Lesson 6: Control Statements Continued

Updated for Java 1.5,

(with additions and modifications by)
Mr. Dave Clausen

Lesson 6: Control Statements Continued

Objectives:

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

Lesson 6: Control Statements Continued

Vocabulary:

- arithmetic overflow
- boundary condition
- combinatorial explosion
- complete code testingcoverage
- equivalence class
- extended if statement
- extreme condition
- logical operator
- nested if statement
- nested loop
- quality assurance
- robust
- truth table

6.1 Logical Operators

- Java includes three logical operators equivalent in meaning to the English words AND, OR, and NOT.
- These operators are used in the Boolean expressions that control the behavior of if, while, and for statements.
- For instance, consider the following sentences:

1. If the sun is shining AND it is 8 am then let's go for a walk else let's stay home.
2. If the sun is shining OR it is 8 am then let's go for a walk else let's stay home.
3. If NOT the sun is shining then let's go for a walk else let's stay home.

6.1 Logical Operators

- The phrases "the sun is shining" and "it is 8 am" are operands and the words AND, OR, and NOT are operators.
- In the first sentence, the operator is AND.
  - Consequently, if both operands are true, the condition as a whole is true. If either or both are false, the condition is false.
- In the second sentence, which uses OR, the condition is false only if both operands are false; otherwise, it is true.
- In the third sentence, the operator NOT has been placed before the operand, as it would be in Java.
  - If the operand is true, then the NOT operator makes the condition as a whole false.

6.1 Logical Operators

- We summarize these observations in the three parts of Table 6-1.
- Each part is called a ***truth table***, and it shows how the value of the overall condition depends on the values of the operands.
- When there is one operand, there are two possibilities. For two operands, there are four; and for three operands, there are eight.
- In general there are 2^n combinations of true and false for n operands.

6.1 Logical Operators

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

6.1 Logical Operators

- Dropping the column labeled "action taken," we can combine the three truth tables into one, as illustrates in Table 6-2.
- The letters P and Q represent the operands.

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

6.1 Logical Operators and ( )

1. ***If*** (the sun is shining **AND** it is 8 am) **OR** (**NOT** your brother is visiting) ***then*** let's go for a walk ***else*** let's stay at home.
   - Expressions inside parentheses are evaluated before those that are not.
   - We will go for a walk at 8 A.M. on sunny days or when your brother does not visit.
2. ***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.
   - Before we go for a walk, the sun must be shining.
   - In addition, one of two things must be true.
   - Either it is 8 am or your brother is not visiting.

6.1 Logical Operators

Java's Logical Operators and Their Precedence

- In Java the operators AND, OR, and NOT are represented by &&, ||, and !, respectively.
- Their precedence is shown in Table 6-3.
- Observe that NOT (!) has the same high precedence as other unary operators, while AND (&&) and OR (||) have low precedence, with OR below AND.

6.1 Logical Operators

OPERATION	SYMBOL	PRECEDENCE (FROM HIGHEST TO LOWEST)	ASSOCIATION
Grouping	( )	1	Not applicable
Method selector	.	2	Left to right
Unary plus	+	3	Not applicable
Unary minus	–	3	Not applicable
Not	!	3	Not applicable
Multiplication	*	4	Left to right
Division	/	4	Left to right
Remainder or modulus	%	4	Left to right
Addition	+	5	Left to right
Subtraction	–	5	Left to right
Relational operators	< <= > >= == !=	6	Not applicable
And	&&	8	Left to right
Or	||	9	Left to right
Assignment operators	= *= /= %= += -=	10	Right to left

6.1 Logical Operators

Examples

- Following are some illustrative examples based on the employment practices at ABC Company.
- The company screens all new employees by making them take two written tests.
- A program then analyzes the scores and prints a list of jobs for which the applicant is qualified. Here is the relevant code:

6.1 Logical Operators

```
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");
```

6.1 Logical Operators

Boolean Variables

- The complex Boolean expressions in the preceding examples can be simplified by using Boolean variables.
- A Boolean variable can be true or false and is declared to be of type boolean.
- Primitive data type
- Example:
  - ```
    boolean b1 = true;
    if ( b1 )
    {
        <do something>
    }
    ```
    ```

6.1 Logical Operators

