

ICOM 4015: Advanced Programming

Lecture 6

Chapter Six: Iteration



Chapter Six: Iteration

Big Java by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.

Chapter Goals

- To be able to program loops with the `while`, `for`, and `do` statements
- To avoid infinite loops and off-by-one errors
- To understand nested loops
- To learn how to process input
- To implement simulations
- To learn about the debugger

while **Loops**

- Executes a block of code repeatedly
- A condition controls how often the loop is executed

```
while (condition)  
    statement
```

- Most commonly, the statement is a block statement (set of statements delimited by { })

Calculating the Growth of an Investment

- Invest \$10,000, 5% interest, compounded annually

Year	Balance
0	\$10,000
1	\$10,500
2	\$11,025
3	\$11,576.25
4	\$12,155.06
5	\$12,762.82

Calculating the Growth of an Investment

- When has the bank account reached a particular balance?

```
while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

ch06/invest1/Investment.java

```
01: /**
02:     A class to monitor the growth of an investment that
03:     accumulates interest at a fixed annual rate.
04: */
05: public class Investment
06: {
07:     /**
08:         Constructs an Investment object from a starting balance and
09:         interest rate.
10:         @param aBalance the starting balance
11:         @param aRate the interest rate in percent
12:     */
13:     public Investment(double aBalance, double aRate)
14:     {
15:         balance = aBalance;
16:         rate = aRate;
17:         years = 0;
18:     }
19:
20:     /**
21:         Keeps accumulating interest until a target balance has
22:         been reached.
23:         @param targetBalance the desired balance
24:     */
```

ch06/invest1/Investment.java (cont.)

```
25:     public void waitForBalance(double targetBalance)
26:     {
27:         while (balance < targetBalance)
28:         {
29:             years++;
30:             double interest = balance * rate / 100;
31:             balance = balance + interest;
32:         }
33:     }
34:
35:     /**
36:      Gets the current investment balance.
37:      @return the current balance
38:     */
39:     public double getBalance()
40:     {
41:         return balance;
42:     }
43:
44:     /**
45:      Gets the number of years this investment has accumulated
46:      interest.
```

ch06/invest1/Investment.java (cont.)

```
47:         @return the number of years since the start of the investment
48:     */
49:     public int getYears()
50:     {
51:         return years;
52:     }
53:
54:     private double balance;
55:     private double rate;
56:     private int years;
57: }
```

ch06/invest1/InvestmentRunner.java

```
01: /**
02:     This program computes how long it takes for an investment
03:     to double.
04: */
05: public class InvestmentRunner
06: {
07:     public static void main(String[] args)
08:     {
09:         final double INITIAL_BALANCE = 10000;
10:         final double RATE = 5;
11:         Investment invest = new Investment(INITIAL_BALANCE, RATE);
12:         invest.waitForBalance(2 * INITIAL_BALANCE);
13:         int years = invest.getYears();
14:         System.out.println("The investment doubled after "
15:             + years + " years");
16:     }
17: }
```

ch06/invest1/InvestmentRunner.java (cont.)

Output:

The investment doubled after 15 years

Animation 6.1 –

Program Code

```
int i = 1;
int s = 0;
while (i <= 10)
{
    s = s + i;
    i++;
}
```

Local Variables

i	s

This animation demonstrates the process of hand tracing a loop. When you trace a loop, you keep track of the current line of code and the current values of the variables. Whenever a variable's value changes, you cross out the old value and write in the new value. Click on the "Next" button to see the next tracing step. It is a good idea for you to predict what action will occur *before* hitting the button. Keep clicking the "Next" button until the loop exits.

6-01 Tracing a Loop



while Loop Flowchart

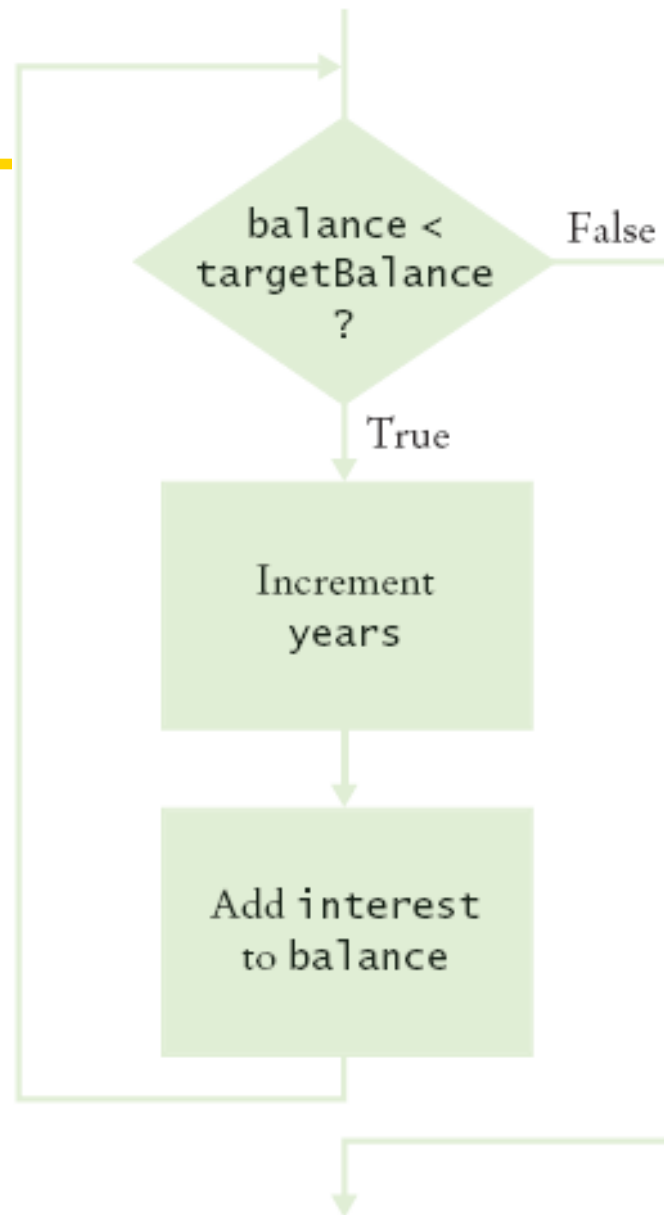


Figure 1 Flowchart of a while Loop

Syntax 6.1 The `while` Statement

```
while (condition)  
    statement
```

Example:

```
while (balance < targetBalance)  
{  
    years++;  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

Purpose:

To repeatedly execute a statement as long as a condition is true.

Self Check 6.1

How often is the statement in the loop

```
while (false) statement;
```

executed?

Answer: Never.

Self Check 6.2

What would happen if `RATE` was set to 0 in the `main` method of the `InvestmentRunner` program?

Answer: The `waitForBalance` method would never return due to an infinite loop.

Common Error: Infinite Loops

- ```
int years = 0;
while (years < 20)
{
 double interest = balance * rate / 100;
 balance = balance + interest;
}
```
- ```
int years = 20;
while (years > 0)
{
    years++; // Oops, should have been years-
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```
- Loops run forever – must kill program

Common Error: Off-by-One Errors

```
• int years = 0;
  while (balance < 2 * initialBalance)
  {
      years++;
      double interest = balance * rate / 100;
      balance = balance + interest;
  }
  System.out.println("The investment reached the target
    after " + years + " years.");
```

Should `years` start at 0 or 1?

Should the test be `<` or `<=`?

Avoiding Off-by-One Error

- Look at a scenario with simple values:
initial balance: \$100
interest rate: 50%
after year 1, the balance is \$150
after year 2 it is \$225, or over \$200
so the investment doubled after 2 years
the loop executed two times, incrementing `years` each time
Therefore: `years` must start at 0, not at 1.
- interest rate: 100%
after one year: balance is $2 * \text{initialBalance}$
loop should stop
Therefore: must use `<`
- Think, don't compile and try at random

do **Loops**

- Executes loop body at least once:

```
do
    statement
while (condition);
```

- Example: Validate input

```
double value;
do
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
}
while (value <= 0);
```

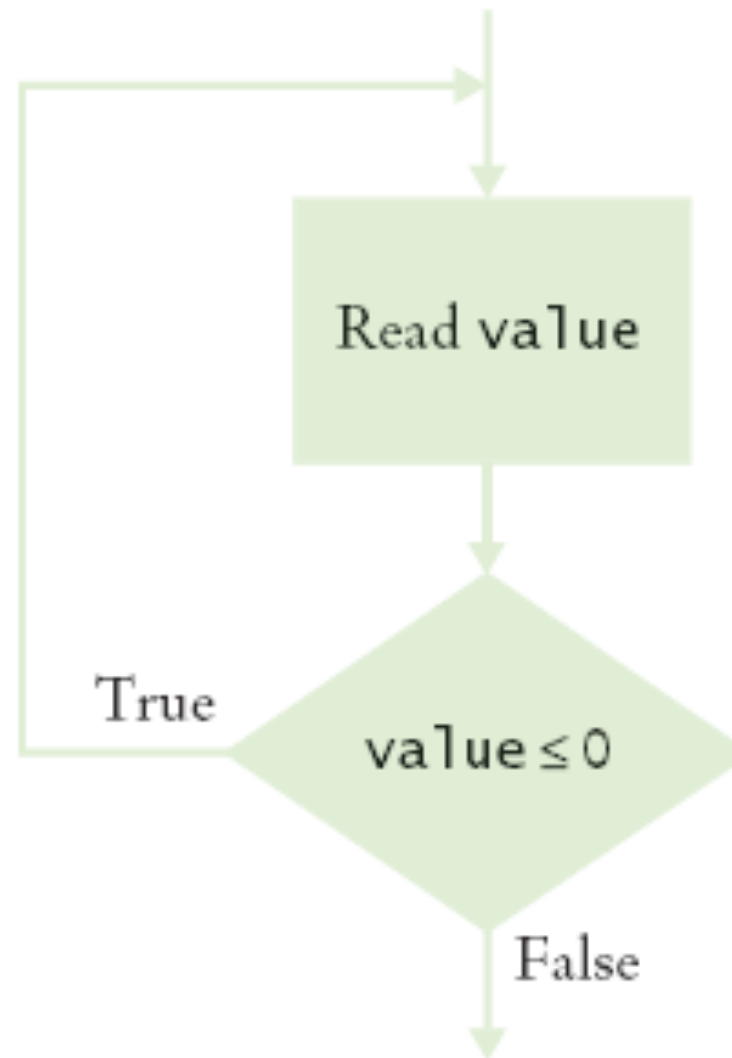
Continued

do **Loops (cont.)**

- **Alternative:**

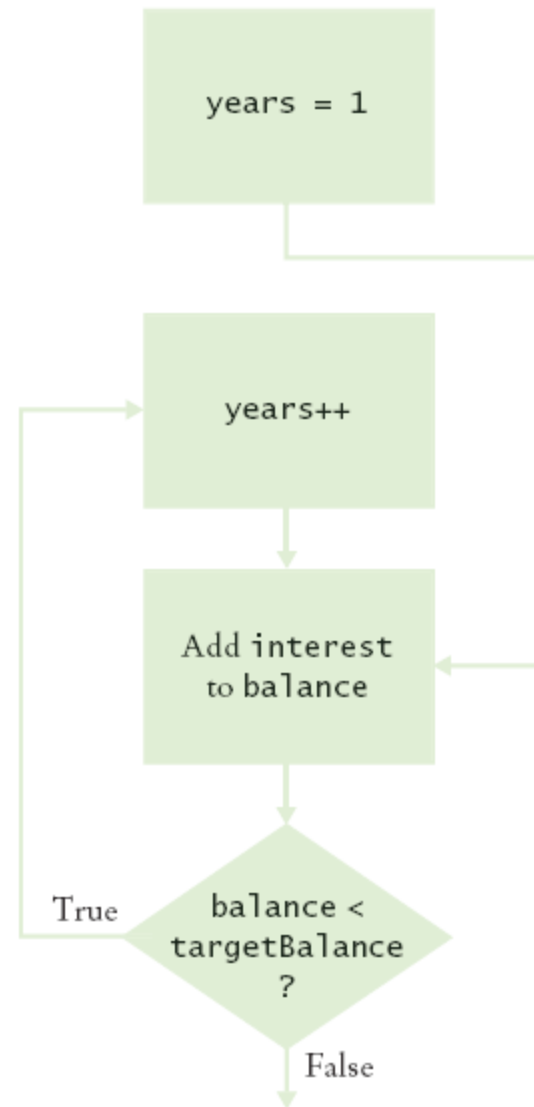
```
boolean done = false;
while (!done)
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
    if (value > 0) done = true;
}
```

do Loop Flowchart



Flowchart of a do Loop

Spaghetti Code



Spaghetti Code

for **Loops**

- `for (initialization; condition; update)`
 statement

- **Example:**

```
for (int i = 1; i <= n; i++)  
{  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

- **Equivalent to**

```
initialization;  
while (condition)  
{ statement;  
  update; }
```

Continued

for **Loops** (cont.)

- Other examples:

```
for (years = n; years > 0; years--) . . .
```

```
for (x = -10; x <= 10; x = x + 0.5) . . .
```

for Loop Flowchart

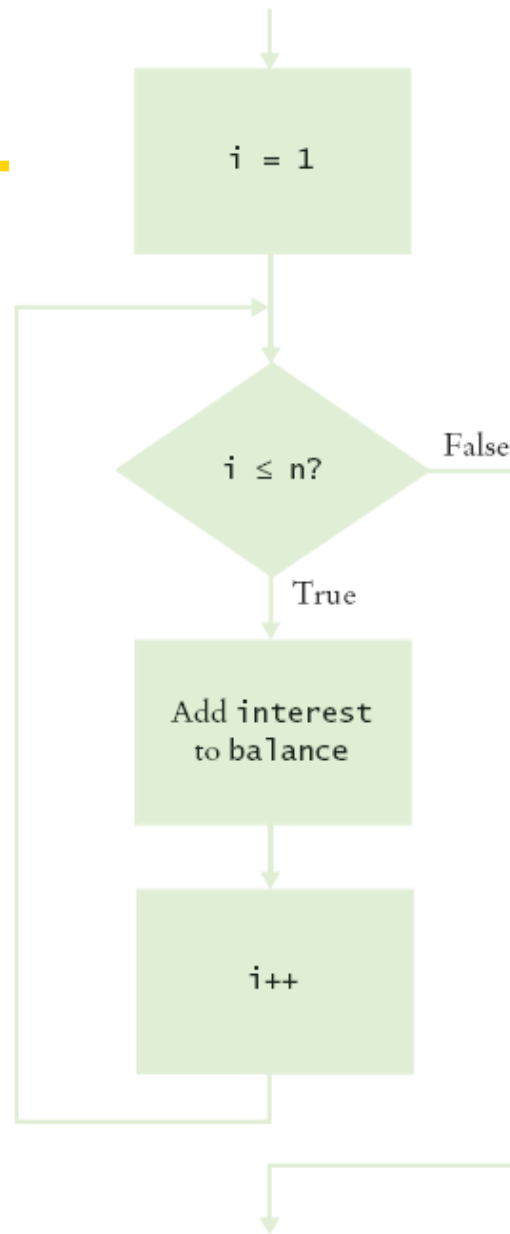


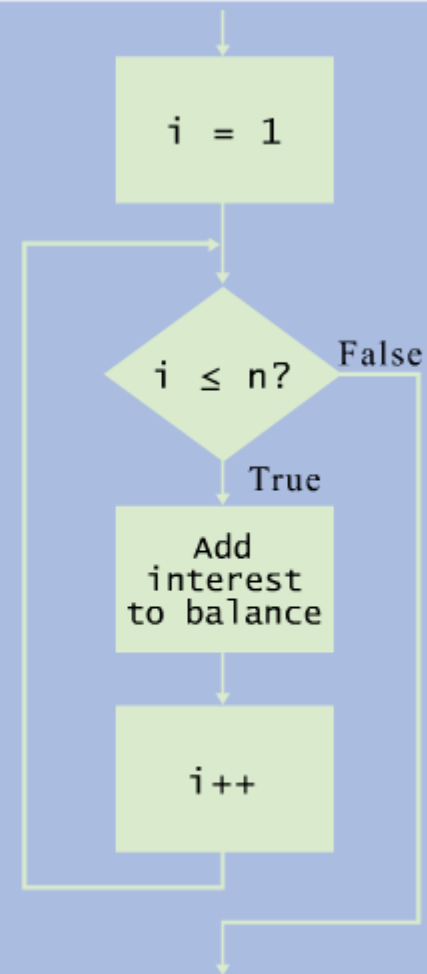
Figure 2 Flowchart of a for Loop

Animation 6.2 –

```
for (int i = 1; i <= n; i++)  
{  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

i	n	balance	rate	interest
	3	1000	10	

This animation demonstrates the for loop.



Syntax 6.2 The `for` Statement

