUnsafe

Enter your age > twenty-one
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:909)
        at java.util.Scanner.next(Scanner.java:1530)
        at java.util.Scanner.nextInt(Scanner.java:2160)
        at java.util.Scanner.nextInt(Scanner.java:2119)
        at LoopingWhileTypeUnsafe.main(LoopingWhileTypeUnsafe.java:26)

----jGRASP wedge2: exit code for process is 1.
----jGRASP: operation complete.
```

```java
import java.util.Scanner;

public class LoopingWhileTypeUnsafe {

    public static void main(String [] args) {

        // declarations and initializations
        int age;
        Scanner input = new Scanner(System.in);

        // program is implicitly expecting
        // to read and echo an int value
        System.out.print("Enter your age > ");
        age = input.nextInt();
        System.out.println("Your age is " + age);

    } // end main

} // end class
```

See ***LoopingWhileTypeUnsafe.java*** in **Example Source Code**

Type-Safe Input: The Solution

- Check, before we read, that the next input token matches the expected input type
  - Use the ***Scanner*** class ***hasNext…()*** methods (next slide)
  - These "look ahead" to check next token's type, and return a *boolean* status
  - The **!*hasNext…()*** of the expected type (the "fail" case) becomes the *while* loop's **continuation condition**
- If next datatype matches the expected type:
  - **Safely read** the following value, and continue on
- If next datatype does NOT match the expected type:
  - **Flush** the next input using ***nextLine()*** (read it as a String and ignore it)
  - **Reprompt** the user, and take one more loop iteration
- **Using the above approach, we keep reprompting the user until we get the expected input datatype**

Scanner Class *hasNext…()* Methods

Return type	Method name and argument list
boolean	***hasNextInt( )***
boolean	***hasNextDouble( )***
boolean	***hasNextFloat( )***
boolean	***hasNextByte( )***
boolean	***hasNextShort( )***
boolean	***hasNextLong( )***
boolean	***hasNextBoolean( )***
boolean	***hasNext( )*** use this for String input

- Each method returns *true* if the next input stream token can be interpreted as the specified data type, or *false* otherwise.

Scanner Class *nextLine()* Method

Return type	Method name and argument list
String	***nextLine( )*** returns the remaining input on the line as a *String*

Pseudocode for any type-safe input:

```
prompt for input
while ( input does not match type requested ) {
   flush input  (or else it could loop endlessly)
   reprompt for input
}
read what is now known to be good input
```

Type-Safe Input: Solution Example

- Now, we “look ahead” first, to see if the input is of expected type
- If not, we “flush” it and keep retrying
- If input is good, we exit the loop and read safely

