Mat 2170
Chapter Four – Part A
Control Statements – Iteration
Spring 2014

Week 4
Student Responsibilities
▶ Reading: Textbook, Chapter 4.1 – 4.2, 4.5 – 4.6
▶ Lab preparation & lab
▶ Attendance

Chapter Four Overview: 4.1 – 4.4
▶ A little review
▶ Java statement types
▶ Control statements and problem solving
▶ The while statement
▶ The for statement

Compound Assignment Statements
There are five forms of the compound assignment statement:
+=, -=, *=, /=, and %=.

<table>
<thead>
<tr>
<th>Before</th>
<th>Assignment</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 2;</td>
<td>i += 3;</td>
<td>i is 5</td>
</tr>
<tr>
<td>i is 2</td>
<td></td>
<td>i is 5</td>
</tr>
<tr>
<td>i is 5</td>
<td>i -= 1;</td>
<td>i is 4</td>
</tr>
<tr>
<td>i is 4</td>
<td>i *= 3;</td>
<td>i is 12</td>
</tr>
<tr>
<td>i is 12</td>
<td>i /= 3;</td>
<td>i is 4</td>
</tr>
<tr>
<td>i is 4</td>
<td>i %= 4;</td>
<td>i is 0</td>
</tr>
</tbody>
</table>

Increment and Decrement
It is often the case that we wish to add or subtract one from a numeric object. There are many equivalent statements to accomplish this.

To add one to int object k:

```java
k = k + 1; k += 1; k++; ++k;
```

To subtract one from object k:

```java
k = k - 1; k -= 1; k--; --k;
```

The boolean type
▶ One of the built-in primitive data types
▶ Has two values only: true and false
▶ Is useful for loops—the while statement
▶ Is useful for conditionals—the if statement
▶ Examples of Boolean objects:
  boolean doAgain = true;
  boolean bigger = false;
▶ The type of the parameter passed in the GRect and GOval message setFilled():

```java
GRect MyRect = new GRect(x, y, w, h);
MyRect.setFilled(true);
```

Boolean operators — AND
▶ The operator + is used to combine two numeric objects
▶ The operator && is used to combine two boolean objects:

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P &amp;&amp; Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Both operands must be true to obtain true.
This is similar to good parenting when both parents must say “Yes.”
Boolean operators — OR

The operator || is used to combine two boolean objects:

| P  | Q  | P || Q |
|----|----|-------|
| true | true | true  |
| true | false | true  |
| false| true | true  |
| false| false | false |

If either operand is true, the result is true. This is similar to not-so-good parenting, where only one must say "Yes."

Boolean operators — NOT

The operator ! is used to negate one boolean object:

<table>
<thead>
<tr>
<th>P</th>
<th>!P</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

The ! operator simply "flips" the truth value.

Relational Operators

<table>
<thead>
<tr>
<th>Mathematics</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>≤</td>
<td>&lt;=</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>≥</td>
<td>&gt;=</td>
</tr>
<tr>
<td>≠</td>
<td>!=</td>
</tr>
<tr>
<td>=</td>
<td>==</td>
</tr>
</tbody>
</table>

Notes:
- The result of a relational operator is a boolean value
- Testing for equality requires the == operator.
- The = operator is used for assignment.

Examples of relational operators

Assume these declarations are in effect:

```java
int i = 1;
int j = 2;
int k = 2;
```

<table>
<thead>
<tr>
<th>Boolean Expression</th>
<th>true or false?</th>
<th>Boolean Expression</th>
<th>true or false?</th>
</tr>
</thead>
<tbody>
<tr>
<td>i &lt; j</td>
<td>true</td>
<td>i &lt;= (j + k)</td>
<td>false</td>
</tr>
<tr>
<td>j == k</td>
<td>j &lt;= k</td>
<td>i == k</td>
<td>true</td>
</tr>
<tr>
<td>j &lt; k</td>
<td>i &lt;= k</td>
<td>j != 2</td>
<td>false</td>
</tr>
<tr>
<td>i<em>i &gt; k</em>k</td>
<td>true</td>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>

Java Statement Types

- Java programs consist of a set of classes.
- Classes contain methods, and each method consists of a sequence of statements.
- There are three basic types of Java statements:
  1. Simple
  2. Compound
  3. Control

- Simple statements are formed by adding a semicolon (;) to the end of a Java expression
- Compound statements (aka blocks) consist of a sequence of statements enclosed in curly braces: `{ }`
- Control statements fall into two categories:
  1. Conditional (selection) statements that make choices
  2. Iterative (looping) statements that specify repetition
Control Statements and Problem Solving

Before looking at the details of control statements, it may help to look at common control patterns — when and how they are used.

