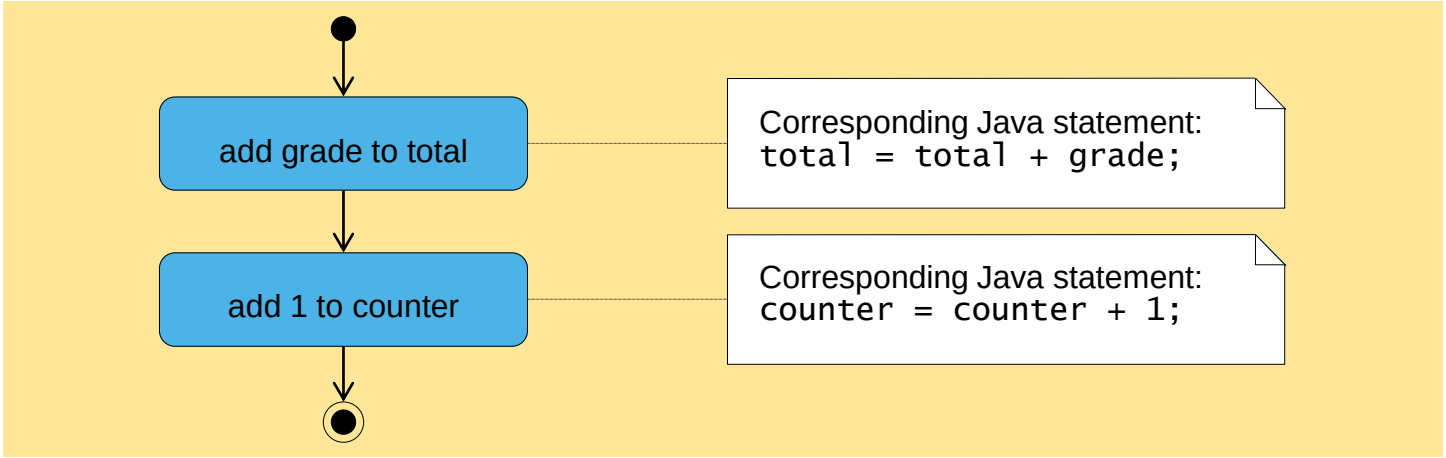
Chapter 2 - Control Structures

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



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

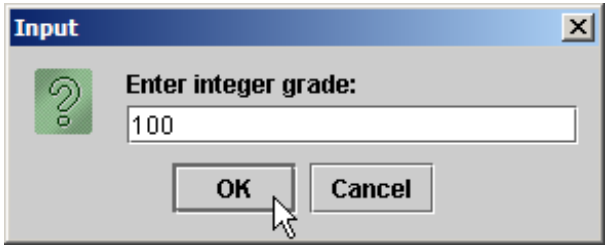
```
1  // Fig. 4.7: Average1.java
2  // Class-average program with counter-controlled repetition.
3  import javax.swing.JOptionPane;
4
5  public class Average1 {
6
7     public static void main( String args[] )
8     {
9        int total;        // sum of grades input by user
10       int gradeCounter; // number of grade to be entered next
11       int grade;        // grade value
12       int average;      // average of gra
13
14       String gradeString; // grade typed by
15
16       // initialization phase
17       total = 0;        // initialize total
18       gradeCounter = 1; // initialize loop counter
19
20       // processing phase
21       while ( gradeCounter <= 10 ) {  // loop 10 times
22
23          // prompt for input and read grade from user
24          gradeString = JOptionPane.showInputDialog(
25             "Enter integer grade: " );
26
27          // convert gradeString to int
28          grade = Integer.parseInt( gradeString );
29
```

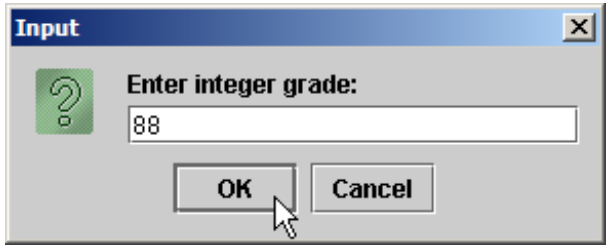
Declare variables; `gradeCounter` is the counter

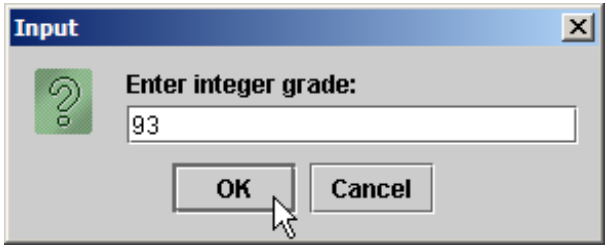
Continue looping as long as `gradeCounter` is less than or equal to 10

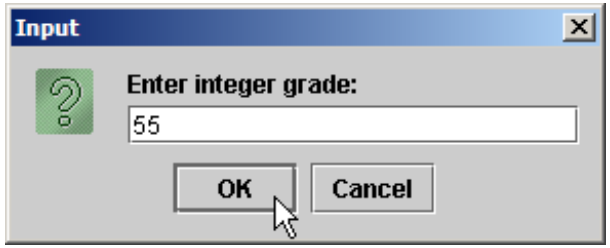
```
30          total = total + grade;           // add grade to total
31          gradeCounter = gradeCounter + 1;  // increment counter
32
33       } // end while
34
35       // termination phase
36       average = total / 10;  // integer division
37
38       // display average of exam grades
39       JOptionPane.showMessageDialog( null, "Class average is " + average,
40          "Class Average", JOptionPane.INFORMATION_MESSAGE );
41
42       System.exit( 0 );  // terminate the program
43
44    } // end main
45
46 } // end class Average1
```