```
----jGRASP exec: java LoopingWhileTypeSafe

Enter your age > 50.0

Please enter an integer age > fifty

Please enter an integer age > 50
Your age is 50

----jGRASP: operation complete.
```

```
13 import java.util.Scanner;
14
15 public class LoopingWhileTypeSafe {
16
17     public static void main(String [] args) {
18
19         // declarations and initializations
20         int age;
21         Scanner input = new Scanner(System.in);
22         String garbage;
23
24         // program is implicitly expecting
25         // to read and echo an int value
26         System.out.print("Enter your age > ");
27
28         // loop until we get suitable input
29         while ( !input.hasNextInt() ) {
30             // flush input buffer
31             garbage = input.nextLine();
32             System.out.print("\nPlease enter an integer age > ");
33         }
34
35         // at this point, input is known to be an int
36         age = input.nextInt();
37         System.out.println("Your age is " + age);
38
39     } // end main
40
41 } // end class
```

See ***LoopingWhileTypeSafe.java*** in **Example Source Code**

Type-Safe Input From GUI

- For JOptionPane, things must be handled differently
- Now, we must use a ***try-catch*** block to intercept possible conversion **exceptions** arising from the wrapper class conversion
- A ***catch*** block allows for the graceful interception of exceptions, and possible corrective action
- Rather than the program terminating, we can simply reprompt and try again
- **Exception handling** will be fully covered in CS-13; this is just an advance preview…

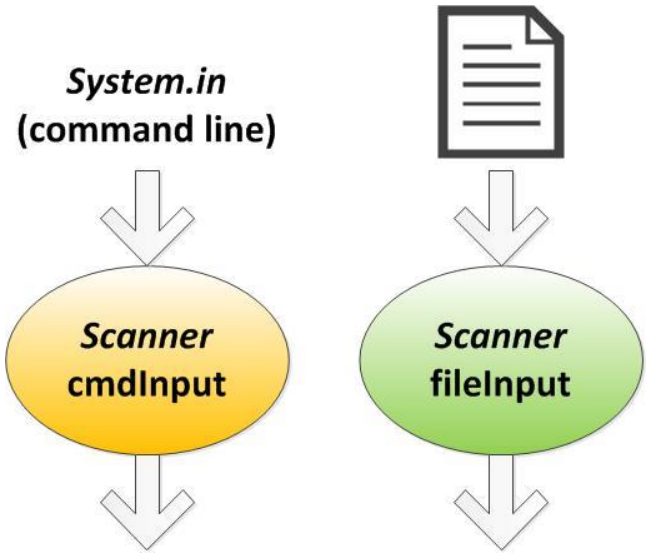
```
13 import javax.swing.JOptionPane;
14
15 public class LoopingWhileTypeSafeGUI {
16
17     public static void main(String [] args) {
18
19         // declarations
20         int age = 0;
21         String ageStr;
22         String prompt = "Please enter your age: ";
23         boolean failed = true;
24
25         while (failed == true) {
26             try {
27                 // this is fine: anything we enter can become a String
28                 ageStr = JOptionPane.showInputDialog(null, prompt);
29
30                 // this next statement could possibly fail
31                 age = Integer.parseInt(ageStr);
32
33                 // if we get here, good input, so toggle the flag
34                 failed = false;
35             }
36             catch(NumberFormatException nfe) {
37                 // intercepts a failed attempt to convert to an int
38             }
39         }
40
41         // echo the resulting input, and demonstrate it's really an int
42         JOptionPane.showMessageDialog(null,
43                                       "You are " + (age++) + " years old.\n" +
44                                       "Next year you will be: " + age);
45
46     } // end main
47
48 } // end class
```

See ***LoopingWhileTypeSafeGUI.java*** in **Example Source Code**

Looping Application: **Read From File**

- Looping allows us to read data in from a file
  - In general, we won't know ahead of time how many records there are
- For file reading, we need a couple new capabilities
  - A ***File*** object
  - An alternate constructor form of ***Scanner***, using a ***File*** object instead of standard input (*System.in*)
  - The ability of ***Scanner*** to "look ahead" and see if there is still data to be read
- A more complete treatment of file I/O will be covered in CS-13
  - This is just an initial preview

- *Scanner* allows us to set up alternate input pathways
  - And we can have multiple *Scanner* objects on hand for different input streams



Needed New Capabilities

- *Scanner* constructors

***Scanner* (*System.in*)** **[we've been using this]**
Creates a *Scanner* object associated with the standard input stream (the command line)

***Scanner* (*File fileObj*)** **[new]**
Creates a *Scanner* object associated with the specified *File* object

- Selected *Scanner* method

boolean hasNext()
"Looks ahead" to see if there is still additional data remaining in the input stream
Returns ***true*** if there is still data remaining in the file; returns ***false*** if EOF is reached

- *File* constructor

***File* (*String pathname*)**
Creates a *File* object associated with the specified file pathname

Looping Application: **Read From File**

- Note the required 2 new imports
- Line 19: use of a ***File*** object required the possibility of an exception to be acknowledged
- Once the 2nd ***File***-related ***Scanner*** object is created, use it to read data as you normally would

```
13 import java.util.Scanner;
14 import java.io.File;
15 import java.io.IOException;
16
17 public class LoopingWhileFileEcho {
18
19     public static void main(String [] args) throws IOException {
20
21         // declarations
22         String filename, text;
23         int numLines = 0;
24
25         // set up a Scanner to read an input filename
26         Scanner cmdInput = new Scanner(System.in);
27         System.out.print("Enter text file name: ");
28         filename = cmdInput.nextLine();
29
30         // set up a second Scanner to read from that file
31         File infile = new File(filename);
32         Scanner fileInput = new Scanner(infile);
33
34         // read and echo each line of the file
35         System.out.println("Reading from local file: " + filename + "\n");
36         while (fileInput.hasNext()) {
37             text = fileInput.nextLine();
38             numLines++;
39             System.out.println(text);
40         }
41         System.out.println("\nFinished, read in " + numLines + " lines");
42
43     } // end main
44
45 } // end class
```