```
Scanner reader = new Scanner (System.in);
int score1, score2;
boolean bothHigh, atLeastOneHigh, atLeastOneModerate, noLow;
System.out.print ("Enter the first test score: ");
score1 = reader.nextInt( );
System.out.print ("Enter the second test score: ");
score2 = reader.nextInt( );
bothHigh          =  (score1 >= 90 && score2 >= 90); // parentheses
atLeastOneHigh    =  (score1 >= 90 || score2 >= 90); // optional
atLeastOneModerate =  (score1 >= 70 || score2 >= 70); // here
noLow             = !(score1 <  50 || score2 <  50);
if (bothHigh)
   System.out.println("Qualified to be a manager");
if (atLeastOneHigh)
   System.out.println("Qualified to be a supervisor");
if (atLeastOneModerate && noLow)
   System.out.println("Qualified to be a clerk");
```

6.1 Logical Operators

In order to rewrite the previous code, we first create a truth table for the complex if statement, as shown in Table 6-4.

P: THE SUN SHINES	Q: YOU HAVE TIME	R: IT IS SUNDAY	P && (Q || R)	ACTION TAKEN
true	true	true	true	walk
true	true	false	true	walk
true	false	true	true	walk
true	false	false	false	stay home
false	true	true	false	stay home
false	true	false	false	stay home
false	false	true	false	stay home
false	false	false	false	stay home

6.1 Logical Operators

- Then implement each line of the truth table with a separate if statement involving only && (AND) and ! (NOT).
- Applying the technique here yields:

```
if ( the sun shines &&  you have time &&  it is Sunday) walk;
if ( the sun shines &&  you have time && !it is Sunday) walk;
if ( the sun shines && !you have time &&  it is Sunday) walk;
if ( the sun shines && !you have time && !it is Sunday) stay
home;
if (!the sun shines &&  you have time &&  it is Sunday) stay
home;
if (!the sun shines &&  you have time && !it is Sunday) stay
home;
if (!the sun shines && !you have time &&  it is Sunday) stay
```

6.1 Logical Operators

- In this particular example, the verbosity can be reduced without reintroducing complexity by noticing that the first two if statements are equivalent to:

  if (the sun shines && you have time) walk;

- And the last four are equivalent to

  if (!the sun shines) stay home;

- Putting all this together yields:

```
if ( the sun shines &&  you have time) walk;
if ( the sun shines && !you have time &&  it is Sunday)
walk;
if (!the sun shines) stay home;
```

6.1 Logical Operators

Some Useful Boolean Equivalences

- There is often more than one way to write a Boolean expression.
- For instance, the following pairs of Boolean expressions are equivalent as truth tables readily confirm: (The first two are De Morgan's Laws)

!(p || q)	equivalent to	!p && !q
!(p && q)	equivalent to	!p || !q
p || (q && r)	equivalent to	(p || q) && (p || r)
p && (q || r)	equivalent to	(p && q) || (p && r)

6.1 Logical Operators

Short-circuit evaluation

- The Java virtual machine sometimes knows the value of a Boolean expression before it has evaluated all of its parts.
  - In the expression (p && q), if p is false, then so is the expression, and there is no need to evaluate q.
  - In the expression (true || false) the entire condition is true because the first operand of the Boolean expression is true, the second operand is not examined at all
- When evaluation stops as soon as possible, is called ***short-circuit evaluation***.
- In contrast, some programming languages use ***complete evaluation*** in which all parts of a Boolean expression are always evaluated.

Case Study 1

- Compute Weekly Pay

PayrollSystemApp.java

PayrollSystemApp.txt

PayrollSystemInterfaceWithOutBreak.java

PayrollSystemInterfaceWithOutBreak.txt

Employee.java

Employee.txt

Testing String Equivalence

- Be careful when testing the equivalence of Strings (Remember that Strings are objects, not primitive data types):

  String a, b;

  The Boolean, if (a = = b) only determines if a and b reference the same object.

  The expression, if (a.equals(b)) determines is the Strings are equal and identical.

6.2 Testing if Statements

- ***Quality assurance*** is the ongoing process of making sure that a software product is developed to the highest standards possible subject to the ever-present constraints of time and money.
- Faults are fixed most inexpensively early in the development life cycle.
- Test data should try to achieve ***complete code coverage***, which means that every line in a program is executed at least once.

6.2 Testing if Statements

- All the sets of test data that exercise a program in the same manner are said to belong to the same ***equivalence class***, which means they are equivalent from the perspective of testing the same paths through the program.
- The test data should also include cases that assess a program's behavior under ***boundary conditions*** - that is, on or near the boundaries between equivalence classes.
- We should test under ***extreme conditions*** - that is, with data at the limits of validity.

6.2 Testing if Statements

- Data validation rules should also be tested.
- We need to enter values that are valid and invalid, and we must test the boundary values between the two.

6.2 Testing if Statements

Table 6-5 summarizes our planned tests:

TYPE OF TEST	DATA USED
Code coverage	employee type: 1 hourly rate: 10 hours worked: 30 and 50
Boundary conditions	employee type: 1 hourly rate: 10 hours worked: 39, 40, and 41
Extreme conditions	employee type: 1 hourly rate: 10 hours worked: 0 and 168
Tests when the employee type is 2	employee type: 2 hourly rate: 10 hours worked: 30 and 50
Data validation rules	type: 0, 1, 2, and 3 hourly rate: 6.49, 6.50, 10, 30.50, and 30.51 hours worked: 0, 1, 30, 60, and 61

6.3 Nested if Statements

- Here is an everyday example of nested if's written in Javish:

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

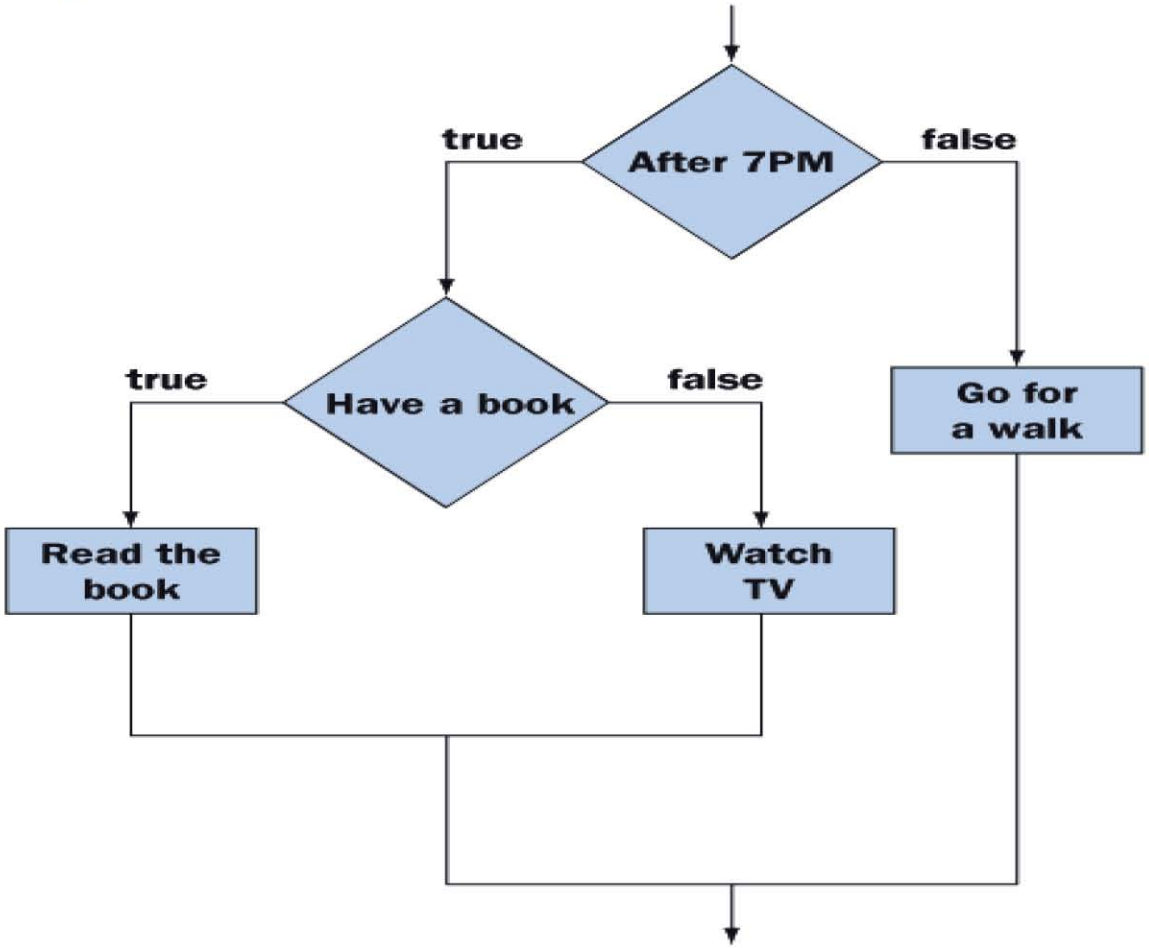
6.3 Nested if Statements

- Although this code is not complicated, it is a little difficult to determine exactly what it means without the aid of the truth table illustrated in Table 6-6.

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

6.3 Nested if Statements

- As a substitute for a truth table, we can draw a flowchart as shown in Figure 6-4.



Nested if Statements

- Nested if statement
  - an if statement used within another if statement where the "true" statement or action is.