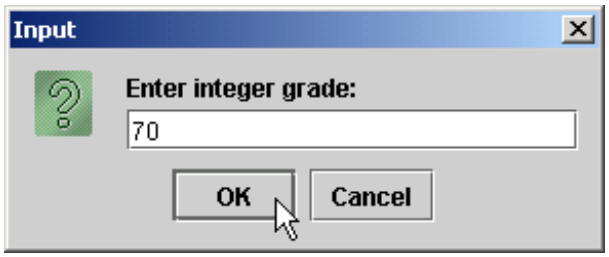
Average1.java

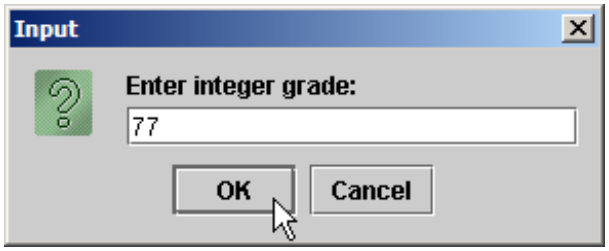




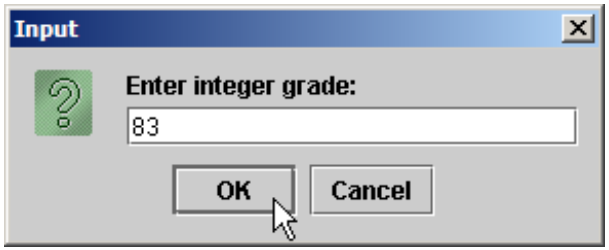


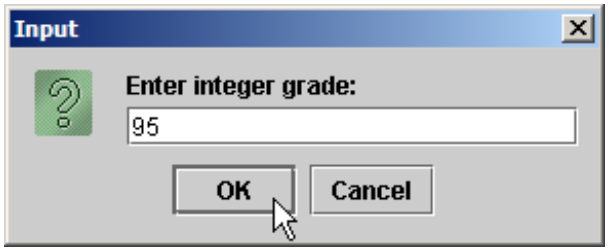


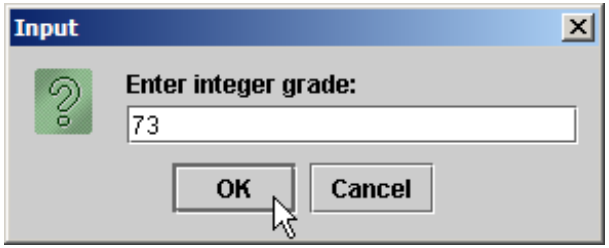


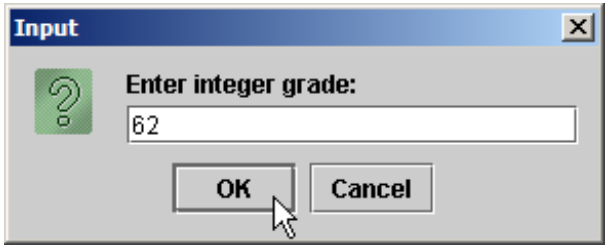


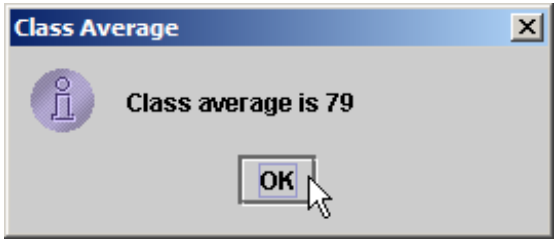
Average1.java











(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

```
1     // Fig. 4.9: Average2.java
2     // Class-average program with sentinel-controlled repetition.
3     import java.text.DecimalFormat;  // class to format numbers
4     import javax.swing.JOptionPane;
5
6     public class Average2 {
7
8        public static void main( String args[] )
9        {
10          int total;           // sum of grades
11          int gradeCounter;    // number of grades entered
12          int grade;           // grade value
13
14          double average;  // number with decimal point for average
15
16          String gradeString;  // grade typed by user
17
18          // initialization phase
19          total = 0;         // initialize total
20          gradeCounter = 0;  // initialize loop counter
21
22          // processing phase
23          // get first grade from user
24          gradeString = JOptionPane.showInputDialog(
25             "Enter Integer Grade or -1 to Quit:" );
26
27          // convert gradeString to int
28          grade = Integer.parseInt( gradeString );
29
```

