Iteration

Advanced Programming

ICOM 4015

Lecture 6

Reading: Java Concepts Chapter 7

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

`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)
{
   year++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

File 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
09:       and 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:
```

Continued…

File Investment.java

```
20:    /**
21:       Keeps accumulating interest until a target balance has
22:       been reached.
23:       @param targetBalance the desired balance
24:    */
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:    */
```

Continued…

File Investment.java

```
39:    public double getBalance()
40:    {
41:       return balance;
42:    }
43:
44:    /**
45:       Gets the number of years this investment has
46:       accumulated interest.
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: }
```

File InvestmentTester.java

```
01: /**
02:    This program computes how long it takes for an investment
03:    to double.
04: */
05: public class InvestmentTester
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: }
```

Continued…

File InvestmentTester.java

Output

```
The investment doubled after 15 years
```

`while` Loop Flowchart

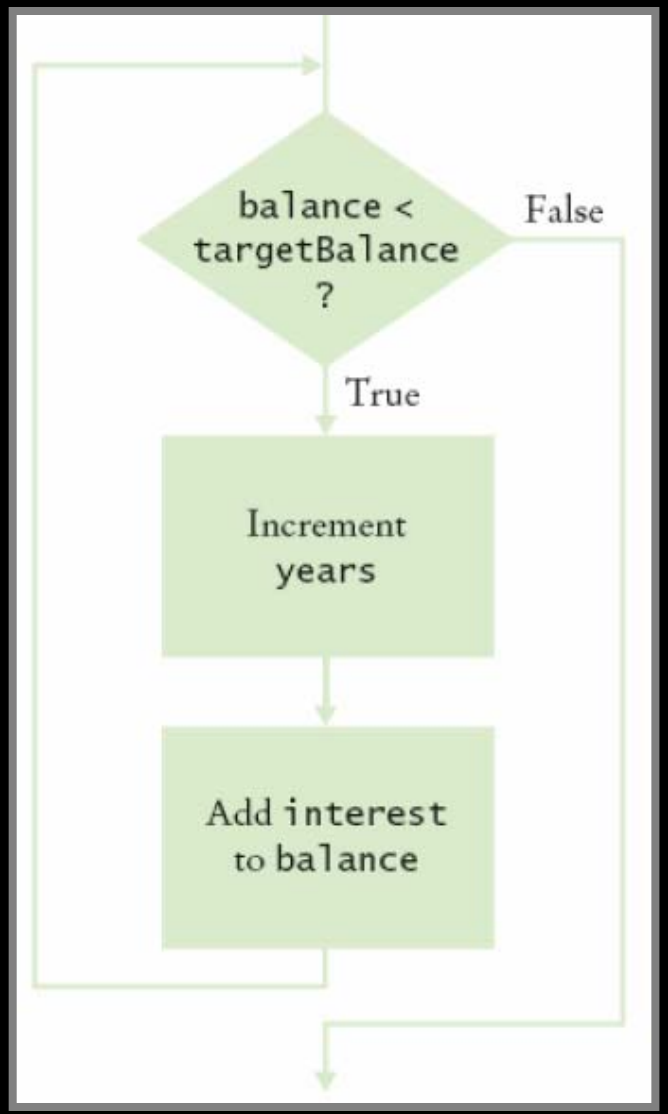


Figure 1:
Flowchart of a `while` Loop

Syntax 7.1: The `while` Statement

