Chapter 2 - Control Structures
Control Structures

• Sequential execution
  – Program statements execute one after the other

• Transfer of control
  – Three control statements can specify order of statements
    • Sequence structure
    • Selection structure
    • Repetition structure

Corresponding Java statement:

```java
add grade to total

total = total + grade;
```

```java
add 1 to counter

counter = counter + 1;
```
### Java Keywords

<table>
<thead>
<tr>
<th>abstract</th>
<th>assert</th>
<th>boolean</th>
<th>break</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
<td>catch</td>
<td>char</td>
<td>class</td>
<td>continue</td>
</tr>
<tr>
<td>default</td>
<td>do</td>
<td>double</td>
<td>else</td>
<td>extends</td>
</tr>
<tr>
<td>final</td>
<td>finally</td>
<td>float</td>
<td>for</td>
<td>if</td>
</tr>
<tr>
<td>implements</td>
<td>import</td>
<td>instanceof</td>
<td>int</td>
<td>interface</td>
</tr>
<tr>
<td>long</td>
<td>native</td>
<td>new</td>
<td>package</td>
<td>private</td>
</tr>
<tr>
<td>protected</td>
<td>public</td>
<td>return</td>
<td>short</td>
<td>static</td>
</tr>
<tr>
<td>strictfp</td>
<td>super</td>
<td>switch</td>
<td>synchronized</td>
<td>this</td>
</tr>
<tr>
<td>throw</td>
<td>throws</td>
<td>transient</td>
<td>try</td>
<td>void</td>
</tr>
<tr>
<td>volatile</td>
<td>while</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Keywords that are reserved, but not currently used*

| const | goto |

*Java keywords.*
Control Structures

• Java has a sequence structure “built-in”

• Java provides three selection structures
  – if
  – If...else
  – switch

• Java provides three repetition structures
  – while
  – do...while
  – For

• Each of these words is a Java keyword
if Single-Selection Statement

- Single-entry/single-exit control structure
- Perform action only when condition is true
- Action/decision programming model

if...else Selection Statement

- Perform action only when condition is true
- Perform different specified action when condition is false
- Conditional operator (?:)
- Nested if...else selection structures
(Counter-Controlled Repetition)

- **Counter**
  - Variable that controls number of times set of statements executes

- **Average1.java** calculates grade averages
  - uses counters to control repetition
// Fig. 4.7: Average1.java
// Class-average program with counter-controlled repetition.
import javax.swing.JOptionPane;

public class Average1 {
    public static void main( String args[] ) {
        int total; // sum of grades input by user
        int gradeCounter; // number of grade to be entered next
        int grade; // grade value
        int average; // average of grades
        String gradeString; // grade typed by user

        // initialization phase
        total = 0; // initialize total
        gradeCounter = 1; // initialize loop counter

        // processing phase
        while ( gradeCounter <= 10 ) { // loop 10 times
            // prompt for input and read grade from user
            gradeString = JOptionPane.showInputDialog(
                "Enter integer grade: ");

            // convert gradeString to int
            grade = Integer.parseInt( gradeString );
        }
    }
}
total = total + grade;  // add grade to total
gradeCounter = gradeCounter + 1;  // increment counter

} // end while

// termination phase
average = total / 10;  // integer division

// display average of exam grades
JOptionPane.showMessageDialog( null, "Class average is " + average, "Class Average", JOptionPane.INFORMATION_MESSAGE );

System.exit( 0 ); // terminate the program

} // end main

} // end class Average1
Average1.java
(Sentinel-Controlled Repetition)

• Sentinel value
  – Used to indicated the end of data entry

• `Average2.java` has indefinite repetition
  – User enters sentinel value (`-1`) to end repetition
// Fig. 4.9: Average2.java
// Class-average program with sentinel-controlled repetition.
import java.text.DecimalFormat;  // class to format numbers
import javax.swing.JOptionPane;

public class Average2 {

    public static void main( String args[] )
    {
        int total;    // sum of grades
        int gradeCounter;    // number of grades entered
        int grade;    // grade value

        double average;  // number with decimal point for average
        String gradeString;  // grade typed by user

        // initialization phase
        total = 0;        // initialize total
        gradeCounter = 0;  // initialize loop counter

        // processing phase
        // get first grade from user
        gradeString = JOptionPane.showInputDialog(  
            "Enter Integer Grade or -1 to Quit:" );