```
for (initialization; condition; update)  
    statement
```

Example:

```
for (int i = 1; i <= n; i++)  
{  
    double interest = balance * rate / 100;  
    balance = balance + interest;  
}
```

Purpose:

To execute an initialization, then keep executing a statement and updating an expression while a condition is true.

ch06/invest2/Investment.java

```
01: /**
02:     A class to monitor the growth of an investment that
03:     accumulates interest at a fixed annual rate
04: */
05: public class Investment
06: {
07:     /**
08:         Constructs an Investment object from a starting balance and
09:         interest rate.
10:         @param aBalance the starting balance
11:         @param aRate the interest rate in percent
12:     */
13:     public Investment(double aBalance, double aRate)
14:     {
15:         balance = aBalance;
16:         rate = aRate;
17:         years = 0;
18:     }
19:
20:     /**
21:         Keeps accumulating interest until a target balance has
22:         been reached.
```

ch06/invest2/Investment.java (cont.)

```
23:         @param targetBalance the desired balance
24:     */
26:     {
27:         while (balance < targetBalance)
28:         {
29:             years++;
30:             double interest = balance * rate / 100;
31:             balance = balance + interest;
32:         }
33:     }
34:
35:     /**
36:      Keeps accumulating interest for a given number of years.
37:      @param n the number of years
38:     */
39:     public void waitYears(int n)
40:     {
41:         for (int i = 1; i <= n; i++)
42:         {
43:             double interest = balance * rate / 100;
44:             balance = balance + interest;
```

ch06/invest2/Investment.java (cont.)

```
45:         }
46:         years = years + n;
47:     }
48:
49:     /**
50:      * Gets the current investment balance.
51:      * @return the current balance
52:      */
53:     public double getBalance()
54:     {
55:         return balance;
56:     }
57:
58:     /**
59:      * Gets the number of years this investment has accumulated
60:      * interest.
61:      * @return the number of years since the start of the investment
62:      */
63:     public int getYears()
64:     {
65:         return years;
66:     }
```

ch06/invest2/Investment.java (cont.)

```
67:
68:     private double balance;
69:     private double rate;
70:     private int years;
71: }
```

ch06/invest2/InvestmentRunner.java

```
01: /**
02:     This program computes how much an investment grows in
03:     a given number of years.
04: */
05: public class InvestmentRunner
06: {
07:     public static void main(String[] args)
08:     {
09:         final double INITIAL_BALANCE = 10000;
10:         final double RATE = 5;
11:         final int YEARS = 20;
12:         Investment invest = new Investment(INITIAL_BALANCE, RATE);
13:         invest.waitYears(YEARS);
14:         double balance = invest.getBalance();
15:         System.out.printf("The balance after %d years is %.2f\n",
16:             YEARS, balance);
17:     }
18: }
```

Output:

The balance after 20 years is 26532.98

Self Check 6.3

Rewrite the for loop in the `waitYears` method as a `while` loop.

Answer:

```
int i = 1; while (i <= n) { double interest =  
    balance * rate / 100; balance = balance + interest;  
    i++; }
```

Self Check 6.4

Rewrite the `for` loop in the `waitYears` method as a `while`. How many times does the following `for` loop execute?

```
for (i = 0; i <= 10; i++)  
    System.out.println(i * i);
```

Answer: 11 times.

Common Errors: Semicolons

- A missing semicolon

```
for (years = 1;  
    (balance = balance + balance * rate / 100) <  
        targetBalance;  
    years++)  
    System.out.println(years);
```

- A semicolon that shouldn't be there

```
sum = 0;  
for (i = 1; i <= 10; i++);  
    sum = sum + i;  
System.out.println(sum);
```

Nested Loops

- Create triangle pattern

```
[]  
[] []  
[] [] []  
[] [] [] []
```

- Loop through rows

```
for (int i = 1; i <= n; i++)  
{  
    // make triangle row  
}
```

- *Make triangle row* is another loop

```
for (int j = 1; j <= i; j++)  
    r = r + "[]";  
r = r + "\n";
```

- Put loops together → Nested loops

ch06/triangle1/Triangle.java

```
01: /**
02:     This class describes triangle objects that can be displayed
03:     as shapes like this:
04:     []
05:     [][]
06:     [][][]
07: */
08: public class Triangle
09: {
10:     /**
11:         Constructs a triangle.
12:         @param aWidth the number of [] in the last row of the triangle.
13:     */
14:     public Triangle(int aWidth)
15:     {
16:         width = aWidth;
17:     }
18:
19:     /**
20:         Computes a string representing the triangle.
21:         @return a string consisting of [] and newline characters
22:     */
```

ch06/triangle1/Triangle.java (cont.)

```
23:     public String toString()
24:     {
25:         String r = "";
26:         for (int i = 1; i <= width; i++)
27:         {
28:             // Make triangle row
29:             for (int j = 1; j <= i; j++)
30:                 r = r + "[";
31:             r = r + "\n";
32:         }
33:         return r;
34:     }
35:
36:     private int width;
37: }
```

File TriangleRunner.java

```
01: /**
02:     This program prints two triangles.
03: */
04: public class TriangleRunner
05: {
06:     public static void main(String[] args)
07:     {
08:         Triangle small = new Triangle(3);
09:         System.out.println(small.toString());
10:
11:         Triangle large = new Triangle(15);
12:         System.out.println(large.toString());
13:     }
14: }
```

File TriangleRunner.java (cont.)

Output:

```
[]  
[] []  
[] [] []  
  
[]  
[] []  
[] [] []  
[] [] [] []  
[] [] [] [] []  
[] [] [] [] [] []  
[] [] [] [] [] [] []  
[] [] [] [] [] [] [] []  
[] [] [] [] [] [] [] [] [] [] [] [] [] [] []  
[] [] [] [] [] [] [] [] [] [] [] [] [] [] []  
[] [] [] [] [] [] [] [] [] [] [] [] [] [] []  
[] [] [] [] [] [] [] [] [] [] [] [] [] [] []
```

Continued

File TriangleRunner.java (cont.)

Output (continued):

```
[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []  
[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []
```

Self Check 6.5

How would you modify the nested loops so that you print a square instead of a triangle?

Answer: Change the inner loop to `for (int j = 1; j <= width; j++)`

Self Check 6.6

How would you modify the nested loops so that you print a square instead of a What is the value of n after the following nested loops?

```
int n = 0;
for (int i = 1; i <= 5; i++)
    for (int j = 0; j < i; j++)
        n = n + j;
```

Answer: 20.

Processing Sentinel Values

- Sentinel value: Can be used for indicating the end of a data set
- 0 or -1 make poor sentinels; better use Q

```
System.out.print("Enter value, Q to quit: ");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
    We are done
else
{
    double x = Double.parseDouble(input);
    . . .
}
```

Loop and a half

- Sometimes termination condition of a loop can only be evaluated in the middle of the loop
- Then, introduce a boolean variable to control the loop:

```
boolean done = false;
while (!done)
{
    Print prompt
    String input = read input;
    if (end of input indicated)
        done = true;
    else
    {
        Process input
    }
}
```

ch06/dataset/DataAnalyzer.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program computes the average and maximum of a set
05:     of input values.
06: */
07: public class DataAnalyzer
08: {
09:     public static void main(String[] args)
10:     {
11:         Scanner in = new Scanner(System.in);
12:         DataSet data = new DataSet();
13:
14:         boolean done = false;
15:         while (!done)
16:         {
17:             System.out.print("Enter value, Q to quit: ");
18:             String input = in.next();
19:             if (input.equalsIgnoreCase("Q"))
20:                 done = true;
```