```
while (condition)
   statement
```

Example:

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

Purpose:

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

Self Check

1. How often is the statement in the loop

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

   executed?

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

Answers

1. Never
2. 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.

Continued…

Avoiding Off-by-One Error

- interest rate: 100%
  after one year: `balance` is 2 * `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

- **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

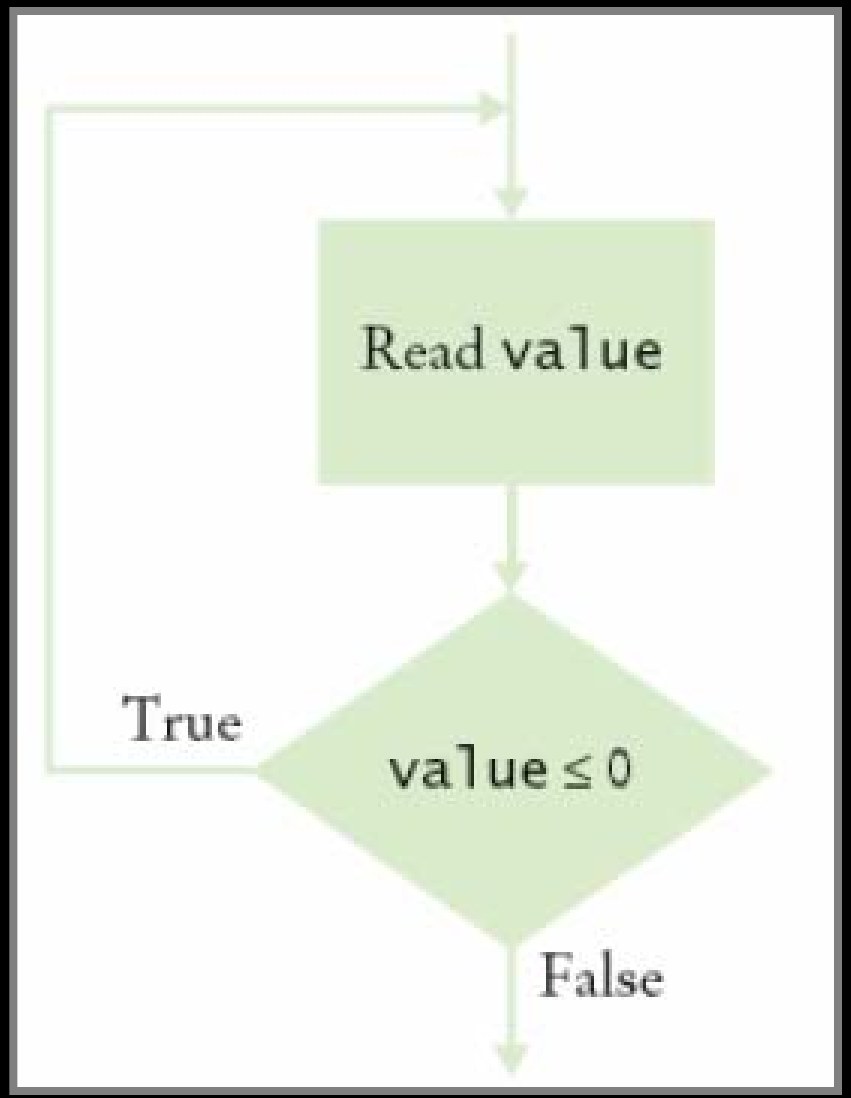


Figure 2:
Flowchart of a do Loop

Spaghetti Code

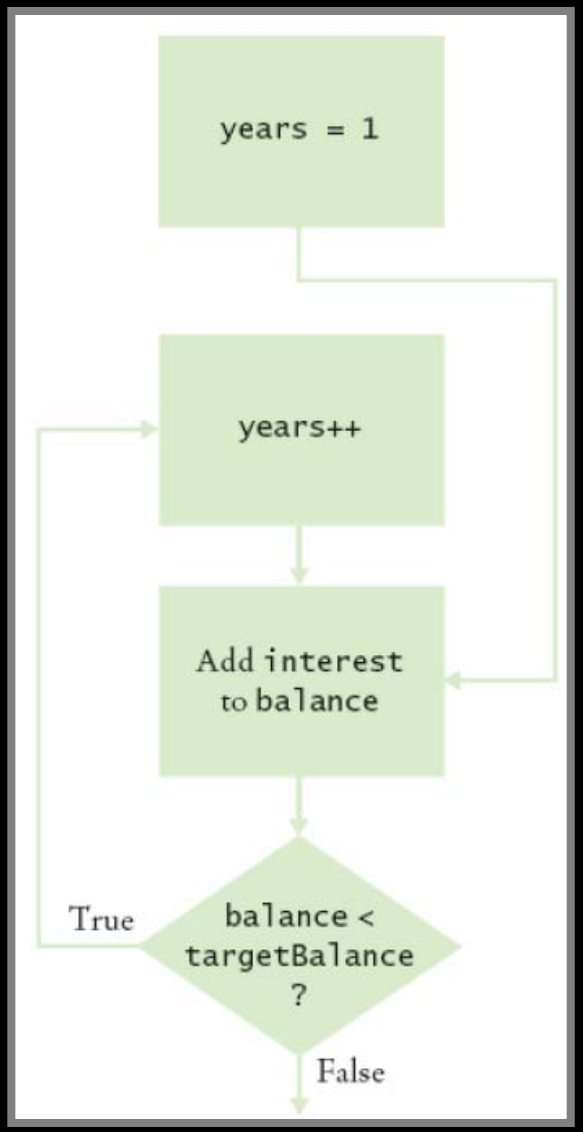


Figure 3:
Spaghetti Code

for Loops

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

Example:

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

Continued…
```