        // convert gradeString to int
        grade = Integer.parseInt( gradeString );
    }
// loop until sentinel value read from user
while ( grade != -1 ) {
    total = total + grade;
    gradeCounter = gradeCounter + 1;
}

// get next grade from user
gradeString = JOptionPane.showInputDialog(
    "Enter Integer Grade or -1 to Quit: ");

// convert gradeString to int
grade = Integer.parseInt( gradeString );

} // end while

// termination phase
DecimalFormat twoDigits = new DecimalFormat( "0.00" );

// if user entered at least one grade...
if ( gradeCounter != 0 ) {

    // calculate average of all grades entered
    average = (double) total / gradeCounter;

    // display average with two digits of precision
    JOptionPane.showMessageDialog( null,
        "Class average is "+ twoDigits.format( average ),
        "Class Average", JOptionPane.INFORMATION_MESSAGE );

} // end if part of if...else

loop until gradeCounter equals sentinel value (-1)
Format numbers to nearest hundredth
else // if no grades entered, output appropriate message
        JOptionPane.showMessageDialog( null, "No grades were entered", "Class Average", JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 ); // terminate application

    } // end main

} // end class Average2

Average2.java
// Fig. 4.11: Analysis.java
// Analysis of examination results.
import javax.swing.JOptionPane;

public class Analysis {

    public static void main( String args[] ) {
        // initializing variables in declarations
        int passes = 0; // number of passes
        int failures = 0; // number of failures
        int studentCounter = 1; // student counter
        int result; // one exam result

        String input; // user-entered value
        String output; // output string

        // process 10 students using counter - controlled loop
        while ( studentCounter <= 10 ) { // Loop until student counter is greater than 10
            // prompt user for input and obtain value from user
            input = JOptionPane.showInputDialog("Enter result (1 = pass, 2 = fail)" );

            // convert result to int
            result = Integer.parseInt( input );

            // if result 1, increment passes; if...else nested in while
            if ( result == 1 )
                passes = passes + 1;

            // Nested control structure (Nested Control Structures)
        }
    }
}
else // if result not 1, increment failures
    failures = failures + 1;

    // increment studentCounter so loop eventually terminates
    studentCounter = studentCounter + 1;

} // end while

// termination phase; prepare and display results
output = "Passed: " + passes + "\nFailed: " + failures;

// determine whether more than 8 students passed
if ( passes > 8 )
    output = output + "\nRaise Tuition";

JOptionPane.showMessageDialog( null, output, "Analysis of Examination Results", JOptionPane.INFORMATION_MESSAGE);

System.exit( 0 ); // terminate application

} // end main

} // end class Analysis
Compound Assignment Operators

• Assignment Operators
  – Abbreviate assignment expressions
  – Any statement of form
    • \texttt{variable = variable operator expression;}
  – Can be written as
    • \texttt{variable operator= expression;}
  – e.g., addition assignment operator +=
    • \texttt{c = c + 3}
  – can be written as
    • \texttt{c += 3}
<table>
<thead>
<tr>
<th>Assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>c += 7</code></td>
<td><code>c = c + 7</code></td>
<td>10 to <code>c</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>d -= 4</code></td>
<td><code>d = d - 4</code></td>
<td>1 to <code>d</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>e *= 5</code></td>
<td><code>e = e * 5</code></td>
<td>20 to <code>e</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>f /= 3</code></td>
<td><code>f = f / 3</code></td>
<td>2 to <code>f</code></td>
</tr>
<tr>
<td><code>%=</code></td>
<td><code>g %= 9</code></td>
<td><code>g = g % 9</code></td>
<td>3 to <code>g</code></td>
</tr>
</tbody>
</table>

Arithmetic assignment operators.
Increment and Decrement Operators

- Unary increment operator (++)
- Unary decrement operator (--)
- Preincrement / predecrement operator
- Post-increment / post-decrement operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Called</th>
<th>Sample expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>preincrement</td>
<td>++a</td>
<td>Increment a by 1, then use the new value of a in the expression in which a resides.</td>
</tr>
<tr>
<td>++</td>
<td>postincrement</td>
<td>a++</td>
<td>Use the current value of a in the expression in which a resides, then increment a by 1.</td>
</tr>
<tr>
<td>--</td>
<td>predecrement</td>
<td>--b</td>
<td>Decrement b by 1, then use the new value of b in the expression in which b resides.</td>
</tr>
<tr>
<td>--</td>
<td>postdecrement</td>
<td>b--</td>
<td>Use the current value of b in the expression in which b resides, then decrement b by 1.</td>
</tr>
</tbody>
</table>