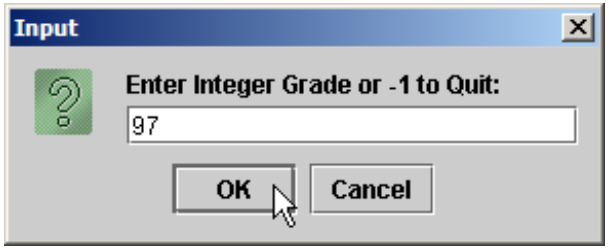
```
30          // loop until sentinel value read from user
31          while ( grade != -1 ) {
32             total = total + grade;
33             gradeCounter = gradeCounter + 1;
34
35             // get next grade from user
36             gradeString = JOptionPane.showInputDialog(
37                "Enter Integer Grade or -1 to Quit:" );
38
39             // convert gradeString to int
40             grade = Integer.parseInt( gra
41
42          } // end while
43
44          // termination phase
45          DecimalFormat twoDigits = new DecimalFormat( "0.00" );
46
47          // if user entered at least one grade...
48          if ( gradeCounter != 0 ) {
49
50             // calculate average of all grades entered
51             average = (double) total / gradeCounter;
52
53             // display average with two digits of precision
54             JOptionPane.showMessageDialog( null,
55                "Class average is " + twoDigits.format( average ),
56                "Class Average", JOptionPane.INFORMATION_MESSAGE );
57
58          } // end if part of if...else
59
```

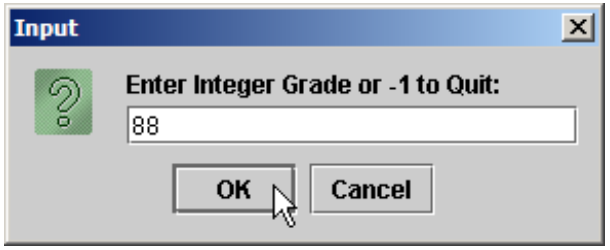
loop until `gradeCounter` equals sentinel value (`-1`)

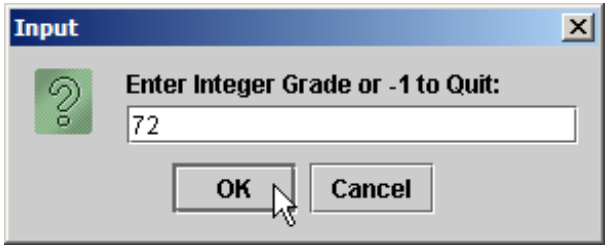
Format numbers to nearest hundredth

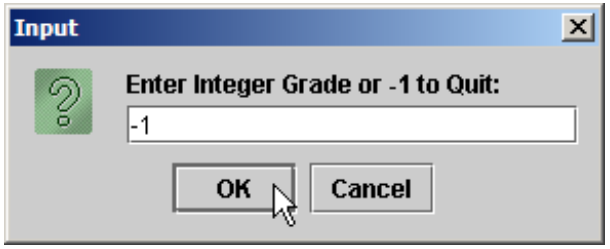
```
60          else // if no grades entered, output appropriate message
61             JOptionPane.showMessageDialog( null, "No grades were entered",
62                "Class Average", JOptionPane.INFORMATION_MESSAGE );
63
64          System.exit( 0 );  // terminate application
65
66       } // end main
67
68    } // end class Average2
```

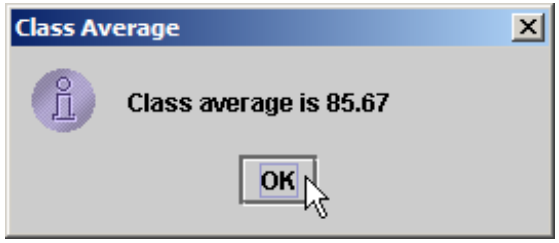
Average2.java











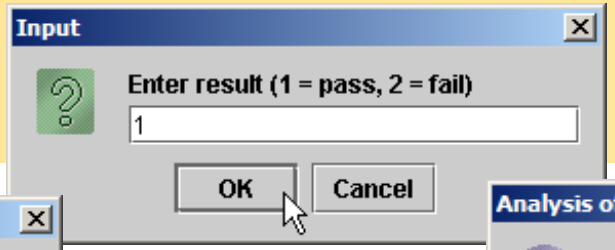
(Nested Control Structures)

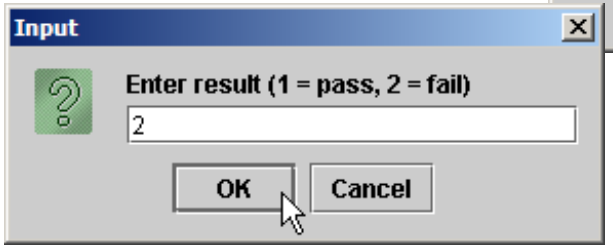

```
1   // Fig. 4.11: Analysis.java
2   // Analysis of examination results.
3   import javax.swing.JOptionPane;
4
5   public class Analysis {
6
7      public static void main( String args[] )
8      {
9        // initializing variables in declarations
10       int passes = 0;          // number of passes
11       int failures = 0;        // number of failures
12       int studentCounter = 1;  // student counter
13       int result;              // one exam result
14
15       String input;            // user-entered value
16       String output;           // output string
17
18       // process 10 students using counte
19       while ( studentCounter <= 10 ) {
20
21          // prompt user for input and obtain value from user
22          input = JOptionPane.showInputDialog(
23             "Enter result (1 = pass, 2 = fail)" );
24
25          // convert result to int
26          result = Integer.parseInt( input );
27
28          // if result 1, increment passes; if...else nested in while
29          if ( result == 1 )
30             passes = passes + 1;
```