```
----jGRASP exec: java LoopingWhileFileEcho
Enter text file name: zzz.txt
Reading from local file: zzz.txt

This is some data
34, 45, 56, 67
11/1/2014
Add one more line
Oh heck, add another

Finished, read in 5 lines

----jGRASP: operation complete.
```

See ***LoopingWhileFileEcho.java*** in **Example Source Code**

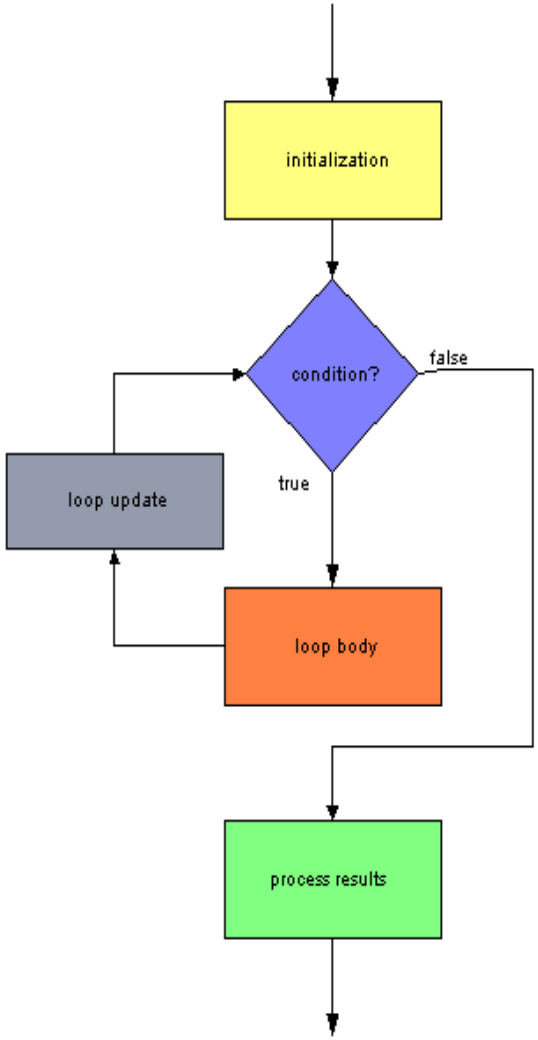
Looping Types Compared

- **Event-controlled looping** is used when we DON'T KNOW in advance how many loop iterations will be executed
  - Two types: ***while*** and ***do-while*** loops
  - Examples: read from keyboard input, read from file
- **Count-controlled looping** is used when we KNOW in advance exactly how many iterations will be executed
  - One type: ***for*** loops
  - Examples: giving the same raise percentage to all staff

Count-Controlled *for* Looping

- Examples of when a *for* loop may be useful:
  - When you know exactly how many iterations to perform:
    - Sum the numbers from 1 to 100
    - Find the maximum of 20 numbers
  - Or when you need to do something at specific intervals:
    - Convert Fahrenheit to Celsius at 5-degree intervals
    - Print the odd numbers from 1 to 100
  - Or if you need to count backwards
    - Some sort of countdown clock
    - Inverting text
  - Or if you need to index thru an array or list of data
    - We'll cover this a few lectures from now…

for Loop Flowchart and General Form



for (**loop initialization**; **loop condition**; **loop update**) {

    // loop body

    // repeated until the loop condition is *false*

}

// process results

NOTES:

1) Semicolons between loop header elements
2) No semicolon after loop header (empty loop)
3) Curly braces optional, but required if more than 1 loop statement

for Loop Operation

- **Loop initialization**:
  - Initialize the value of our **loop control variable**, i
  - Often these will be declared "on the spot"
    - A common exception to "declare vars up top"
  - Often these will be "throwaway" variables such as i, j, k, etc.
    - **A common exception to "meaningful variable names"**
- **Loop condition**:
  - On each iteration, the **loop condition** is checked
  - Is this *boolean* condition *true* or *false*?
- **Loop body:**
  - Executed ONLY IF the **loop condition** is true
  - Any variables declared in here have **local scope**
  - The **loop control variable** is also local, if it is declared in the **loop initialization** statement
- **Loop update**:
  - At the conclusion of each **loop body** execution, the **loop update** statement is executed

