Repetition Structures

Chapter 8
Chapter Contents

8.1 Intro Example: The Punishment of Gauss
8.2 Repetition: The While Loop
8.3 Repetition: The Do Loop
8.4 Input Loops
8.5 Guidelines for Using Loops
8.6 Intro to Recursion
8.7 Graphical/Internet Java: A Guessing Game

Part of the Picture: Intro to Algorithm Analysis
Chapter Objectives

- Expand on intro to repetition, Chapter 4
- Examine *for* loops in more detail
- Compare, contrast *while* and *do* loops
- Introduce recursion
- Look at event-driven programming and state diagrams
- Take a first look at algorithm analysis
8.1 Introductory Example: the Punishment of Gauss

Problem:
As a young student, Gauss was disciplined with the task of summing the numbers from 1 through 100. He solved the problem almost immediately. We will learn his strategy later.

We will construct a method, that given $n$, will sum $1 + 2 + \ldots + n$
Object Centered Design

Objects:

<table>
<thead>
<tr>
<th>Objects</th>
<th>Kind</th>
<th>Type</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>limit value, n</td>
<td>varying</td>
<td>int</td>
<td>received</td>
<td>n</td>
</tr>
<tr>
<td>1+2+...+n</td>
<td>varying</td>
<td>int</td>
<td>returned</td>
<td></td>
</tr>
</tbody>
</table>

Specification of method

```java
public class Formula {
    public static int summation(int n) {
        // . . .
    }
}
```
Operations/Algorithm

1. Initialize \texttt{runningTotal} to 0
2. Initialize \texttt{count} to 1
3. Loop as long as \texttt{count} $\leq n$
   a. Add \texttt{count} to \texttt{runningTotal}
   b. Add 1 to \texttt{count}
4. Return \texttt{runningTotal}
for Loop Version of \texttt{summation()} Method

\begin{verbatim}
public static int summation(int n) {
    int runningTotal = 0;
    for (int count = 1; count <= n; count++)
        runningTotal += count;
    return runningTotal;
}
\end{verbatim}

Note driver program for \texttt{summation()}, Figure 8.2
Counter-controlled loops: loops where a set of statements is executed once for each value in a specified range.

```c
for (int count = n; count >= 0; count--)
    runningTotal += count; // descending form
```
The statement that appears within a `for` statement may itself be a `for` statement.

```java
for (int x = 1; x < lastX; x++)
    for (int y = 1; y <= lastY; y++)
    {
        product = x * y;
        theScreen.println("x * y = " + product);
    }
```

**Outer loop**

**Inner loop**
Warning

If the body of a counting loop alters the values of any variables involved in the loop condition, then the number of repetitions may be changed.

```java
for (int I = 0; I <= limit; I++)
{
    theScreen.println(I);
    limit++;
}
```

What happens in this situation?
Occasionally a need exists for a loop that runs for an indefinite number of times:

```c
for ( ; ; )
    statement;
```

The above statement will do this ... it will run forever ... unless ...

The body of the loop contains a statement that will terminate the loop when some condition is satisfied.
The **break** Statement

- Two forms
  - `break;`
  - `break identifier;`

- First form used to terminate execution of an enclosing loop or `switch` statement

- Second form used to transfer control to a statement with `identifier` as the label
  - `identifier : Statement;`
Use of the `break` Statement

```
for ( ; ; ) {
    statement;
    . . .
    if (termination_condition)
        break;
    . . .
    statement;
}
```

In either case, when the user enters 'Q' to Quit, the termination condition is met and the loop terminates.

Could be statements which offer a menu choice. Termination condition would be `choice == 'Q'`

Then these would be statements which process the menu choice when it is not 'Q'
The `continue` Statement

Two forms:
- `continue;`
- `continue label;`

First form transfers control to the innermost enclosing loop
- current iteration terminated, new one begins

Second form transfers control to enclosing labeled loop
- current iteration terminated, new one begins
Returning From a Loop

```java
for ( ; ; )
{
    theScreen.print(MENU);
    choice = theKeyboard.readChar();
    if (choice >=0 && choice <= 5)
        return choice;
    theScreen.println( "error .. ");
}
```

Assuming this forever loop is in a value returning method when one of options 0 – 5 is chosen.
- the loop is terminated and ...
- the menu choice returned by the function
- invalid choices keep the loop running
8.3 Repetition: The \texttt{while} loop

This is a looping structure that tests for the termination condition \textit{at the top} of the loop

also called a \texttt{pretest} loop

a simpler syntax than placing a \texttt{break} in an \texttt{if} statement at the beginning of a forever loop
Example: Follow the Bouncing Ball

Consider a ball that when dropped, it bounces to a height one-half to its previous height.