Continued

ch06/dataset/DataAnalyzer.java (cont.)

```
21:         else
22:         {
23:             double x = Double.parseDouble(input);
24:             data.add(x);
25:         }
26:     }
27:
28:     System.out.println("Average = " + data.getAverage());
29:     System.out.println("Maximum = " + data.getMaximum());
30: }
31: }
```

ch06/dataset/DataSet.java

```
01: /**
02:     Computes the average of a set of data values.
03: */
04: public class DataSet
05: {
06:     /**
07:         Constructs an empty data set.
08:     */
09:     public DataSet()
10:     {
11:         sum = 0;
12:         count = 0;
13:         maximum = 0;
14:     }
15:
16:     /**
17:         Adds a data value to the data set
18:         @param x a data value
19:     */
20:     public void add(double x)
21:     {
```

Continued

ch06/dataset/DataSet.java (cont.)

```
22:         sum = sum + x;
23:         if (count == 0 || maximum < x) maximum = x;
24:         count++;
25:     }
26:
27:     /**
28:      Gets the average of the added data.
29:      @return the average or 0 if no data has been added
30:     */
31:     public double getAverage()
32:     {
33:         if (count == 0) return 0;
34:         else return sum / count;
35:     }
36:
37:     /**
38:      Gets the largest of the added data.
39:      @return the maximum or 0 if no data has been added
40:     */
```

Continued

ch06/dataset/DataSet.java (cont.)

```
41:     public double getMaximum()  
42:     {  
43:         return maximum;  
44:     }  
45:  
46:     private double sum;  
47:     private double maximum;  
48:     private int count;  
49: }
```

Output:

```
Enter value, Q to quit: 10  
Enter value, Q to quit: 0  
Enter value, Q to quit: -1  
Enter value, Q to quit: Q  
Average = 3.0  
Maximum = 10.0
```

Self Check 6.7

Why does the `DataAnalyzer` class call `in.next` and not `in.nextDouble`?

Answer: Because we don't know whether the next input is a number or the letter `Q`.

Self Check 6.8

Would the `DataSet` class still compute the correct maximum if you simplified the update of the maximum field in the `add` method to the following statement?

```
if (maximum < x) maximum = x;
```

Answer: No. If *all* input values are negative, the maximum is also negative. However, the `maximum` field is initialized with 0. With this simplification, the maximum would be falsely computed as 0.

Random Numbers and Simulations

- In a simulation, you repeatedly generate random numbers and use them to simulate an activity
- Random number generator

```
Random generator = new Random(); int n =  
generator.nextInt(a); // 0 <= n < a double x =  
generator.nextDouble(); // 0 <= x < 1
```

- Throw die (random number between 1 and 6)

```
int d = 1 + generator.nextInt(6);
```

ch06/random1/Die.java

```
01: import java.util.Random;
02:
03: /**
04:     This class models a die that, when cast, lands on a random
05:     face.
06: */
07: public class Die
08: {
09:     /**
10:         Constructs a die with a given number of sides.
11:         @param s the number of sides, e.g. 6 for a normal die
12:     */
13:     public Die(int s)
14:     {
15:         sides = s;
16:         generator = new Random();
17:     }
18:
19:     /**
20:         Simulates a throw of the die
21:         @return the face of the die
22:     */
```

Continued

ch06/random1/Die.java (cont.)

```
23:     public int cast()
24:     {
25:         return 1 + generator.nextInt(sides);
26:     }
27:
28:     private Random generator;
29:     private int sides;
30: }
```

ch06/random1/DieSimulator.java

```
01: /**
02:     This program simulates casting a die ten times.
03: */
04: public class DieSimulator
05: {
06:     public static void main(String[] args)
07:     {
08:         Die d = new Die(6);
09:         final int TRIES = 10;
10:         for (int i = 1; i <= TRIES; i++)
11:         {
12:             int n = d.cast();
13:             System.out.print(n + " ");
14:         }
15:         System.out.println();
16:     }
17: }
```

ch06/random1/DieSimulator.java (cont.)

Output:

6 5 6 3 2 6 3 4 4 1

Second Run:

3 2 2 1 6 5 3 4 1 2

Buffon Needle Experiment

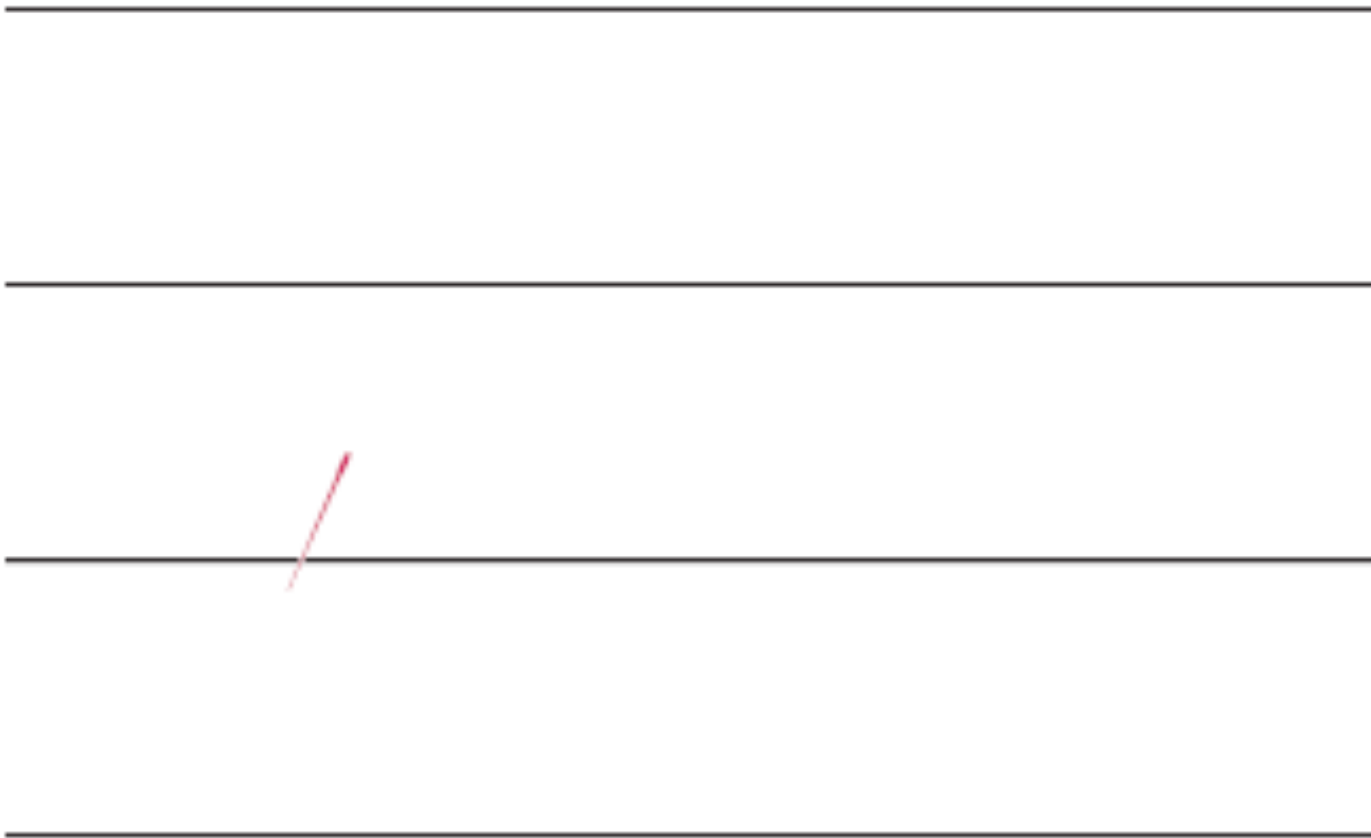


Figure 3 The Buffon Needle Experiment

Needle Position

Buffon Needle Experiment

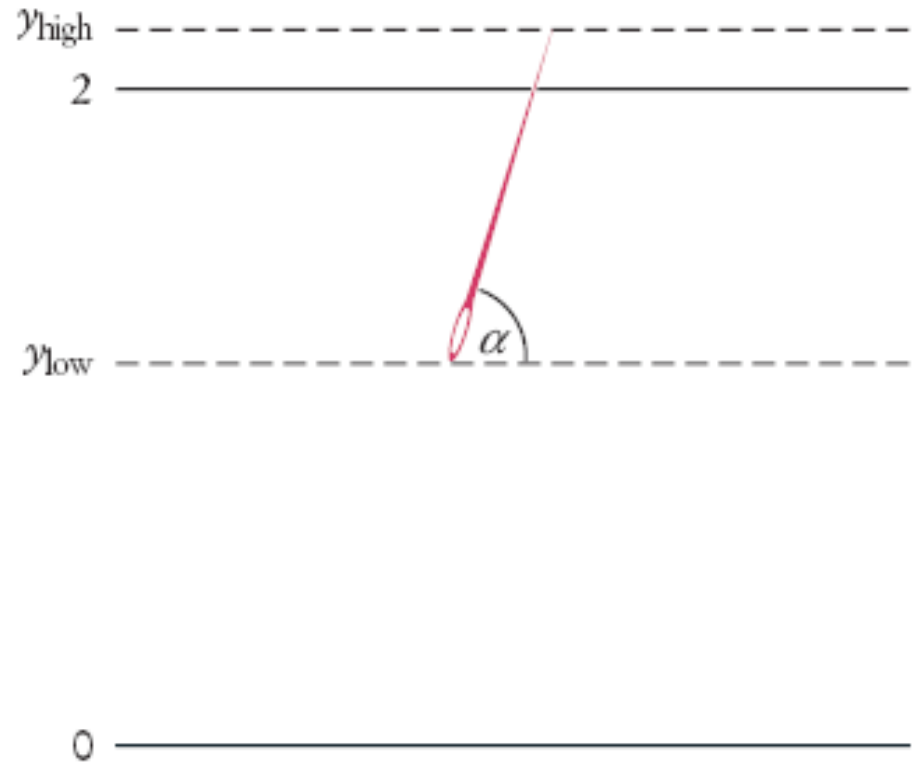


Figure 4

When Does the Needle Fall on a Line?

Needle Position

- Needle length = 1, distance between lines = 2
- Generate random y_{low} between 0 and 2
- Generate random angle α between 0 and 180 degrees
- $y_{high} = y_{low} + \sin(\alpha)$
- Hit if $y_{high} \geq 2$

ch06/random2/Needle.java

```
01: import java.util.Random;
02:
03: /**
04:     This class simulates a needle in the Buffon needle experiment.
05: */
06: public class Needle
07: {
08:     /**
09:         Constructs a needle.
10:     */
11:     public Needle()
12:     {
13:         hits = 0;
14:         tries = 0;
15:         generator = new Random();
16:     }
17:
18:     /**
19:         Drops the needle on the grid of lines and
20:         remembers whether the needle hit a line.
21:     */
```

Continued

ch06/random2/Needle.java (cont.)

```
22:     public void drop()
23:     {
24:         double ylow = 2 * generator.nextDouble();
25:         double angle = 180 * generator.nextDouble();
26:
27:         // Computes high point of needle
28:
29:         double yhigh = ylow + Math.sin(Math.toRadians(angle));
30:         if (yhigh >= 2) hits++;
31:         tries++;
32:     }
33:
34:     /**
35:      * Gets the number of times the needle hit a line.
36:      * @return the hit count
37:      */
38:     public int getHits()
39:     {
40:         return hits;
41:     }
42:
```

Continued

ch06/random2/Needle.java (cont.)

```
43:    /**
44:        Gets the total number of times the needle was dropped.
45:        @return the try count
46:    */
47:    public int getTries()
48:    {
49:        return tries;
50:    }
51:
52:    private Random generator;
53:    private int hits;
54:    private int tries;
55: }
```

Output:

```
Tries = 10000, Tries / Hits = 3.08928
Tries = 1000000, Tries / Hits = 3.14204
```

ch06/random2/NeedleSimulator.java

```
01: /**
02:     This program simulates the Buffon needle experiment
03:     and prints the resulting approximations of pi.
04: */
05: public class NeedleSimulator
06: {
07:     public static void main(String[] args)
08:     {
09:         Needle n = new Needle();
10:         final int TRIES1 = 10000;
11:         final int TRIES2 = 1000000;
12:
13:         for (int i = 1; i <= TRIES1; i++)
14:             n.drop();
15:         System.out.printf("Tries = %d, Tries / Hits = %8.5f\n",
16:             TRIES1, (double) n.getTries() / n.getHits());
17:
18:         for (int i = TRIES1 + 1; i <= TRIES2; i++)
19:             n.drop();
20:         System.out.printf("Tries = %d, Tries / Hits = %8.5f\n",
21:             TRIES2, (double) n.getTries() / n.getHits());
22:     }
23: }
```

ch06/random2/NeedleSimulator.java

Output:

```
Tries = 10000, Tries / Hits = 3.08928 Tries = 1000000,  
Tries / Hits = 3.14204
```

Self Check 6.9

How do you use a random number generator to simulate the toss of a coin?

Answer: `int n = generator.nextInt(2); // 0 = heads,
1 = tails`

Self Check 6.10

Why is the `NeedleSimulator` program not an efficient method for computing π ?

Answer: The program repeatedly calls `Math.toRadians(angle)`. You could simply call `Math.toRadians(180)` to compute π .

Using a Debugger

- Debugger = program to run your program and analyze its run-time behavior
- A debugger lets you stop and restart your program, see contents of variables, and step through it
- The larger your programs, the harder to debug them simply by inserting print commands
- Debuggers can be part of your IDE (e.g. Eclipse, BlueJ) or separate programs (e.g. JSwat)
- Three key concepts:
 - *Breakpoints*
 - *Single-stepping*
 - *Inspecting variables*