The increment and decrement operators.
// Fig. 4.14: Increment.java
// Preincrementing and postincrementing operators.

public class Increment {

    public static void main( String args[] )
    {
        int c;

        c = 5;                // assign 5 to c
        System.out.println( c ); // print 5
        System.out.println( c++ ); // print 5 then postincrement
        System.out.println( c );   // print 6

        System.out.println();    // skip a line

        c = 5;                // assign 5 to c
        System.out.println( c ); // print 5
        System.out.println( ++c ); // preincrement then print 6
        System.out.println( c ); // print 6

    } // end main
} // end class Increment
Primitive Types

• Primitive types
  – “building blocks” for more complicated types

• Java is strongly typed
  – All variables in a Java program must have a type

• Java primitive types
  – portable across computer platforms that support Java
<table>
<thead>
<tr>
<th>Type</th>
<th>Size in bits</th>
<th>Values</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td></td>
<td>true or false</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Note: The representation of a boolean is specific to the Java Virtual Machine on each computer platform.]</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>'u0000' to 'uFFFF'</td>
<td>(ISO Unicode character set)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0 to 65535)</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>–128 to +127</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(–27 to 27 – 1)</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>–32,768 to +32,767</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(–215 to 215 – 1)</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>–2,147,483,648 to +2,147,483,647</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(–231 to 231 – 1)</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(–263 to 263 – 1)</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>Negative range:</td>
<td>(IEEE 754 floating point)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–3.4028234663852886E+38 to –1.40129846432481707e–45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Positive range:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.40129846432481707e–45 to 3.4028234663852886E+38</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>Negative range:</td>
<td>(IEEE 754 floating point)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–1.7976931348623157E+308 to –4.94065645841246544e–324</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Positive range:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.94065645841246544e–324 to 1.7976931348623157E+308</td>
<td></td>
</tr>
</tbody>
</table>
Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires:
  - Control variable (loop counter)
  - Initial value of the control variable
  - Increment/decrement of control variable through each loop
  - Condition that tests for the final value of the control variable

```plaintext
for ( int counter = 1; counter <= 10; counter++ )
```

- **for** keyword
- **Control variable**: `counter`
- **Required separator**: `;`
- **Final value of control variable for which the condition is true**: `10`
- **Increment of control variable**: `counter++`
for Repetition Structure (cont.)

for ( initialization; 
     loopContinuationCondition; increment )
     statement;

can usually be rewritten as:

initialization;
while ( loopContinuationCondition ){
     statement;
     increment;
}
Examples Using the \texttt{for} Statement

• Varying control variable in \texttt{for} statement
  – Vary control variable from 1 to 100 in increments of 1
    • \texttt{for ( int i = 1; i <= 100; i++ )}
  – Vary control variable from 100 to 1 in increments of \( -1 \)
    • \texttt{for ( int i = 100; i >= 1; i-- )}
  – Vary control variable from 7 to 77 in increments of 7
    • \texttt{for ( int i = 7; i <= 77; i += 7 )}
// Fig. 5.5: Sum.java
// Summing integers with the for statement.
import javax.swing.JOptionPane;

public class Sum {

    public static void main( String args[] )
    {
        int total = 0; // initialize sum

        // total even integers from 2 through 100
        for ( int number = 2; number <= 100; number += 2 )
            total += number;

        // display results
        JOptionPane.showMessageDialog( null, "The sum is " + total,
                                   "Total Even Integers from 2 to 100",
                                   JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 ); // terminate application
    }

} // end class Sum
**do...while Repetition Statement**

- **do...while structure**
  - Similar to `while` structure
  - Tests loop-continuation after performing body of loop
    - i.e., loop body always executes at least once

![Diagram of do...while statement]

- **Diagram**
  - Action state
  - Condition: [true] [false]
  - Start state
**switch** Multiple-Selection Statement