`for` Loops

- **Equivalent to**

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

- **Other examples:**

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

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

Flowchart for `for` Loop

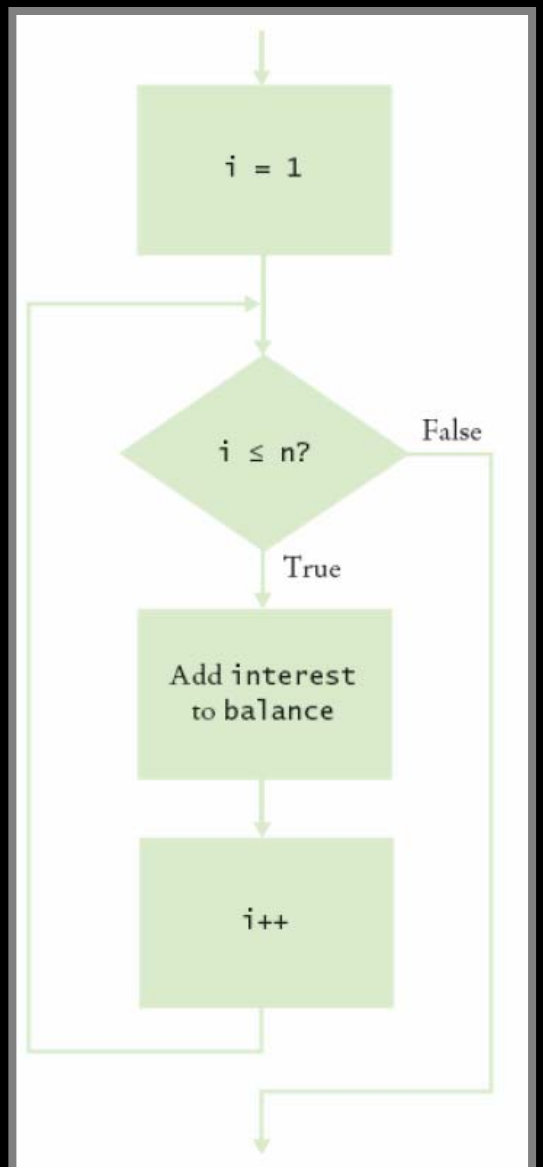


Figure 4:
Flowchart of a `for` Loop

Syntax 7.2: 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

File 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
09:       balance and 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:    }
```

Continued…

File Investment.java

```
19:
20:    /**
21:       Keeps accumulating interest until a target balance
22:       has been reached.
23:       @param targetBalance the desired balance
24:    */
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:
```

Continued…

File Investment.java

```
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;
45:       }
46:       years = years + n;
47:    }
48:
49:    /**
50:       Gets the current investment balance.
51:       @return the current balance
52:    */
```

Continued…

File Investment.java

```
53:    public double getBalance()
54:    {
55:       return balance;
56:    }
57:
58:    /**
59:       Gets the number of years this investment has
60:       accumulated interest.
61:       @return the number of years since the start of the
             investment
62:    */
63:    public int getYears()
64:    {
65:       return years;
66:    }
67:
```

Continued…

File Investment.java

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

File InvestmentTester.java

```
01: /**
02:    This program computes how much an investment grows in
03:    a given number of years.
04: */
05: public class InvestmentTester
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: }
```

Continued…

File Investment.java

Output