The Debugger Stopping at a Breakpoint

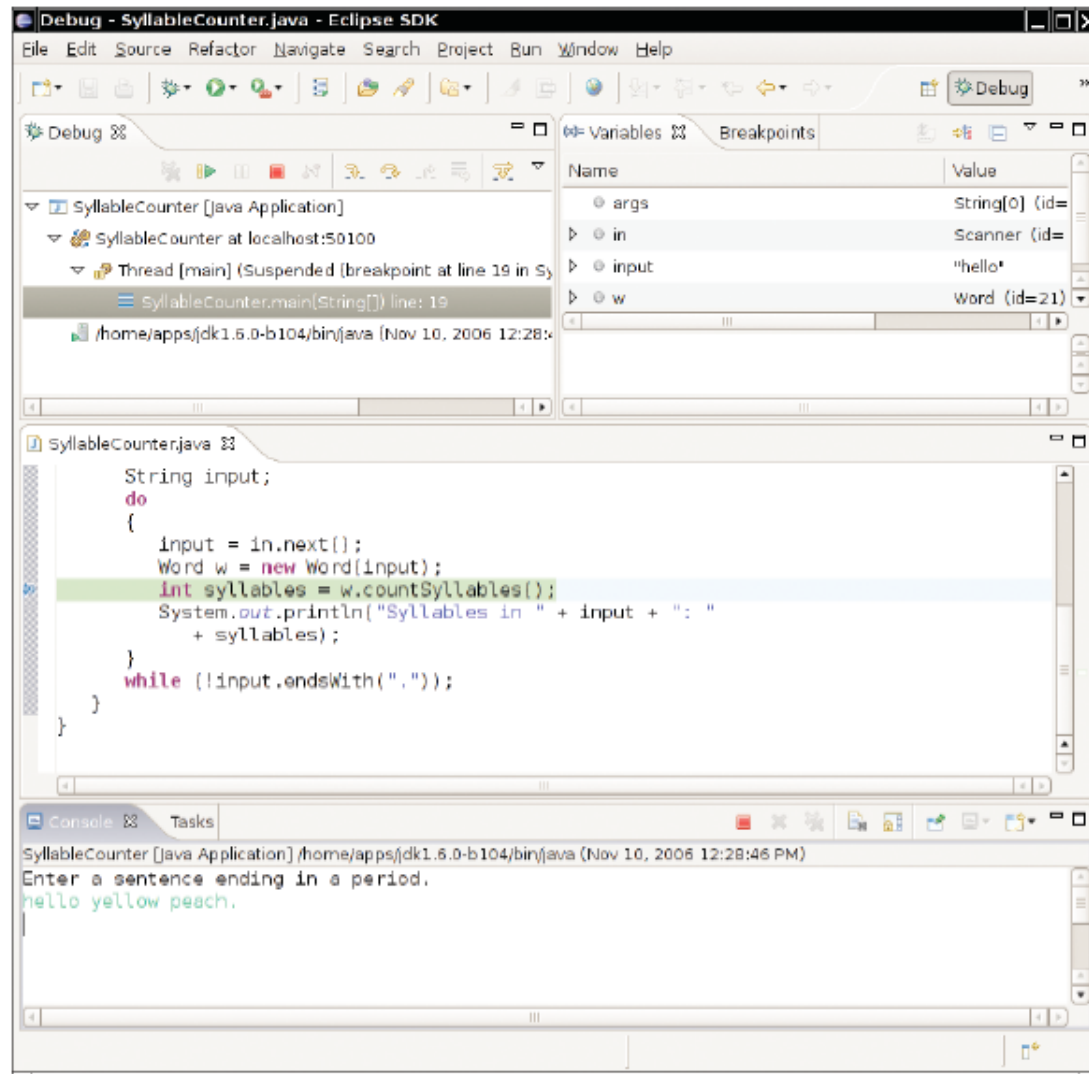


Figure 5 Stopping at a Breakpoint

Inspecting Variables

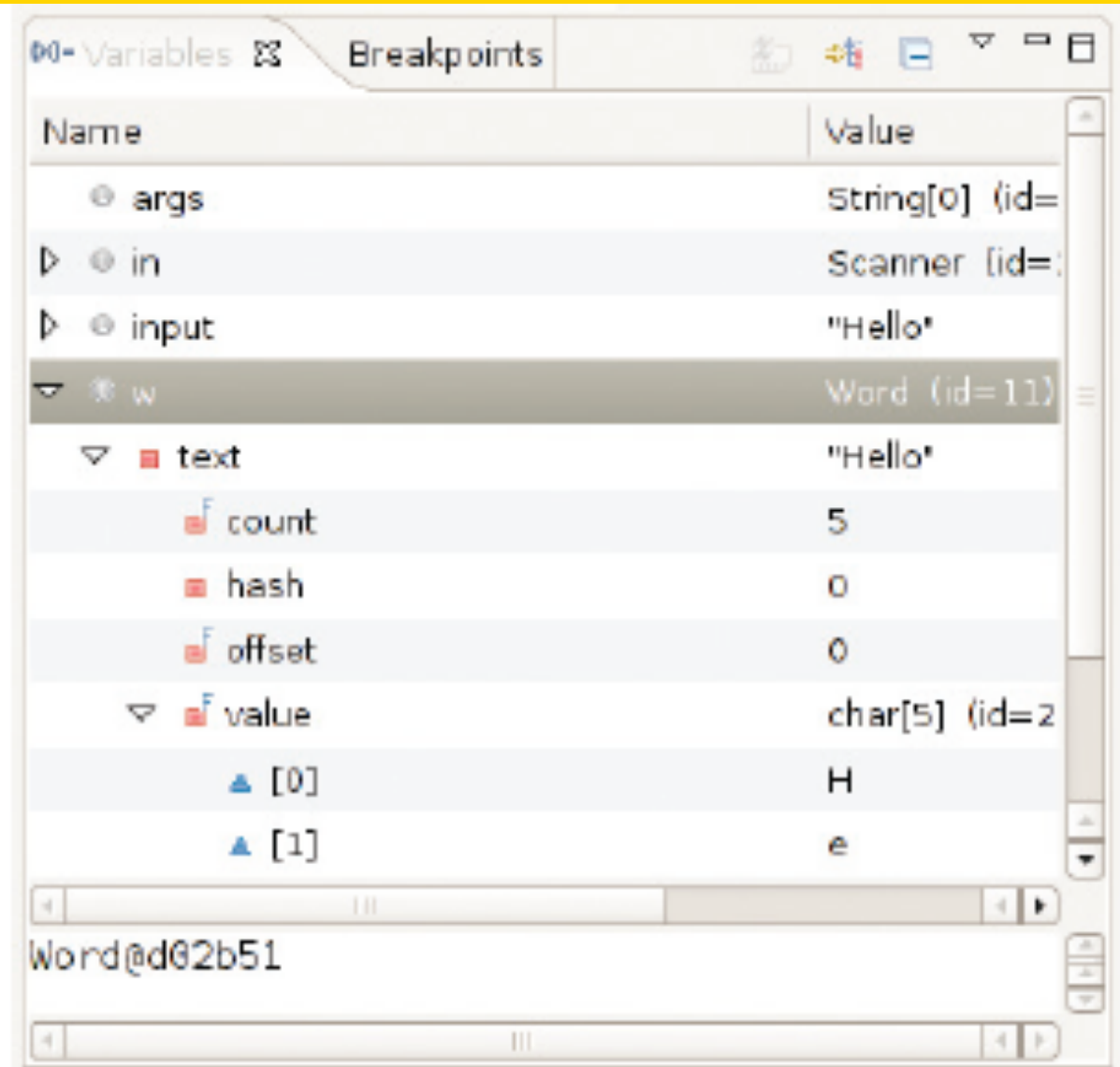


Figure 6

Inspecting Variables

Debugging

- Execution is suspended whenever a breakpoint is reached
- In a debugger, a program runs at full speed until it reaches a breakpoint
- When execution stops you can:
 - *Inspect variables*
 - *Step through the program a line at a time*
 - *Or, continue running the program at full speed until it reaches the next breakpoint*
- When program terminates, debugger stops as well
- Breakpoints stay active until you remove them
- Two variations of single-step command:
 - *Step Over: skips method calls*
 - *Step Into: steps inside method calls*

Single-step Example

Current line:

```
String input = in.next();  
Word w = new Word(input);  
int syllables = w.countSyllables();  
System.out.println("Syllables in " + input + ": " +  
    syllables);
```

When you step over method calls, you get to the next line:

```
String input = in.next();  
Word w = new Word(input);  
int syllables = w.countSyllables();  
System.out.println("Syllables in " + input + ": " +  
    syllables);
```

Continued

Single-step Example (cont.)

However, if you step into method calls, you enter the first line of the `countSyllables` method

```
public int countSyllables()  
{  
    int count = 0;  
    int end = text.length() - 1;  
    . . .  
}
```

Self Check 6.11

In the debugger, you are reaching a call to `System.out.println`. Should you step into the method or step over it?

Answer: You should step over it because you are not interested in debugging the internals of the `println` method.

Self Check 6.12

In the debugger, you are reaching the beginning of a long method with a couple of loops inside. You want to find out the return value that is computed at the end of the method. Should you set a breakpoint, or should you step through the method?

Answer: You should set a breakpoint. Stepping through loops can be tedious.

Sample Debugging Session

- `Word` class counts syllables in a word
- Each group of adjacent vowels (a, e, i, o, u, y) counts as one syllable
- However, an e at the end of a word doesn't count as a syllable
- If algorithm gives count of 0, increment to 1
- Constructor removes non-letters at beginning and end

ch06/debugger/Word.java

```
01: /**
02:     This class describes words in a document.
03: */
04: public class Word
05: {
06:     /**
07:         Constructs a word by removing leading and trailing non-
08:         letter characters, such as punctuation marks.
09:         @param s the input string
10:     */
11:     public Word(String s)
12:     {
13:         int i = 0;
14:         while (i < s.length() && !Character.isLetter(s.charAt(i)))
15:             i++;
16:         int j = s.length() - 1;
17:         while (j > i && !Character.isLetter(s.charAt(j)))
18:             j--;
19:         text = s.substring(i, j);
20:     }
21:
```

Continued

ch06/debugger/Word.java (cont.)

```
22:    /**
23:        Returns the text of the word, after removal of the
24:        leading and trailing non-letter characters.
25:        @return the text of the word
26:    */
27:    public String getText()
28:    {
29:        return text;
30:    }
31:
32:    /**
33:        Counts the syllables in the word.
34:        @return the syllable count
35:    */
36:    public int countSyllables()
37:    {
38:        int count = 0;
39:        int end = text.length() - 1;
40:        if (end < 0) return 0; // The empty string has no syllables
41:
```

Continued

ch06/debugger/Word.java (cont.)

```
42:         // An e at the end of the word doesn't count as a vowel
43:         char ch = Character.toLowerCase(text.charAt(end));
44:         if (ch == 'e') end--;
46:         boolean insideVowelGroup = false;
47:         for (int i = 0; i <= end; i++)
48:         {
49:             ch = Character.toLowerCase(text.charAt(i));
50:             String vowels = "aeiouy";
51:             if (vowels.indexOf(ch) >= 0)
52:             {
53:                 // ch is a vowel
54:                 if (!insideVowelGroup)
55:                 {
56:                     // Start of new vowel group
57:                     count++;
58:                     insideVowelGroup = true;
59:                 }
60:             }
61:         }
62:
```

Continued

ch06/debugger/Word.java (cont.)

```
63:         // Every word has at least one syllable
64:         if (count == 0)
65:             count = 1;
66:
67:         return count;
68:     }
69:
70:     private String text;
71: }
```

ch06/debugger/SyllableCounter.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program counts the syllables of all words in a sentence.
05: */
06: public class SyllableCounter
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner in = new Scanner(System.in);
11:
12:         System.out.println("Enter a sentence ending in a period.");
13:
14:         String input;
15:         do
16:         {
17:             input = in.next();
18:             Word w = new Word(input);
19:             int syllables = w.countSyllables();
20:             System.out.println("Syllables in " + input + ": "
21:                 + syllables);
22:         }
```

Continued

ch06/debugger/SyllableCounter.java (cont.)