We will extend the Add2Integers program from lab 1 to create programs that add longer lists of integers.

We will illustrate three different strategies:

1. Add new code to process each (additional) input value
2. Repeat the input cycle a predetermined number of times (for loop)
3. Repeat the input cycle until a special sentinel value is entered by the user (while loop)

The Add4Integers Problem

At this point, the only way to increase the number of inputs is to add new statements for each one:

```java
public class Add4Integers extends ConsoleProgram {
    public void run() {
        println("This program adds four numbers.");
        int n1 = readInt("Enter first number: ");
        int n2 = readInt("Enter second number: ");
        int n3 = readInt("Enter third number: ");
        int n4 = readInt("Enter fourth number: ");
        int total = n1 + n2 + n3 + n4;
        println("The total is " + total + ".");
    }
}
```

This strategy is difficult to generalize and would be cumbersome if we needed to add 100 values!

The Repeat–N–times Pattern

The Repeat–N–times Pattern:
execute a set of statements a specified number of times.

The general form of the pattern:

```java
for (int i = 0; i < repsDesired; i++) {
    statements to be repeated;
}
```

The AddNIntegers Program

This program uses the Repeat–N–Times pattern to compute the sum of a predetermined number of integer values, specified by the named constant N.

```java
public class AddNIntegers extends ConsoleProgram {
    private static final int N = 100;
    public void run() {
        println("This program adds " + N + " numbers.");
        int total = 0;
        for (int i = 0; i < N; i++) {
            int value = readInt("Enter number "+i+": ");
            total += value;
        }
        println("The total is " + total + ".");
    }
}
```

The Repeat–Until–Sentinel Pattern

The programs on the previous slides haven’t been flexible: you must add either four integers, or 100 integers. Sometimes we may not know how many integers are on the list to sum.

The Repeat–Until–Sentinel Pattern executes a set of statements until the user enters a specific value, called a sentinel, to signal the end of the list:

```java
while (value != sentinel) {
    process value;
    prompt user and read in a value;
}
```
This approach works for any number of values.

The sentinel value chosen should not be a possible legitimate data value.

Define the sentinel as a named constant to make it easy to change.

Note the initialization of value before the loop, then the same prompt and read inside the loop.

The AddIntegerList Program

Compute the sum of a list of non–negative integer values:

```java
public class AddIntegerList extends ConsoleProgram {
    public void run() {
        println("Add a list of non-negative integers.");
        println("Enter one value per line, "+SENTINEL);
        println("to signal the end of input.");
        int total = 0;
        int value = readInt("Enter number, "+SENTINEL);
        while (value != SENTINEL) {
            total += value;
            value = readInt("Enter number, "+SENTINEL);
        }
        println("The total is "+total+". ");
    }
    private static final int SENTINEL = -1;
}
```

Exercise: Control Patterns

Using the AddIntegerList program as a basis, write a new AverageList program that reads a set of non–negative integers from the user and displays their average.

It is important to keep in mind that the average of a set of integers may well not be an integer itself.

The AverageList program will require the following changes:

- Convert the value of total to a double before computing the average
- Keep a count of the number of input values, along with the sum
- Update the user messages and program documentation

```java
public class AverageList extends ConsoleProgram {
    public void run() {
        println("Average a list of non-negative integers.");
        println("Enter one value per line, "+SENTINEL);
        println("to signal the end of input.");
        int total = 0;
        *** //counter
        int value = readInt("Enter number, "+SENTINEL);
        *** //update
        while (value != SENTINEL) {
            *** //calculate
            total += value;
            *** //display
            value = readInt("Enter number, "+SENTINEL);
        }
        *** //calculate
        *** //display
        private static final int SENTINEL = -1;
    }
}
```

The while loop

When Java encounters a while statement, it begins by evaluating the condition in parentheses, which must have a boolean value.

If the value of condition is true, Java executes the statements in the body of the loop.

At the end of each cycle, Java re–evaluates condition to see whether its value has changed.