```
if (score >=50)
    if (score>=69.9)
       //blah, blah, blah true for score>=69.9 and score>=50
    else
      //blah, blah, blah false score>=69.9 true score >=50
else
    //blah, blah, blah  false for score >=50
```

Extended if statements

- Extended if statements are Nested if statements where another if statement is used with the ***else*** clause of the original if statement.

```
if  (condition 1)
     action1
else if (condition2)
          action2
       else
            action3
```

Extended if Example Form 1

```
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");
         }
      }
   }
}
```

Extended if Example Form 2

- This is the same example using different indentation.

```
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");
```

6.4 Logical Errors in Nested ifs

Misplaced Braces

- One of the most common mistakes involves misplaced braces.

```
// 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;
```

6.4 Logical Errors in Nested ifs

- To demonstrate the differences between the two versions, we construct a truth table - as shown in Table 6-7:

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

6.4 Logical Errors in Nested ifs

Removing the Braces

- When the braces are removed, Java pairs the else with the 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;//what's missing here?
      walk;              //what's missing here?
   else
      run;
```

6.4 Logical Errors in Nested ifs

Computation of Sales Commissions

- We now attempt to compute a salesperson's commission and introduce a logical error in the process.
- Commissions are supposed to be computed as follows:
  - 10% if sales are greater than or equal to $5,000
  - 20% if sales are greater than or equal to $10,000

```
if (sales >= 5000)
   commission = commission * 1.1;      // line a
else if (sales >= 10000)
   commission = commission * 1.2;      // line b
//order of ifs should be reversed
```

6.4 Logical Errors in Nested ifs

- To determine if the code works correctly, we check it against representative values for the sales, namely, sales that are:
  - less than $5,000,
  - equal to $5,000,
  - between $5,000 and $10,000,
  - equal to $10,000,
  - and greater than $10,000.
- As we can see from Table 6-8, the code is not working correctly.

6.4 Logical Errors in Nested ifs

Table 6-8

VALUE OF SALES	LINES EXECUTED	VALIDITY
1,000	neither line a nor line b	correct
5,000	line a	correct
7,000	line a	correct
10,000	line a	incorrect
12,000	line a	incorrect

6.4 Logical Errors in Nested ifs

Corrected Computation of Sales Commissions

- After a little reflection, we realize that the conditions are in the wrong order.
- Here is the corrected code:

```
if (sales >= 10000)
   commission = commission * 1.2;       // line b
else if (sales >= 5000)
   commission = commission * 1.1;       // line a
```

6.4 Logical Errors in Nested ifs

- Table 6-9 confirms that the code now works correctly:

VALUE OF SALES	LINES EXECUTED	VALIDITY
1,000	neither line a nor line b	correct
5,000	line a	correct
7,000	line a	correct
10,000	line b	correct
12,000	line b	correct

6.4 Logical Errors in Nested ifs

Avoiding Nested ifs (Author's opinion)

