

# **Chapter 6**

## **Control Statements Continued**

Fundamentals of Java

---

# Objectives

- Construct complex Boolean expressions using the logical operators `&&` (AND), `||` (OR), and `!` (NOT).
- Construct truth tables for Boolean expressions.
- Understand the logic of nested `if` statements and extended `if` statements.

## Objectives (cont.)

- Test `if` statements in a comprehensive manner.
- Construct nested loops.
- Create appropriate test cases for `if` statements and loops.
- Understand the purpose of assertions, invariants, and loop verification.

# Vocabulary

- Arithmetic overflow
- Boundary condition
- Combinatorial explosion
- Complete code coverage
- Equivalence class
- Extended `if` statement

# Vocabulary (cont.)

- Extreme condition
- Input assertion
- Logical operator
- Loop invariant
- Loop variant
- Nested `if` statement

# Vocabulary (cont.)

- Nested loop
- Output assertion
- Quality assurance
- Robust
- Truth table

# Logical Operators

- Used in Boolean expressions that control the behavior of `if`, `while`, and `for` statements
- Three logical operators:
  - **AND**
  - **OR**
  - **NOT**
- Can combine smaller logical expressions into larger ones

# Logical Operators (cont.)

- **Truth tables:** For complex logical expressions, shows how value of overall condition depends on values of operands
  - All combinations of values considered
  - $2^n$  combinations of true and false for  $n$  operands

# Logical Operators (cont.)

THE SUN IS SHINING	IT IS 8 A.M.	THE SUN IS SHINING AND IT IS 8 A.M.	ACTION TAKEN
true	true	true	go for a walk
true	false	false	stay at home
false	true	false	stay at home
false	false	false	stay at home
THE SUN IS SHINING	IT IS 8 A.M.	THE SUN IS SHINING OR IT IS 8 A.M.	ACTION TAKEN
true	true	true	go for a walk
true	false	true	go for a walk
false	true	true	go for a walk
false	false	false	stay at home
THE SUN IS SHINING	NOT THE SUN IS SHINING	ACTION TAKEN	
true	false	stay at home	
false	true	go for a walk	

Table 6-1: Truth tables for three example sentences

# Logical Operators (cont.)

P	Q	P AND Q	P OR Q	NOT P
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Table 6-2: General rules for AND, OR, and NOT

# Logical Operators (cont.)

- Can use parentheses with complex expressions
  - If (the sun is shining AND it is 8 a.m.) OR (NOT your brother is visiting) then let's go for a walk else let's stay at home.
  - If the sun is shining AND (it is 8 a.m. OR (NOT your brother is visiting)) then let's go for a walk else let's stay at home.
- Truth tables would clarify these statements.

# Logical Operators (cont.)

- Java's logical operators:
  - AND: `&&`
  - OR: `||`
  - NOT: `!`
- Precedence rules:
  - `!` has same precedence as other unary operators.
  - `&&` and `||` only have higher precedence than the assignment operators.
    - `&&` has higher precedence than `||`.

# Logical Operators (cont.)

- Example:

```
Scanner reader = new Scanner(System.in);
int score1, score2;
System.out.print("Enter the first test score: ");
score1 = reader.nextInt();
System.out.print("Enter the second test score: ");
score2 = reader.nextInt();

// Managers must score well (90 or above) on both tests.
if (score1 >= 90 && score2 >= 90)
    System.out.println("Qualified to be a manager");

// Supervisors must score well (90 or above) on just one test
if (score1 >= 90 || score2 >= 90)
    System.out.println("Qualified to be a supervisor");

// Clerical workers must score moderately well on one test
// (70 or above), but not badly (below 50) on either.
if ((score1 >= 70 || score2 >= 70) &&
    !(score1 < 50 || score2 < 50))
    System.out.println("Qualified to be a clerk");
```