- **switch** statement used for multiple selections

```
case a 
    [true] case a action(s) -> break
    [false] case b action(s) -> break

        . .

        case z 
    [true] case z action(s) -> break
    [false] default action(s) ->

...```
break and continue Statements

• break/continue
  – Alter flow of control

• break statement
  – Causes immediate exit from control structure
    • Used in while, for, do...while or switch statements

• continue statement
  – Skips remaining statements in loop body
  – Proceeds to next iteration
    • Used in while, for or do...while statements
// Fig. 5.11: BreakTest.java
// Terminating a loop with break.
import javax.swing.JOptionPane;

public class BreakTest {

    public static void main( String args[] )
    {
        String output = "";
        int count;

        for ( count = 1; count <= 10; count++ ) { // loop 10 times
            if ( count == 5 ) // if count is 5,
                break; // terminate loop

            output += count + " ";
        } // end for

        output += "\nBroke out of loop at count = " + count;
        JOptionPane.showMessageDialog( null, output );

        System.exit( 0 ); // terminate application
    } // end main

} // end class BreakTest

exit for structure (break) when count equals 5
Loop 10 times
// Fig. 5.12: ContinueTest.java
// Continuing with the next iteration of a loop.
import javax.swing.JOptionPane;

public class ContinueTest {
    public static void main( String args[] )
    {
        String output = "";
        for ( int count = 1; count <= 10; count++ ) {
            // loop 10 times
            if ( count == 5 ) // if count is 5,
                continue; // skip remaining code in loop
            output += count + " ";
        } // end for
        output += "\nUsed continue to skip printing 5";
        JOptionPane.showMessageDialog( null, output );
        System.exit( 0 ); // terminate application
    } // end main
} // end class ContinueTest
Labeled break and continue Statements

- **Labeled block**
  - Set of statements enclosed by `{}`
  - Preceded by a label

- **Labeled `break` statement**
  - Exit from nested control structures
  - Proceeds to end of specified labeled block

- **Labeled `continue` statement**
  - Skips remaining statements in nested-loop body
  - Proceeds to beginning of specified labeled block
// Fig. 5.13: BreakLabelTest.java
// Labeled break statement.
import javax.swing.JOptionPane;

public class BreakLabelTest {
    public static void main(String args[])
    {
        String output = "";
        stop: {
            // labeled block
            // count 10 rows
            for (int row = 1; row <= 10; row++) {
                // count 5 columns
                for (int column = 1; column <= 5; column++) {
                    if (row == 5) // if row is 5,
                        break stop; // jump to end of stop block
                    output += "* ";
                } // end inner for
                output += "\n";
            } // end outer for
        } // end stop block
    }
}
output += "nLoops terminated normally";

} // end labeled block

JOptionPane.showMessageDialog( null, output, "Testing break with a label", JOptionPane.INFORMATION_MESSAGE );

System.exit( 0 ); // terminate application

} // end main

} // end class BreakLabelTest
// Fig. 5.14: ContinueLabelTest.java
// Labeled continue statement.
import javax.swing.JOptionPane;

public class ContinueLabelTest {
    public static void main( String args[] ) {
        String output = "";
        nextRow: // target label of continue statement
            // count 5 rows
            for ( int row = 1; row <= 5; row++ ) {
                output += "\n";

                // count 10 columns per row
                for ( int column = 1; column <= 10; column++ ) {
                    // if column greater than row, start next row
                    if ( column > row )
                        continue nextRow; // next iteration of labeled loop
                    output += "*  ";
                } // end inner for
            } // end outer for
    }
}
JOptionPane.showMessageDialog( null, output,
    "Testing continue with a label",
    JOptionPane.INFORMATION_MESSAGE);

System.exit( 0 );  // terminate application

} // end main

} // end class ContinueLabelTest
**Precedence Rules**

*Highest Precedence*

First: the unary operators: +, -, ++, --, and!

Second: the binary arithmetic operators: *, /, and %

Third: the binary arithmetic operators: + and -

*Lowest Precedence*
**Precedence and Associativity Rules**

- When the order of two adjacent operations must be determined, the operation of higher precedence (and its apparent arguments) is grouped before the operation of lower precedence

  \[ \text{base} + \text{rate} \times \text{hours} \]  
  is evaluated as

  \[ \text{base} + (\text{rate} \times \text{hours}) \]

- When two operations have equal precedence, the order of operations is determined by *associativity* rules
Precedence and Associativity Rules