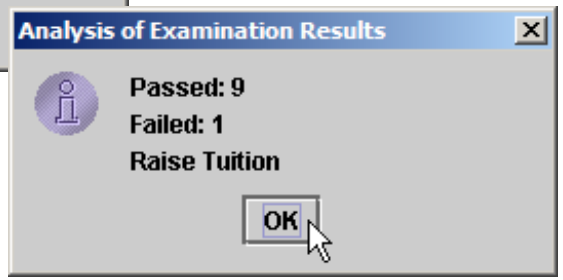
Loop until `student` counter is greater than 10

Nested control structure

```
31
32             else // if result not 1, increment failures
33                failures = failures + 1;
34
35             // increment studentCounter so loop eventually terminates
36             studentCounter = studentCounter + 1;
37
38          } // end while
39
40          // termination phase; prepare and display results
41          output = "Passed: " + passes + "\nFailed: " + failures;
42
43          // determine whether more than 8 students passed
44          if ( passes > 8 )
45             output = output + "\nRaise Tuition";
46
47          JOptionPane.showMessageDialog( null, output,
48             "Analysis of Examination Results",
49             JOptionPane.INFORMATION_MESSAGE );
50
51          System.exit( 0 );  // terminate application
52
53       } // end main
54
55    } // end class Analysis
```







Compound Assignment Operators

- Assignment Operators
  - Abbreviate assignment expressions
  - Any statement of form
    - *variable* = *variable operator expression*;
  - Can be written as
    - *variable operator*= *expression*;
  - e.g., addition assignment operator +=
    - `c = c + 3`
  - can be written as
    - `c += 3`

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to `c`
`-=`	`d -= 4`	`d = d - 4`	1 to `d`
`*=`	`e *= 5`	`e = e * 5`	20 to `e`
`/=`	`f /= 3`	`f = f / 3`	2 to `f`
`%=`	`g %= 9`	`g = g % 9`	3 to `g`

Arithmetic assignment operators.

Increment and Decrement Operators

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

Operator	Called	Sample expression	Explanation
`++`	preincrement	`++a`	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
`++`	postincrement	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
`--`	predecrement	`--b`	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
`--`	postdecrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

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
```

```
5
5
6

5
6
6
```

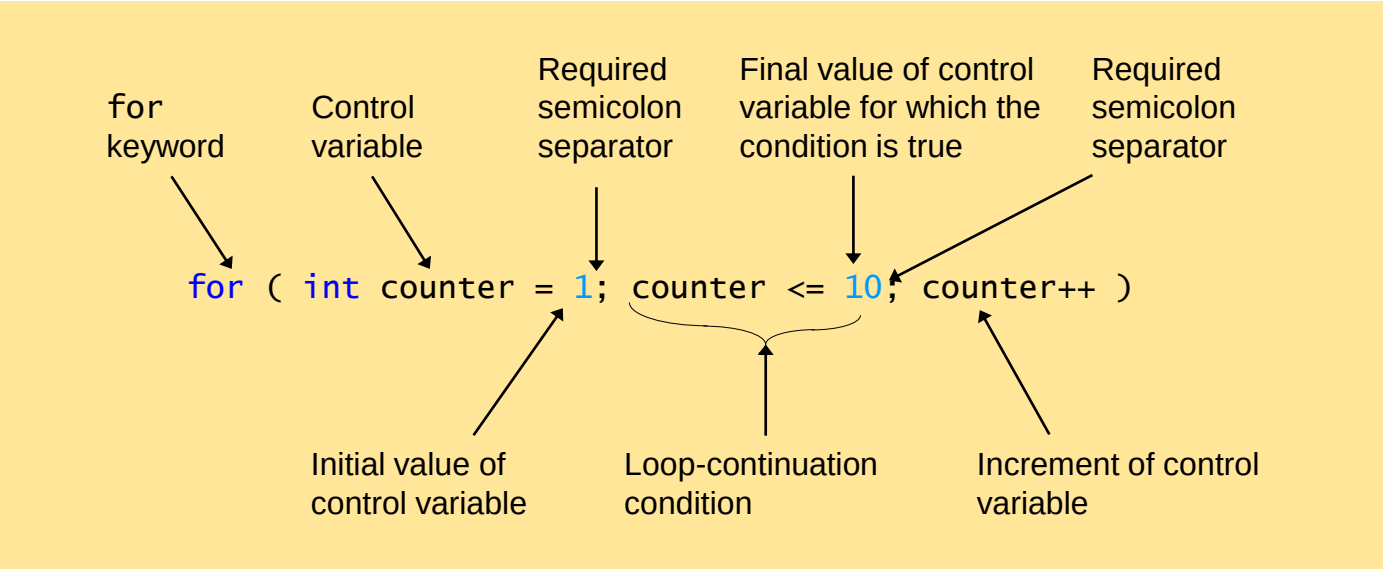
Primitive Types

- 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

Type	Size in bits	Values	Standard
`boolean`		`true` or `false`	
		[*Note:* The representation of a boolean is specific to the Java Virtual Machine on each computer platform.]	
`char`	16	`'\u0000'` to `'\uFFFF'` (0 to 65535)	(ISO Unicode character set)
`byte`	8	–128 to +127 (-2^7 to $2^7 - 1$)	
`short`	16	–32,768 to +32,767 (-2^{15} to $2^{15} - 1$)	
`int`	32	–2,147,483,648 to +2,147,483,647 (-2^{31} to $2^{31} - 1$)	
`long`	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (-2^{63} to $2^{63} - 1$)	
`float`	32	*Negative range:* –3.4028234663852886E+38 to –1.40129846432481707e–45 *Positive range:* 1.40129846432481707e–45 to 3.4028234663852886E+38	(IEEE 754 floating point)
`double`	64	*Negative range:* –1.7976931348623157E+308 to –4.94065645841246544e–324 *Positive range:* 4.94065645841246544e–324 to 1.7976931348623157E+308	(IEEE 754 floating point)

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



`for` Repetition Structure (cont.)