```
The balance after 20 years is 26532.98
```

Self Check

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

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

Answers

1.

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

1. 11 times

Common Errors: Semicolons

- **A semicolon that shouldn't be there**

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

- **A missing semicolon**

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

Nested Loops

- **Create triangle pattern**

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

- **Loop through rows**

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

Nested Loops

- ***Make triangle row*** **is another loop**

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

- **Put loops together → Nested loops**

File Triangle.java

```
01: /**
02:    This class describes triangle objects that can be
03:    displayed 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:
```

Continued…

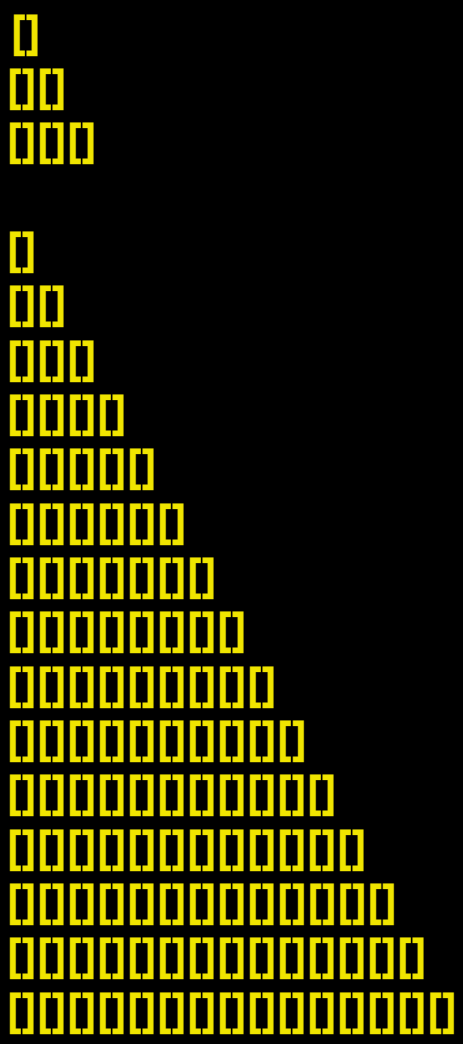
File Triangle.java

```
19:    /**
20:       Computes a string representing the triangle.
21:       @return a string consisting of [] and newline
             characters
22:    */
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 TriangleTester.java

```
01: /**
02:    This program tests the Triangle class.
03: */
04: public class TriangleTester
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: }
```

Output



Self Check

1. How would you modify the nested loops so that you print a square instead of a triangle?
2. 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;
```

Answers

1. Change the inner loop to

```
for (int j = 1; j <= width; j++)
```

2. 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
   }
}
```

File InputTester.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 InputTester
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:       {
```

Continued...

File InputTester.java

```
17:             System.out.print("Enter value, Q to quit: ");
18:             String input = in.next();
19:             if (input.equalsIgnoreCase("Q"))
20:                done = true;
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: }
```

File 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:    */
```

Continued...

File DataSet.java

```
20:    public void add(double x)
21:    {
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:
```

Continued...

File DataSet.java

```
37:    /**
38:       Gets the largest of the added data.
39:       @return the maximum or 0 if no data has been added
40:    */
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

1. Why does the `InputTester` class call `in.next` and not `in.nextDouble`?
2. 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;
```

Answers

1. Because we don't know whether the next input is a number or the letter Q.
2. 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);
```

File Die.java

```
01: import java.util.Random;
02:
03: /**
04:    This class models a die that, when cast, lands on a
05:    random 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:
```

Continued…

File Die.java

```
19:    /**
20:       Simulates a throw of the die
21:       @return the face of the die
22:    */
23:    public int cast()
24:    {
25:       return 1 + generator.nextInt(sides);
26:    }
27:
28:    private Random generator;
29:    private int sides;
30: }
```

File DieTester.java