- Unary operators of equal precedence are grouped right-to-left
  \[ ++rate \] is evaluated as \[ +(-(+rate)) \]
- Binary operators of equal precedence are grouped left-to-right
  \[ \text{base} + \text{rate} + \text{hours} \] is evaluated as
  \[ (\text{base} + \text{rate}) + \text{hours} \]
- Exception: A string of assignment operators is grouped right-to-left
  \[ n1 = n2 = n3; \] is evaluated as \[ n1 = (n2 = n3); \]
<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>right to left</td>
<td>unary postfix</td>
</tr>
<tr>
<td>--</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>++</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>-</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>!=</td>
<td>left to right</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
<td>boolean logical AND</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
<td>boolean logical exclusive OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
<td>conditional AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td>=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td>+=</td>
<td>right to left</td>
<td></td>
</tr>
<tr>
<td>-=</td>
<td>right to left</td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td>right to left</td>
<td></td>
</tr>
<tr>
<td>/=</td>
<td>right to left</td>
<td></td>
</tr>
<tr>
<td>%=</td>
<td>right to left</td>
<td></td>
</tr>
</tbody>
</table>

Precedence/associativity of the operators discussed so far.
Logical Operators

• Logical operators
  – Allows for forming more complex conditions
  – Combines simple conditions

• Java logical operators
  – && (conditional AND)
  – & (boolean logical AND)
  – || (conditional OR)
  – | (boolean logical inclusive OR)
  – ^ (boolean logical exclusive OR)
  – ! (logical NOT)
<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 &amp;&amp; expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

&& (conditional AND) operator truth table.

| expression1 | expression2 | expression1 || expression2 |
|------------|------------|-------------------|
| false      | false      | false              |
| false      | true       | true               |
| true       | false      | true               |
| true       | true       | true               |

|| (conditional OR) operator truth table.
<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 $\land$ expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

$\land$ (boolean logical exclusive OR) operator truth table.

<table>
<thead>
<tr>
<th>expression</th>
<th>$\neg$expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

$\neg$ (logical negation, or logical NOT) operator truth table.
// Fig. 5.19: LogicalOperators.java
// Logical operators.
import javax.swing.*;

public class LogicalOperators

    public static void main( String args[] )
    {
        // create JTextArea to display results
        JTextArea outputArea = new JTextArea( 17, 20 );

        // attach JTextArea to a JScrollPane so user can scroll results
        JScrollPane scroller = new JScrollPane( outputArea );

        // create truth table for && (conditional AND) operator
        String output = "Logical AND (&&)" +
            "false && false: " + ( false && false ) +
            "false && true: " + ( false && true ) +
            "true && false: " + ( true && false ) +
            "true && true: " + ( true && true );

        // create truth table for || (conditional OR) operator
        output += "Logical OR (||)" +
            "false || false: " + ( false || false ) +
            "false || true: " + ( false || true ) +
            "true || false: " + ( true || false ) +
            "true || true: " + ( true || true );

        // display results
        outputArea.setText( output );
        scroller.setViewportView( outputArea );
        JOptionPane.showMessageDialog( null, scroller );
    }

// create truth table for & (boolean logical AND) operator
output += "Boolean logical AND (&)" +
  "false & false: " + (false & false) +
  "false & true: " + (false & true) +
  "true & false: " + (true & false) +
  "true & true: " + (true & true);

// create truth table for | (boolean logical inclusive OR) operator
output += "Boolean logical inclusive OR (|)" +
  "false | false: " + (false | false) +
  "false | true: " + (false | true) +
  "true | false: " + (true | false) +
  "true | true: " + (true | true);

// create truth table for ^ (boolean logical exclusive OR) operator
output += "Boolean logical exclusive OR (^)" +
  "false ^ false: " + (false ^ false) +
  "false ^ true: " + (false ^ true) +
  "true ^ false: " + (true ^ false) +
  "true ^ true: " + (true ^ true);

// create truth table for ! (logical negation) operator
output += "Logical NOT (!)" +
  "false: " + (!false) +
  "true: " + (!true);

outputArea.setText(output); // place results in JTextArea
JOptionPane.showMessageDialog( null, scroller,
   "Truth Tables", JOptionPane.INFORMATION_MESSAGE );

System.exit( 0 ); // terminate application

} // end main

} // end class LogicalOperators