```
for ( initialization;
loopContinuationCondition; increment )
    statement;
```

can usually be rewritten as:

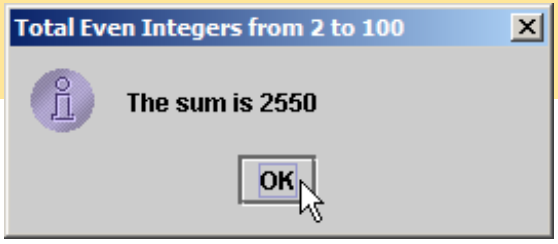
```
initialization;
while ( loopContinuationCondition ) {
    statement;
    increment;
}
```

Examples Using the `for` Statement

- Varying control variable in `for` statement
  - Vary control variable from 1 to 100 in increments of 1
    - `for ( int i = 1; i <= 100; i++ )`
  - Vary control variable from 100 to 1 in increments of –1
    - `for ( int i = 100; i >= 1; i-- )`
  - Vary control variable from 7 to 77 in increments of 7
    - `for ( int i = 7; i <= 77; i += 7 )`

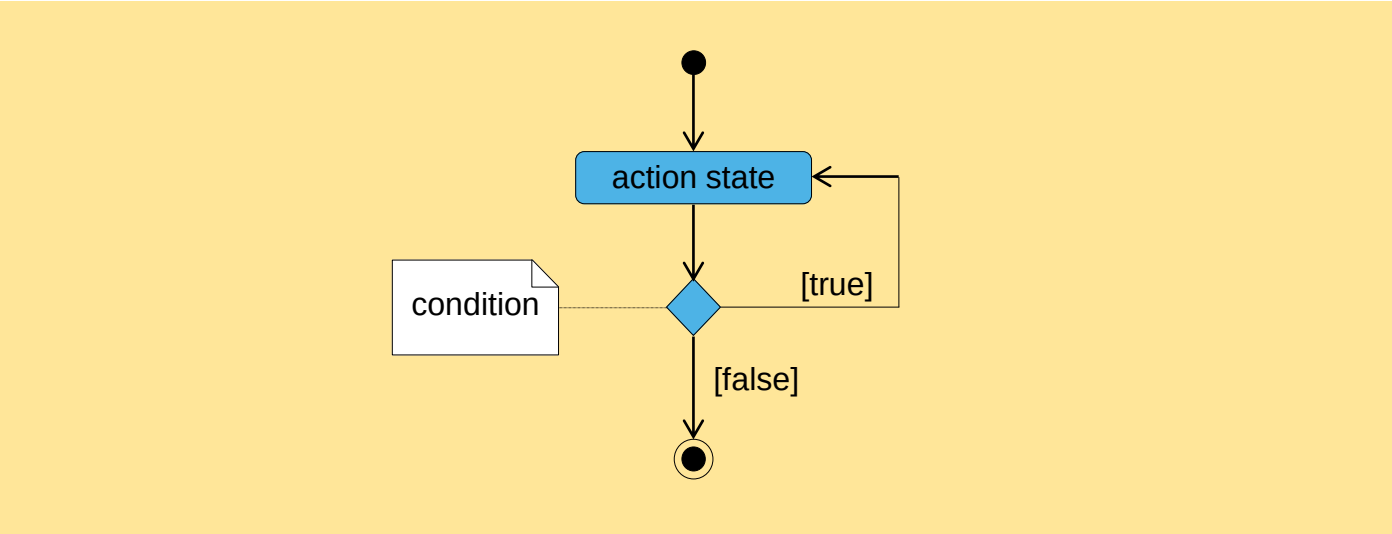
```
1   // Fig. 5.5: Sum.java
2   // Summing integers with the for statement.
3   import javax.swing.JOptionPane;
4
5   public class Sum {
6
7      public static void main( String args[] )
8      {