If the condition evaluates to false, Java exits from the loop and continues with the statement following the closing brace at the end of the while body.
### A simple while loop

```c
int n = 1;
int sum = 0;
while (n <= 7)
{
    sum += n;
    n += 2;
}
```

**Desk check:**

<table>
<thead>
<tr>
<th></th>
<th>Initially</th>
<th>1st pass</th>
<th>2nd pass</th>
<th>3rd pass</th>
<th>4th pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>0</td>
<td>0 + 1</td>
<td>0 + 1 + 3</td>
<td>0 + 1 + 3 + 5</td>
<td>0 + 1 + 3 + 5 + 7</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>1 → 3</td>
<td>3 → 5</td>
<td>5 → 7</td>
<td>7 → 9</td>
</tr>
</tbody>
</table>

### Questions to consider when writing while loops

- Which objects need to be initialized before the loop begins?
- Does the Boolean expression match what is needed? Have we checked for off by one errors? (E.g., < vs <=)
- Does the loop body do what is needed? Are the correct objects updated during each pass?
- Has a desk check been performed?
- Will any of the objects declared in the loop body be needed later in the program? (scope)

### Another while loop

```c
int n = 1;
int sum = 0;
while (n < 7)
{
    sum += n;
    n++;
}
```

The object `n` increases by 1 on each loop iteration

- `n`: 1 → 2 → 3 → 4 → 5 → 6 → 7
- the loop body is performed 6 times, `n = 1` through `n = 6`
- When `n = 7`, execution drops out of the loop
- Final value of sum:

### An Equivalent while loop

```c
int n = 1;
int sum = 0;
while (n <= 6)
{
    sum += n;
    n++;
}
```

The object `n` increases by 1 on each loop iteration

- `n`: 1 → 2 → 3 → 4 → 5 → 6 → 7
- the loop body is performed 6 times, `n = 1` through `n = 6`
- When `n = 7`, execution drops out of the loop
- Final value of sum:

### Order of Statements is Important

```c
int n = 1;
int sum = 0;
while (n <= 6)
{
    n++;
    sum += n;
}
```

How does switching the increment of `n` with the update of `sum` change the outcome?

### A while loop which counts down

```c
int n = 7;
int sum = 0;
while (n > 0)
{
    sum += n;
    n--;
}
```

`n` decreases by 1 on each loop iteration

- `n`: 7 → 6 → 5 → 4 → 3 → 2 → 1 → 0
- the loop body is performed 7 times, `n = 7` through `n = 1`
- When `n = 0`, execution drops out of the loop
- Final value of sum:
Accumulating a Product

```java
int n = 1;
int product = 1;
while (n <= 3)
{
    product *= n;
    n++;
}
```

**Desk check:**

<table>
<thead>
<tr>
<th>Initially</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>1 → 2</td>
<td>3 → 4</td>
</tr>
</tbody>
</table>

Infinite Loops

Loops which have no chance of terminating are called infinite loops. The type of errors which cause this will not be detected by the compiler.

**Possible Causes**
- Loop body doesn’t change objects tested in the loop condition
- Loop control object skips over the limit value
- Loop control object moves away from the limiting value
- Accidental use of = in place of ==
- Accidental placement of semicolon after the loop header — while (BooleanExpression); for (init : test : step);

For Statement

```java
for (init ; test ; step)
{
    statements to be repeated
}
```

Java evaluates a for statement by executing the steps:

1. **Evaluate** init, typically a control variable declaration
2. **Evaluate** test and exit from loop if the value is false
3. **Execute the body** of the loop
4. **Evaluate** step, which usually updates the control variable
5. **Return to step 2** to begin the next loop cycle

These Loops Are Functionally Equivalent

```java
for (init ; test ; step)
{
    statements to be repeated
}
```

```java
init;
while (test)
{
    statements to be repeated
    step;
}
```

The advantage of the for statement is that everything you need to know to understand how many times the loop will execute is explicitly included in the header line.

To output the integers 1 . . . 10

- as a while loop
  ```java
  int i = 1;
  while (i <= 10){
      println(i);
      i++;
  }
  ```

- as a for loop
  ```java
  for (int i = 1; i <= 10; i++)
      println(i);
  ```
Summing multiples of 5

// Initialize data
int N = readInt("Enter an integer: ");
int Sum = 0;

// Sum multiples of 5
for (int i = 1; i <= N; i++)
    Sum += 5*i;

// Display answer
println("The sum of the first "+ N + " multiples of 5 is "+ Sum);

Analyzing for Statement Headers

Describe the effect of each of the following:
1. for (int i = 1; i <= 10; i++)
2. for (int i = 0; i < N; i++)
3. for (int n = 99; n >= 1; n -=2)
4. for (int x = 1; x < 1024; x *= 2)

Working with integers and a while loop

Give Java statements that:
- Print the digits in an integer, n, in reverse order
- Sum the digits in an integer, n

Nested for Statements

- The body of a control statement can contain other statements, which are said to be nested within the control statement.
- Many applications require nested loops – for example, displaying a checkerboard pattern.
- The for loops in the Checkerboard program:

```java
for (int row = 0; row < N_ROWS; row++)
    {
        for (int col = 0; col < N_COLUMNS; col++)
            {
                double x = col * sqSize;
                double y = row * sqSize;
                GRect sq = new GRect(x, y, sqSize, sqSize);
                sq.setFilled((row + col) % 2 != 0);
                sq.setFillColor(Color.RED);
                add(sq);
            }
    }
```

- Because the entire inner loop runs for each cycle of the outer loop, the program displays N_ROWS × N_COLUMNS squares.