- Sometimes getting rid of nested if's is the best way to avoid logical errors.
- This is easily done by rewriting nested ifs as a sequence of independent if statements.

```
if (5000 <= sales && sales < 10000)
   commission = commission * 1.1;
if (10000 <= sales)
   commission = commission * 1.2;
```

6.4 Logical Errors in Nested ifs

- And here is another example involving the calculation of student grades:

```
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;
```

Avoid Sequential Selection

- Mr. Clausen's Opinion:
- This is not a good programming practice.
  - Less efficient
  - only one of the conditions can be true
    - these are called mutually exclusive conditions

```
if (condition1)                    //avoid this structure
     action1                       //use extended selection
if (condition2)
     action2
if (condition3)
     action3
```

- I suggest avoiding nested ifs, while using extended ifs.

Switch Statements

- Allows for multiple selection that is easier to follow than nested or extended if statements.

(Not part of APCS Subset)

```
switch (age)   //age is of type int
{
   case 18:      <statement1>
                 break;  //"only acceptable use of break
   case 19:      <statement2>
                 break; //in beginning course" (Mr. C.)
   case 20:      <statement3>
                 break;
   default:      <default statement>
}
```

Switch: Flow of Execution

- The selector (argument) for switch must be of an ordinal type (not double)
  - switch (age)
  - The variable "age" is called the selector in our example.
  - If the first instance of the variable is found among the labels, the statement(s) following this value is executed until reaching the next break statement.
  - Program control is then transferred to the next statement following the entire switch statement.
  - If no value is found, the default statement is executed.
  - Switch statements are the same in Java as in C++

Switch Statement Example 2

```
switch (grade)              //grade is of type char
{
    case 'A' :
    case 'B' :      System.out.println("Good work!");
                    break;
    case 'C' :      System.out.println(" Average work");
                    break;
    case 'D' :
    case 'F' :       System.out.println(" Poor work");
                    break;
    default  :       System.out.println("'" + grade + " is not a valid
                    letter grade.");
                    break;
}
```

6.5 Nested Loops

- There are many programming situations in which loops are nested within loops - these are called nested loops.
- For example (don't use break to exit nested loops either):

```
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");
}
```

Nested loops w/o break

```
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);
   while((divisor<= innerLimit ) && (userNumber % divisor != 0))
     {
       divisor++;
     }

   if (d > limit)
      System.out.println (n + " is prime");
}
```

Nested Loops 2

- Nested loop
  - when a loop is one of the statements within the body of another loop.

```
for (k=1; k<=5; ++k)     //outer loop
   for (j=1; j<=3; ++j)   //inner loop
      System.out.println(k+j);  //body of loop
```

  - Each loop needs to have its own level of indenting.
  - Use comments to explain each loop
  - Blank lines around each loop can make it easier to read

Repetition and Selection

- The use of an if statement within a loop to look for a certain condition in each iteration of the loop. (not to break out of the loop)
  - Examples:
    - to generate a list of Pythagorean Triples
    - to perform a calculation for each employee
    - to find prime numbers

Repetition and Selection Example

```
//void List_All_Primes(int number){
    boolean prime;
    int candidate, divisor;
    double limit_for_check;
    for (candidate = 2; candidate <= number; candidate++){
        prime = true;
        divisor = 2;
        limit_for_check = Math.sqrt(candidate);
        while ((divisor <= limit_for_check) && prime)
            if (candidate % divisor == 0)
                prime = false; // candidate has a divisor
            else
                divisor = divisor + 1;
        if (prime)   //Print list of primes
            System.out.println (candidate + "is prime");
    }
}
```

6.6 Testing Loops

- The presence of looping statements in a program increases the challenge of designing good test data.
- When designing test data, we want to cover all possibilities.
- Loops often do not iterate some fixed number of times
- We need to design test data to cover situations where a loop iterates:
  - Zero times
  - One time
  - Multiple times

6.6 Testing Loops

- To illustrate, we develop test data for the print divisors program:

```
// 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 + " ");
}
```

6.6 Testing Loops

- By analyzing the code, we conclude that if n equals 0, 1, 2, or 3, the limit is less than 2, and the loop is never entered.
- If n equals 4 or 5, the loop is entered once.
- If n is greater than 5, the loop is entered multiple times.
- All this suggests the test data shown in Table 6-10.