9         int total = 0;  // initialize sum
10
11       // total even integers from 2 through 100
12       for ( int number = 2; number <= 100; number += 2 )
13          total += number;
14
15       // display results
16       JOptionPane.showMessageDialog( null, "The sum is " + total,
17          "Total Even Integers from 2 to 100",
18          JOptionPane.INFORMATION_MESSAGE );
19
20       System.exit( 0 );   // terminate application
21
22    } // end main
23
24 } // end class Sum
```

increment `number` by 2 each iteration



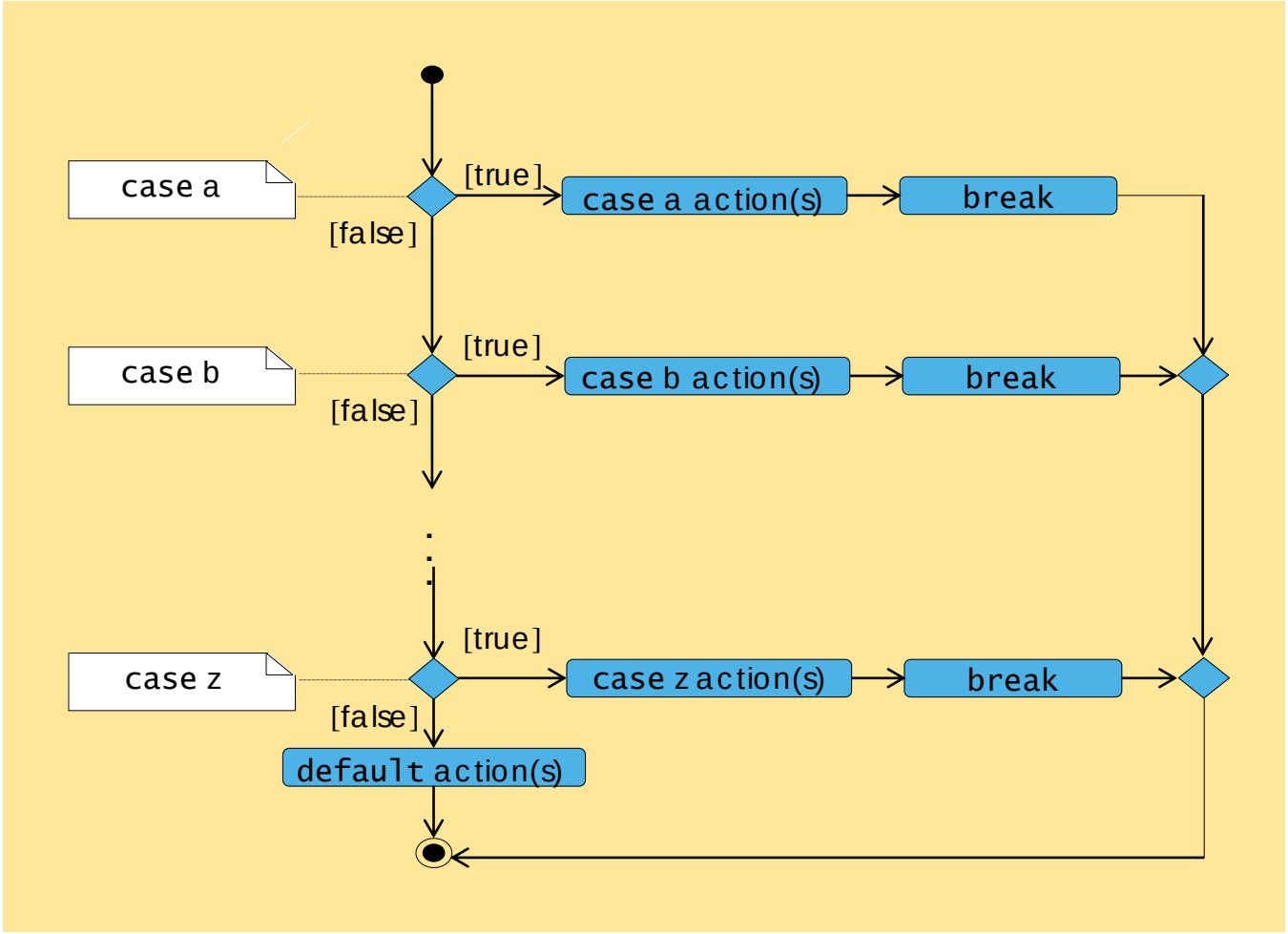
do…while Repetition Statement

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



`switch` Multiple-Selection Statement

- `switch` statement used for multiple selections



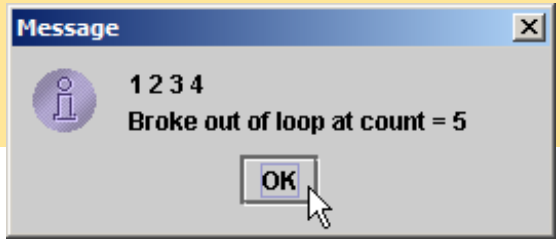
`break` and `continue` Statements

- `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