We seek a program which displays the number of the bounce and the height of the bounce, until the height of the bounce is very small.
For ball bounce results:
Enter height of ball drop -> 10

Bounce 1 = 5
Bounce 2 = 2.5
...

Prompt for and receive height of drop
Display bounce number and height of bounce
<table>
<thead>
<tr>
<th>Objects</th>
<th>Kind</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>current height</td>
<td>varying</td>
<td>real</td>
<td>height</td>
</tr>
<tr>
<td>bounce number</td>
<td>varying</td>
<td>int</td>
<td>bounce</td>
</tr>
<tr>
<td>a very small number</td>
<td>constant</td>
<td>real</td>
<td>SMALL_NUM</td>
</tr>
</tbody>
</table>
1. Initialize `bounce` to 0
2. Prompt, read value for `height`
3. Display original `height` value with label
4. Loop:
   a. if `height < SMALL_NUM`, terminate loop
   b. replace `height` with `height/2`
   c. add 1 to `bounce`
   d. display `bounce` and `height`
End Loop
Coding and Trial

Note source code, Figure 8.4 sample run

Note use of while statement

```java
while (height >= SMALL_NUMBER) {
    height *= REBOUND_FACTOR;
    bounce++;
    theScreen.println( ... );
}
```
Syntax

while (loop_condition) 
statement;

Where

- `while` is a keyword
- `loop_condition` is a boolean expression
- `statement` is a simple or compound statement
while (loop_condition)
statement;

1. loop_condition evaluated
2. If loop_condition is true
   statement executed
   control returns to step 1
Otherwise:
   Control transferred to statement following the while

Note – possible that statement is never executed – called "zero-trip behavior"
Loop Conditions vs. Termination Conditions

- **Forever loop**
  - continues repetition when condition is **false**
  - terminates when condition is **true**
- **while** loop is exactly opposite
  - continues repetition while condition is **true**
  - terminates when it goes **false**

**Warning for either case:**

Make sure condition is affected by some statement in the loop to eventually result in loop termination
8.4 Repetition: The **do** Loop

- **while** loop evaluates loop condition **before** loop body is executed.
- We sometimes need looping with a **posttest** structure.
  - The loop body will **always** execute at least once.
- Example: Making a Program Pause
  We seek a method that, given a length of time will make a program pause for that length of time.
Preliminary Analysis

- **System** class from *java.lang* provides a method `currentTimeMillis()`.
- Returns number of millisec since 1/1/1970.
- We will record the results of the function at the start of our pause method.
- Repeatedly view results to determine elapsed time.
- Called “busy–waiting” technique.
### Objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Kind</th>
<th>Type</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>num sec</td>
<td>varying</td>
<td>double</td>
<td>received</td>
<td>seconds</td>
</tr>
<tr>
<td>num millsec</td>
<td>varying</td>
<td>long</td>
<td></td>
<td>milliseconds</td>
</tr>
<tr>
<td>starting time</td>
<td>constant</td>
<td>long</td>
<td></td>
<td>START_TIME</td>
</tr>
<tr>
<td>current time</td>
<td>varying</td>
<td>long</td>
<td></td>
<td>currentTime</td>
</tr>
</tbody>
</table>

**Method specification**

```java
public class Controller {
    public static void pause(double seconds) {
        // . . .
    }
}
```
1. Receive *seconds*
2. Initialize `START_TIME`
3. If *seconds* > 0
   a. compute *milliseconds* from *seconds*
   b. loop
      get `currentTime`
      if `currentTime-START_TIME` > *milliseconds*
          terminate repetition
      End loop
  else
  display error message
Coding and Testing

Note source code Figure 8.5, driver Figure 8.6

Note looping mechanism

do
  currentTime = System.currentTimeMillis();
while (currentTime – START_TIME <= milliSeconds);
Syntax

do
    statement
while (loop_condition);