6.6 Testing Loops

Table 6-10

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

6.6 Testing Loops

- **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.

6.6 Testing Loops

- **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.

Case Study 2

Fibonacci.java

Fibonacci.txt

FibonacciNoBreaks.java

FibonacciNoBreaks.txt

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`
- Example 6.1: Assert that x != 0

```
public class TestAssert{

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

Loop Verification (cont.)

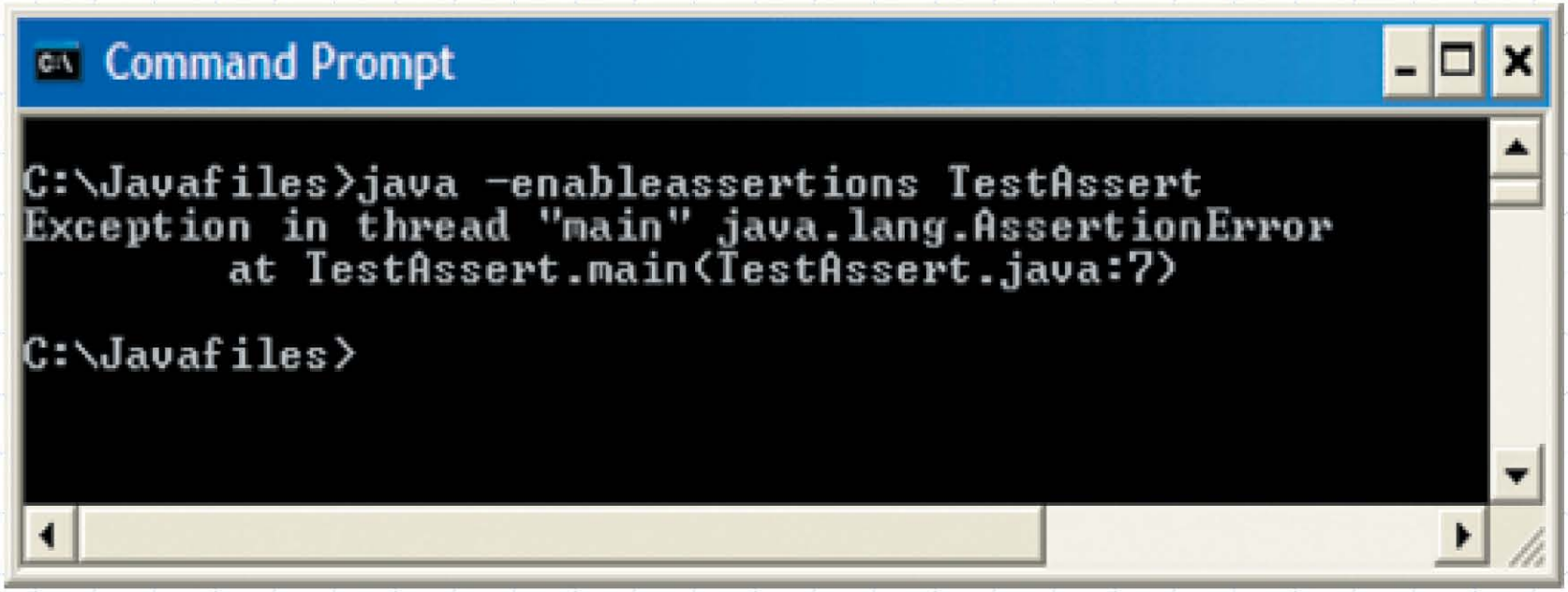


Figure 6-6: Failure of an `assert` statement

Loop Verification: Assertions with Loops

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

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.)

- Some 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