The Checkerboard Program

// determine size of one square
double sqSize = (double) getHeight() / N_ROWS;

// Display an N_ROWS by N_COLUMNS grid
// Outside loop controls row (and thus current y-value),
// from top to bottom
for (int row = 0; row < N_ROWS; row++)
    {
        // In current row, displays each block, left to right
        for (int col = 0; col < N_COLUMNS; col++)
            {
                double x = col * sqSize;
                double y = row * sqSize;
                GRect sq = new GRect(x, y, sqSize, sqSize);
                sq.setFilled((row + col) % 2 != 0);
                sq.setFillColor(Color.RED);
                add(sq);
            }
    }
```

Example Executions

N_ROWS = 5, N_COLUMNS = 5
N_ROWS = 8, N_COLUMNS = 3
N_ROWS = 3, N_COLUMNS = 8, (window width modified to show entire board)
Triangle Number Table

We wish to write a program that displays the sum of the first $n$ integers, as $n$ runs from 1 to 10. Such numbers are called triangle numbers:

1 = 1  
1 + 2 = 3  
1 + 2 + 3 = 6  
1 + 2 + 3 + 4 = 10  
1 + 2 + 3 + 4 + 5 = 15  
1 + 2 + 3 + 4 + 5 + 6 = 21  
1 + 2 + 3 + 4 + 5 + 6 + 7 = 28  
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36  
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45  
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55

Triangle Number Design Issues

▶ Can the problem be solved with a single loop? Why / why not?
▶ The outer loop has to run through each of the values from 1 to the maximum, MAX.
▶ Another loop is needed to print a series of values on each line.
▶ print is similar to println, but doesn’t return the cursor to the beginning of the next line.
▶ The $n^{th}$ output line contains $n$ values before the equal sign, but only $n-1$ plus signs. To avoid the problem of a “+” sign after the last term, we wait to print it after the inner loop has finished.

TriangleTable Program

// Display MAX_VALUE lines of initial triangle numbers
for (int line = 1; line <= MAX_VALUE; line++)
{  // Display current line
    int total = 0;
    for (int term = 1; term < line; term++)
    {  // Display the current term's value in the current line
        print(term + " + ");
        total += term;
    }
    // Last term (line), to avoid too many "+", signs
    println(line + " = " + (total + line));
} // end of run()

// Constant declaration section
private static final int MAX_VALUE = 10;

Simple Graphical Animation

▶ Loops make it possible to implement simple graphical animation.
▶ The basic strategy is to create a set of graphical objects and then execute the following loop:

```java
for (int i = 0; i < N_STEPS; i++) {
    update GOObjects by small amount;
    pause(PAUSE_TIME);
}
```

▶ On each cycle of the loop, this pattern updates each animated object by moving it slightly or changing some other property of the object, such as its color. Each cycle is called a time step.
▶ After each time step, pause is invoked to slow the animation to human time. PAUSE_TIME is an integer constant given in milliseconds.

The AnimatedSquare Program

// Move a square from the top-left corner to the bottom-right corner of the graphics window in N_STEPS
for (int i = 0; i < N_STEPS; i++)
{  // Create and display square
    GRect square = new GRect(0.0,0.0,SQUARE_SIZE,SQUARE_SIZE);
    square.setFilled(true);
    square.setFillColor(Color.RED);
    add(square);

    // Calculate displacement
    double dx = (double) (getWidth() - SQUARE_SIZE) / N_STEPS;
    double dy = (double) (getHeight() - SQUARE_SIZE) / N_STEPS;

    // Animate square
    for (int j = 0; j < N_STEPS; j++)
    {
        square.move(dx, dy);
        pause(PAUSE_TIME);
    } // end for-animation

// Calculate displacement
double dx = (double) (getWidth() - SQUARE_SIZE) / N_STEPS;
double dy = (double) (getHeight() - SQUARE_SIZE) / N_STEPS;
```

Taking One Step...

Suppose we want to move a square half its width and height. How can we find $\Delta x$, $\Delta y$, and position $Q$?