Where
- do and while are keywords
- statement is simple or compound
- loop_condition is a boolean expression
- note requirement of semicolon at end
Behavior

When execution reaches a do loop:

1. **statement** is executed
2. **loop_condition** is evaluated
3. if **loop_condition** is true
   control returns to step 1.
   otherwise
   control passes to 1\textsuperscript{st} statement following loop structure
Loop Conditions vs. Termination Conditions

- **do** loop and **while** loop both use loop condition that continues repetition as long as it is true and terminates repetition when false.
- **forever** loop does the opposite.
- If **do** loop or **while** loop used $x \leq 7$ then forever loop would use $x > 7$ (exact opposite comparison).
Input do Loops: The Query Approach

- Recall two different types of input loops
  - counting approach
  - sentinel approach

  Counting approach asked for number of inputs to be entered, used `for()` loop
  - requires foreknowledge of how many inputs

  Sentinel approach looked for special valued input to signify termination
  - requires availability of appropriate sentinel value
Query Approach

- Use a do loop
- Loop body always executed at least one time
- Query user at end of loop body
- User response checked in loop condition

```java
do {
    // whatever ...
    
    theScreen.print("More inputs? (Y/N) : ");
    response = theKeyboard.readChar();
} while (response=='y' || response =='Y');
```
Query Methods

- Note the excess code required in the loop to make the query to check results.
- This could be simplified by writing a method to do the asking.
  - Method returns boolean result.
  - Use call of query method as the loop condition.

```java
do { ...
    ... } while (Query.moreValues());
```
8.5 Choosing the Right Loop

- Determined by the nature of the problem

Decisions to be made:

1. Counting or general loop?
   - Ask: does algorithm require counting through fixed range of values?
2. Which general loop?
   - pretest or posttest
   - forever loop with test in mid-loop
Introduction to Recursion

- We have seen one method call another method
- Most often the calling method is `main()`
- It is also possible for a method to call itself
- This is known as recursion
Example: Factorial Problem Revisited

Recall from section 5.3
Given an integer \( n \), calculate \( n \)-factorial
\[
1 \times 2 \times 3 \times \ldots \times (n - 1) \times n
\]
One way to define factorials is
\[
n! = \begin{cases} 
1 & n = 0 \\
n \times (n - 1)! & n > 0 
\end{cases}
\]
This is a recursive definition
Recursive Definitions

\[ n! = \begin{cases} 
1 & n = 0 \\
 n \times (n - 1)! & n > 0 
\end{cases} \]

An operation is defined recursively if:

- it has an anchor or base case
- it has an inductive or recursive step where the current value produced is defined in terms of previously defined results
Recursive Method for $n!$

```java
public static int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

- Consider: what happens when $n < 0$?
- Why is this called infinite recursion?
Example 2: Recursive Exponentiation

Raising a number to an integer power can be also be done with recursion.

\[ x^n = \begin{cases} 
1 & n = 0 \\
 x^{n-1} \times x & n > 0 
\end{cases} \]

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>base value</td>
<td>double</td>
<td>varying</td>
<td>received</td>
<td>x</td>
</tr>
<tr>
<td>exponent</td>
<td>int</td>
<td>varying</td>
<td>received</td>
<td>n</td>
</tr>
<tr>
<td>(x^n)</td>
<td>double</td>
<td>varying</td>
<td>returned</td>
<td></td>
</tr>
</tbody>
</table>
Recursive Exponentiation Method