```
1     // Fig. 5.11: BreakTest.java
2     // Terminating a loop with break.
3     import javax.swing.JOptionPane;
4
5     public class BreakTest {
6
7        public static void main( String args[] )
8        {
9           String output = "";
10          int count;
11
12          for ( count = 1; count <= 10; count++ ) {  // loop 10 times
13
14             if ( count == 5 )  // if count is 5,
15                break;          // terminate loop
16
17             output += count + " ";
18
19          } // end for
20
21          output += "\nBroke out of loop at count = " + count;
22          JOptionPane.showMessageDialog( null, output );
23
24          System.exit( 0 );  // terminate application
25
26       } // end main
27
28    } // end class BreakTest
```

exit **for** structure (**break**) when **count** equals 5

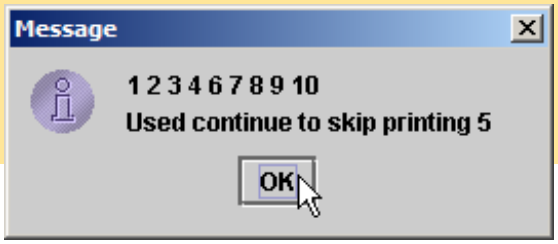
Loop 10 times



```
1   // Fig. 5.12: ContinueTest.java
2   // Continuing with the next iteration of a loop.
3   import javax.swing.JOptionPane;
4
5   public class ContinueTest {
6
7      public static void main( String args[] )
8      {
9         String output = "";
10
11        for ( int count = 1; count <= 10; count++ ) {  // loop 10 times
12
13           if ( count == 5 ) // if count is 5,
14              continue;       // skip remaining code in loop
15
16           output += count + " ";
17
18        } // end for
19
20        output += "\nUsed continue to skip printing 5";
21        JOptionPane.showMessageDialog( null, output );
22
23        System.exit( 0 );  // terminate application
24
25     } // end main
26
27  } // end class ContinueTest
```

Skip line 16 and proceed to line 11 when count equals 5

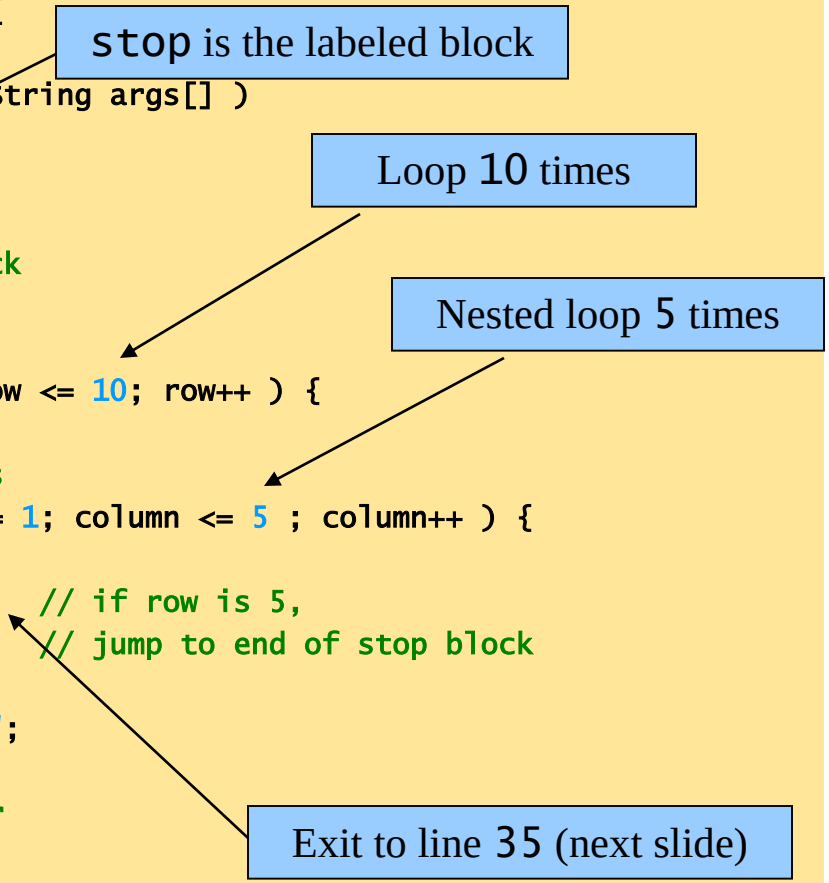
Loop 10 times



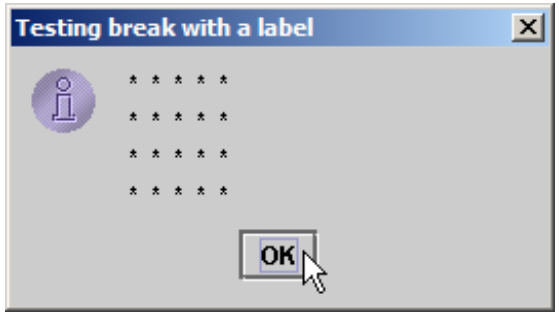
Labeled `break` and `continue` Statements

- 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

```
1  // Fig. 5.13: BreakLabelTest.java
2  // Labeled break statement.
3  import javax.swing.JOptionPane;
4
5  public class BreakLabelTest {
6
7     public static void main( String args[] )
8     {
9        String output = "";
10
11       stop: {  // labeled block
12
13          // count 10 rows
14          for ( int row = 1; row <= 10; row++ ) {
15
16             // count 5 columns
17             for ( int column = 1; column <= 5 ; column++ ) {
18
19                if ( row == 5 )  // if row is 5,
20                   break stop;   // jump to end of stop block
21
22                output += "*  ";
23
24             } // end inner for
25
26             output += "\n";
27
28          } // end outer for
29
```



```
30             // following line is skipped
31             output += "\nLoops terminated normally";
32
33          } // end labeled block
34
35          JOptionPane.showMessageDialog( null, output,
36             "Testing break with a label",
37             JOptionPane.INFORMATION_MESSAGE );
38
39          System.exit( 0 );  // terminate application
40
41       } // end main
42
43    } // end class BreakLabelTest
```