```
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.
```

Using assert with JCreator

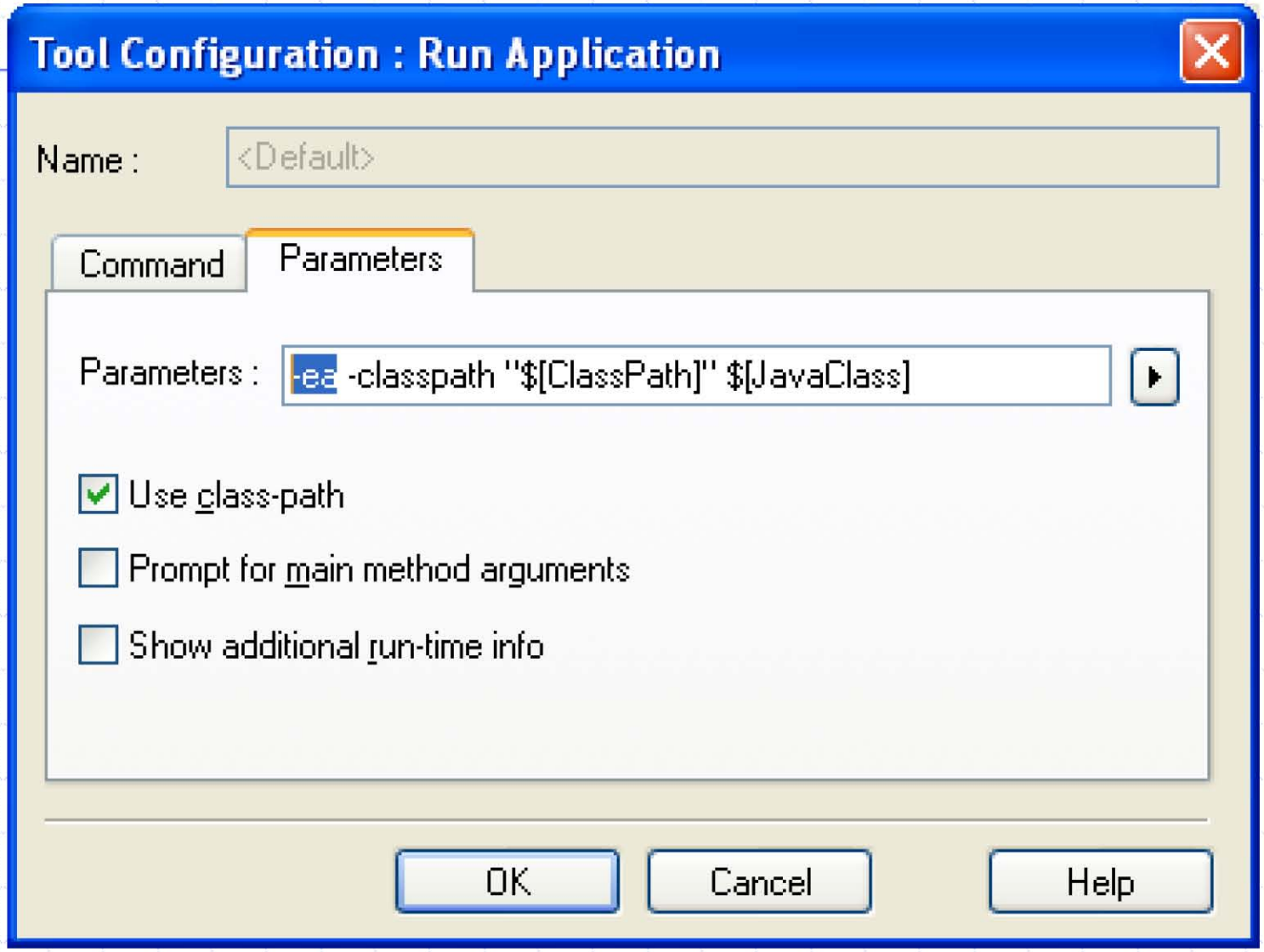
To use assertions with Jcreator:

1. Go to **Configure** menu > **Options**
2. Now in the "Options" dialog, click on **JDK Tools** item on the left
3. From the **"Select Tool Type"** drop down list on the right side, select **"Run Applications"**.
4. In the list below, click **"<Default>"** (even though it is "grayed out") and click the **Edit** button
5. In the "Tool Configuration: Run Application" dialog box, click the

"Parameters" tab

6. In the "Parameters" text box, begin with **–ea** or **-enableassertions** before -classpath "$[ClassPath]" $[JavaClass]

Jcreator assert Setup



Assert Test

TestAssert.java

TestAssert.txt

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.