```
23:         while (!input.endsWith("."));  
24:     }  
25: }
```

Debug the Program

- Buggy output (for input "hello yellow peach."):
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
- Set breakpoint in first line of `countSyllables` of `Word` class
- Start program, supply input. Program stops at breakpoint
- Method checks if final letter is 'e'

Continued

Debug the Program (cont.)

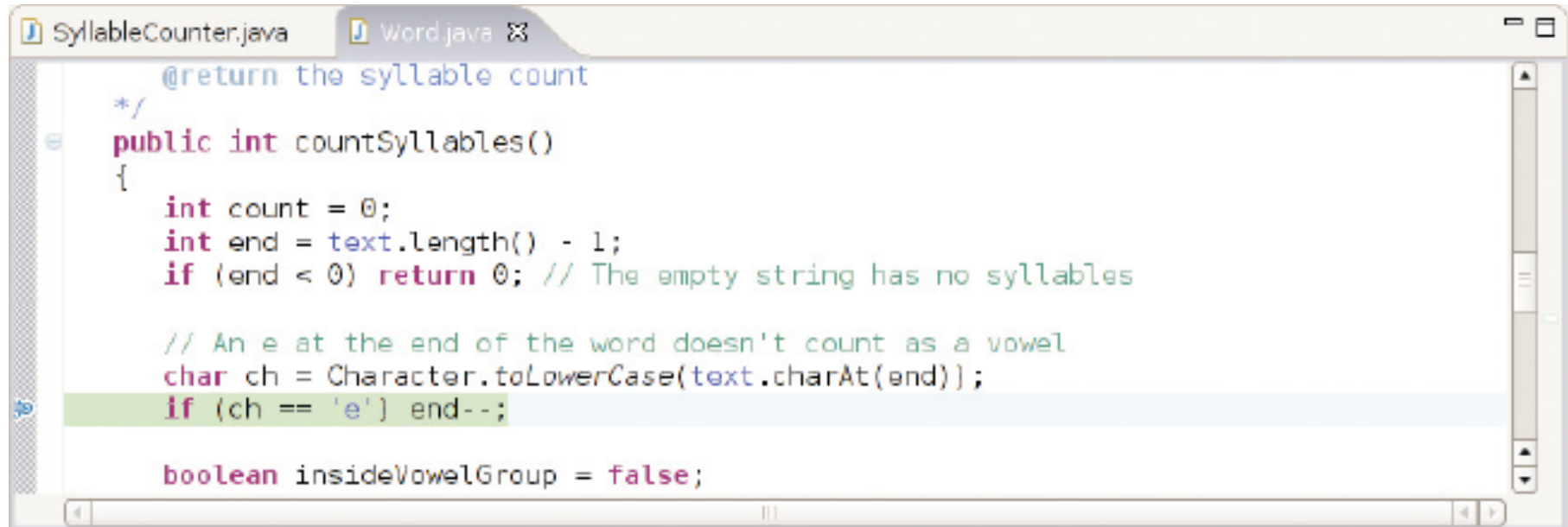
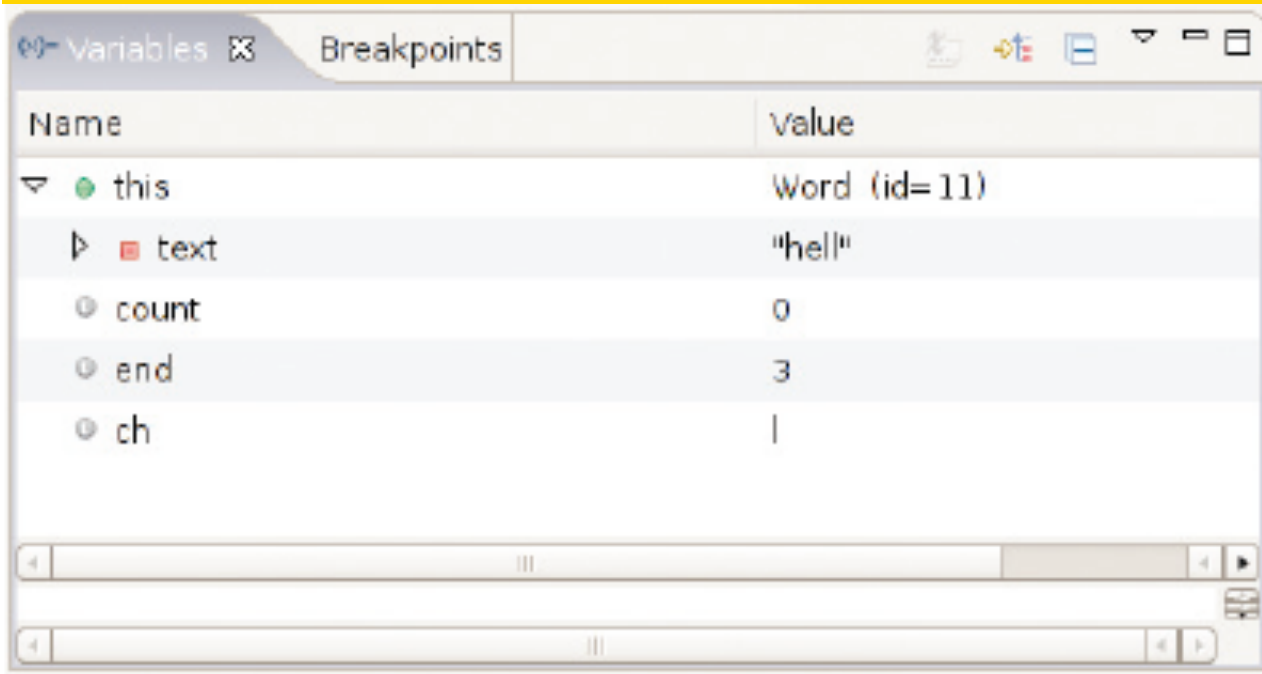


Figure 7 Debugging the `countSyllables` Method

- Check if this works: step to line where check is made and inspect variable `ch`
- Should contain final letter but contains 'l'

More Problems Found



The screenshot shows a Java IDE's 'Variables' window. It has two tabs: 'Variables' and 'Breakpoints'. The 'Variables' tab is active, displaying a table of variables. The table has two columns: 'Name' and 'Value'. The variables listed are 'this' (a Word object with id=11), 'text' (a String with value 'hell'), 'count' (an int with value 0), 'end' (an int with value 3), and 'ch' (a char with value 'l'). The 'text' variable is highlighted with a mouse cursor. Below the table are two scroll bars.

Name	Value
this	Word (id=11)
text	"hell"
count	0
end	3
ch	l

Figure 8 The Current Values of the Local and Instance Variables

- `end` is set to 3, not 4
- `text` contains "hell", not "hello"
- No wonder `countSyllables` returns 1

Continued

More Problems Found (cont.)

- Culprit is elsewhere
- Can't go back in time
- Restart and set breakpoint in `Word` constructor

Debugging the Word Constructor

- Supply "hello" input again
- Break past the end of second loop in constructor
- Inspect `i` and `j`
- They are 0 and 4 – makes sense since the input consists of letters
- Why is `text` set to "hell"?
- Off-by-one error: Second parameter of `substring` is the first position *not* to include