# Logical Operators (cont.)

- **Boolean variables:** Can have value of `true` or `false`
  - Primitive data type
  - Example:
    - ```
boolean b1 = true;
if ( b1 ) {
    <do something>
}
```

# Logical Operators (cont.)

- Can break down complex Boolean expressions into a series of simpler ones
- Useful equivalences:
  - $!(p \ || \ q) \rightarrow !p \ \&\& \ !q$
  - $!(p \ \&\& \ q) \rightarrow !p \ || \ !q$
  - $p \ || \ (q \ \&\& \ r) \rightarrow (p \ || \ q) \ \&\& \ (p \ || \ r)$
  - $p \ \&\& \ (q \ || \ r) \rightarrow (p \ \&\& \ q) \ || \ (p \ \&\& \ r)$

# Logical Operators (cont.)

- **Short-circuit evaluation:** JVM may know result of a Boolean expression before evaluating all of its parts.
  - Stops evaluation as soon as result is known
  - Example:
    - `if( true || false )`
    - Entire condition is `true` because first operand of the Boolean expression is `true`.
      - Second operand not examined at all

# Testing `if` Statements

- **Quality assurance:** Ongoing process of making sure that a software product is developed to the highest standards possible
  - Subject to the constraints of time and money
- Test data should try to achieve **complete code coverage**.
  - Every line in a program executed at least once

# Testing `if` Statements (cont.)

- All sets of test data that exercise a program in the same manner belong to same **equivalence class**.
  - Test same paths through the program
- Test data should include cases that assess program's behavior under **boundary conditions**.
  - On or near boundaries between equivalence classes

# Testing `if` Statements (cont.)

- Test under **extreme conditions**
  - With data at the limits of validity
- Test data validation rules
  - Enter values that are valid and invalid.
  - Test boundary values between the two.

# Nested `if` Statements

- Placing `if` statements within other `if` statements
- Alternative to using logical operators

```
if (the time is after 7 PM){  
    if (you have a book)  
        read the book;  
    else  
        watch TV;  
}else  
    go for a walk;
```

# Nested `if` Statements (cont.)

- A nested `if` statement can be translated to an equivalent `if` statement that uses logical operators.
  - The reverse is also true.

| AFTER 7 P.M. | HAVE A BOOK | ACTION TAKEN |
|--------------|-------------|--------------|
| true         | true        | read book    |
| true         | false       | watch TV     |
| false        | true        | walk         |
| false        | false       | walk         |

Table 6-6: Truth table for reading a book, watching TV, or going for a walk

# Nested `if` Statements (cont.)

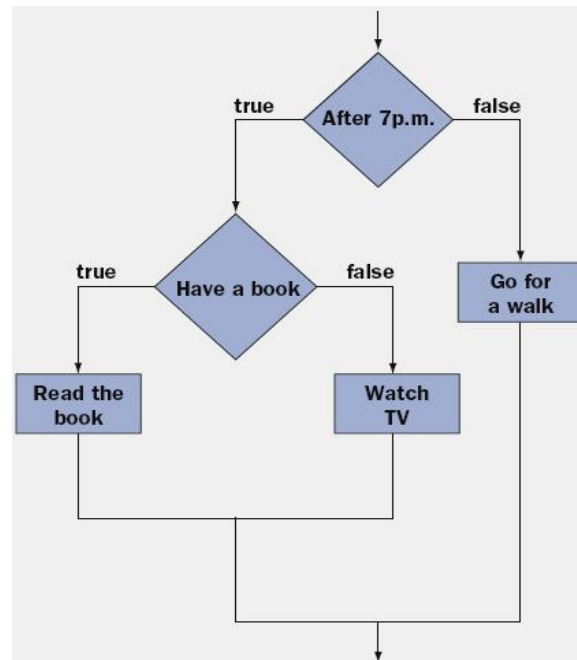


Figure 6-4: Flowchart for reading a book, watching TV, or going for a walk

# Nested if Statements (cont.)

- Example:

```
System.out.print("Enter the test average: ");
testAverage = reader.nextInt();
if (testAverage >= 90)
    System.out.println("grade is A");
else{
    if (testAverage >= 80)
        System.out.println("grade is B");
    else{
        if (testAverage >= 70)
            System.out.println("grade is C");
        else{
            if (testAverage >= 60)
                System.out.println("grade is D");
            else{
                System.out.println("grade is F");
            }
        }
    }
}
```

# Nested if Statements (cont.)

- **Extended if statement:**

```
System.out.print("Enter the test average: ");
testAverage = reader.nextInt();
if (testAverage >= 90)
    System.out.println("grade is A");
else if (testAverage >= 80)
    System.out.println("grade is B");
else if (testAverage >= 70)
    System.out.println("grade is C");
else if (testAverage >= 60)
    System.out.println("grade is D");
else
    System.out.println("grade is F");
```

# Logical Errors in Nested `if` Statements

- Misplaced braces example:

```
// Version 1
if (the weather is wet){
    if (you have an umbrella)
        walk;
    else
        run;
}
```