```
int start = 1;
int end = 10;
int incr = 1;
int count = 0;

for (int i=start; i <= end; i+=incr) {
   int square = i * i;
   System.out.println(i + "\t" + square);
   count++;
}
System.out.println(count + " times");
```

Output →

1	1
2	4
3	9

etc…

10 times

Example: Find Sum of N Integers

```
set limit N
set total to 0
for i = 1 to N by 1 {
   read number
   add number to total
 }
 print the total
```

NOTE: to use *total* (or any other variable) after the loop, it must be declared outside the loop

See ***LoopingForSum.java*** in **Example Source Code**

```
13 public class LoopingForSum {
14
15     public static void main(String [] args) {
16
17         // declarations
18         int max;
19         int total = 0;
20         int num;
21
22         // determine max limit
23         max = UtilsFL.readInt("Enter how many numbers to sum: ");
24
25         // sum numbers 1 to max
26         for (int i=1; i<=max; i++) {
27             num = UtilsFL.readInt("Enter number: ");
28             total += num;
29         }
30
31         // display the final total
32         System.out.println("The sum of these " + max +
33                            " numbers is: " + total);
34
35     } // end main
36
37 } // end class
```

```
----jGRASP exec: java LoopingForSum

Enter how many numbers to sum: 3
Enter number: 4
Enter number: 7
Enter number: 10
The sum of these 3 numbers is: 21

----jGRASP: operation complete.
```

Example: Update Increment > 1

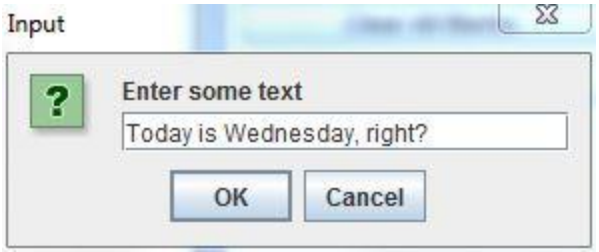
```
14 public class LoopingForEvens {
15
16     public static void main(String [] args) {
17
18         // initially empty print string
19         String printStr = "";
20
21         // assemble and print the evens up to 30
22         for (int i=0; i<=30; i+=2) {
23             printStr += (i + " ");
24         }
25         System.out.println(printStr);
26
27     } // end main
28
29 } // end class
```

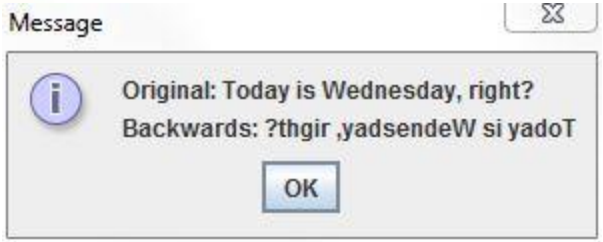
```
----jGRASP exec: java LoopingForEvens

0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

----jGRASP: operation complete.
```

See ***LoopingForEvens.java*** in **Example Source Code**

Example: Update Increment < 0

```
13 import javax.swing.JOptionPane;
14
15 public class LoopingForBackwards {
16
17     public static void main(String [] args) {
18
19         String forward;  // the original string
20         String backward = "";
21
22         forward = JOptionPane.showInputDialog(null,
23                                     "Enter some text");
24
25         // we always know exactly how long any string is
26         // build new string up from back to front, char by char
27         for (int i=forward.length()-1; i>=0; i--) {
28             backward += forward.charAt(i);
29         }
30
31         JOptionPane.showMessageDialog(null,
32                                 "Original: " + forward +
33                                 "\nBackwards: " + backward);
34     } // end main
35
36 } // end class
```





See ***LoopingForBackwards.java*** in **Example Source Code**

Example: Forward *String* Processing

Correct:

```
for ( int i = 0;  i < word.length( );  i++ ) {
    // processing actions here…
}
```

Incorrect:

```
for ( int i = 0;  i <= word.length( );  i++ ) {
    // processing actions here…
}
```

→ **There is no character at *word.length( )***

Example: Backward *String* Processing

Correct:

```
for ( int i = word.length( ) - 1;  i >= 0;  i-- ) {
    // processing goes here…
}
```

Incorrect:

```
 for ( int i = word.length( );  i >= 0;  i-- ) {
     // processing goes here…
 }
```

→ **There is no character at *word.length( )***

```
 for ( int i = word.length( ) - 1;  i > 0;  i-- ) {
     // processing goes here…
 }
```

→ **This does not process the first character (at *index* = 0)**

Example: Uniformly-Spaced Intervals

- This example demonstrates:
  - Loop parameters as variables
    - Initialized upstream, before looping
    - Could also have been user inputs
    - No matter how originated, bounds are known before looping begins
  - Method calls within each loop iteration
    - Within the loop body, any number of valid Java statements permitted

```
----jGRASP exec: java LoopingForIntervalOutput

°F      °C
50.0    10.0
55.0    12.8
60.0    15.6
65.0    18.3
70.0    21.1
75.0    23.9
80.0    26.7
85.0    29.4
90.0    32.2
95.0    35.0
100.0   37.8

----jGRASP: operation complete.
```

```
13 import java.text.DecimalFormat;
14
15 public class LoopingForIntervalOutput {
16
17     public static void main(String [] args) {
18
19         // one decimal place accuracy
20         DecimalFormat temps = new DecimalFormat("##0.0");
21
22         // temperature table range and delta
23         final char DEGS = 0x00B0;
24         int minTempF = 50;
25         int maxTempF = 100;
26         int delTempF = 5;
27
28         // print headers
29         System.out.println(DEGS + "F\t" + DEGS + "C");
30
31         // do this for each row of the table
32         for (int t=minTempF; t<=maxTempF; t+=delTempF) {
33
34             // method call on each loop iteration
35             double tempC = convertF2C(t);
36
37             // print data in 2-column F/C format
38             System.out.println(temps.format(t) + "\t" +
39                                temps.format(tempC));
40         }
41
42     } // end main
43
44     // convert temperature from Fahrenheit to Celsius
45     private static double convertF2C(double tempF) {
46         double tempC;
47         tempC = (5.0/9.0) * (tempF - 32);
48         return tempC;
49
50     } // end convertF2C
51
52 } // end class
```

See ***LoopingForIntervalOutput.java*** in **Example Source Code**

Testing Considerations With *for* Loops

- Make sure the starting and ending values of the loop are set correctly
- Make sure the loop executes the proper # of times:
  - for (int i=0; i < 5; i++) 5 times
  - for (int i=0; i <= 5; i++) 6 times, not 5 times
- When working with a *String*, don't exceed the length:
  - for (int i=0; i <= **str.length()**; i++) last char is at length()-1
- Test with data that causes the loop to execute 0 times (no iterations)
  - For example, test with an empty *String*

Loop Nesting

- Any types of loop can be nested within any other type of loop, to arbitrary depth:
  - *while* loops within *while* loops
  - *for* loops within *for* loops (frequently seen with matrices)
  - *while* loops within *for* loops, and vice versa
- Furthermore, selection structures (*if*, *if-else*, *switch*) can be nested within loops, and vice versa
- Between selection and looping, you can create any framework that is needed for your particular application's logic needs

Example: Nested *while* Loops

- The pseudocode at right represents an update to our earlier example of the grocery cashier
- **2 nested *while* loops**:
  - Outer loop: actions for each customer
  - Inner loop: actions for each customer's items
- Each loop has all the usual *while* loop elements
  - Priming read
    - Look for customer, reach for first item
  - Loop condition
    - Customer in line?, item not divider bar
  - Loop body
    - Scan customer's items, scan each item
  - Update read
    - Next customer, next item