```
01: /**
02:    This program simulates casting a die ten times.
03: */
04: public class DieTester
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: }
```

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

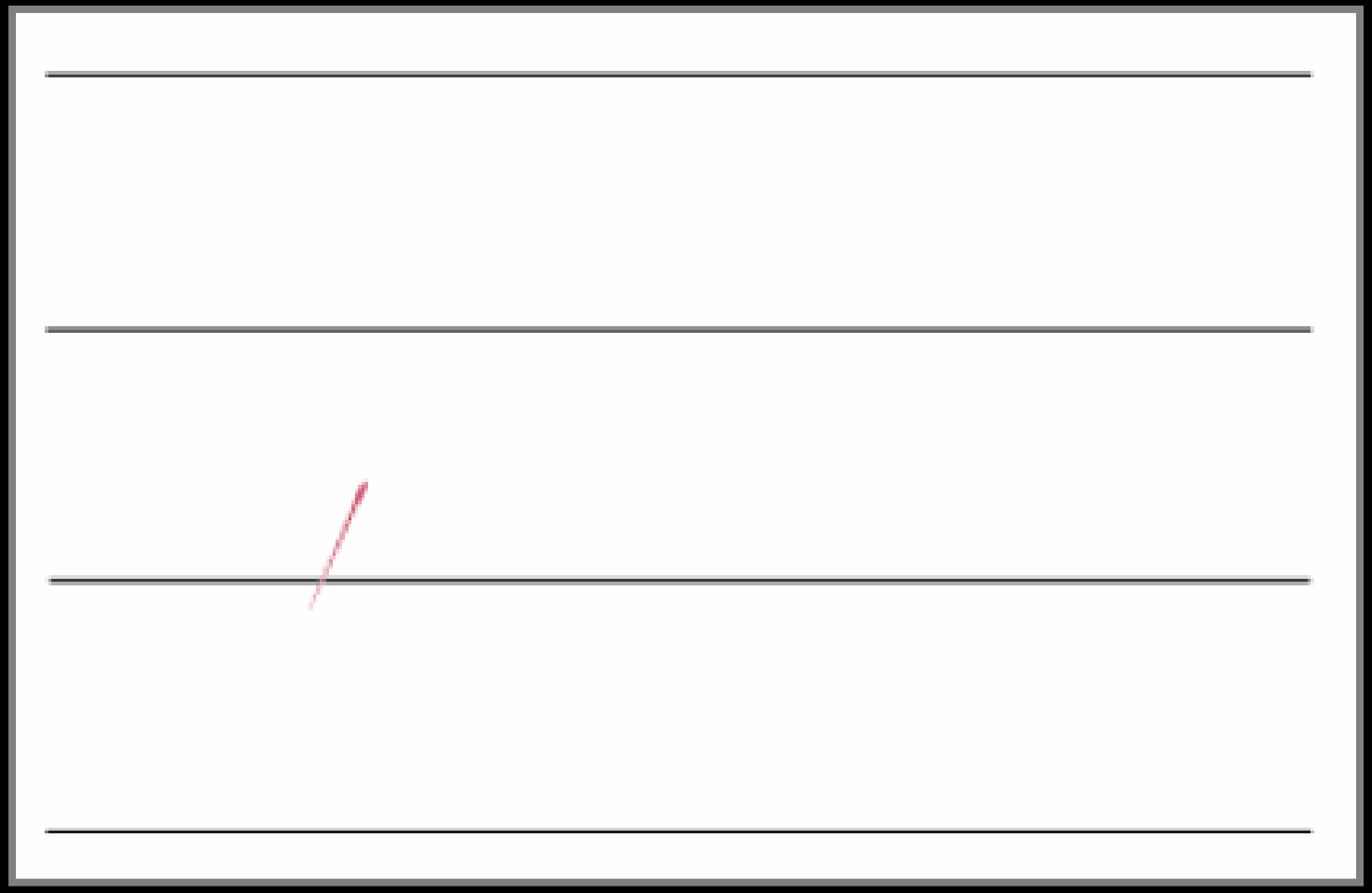


Figure 5:
The Buffon Needle Experiment

Needle Position

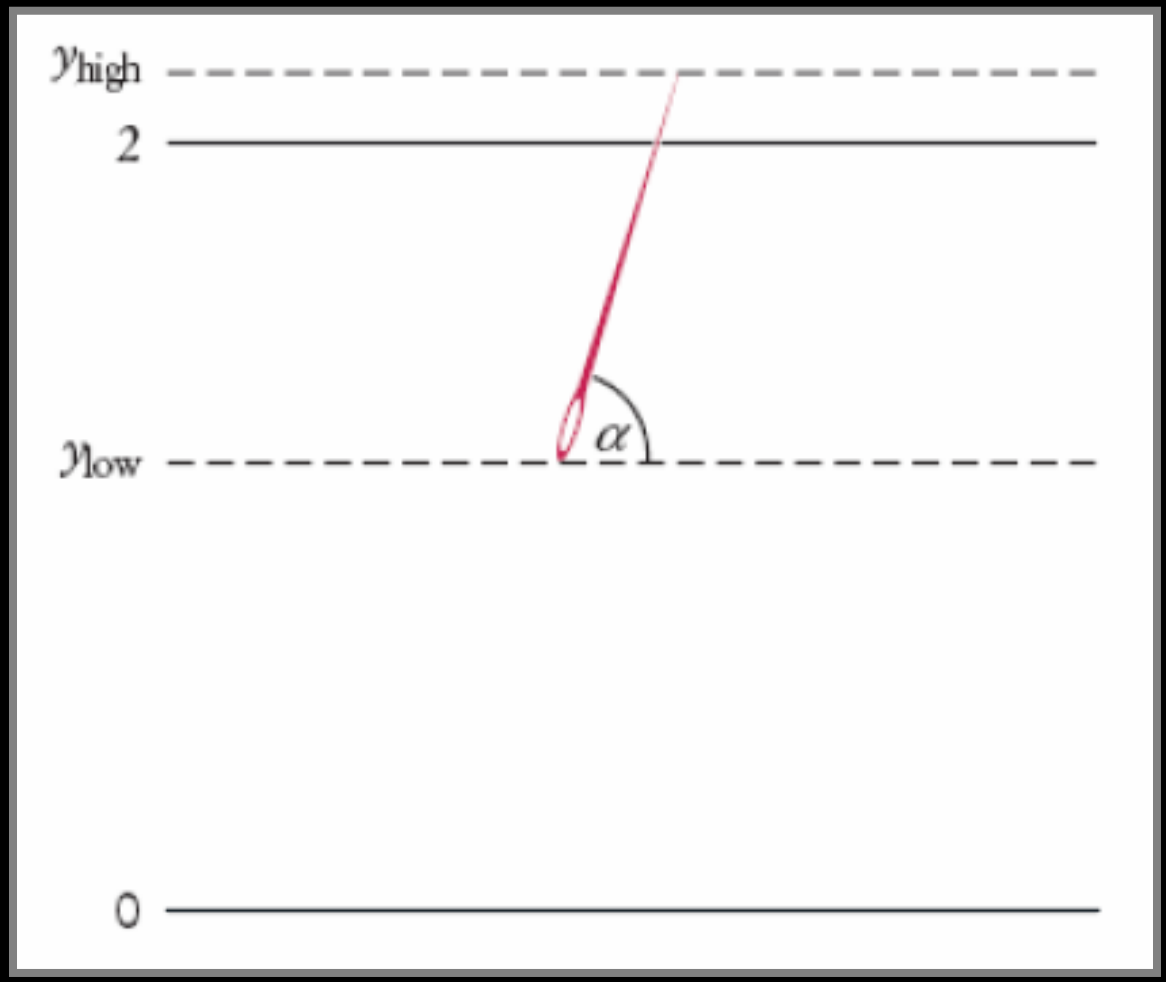


Figure 6:
When Does a Needle Fall on a Line?

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
- $y_{high} = yl_{ow} + \sin(\alpha)$
- Hit if $y_{high} \geq 2$

File 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:
```

Continued…

File Needle.java

```
18:    /**
19:       Drops the needle on the grid of lines and
20:       remembers whether the needle hit a line.
21:    */
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:    */
```

Continued...

File Needle.java

```
38:    public int getHits()
39:    {
40:       return hits;
41:    }
42:
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: }
```

File NeedleTester.java

```
01: /**
02:    This program simulates the Buffon needle experiment
03:    and prints the resulting approximations of pi.
04: */
05: public class NeedleTester
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:
```

Continued…

File NeedleTester.java

```
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: }
```

Output

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

Self Check

1. How do you use a random number generator to simulate the toss of a coin?
2. Why is the `NeedleTester` program not an efficient method for computing π?

Answers

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

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