```
// Version 2
if (the weather is wet){
    if (you have an umbrella)
        walk;
}else
    run;
```

# Logical Errors in Nested `if` Statements (Cont.)

| THE WEATHER IS WET | YOU HAVE AN UMBRELLA | VERSION 1 OUTCOME | VERSION 2 OUTCOME |
|--------------------|----------------------|-------------------|-------------------|
| true               | true                 | walk              | walk              |
| true               | false                | run               | none              |
| false              | true                 | none              | run               |
| false              | false                | none              | run               |

Table 6-7: Truth table for version 1 and version 2

# Logical Errors in Nested `if` Statements (cont.)

- If braces removed, Java pairs the `else` with closest preceding `if`.
  - Can create logic or syntax errors if not careful
  - Can you spot the error in the following code?

```
if (the weather is wet)
    if (you have an umbrella)
        open umbrella;
        walk;
else
    run;
```

# Logical Errors in Nested `if` Statements (cont.)

- Better to over-use braces than under-use
- Order of conditions often important in extended `if-else` constructs
  - Can you spot the logic error?

```
if (sales >= 5000)
    commission = sales * 0.1;           // line a
else if (sales >= 10000)
    commission = sales * 0.2;         // line b
```

# Logical Errors in Nested `if` Statements (cont.)

- Avoiding nested `if` statements can improve clarity, as in:

`if (90 <= average ) grade is A;`

`if (80 <= average && average < 90) grade is B;`

`if (70 <= average && average < 80) grade is C;`

`if (60 <= average && average < 70) grade is D;`

`if ( average < 60) grade is F;`

# Nested Loops

- A loop within another loop
- Example:

```
System.out.print("Enter the lower limit: ");
lower = reader.nextInt();
System.out.print("Enter the upper limit: ");
upper = reader.nextInt();
for (n = lower; n <= upper; n++){
    innerLimit = (int)Math.sqrt (n);
    for (d = 2; d <= innerLimit; d++){
        if (n % d == 0)
            break;
    }
    if (d > innerLimit)
        System.out.println (n + " is prime");
}
```

# Testing Loops

- Loops increase difficulty of testing.
  - Often do not iterate some fixed number of times
  - Design test data to cover situation where loop iterates:
    - Zero times
    - One time
    - Multiple times

# Testing Loops (cont.)

```
// Display the proper divisors of a number
System.out.print("Enter a positive integer: ");
int n = reader.nextInt();
int limit = n / 2;
for (int d = 2; d <= limit; d++){
    if (n % d == 0)
        System.out.print (d + " ");
}
```

| TYPE OF TEST                                      | DATA USED      |
|---------------------------------------------------|----------------|
| No iterations                                     | 0, 1, 2, and 3 |
| One iteration                                     | 4 and 5        |
| Multiple iterations for a number with divisors    | 24             |
| Multiple iterations for a number without divisors | 29             |

Table 6-10: Test data for the count divisors program

# Testing Loops (cont.)

- **Combinatorial Explosion:** Creating test data to verify multiple dependant components can result in huge amount of test data.
  - Example:
    - 3 dependent components, each of which requires 5 tests to verify functionality
    - Total number of tests to verify entire program is  $5*5*5=125$ .

# Testing Loops (cont.)

- **Robust program:** Tolerates errors in user inputs and recovers gracefully
- Best and easiest way to write robust programs is to check user inputs immediately on entry.
  - Reject invalid user inputs.

# Loop Verification

- Process of guaranteeing that a loop performs its intended task
  - Independently of testing
- **assert statement:** Allows programmer to evaluate a Boolean expression and halt the program with an error message if the expression's value is false
  - General form:
    - `assert <Boolean expression>`

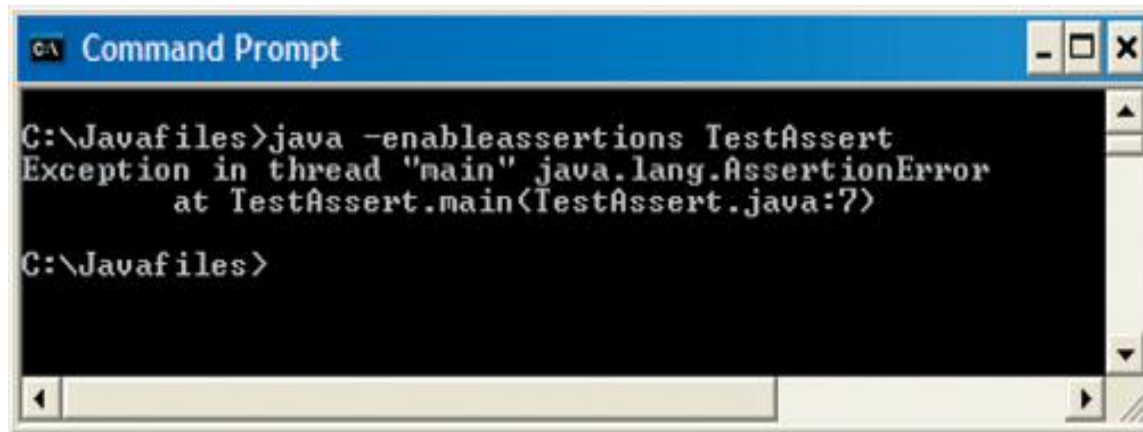
# Loop Verification (cont.)

- To enable when running the program:
  - `java -enableassertions AJavaProgram`