```
text = substring(i, j);
```

should be

```
text = substring(i, j + 1);
```

Debugging the Word Constructor

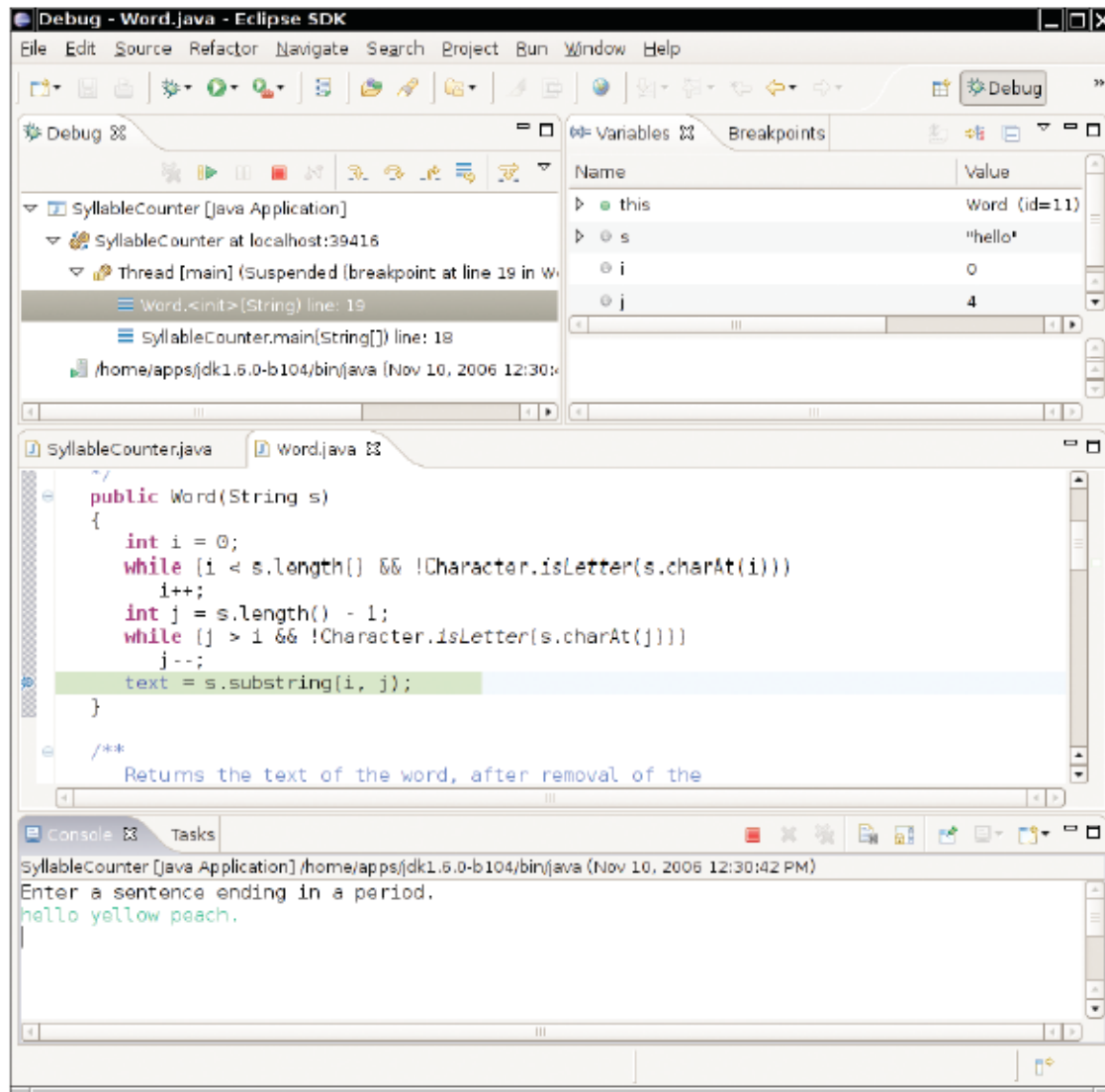


Figure 9 Debugging the Word Constructor

Another Error

- Fix the error
- Recompile
- Test again:

```
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
```
- Oh no, it's still not right
- Start debugger
- Erase all old breakpoints and set a breakpoint in `countSyllables` method
- Supply input `"hello."`

Debugging `countSyllables` (again)

Break in the beginning of `countSyllables`. Then, single-step through loop

```
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
    ch = Character.toLowerCase(text.charAt(i));
    if ("aeiouy".indexOf(ch) >= 0)
    {
        // ch is a vowel
        if (!insideVowelGroup)
        {
            // Start of new vowel group
            count++;
            insideVowelGroup = true;
        }
    }
}
```

Continued

Debugging `countSyllables` (again)

- First iteration (`'h'`) : skips test for vowel
- Second iteration (`'e'`): passes test, increments `count`
- Third iteration (`'l'`): skips test
- Fifth iteration (`'o'`): passes test, but second `if` is skipped, and `count` is not incremented

Fixing the Bug

- `insideVowelGroup` **was never reset to** `false`

- **Fix**

```
if ("aeiouy".indexOf(ch) >= 0)
{
    . . .
}
else insideVowelGroup = false;
```

- **Retest: All test cases pass**

```
Syllables in hello: 2
Syllables in yellow: 2
Syllables in peach.: 1
```

- **Is the program now bug-free? The debugger can't answer that.**

Self Check 6.13

What caused the first error that was found in this debugging session?

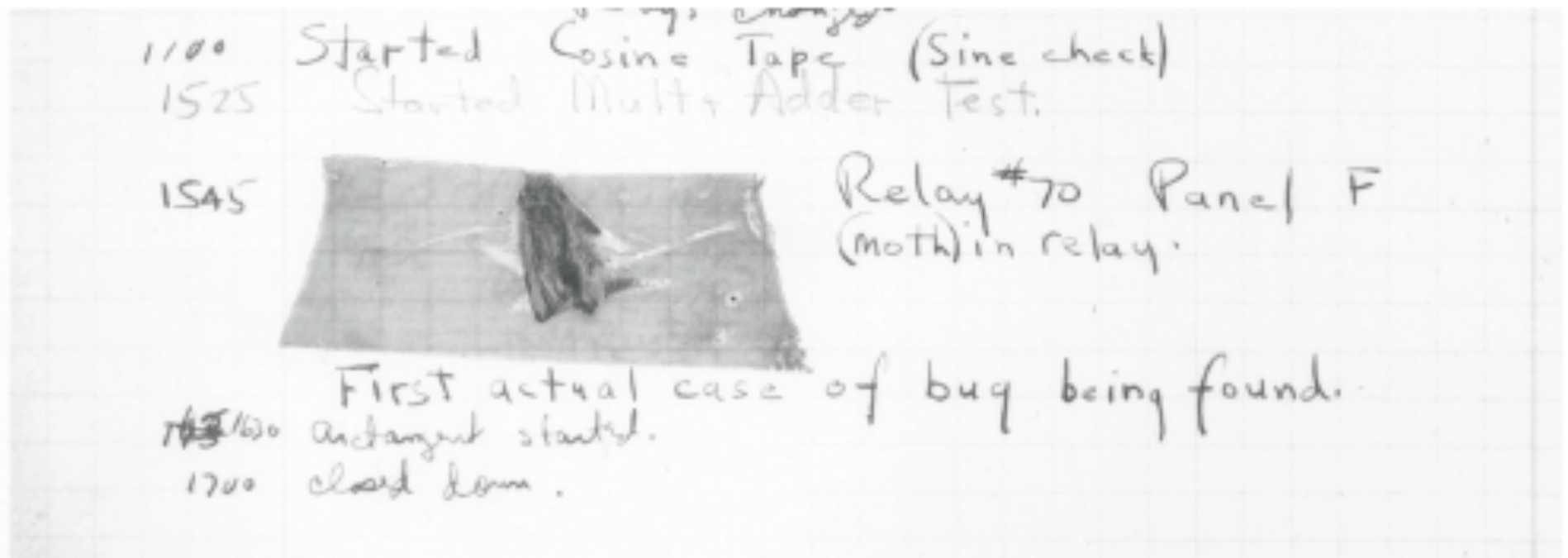
Answer: The programmer misunderstood the second parameter of the substring method—it is the index of the first character not to be included in the substring.

Self Check 6.14

What caused the second error? How was it detected?

Answer: The second error was caused by failing to reset `insideVowelGroup` to false at the end of a vowel group. It was detected by tracing through the loop and noticing that the loop didn't enter the conditional statement that increments the vowel count.

The First Bug



The First Bug