```
look for a customer in line
while ( there is a customer in line ) {

    set total price to $0.00
    reach for first item
    while (item is not the divider bar) {
        add price to total
        reach for next item
    }
    output the total price

    look for another customer in line
}
```

Example: Nested *for* Loops

- Suppose we wanted to generate the output shown at the right. How would we go about it?
- This has the appearance of two nested *for* loops:
  - There are 5 rows, 1-5 (outer loop)
  - The highest # on each line is its row # (inner loop)
- Nested *for* loops may be (often are) coupled in some way
  - The looping index of an outer loop may be utilized within any inner loop
  - This is frequently seen in 2-D matrices

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

Example: Nested *for* Loops

- Two nested loops:
  - Outer loop from 1→12 on *day*
  - Inner loop, from 1→ *line*
- Here, the loops are **coupled**
  - Inner loop depends on the outer loop
  - The reverse is not possible:
    - Loop parameters within an inner loop are local in scope
    - Not visible outside that loop

```
15 public class LoopingForNested {
16
17     public static void main(String [] args) {
18
19         // gifts given during The Twelve Days of Christmas
20         int sum = 0;
21         final int DAYS = 12;
22
23         // outer loop: loops over all the days
24         for (int day=1; day<=DAYS; day++) {
25
26             // inner loop loops over the gifts on one day,
27             // in song order (N to 1)
28             for (int gift=day; gift>=1; gift--) {
29                 // print the gifts and tally them
30                 System.out.print(gift + " ");
31                 sum += gift;
32             }
33             System.out.println();   // newline after each day
34         }
35
36         // Report the total gift-giving aftermath
37         System.out.println("Total gifts given: " + sum);
38
39     } // end main
40
41 } // end class
```

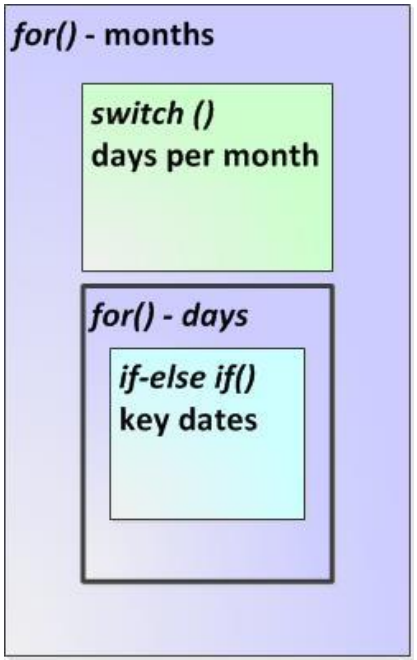
```
----jGRASP exec: java LoopingForNested

1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1
11 10 9 8 7 6 5 4 3 2 1
12 11 10 9 8 7 6 5 4 3 2 1
Total gifts given: 364

----jGRASP: operation complete.
```

See ***LoopingForNested.java*** in **Example Source Code**

Example: Hybrid Looping/Selection

- **Goal**: print all the days of the year and their numerical ordering. Also, flag some "key" dates.
- Nested logical constructs used:
  - *for*-looping
    - Loop over months
    - Loop over days in month
  - *switch*()
    - How many days in month?
  - *if-else if*
    - Identify "red-letter days"

```
110:    4/20/2015
111:    4/21/2015
112:    4/22/2015
113:    4/23/2015
114:    4/24/2015
115:    4/25/2015
116:    4/26/2015
117:    4/27/2015
118:    4/28/2015
119:    4/29/2015
120:    4/30/2015

121:    5/1/2015
122:    5/2/2015
123:    5/3/2015
124:    5/4/2015
125:    5/5/2015
126:    5/6/2015
127:    5/7/2015
128:    5/8/2015
129:    5/9/2015
130:    5/10/2015
131:    5/11/2015
132:    5/12/2015
133:    5/13/2015
134:    5/14/2015
135:    5/15/2015
136:    5/16/2015
137:    5/17/2015
138:    5/18/2015
139:    5/19/2015
140:    5/20/2015 <=== LAST DAY OF CLASS
141:    5/21/2015
142:    5/22/2015 <=== END OF SEMESTER
143:    5/23/2015
144:    5/24/2015
145:    5/25/2015
146:    5/26/2015
147:    5/27/2015
148:    5/28/2015
149:    5/29/2015
150:    5/30/2015
```



See ***LoopingForDayNumbers.java*** in **Example Source Code**