```
public class TestAssert{  
  
    public static void main(String[] args){  
        int x = 0;  
        assert x != 0;  
    }  
}
```

Example 6.1: Assert that `x != 0`

# Loop Verification (cont.)



```
CA Command Prompt
C:\Javafiles>java -enableassertions TestAssert
Exception in thread "main" java.lang.AssertionError
    at TestAssert.main<TestAssert.java:7>
C:\Javafiles>
```

Figure 6-6: Failure of an `assert` statement

# Loop Verification: Assertions with Loops

- **Input assertions:** State what can be expected to be true before a loop is entered
- **Output assertions:** State what can be expected to be true when the loop is exited

| INTEGER | PROPER DIVISORS | SUM |
|---------|-----------------|-----|
| 6       | 1, 2, 3         | 6   |
| 9       | 1, 3            | 4   |
| 12      | 1, 2, 3, 4, 6   | 16  |

Table 6-12: Sums of the proper divisors of some integers

# Loop Verification: Assertions with Loops (cont.)

- Sum proper divisors of positive integer:

```
divisorSum = 0;
for (trialDivisor = 1; trialDivisor <= num / 2; ++trialDivisor)
    if (num % trialDivisor == 0)
        divisorSum = divisorSum + trialDivisor;
```

- Input assertions:

- num is a positive integer
- divisorSum == 0

- Output assertion:

- divisorSum is sum of all proper divisors of num

# Loop Verification: Invariant and Variant Assertions

- **Loop invariant:** Assertion that expresses a relationship between variables that remains constant throughout all loop iterations
- **Loop variant:** Assertion whose truth changes between the first and final iteration
  - Stated so that it guarantees the loop is exited
- Usually occur in pairs

# Loop Verification: Invariant and Variant Assertions (cont.)

```
divisorSum = 0;

// 1. num is a positive integer.                (input assertion)
// 2. divisorSum == 0.

assert num > 0 && divisorSum == 0;

for (trialDivisor = 1; trialDivisor <= num / 2; ++trialDivisor)

    // trialDivisor is incremented by 1 each time        (variant assertion)
    // through the loop. It eventually exceeds the
    // value (num / 2), at which point the loop is exited.

    if (num % trialDivisor == 0)
        divisorSum = divisorSum + trialDivisor;

// divisorSum is the sum of proper divisors of          (invariant assertion)
// num that are less than or equal to trialDivisor.