```
1  // Fig. 5.14: ContinueLabelTest.java
2  // Labeled continue statement.
3  import javax.swing.JOptionPane;
4
5  public class ContinueLabelTest {
6
7     public static void main( String args[] )
8     {
9        String output = "";
10
11       nextRow:  // target label of continue statement
12
13          // count 5 rows
14          for ( int row = 1; row <= 5; row++ ) {
15             output += "\n";
16
17             // count 10 columns per row
18             for ( int column = 1; column <= 10; column++ ) {
19
20                // if column greater than row, start next row
21                if ( column > row )
22                   continue nextRow; // next iteration of labeled loop
23
24                output += "*  ";
25
26             } // end inner for
27
28          } // end outer for
```

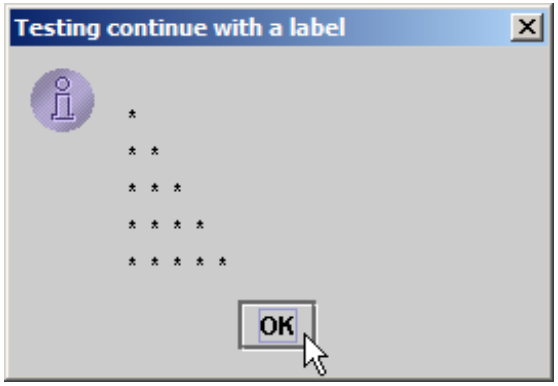
nextRow is the labeled block

Loop 5 times

Nested loop 10 times

continue to line 11 (nextRow)

```
29
30          JOptionPane.showMessageDialog( null, output,
31             "Testing continue with a label",
32             JOptionPane.INFORMATION_MESSAGE );
33
34          System.exit( 0 );  // terminate application
35
36       } // end main
37
38    } // end class ContinueLabelTest
```



Precedence Rules

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

  base + rate * hours is evaluated as

  base + (rate * 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

  `base + rate + hours` is evaluated as

  `(base + rate) + hours`

- Exception: A string of assignment operators is grouped right-to-left

  `n1 = n2 = n3;` is evaluated as `n1 = (n2 = n3);`

Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - !	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
|	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
||	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

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)

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

&& (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.

expression1	expression2	expression1 ^ expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`false`

^ (boolean logical exclusive OR) operator truth table.

expression	!expression
`false`	`true`
`true`	`false`

! (logical negation, or logical NOT) operator truth table.

```
1     // Fig. 5.19: LogicalOperators.java
2     // Logical operators.
3     import javax.swing.*;
4
5     public class LogicalOperators
6
7        public static void main( String args[] )
8        {
9           // create JTextArea to display results
10          JTextArea outputArea = new JTextArea( 17, 20 );
11
12          // attach JTextArea to a JScrollPane so user can scroll results
13          JScrollPane scroller = new JScrollPane( output
14
15          // create truth table for && (conditional AND) operator
16          String output = "Logical AND (&&)" +
17             "\nfalse && false: " + ( false && false ) +
18             "\nfalse && true: " + ( false && true ) +
19             "\ntrue && false: " + ( true && false ) +
20             "\ntrue && true: " + ( true && true );
21
22          // create truth table for || (conditional OR) operator
23          output += "\n\nLogical OR (||)" +
24             "\nfalse || false: " + ( false || false ) +
25             "\nfalse || true: " + ( false || true ) +
26             "\ntrue || false: " + ( true || false ) +
27             "\ntrue || true: " + ( true || true );
28
```

Conditional AND truth table

Conditional OR truth table

```
29          // create truth table for & (boolean logical AND) operator
30          output += "\n\nBoolean logical AND (&)" +
31             "\nfalse & false: " + ( false & false ) +
32             "\nfalse & true: " + ( false & true ) +
33             "\ntrue & false: " + ( true & false ) +
34             "\ntrue & true: " + ( true & true );
35
36          // create truth table for | (boolean logical inclusive OR) operator
37          output += "\n\nBoolean logical inclusive OR (|)" +
38             "\nfalse | false: " + ( false | false ) +
39             "\nfalse | true: " + ( false | true ) +
40             "\ntrue | false: " + ( true | false ) +
41             "\ntrue | true: " + ( true | true );
42
43          // create truth table for ^ (boolean logical exclusive OR) operator
44          output += "\n\nBoolean logical exclusive OR (^)" +
45             "\nfalse ^ false: " + ( false ^ false ) +
46             "\nfalse ^ true: " + ( false ^ true ) +
47             "\ntrue ^ false: " + ( true ^ false ) +
48             "\ntrue ^ true: " + ( true ^ true );
49
50          // create truth table for ! (logical negation) operator
51          output += "\n\nLogical NOT (!)" +
52             "\n!false: " + ( !false ) +
53             "\n!true: " + ( !true );
54
55          outputArea.setText( output );  // place results in JTextArea
56
```

Boolean logical AND truth table

Boolean logical inclusive OR truth table

Boolean logical exclusive OR truth table

Logical NOT truth table

```
57          JOptionPane.showMessageDialog( null, scroller,
58             "Truth Tables", JOptionPane.INFORMATION_MESSAGE );
59
60          System.exit( 0 );  // terminate application
61
62       } // end main
63
64    } // end class LogicalOperators
```