```java
public static double power(double x, int n) {
    if (n == 0)
        return 1.0;
    else if (n > 0)
        return power(x, n - 1) * x;
    else { theScreen.println("error");
        return 1.0; }
}
```

What keeps this method from infinite recursion?
8.7 Graphical/Internet Java: A Guessing Game

Twenty Guesses:

- One player thinks of an integer
- Second player allowed up to 20 guesses to determine the integer
- Incorrect guess:
  - tell the other player whether guess is high or low
  - other player uses this information to improve guess
Problem: Twenty Guesses

- We seek a program using GUI to play the role of the guesser.
- Strategy used is binary-search.
- Guesser knows high and low bounds of search.
- Guesses halfway.
- Use high/low feedback to guess halfway of smaller bound.
Behavior of Program: Transition Diagram

- **Twenty Guesses**
  - Think of an integer
  - **Begin**
  - **Quit**

- **Twenty Guesses**
  - My guess is X
  - **Lower**
  - **Equal**
  - **Higher**
  - **Reset**
  - **Quit**

- **Twenty Guesses**
  - I win!
  - **Reset**
  - **Quit**

- **Twenty Guesses**
  - I lose!
  - **Reset**
  - **Quit**

- **Twenty Guesses**
  - Exceed 20 Questions
  - **Equal**
  - **Reset**

- **Twenty Guesses**
  - Quit

- **Twenty Guesses**
  - Quit
<table>
<thead>
<tr>
<th>Object</th>
<th>Type</th>
<th>Kind</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td></td>
<td>varying</td>
<td></td>
</tr>
<tr>
<td>prompt</td>
<td>JLabel</td>
<td>varying</td>
<td>myPromptLabel</td>
</tr>
<tr>
<td>Prompt message</td>
<td>String</td>
<td>varying</td>
<td></td>
</tr>
<tr>
<td>&quot;Begin&quot; button</td>
<td>JButton</td>
<td>varying</td>
<td>myBeginButton</td>
</tr>
<tr>
<td>&quot;Quit&quot; button</td>
<td>JButton</td>
<td>varying</td>
<td>myQuitButton</td>
</tr>
<tr>
<td>&quot;Lower&quot; button</td>
<td>JButton</td>
<td>varying</td>
<td>myLowerButton</td>
</tr>
<tr>
<td>&quot;Equal&quot; button</td>
<td>JButton</td>
<td>varying</td>
<td>myEqualButton</td>
</tr>
<tr>
<td>&quot;Higher&quot; button</td>
<td>JButton</td>
<td>varying</td>
<td>myHigherButton</td>
</tr>
<tr>
<td>&quot;Reset&quot; button</td>
<td>JButton</td>
<td>varying</td>
<td>myResetButton</td>
</tr>
<tr>
<td>a guess</td>
<td>int</td>
<td>varying</td>
<td>myGuess</td>
</tr>
<tr>
<td>Count of guesses</td>
<td>int</td>
<td>varying</td>
<td>myGuessCount</td>
</tr>
<tr>
<td>low bound</td>
<td>int</td>
<td>varying</td>
<td>myLoBound</td>
</tr>
<tr>
<td>high bound</td>
<td>int</td>
<td>varying</td>
<td>myHiBound</td>
</tr>
</tbody>
</table>
Operations

1. We need a constructor to build the GUI
2. An `actionPerformed()` method
   a. implement the `ActionListener` interface
   b. register itself as listener for each button
   c. send `addActionListener()` message to each button
3. A `main()` method
   a. create an instance of the class
   b. make it visible
A transition diagram with most details removed

- **Starting State**
  - Begin
  - Quit
  - Reset

- **Guessing State**
  - Higher, Lower
  - Equal
  - Exceed 20 Questions

- **Win State**
  - Quit
  - Reset

- **Lose State**
  - Quit
  - Reset
Coding

- Write a method for each state
  - define its appearance in that state
- **JButtons** have `setText()` method for setting label attribute
- button with “Begin” in starting state has “Reset” in the other states
- Note full source code Figure 8.14
Applet Version of GUI Guessing Program

- Make the class extend `JApplet` instead of `CloseableFrame`.
- Replace `main()` with non-static `init()` method.
- Adjust dimensions of applet frame in HTML file to resemble frame for application.
How did Gauss figure the sum so quickly?

Consider
\[
\begin{align*}
\text{sum} &= 1 + 2 + \ldots + 99 + 100 \\
\text{sum} &= 100 + 99 + \ldots + 2 + 1
\end{align*}
\]

Thus
\[
2 \times \text{sum} = 101 + 101 + \ldots + 101 + 101
\]
100 terms

So
\[
\text{sum} = \frac{100 \times 101}{2} = 5050
\]
Analyzing the Algorithms

In general, the formula is:

\[ \text{sum} = \frac{n \times (n - 1)}{2} \]

This is more efficient than the looping algorithm

- many less operations (additions, assignments, increments, comparisons)
- This algorithm actually has same number of operations regardless of the value of \( n \)

Important to analyze algorithms for efficiency

- evaluate number of operations required