// divisorSum is the sum of                            (output assertion)
// all proper divisors of num.
```

# Graphics and GUIs: Timers and Animations

- In animation, slightly different frames are displayed in order at a rapid rate.
  - Produces illusion of continuous movement
- Moving objects have:
  - Velocity
    - Distance (in pixels) traveled in a given unit of time
  - Direction

# Graphics and GUIs: Timers and Animations (cont.)

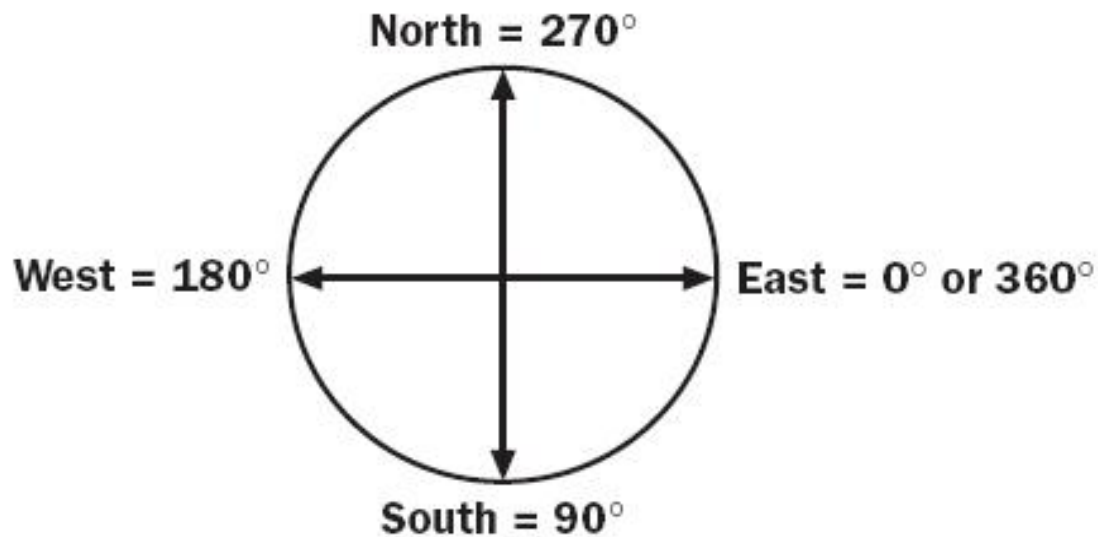


Figure 6-7: Representing directions in two dimensions

# Graphics and GUIs: Timers and Animations (cont.)

- Algorithm for animating a graphical object:

```
Set the initial position of the shape
At regular intervals
  Move the object
  Repaint the panel
```

- Java provides a **timer** object to schedule events at regular intervals.

# Graphics and GUIs: Timers and Animations (cont.)

- Timer for animations is an instance of the class `Timer`.

```
javax.swing.Timer projectortimer;  
timer = new javax.swing.Timer(5, new MoveListener());  
timer.start();
```

```
private class MoveListener implements ActionListener{  
  
    public void actionPerformed(ActionEvent e){  
        // Code for moving objects goes here  
        repaint();  
    }  
}
```

# Graphics and GUIs: Timers and Animations (cont.)

| TIMER METHOD                                 | WHAT IT DOES                                                                 |
|----------------------------------------------|------------------------------------------------------------------------------|
| <code>boolean isRunning()</code>             | Returns true if the timer is firing events or false otherwise                |
| <code>void restart()</code>                  | Restarts a timer, causing it to fire the first event after its initial delay |
| <code>void setDelay(int delay)</code>        | Sets the timer's delay to the number of milliseconds between events          |
| <code>void setInitialDelay(int delay)</code> | Sets the timer's initial delay, which by default is its between-event delay  |
| <code>void stop()</code>                     | Stops the timer, causing it to cease firing events                           |

Table 6-15: Some commonly used `Timer` methods

# Design, Testing, and Debugging Hints

- Most errors involving selection statements and loops are not syntax errors caught at compile time.
- The presence or absence of braces can seriously affect the logic of a selection statement or loop.

# Design, Testing, and Debugging Hints (cont.)

- When testing programs with `if` or `if-else` statements, use data that force the program to exercise all logical branches.
- When testing programs with `if` statements, formulate equivalence classes, boundary conditions, and extreme conditions.
- Use an `if-else` statement rather than two `if` statements when the alternative courses of action are mutually exclusive.

# Design, Testing, and Debugging Hints (cont.)

- When testing a loop, use limit values as well as typical values.
- Check entry and exit conditions for each loop.
- For a loop with errors, use debugging output statements to verify the control variable's value on each pass through the loop.
  - Check value before the loop is initially entered, after each update, and after the loop is exited.

# Summary

- A complex Boolean expression contains one or more Boolean expressions and the logical operators `&&` (AND), `||` (OR), and `!` (NOT).
- A truth table can determine the value of any complex Boolean expression.
- Java uses short-circuit evaluation of complex Boolean expressions.

## Summary (cont.)

- Nested `if` statements are another way of expressing complex conditions.
- A nested `if` statement can be translated to an equivalent `if` statement that uses logical operators.
- An extended or multiway `if` statement expresses a choice among several mutually exclusive alternatives.

## Summary (cont.)

- Loops can be nested in other loops.
- Equivalence classes, boundary conditions, and extreme conditions are important features used in tests of control structures involving complex conditions.
- Loops can be verified to be correct by using assertions, loop variants, and loop invariants.